# Preference Queries Processing over Imprecise Data

**A DISSERTATION**
**SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL**
**OF THE UNIVERSITY OF MINNESOTA**
**BY**

**Mohamed E. Khalefa**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS**
**FOR THE DEGREE OF**
**Doctor of Philosophy**

**May, 2011**

# Acknowledgements

I would like to express my deepest thanks to my research advisor, Prof. Mohamed F. Mokbel, for helping me to complete my PhD in University of Minnesota. I feel very lucky to work with him. He was the inspiration and motivation behind many nice research ideas. I still remembered the first chat with him about skyline computation which blossom into a large research area! He also taught me the importance of publishing in top conferences and journals, and the most important of all, how to be a good researcher. I would like to thank Prof. Shekhar, Prof. Adomavicius and Prof. Tripathi for their presence in my PhD examination committee. They have been very supportive throughout, and their comments on my work are encouraging. I am indebted to professors who taught my classes. I learnt a wide spectrum of knowledge from them. Thanks to Prof. Devor, and Prof. Du from DISC group. I enjoyed DISC group meetings as they gave me a different research prospective.

I feel very fortunate to know great co-workers, including Justin Levandoski and Ahmed El-dawy. I want to express my thanks and love to my family. They have been very supportive in these five years, and I am glad that they are happy about me. I feel I have improved a lot after these five years. I believe UMN, as well as its computer science department, will continue to flourish.

# Dedication

To my family.

## Abstract

With the increasing availability of various data sources, the preference queries are essential to find the *relevant* results to users. Several preference functions has been introduced in literature including: top-k [1], skylines [2], distributed skyline [3], spatial skyline [4], multi-objective [5], k-dominance [6], k-frequency [7], ranked skylines [8], k-representative dominance [9], distance-based dominance [10], $\epsilon$-skylines [11], top-k dominance [12], and stochastic skyline [13]. With the growing number of applications that generate *imprecise* data, e.g., sensor readings, human reading errors, and data imperfection, it has become essential to support preference queries of various types over imprecise data. *Imprecise* data can be classified into two categories: *incomplete* and *uncertain* data.

Unfortunately, existing work for preference queries for the *imprecise* data are limited and isolated. This thesis addresses efficiently extending DBMS to be preference-aware over imprecise data. First, we address the problem of skyline queries over *incomplete* data where multi-dimensional data items are missing some values of their dimensions. We show that with *incomplete* data, the dominance relation among data points may not be transitive, thus, almost all existing techniques for skyline queries are not applicable. We propose an efficient algorithm to compute the skyline over incomplete data. Then, we define preference queries over uncertain data, represented as a continuous range. We propose a novel, efficient framework to answer these preference queries. Then, we present *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries where the set of preferred attributes reside in more than one relation. The main idea of *PrefJoin* is to make the join operator aware of the required preference functionality. Finally, we extend *PrefJoin* framework to realize an efficient preference-aware operator which support *imprecise* data. The extended framework is denoted as *UPrefJoin*.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

With the increasing availability of various data sources, the preference queries are more essential to find the appropriate results [14, 15] because they give users the ability to receive query answers that are *relevant* to their requirements. Several research efforts have been dedicated to define and integrate preferences with databases, for example, [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]. An instance of preference queries is *"find my preferred restaurants based on price and distance"*. Finding the *preference* answer mainly depends on the type of the preference method employed. For example, if the preference query is a *skyline* query [2], then the preferred set of restaurants are those that are not dominated by other restaurants. A restaurant $x$ dominates restaurant $y$ if $x$ is better than $y$ in one of the dimensions (price or distance) while it is equal or better in the other dimension. Conversely, if we employ the *top-k* [1] method to answer the preference query, then the answer is the $k$ restaurants with the highest score using a function that combines price and distance together. In addition, there are numerous other preference methods we can employ, and each will return a set $\mathcal{P}$ of preferred restaurants, e.g., *multi-objective* [5], *k-dominance* [6], and *top-k dominating* [12]. However, each method differs in the way it determines the definition of $\mathcal{P}$. Due to increasing size of input data, the naïve approach of applying the preference over the output of queries yields an inefficient solution. This is practically true for imprecise data which are more expensive to compute, as we will show in the next chapters.

With the growing number of applications that generate *imprecise* data, e.g., sensor readings, human reading errors, and data imperfection, it has become essential to

support preference queries of various types over imprecise data. Generally speaking *imprecise* data can be divided into to wide categories: *incomplete* and *uncertain* data. Recent research efforts in preference query processing for uncertain data can be classified into two categories based on the uncertainty model: (a) *Discrete*, in which uncertain data is represented as a set of discrete values, and (b) *Continuous*, where uncertain data is represented as a continuous range of values. In this thesis, we integrate the preference queries over imprecise data to the operator level. Thus, we can realize useful, non-trivial applications, ranging from multi-criteria decision-making tools to *personalized* databases [14, 15].

As it will be shown in the next chapters, existing work for preference queries for the *imprecise* data are limited and isolated. For example, existing preference methods that handle the continuous uncertainty model are limited only to *nearest-neighbor* queries (i.e., *top-1*) queries [27, 28], with no obvious extension to handle the myriad other preference methods (e.g, skyline [2], multi-objective [5]). On the other hand, research efforts assuming a discrete uncertainly are [29] for *skyline* queries and [30, 31, 32] for *top-k* queries. Those research efforts cannot be easily extended to support a host of real-life applications that use a *continuous range* for uncertain values (e.g., biological data, spatial databases, sensor monitoring, and location-based services).

In this thesis, we present an integrated framework to support preference queries over uncertain data represented as a continuous range, and incomplete data. Also, we support efficient join preference-aware join query operator than can support certain and uncertain data presented as a range. More specifically, we first go beyond the completeness assumption of multi-dimensional input data where we develop new algorithms for efficient computation of skyline queries over incomplete data sets. The main reason for the need of a new set of algorithms for incomplete data is that the transitive dominance relation no longer holds. Then, we propose a framework for performing preference query evaluation over uncertain data represented as a continuous range. While our approach is novel in its scope, it has the added benefit of being able to support many well-known preference methods within a *single* framework. Our framework aims to find the probability that each object is in the list $\mathcal{P}$ of preferred answers. Given that preference answers are probabilistic in nature, we introduce two parameters that govern the size of the answer and tune query processing performance: a user-defined threshold

$H$ that defines the lowest probability an object can have to be added to $\mathcal{P}$ and a tolerance parameter $\delta$ that defines the precision for calculating an object's probability of being a preference answer. Then, we propose *PrefJoin*; an efficient preference-aware join query operator. *PrefJoin* is generic for a wide variety of preference functions, does not assume the existence of any index structure, and achieves orders of magnitude performance over previous approaches [33, 34]. The main goal of *PrefJoin* is to make the join operation aware of the required preference functionality, and hence the join operation would be able to early prune those tuples that have no chance of being a preferred tuple without actually doing the join operation. Finally, as the state-of-art for evaluating preference for uncertain data assume that the set of preferred attributes are stored in only one relation, waiving on a wide set of queries that include preference computations over multiple uncertain relations. We present *PrefJoin**, an efficient preference-aware join query operator, designed specifically to deal with preference queries over multiple *imprecise* relations.

## 1.1 Data Uncertainty Models

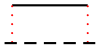
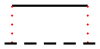
| **Image** | **Id** | **Model** | **Conf.** |
|---|---|---|---|
| 1 | $t_1$ | Chevy | 40% |
| | $t_2$ | Mazda | 10% |
| | $t_3$ | Honda | 50% |
| 2 | $t_4$ | Honda | 70% |
| | $t_5$ | Toyota | 30% |
| 3 | $t_6$ | Nissan | 50% |
| | $t_7$ | Toyota | 40% |

| **Possible World** | **Probability** |
|---|---|
| $W_1$=$t_1$,$t_4$,$t_6$ | 14% |
| $W_2$=$t_1$,$t_4$,$t_7$ | 11.2% |
| $W_3$=$t_1$,$t_4$ | 2.8% |
| $W_4$=$t_1$,$t_5$,$t_6$ | 6% |
| $W_5$=$t_1$,$t_5$,$t_7$ | 4.8% |
| $W_6$=$t_1$,$t_5$ | 1.2% |
| $W_7$=$t_2$,$t_4$,$t_6$ | 3.5% |
| $W_8$=$t_2$,$t_4$,$t_7$ | 2.8% |
| $W_9$=$t_2$,$t_4$ | 0.7% |
| $W_{10}$=$t_2$,$t_5$,$t_6$ | 1.5% |
| $W_{11}$=$t_2$,$t_5$,$t_7$ | 1.2% |
| $W_{12}$=$t_2$,$t_5$ | 0.3% |
| $W_{13}$=$t_3$,$t_4$,$t_6$ | 17.5% |
| $W_{14}$=$t_3$,$t_4$,$t_7$ | 14% |
| $W_{15}$=$t_3$,$t_4$ | 3.5% |
| $W_{16}$=$t_3$,$t_5$,$t_6$ | 7.5% |
| $W_{17}$=$t_3$,$t_5$,$t_7$ | 6% |
| $W_{18}$=$t_3$,$t_5$ | 1.5% |

Figure 1.1: Discrete Uncertain Model

As mentioned earlier, uncertain data model can be divided into two categories: discrete and continuous data. Several research efforts develop system to handle uncertainty including [35, 36, 37]. In this section, we show that discrete data can be considered as a special case of continuous uncertain data.

Figure 1.1a gives a discrete uncertain data example, where car images are identified by a visualization software. Therefore, car can only be probabilistically identified. For example, the first image is identified to be Chevy with 40% probability, Mazda with 10% probability and Honda with probability of 50%. We refer to each possible combinations of image as a possible world [32], and the probability of the this world is the product of each instance probability. Figure 1.1b represents all possible worlds along with their probability. In the rest of the thesis, we use discrete and possible world semantics interchangeably.

| **Sensor** | **Temperature** |
|---|---|
| $t_1$ |   |
| $t_2$ |   |

| **World** | **Probability** |
|---|---|
| $W_1 = t_1, t_2$ |   |

Figure 1.2: Continuous Uncertain Model

As have been discussed, discrete data limits the uncertainty within a few possible values, while continuous uncertain data assumes infinite number of values within a range [38]. A practical example is the dead-reckoning approach, as the sensor sends its reading v together with a bound d. On receiving these values, the system can treat the sensor reading as being within [v .. d, v + d], even though it does not know the sensor's actual reading. Notice that the value of d controls the trade-off between data accuracy and the communication costs. Other example is the user current GPS location, which is uncertain within a few meters. Due to infinite number of possible values, the number of possible worlds, as defined earlier, is therefore infinite. However, we can represent the probability of the possible world as a pdf function. Figure 1.2 gives a simple example for continuous uncertain model, where two sensors are used to detect current temperature. The first sensor reports temperature between 20 and 24, and the second reported temperature is uncertain between 22 to 26. The possible world represents the average temperature reading which is non-uniform and lays between 21

and 25.

## 1.2 Preference Queries

Several preference functions has been introduced in literature including: top-k [1], skylines [2], distributed skyline [3], multi-objective [5], k-dominance [6], k-frequency [7], and ranked skylines [8], k-representative dominance [9], distance-based dominance [10], $\epsilon$-skylines [11], top-k dominance [12], and stochastic skyline [13]. In this section, we highlight some widely used preference functions.

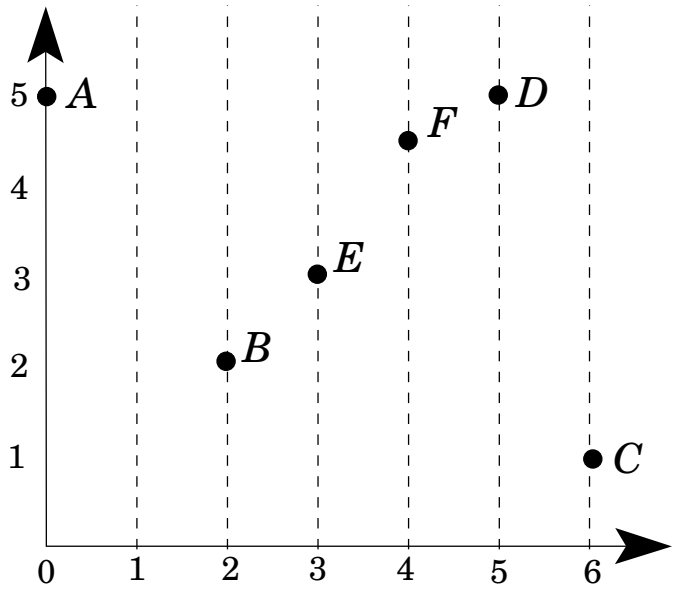



Figure 1.3: Sample Dataset

Skyline queries has been coined out in the database literature [2] to refer to the secondary storage version of the maximal vector set problem [39, 40]. Several research efforts have been proposed for efficient computations for skyline queries including: [2, 41]),[42], [43, 44, 45]). Skyline queries can be defined as follow:

**Definition 1 Skyline Query** *Skyline query in a multi-dimensional space provides the minimum set of candidates for such purposes by removing all points not preferred by any monotonic utility/scoring functions. In other words, skyline removes all objects not preferred by any user no mater how their preferences vary.*

For example, skyline query over the data set presented in Figure 1.3, assuming minimum values are better in both dimensions, returns set $\{A,B,C\}$.

Top-K has been introduced in database literature in [1]. A naïve approach to answer Top-k queries can be evaluated by first computing the score of all objects in the dataset, then order the objects based on the score, and then return the best $k$ objects according to the score. And it can be defined as follow:

**Definition 2 Top-K Query** *Top-k query in a multi-dimensional space returns the k objects having the best score according to a* monotone *scoring function.*

For example, Top-3 query over the data set presented in Figure 1.3, assuming scoring function is the sum of the dimensions, returns set $\{B,A,E\}$.

Other preference function that has been defined is top-k dominating queries [12]. This query is an important tool for decision support since it provides data analysts an intuitive way for finding significant objects. It tries to combine the skyline and top-k semantic into a single function. It can be defined as follow:

**Definition 3 Top-K dominating Query** *The top-k dominating query returns k data objects which dominate the highest number of objects in a dataset. A D-dimensional object $x$ is said to dominate D-dimensional object $y$, if the value of $x.k$ is better than or equal than that of $y.k$ over all dimensions $1 \leq k \leq D$ and with a dimension $l$ such that $x.l > y.l$.*

For example, Top-3 dominating query over the data set presented in Figure 1.3 returns set $\{B,E,F\}$.

On the other hand, a *multi-objective* preference query [5] combines subsets of preference attributes using monotone scoring functions, and performs a skyline over the new transformed combined attributes. It combines skyline and top-k computations into a single function. It can be defined as follow:

**Definition 4 Multi-objective Query** *Given a dataset $\mathcal{D}$ of $l$-dimensional object, and $n$ monotone objective functions over object's attributes $f_1$, $f_2$, ..., $f_n$, a multi-objective query finds the set of object that are not dominated with respect to the objective functions.*

The multi-objective query can be reduced into top-k if the monotone objective function combines all the dimensions into one dimension. Also, it can be reduced to be skyline query if each objective function is defined only on a single attribute.

## 1.3 Thesis Organization

To summarize, this thesis addresses efficiently realizing a preference-aware database management system that can handle imprecise data. Specifically, we study how preference queries can be answered in a correct, efficient and scalable manner for incomplete and uncertain data. Our main contributions are:

- We introduce and define preference queries over one or more relation of imprecise data. Imprecise data includes imprecise and uncertain data represented as a continuous range of values.
- We introduce efficient and scalable framework to answer preference queries over one or more relation of imprecise data.
- We give experimental evidence for scalability and efficiency of our proposed framework. Also, we give proof of correctness.
- We implement our proposed frameworks inside Postgresql, an open-source DBMS.

The organization of this thesis is as follows:

Existing preference algorithms assume that all dimensions are available for all objects. In Chapter 2, we goes beyond this restrictive assumption as we address the more practical case of involving incomplete data items (i.e., data items missing values in some of their dimensions). In this chapter, we only address *skyline* queries, as it inherits an interesting dominance property. For the traditional skyline queries the dominance relation is transitive. On the contrary, for incomplete skyline, the dominance relation is non-transitive which may lead to a cyclic dominance behavior, and invalidate the existing skyline techniques. We first propose two algorithms, namely, "Replacement" and "Bucket" that use traditional skyline algorithms for incomplete data. Then, we propose the "ISkyline" algorithm that is designed specifically for the case of incomplete data. The "ISkyline" algorithm employs two optimization techniques, namely,

virtual points and shadow skylines to tolerate cyclic dominance relations. Experimental evidence shows that the "ISkyline" algorithm significantly outperforms variations of traditional skyline algorithms.

Then in Chapter 3, we addresses the problem of efficiently processing preference queries (e.g., skyline, multi-objective, top-$k$) for uncertain data represented as a range. Therefore, query answers are probabilistic, i.e., each object is associated with a probability value of being a preferred answer. We present UPref, an efficient and flexible framework to answer preference queries over uncertain data. UPref users can specify a probability threshold that each object must exceed in order to be considered a preference answer, and a tolerance value that defines the allowed error margin in probability calculation. UPref employs an efficient filter-refine paradigm to answer these preference queries. UPref framework consists of two phases. The first phase compares data objects pairwise, and uses a novel method that immediately *filters* objects that are guaranteed not to be preferred answers (i.e., have probabilities *below* a given threshold). The second phase refines the probability of each object not filtered in the first phase by implementing an efficient stepwise method to calculate an object's final probability within a given tolerance value. In short, UPref realizes a *flexible*, and *efficient* framework that is capable of answering skyline, multi-objective, and top-$k$ preference queries all within a *single* framework. Most of the evaluation techniques for preference queries assume that the set of preferred attributes are stored in only one relation, waiving on a wide set of queries that include preference computations over multiple relations.

Chapter 4 presents *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries over multiple relations. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively. *PrefJoin* supports a variety of preference function including skyline, multi-objective and $k$-dominance preference queries.

In Chapter 5, we extend *PrefJoin* framework presented in Chapter 4 to address uncertain and incomplete data. The proposed framework is denoted as *PrefJoin**. For

uncertain data, we employ a user-defined threshold ($H$) and tolerance($\delta$), as presented in Chapter 3, to reduce the burden of computations and to give only the interesting objects to users. The extended framework supports a variety of preference function including skyline, multi-objective and $k$-dominance uncertain preference queries, and incomplete skyline.

# Chapter 2

# Skyline Query Processing for Incomplete Data

Recently, there has been much interest in processing skyline queries for various applications that include decision making, personalized services, and search pruning. Skyline queries aim to prune a search space of large numbers of multi-dimensional data items to a small set of interesting items by eliminating items that are dominated by others. Existing skyline algorithms assume that all dimensions are available for all data items. This chapter goes beyond this restrictive assumption as we address the more practical case of involving incomplete data items (i.e., data items missing values in some of their dimensions). In contrast to the case of complete data where the dominance relation is transitive, incomplete data suffer from non-transitive dominance relation which may lead to a cyclic dominance behavior. We first propose two algorithms, namely, "Replacement" and "Bucket" that use traditional skyline algorithms for incomplete data. Then, we propose the "ISkyline" algorithm that is designed specifically for the case of incomplete data. The "ISkyline" algorithm employs two optimization techniques, namely, virtual points and shadow skylines to tolerate cyclic dominance relations. Experimental evidence shows that the "ISkyline" algorithm significantly outperforms variations of traditional skyline algorithms.

## 2.1 Introduction

Given a search space of $D$ independent dimensions, $u_1$, $u_2$, $\cdots$, $u_d$, a point $p_i$ is said to *dominate* another point $p_j$ if the value of $p_i.u_k$ is better than or equal than that of $p_j.u_k$ over all dimensions $1 \leq k \leq D$ and with a dimension $l$ such that $p_i.u_l > p_j.u_l$. A skyline query over a set $S$ of $D$-dimensional points aims to find a set of points $S_{sky} \subseteq S$ where any point $p_{sky} \in S_{sky}$ is not *dominated* by any point in $S$ while each point $p_i \in S - S_{sky}$ is *dominated* by some point in $S$. In general, a skyline query reduces the search space $S$ to only the set of skyline points $S_{sky}$ that are of interest to the user. Skyline queries are widely applicable to multi-criteria decision making applications. For example, consider the classical scenario where a user wants to reserve a hotel that is near to the conference site and cheaper in price among a large set of hotels. A hotel $h_i$ is represented as a two-dimensional point $(d_i, r_i)$ where $d_i$ and $r_i$ represent the distance and price of the hotel, respectively. Rather than investigating in the whole space of the hotels, a skyline query eliminates any hotel $h_j$ where there is another hotel $h_k$ that is both cheaper and closer to the conference site than $h_j$. Another example of skyline queries is a movie rating application (e.g., MovieLens [46]) in which $D$ system users rank various movies. In this case, each movie is represented as a $D$-dimensional point where each dimension corresponds to a certain user. When searching for the best movie, a skyline query eliminates those movies for which all users agree there exists at least one other superior (i.e., overall better-ranked) movie.

Due to the importance of skyline queries, several research efforts have been dedicated to develop efficient skyline query processors (e.g., see [6, 42, 43, 44, 45, 47, 48]). Almost all of these algorithms rely mainly on two implicit assumptions: (1) Data are complete, i.e., all dimensions are available for all data items. Such an assumption of completeness is not practical in many cases. For example, consider the movie rating application [46] with hundreds of users rating thousands of movies. It is highly unlikely that every single user will rate all movies. Instead, a user will rate only the movies that interest her. As a result, each movie will be represented as a $D$-dimensional point with several *blank* (i.e., *incomplete*) dimensions. Another example is from the hotel application where some hotels may not disclose some of their properties. These undisclosed properties are represented as *incomplete* entries within the hotel multi-dimensional point

representation. (2) With the exception of [6], all skyline algorithms assume transitivity in the dominance relation, i.e., if data item $p_i$ dominates $p_j$ while $p_j$ dominates $p_k$, then $p_i$ dominates $p_k$. Using the transitivity property, skyline query processing algorithms exploit various ways of data pruning and indexing. Unfortunately, as will be seen in this chapter, the transitive dominance relation is not applicable to the case of *incomplete* data.

In this chapter, we go beyond the completeness assumption of multi-dimensional input data where we develop new algorithms for efficient computation of skyline queries over incomplete data sets. The main reason for the need of a new set of algorithms for incomplete data is that the transitive dominance relation no longer holds. For example, we could have three data items $p_i$, $p_j$, and $p_k$, where $p_i$ dominates $p_j$, $p_j$ dominates $p_k$, while $p_k$ dominates $p_i$. In this case, we are not only missing the transitive dominance relation as $p_i$ does not dominate $p_k$, but we also face another problem where we have a *cyclic* dominance relation between $p_i$, $p_j$, $p_k$. Under this *cyclic* dominance relation, none of these three points can be considered a skyline as each point is dominated by at least one other point.

We start by introducing two variations of traditional skyline algorithms to accommodate the existence of *incomplete* data, namely, the *Replacement*, and the *Bucket* algorithms. Then, we introduce the *ISkyline* algorithm as a specialized algorithm for the case of *incomplete* data. The *ISkyline* algorithm employs two new concepts, namely, *virtual points* and *shadow skylines* to enable efficient execution of skyline queries over *incomplete* data. For an input data item $p$ to be reported as a skyline by the *ISkyline* algorithm, it has to pass through three phases where $p$ should be considered as a *local* skyline in the first phase, then, as a *candidate* skyline in the second phase, and finally, as a *global* skyline (i.e., query result) in the third phase. The *ISkyline* algorithm does not assume any preprocessing for input data items as input is streamed into the algorithm directly. In general, the contributions of this chapter can be summarized as follows:

- We define the *dominance* relation for *incomplete* data and we show that the transitive dominance relation does not hold for *incomplete* data.
- We introduce two new algorithms, namely, *Replacement* and *Bucket* algorithms that utilize existing skyline algorithms to accommodate *incomplete* data.

- We introduce the *ISkyline* algorithm as a novel algorithm designed specifically for efficient skyline computation over *incomplete* data.
- We provide a proof of correctness for the *ISkyline* algorithm.
- We give experimental evidence that the *ISkyline* algorithm is efficient, scalable, and clearly outperforms the variations of traditional skyline algorithms.

The rest of this chapter is organized as follows: Section 2.2 highlights related work. Preliminary discussion and problem formulation are given in Section 2.3. Section 2.4 provides two variations of traditional skyline algorithms for *incomplete* data. The *ISkyline* algorithm is introduced in Section 2.5 while its proof of correctness is given in Section 2.6. Section 2.7 gives experimental evidence for the efficiency of our algorithms. Finally, Section 2.8 concludes this chapter.

## 2.2 Related Work

The term *skyline queries* has been coined out in the database literature [2] to refer to the secondary storage version of the maximal vector set problem [39, 40, 49]. Since then, several algorithms have been proposed for skyline queries that include no preprocessing solutions (e.g., [2, 41]), presorting solutions (e.g., [42]), and index-based solutions (e.g., [43, 44, 45, 50, 51, 52]). Due to its practicality, several research efforts have been dedicated to developing various *skyline* algorithms for various environments, e.g., partially-ordered domains [53], high-dimensional data [6, 54, 47], skyline cube [47, 55, 56], sliding window [57, 58], reverse skyline [59, 60], weak skyline [61], continuous skyline computations [62, 63], mobile ad-hoc networks [64], spatial skylines [4], indexing for skyline [65], cardinality estimation [66], web information systems [3], and data mining [67]. Unfortunately, all these algorithms consider only the case of complete data with no direct extension of considering the case of *incomplete* data where the dominance relation is not transitive.

The closest work to ours is the $k$-dominant skyline problem [6] in which a point $p$ is considered to dominate point $q$ if only a subset of size $k$ of the dimensions in $p$ dominates the corresponding dimensions in $q$. Under this definition, the dominance relation turns to be non-transitive, which is the case also for *incomplete* data. The $k$-dominant skyline

algorithm overcomes the non-transitive property by discarding only those points that are dominated in all dimensions while keeping those points that are only dominated in $k$ dimensions. As the $k$-dominant skyline algorithm considers only the case of complete data, applying it directly to the case of *incomplete* data misses the opportunity to make use of *incomplete* subspaces Thus, applying the $k$-dominant algorithm directly to the case of *incomplete* data would result in prohibitive costs that can be avoided with the knowledge of *incomplete* dimensions.

## 2.3 Preliminaries

This section presents a preliminary discussion about *incomplete* data. Throughout the rest of this chapter, we denote *incomplete* (i.e., unknown) dimensions by a dash "−". For example, a three-dimensional point $p$ with values $a$ and $b$ in the first two dimensions and an unknown value in the third dimension is represented as $(a, b, -)$. Without loss of generality, we assume that all dimension values have a total order in which greater values are considered superior. With these two considerations, the problem of computing skylines over *incomplete* data is formulated as follows:

**Problem Formulation.** *Given a set $S$ of $D$-dimensional points where each point $P = (u_1, u_2, \cdots, u_d)$ has at least one known dimension $u_i$, while all other dimensions have a non-zero probability of being unknown (i.e., there is a non-zero probability that $u_k =' -', k \neq i$), find the set of skyline points $S_{sky} \subset S$ such that every point $P \in S_{sky}$ is not dominated by any other point in $S$ while every point $Q \in S - S_{sky}$ is dominated by some other point in $S$.*

### 2.3.1 Dominance Relation for *Incomplete* Data

For complete data, a point $p_i$ is said to dominate point $p_j$ if $p_i$ is better than or equal to $p_j$ in all dimensions and is strictly better than $p_j$ in at least one dimension. Unfortunately, with the existence of some *incomplete* dimensions, we cannot simply use the traditional definition of the dominance relation as it is not immediately clear how to compare an *incomplete* dimensions with a corresponding compete dimension. For example, if $p_i = (1, -, 3)$ and $p_j = (-, 2, -)$, we cannot judge which point is superior in any of the three dimensions. To accommodate the existence of *incomplete* data, we introduce the

following new definition of the dominance relation:

**Definition 5** *Given any two D-dimensional points P and Q that may have incomplete dimensions, a point P is said to dominate another point Q if the following two conditions hold: (1) There is at least one dimension $u_i$ where both $P.u_i$ and $Q.u_i$ are known, and $P.u_i > Q.u_i$ (2) For all other dimensions $j$, $j \neq i$, either $P.u_j$ is unknown, $Q.u_j$ is unknown, or $P.u_j \geq Q.u_j$.*

In other words, for any two *incomplete* points, $p_i$ and $p_j$, we consider only the common dimensions that are known in both points. Among these common dimensions only, we apply the traditional dominance relation to decide which point dominates the other, if any. For example, consider the four-dimensional points $p_i = (1, -, -, 3)$ and $p_j = (-, -, 3, 1)$; $p_i$ is said to dominate $p_j$ as the only common dimension is the fourth, for which $p_i.u_4 > p_j.u_4$. As another example, consider the case where $q_i = (1, -, -, 3)$ and $q_j = (-, 1, 2, -)$. In this case, no single dimension $u$ exists for which $q_i.u$ and $q_j.u$ are known. Thus, neither $q_i$ nor $q_j$ dominate the other.

**"Cyclic" and "Non-transitive" dominance relation.** Unfortunately, with this definition of the dominance relation over incomplete data, we: (1) lose the transitive dominance property that was the basis of almost all previous skyline query processing algorithms, and (2) may end up having a *cyclic* dominance relation in which none of the points in a data set is considered a skyline. For example, consider the following three *incomplete* points $p_1 = (4, 3, 4, -)$, $p_2 = (2, 1, -, 5)$, and $p_3 = (-, -, 5, 2)$. According to Definition 5, $p_1$ dominates $p_2$ as $p_1$ is greater in the common dimensions (i.e., first and second dimensions). Also, $p_2$ dominates $p_3$ as the only common dimension is the fourth one in which $p_2$ is greater. However, when comparing $p_1$ to $p_3$, the third dimension is the only common dimension in which $p_3$ is greater. Thus, $p_1$ does not dominate $p_3$ which means that the dominance relation is *non-transitive*. Moreover, $p_3$ does nominate $p_1$ which means that the dominance relation ends to be *cyclic*. In this case of *cyclic* dominance, none of the three points can be considered a skyline as all of them are dominated.

### 2.3.2 Bitmap Representation

For ease of representation and computation, we represent a $D$-dimensional *incomplete* point $P$ by a bitmap vector $P.B$ of $D$ bits that include 1's for all complete dimensions and 0's for all *incomplete* dimensions. For example, points $P$=(4,-,5,-) and $Q$=(-,3,3,2) are represented by the bitmaps $P.B = 1010$ and $Q.B = 0111$, respectively. With bitmap representation, two points are considered *comparable* if there is at least one common complete dimension between their two bitmaps, i.e., the bitwise AND operation of their bitmaps has a non-zero value. For example, the previous points $P$ and $Q$ are *comparable* as 1010 AND 0111 is 0010. Formally, the *comparable* relation is defined as follows:

**Definition 6** *Two points $P$ and $Q$ are comparable if and only if the bitwise-and of their bitmaps is not zero.*

## 2.4 Using Traditional Skyline Algorithms for Incomplete Data

As *incomplete* data suffer from a *cyclic* and *non-transitive* dominance relation, we cannot simply use existing traditional skyline algorithms. A naive solution for *incomplete* data is to do an exhaustive pairwise comparison between all input points and select only those points that are not dominated. For very large input sizes, this naive solution is not feasible. In this section, we improve upon the naive solution by introducing two new algorithms, namely, the *Replacement* and the *Bucket* algorithms that tailor existing skyline algorithms to work for *incomplete* data.

**The *Replacement* Algorithm.** The main idea of the *Replacement* algorithm is to replace any *incomplete* dimension in a data item by $-\infty$. By doing so, all *incomplete* dimensions are transformed to *complete* dimensions. Then, we can apply any traditional skyline algorithm to get the set $S_{sky-\infty}$ of current skylines. Then, for all points in $S_{sky-\infty}$, we *replace* the $-\infty$ values by an *incomplete* dimension, i.e., return to the original form. Finally, we perform an exhaustive pairwise comparison between all *incomplete* points that are in $S_{sky-\infty}$ to find the actual skyline points. The *replacement* algorithm greatly improves upon the naive method as the exhaustive pairwise comparison is done only for those points in $S_{sky-\infty}$ rather than all input data points.

The correctness of this algorithm comes from the fact that if an *incomplete* point $P$ is a skyline in a set $S$, then the point $P_{-\infty}$ would be a skyline in $S_{-\infty}$. $P_{-\infty}$ and $S_{-\infty}$ are formed by replacing *incomplete* dimensions of $P$ and all points in $S$ by $-\infty$, respectively. The rationale behind this argument is that if $P$ is a skyline in $S$, then there is no point $Q$ in $S$ that dominates $P$. This means that within the comparable dimensions of $P$ and $Q$, $P$ would be superior. So, when forming $P_{-\infty}$ and $Q_{-\infty}$, we would still maintain the values of the comparable dimensions as they were in $P$ and $Q$. Thus, $Q_{-\infty}$ cannot dominate $P_{-\infty}$ and thus $P_{-\infty}$ would still be a skyline in $S_{-\infty}$. Notice that although $P$ dominates $Q$ in $S$, there is not guarantee that $P_{-\infty}$ would dominate $Q_{-\infty}$. For example, consider $P = (5, 2, -, 2)$, $Q = (3, -, 5, 1)$, although $P$ dominates $Q$, $P_{-\infty} = (5, 2, -\infty, 2)$ does not dominate $Q_{-\infty} = (3, -\infty, 5, 1)$. Thus, the skyline points in $S_{-\infty}$ is a superset of the skyline points in $S$.

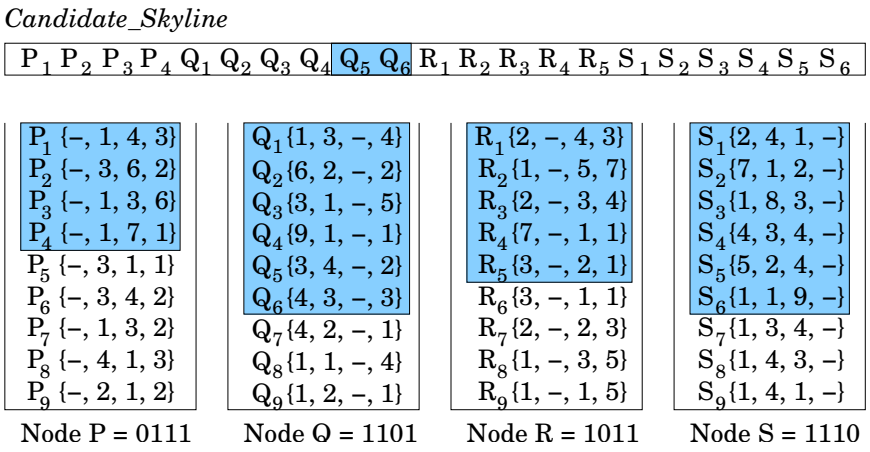



Figure 2.1: The Bucket algorithm

**The *Bucket* Algorithm.** The main idea of the *Bucket* algorithm is to divide all incoming points into distinct buckets where all points in each bucket have the same bitmap representation. By doing so, the transitive dominance relation would hold among all points within the same bucket. Then, we can apply a traditional skyline algorithm over all points within each bucket by ignoring the *incomplete* dimensions. We would call the set of skylines for each bucket as a *local* skyline. Finally, we collect the points from all *local* skyline sets and include them in one list, termed *candidate* skyline, list in

which we perform an exhaustive pairwise comparison among all points to get the query answer. The correctness of the *Bucket* algorithm comes from the fact that for a point to be a skyline, it has first to be a *local* skyline among all points in its bucket. Also, if a point $P_i$ is a *local* skyline in bucket $P$, it needs to be compared only against all *local* skyline of other *comparable* buckets.

Figure 2.1 gives an example of the *Bucket* algorithm in which 36 points are divided into four buckets, $P$, $Q$, $R$, and $S$ based on their bitmaps. For each bucket, we compute the *local* skylines separately, depicted by shaded rectangle in Figure 2.1. Overall, we have 21 *local* skyline points that are considered as *candidate* skylines in which we perform an exhaustive pairwise comparison to conclude that $Q_5$ and $Q_6$ are the skylines over all 36 points.

In general, the *Bucket* algorithm gives better performance than the *Replacement* algorithm for two reasons: (1) The size of the *candidate* list in the *Bucket* algorithm is likely to be much less than the size of the set $S_{sky-\infty}$ in the *Replacement* algorithm, thus, the exhaustive pairwise comparison would be cheaper. (2) Applying a traditional skyline algorithm several times for few data items in each bucket, as in the *Bucket* algorithm, is cheaper than applying it once over all data items, as in the *Replacement* algorithm.

## 2.5 Efficient Skyline Computation for Incomplete Data

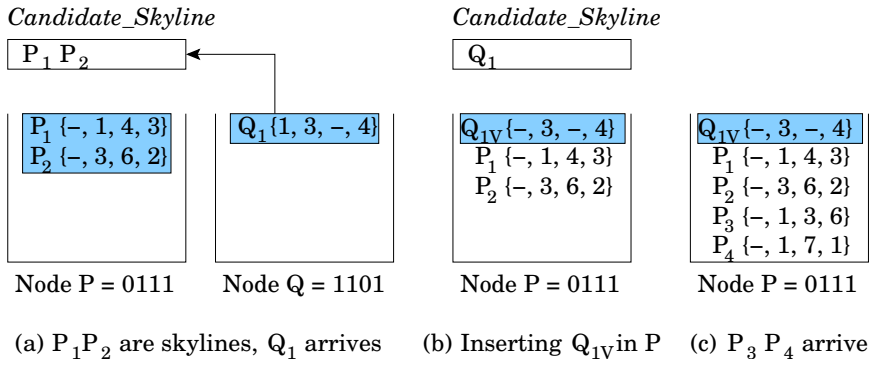



Figure 2.2: Virtual point insertion

The *Bucket* algorithm presented in Section 2.4 suffers from two main drawbacks.

First, the size of the *candidate* skylines may be excessive as it is the union of all *local* skylines in all buckets. With such excessive size, the exhaustive pairwise comparison among *candidate* points would dominate the algorithm running time. Second, the *local* skyline at each bucket is computed independently from all other buckets, hence, missing a chance of using other bucket data to reduce the number of comparisons needed for *local* skyline computation. In this section, we introduce, the *ISkyline* algorithm for efficient skyline computation of incomplete data. The *ISkyline* algorithm avoids the drawbacks of the *Bucket* algorithm by introducing two main concepts, namely, *virtual points* and *shadow skylines*. In the rest of this section, we will introduce and motivate the concepts of *virtual points* and *shadow skylines*. Then we will discuss the details of the *ISkyline* algorithm.

### 2.5.1 Virtual Points and Shadow Skylines

**Virtual Points.** The main purpose of *virtual points* is to reduce the number of points in the *candidate* skyline list. The main idea is that a point $X$ in a bucket $N_i$ can be used to reduce the number of *local* skylines in a bucket $N_j$ where $i \neq j$. By doing so, the number of *local* skyline at each bucket can be reduced, and hence, the number of *candidate* skylines can be reduced significantly. Figure 2.2 illustrates the idea of *virtual points* when applied to the example in Figure 2.1. In this case, we compute the *local* skyline for each bucket and the *candidate* skyline online while we read the input data, as no pre-processing is assumed. The current *local* skyline for node $P$ is $P_1$ and $P_2$; these points are also inserted into the *candidate* skyline list. Figure 2.2a shows the time instance in which we read $Q_1$ as a skyline point for node $Q$. In this case, we compare $Q_1$ against all points in the *candidate* skyline list that have *comparable* bitmaps to that of $Q_1$, i.e., $P_1$ and $P_2$. Since $Q_1$ dominates both points, we remove $P_1$ and $P_2$ from the *candidate* list while keeping only $Q_1$. Notice that, up to now, this scenario is also applicable to the *Bucket* algorithm.

However, the *ISkyline* algorithm distinguishes itself as it creates a *virtual point* $Q_{1v}$ out of $Q_1$. $Q_{1v}$ will be inserted in node $P$ to reduce the number of *local* skylines. The main idea is that for an incoming point $P_x$ to be a *local* skyline in $P$, it must not be dominated by $Q_{1v}$. $Q_{1v}$ is formed by considering only the common dimensions in the bitmaps of nodes $P$ and $Q$. Figure 2.2b gives an example where $Q_{1v}$ is formed as

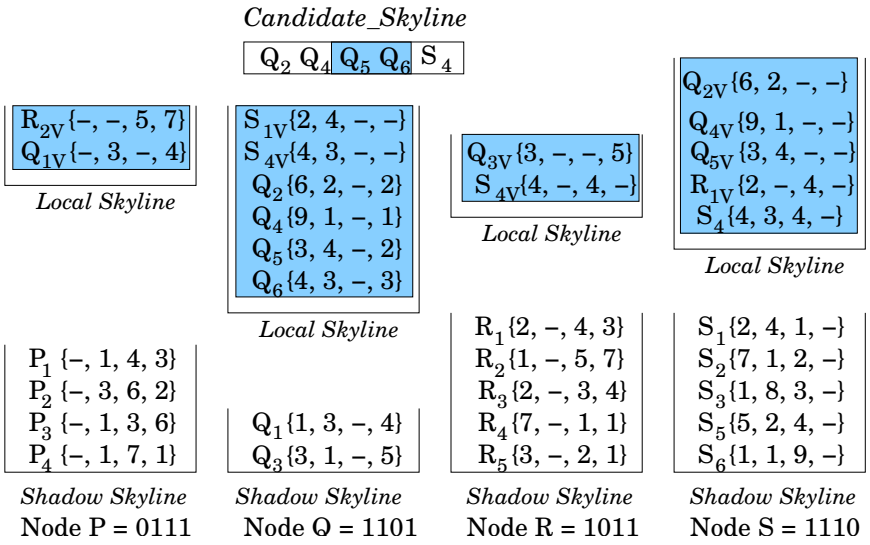



Figure 2.3: Final effect of Shadow Skyline

$(-, 3, -, 4)$ and inserted into $P$. Notice that currently, the *local* skyline of $P$ includes only $Q_{1v}$. Figure 2.2c gives the process of reading input points $P_3$, $P_4$, and $P_5$. Since all points are dominated by the *virtual point* $Q_{1v}$, we ignore $P_3$ , $P_4$ and $P_5$ by neither storing them as *local* skylines nor propagating them to be *candidate* skylines. By doing so, we significantly reduce the size of the *candidate* skyline list. For example, compare the *local* skyline list of $P$ in Figure 2.2c that includes only one point, $Q_{1v}$, to that of Figure 2.1 that includes four points $P_1$, $P_2$, $P_3$, and $P_4$. Moreover, as $Q_{1v}$ is a *virtual point*, it is not propagated to the *candidate* skyline list. So, in the *ISkyline* algorithm (Figure 2.2), none of the points in $P$ becomes *candidates*, while in the *Bucket* algorithm (Figure 2.1), four points from $P$ are *candidates*.

Figure 2.4 gives the end result of *local* skylines at each bucket and the *candidate_skyline* list after reading all the input data and employing the *virtual point* concept. It can be seen that employing the *virtual point* concept reduces the size of the *candidate_skyline* list to 5 instead of 26, as in the *Bucket* algorithm.

**Shadow Skylines.** With *virtual points*, we cannot simply perform an exhaustive pairwise comparison of all points in the *candidate* skyline list to get the query result. Instead, a point $X$ in the *candidate* list should be compared against any point $Y$ with a comparable bitmap regardless of $Y$ being a *candidate* skyline or not. Thus, in contrast to the

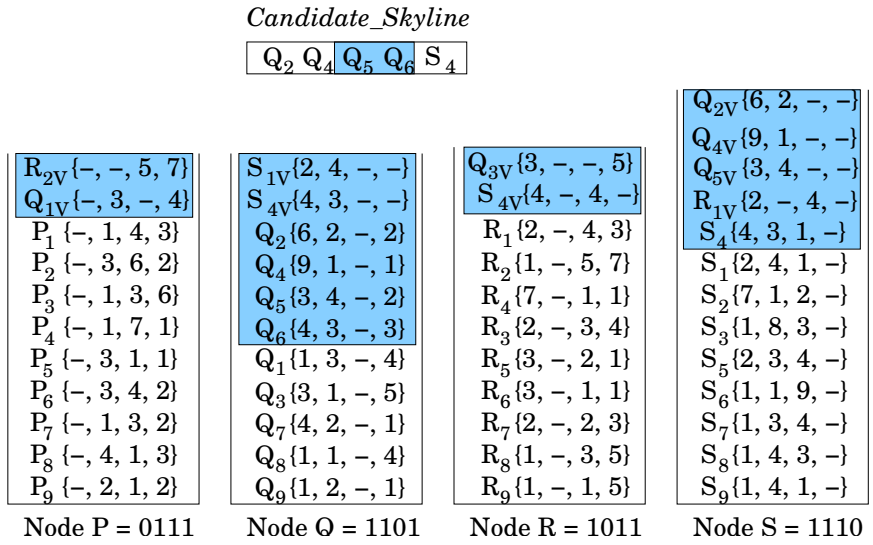



Figure 2.4: Final effect of Virtual points

*Bucket* algorithm, we cannot simply discard a point $Y$ because it is not a *local* skyline as $Y$ may help later in dominating *candidate* skylines. For example, in Figure 2.4, although $P_3$ is not stored in the *local* skyline list of $P$, it dominates $Q_4$ from the *candidate* list. To see how this scenario may take place, consider the case of Figure 2.2c in which $P_3$ is not stored in the *local* skyline list as it is dominated by $Q_{1v}$. Whenever $Q_4$ arrives, $Q_4$ will not be compared to $P_3$ as $P_3$ is not a *local* skyline. Thus, $Q_4$ will be stored as a *candidate* skyline although it is dominated by $P_3$. This scenario does not affect the correctness of the algorithm, however, it causes an overhead of not being able to discard any dominated points. Thus, for bucket $P$, we store all points $P_1$ to $P_9$, instead of storing only $P_1$ to $P_4$ as in the *Bucket* algorithm. It is important to note that even with this side effect, *virtual points* still perform better than the *Bucket* algorithm as the savings in the size of *candidate* and *local* skylines is much more powerful than the drawback of storing and comparing all points.

The *ISkyline* algorithm introduces the concept of *shadow skylines* that works together with *virtual points* to alleviate the problem of storing and comparing all input data. The main idea is that we do not need to store all points in each bucket, instead, we only need to store the skyline set of points not found in the local skyline list. For example, in bucket $P$, instead of storing all points $P_1$ to $P_9$, we need to store only $P_1$ to $P_4$ as

these are the skyline points of $P_1$ to $P_9$. In this case, we will call $P_1$ to $P_4$ as the *shadow* skyline of $P$. The *shadow skyline* of a bucket $N$ is the set of points that are real skylines among all points in $N$ minus those points that are stored in the *local* skyline of $N$. For example, consider bucket $Q$ in Figure 2.1, where the original skyline set includes points $Q_1$ to $Q_6$. However, with *virtual points* (Figure 2.4), only points $Q_2$, $Q_4$, $Q_5$, and $Q_6$ are stored in the *local* skyline set of $Q$. Thus, the *shadow* skyline of $Q$ includes $Q_1$ and $Q_3$. Figure 2.3 gives the list of *local* and *shadow* skylines of each bucket. Points that are not shown in the figure are discarded by the *ISkyline* algorithm. By doing so, the storage increase of the *ISkyline* algorithm over the *Bucket* algorithm is limited only to the *virtual points*. Moreover, as we will show in the algorithm details, the *candidate* skylines need to be compared only against points in the *shadow* skyline rather than all points.

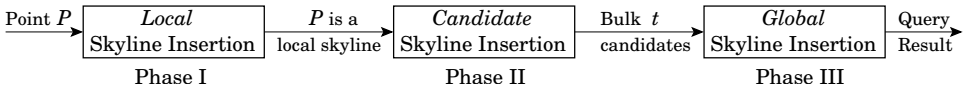



Figure 2.5: Phases of the ISkyline Algorithms

### 2.5.2 The *ISkyline* Algorithm

This section presents the *ISkyline* algorithm that employs the concepts of *virtual points* and *shadow skylines* for efficient skyline computation of incomplete data. The *ISkyline* algorithm has a tuning parameter $t$ that controls the frequency of updating the skyline result. Basically, the *ISkyline* algorithm bulks $t$ *candidate* skyline points together and process them once in order to get the query result. A small value of $t$ indicates that the query result will be updated more frequently than that of a high value of $t$.

**Data structure.** With each bucket node $N$ associated with the bitmap $N.B$, we store three pieces of information: (1) The *local_skyline* list that may contain both real and *virtual* points as shown in Figure 2.3, (2) The *shadow_skyline* list that contains only real data points as shown in Figure 2.3, and (3) A flag *updated* that is turned on only when the *shadow_skyline* list is modified. Such flag significantly prunes the search space by avoiding looking at unmodified buckets. It is important to note that the number of buckets we maintain is the same as the number of distinct bitmaps of all input data. To

access bucket nodes by their bitmaps, we maintain a hash table with the entry < *bitmap*, *node pointer*> that associates each available bitmap with one bucket node. Finally, we maintain two lists, *candidate_skyline* and *global_skyline* that maintain current *candidate* skylines and the query result, respectively.

Figure 2.5 gives an overview of the *ISkyline* algorithm that reads data sequentially from an input file with no assumptions about index availability or data preprocessing. The main idea is that each input point $P$ may pass through *up to* three phases (depicted by rectangles in Figure 2.5). In Phase I, for each point $P$ in node $N$, we check if $P$ needs to be (a) stored in the *local_skyline* list of $N$, (b) stored in the *shadow_skyline* list of $N$, or (c) completely discarded. Only those points that are stored in the *local_skyline* list go onto Phase II. For each point $P$ in Phase II, we check if $P$ needs to be stored in the *candidate_skyline* list. This phase will also determine whether *virtual points* should be inserted in other node buckets based on the comparison of $P$ with other points in the *candidate_skyline* list. Once we have $t$ points in the *candidate_skyline* list, we move to Phase III where we update the list of points in the *global_skyline* list, i.e., the current query answer.

**Algorithm 1** Skyline Computation for Incomplete Data

```
1:  Function ISkyline(Data Set S, Threshold t)
2:  global_skyline ← {},Candidate_skyline ← {}
3:  repeat
4:    Read point P from input S
5:    Node N ← Node that corresponds to P bitmap from Hash Table
6:    if N = φ, then create and initialize node N with P bitmap
7:    Is_skyline ← Insert_Local_Skyline(P,N)  (see Algorithm 2)
8:    if Is_skyline = true then
9:      Insert_Candidate_Skyline(P)  (see Algorithm 3)
10:     if |Candidate_Skyline| > t then
11:       Update_Global_Skyline()  (see Algorithm 4)
12:       Candidate_Skyline ← {}
13:     end if
14:   end if
15: until End of input S
16: Update_Global_Skyline()  (see Algorithm 4)
17: return global_skyline
```

Algorithm 1 gives the pseudo code of the *ISkyline* algorithm. The input to the

algorithm is: (1) a set $S$ of data points and (2) the tuning parameter $t$. The output of the algorithm is the set of *global* skylines. The algorithm starts by initializing the *global_skyline* and *candidate_skyline* lists. Then, for every input point $P \in S$, we either retrieve its corresponding node $N$ from the hash table or create $N$ if it does not exist (Lines 5 to 6 in Algorithm 1). Then, we start in Phase I where we attempt to insert $P$ into the *local_skyline* list (Line 7 in Algorithm 1). If $P$ ends up to be a *local* skyline of $N$, we start Phase II where we attempt to add $P$ to the *candidate_skyline* list (Line 9 in Algorithm 1). Whenever the number of points in the *candidate_skyline* list exceeds $t$, we move to Phase III where we update the list of *global skylines* and clear the *candidate_skyline* list (Lines 11 to 12 in Algorithm 1). Finally, once we reach to the end of input, we update the list of *global skylines* and conclude the algorithm (Line 16 in Algorithm 1). In the rest of this section, we will discuss in details the three phases of the *ISkyline* algorithm.

**Phase I: *Local* Skyline Insertion**

Algorithm 2 gives the pseudo code of Phase I in which, for each point $P$, we either store it in the *local_skyline* list, store it in the *shadow_skyline* list, or discard it. Basically, we check if $P$ is not dominated by any point in the *local_skyline* list. If this is the case, we decide to store $P$ in the *local_skyline* list, update the entries in both the *local_skyline* and *shadow_skyline* lists accordingly, and return *true* to indicate that $P$ is a *skyline* for node $N$ (Lines 3 to 5 in Algorithm 2). It is important to note that we do not remove *virtual points* from the *local_skyline* even those *virtual points* are dominated by $P$. The main idea is that those virtual points may dominate other points that cannot be dominated by $P$. For example, in Figure 2.3, although point $S_4$ dominates the virtual point $R_{1v}$, we did not remove $R_{1v}$. By doing so, $R_{1V}$ later dominated point $S_3$ which is not dominated by $S_4$. Thus, we intentionally do not remove *virtual points* as they could help in reducing the search space for *local* and *candidate* skylines. On the other hand, if $P$ ends up to be dominated by some point in the *local_skyline* list, we check if $P$ is dominated only by *virtual points*. If this is the case, we decide to insert $P$ in the *shadow_skyline* list of $N$, set the *updated* flag of $N$ to be *true* to indicate a change in the *shadow_skyline* list, update the list of *shadow* skylines accordingly, and return *false* (Lines 6 to 11 in Algorithm 2). It is important to note that by being dominated by *virtual points* only, $P$

is considered to be a *skyline* among all current points of the same bitmap. That is why we keep $P$ in the *shadow* list. Finally, If $P$ was dominated by at least one real point from the *local_skyline* list of $N$, we simply discard $P$ and return *false*.

**Algorithm 2** Phase I: *Local* Skyline Insertion

```
1:  Function Insert_Local_Skyline(Point P, Node N)
2:  if P is not dominated by any point in the local_skyline list of N then
3:      Insert P into local_skyline list of N
4:      Delete all real points that are dominated by P from the local_sklyine and
        shadow_skyline lists of N
5:      return true
6:  else if P is dominated only by a virtual point then
7:      Insert P into shadow_skyline list of N.
8:      N.updated_flag ← true
9:      Delete all points that are dominated by P from the shadow_skyline list
10: end if
11: return false
```

### Phase II: *Candidate* Skyline Insertion

Algorithm 3 gives the pseudo code of Phase II which aims to insert those *local* skyline points from Phase I into the *candidate_skyline* list. Basically, we compare $P$ against all *comparable* points in the *candidate_skyline list* (i.e., those points that have common complete dimensions with $P$). For each comparable point $Q$, we check if either $P$ or $Q$ dominates the other. If it is the case that $P$ dominates $Q$, we delete $Q$ from the *candidate_skyline* list and insert $P$ as a *virtual point* in the $Q$'s node (Line 5 in Algorithm 3). For the case where $Q$ dominates $P$, we just insert $Q$ as a *virtual point* in $P$'s node (Line 7 in Algorithm 3). Finally, if no point in the *candidate_skyline* list dominates $P$, we insert $P$ into the *candidate_skyline* list (Line 10 in Algorithm 3).

Inserting a *virtual point* $P$ into a node $N$ is mainly performed in three steps: (1) All *real* points in the *local_skyline* list of $N$ that are dominated by $P$ are moved to the *shadow_skyline* list of $N$. For example, consider Figure 2.2a; when we insert $Q_1$ as a *virtual point* in $P$, we find that $Q_1$ dominates both $P_1$ and $P_2$, thus, we move $P_1$ and $P_2$ to the *shadow_skyline* list as depicted in Figure 2.3. (2) All *virtual* points in the *local_skyline* list of $N$ that are dominated by $P$ and have complete dimensions that are

a superset of, or same as $P$'s complete dimensions are removed. The main idea is that if two *virtual points* $P_{iv}$ and $P_{jv}$ have the same bitmap and $P_{iv}$ dominates $P_{jv}$, then there is no need to store $P_{jv}$ as any point that will be dominated by $P_{jv}$ will also be dominated by $P_{iv}$. Similar arguments hold for the case of $P_{jv}$ having a superset bitmap of $P_{iv}$. (3) We insert a *virtual point* $P_v$ in the *local_skyline* list of $N$. $P_v$ is created by copying the values from $P$ for *only* the common dimensions of $P$ and $N$ bitmap while having *incomplete* in other dimensions.

**Algorithm 3** Phase II: *Candidate* Skyline Insertion

```
1:  Procedure Insert_Candidate_Skyline(Point P)
2:  for each point Q ∈ Candidate_Skyline where P and Q are comparable do
3:     if P dominates Q then
4:        Delete Q from Candidate_Skyline list
5:        Insert_Virtual_Point (P, Node N of Q )
6:     else if Q dominates P then
7:        Insert_Virtual_Point (Q, Node N of P )
8:     end if
9:  end for
10: if P is not dominated by any point, then Insert P in Candidate_Skyline list
```

### Phase III: *Global* Skyline Insertion

Algorithm 4 gives the pseudo code of Phase III in which we propagate qualified points from being *candidate* skylines to be *global* skylines. At the same time, we *validate* existing *global* skyline points against newly incoming points that were read since the last computation of the *global* skyline. The algorithm mainly has four steps: (1) Checking existing *candidate* and *global* points against each other for the dominance relation to remove any points that are dominated in any of these two lists (Lines 2 to 5 in Algorithm 4). The main idea of this step is to early prune those dominated points as there is no point in processing them further with the following expensive steps. (2) All remaining points in the *global_skyline* list are compared against all *shadow_skyline* lists of *comparable* but not equal nodes with a true *updated* flag. If at least one point in the compared *shadow_skyline* lists dominates a point $P$ in the *global_skyline* list, we immediately delete $P$ from the *global* skylines (Lines 6 to 10 in Algorithm 4). Notice the importance of the *updated* flag as an *optimization* technique that avoids comparing with *shadow_skyline*

**Algorithm 4** Phase III: *Global* Skyline Insertion

```
1: Procedure Update_global_Skyline()
2: for each pair of comparable points P ∈ Global_Skyline and Q ∈ Candidate_Skyline do
3:    if P dominates Q OR Q dominates P, then Mark the dominated point
4: end for
5: Delete all marked points from Candidate_Skyline and Global_Skyline lists
6: for each point P ∈ Global_Skyline do
7:    for each node N with comparable bitmap to P and a true updated flag do
8:       if any point in N shadow_skyline list dominates P, then delete P from the
         Global_Skyline list
9:    end for
10: end for
11: for each point Q ∈ Candidate_Skyline do
12:    for each node N with comparable bitmap to Q do
13:       if any point in N Shadow_Skyline list dominates Q, then delete Q from the
          Candidate_Skyline list
14:    end for
15: end for
16: Global_Skyline ← Global_Skyline ∪ Candidate_Skyline
17: set all updated flags to false
```

lists that did not change recently. Also, it is important to note that we do *not* need to compare *global* skyline points against the *local_skyline* list of comparable nodes as any *real* point in the *local_skyline* list is also stored in the *candidate_sklyine* list and hence it has been considered through the first step. (3) This step aims to process remaining points in the *candidate_skyline* list in the same way as points in the *global_skyline* list are processed in the second step with the *exception* that points in the *candidate_skyline* list are compared against *all* comparable nodes regardless of the status of the *updated* flag (Lines 11 to 15 in Algorithm 4). The main idea for ignoring the *updated* flag is that points in the *candidate_skyline* list have recently arrived, and thus, are not compared yet with points in the *shadow_skyline* lists. (4) Finally, the *global_skyline* list (i.e., the current query answer) is formed by combining all remaining *candidate* and *global* skylines together. Also, we reset all *updated* flags to *false* to indicate that the current answer is up to date. (Lines 16 to 17 in Algorithm 4). It is important to note that throughout Algorithm 4, deleting a point from either the *candidate* or the *global* lists indicates that

the point is stored in the *shadow_skyline* list of its corresponding node.

## 2.6 Proof of Correctness

This section proves the correctness of the *ISkyline* algorithm by proving that: (1) All *skyline* points are reported from the *ISkyline* algorithm, and (2) Any point returned from the *ISkyline* algorithm is a skyline over all input data.

**Theorem 1** *Any point P that is a skyline over all input data items, will be reported by the* ISkyline *algorithm*

**Proof 1** *Assume that there exist a point P that is a skyline over all input data items, however, P is not reported by the* ISkyline *algorithm. Throughout the* ISkyline *algorithm, a point is discarded only if it is dominated by either a* real *or a* virtual *point. Thus, we have two cases: (1)* ***Case 1:*** *P* is dominated by a real point. *Since P is already a skyline among all data points, then, by the definition of skyline, there cannot be any real point that dominates P. So, this case cannot take place. (2)* ***Case 2:*** *P* is dominated by a virtual point. *For a virtual point $Q_v$ to dominate P, the original real point of $Q_v$ (i.e., Q) should also dominate P. This comes from the definition of a virtual point that the common dimensions between $Q_v$ and P are the same as those between Q and P. As no real point Q can dominate the skyline point P, then this case cannot take place. From Cases 1 and 2, we conclude that the assumption that P is not reported by the* ISkyline *algorithm is not possible. Thus,* ISkyline *reports all existing skylines.*

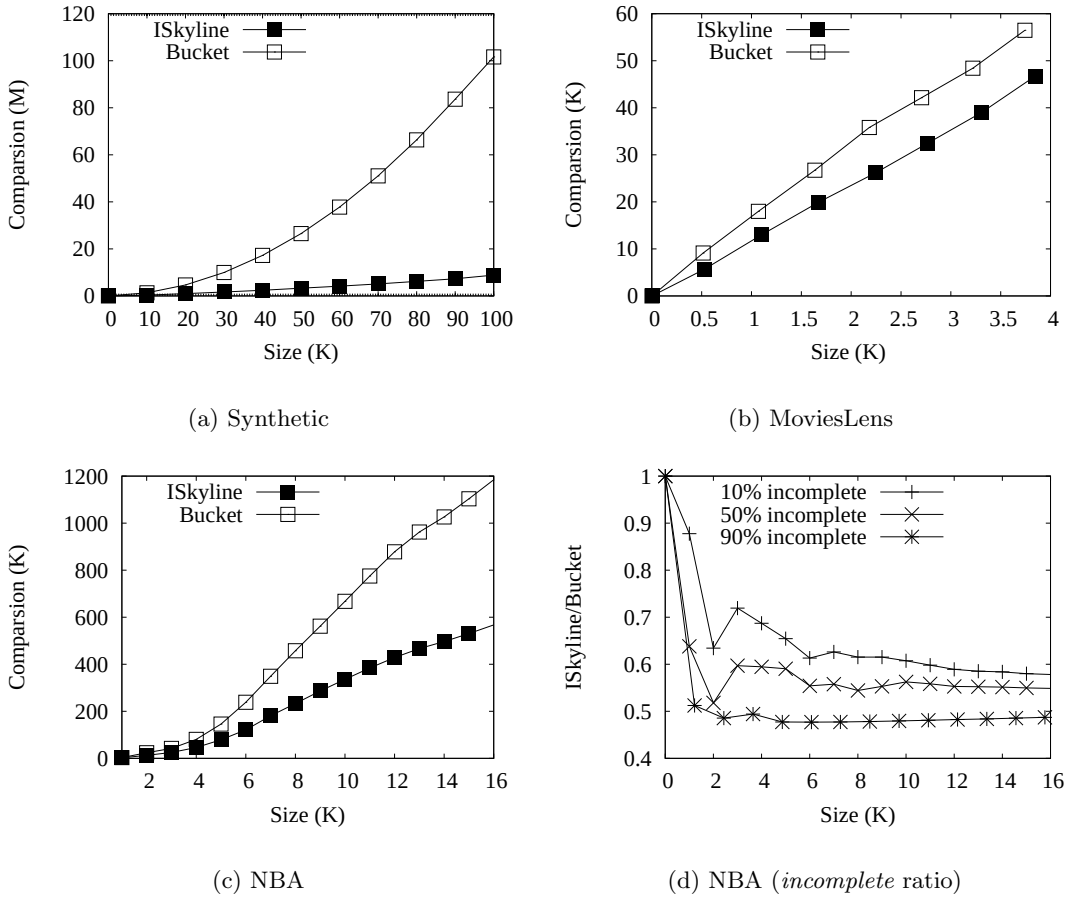
**Theorem 2** *Any point P returned from the* ISkyline *algorithm is a skyline over all input data items.*

**Proof 2** *Assume that there exists a point P that is reported from the* ISkyline *algorithm, however, there exist another real point Q in the input data set that dominates P, i.e., P is not a true skyline. As point P is reported as a result, it is stored in the* global_skyline *list. On the other hand, point Q may be in one of five cases: (1)* ***Case 1:*** *Q is stored in* candidate_skyline. *As depicted in Line 3 Algorithm 4, all points in the* candidate *list are compared against all points in the* global *list. Then, the dominated points will be deleted from both lists. This means that if Q dominates P, then P will be*

*removed from the* global_skyline*, and hence would not be reported by the algorithm. So, this case cannot take place. (2)* ***Case 2:*** *Q is stored in the* global_skyline*. As depicted by Line 16 Algorithm 4, to be stored in* global_skyline*, Q has to go through the* candidate_skyline *first. This means that it should have been compared against P as in Case 1. Since, Case 1 cannot take place, then this case also cannot take place. (3)* ***Case 3:*** *Q is stored in* local_skyline*. By the definition of* local_skyline*, any* real *point that is stored in a* local_skyline *will be stored also in the* candidate_skyline *list. This means that Q is also in the* candidate_skyline *list and hence compared to P as in Case 1. Since, Case 1 cannot take place, then this case also cannot take place. (4)* ***Case 4:*** *Q is stored in* shadow_skyline*. As depicted in Lines 7 to 8 Algorithm 4, point P will be compared against all points in all comparable recently changed* shadow_sklyines*. If P was dominated by any point, it will be removed from the* global_skyline *list. For comparable* shadow_skyline *lists that are not recently updated, P will be compared with them before being a* global_skyline *as in Lines 12 to 13 Algorithm 4. Since P is already reported, then no point in a* shadow_skyline *list has dominated it. So, Case 4 cannot take place. (5)* ***Case 5:*** *Q is discarded. This means that there exist a point R in either the* local_skyline *or* shadow_skyline *lists of the node that corresponds to Q where R dominates Q. So, this case boils down to either Case 3 or Case 4. Since both cases cannot take place, Case 5 also cannot take place. From Cases 1 to 5, we conclude that the assumption that there exists point Q that dominates P is not possible. So, all points reported by the* ISkyline *algorithm are skylines.*

## 2.7 Experimental Results

This section experimentally evaluates the performance of the proposed algorithms. As this is the first attempt for skyline query processing in *incomplete* data, we could not compare with any previous technique. Also, initial experiments show that our proposed *Replacement* algorithm performs severely worse than our proposed *Bucket* and *ISkyline* algorithms. This is mainly due to the fact that the skyline set $S_{-\infty}$ in the *Replacement* algorithm is of a very large size. So, in this section, we focus only in the performance analysis and comparison of both the *Bucket* and *ISkyline* algorithms. Our test bed includes three data sets: (1) MovieLens [46]. This is a real data set of 3900 points




(a) Synthetic

(b) MoviesLens

(c) NBA

(d) NBA (*incomplete* ratio)

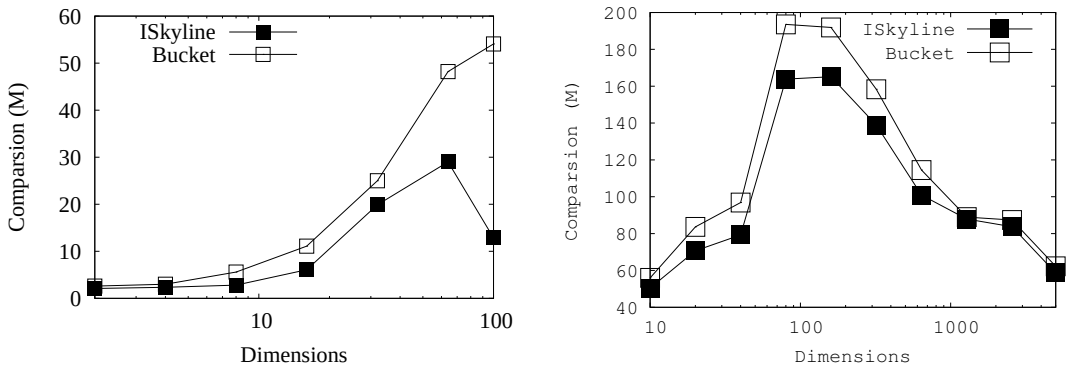
Figure 2.6: Scalability

where each point is of 6000 dimensions that represent the user ratings (6000 users) of 3900 movies. There is only about 1 Million reviews, which means that this data set is 95% *incomplete*, i.e., only 5% of the ratings are available. (2) NBA [68]. This is a real data set containing records for 16,000 NBA players where each record has 17 dimensions representing various statistics about basketball skills. The NBA data is rather complete, however, we explicitly remove values in order to test the performance of our algorithms. Removed values represent missing statistics about the players for some years. Unless mentioned otherwise, the default incomplete percentage for the NBA dataset is 20%. (3) Synthetic. We generated a 20% incomplete synthetic data set of 100,000 points, each with 100 dimensions.

Our first set of experiments (not shown for space limitation) suggest to set parameter $t$ in the *ISkyline* algorithm to be 20, 100, 200 for NBA. Synthetic, and MovieLens data, respectively. Unless mentioned otherwise, our performance metric is the number of comparisons for each algorithm.

### 2.7.1 Scalability

Figures 2.6a, 6b, and 6c give the scalability of *ISkyline* for Synthetic, MovieLens, and NBA datasets, respectively. It is clear that in all cases the *ISkyline* algorithm is superior to the *Bucket* algorithm. In general, the difference in cost between *ISkyline* and *Bucket* comes from the fact that *ISkyline* exploits the *virtual points* and *shadow skylines* to minimize the number of *local* skylines at each bucket. For Synthetic data (Figure 2.6a), *ISkyline* performs only 10% of the comparisons needed by *Bucket*. The main idea is that with only 20% incompleteness, we end up having large numbers of *comparable* buckets as the bitmap of each bucket is highly likely to have many 1's. Thus, *ISkyline* is able to find room in which the concepts of *virtual points* and *shadow skylines* can be exploited. Notice that the number of buckets, and hence, the number of *local* skylines increases with the increase of data size. Thus, the performance ratio of *ISkyline* over *Bucket* increases. For MovieLens data (Figure 2.6b), although *ISkyline* steadily outperforms the *Bucket* algorithm, however, the performance ratio is not as strong as the case of Synthetic data. The main reason is that MovieLens data has 6000 dimensions, which means that the 1 Million entries have been distributed over large number of buckets where each bucket has very few entries (e.g., a bucket would have two entries *only* if two movies have been rated by the exact set of reviewers). So, *virtual points* and *shadow_skyline* may not take place in all buckets. So, the difference in performance between *ISkyline* and *Bucket* in MovieLens indicates the number of buckets that get benefit from *virtual points* and *shadow_skyline*. For NBA data (Figure 2.6c), similar to other data sets, *ISkyline* steadily outperforms *Bucket*. Notice that in NBA data set, the number of *comparable* buckets would be between the Synthetic and the MovieLens data. So, the superiority of *ISkyline* over *Bucket* is more than the case of MovieLens but less than the case of Synthetic.




(a) Synthetic dataset

(b) Movies

Figure 2.7: Data Dimensionality

### 2.7.2 Ratio of Completeness.

Figure 2.6d gives the effect of the ratio of *incomplete* entries on the performance of *ISkyline* over *Bucket*. We plot the number of comparisons incurred by *ISkyline* as a ratio from that of *Bucket*. We also plot three entries of *ISkyline* with *incomplete* ratios of 10%, 50%, and 90%. It is clear that with the increase of the ratio of *incomplete* data, *ISkyline* would be a better enhancement over *bucket*, i.e., the ratio of the number of comparisons is decreased. The main reason for this is that with more *incomplete* data, *virtual points* can reduce the number of *local* skylines at each bucket and the overhead needed to update the list of *global* skylines. Such role of *virtual points* becomes more clear with the increase of the *incompleteness* ratio. This experiment reflects the fact that *ISkyline* is designed specifically with the *incompleteness* problem in mind while *Bucket* uses an adaptation of existing skyline algorithms to accommodate *incomplete* data. It is important to note that the performance ratio between *ISkyline* and *Bucket* is stable with the increase of data size.

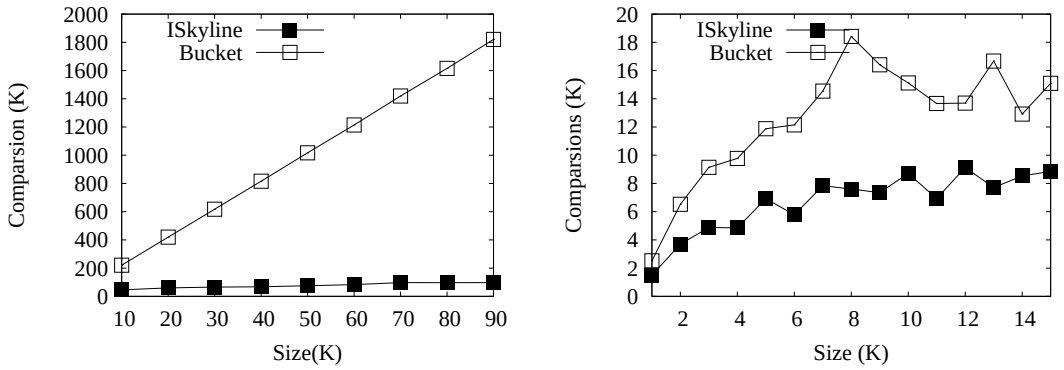
### 2.7.3 Data Dimensionality

Figure 2.7 gives the effect of increasing the dimensionality (represented by a log scale) on the performance of both *ISkyline* and *Bucket* for Synthetic and MovieLens datasets. As

in previous experiments, *ISkyline* steadily outperforms *Bucket* for up to 100 dimensions in the Synthetic data and 5000 dimensions in MovieLens data. In both data sets, the number of required comparisons by *ISkyline* rises up for medium dimensions (i.e., 100-500 dimensions) and then goes down for higher dimensions. The main reason is that the performance depends mainly on the *comparability* of data items. If most data items are comparable with each other, the performance will be worse. With few dimensions, high ratio of the *incomplete* data will be removed from the input as a data item may include only *incomplete* dimensions. Thus, the number of *comparable* of points would be less. With the increase of dimensions, the *comparability* ratio increases till we reach to a peak point. Then, with the increase of dimensions, the number of possible buckets would increase, and hence the data items would have different buckets with different bitmaps, reducing the comparability ratio.

### 2.7.4 Incremental behavior

Figure 2.8 gives the *incremental* behavior of both *ISkyline* and *Bucket*. We measure the number of comparisons needed to "refresh" the query answer after adding 1,000 new data items for both synthetic and NBA data. Due to its *incremental* properties, managed by the *updated* flag, *ISkyline* clearly outperforms *Bucket* in all cases. This indicates that *ISkyline* smartly avoids reevaluation and redundant processing that are done by *Bucket* to maintain the current answer of *global* skylines up to date. It is important to note also that for large data sizes, e.g., more than 50K in Figure 2.8a, adding 1K of data by the *ISkyline* would have the same cost regardless of the current data size, while in *Bucket*, the cost will be increased linearly with the increase of data size. This is mainly due to the fact that the metadata stored as *virtual points*, *shadow skylines*, and *updated* flag aid *ISkyline* to focus only on the new 1K additions of data rather than reconsidering all data as in the case of *Bucket*.




(a) Synthetic dataset

(b) NBA

Figure 2.8: Incremental behavior

## 2.8 Conclusion

This chapter has addressed the problem of skyline queries over *incomplete* data where multi-dimensional data items are missing some values of their dimensions. We showed that with *incomplete* data, the dominance relation among data points may not be transitive, thus, almost all existing techniques for skyline queries are not applicable. We have proposed two new algorithms, namely, the *Replacement* and the *Bucket* algorithms that utilize variations of traditional skyline algorithms to accommodate *incomplete* data. Then, we proposed the *ISkyline* algorithm that is designed specifically for *incomplete* data. The *ISKyline* algorithm employs two optimization techniques, namely *virtual points* and *shadow skylines* to exploit the properties of *incomplete*. The correctness of the *ISkyline* is proved in terms that produce only and all skyline points. Experimental results based on real and synthetic data sets show the efficiency and scalability of the *ISkyline* algorithm.

# Chapter 3

# UPref: Preference Query Processing for Uncertain Data

The previous chapter covers the case where some dimensions are incomplete, on the other hand, in this chapter, we addresses the problem of efficiently processing preference queries (e.g., skyline, multi-objective, top-$k$) for uncertain data represented as a range. In this setting, query answers are probabilistic, where each object is associated with a probability value of being a preferred answer. We present UPref, an efficient and flexible framework to answer preference queries over uncertain data. UPref users can specify a probability threshold that each object must exceed in order to be considered a preference answer, and a tolerance value that defines the allowed error margin in probability calculation. Other research efforts use threshold to reduce necessary computation over uncertain data (e.g.,[69]). UPref employs an efficient two-phase query processing algorithm. The first phase compares data objects pairwise, and uses a novel method that immediately *filters* objects that are guaranteed not to be preferred answers (i.e., have probabilities *below* a given threshold). The second phase refines the probability of each object not filtered in the first phase by implementing an efficient stepwise method to calculate an object's final probability within a given tolerance value. UPref is a *flexible*, capable of answering skyline, multi-objective, and top-$k$ preference queries all within a *single* framework. Extensive experiments show that UPref exhibits orders of magnitude improvement over other methods to process preference queries over uncertain data.

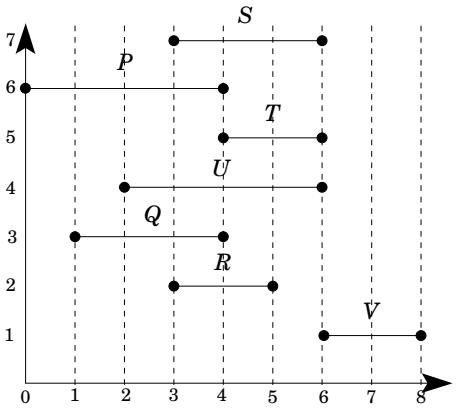
## 3.1 Introduction

Preference queries give users the ability to receive query answers that are *relevant* to their preferences. The concept of preferences in database systems has become very important as it helps realize useful, non-trivial applications, ranging from multi-criteria decision-making tools to *personalized* databases [14, 15]. The main idea of preference queries is to find a *set of customized answers* among a large set of multi-dimensional objects that a user will like the most. An example of preference queries is *"find my preferred restaurants based on price and distance"*. Finding the *preference* answer mainly depends on the type of the preference method employed. For example, if the preference query is a *skyline* query [2], then the preferred set of restaurants are those that are not dominated by other restaurants. A restaurant $x$ dominates restaurant $y$ if $x$ is better than $y$ in one of the dimensions (price or distance) while it is equal or better in the other dimension. Conversely, if we employ the *top-k* [1] method to answer the preference query, then the answer is the $k$ restaurants with the highest score using a function that combines price and distance together. In addition, there are numerous other preference methods we can employ, and each will return a set $\mathcal{P}$ of preferred restaurants, e.g., *multi-objective* [5], *k-dominance* [6], and *top-k dominating* [12]. However, each method differs in the way it determines the contents of $\mathcal{P}$.

With the growing number of applications that generate uncertain data, e.g., sensor readings, human reading errors, and data imperfection, it has become essential to support preference queries of various types over uncertain data. Recent research in preference query processing for uncertain data can be classified to two categories based on the uncertainty model: (a) *Discrete*, in which uncertain data is represented as a set of discrete values, and (b) *Continuous*, where uncertain data is represented as a continuous range of values. Unfortunately, existing work for preference queries for the discrete uncertainty model (i.e., [29] for *skyline* queries and [30, 31, 32] for *top-k* queries) cannot be easily extended to support a host of real-life applications that use a *continuous range* for uncertain values (e.g., biological data, spatial databases, sensor monitoring, and location-based services). On the other hand, existing preference methods that handle the continuous uncertainty model are limited only to *nearest-neighbor* queries (i.e., *top-1*) queries [27, 28], with no obvious extension to handle the myriad other preference

methods (e.g, skyline [2], multi-objective [5]).

In this chapter, we propose a framework for performing preference query evaluation over uncertain data represented as a continuous range. While our approach is novel in its scope, it has the added benefit of being able to support many well-known preference methods within a *single* framework. Our framework aims to find the probability that each object is in the list $\mathcal{P}$ of preferred answers. Given that preference answers are probabilistic in nature, we introduce two parameters that govern the size of the answer and tune query processing performance: a user-defined threshold $H$ that defines the lowest probability an object can have to be added to $\mathcal{P}$ and a tolerance parameter $\delta$ that defines the precision for calculating an object's probability of being a preference answer.




(a) Running Example

| Point | Skyline | Top–2 |
|---|---|---|
| $P$ | 60% | 48% |
| $Q$ | 91% | 86% |
| $R$ | 100% | 47% |
| $S$ | 0.5% | 0% |
| $T$ | 0% | 0% |
| $U$ | 16% | 18% |
| $V$ | 100% | 0% |

(b) Probabilities

Figure 3.1: Example

As an example, consider the seven uncertain objects in Figure 3.1a where the $x$ dimension of each object is presented as a continuous range while the $y$ dimension is an exact value. Figure 3.1b gives the probability that each uncertain object is in $\mathcal{P}$ for the case that $\mathcal{P}$ is computed using both the *skyline* and the *top-k* method ($k$=2), where the latter method uses the ranking function $f = 2x + y$. With a threshold $H = 50\%$, the query answer is $\{P, Q, R, V\}$ and $\{Q\}$ for the skyline and top-$k$ method, respectively. Notice that we return only *one* answer for the case of the *top-k* query as this is the only object that has more than a 50% probability to be a top-2 object. To the authors' knowledge, the problem of preference queries over uncertain data represented as

a continuous range was not studied before beyond *top-1* queries (i.e., *nearest-neighbors*) in [27, 28, 70]. Furthermore, no previous work has studied an approach to preference query processing for uncertain data capable of answering numerous preference query types within a *single* framework.

Computing a preference answer $\mathcal{P}$ to an exact probability precision is exponential in nature [71], therefore we present four methods that bound the probability of each object being in $\mathcal{P}$: *uncertainty reduction*, *pairwise comparison*, *segmentation*, and *bound tightening*. We also present a general two-phase algorithmic framework that efficiently integrates these four techniques to efficiently answer preference queries. The first phase is responsible for applying the *uncertainty reduction* and *pairwise comparison* methods to all objects in the data set while preparing relevant data structures to be used in the second phase. The second phase is responsible for *segmentation* and *bound tightening*. The framework follows a filter-refine approach where the first phase prunes a set of objects from consideration before the more expensive second phase begins. Also, within the second phase, we continuously prune objects from the data set in a way that avoids the expensive exact probability calculations. For clarity and ease of explanation, we will discuss the details of our framework in the context of *skyline* queries. We then discuss how the framework answers both top-$k$ [1] and multi-objective [5] queries. In general, the contributions can be summarized as follow:

- We define preference query answers over uncertain data represented as a continuous range as the set of objects having a probability more than $H$ to be a preferred object. We also calculate the probability of these objects with an accuracy $\delta$.
- We present four methods for bounding the object probability to be in the preferred set $\mathcal{P}$: *uncertainty reduction*, *pairwise comparison*, *segmentation*, and *bound tightening*.
- We present a two-phase framework that integrates our proposed bounding methods, and show how this single framework can support skyline, top-k, and multi-objective preference methods.
- We provide experimental evidence that our proposed framework is scalable and efficient; showing orders of magnitude improvement over the existing algorithms

extended to handle uncertain data.

The rest of the chapter is organized as follows: Section 3.2 highlights related work. Section 3.3 presents the problem formulation. Our proposed framework is presented in Section 3.4, with primary focus given to the skyline method. Section 3.6.1 discusses how our framework supports both *top-k* and *multi-objective* queries. Experimental results are presented in Section 3.8 while Section 3.9 concludes this chapter.

## 3.2 Related Work

There is a plethora of work to support efficient evaluation of various preference functions for certain data. Examples of these preferences functions include *skyline* [6, 42, 43, 44, 45, 11], *top-k* [32, 27, 72, 73, 74, 75], *multi-objective* [5, 76], *k-dominance* [6], *nearest-neighbors* [77, 78, 79, 80, 81], *reverse nearest neighbors* [82, 83, 84, 85, 86, 87, 88], and *top-k dominating* [12]. On the other side, research efforts in preference queries over uncertain data is limited to *skyline* and *top-k* queries and can be classified to two categories based on the underlying uncertainty model:

**Discrete uncertainty model**. In this model, each uncertain object is represented as a finite set of instances, i.e., an object $P$ may only have $n$ instances, $a_1$, $a_2$, $\cdots$, $a_n$. Then, the probability of an object to be in the preference set $\mathcal{P}$ is calculated as the sum of the probabilities of each instance $a_i$ to be in $\mathcal{P}$. Most of the work in this category has focused on *top-k* queries [31, 32, 30] with the exception of [29] that supports *skyline* queries.

**Continuous uncertainty model.** In this model, each uncertain object is represented as a range of values, i.e., an object $P$ may be anywhere within a range $[a_l - a_u]$. Due to its challenging nature, existing work for preference queries for this uncertainty model is limited to the case of *nearest-neighbor* queries, i.e., *top-1* [27, 28, 70]. This problem is first coined in [70] to find the probability of each object to be a *nearest-neighbor* to the user point. An extension was later proposed in [28] to find only those objects that above a certain threshold to be a *nearest-neighbor*. Finally, [27] extends this problem to find the $k$ objects that have the highest probabilities to be a nearest-neighbor.

Our work lies in the second category where we aim to find those objects that have a probability more than a certain threshold $H$ to be in the preferred set $\mathcal{P}$, where their

probabilities are computed within a precision $\delta$. Up to the authors' knowledge, this work is the first attempt to discuss preference queries for continuous uncertainty model as previous work in this model did not go beyond the case of *nearest-neighbor*, i.e., *top-1* queries [27, 28, 70].

**CareDB**. UPref is part of CareDB, a context and preference-aware database system [89]. CareDB [90] features a generic and extensible query processor capable of embedding myriad preference methods inside a database system (CareDB is implemented in PostgreSQL). The FlexPref framework [34] within CareDB supports preference query processing over standard relational data. CareDB has also been extended in a generic fashion to efficiently support join queries [91] and access to attributes that are expensive to derive at query runtime [92]. UPref is the fourth integral piece of CareDB that provides generic preference query processing support for uncertain data.

## 3.3 Preference Methods and Problem Formulation

This section first describes the three preference methods UPref is capable of handling, namely, skyline, multi-objective, and top-$k$. We then provide a formal definition of preference queries over uncertain data.

The preference method our framework supports are: (1) *Skyline*. Given a dataset $D$, a skyline query computes the Pareto-optimal set $S$ of $D$. $S$ are those objects that are *not dominated* by any other objects. An $n$-dimensional object $x$ is said to dominate another object $y$ if $x$ is better than or equal to $y$ in $n$ dimensions, and *strictly* better than $y$ in at least one dimension. Without loss of generality, we assume less is better. (2) *Multi-objective*. Given a set of $n$-dimensional objects, multi-objective evaluation combines $m$ dimensions ($1 < m < n$) using a monotonic ranking function $\mathcal{F}$ (e.g., sum) that returns a single real value. A skyline is then computed over the resulting $(n-m)+1$ independent dimensions. (3) *Top-k*. This method combines *all* of an object's attributes using a monotonic ranking function $\mathcal{F}$, essentially scoring each object by a real value. The preference answer consists of the $k$ objects with the best (e.g., lowest, highest) scores.

**Definition 7 Preference Queries for uncertain data**. *Given (1) a dataset $D$ of $n$-dimensional objects with uncertain dimensions presented as continuous range, (2) preference criteria $\mathcal{C}$ for any one of the skyline, multi-objective, or top-k methods, (3) a threshold value $H$, and (4) a tolerance value $\delta$, find the answer set $\mathcal{A}$ that includes all objects in $D$ that have a probability more than $H$ to satisfy the criterion $\mathcal{C}$; the probability value for each object in $\mathcal{A}$ is guaranteed to have a calculated error bound less than or equal to $\delta$. Throughout the rest of this chapter, we refer to the probability that an object satisfies $\mathcal{C}$ as the "preference probability".*

Without loss of generality and for ease of description, we assume that each object has only one uncertain dimension, namely the first, i.e., each object in the dataset $D$ has the form ($[d_l$-$d_u]$, $d_2$, ..., $d_n$) where $[d_l$-$d_u]$ represents a continuous uncertainty range of the first dimension. We also assume that an object has a *uniform* probability distribution of being anywhere in its uncertainty range. However, our techniques are not constrained by these assumptions, and can easily accommodate multiple uncertain dimensions and arbitrary probability distribution with only adding the cost of more complex probability calculations. In our computations, we assume that, for all dimensions, minimum values are better. Finally, in our algorithms, we do not assume the existence of any index structure nor the preprocessing of input data.

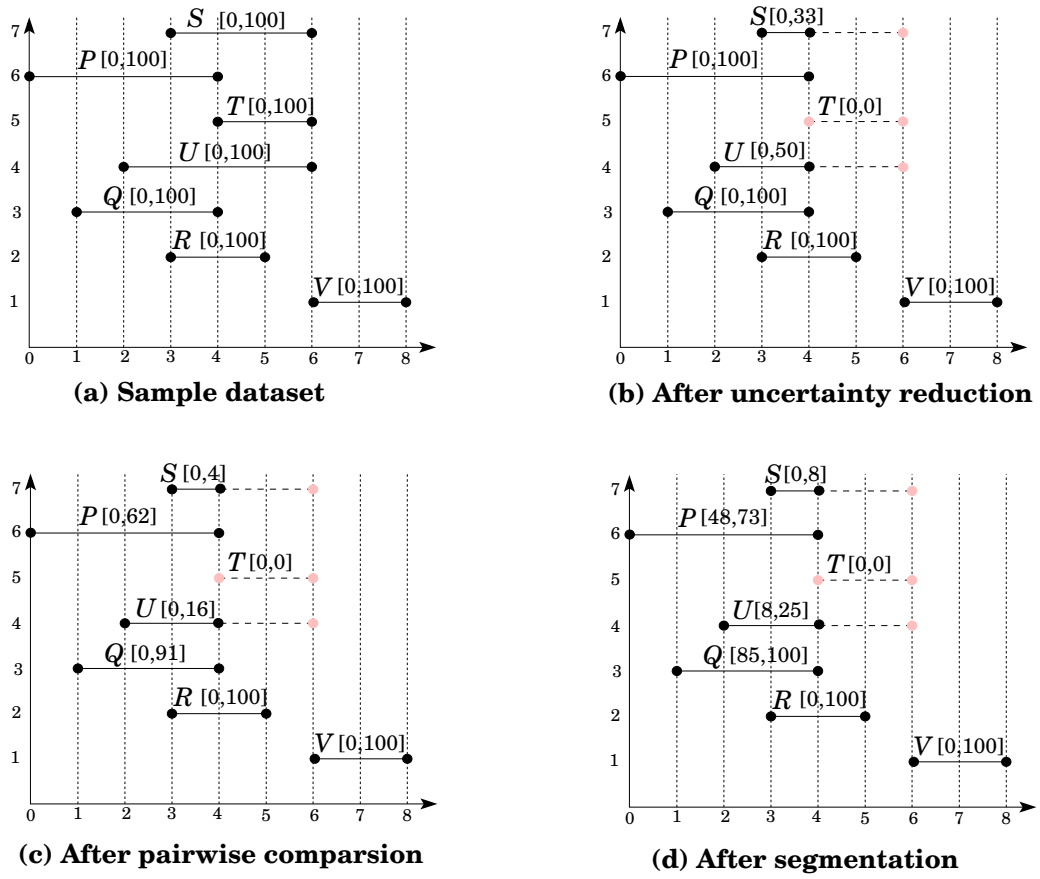



Figure 3.2: Bounding Objects Probabilities (Skyline Example)

## 3.4 Bounding Object Probability

Computing the *exact* preference probability for each object in the dataset is prohibitively expensive[71]. In this section, we present four methods that efficiently and gradually bound the preference probability of each object. These bounds are used to prune objects early in query processing that are not of interest, i.e., those objects that are found to have a preference probability less than $H$. Later, in Section 3.5, we describe how UPref uses these techniques in the complete query processing framework.

The four methods we propose are: (1) *Uncertainty reduction* (Section 3.4.1) that aims to place an upper bound on the preference probability while reducing the object uncertainty region, (2) *Pairwise comparison* (Section 3.4.2) that tightens the upper

bound, (3) *Segmentation* (Section 3.4.3) that computes a lower bound while continuing to tighten the upper bound, and (4) *Bound tightening* (Section 3.4.4) that iteratively tightens the lower and upper bound probabilities of an object to be within the required tolerance $\delta$.

Figure 3.2 and 3.3 provides a running example throughout this section for the skyline and top-2 preference methods, respectively. For the top-2 example, all objects are ranked using a single uncertain dimension. The data provided in both figures are seven two-dimensional uncertain objects; the horizontal axis represents the uncertain values as a continuous range while the vertical axis represents a certain value. For each object, we associate an initial lower and upper bound probability to be a preferred object as 0% and 100%, respectively. We do not provide a running example for the multi-objective preference method [5], as this would be similar to the skyline example augmented with a pre-processing step to aggregate multiple dimensions into one using a monotonic function $\mathcal{F}$.

### 3.4.1 Uncertainty Reduction

**Objective.** Uncertainty reduction employs simple probability calculations on a single object $P$ to achieve two main objectives: (a) Setting an upper bound preference probability for $P$; if this bound is below $H$, then there is no need to consider $P$ in any further computations, and (b) Reducing the uncertainty range of $P$, this is particularly important as it reduces the chance that $P$ is involved in further complex computations.
**Affected Objects.** Uncertainty reduction is computed in a *pairwise* fashion. For skyline and multi-objective queries, an ordered pair of objects ($Q$,$P$) qualifies for *uncertainty reduction* only if the *endpoint* of $Q$ dominates the *endpoint* of $P$. The endpoint $e_P$ of an object $P = ([d_l\text{-}d_u], d_2, \cdots, d_m)$ is formulated by substituting $P$'s uncertainty range by a single point that represents the end of its uncertainty range, formally $e_P = (d_u, d_2, \cdots, d_m)$. By comparing endpoints, it is straightforward to determine the dominance relation between $e_P$ and $e_Q$. For top-$k$ queries, the qualification condition is slightly more restrictive. An object pair ($P$,$Q$) qualifies for uncertainty reduction if: (a) the endpoint of $Q$ is less than that of $P$, and (b) $Q$ has the $k^{th}$ endpoint among all objects considered so far. The main idea is that that any object that starts after $Q$ endpoint has no chance to be a preferred object, and because $Q$ has the $k^{th}$ endpoint,

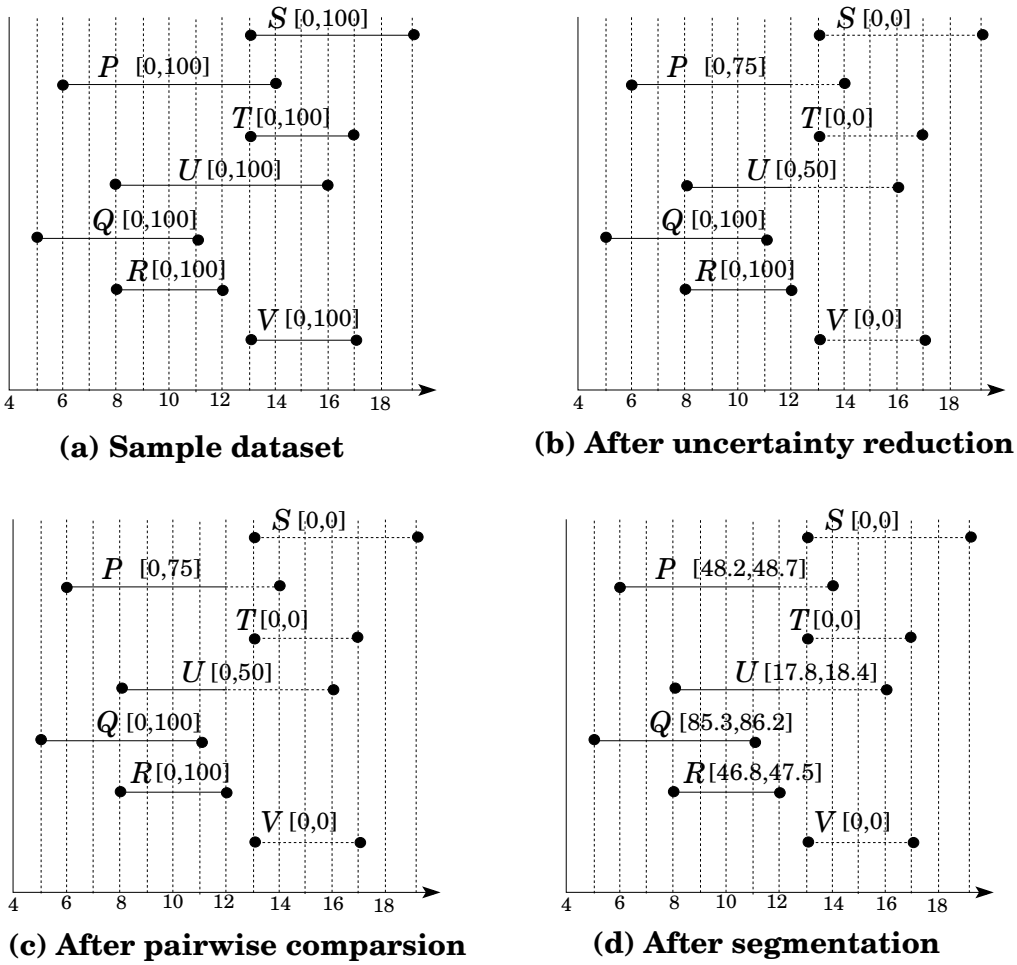



Figure 3.3: Bounding Objects Probabilities (Top-2 Example)

it has the privilege to reduce the uncertainty range of other points.

As an example, in Figure 3.2a, the pair ($Q$,$U$) qualify for *uncertainty reduction* for a skyline query as $e_Q$=4 dominates $e_U = 6$. Meanwhile, neither object pair ($Q$,$R$) nor ($R$,$Q$) in Figure 3.2a qualify for *uncertainty reduction* as neither $e_Q$ nor $e_R$ dominate each other. In Figure 3.3a, assuming object endpoint $e_R$ of object $R$ is the top-2 endpoint seen so far, then object pair ($U$,$R$) qualify for *uncertainty reduction*. Meanwhile pair ($R$,$V$) do not qualify for *uncertainty reduction*, as the uncertainty range of $V$ starts after the endpoint of $R$.

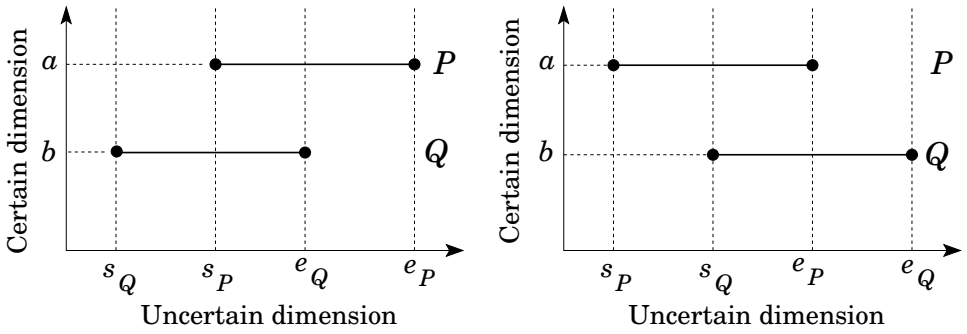
**Main Idea.** The main idea of *uncertainty reduction* is to reduce the upper bound

preference probability for an object $P$ by removing a portion of its uncertainty range that gives $P$ a zero probability of being a preference answer. For example, consider the pair ($Q$,$U$) in Figure 3.2a; since $e_Q$ dominates $e_U$, we can see that if the exact value of $U$ in its uncertainty range lies in the portion [$e_Q$-$e_U$], then $U$ will be completely dominated by $Q$, i.e., $U$ would have no chance of being a preferred answer. Therefore, the uncertainty range [$s_U$-$e_U$] for object $U$ can be reduced by removing the segment [$e_Q$-$e_U$]. Thus, $U$'s new uncertainty range would be [$s_P$-$e_Q$] which still gives $U$ the chance to be a preferred object. Finally, by removing this portion, the upper bound probability of $U$ can be reduced by the ratio proportionate to [$e_Q$-$e_U$]. *Uncertainty reduction* still guarantees the answer correctness as removing a portion from the object uncertainty region does not affect its final preference probability, nor does it affect the preference probability of other objects in the dataset. Similarly, Figure 3.3a, the starting point of object $S$ is greater than the second endpoint (i.e., $e_R$). Hence, the object $S$ has no chance to be in the top-2 set.

**Skyline Example** As an example for the skyline query, we apply the *uncertainty reduction* over all the uncertain objects in Figure 3.2a starting from $V$ with the lowest certain value. For object $V$, as the endpoint of the uncertainty range $e_V$ does not dominate any corresponding endpoint, $V$ does not result in any uncertainly reduction for any other object. For object $R$, $e_R$ dominates $e_U$, $e_T$, and $e_S$, so, the pairs ($R$,$U$), ($R$,$T$), and ($R$,$S$) qualify for *uncertainty reduction*. This results in reducing the uncertainty range of $U$ to be [2-5] instead of [2-6]. Since the reduced range is one quarter of the original range, the upper bound probability of $U$ is set to 75%. Similarly, the uncertainty ranges of $T$ and $S$ are reduced to [4-5] and [3-5] with an upper bound probability of 50% and 66%, respectively. Then, considering object $Q$, $e_Q$ dominates $e_U$, $e_T$, $e_S$. So, the uncertainty ranges of $U$, $T$, and $S$ are reduced to be [2-4] , [4-4] and [3-4] with an upper bound probability of 50%, 0%, and 33%, respectively. Finally, $e_U$, $e_T$, $e_P$, and $e_S$ do not dominate any endpoint, thus, they do not affect the probability of other objects. Notice that $e_P$ does not dominate $e_S$ as $S$ uncertainty range has been reduced to be [3-4]. Figure 3.2b gives the result of all points after the uncertainty reduction and the upper probability bound. If the probability threshold $H$ is 50%, then *uncertainty reduction* would result in pruning the objects $S$, $T$, and $U$ from any further processing.

**Top-2 Example** Figure 3.3a gives an example for the Top-2 query. The second end

point is $e_R$. Any object starts after the endpoint $e_R$ has a probability of 0% to be top-2 object. These objects are $V$, $T$ and $S$. For the object $P$, the endpoint is greater than endpoint of $R$. The upper bound probability of $P$ is caluculated as 100% - $\frac{e_P - e_R}{e_P - s_P}$ = 100% - 25% = 75%. Similary, for object $U$, the upper bound probability is reduced to be 50%. When the probability threshold $H$ is 50%, then *uncertainty reduction* would result in pruning the objects $S$, $T$, and $V$ from any further processing.




(a) Uncertainty reduction & pairwise comparison

(b) Pairwise comparison

Figure 3.4: Example

### 3.4.2 Pairwise Comparison

**Objective.** Pairwise comparison aims to calculate an upper bound probability for two objects $P$ and $Q$ to be in the preferred set. As the probability bound is set for two objects, *pairwise comparison* employs more expensive computations than that of the *uncertainty reduction* that calculates a probability bound for only one object.

**Affected Objects.** Similar to the case of *uncertainty reduction*, *pairwise comparison* is computed in a *pairwise* fashion. However, the condition that two objects qualify for pairwise comparison is differs.

For skyline and multi-objective queries, an ordered pair of objects ($Q$,$P$) qualifies for *pairwise comparison* only if: (a) the uncertainty ranges of $P$ and $Q$ overlap, and (b) the certain part of $Q$ either is equal to or dominates the certain part of $P$. The certain part of object $P$=$\{[d_l - d_u], d_2, \ldots, d_n\}$, $C(P)$, is formulated by removing P's uncertain dimension, formally, $C(P)$=$\{d_2, \ldots, d_n\}$.

As the qualifying condition includes equality, it could be the case that both ordered pairs ($P$,$Q$) and ($Q$,$P$) qualify for *pairwise comparison*. This case takes place if $C(P)$

$= C(Q)$. In both Figures 3.4a and 3.4b, only the pair ($Q$,$P$) qualifies for *pairwise comparison*. Similar to the case of *uncertainty reduction*, pairwise comparison calculations for a qualified ordered pair ($Q$,$P$) in *top-k* queries is the same as that of *skyline* queries. However, they are different in the qualification conditions where ($Q$,$P$) qualifies for *top-k* pairwise comparison only if: (a) either the endpoint of $Q$ or $P$ is the $k^{th}$ endpoint among all considered points so far, (b) there exist $k$-1 endpoints before the start point of $P$ or $Q$, and (c) the uncertainty ranges of $Q$ and $P$ overlap.

**Main Idea.** The main idea of *pairwise comparison* is that for a qualified ordered pair of objects ($Q$,$P$), since $Q$ already dominates or equal $P$ in the certain dimensions, if $Q$ would also dominate $P$ in the uncertain dimension, then $P$ would have no chance to be in the preferred set. This means that the only chance for $P$ to still be a preferred object is to be *not* dominated by $Q$ in the uncertain dimension. Thus, we can have an upper bound probability for object $P$ to be in the preferred set as the probability that the uncertain range of $P$ is not dominated by the uncertain range of $Q$. For a qualified pair of objects ($Q$,$P$) with overlapping uncertainty ranges [$s_P$-$e_P$] and [$s_Q$-$e_Q$], respectively, *pairwise comparison* distinguishes between four cases:

- *Case 1:* $s_Q < s_P$ and $e_Q < e_P$. This is the case depicted in Figure 3.4a. In this case, the probability that the uncertainly range of $P$ is not dominated by that of $Q$, i.e., the upper bound probability of object $P$ to be a preferred object, is half the probability that both objects $P$ and $Q$ lie in the overlapped part of the uncertainty region, [$s_P$-$e_Q$]. Formally, $P_{upper} = \frac{1}{2}(Pr\{P \in [s_P\text{-}e_Q]\} * Pr\{Q \in [s_P\text{-}e_Q]\})$.

- *Case 2:* $s_Q < s_P$ and $e_P < e_Q$. This is the case where the uncertainty range of $P$ is a subset of that of $Q$. In this case, the probability that $P$ is not dominated by $Q$ is the sum of the probability that $Q$ lies in its uncertainty part that after $P$ range, i.e., [$e_P$-$e_Q$], plus half the probability that $Q$ is in the overlapped uncertainty part, i.e, [$s_P$-$e_P$]. Formally, $P_{upper} = Pr\{Q \in [e_P\text{-}e_Q]\} + \frac{1}{2}Pr\{Q \in [s_P\text{-}e_P]\}$.

- *Case 3:* $s_P < s_Q$, $e_P < e_Q$. This is the case depicted in Figure 3.4b. In this case, the probability that the uncertainly range of $P$ is not dominated by that of $Q$ is one minus the probability that $Q$ dominates $P$. Formally, $P_{upper} = 1 - \frac{1}{2}(Pr\{P \in [s_Q\text{-}e_P]\} * Pr\{Q \in [s_Q\text{-}e_P]\})$.

- *Case 4:* $s_P < s_Q$, $e_Q < e_P$. This is the case where the uncertainty range of $Q$ is a subset of that of $P$. In this case, the probability that $P$ is not dominated by $Q$ is the sum of the probability that $P$ lies in its uncertainty part that is before $Q$ range, i.e., $[s_P\text{-}s_Q]$, plus half the probability that $P$ is in the overlapped uncertainty part, i.e, $[s_Q\text{-}e_Q]$. Formally, $P_{upper} = Pr\{P \in [s_P\text{-}s_Q]\} + \frac{1}{2}Pr\{P \in [s_Q\text{-}e_Q]\}$.

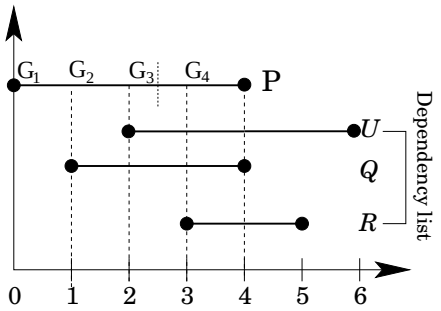
**Skyline Example.** Continuing the example shown in Figure 3.2b. Object $V$ does not qualify for *pairwise comparison* with any other object as its uncertainty range does not overlap with others. For object $R$, the ordered pairs ($R$,$Q$), ($R$,$U$), ($R$,$P$), and ($R$,$S$) qualify for *pairwise comparison*. For ($R$,$Q$), according to Case 3, the upper bound probability of $Q$ is reduced to 1-$\frac{1}{2}$*$\frac{1}{3}$*$\frac{1}{2}$=91%. Similarly, $U$ upper bound is reduced to 31% (Case 3), $P$ upper bound is reduced to 87% (Case 3), and $S$ upper bound is reduced to 25% (Case 1). Notice that we do not consider the pair ($R$,$T$) as $T$ is already determined to have no probability of being a preferred object. Should we have considered a 50% threshold, we would not also considered the pairs ($R$,$U$) and ($R$,$S$). For object $Q$, the pairs ($Q$,$U$), ($Q$,$P$), and ($Q$,$S$) qualify for *pairwise comparison* where $U$, $P$ and $S$ upper bounds are reduced to 16%, 62%, and 4% respectively. For object $U$, the pairs ($U$,$S$) and ($U$,$P$) qualify for pairwise comparison, however the upper bound for $U$ and $S$ cannot be reduced less than the 16% and 4% we had so far. For object $P$, the pair ($P$,$S$) qualifies for pairwise comparison, yet $S$ upper bound can not be reduced less than 4%. Finally, object $S$ does not qualify for pairwise comparison with other objects.

**Top-2 Example.** Continuing the example shown in Figure 3.3b. No pair of objects is applicable for pairwise comparison, for example the pair ($R$,$P$) does not qualify for pairwise comparison as the second condition does not hold.

### 3.4.3 Segmentation

**Objective.** *Segmentation* mainly has two objectives: (a) calculating both lower and upper bounds on the object probability to be a preferred object, and (b) Preparing all objects to the *bound tightening* method, (Section 3.4.4).

**Affected Objects.** In contrast to *uncertainty reduction* and *pairwise comparison* that are applied for a pair of objects, *segmentation* is applied for a given object $P$ and a




(a) Segmentation

| $G_i$ | $s_i$ | $e_i$ | lower | upper | error |
|---|---|---|---|---|---|
| $G_1$ | 0 | 1 | 1 | 1 | 0 |
| $G_2$ | 1 | 2 | 0.67 | 1 | 0.08 |
| $G_3$ | 2 | 3 | 0.25 | 0.67 | 0.10 |
| $G_4$ | 3 | 4 | 0 | 0.25 | 0.06 |

(b) Bound tightening

Figure 3.5: Segmentation of Object $P$

list of objects $DL_P$, termed the *dependency list* of $P$. An object $Q \in DL_P$ if and only if the pair ($Q$,$P$) qualifies for *pairwise comparison*. In other words, $DL_P$ includes all objects in the data set that affect the probability of $P$ being a preferred object. For example, in Figure 3.2a, $DL_P = \{R, Q, U\}$. The procedure holds for skyline, top-k and multi-objective preference functions.

**Main Idea.** Applying *segmentation* to an object $P$ goes through three main steps:

1. We divide the uncertainty range of $P$ to a set of $n$ disjoint segments, $G_1$, $G_2$, $\cdots$, $G_n$, in a way that the endpoint of segment $G_i$ is the starting point of segment $G_{i+1}$. The segment boundaries are defined by the endpoints of all objects in $P$'s dependency list, $DL_P$, that lie within the $P$ uncertainty range.

2. For each segment $G_i = [s_{G_i}\text{-}e_{G_i}]$, we compute two probabilities, $G_{i_{lower}}$ and $G_{i_{upper}}$, as the lower and upper bound probabilities that $P$ would be a preferred object should it lie anywhere within segment $G_i$. $G_{i_{lower}}$ would be the probability that $P$ is a preferred object should it have the value $e_{G_i}$ in its uncertainty range. This happens only if *all* the objects in $DL_P$ lie in the range that is beyond $e_{G_i}$ in their uncertainty ranges. So, $G_{i_{lower}}$ is computed as the multiplication of all probabilities of objects in $DL_P$ having values more than $e_{G_i}$ in their uncertainty ranges. Formally, $G_{i_{lower}} = \Pi_{X \in DL_P} Pr\{X > e_{G_i}\}$. Similarly, $G_{i_{upper}}$ is the probability that $P$ is a preferred object should it have the value $s_{G_i}$ in its uncertainty range. Formally, $G_{i_{upper}} = \Pi_{X \in DL_P} Pr\{X > s_{G_i}\}$.

3. Given the computed lower and upper bound probabilities, $G_{i_{lower}}$ and $G_{i_{upper}}$, for each segment $G_i$, we compute lower and upper bound probabilities of $P$ to be a preferred object, as the weighted sum over all segments of $P$. The sum is weighted based on the probability of $P$ being in each segment. Formally, $P_{lower} = \sum_{\forall G_i \in P} G_{i_{lower}} * Pr\{P \in G_i\}$ and $P_{upper} = \sum_{\forall G_i \in P} G_{i_{upper}} * Pr\{P \in G_i\}$.

By doing so, the probability of $P$ being a preferred object is within the range [$P_{lower}$-$P_{upper}$]. It is important to note that the more segments we have, the tighter the bound we can get for $P$. If $P_{upper}$ is less than $H$, then we do not need to consider $P$ for any further computations as we are sure that $P$ is not a query answer. Similarly, if $P_{lower} > H$ and $(P_{upper} - P_{lower}) < \delta$, we do not need to do any further computations on $P$ as we are sure that its a query answer with the required accuracy.

**Skyline Example.** Figure 3.5a focus on object $P$ and the objects in its dependency list, $R$, $Q$, and $U$. In this example, $P$ is divided into four segments based on the endpoints of $R$, $Q$, and $U$ as $G_1$ = [0-1], $G_2$ = [1-2], $G_3$ = [2-3], and $G_4$ = [3-4]. Focusing on $G_2$, for example, the lower-bound probability of segment $G_2$, $G_{2_{lower}}$, is the probability that if $P$ has the value $e_{G_2} = 2$, $P$ will not be dominated by any of the points in its dependency list. So, $G_{2_{lower}} = 1 * \frac{2}{3} * 1 = 67\%$ where if $P$ lies at $e_{G_2}$ it will have 100%, 67%, and 100% probability not to be dominated by $U$, $Q$, and $R$, respectively. Similarly, $G_{2_{upper}} = 1 * 1 * 1 = 1$ where if $P$ lies at $s_{G_2}$, it will not be dominated by any of the points with 100% probability. Similar calculations will be performed for the rest of the segments; the final result is given in Figure 3.5b. Finally, the lower and upper-bound probabilities that $P$ is a preferred object are: $P_{lower} = (1 * \frac{1}{4}) + (.67 * \frac{1}{4}) + (.25 * \frac{1}{4}) + (0 * \frac{1}{4}) = 48\%$. and $P_{upper} = (1 * \frac{1}{4}) + (1 * \frac{1}{4}) + (.67 * \frac{1}{4}) + (.25 * \frac{1}{4}) = 73\%$. The lower and upper bounds probability for all objects are shown in Figure 3.2(d).

**Top-2 Example.** Figure 3.3(d) shows the lower and upper bounds probability for all objects. We divide the object into segments, exactly as shown in figure 3.5a, and calculate the lower and upper bound probability for each segment. For example, object $Q$ is divided into two segments: $Q_1$ [5,8] and $Q_2$ [8,11]. The lower and upper bound probability for segment $Q_1$ is 100%.

### 3.4.4 Bound Tightening

**Objective.** Bound tightening starts from the probability bounds $[P_{lower} - P_{upper}]$ that came out from *segmentation*, then, it aims to iteratively increasing $P_{lower}$ while decreasing $P_{upper}$ until one of the following two conditions hold: (a) $P_{upper} < H$, where we can conclude that $P$ is not part of the query answer or, (b) $P_{lower} > H$ and $P_{upper} - P_{lower} < \delta$ in which we can conclude that $P$ is part of the query answer with the required accuracy.

**Affected Objects.** Similar to the case of *segmentation*, *bound tightening* is applied for a given object $P$ with the list of objects in its *dependency list*, $DL_P$.

**Main Idea.** As was discussed in the *segmentation*, the more segments that we have within $P$, the better accuracy bounds we can get. To this end, *bound tightening* aims to introduce more segments within $P$ uncertainty range. To do so in a very simple, efficient, and greedy way, we *split* one of the segments into two halves. As this will increase the number of segments by one more segment, it will also tighten the current probability bounds $[P_{lower} - P_{upper}]$. We keep doing so, iteratively, till we reach to the stopping condition, ($P_{upper} < H$ OR ($P_{lower} > H$ AND $P_{upper} - P_{lower} < \delta$). The faster we approach to the stopping condition, the more efficient our scheme will be. So, *bound tightening* always selects the segment to split, $G_s$, as the one that has the largest weighted difference in its upper and lower bound probabilities, i.e., $G_s = MAX_{\forall G_i}((G_{i_{upper}} - G_{i_{lower}}) * Pr\{P \in G_i\})$. By doing so, *bound tightening* will reach to its stopping conditions in less iterations. It is important to note that splitting $G_s$ into two halves $G_{s1}$ and $G_{s2}$ results in calculating only the probability $G_{s1_{lower}}$ as it is also equal to $G_{s2_{upper}}$. The other bounds are inherited from $G_s$ where $G_{s1_{upper}} = G_{s_{upper}}$ and $G_{s2_{lower}} = G_{s_{lower}}$. Finally, $P_{lower}$ and $P_{upper}$ are updated incrementally to reflect the new probability bounds.

**Skyline Example.** Consider object $P$ in Figure 3.5. Of all segments, $G_3$ has the largest calculation error (i.e., $(0.67 - 0.25) * \frac{1}{4} = 0.10$). Thus, $G_3$ is chosen to be split into two halves $G1_3$ and $G2_3$. Then, only *one* probability value needs to be calculated, $G1_{3_{lower}} = G2_{3_{upper}}$ =0.44. Consecutively, $P_{lower}$ and $P_{upper}$ are updated to be 0.5% and 0.7%, respectively.

**Top-2 Example.** Consider object $Q$ in figure 3.3d. We need five adaptive iterations to bound the probability of $Q$ between [85.3, 86.2].

## 3.5 UPref: Query Processing for Uncertain Data

This section presents our proposed UPref algorithm algorithm that encapsulates the four probability bounding methods discussed in Section 3.4 together. Our two-phase algorithm follows a filter-refine approach in which Phase I prunes several objects using simple calculations, then Phase II iteratively refines the answer of the first phase to get tighter bounds gradually while pruning more objects in each iteration. In particular, Phase I scans over the input data to apply the *uncertainty reduction* and *pairwise comparison* methods as well as preparing the *dependency list* for objects that still have an upper bound probability more than the user-defined threshold $H$ to be a preferred object. Then, Phase II goes only through those objects that came out of the first phase and apply both *segmentation* and *bound tightening* for each object $P$ until its probability bounds $P_{lower}$ and $P_{upper}$ satisfy any of the following two conditions: (a) $P_{upper} < H$, in which we conclude that $P$ is not a query answer, or (b) $P_{lower} > H$ and $P_{upper} - P_{lower} < \delta$, in which we conclude that $P$ is a query answer with an accurate probability calculation.

### Phase I: Preference and Dependency Lists

**Main Idea.** Phase I performs initial filtering of objects by calculating their upper-bound probability to be preferred objects using *uncertainty reduction* and *pairwise comparison* methods. Phase I also populates the *dependency list* for each object $P$ in the dataset. As both *uncertainty reduction* and *pairwise comparison* are applied pairwise, it is important to consider all possible pairs. We could have done this with a simple nested loop over all objects in both the outer and inner loops, but that would be very expensive. To avoid this, we keep the outer loop as a simple scan over all objects, however, we limit the inner loop to scan over only the set of objects that are already scanned in the outer loop and still have a probability more than $H$ to be a preferred object. By doing so, we still guarantee that all qualified pairs for objects with probability more than $H$ will be considered. However, as Phase I is also responsible on populating the *dependency list* of each preferred object, and as the *dependency list* may contain some of the objects that have an upper bound probability less than $H$, we need to keep those objects that have a non-zero probability less than $H$ in a separate temporarily list to

guarantee the correctness of the *dependency list*.

**Algorithm 5 Phase I**: Initial Preference Computation

```
1: Function Phasel(DataSet D, Threshold H)
2: Initialize Preference and ThresholdDominated to NULL
3: for each Object Q ∈ D do
4:    for each Object P ∈ Preference do
5:       if (P,Q) or (Q,P) qualify for uncertainty reduction then
6:          Apply uncertainty reduction to the dominated object
7:       end if
8:       if the pair (P,Q) qualifies for pairwise comparison then
9:          Apply pairwise comparison to Q
10:         Add P to the dependency list of Q, DL_Q
11:      end if
12:      if the pair (Q,P) qualifies for pairwise comparison then
13:         Apply pairwise comparison to P
14:         Add Q to the dependency list of P, DL_P
15:      end if
16:      if P_upper < H then
17:         if P_upper > 0 then Add P to ThresholdDominated
18:         Remove P from Preference
19:      end if
20:      if Q_upper < H then
21:         if Q_upper > 0 then Add Q to ThresholdDominated
22:         Continue to next object Q
23:      end if
24:   end for
25:   Add Q to Preference
26: end for
27: for each Object P ∈ Preference do
28:    for each Object T ∈ ThresholdDominated do
29:       if P depends on T then add T to P's dependency list
30:    end for
31: end for
32: return Preference
```

**Data Structures.** Each object $P$ in the data set is associated with an upper-bound probability ($P_{upper}$) of being a preferred object that is initialized to one. We also maintain two lists: (1) The $Preference$ list stores objects that currently have an upper bound probability above the threshold $H$; with each object $P \in Preference$, we store a *dependency list*, $DL_P$, that maintains all points that affect $P$'s probability of being a preferred object. (2) The *ThresholdDominated* list stores the objects whose upper bound probability is more than zero and less than $H$. This is a temporarily list that is

used to guarantee the correctness of $DL_P$, $\forall P \in Preference$.

**Algorithm.** Algorithm 5 gives the pseudo-code for Phase I. The algorithm takes as input two parameters: (1) A data set $D$, and (2) a user-given probability threshold $H$. Notice that we do not consider the user tolerance $\delta$ in this phase as it is only used in the second phase. This phase starts by initializing the lists *Preference* and *ThresholdDominated* to null. Then, for each pair of objects $Q \in D$ and $P \in Preference$, we check the ordered pairs ($P$,$Q$) and ($Q$,$P$) for both *uncertainty reduction* and *pairwise comparison* and act accordingly (Lines 5 to 15 in Algorithm 5). It is important to note that *uncertainty reduction* can be applied only to either ($P$,$Q$) or ($Q$,$P$) while the *pairwise comparison* can be applied to both pairs. That is why *pairwise comparison* is checked twice. Within the pairwise comparison process, we will also update the *dependency lists* accordingly.

After checking all the qualifications, we check if the upper bound probabilities of $P$ and/or $Q$ went below the user defined threshold $H$. If this is the case for $P$, we remove $P$ from the $Preference$ list, however, we will still need to add $P$ to the $ThresholdDominated$ list if it has a non-zero probability of being a preferred object. Similarly, if this is the case for $Q$, we do not have to continue checking $Q$ with the rest of objects in $Prferecnes$, however, we will add $P$ to the $ThresholdDominated$ list if it has a non-zero probability of being a preferred object (Lines 16 to 23 in Algorithm 5). The main idea behind keeping $P$ and/or $Q$ in the $ThresholdDominated$ is that these objects may still affect the probability of other objects in $D$ that are not processed yet, and hence may be added to their *dependency lists*. In the mean time, we cannot just store these objects in the $Preference$ list as (a) these objects are not going to be reported in the final answer, thus, there is no need to pass them to the second phase and (b) in contrast to the objects in the $Preference$ list, there is no need to keep an upper bound probability or dependency list for these objects.

If after comparing object $Q$ with all objects in the $Preference$ list, $Q$'s upper bound probability is still above $H$, then we add $Q$ to the $Preference$ list (Line 25 in Algorithm 5). Finally, Phase I is concluded by comparing each object $P \in Preference$ with each object $T \in ThresholdDominated$ to find if there is any object $T$ that needs to be added to the *dependency list* of any object $P$ (Lines 27 to 31 in Algorithm 5). Phase I returns the $Preference$ list as its output that is passed to Phase II.

**Example.** Table 3.1 gives the result of applying Phase I to our running example in Figure 3.2a when $H = 50\%$ and the data is read in the order of $U$, $R$, $S$, $V$, $Q$, $T$, and $P$. Each row in the table gives the status of the $Preference$ and $ThresholdDominated$ lists after the insertion of each object. The objects in the $Preference$ list are presented as triples of (*object ID*, *upper bound probability*, and *dependency list*). When reading the first object $U$, it will be inserted in the $Preference$ list with 100% probability and empty dependency list. When reading $R$, the pair ($R$,$U$) qualifies for both *uncertainty reduction* and *pairwise comparison* that would reduce the probability of $U$ to 50% while adding $R$ to the dependency list of $U$. Since the pair ($U$,$R$) does not qualify for pairwise comparison, nothing will affect $R$ entry in the list. Upon reading $S$, it qualifies for pairwise comparison with $R$ which reduces $S$ probability to 33%. Thus, $S$ is inserted in the $ThresholdDominated$ list without further comparison with $U$. Upon reading $V$, we find that it does not qualify to neither uncertainty reduction nor pairwise comparison with any other object, so, we just add it to the $Preference$ list with 100% probability and empty dependency list. Upon reading $Q$, the pair ($R$,$Q$) qualifies for pair comparison and reduces $Q$ probability to be 91% while adding $R$ to the $Q$ dependency list. Then, the pair ($Q$,$U$) qualifies for uncertainty reduction where $U$ probability is dropped below $H$, and thus will be moved from the $Preference$ list to the $ThresholdDominated$ list. Upon reading $T$, it is totally dominated by $Q$, so, will not be stored anywhere. Finally, upon reading $P$, both $R$ and $Q$ affect $P$ probability, they are added to its dependency list and reduce its probability to 62%.

| | Preference List | ThDom |
|---|---|---|
| $U$ | ($U$,100%,{}) | {} |
| $R$ | ($R$,100%,{}), ($U$,50%,{$R$}) | {} |
| $S$ | ($R$,100%,{}), ($U$,50%,{$R$}) | {$S$} |
| $V$ | ($V$,100%,{}), ($R$,100%,{}), ($U$,50%,{$R$}) | {$S$} |
| $Q$ | ($V$,100%,{}), ($R$,100%,{}), ($Q$,91%,{$R$}) | {$U$,$S$} |
| $T$ | ($V$,100%,{}), ($R$,100%,{}), ($Q$,91%,{$R$}) | {$U$,$S$} |
| $P$ | ($V$,100%,{}), ($R$,100%,{}), (($Q$,91%,{$R$}), ($P$,62%,{$R$, $Q$}) | {$U$,$S$} |

Table 3.1: Example for Phase I

**Phase II: Final Probability Calculation**

**Main Idea.** Phase II is applied only to those objects in the $Preference$ list that have a non-empty *dependency list*. Other objects in the $Preference$ list with empty dependency list are guaranteed to be in the query answer, with probability 100%, as no other objects can affect their probabilities. The main idea of Phase II is for each object $P \in Preference$, where $DL_P \neq \phi$, we first apply *segmentation* using the entries in $DL_P$. Then, we iteratively apply *bound tightening* until either (a) $P_{upper} < H$, where we conclude that $P$ is not a query answer, or (b) $P_{lower} > H$ and $P_{upper} - P_{lower} < \delta$, where we conclude that $P$ is a query answer with an accurate probability calculations.

**Data Structures.** The second phase does not maintain any particular data structure. Instead, it takes the $Preference$ list from the first phase which keeps an upper bound probability and dependency list for each object with $P_{upper} > H$. However, the second phase adds one more field to the entries in the $Preference$ list that maintains the lower bound probability of each object to be a preferred one.

**Algorithm.** Algorithm 6 gives the pseudo-code for the final probability calculation of Phase II. The algorithm takes as input the list $Preference$ that is passed from the first phase, the user-given threshold $H$, and a user-given tolerance $\delta$. For each object $P$ that qualifies for this phase, i.e., $P \in Prefernce$ and $DL_P \neq \phi$, we apply the *segmentation* method in which upper and lower bounds for $P$ are derived through dividing $P$ into $n$ segments using the endpoints of all objects in $DL_P$ (Lines 3 to 9 in Algorithm 6). If, after *segmentation*, the upper bound probability $P_{upper} > H$ and the probability bound calculation is not accurate enough, i.e., $P_{upper} - P_{lower} > \delta$, then we will need to iteratively apply the *bound tightening* method. In this case, we pick the segment with largest weighted probability difference to be split into two equal segments and update $P_{lower}$ and $P_{upper}$ accordingly. We keep doing so iteratively until either $P_{upper} < H$, in which we conclude that $P$ is not a query answer, or ($P_{lower} > H$ and $P_{upper} - P_{lower} < \delta$), in which we conclude that $P$ is a query answer with accurate probability calculation (Lines 10 to 16 in Algorithm 6). Finally, $P$ is removed from the $Prefernece$ list if its upper bound probability is below $H$. Phase II is concluded by returning the $Preference$ list as the query answer.

**Example.** Considering the final $Preference$ list in the last row of table 3.1 with $H = 50\%$ and $\delta = 5\%$, Phase II will be applied only to objects $Q$ and $P$ as objects

$V$ and $R$ have empty dependency lists, and thus can be reported directly as a query answer. Figure 3.5 gives the example of applying segmentation and bound tightening on object $P$ for only one iteration. Continuing on *bound tightening* for $P$, we will need 10 iterations to have $P_{lower} = 57\%$ and $P_{upper} = 61\%$. Similarly, for object $Q$, we will need 3 iterations to have $Q_{lower} = 89.5\%$ and $Q_{upper} = 93.7\%$, thus, the final returned answer of our algorithm is: $(V, 100\%)$, $(R, 100\%)$, $(Q, [89.5\text{-}93.8]\%)$, and $(P, [57\text{-}61]\%)$.

**Algorithm 6 Phase II:** Final probability calculation

**Function** PhaseII (List $Preference$, Threshold $H$, Tolerance $\delta$)
**for each** $P \in Preference$ where $DL_P \neq \phi$ **do**
  Do *segmentation* on $P$ to get segments $\mathcal{G} = \{G_1, G_2, .., G_n\}$
  **for** each segment $G \in \mathcal{G}$ **do**
    $G_{lower} \leftarrow \Pi_{X \in DL_P} Pr\{X > e_{G_i}\}$
    $G_{upper} \leftarrow \Pi_{X \in DL_P} Pr\{X > s_{G_i}\}$
  **end for**
  $P_{lower} \leftarrow \sum_{\forall G_i \in P} G_{i_{lower}} * Pr\{P \in G_i\}$
  $P_{upper} \leftarrow \sum_{\forall G_i \in P} G_{i_{upper}} * Pr\{P \in G_i\}$
  **while** $P_{upper} > H$ **and** $(P_{upper} - P_{lower}) > \delta$ **do**
    $G_s \leftarrow MAX_{\forall G_i}(G_{i_{upper}} - G_{i_{lower}}) * Pr\{P \in G_i\}$
    Split $G_s$ into two halves $G_{s1}$ and $G_{s2}$
    $G_{s1_{lower}} \leftarrow G_{s_{lower}}$; $G_{s2_{upper}} \leftarrow G_{s_{upper}}$;
    $G_{s1_{upper}} = G_{s2_{lower}} \leftarrow \Pi_{X \in DL_P} Pr\{X > e_{G_s}\}$
    Update $P_{lower}$ and $P_{upper}$ based on the two new segments
  **end while**
  **if** $P_{upper} < H$ **then** remove $P$ from $Preference$ list
**end for**
**return** $Preference$

## 3.6 Supporting Top-$k$ and Multi-Objective Preference Methods

we have discussed our preference query processing framework for uncertain data for using skyline and top-k preference methods. However, a main goal of proposing our framework was flexibility in handling *multiple* preference methods. In this section, we summarize changes in our framework to support two popular preference methods: top-$k$ [1] and multi-objective [5] queries.

### 3.6.1 Top-$k$ Queries

Given a data set $D$, the top-$k$ preference method [1] scores each object $p \in D$ using a monotonic ranking function $f$ applied over all of $p$'s attributes. Traditionally, in environments without data uncertainty, $f$ returns a single real number as a ranking score, and the query answer consists of the $k$ objects with the highest score. In our uncertainty environment, however, the output of function $f$ is a single uncertain dimension represented as a continuous range. For example, if we apply a ranking function $f = 2 * d_1 + d_2$ to object $U$=([2-6],4) from Figure 3.1a, then $U$ is mapped to the single uncertain dimension [8-16].

Given this difference from traditional top-$k$ environments, the objective of top-$k$ query processing for uncertain data in our case is to report those objects that have a probability more than $H$ to be in the *top-k* preferred objects. For those objects, we also report their probabilities with an accuracy $\delta$. This objective implies our framework does not necessarily return $k$ objects. Instead, it returns an arbitrary number of objects $n$ that can be greater, equal to, or less than $k$ where $n$ is the number of objects that have a probability greater than $H$ to be the *top-k* objects. For example, given $H$=50%, the top-2 set in Figure 3.1b contains a *single* object $Q$. Meanwhile, if $H$=1%, the top-2 query result in Figure 3.1b returns four objects {$P$, $Q$, $R$, $U$}.

We now discuss how to apply our framework to top-$k$ queries by first discussing the details of uncertainty reduction, pairwise comparison, segmentation, and bound tightening. We then discuss applying these methods in our two-phase framework to process the top-$k$ queries.

**Uncertainty Reduction.** Calculations for uncertainty reduction for the *top-k* case are the same as that for *skyline* queries. However, the uncertainty reduction qualification conditions for *top-k queries are different*. For *top-k*, an object pair ($Q$,$P$) qualifies for uncertainty reduction only if: (a) the endpoint of $Q$ is less than that of $P$, and (b) $Q$ has the $k^{th}$ endpoint among all objects considered so far. This case means that any object that starts after $Q$ endpoint has no chance to be a preferred object. The main idea is that because $Q$ has the $k^{th}$ endpoint, it has the privilege to reduce the uncertainty range of other points in the data set.

**Pairwise Comparison.** Similar to the case of *uncertainty reduction*, pairwise comparison calculations for a qualified ordered pair ($Q$,$P$) in *top-k* queries is the same as

that of *skyline* queries. However, they are different in the qualification conditions where ($Q$,$P$) qualifies for *top-k* pairwise comparison only if: (a) either the endpoint of $Q$ or $P$ is the $k^{th}$ endpoint among all considered points so far, (b) there exist $k$-1 endpoints before the start point of $P$ or $Q$, and (c) the uncertainty ranges of $Q$ and $P$ overlap.

**Segmentation.** An object $P$ is broken into segments $G_i$ exactly as in the case of skyline queries. However, the upper and lower-bound probability calculations are different than the skyline case. The upper bound probability of segment $G_i$ is the probability that at most $k$-1 objects from P's dependency list, $DL_P$, are smaller than the start point of $G_i$. Similarly, the lower bound of segment $G_i$ is the probability that at most $k$-1 objects in the $DL_P$ are smaller than the end point of $G_i$. Clearly, if the number of objects that are less than the start point (endpoint) of $G_i$ are less than $k$-1, the upper (lower) bound probability for $G_i$ is 100%. Furthermore, if $DL_P$ contains fewer than $k$-1 objects, the upper and lower bound probabilities for $P$ is 100%.

**Bound Tightening.** Bound tightening proceeds exactly the same for both *top-k* and *skyline* queries.

| | Preference List | ThDom |
|---|---|---|
| $U$ | ($U$,100%,{}) | {} |
| $R$ | ($U$,100%,{}), ($R$,100%,{}) | {} |
| $S$ | ($U$,91%,{$S$}), ($R$,100%,{}), | {$S$} |
| $V$ | ($U$,81%,{$S$,$V$}), ($R$,100%,{}) | {$S$,$V$} |
| $Q$ | ($U$,50%,{$S$,$V$,$R$}), ($R$,100%,{}), ($Q$,100%,{}) | {$S$,$V$} |
| $T$ | ($U$,50%,{$S$,$V$,$R$}), ($R$,100%,{}), ($Q$,100%,{}) | {$S$,$V$} |
| $P$ | ($U$,50%,{$S$,$V$,$R$}), ($R$,100%,{}), ($Q$,100%,{}), ($P$,75%,{$R$}) | {$S$,$V$} |

Table 3.2: Example for Phase I on Top-2 query

**Algorithms: Phase I.** Two minor operations in the algorithm for Phase I are different for the *top-k* case compared to *skylines*. (1) When adding an object $Q$ to the *Preference* list (Line 21 in Algorithm 5), the endpoint of $Q$ may be less than the $k^{th}$ endpoint in *Preference*, creating a new $k^{th}$ endpoint. In this case, all objects $P_i$ currently in *Preference* that have endpoints *after* the new $k^{th}$ endpoint go through uncertainty reduction. These objects $P_i$ have portions of their uncertainty range that cannot possibly

be *top-k* values. (2) When populating $DL_Q$ for an object $Q$, we add objects from both *ThresholdDominated* and *Preference* that either (a) *overlap* with $Q$, or (b) have endpoints that come before $Q$'s start point. These are objects that affect $Q$'s probability of being a *top-k* object.

**Algorithms: Phase II.** Phase II is exactly the same for both *top-k* and *skyline* queries.

**Example.** Table 3.2 gives the result of applying Phase I to our running example in Figure 3.3a with $H = 50\%$ and the data is read in the order of $U$, $R$, $S$, $V$, $Q$, $T$, and $P$. When reading $U$, it will be inserted in the $Preference$ list with 100% probability and empty dependency list. Similarly, we read $R$, and insert it in the *Preference list*. Upon reading $S$, it qualifies for uncertainty reduction and pairwise comparison with $U$ which reduces $S$ probability to 8%, and $U$ probability to 91%. Thus, $S$ is inserted in the *ThresholdDominated* list. Upon reading $V$, we find that it qualifies for uncertainty reduction and pairwise comparison with $U$. So, $V$ and $U$ probabilities are reduced to 18.7%, and 81%, respectively. Therefore, we add $V$ to *ThresholdDominated* list. Upon reading $Q$, we find that it does not qualify to neither uncertainty reduction nor pairwise comparison with any other object, so, we just add it to the $Preference$ list with 100% probability. However, $Q$ pushes $R$ to be the $2^{nd}$ endpoint and thus the pair ($R$,$U$) becomes qualified for uncertainty reduction that reduces $U$ probability to 50% while adding $R$ to the $U$ dependency list. Upon reading $T$, it is totally dominated by $R$, so, it will not be stored anywhere. Upon reading $P$, the pair ($R$,$P$) qualifies for uncertainty reduction, hence the $P$ is reduced to 75%. Finally, we update the dependency lists for points in the $Preference$ list, where we add $Q$, $U$ to $DL_P$, $U$, $R$, $P$ to $DL_Q$, $U$, $Q$, $P$ to $DL_R$, and $Q$, $P$ to $DL_U$.

### 3.6.2 Multi-Objective Queries

Given a set of $n$-dimensional objects, the multi-objective preference method [5] combines $m$ dimensions ($1 < m < n$) using a monotonic function $f$. The query answer is the result of a skyline computed over the resulting $(n - m)+1$ independent dimensions. To process multi-objective queries in our framework, we can use the algorithms described in Section 3.5 for the skyline method *without* any changes, adding only a small pre-processing step to evaluate $f$ for each object before being processed by Algorithm 5. We now identify and discuss three distinct cases for evaluating $f$ based on whether $f$

contains uncertain data.

**Function $f$ contains only certain data**. In this case, function $f$ is applied to only certain data attributes of an object, while the uncertain dimension(s) are left unchanged. In this case, function $f$ will yield a single real value. For example, assume we have a data set $D$ containing two objects (1, 2, [3-5]) and (2, 1, [4-6]), and a function $f_1 = d_1 + d_2$ (i.e., the sum of the first two dimensions). In this case, the pre-processing step will evaluate $f$ to form objects (3, [3-5]) and (3, [4-6]), respectively, before they are processed by Algorithm 5.

**Function $f$ contains a single uncertain dimension**. In this case, function $f$ combines one or more certain attributes with a single uncertain attribute. In this case, function $f$ will yield a single uncertain dimension, where $f$ is applied to the begin and ending range of the single uncertain attribute, while its probability density function remains unchanged. For example, assume we have two objects (1, 2, [3-5]) and (2, 1, [4-6]), and a function $f_1 = d_2 + d_3$ (i.e., the sum of the second two dimensions). In this case, the pre-processing step will form objects (1, [5-7]) and (2, [5-7]).

**Function $f$ contains multiple uncertain dimensions**. In this case, function $f$ evaluates two (or more) uncertain attributes, yielding a single uncertain dimension. This case requires *convolution* to combine the probability density functions of the multiple attributes evaluated by function $f$. In general, it is outside the scope of this work to discuss specific convolution methods. However, many efficient algorithms exist in the literature [93].

## 3.7 Multiple and Non-Uniform Uncertain Dimensions.

In this section, we discuss extensions to our framework to handle non-uniform probability density functions and multiple uncertain dimensions.

### 3.7.1 Non-Uniform Probability Distribution

For a non-uniform distribution of object values over the uncertain range, all the techniques that we have described remain unchanged; except for the way we calculate the probability between two objects.

In particular, for *uncertainty reduction* on a qualified pair of objects ($Q$,$P$), the upper

bound probability for object $P$ is calculated as $\int_{s_P}^{e_Q} f_P(p)dp$ where $f_P$ is the probability density function over the uncertainty range.

Also, for *pairwise comparison* on a qualified pair of objects ($Q$, $P$), the upper bound probability for $P$ is set according to the four cases outlined in Section 3.4.2 as:

- **Case 1:** $\int_{s_P}^{e_Q}(\int_{p}^{e_Q} f_Q(q)f_P(p)dq)dp$,
- **Case 2:** $\int_{s_P}^{e_P}(\int_{p}^{e_Q} f_Q(q)f_P(p)dq)dp$
- **Case 3:** $\int_{s_P}^{s_Q} f_P(p)dp + \int_{s_Q}^{e_P}(\int_{p}^{e_Q} f_Q(q)f_P(p)dq)dp$
- **Case 4:** $\int_{s_P}^{s_Q} f_P(p)dp + \int_{s_Q}^{e_Q}(\int_{p}^{e_Q} f_Q(q)f_P(p)dq)dp$.

Finally, for *segmentation* and *bound tightening* over object $P$, the lower bound probability of a segment $G_i$ is $G_{i_{lower}} = \Pi_{X \in DL_P} \int_{e_{G_i}}^{e_X} f_X(x)dx$. Similarly, the upper bound probability for segment $G_i$ is $G_{i_{upper}} = \Pi_{X \in DL_P} \int_{s_{G_i}}^{e_X} f_X(x)dx$.

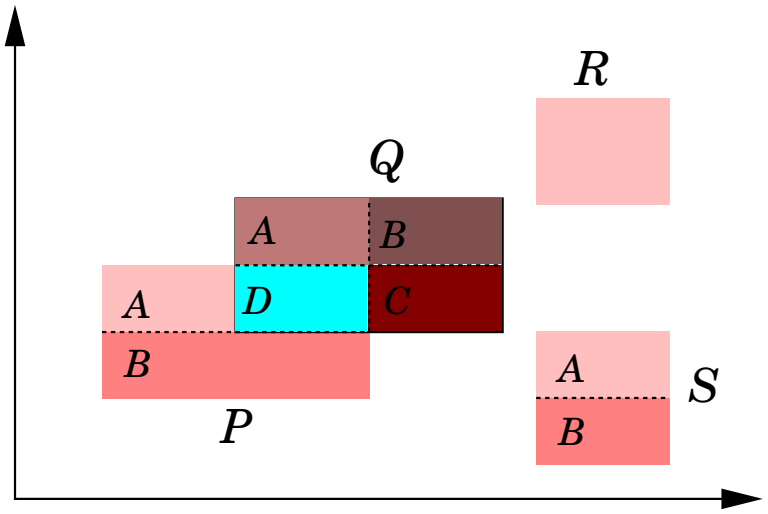



Figure 3.6: 2D uncertain points

### 3.7.2 Multiple Uncertain Dimensions

Our framework can be applied to multiple uncertain dimension by considering that the uncertainty part of each object is represented as an area rather than just as a line segment. In this section, we will show how our four proposed techniques can accommodate

multiple uncertain dimensions. Then, our two-phase framework can simply remain unchanged. Figure 3.6 is used as an illustrative example where four objects, $P$, $Q$, $R$, and $S$ have two uncertain dimensions, thus, their uncertainty areas are represented as rectangles.

**Uncertainty Reduction.** The only changes required in uncertainty reduction to accommodate multiple uncertain dimensions are: (a) the definitions for start point and endpoint are changed to be the points with the *highest* and *lowest* values in all uncertain dimensions, respectively, (b) Whenever an object $Q$ reduces the uncertainty region of object $P$, it removes from $P$ a rectangular region rather than just part of a line segment. In Figure 3.6, consider uncertainty reduction over object pair ($P$,$R$). This pair qualifies for reduction as $e_P$ dominates $e_R$. During reduction, the upper bound probability of $R$ reduces to 0% as start point $s_R$ is dominated by endpoint $e_P$. Also, pair ($P$,$Q$) qualifies for uncertainty reduction, where the upper bound probability for object $Q$ is reduced by the probability that $Q$ exists in segment $Q_B$, i.e., the upper right segment of $Q$.

**Pairwise Comparison.** The definition of "overlap" and upper-bound probability calculation change for pairwise comparison with multiple uncertain dimensions. Two objects can overlap on *any* uncertain dimension in order to qualify for pairwise comparison. For example, in Figure 3.6, object pair ($P$,$Q$) qualifies as $P$ and $Q$ overlap on both uncertain dimensions. Pair ($P$,$S$) also qualifies since $P$ and $S$ overlap on a single dimension. For multiple uncertain dimensions, the number of overlap cases for probability calculation expands. We do not outline each case, however, we give an example outlining the intuition to calculate upper bound probability. Consider object pair ($P$,$S$) that qualifies for pairwise comparison as their uncertain ranges overlap on one dimension. $S$ is only dominated if it exists in region $S_A$ (i.e., upper region of $S$) while $P$ exists region $P_B$. The probability of this case is $\frac{1}{2}*\frac{1}{2}*\frac{1}{2}$. Thus, $S$ is assigned an upper bound probability of $\frac{7}{8}$.

**Segmentation.** The segment dimensionality and probability calculations change for the segmentation operation with multiple-uncertain dimensions. Segments are $m$-dimensional where $m$ is the number of uncertain dimensions. For example, object $Q$ in Figure 3.6 is broken into four two-dimensional segments ($Q_A$, $Q_B$, $Q_C$, $Q_D$) based on its overlap with object $P$. The upper and lower-bound calculations for a segment change as a segment has the probability to exist in $m$-dimensional space. The intuition behind calculating

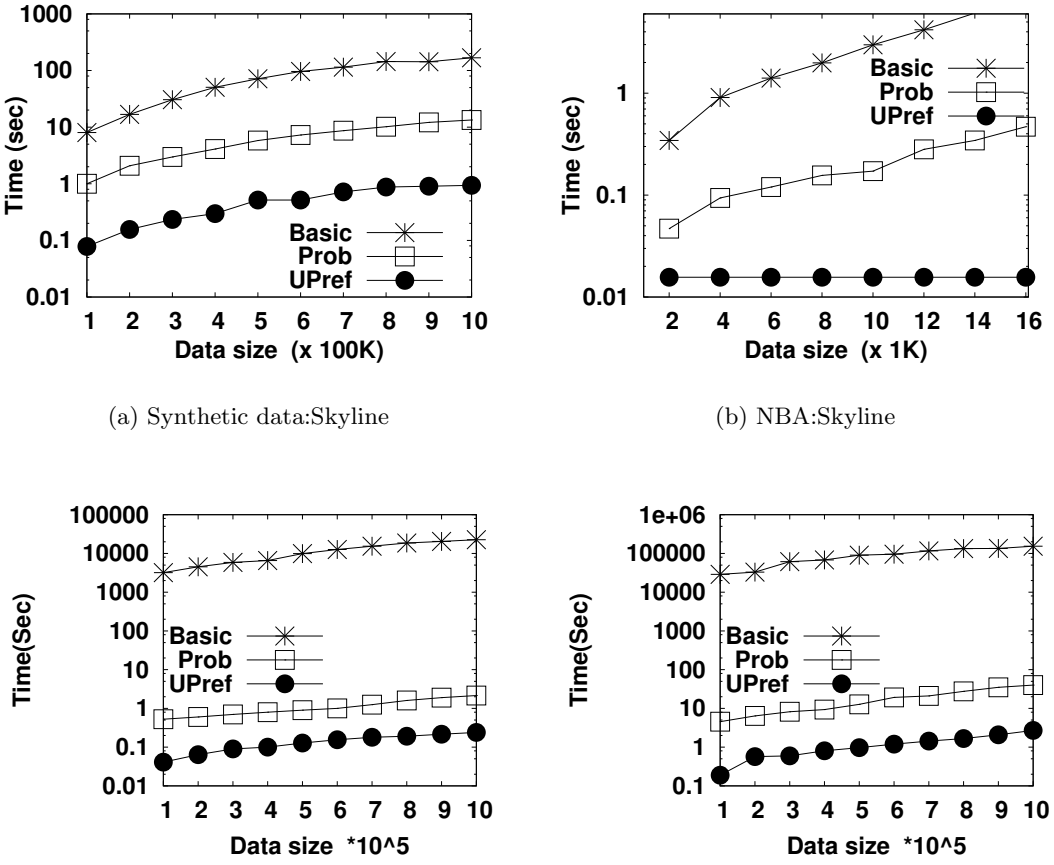
the bounds for $m$-dimensional space is given through an example. Consider object $Q$ in Figure 3.6, segment $Q_D$ has an upper bound probability of 75%, this is the chance that $Q_D$ is dominated if it exists in its start point (i.e., the lower left corner of $Q_D$). This case can only happen if $P$ exists in the lower left quadrant of its segment $P_B$. The lower bound probability of $Q_D$ is 0%, this is the chance that $Q_D$ is dominated when it exists in its endpoint (i.e., upper right corner of $Q_D$). A similar process applies to the rest of $Q$'s segments.

**Bound Tightening.** Bound tightening does not change for multiple uncertain dimensions. The segment that contributes the most to the difference between an object's upper and lower bound probability is always chosen during bound tightening.

## 3.8 Experimental Results

In this section, we analyze the performance of our proposed algorithm, denoted as UP-ref, for different preference functions. As discussed in Sections 3.1 and 3.2, there is no previous work for preference queries over uncertain data represented as continuous range other than for the case of nearest-neighbor queries [27, 28, 70]. Due to the lack of previous work, we consider the following two base algorithms as a basis of comparison: (a) A simple nested loop algorithm over all data with exact probability calculations, denoted as Basic, and (b) The Basic algorithm, yet, we utilize our *segmentation* and *bound tightening* methods for probability calculations instead of doing the exact expensive calculations, denoted as Prob.

We perform the experiments using two datasets: (a) *Synthetic*. We generate a synthetic eight-dimensional data set of one Million points, where each dimension represents a uniform random variable from 0 to 10,000 where only the first dimension is uncertain. (b) *NBA* [68]. This is a real dataset with records for 16,381 NBA players. Each record has 17 dimensions representing various player statistics. The NBA data is rather certain, however, we explicitly add uncertainty ranges to the data. For both data sets, and unless mentioned otherwise, we set the size of the uncertainty range to 20% of the attribute domain, the user-given threshold $H$ to 50%, and the user-given tolerance $\delta$ to 1%. The number of tested dimensions is set to 3 and 17 for the Synthetic and NBA datasets, respectively. All experiments are executed on 1.8 GHz Intel processor with




(a) Synthetic data:Skyline

(b) NBA:Skyline

(c) Synthetic data:Top-k

(d) Synthetic data:Multi-objective

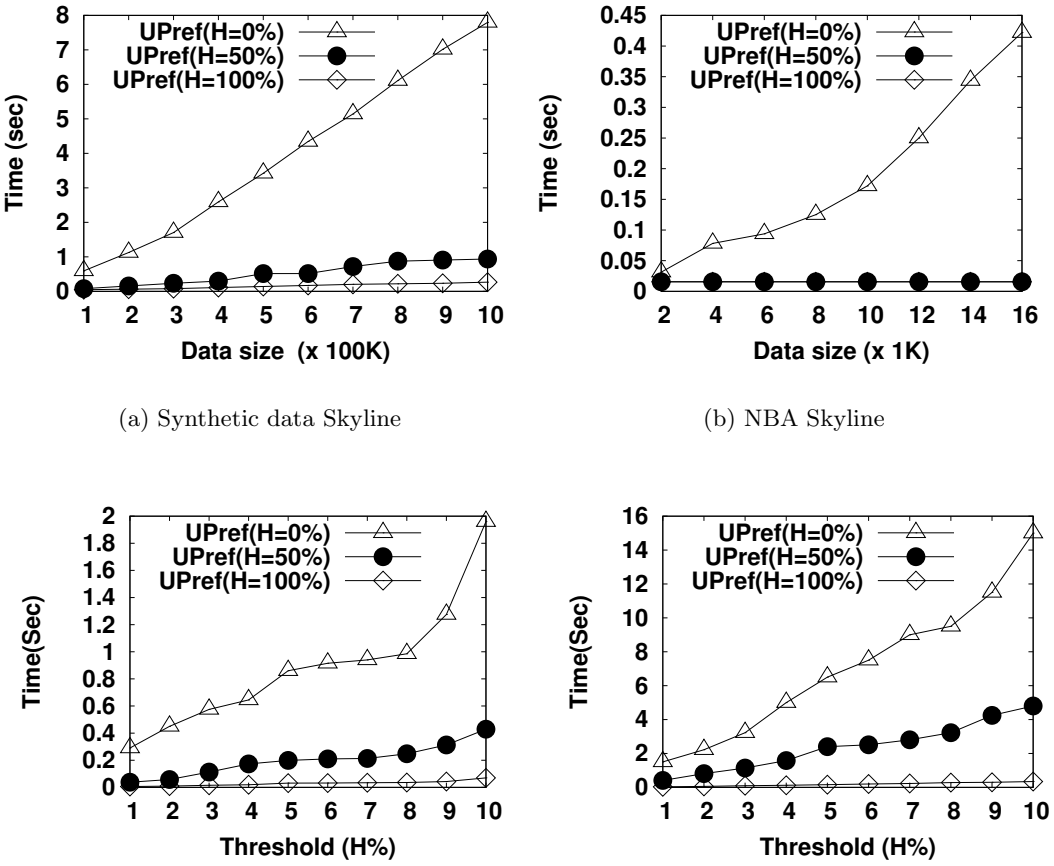
Figure 3.7: Scalability: Basic, Prob, UPref

1 GB of RAM. Our performance metric is the elapsed *wall clock time* to answer the preference query.

### 3.8.1 Scalability

Figures 3.7 gives the scalability of all algorithms when increasing the data size from 100K to 1 Million for Synthetic data and from 2K to 16K for NBA data. In all cases, our UPref exhibits orders of magnitude better performance than the other algorithms. The order of magnitude time difference between UPref and Prob is mainly due to the *uncertainty reeducation* and *pairwise comparison* operations. Meanwhile, the order of

magnitude time difference between Prob and Basic is due to our efficient probability calculation method, namely, *segmentation* and *bound tightening*. The superiority of UPref is even more clear for larger datasets. As an example, for skyline preference function, the total processing time is 0.9 seconds for 1M points, while the time for Prob and Basic is 13.0 seconds and 100 seconds, respectively (Figure 3.7(a)). Since the Basic algorithm is clearly worse than other algorithms by at least two orders of magnitude for both synthetic and NBA datasets, we will exclude it from further experiments.




(a) Synthetic data Skyline

(b) NBA Skyline

(c) Synthetic data Top-k

(d) Synthetic data Multi-objective

Figure 3.8: Scalability of UPref

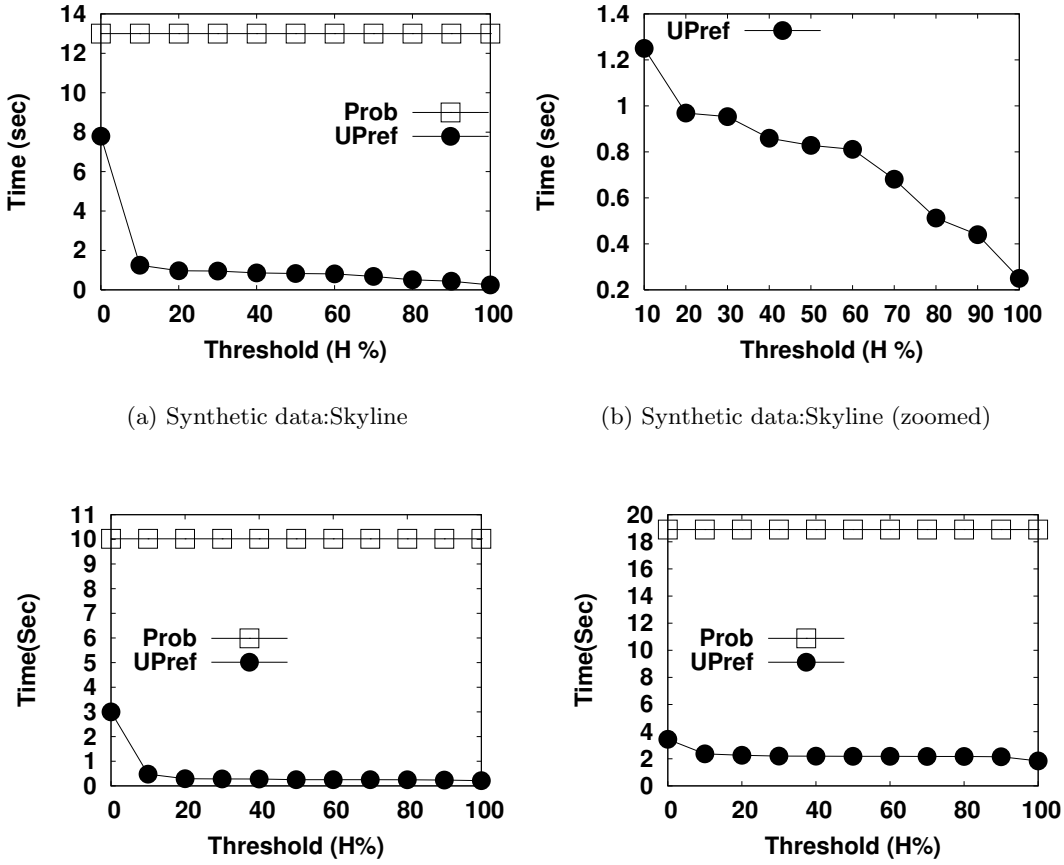
Figure 3.8a gives the effect of the various threshold values (0%, 50%, and 100%) on

UPref as dataset sizes increase. Even for the case of $H = 0\%$, UPref performance is three times better than that of Prob (given in Figure 3.7(a)). This speedup is mainly due to the *uncertainty reduction* operation as, even we could not prune many objects due to the very low value of $H$, we were still able to reduce the uncertainty ranges of several points in the data set. This results in a very simplified probability calculations. When $H = 100\%$, UPref runtime is 0.26 seconds, as it immediately prunes objects that are dependent on other objects found using *pairwise comparison*. For 50% threshold, the performance is close to 100% as the number of points with thresholds between 50% and 100% are close. Experiments for the NBA dataset exhibit similar performance (Figure 3.8b).

### 3.8.2 Effect of Threshold and Tolerance

Figure 3.9(a) gives the effect of increasing the threshold $H$ for our synthetic dataset. The speed up of UPref over Prob reaches up to 128 for a 100% threshold. As the performance of UPref is close for threshold values more than 10%, we plot Figure 3.9(b) that magnifies the behavior of UPref for threshold values of more than 10%. This shows that the performance of UPref increases linearly with the increase of the threshold. With a larger threshold, UPref filters more objects earlier, and hence exhibits a better execution time. Similarly, Figures 3.10(a) and 3.10(b) illustrate the behavior for the higher dimensional NBA data set. The speedup of UPref is still present, but increases at a slower rate than the synthetic case due to smaller dependency lists of each object.

Figure 3.11 gives the effect of increasing the tolerance ($\delta$) from 0.001% to 100% on the execution time (both dimensions are represented as log scale). It is always the case that the runtime of UPref is orders of magnitude faster than Prob as UPref refines *less* objects than that of Prob due to the filtering performed in Phase I. We could not plot the values for Prob for small tolerance values ($\delta < 0.01$) as it is very expensive. This shows that the uncertainty reduction and pairwise comparisons performed by UPref did a very good job filtering a lot of objects, thus we, could calculate almost exact probability, i.e., small $\delta$ for only few objects.




(a) Synthetic data:Skyline

(b) Synthetic data:Skyline (zoomed)

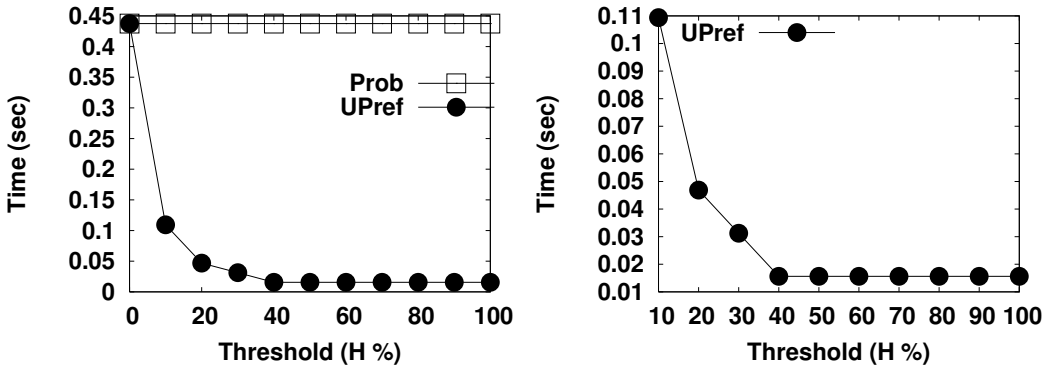
(c) Synthetic data:Top-K

(d) Synthetic data:Multi-objective

Figure 3.9: Threshold: Prob Vs UPref

### 3.8.3 Effect of Dimensionality

Figure 3.14(a) studies the effect of increasing the number of dimensions from two to eight on the total runtime (presented in log scale) for the UPref and Prob algorithms. For all dimensions, UPref has about an order of magnitude better performance than Prob. The execution time for both algorithms increases with the increase of the number of dimensions as the size of the preference list increases with the increase of the number of dimensions. For example, for a 50% threshold, the size of the *Preference* list increases from 5 for two dimensions to 7103 for eight dimensions. Thus, the time required by




(a) NBA (b) NBA (zoomed)

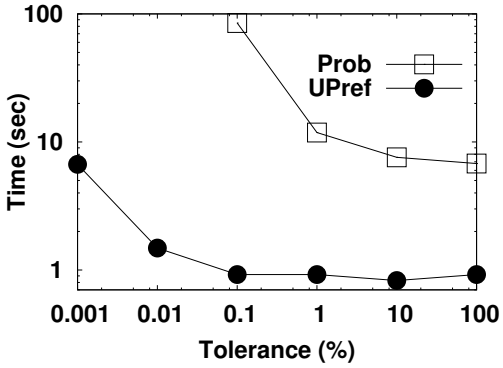
Figure 3.10: Threshold: Prob vs UPref

Phase I greatly increases with an increase in dimensionality. Likewise, the time required by Phase II increases due to the increased size of the *Preference* list.

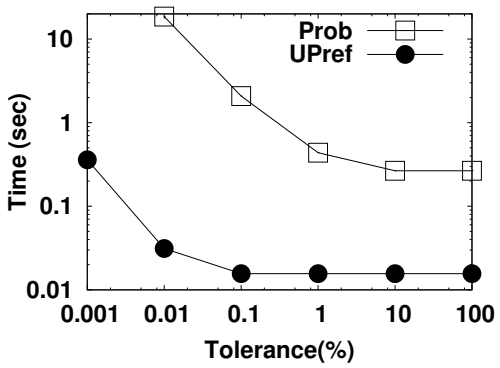
Figure 3.14(b) gives the runtime of UPref for different thresholds. Naturally, increasing the threshold results in a better time for all dimensions. A main reason is that as threshold increases, Phase I performs more filtering.

### 3.8.4 Size of the Uncertainty Range

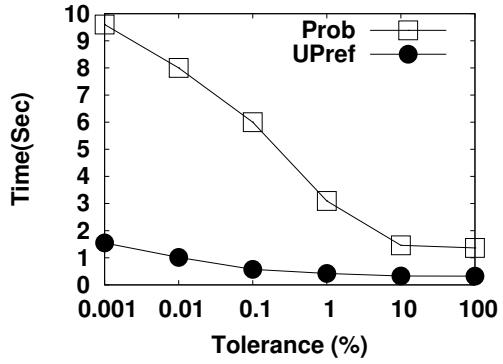
Figure 3.16 gives the performance of both UPref and Prob algorithms when increasing the uncertainty range from 5% to 40% of the space for both Synthetic and NBA data. It is clear that the UPref has superior performance with orders of magnitude better performance than Prob. The performance gain of UPref over Prob significantly increases with the increase in the uncertainty region. This indicates that UPref scales well with the increase of the uncertainty range while the performance of Prob significantly decreases. The main reason behind this behavior is that increasing the uncertainty range directly affects the probability that two objects overlap. This results in two contradictory effects: (1) increasing the processing time for Phase II as the average size of a dependency list increases, and (2) increasing the chance that an object is pruned in Phase I as a larger dependency list implies that on average, each object's probability of being a preferred object decreases. Since UPref employs Phase I, it makes use of the second effect to filter



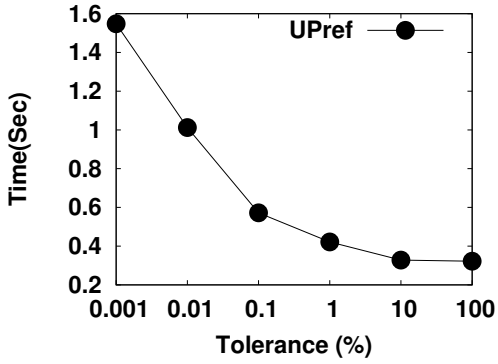

(a) Synthetic data:Skyline



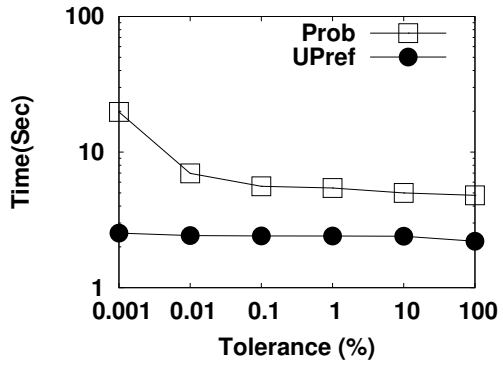

(b) NBA: Skyline



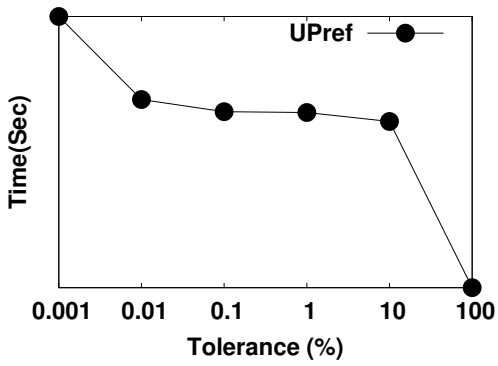

(c) Synthetic data:Top-K




(d) Synthetic data:Top-K (zoomed)




(e) Synthetic data:Multi-objective




(f) Synthetic data:Multi-objective (zoomed)

Figure 3.11: Tolerance: Prob vs. UPref

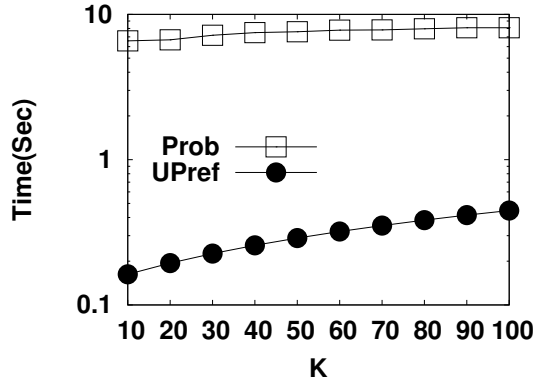
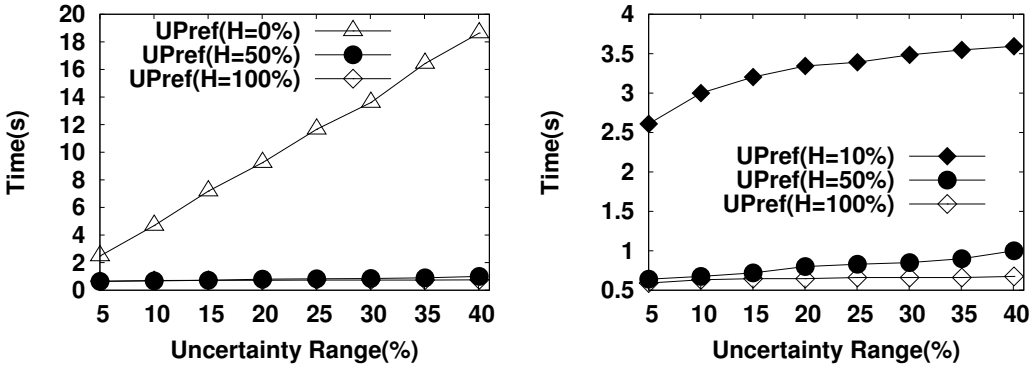



Figure 3.12: Top-k




(a) Synthetic data:Skyline

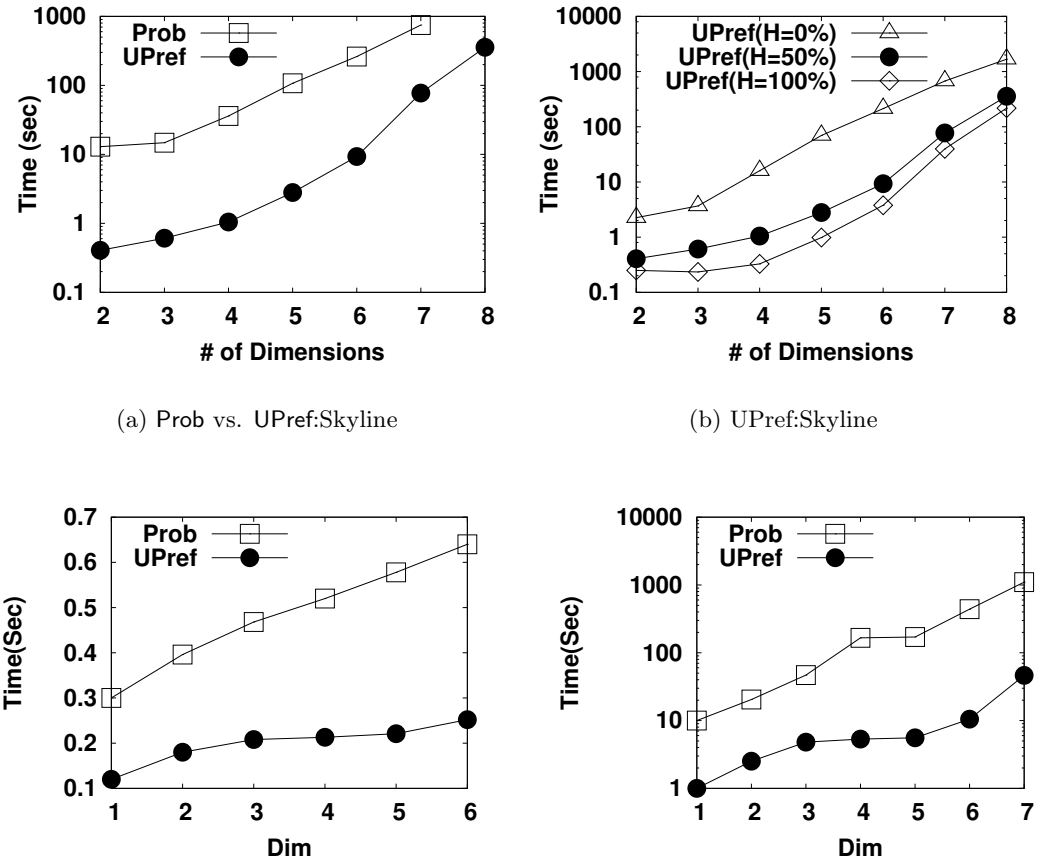
(b) NBA:Skyline

Figure 3.13: The size of uncertainty range

more points with the increase of the uncertainty range.

Figure 3.13(a) illustrates the behavior of UPref for different threshold values (0%, 50%, and 100%) on the Synthetic dataset as the size of the uncertainty region increases. For $H$=0%, the average size of dependency lists increases quickly, thus, the runtime also increases rapidly. Note that runtime of UPref with threshold 0% is 3.6 times faster than Prob (given in Figure 3.16) when the size of the uncertainty range is 40%. This speedup is due to the *uncertainty reduction* performed in Phase I of UPref. As the threshold value increases, the performance is stable with the increase of the overlap factor. We further illustrate the performance of UPref for threshold H=10%, 50% and 100% in

Figure 3.13(b).




(a) Prob vs. UPref:Skyline

(b) UPref:Skyline

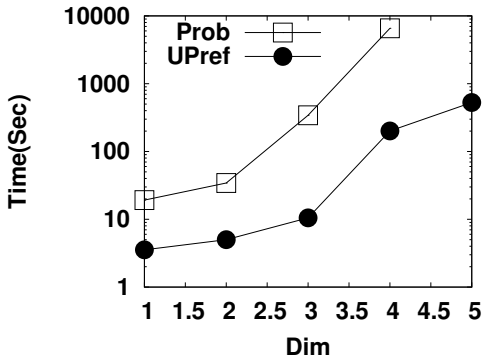
(c) Prob vs. UPref:Top-K

(d) Prob vs. UPref:Multi-objective

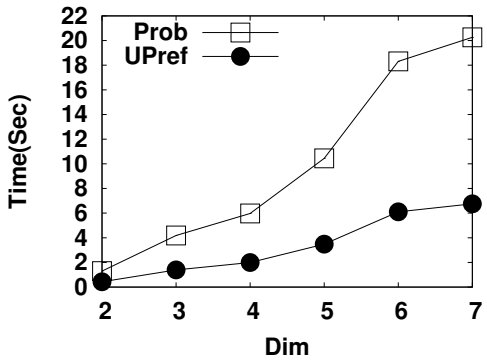
Figure 3.14: Effect of Certain Dimensions

### 3.8.5 Top-k

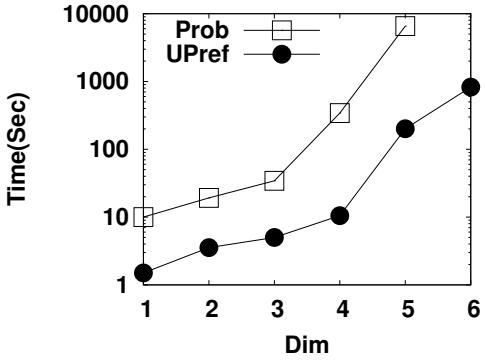
This experiment gives the effect of K on the running time of UPref over the synthetic dataset. For this experiment, only a single uncertain dimension is used as input to the top-k query. Figure 3.12 gives the runtime of UPref as the value of $k$ increases. The running time for both algorithms increases with the $k$, the performance of UPref is superior with an order of magnitude.




(a) Prob vs. UPref:Skyline




(b) Prob vs. UPref:Top-K

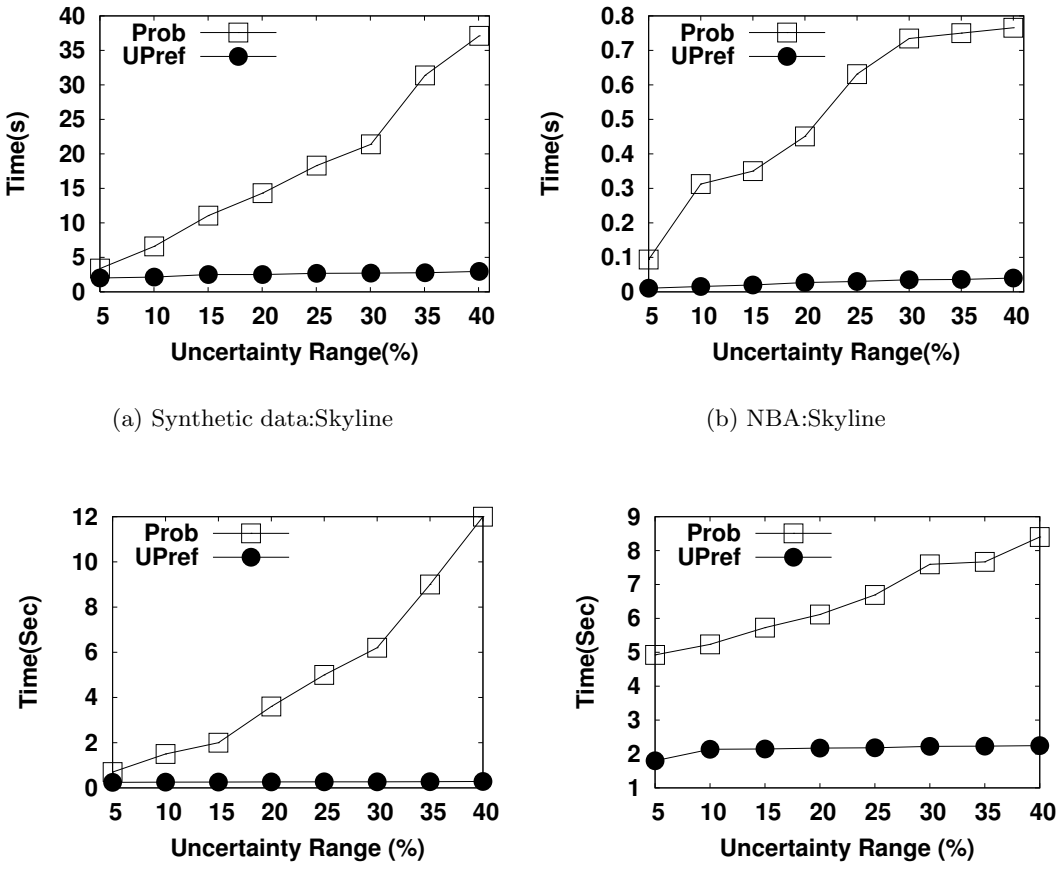



(c) Prob vs. UPref:Multi-objective

Figure 3.15: Effect of Uncertain Dimensions

### 3.8.6 Running Time Analysis for UPref

This final set of experiments analyze the effectiveness of filtering of Phase I of UPref, and also studies the time breakdown between Phase I and II with different threshold and tolerance values. Figures 3.17(a) and 3.17(b) illustrate the effectiveness of Phase I of UPref for uncertainty range sizes of 10% and 40%, respectively. Here, we vary the threshold value from 10% to 100% and compare the size of the *Preference* lists after Phase I (denoted by UPref) with the size of an *optimal* algorithm (bars) that filters points using an *exact* probability calculation. The difference between the sizes represents the




(a) Synthetic data:Skyline

(b) NBA:Skyline

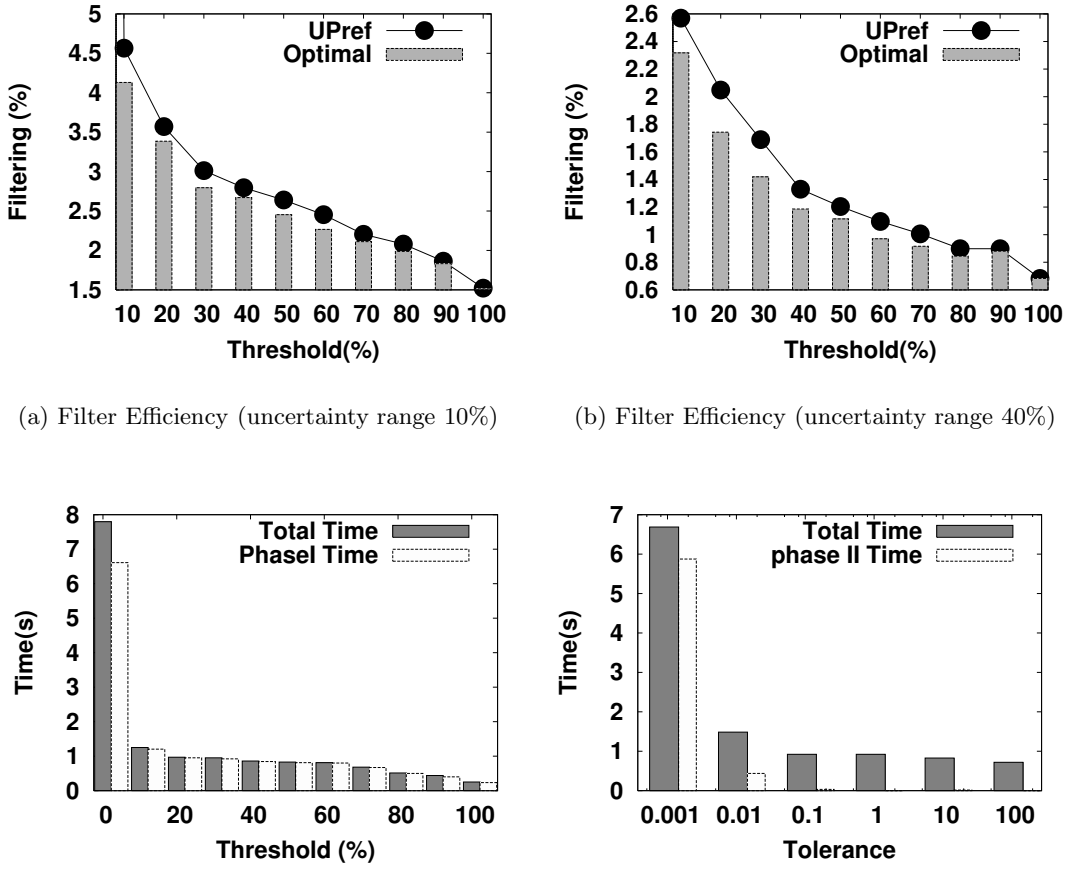
(c) Synthetic data:Top-k

(d) Synthetic data:Mulit-objective

Figure 3.16: The effect of the size of uncertainty range

number of false positives in the *Preference* list left by Phase I of UPref. We present the *Preference* list sizes (y-axis) as a percentage of the size of the *Preference* list when *no* filtering is used. With the increase of the threshold value, the number of false positives decreases rapidly for both uncertainty sizes of 10% and 40%.

Figure 3.17(c) gives the breakdown between Phase I and Phase II of UPref for different threshold values. We show the total time (i.e., the time for both Phase I and Phase II) with only the time of Phase I. For a threshold value of 0%, the total time is




(a) Filter Efficiency (uncertainty range 10%)

(b) Filter Efficiency (uncertainty range 40%)

(c) Threshold

(d) Tolerance

Figure 3.17: Analysis of UPref Algorithm for Skyline Preference function

significantly greater for larger thresholds. Also, the time of Phase I is the main contributor for the total run time. As the tolerance value only affects the time of Phase II, Figure 3.17(d) shows the total run time time with the time of Phase II. For low tolerance (i.e., 0.001%), the time of Phase II dominates the total time, as it needs to tightly bound the object preference probability. For larger tolerance values (e.g., 10%), Phase I is the main contributor to the total time.

## 3.9 Conclusion

We have defined preference queries over uncertain data, and proposed a novel, efficient framework to answer these preference queries. Query answers are probabilistic, where each object is associated with a probability value of being a preferred answer. Users can specify a probability threshold, that each preferred object must exceed, and a tolerance that defines the allowed error margin in probability calculation. We have described our framework in the context of *skyline*, *top-k* and *multi-objective* preference functions. We have proposed four methods to bound each object probability for being a preferred object, namely, we have proposed *uncertainty reduction*, *pairwise comparison*, *segmentation*, and *bound tightening*. Then, we presented a two-phase framework that encapsulates our four proposed methods together using a filter-refine approach. We extended our framework to support multi-dimensional and non-uniform pdf uncertain data. Extensive experiments show that our framework is scalable, efficient, and can tolerate wide uncertainty ranges.

# Chapter 4

# PrefJoin: An Efficient Preference-aware Join Operator

Preference queries are essential to a wide spectrum of applications including multi-criteria decision-making tools and *personalized* databases. Unfortunately, most of the evaluation techniques for preference queries assume that the set of preferred attributes are stored in only one relation, waiving on a wide set of queries that include preference computations over multiple relations. This chapter presents *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries over multiple relations. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively. An interesting characteristic of *PrefJoin* is that it tightly integrates preference computation with join hence we can early prune those tuples that are guaranteed not to be an answer, and hence it saves significant unnecessary computations cost. *PrefJoin* supports a variety of preference function including skyline, multi-objective and $k$-dominance preference queries. We show the correctness of *PrefJoin*. Experimental evaluation based on a real system implementation inside PostgreSQL [94, 95, 96] shows that *PrefJoin* consistently achieves from one to three

orders of magnitude performance gain over its competitors in various scenarios. In the next chapter, we extend this work to support uncertain data, represented in range.

## 4.1 Introduction

Preference queries are essential to a wide spectrum of applications including multi-criteria decision-making tools and personalized databases [14, 15]. Several preference functions have been proposed in the literature including *top-k* [1], *skylines* [2], *multi-objective* [5], *k-dominance* [6], *k-frequency* [7], and *ranked skylines* [8]. Given a set of multi-dimensional tuples, preference queries find a set of interesting tuples, i.e., tuples that are preferred to the user according to some preference function. An example preference query is *"find my best restaurants based on my preferences"* where user preferences for a restaurant can be minimal price and minimal distance.

Most of the research efforts for the preference query evaluation are designed to compute the preference set over a single relation (e.g., see [1, 8, 2, 43, 42, 97, 44, 98, 99, 100]). Unfortunately, such work can not be directly applied to a wide spectrum of preference applications and queries where the preferred attributes are stored in more than one relation. Consider, for example, a scenario where a user is looking for a vacancy destination. for simplicity assume that the user is only concerned about the hotel and the cruise. User preferences for the hotel are lower price, better rating, and closer to the beach and for the cruise are lower price, better rating, and shorter stay. Figure 4.1a gives information about hotels and cruises, stored in relations *Hotels* and *Cruises*, respectively. Assuming minimum rating is better, this user preference request can be represented by the following SQL query:

```
SELECT * from Hotels h, Cruises c
WHERE c.location = h.location
PREFERENCE h.price(min), h.rating(min),
h.beach_dist (min), c.price(min),
c.rating(min), c.days(min)
```

To answer this SQL query using existing single-table preference techniques, we first need to join the two input relations (i.e., *Hotels* and *Cruises*), and then apply the

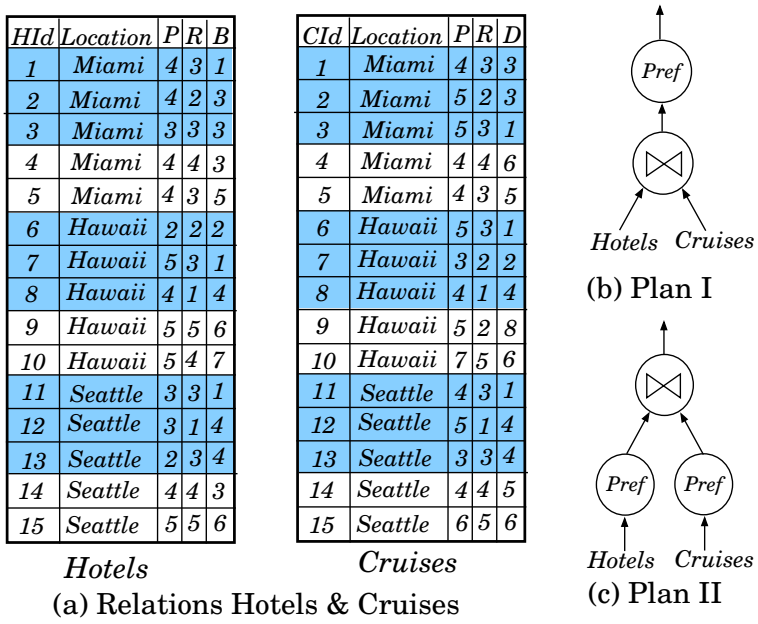



Figure 4.1: Motivating Example

preference query algorithm on the joined relation based on the location attribute, i.e., `Pref` $(Hotels \bowtie_{location} Cruises)$. Figure 4.1b shows such a query plan. Unfortunately, such an approach is very inefficient as it completely isolates the preference functionality from the join operator. As a result, the join operation will produce too many tuples that have no chance of being preferred tuples. Another approach is to push the single-relation preference operation `Pref` before the join operator, as depicted in the query plan in Figure 4.1c. However, this is simply incorrect as the preference is not distributive over the join, i.e., `Pref` $(Hotels \bowtie_{location} Cruises) \neq$ `Pref`$(Hotels) \bowtie_{location}$`Pref`$(Cruises)$. For example, in Figure 4.1, if the preference function is a skyline query, the hotel represented by tuple (2,$\underline{Miami}$,4,2,3) is dominated by hotel (6,$\underline{Hawaii}$,2,2,2), hence it would not proceed to the join operator. Similarly, cruise (3,$\underline{Miami}$,5,3,1) is dominated by cruise (11,$\underline{Seattle}$,4,3,1). However, the joined tuple (2,$\underline{Miami}$,4,2,3, 3,$\underline{Miami}$,5,3,1) is not dominated by any other tuples and hence, it is a valid answer for the SQL query. Note that this query plan can be correct only when the join is a cartesian product.

Very recently, few research efforts have started to address preference queries over multiple relations [33, 34, 101, 102], however such work either focuses on only the skyline

preference function [33, 101, 102] or provides a preliminary generic solution for a first cut generic preference query engine with no particular focus on the join operation [34]. Furthermore, two of these research efforts about skyline queries are mainly relying on the existence of index structures and are geared towards generating progressive results [101, 102].

In this chapter, we propose *PrefJoin*; an efficient preference-aware join query operator. *PrefJoin* is generic for a wide variety of preference functions, does not assume the existence of any index structure, and achieves orders of magnitude performance over previous approaches [33, 34]. The main goal of *PrefJoin* is to make the join operation aware of the required preference functionality, and hence the join operation would be able to early prune those tuples that have no chance of being a preferred tuple without actually doing the join operation. The *PrefJoin* algorithm consists of four phases, namely, *Local Pruning*, *Data Preparation*, *Joining*, and *Refining*. The *Local Pruning* phase filters out, from each input relation those tuples that are guaranteed not to be in the final preference set. The *Data Preparation* phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phases. The *Joining* phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the *Refining* phase finds the *final* preference set from the output of the joining phase.

*PrefJoin* is presented as a general framework that can support a wide variety of preference functions, though we only present three cases in this chapter, namely, *skyline* [2], *multi-objective* [5], and *k-dominance* [6]. Experimental analysis of *PrefJoin*, implemented in PostgreSQL[103], shows from two to three orders of magnitude improvement over existing solutions[34, 33]. The rest of this chapter is organized as follows: Section 4.2 formulates the problem. Section 4.3 highlights related research efforts. Section 4.4 presents the *PrefJoin* framework with three case studies. Section 4.5 discusses the effect of the join order on the performance of the proposed algorithm. Section 4.6 proves the correctness of the proposed algorithm. Section 4.8 gives experimental analysis. Finally, Section 4.9 concludes the chapter.

## 4.2 Problem Formulation

Without loss of generality, we assume that all dimension values have a total order in which smaller values are better.

**Problem Formulation.** Given: (1) $m$ input relations $R_1, R_2, \ldots, R_m$, (2) Equality join condition over $R_1$ to $R_m$, (3) A set of preference attributes $\mathcal{P}$; such that $\forall\ p \in \mathcal{P}$, $\exists\ i$ s.t $p \in R_i$ and $\forall\ R_i$, $R_i \cap \mathcal{P} \neq \phi$, and (4) A preference method $\mathcal{M}$. A preference query $Q$ finds tuples from $R_1 \bowtie R_2 \bowtie \ldots\ R_m$, that are preferred with respect to $\mathcal{P}$ and $\mathcal{M}$.

Applying this formulation to the SQL query given in Section 4.1 gives: (1) Two input relations *Hotels H* and *Cruises C*, (2) The equality join condition is *H.location* = *C.location*, (3) Six preference attributes, *H.price*, *H.rating*, *H.beach_dist*, *C.price*, *C.rating*, and *C.days*, and (4) A skyline preference method. The SQL query finds those tuples, on the form (*HID*, *CID*, *Preference attributes*) that are skylines over the six preference attributes from $H \bowtie C$.

## 4.3 Related work

Due to their applicability, preference queries have received great interest ever since their introduction into databases. Several preference functions have been proposed in the literature including *skyline* [2, 43, 42, 97, 44, 99],*nearest-neighbors* [77, 78, 79, 80, 81], *multi-objective* [5], *Top-K* [104, 105, 106, 107, 108, 109, 1, 110], *k-dominance* [6], *k-frequency* [7], *ranked skylines* [8], *spatial skyline* [4], *k representative skylines* [9], *distance-based dominance* [10], *$\epsilon$-skyline* [11], and *top-k dominance* [12]. With the exception of very recent research [33, 34, 101, 102], all research efforts in preference query processing rely on the assumption that all input data reside in only one relation with no direct extension to support the case where preference attributes are scattered over more than one relation. Hence, the only solution is to completely join the input relations, then apply the preference method on the top of the join result, which is a very expensive solution.

Table 4.1 gives a taxonomy of existing work for preference queries over multiple relations. For completeness, we include a special case of join, where *one* relation is presented as a collection of sorted lists[111]. Each sorted list contains tuples on the

| Algorithm | Query | Join Condition | Sorted Lists | Operator |
|---|---|---|---|---|
| TA & NRA [111] | Top-k Join | Key 1:1 | √ | — |
| KLEE [112] | Top-k Join | Key 1:1 | √ | — |
| J* [113] | Top-k join | General Equality | √ | — |
| NRA-RJ [114] | Top-k join | Key 1:1 | √ | √ |
| Rank-Join [115] | Top-k join | General Equality | √ | √ |
| Index [45] | Skyline join | Key 1:1 | √ | — |
| Multi-relational skyline [33] | Skyline Join | General Equality | — | √ |
| Distributed Skyline Join [116] | Skyline Join | General Equality | — | — |
| FlexPref [34] | Preference Join | General Equality | — | √ |
| *PrefJoin* | Preference Join | General Equality | — | √ |

Table 4.1: Related work

form $(id, value)$ and is sorted based on the *value* attribute. Table 4.1 divides these research efforts with respect to: (a) Query type (e.g., Top-k join or skyline join), (b) Join condition (e.g., a general equality join condition or key only join), (c) Sorted lists (i.e., the input of the query must be in the form of sorted lists), and (d) Operator.

As it can be seen from table 4.1, existing work for preference join can be divided into Top-k join, skyline join, and preference join. Furthermore, Top-k Join can be divided into: top-k over sorted lists (e.g., [111, 114]), approximate Top-k join in a distributed environment [112], and Top-k with general condition join [113, 115]. Other research efforts (e.g., [31, 32]), not shown in the table, compute the top-k join query over uncertain data. Other research effort for top-k included [117, 118]. A survey for top-k query processing is [119]. On the other hand, research efforts to compute skyline and preference join are limited to recent efforts [33, 34, 101, 102] (as seen from the table) which either focus on only the skyline preference function [33, 101, 102] or provide a preliminary generic solution for a general preference query engine with no particular focus on the join operation [34]. Furthermore, two of these research efforts mainly rely on the existence of index structures and are geared towards generating progressive results, turning them not optimized for a full join result [101, 102]. Other research efforts compute skyline join for a single relation, represented as set of sorted list as in [45], a centralized environment[33, 34, 120] or in a distrusted environment [116]. Most

of approaches designed for skyline and preference join (with except of [45]) support a generalized join condition.

There are two levels to embed the preference processing into database engine, as it can either implemented in application level as a layer in top of DBMS or as an operator inside the engine. When the operator is implemented inside the DBMS, the engine is well-aware of the existing of the operator and generates query plans that may use the operator. From the table, the algorithms that are presented as operators are [114, 115, 33]. Our proposed algorithms *PrefJoin* is implemented as an operator inside the PostgreSQL engine.

Our proposed approach *PrefJoin* distinguishes itself from its competitors [33, 34, 101, 102] as it has the following characteristics: (1) Unlike [33, 101, 102], *PrefJoin* is not limited to skyline queries, but it is generic to support a wide variety of preference queries; making it suitable for commercial database systems as it requires small code footprints for various preference functions, (2) Unlike [101, 102], *PrefJoin* does not assume the existence of any indexing data structure making nor geared towards producing progressive output, instead *PrefJoin* aims to support basic join queries that still lack the awareness of various preference functions, (3) On top of all other approaches, *PrefJoin* does not only employ elegant early punning techniques, but also, utilizes some meta information about input tuples to early decide whether two tuples should be joined together. With this, based on actual implementation inside PostgreSQL, *PrefJoin* achieves two to three orders of magnitude better performance over global skyline [33] for *skyline* queries and over FLexPref [34] for *skyline*, *k-dominance*, and *multi-objective* queries.

## 4.4 PrefJoin: A Preference-Aware Join Operator

In this section, we present our proposed algorithm, *PrefJoin*; an efficient preference-aware join query operator for a wide variety of preference functions. In particular, we focus, in this chapter, on *skyline* [2], *k-dominance* [6] and *multi-objective* [5] preference functions. However, other preference functions that include *top-k dominating* [12] and *k-frequency*[7] can be also supported. The main goal of *PrefJoin* is to make the join operator aware of the required preference functionality, and hence the join operation would be able to prune those tuples that have no chance of being a preferred tuple with

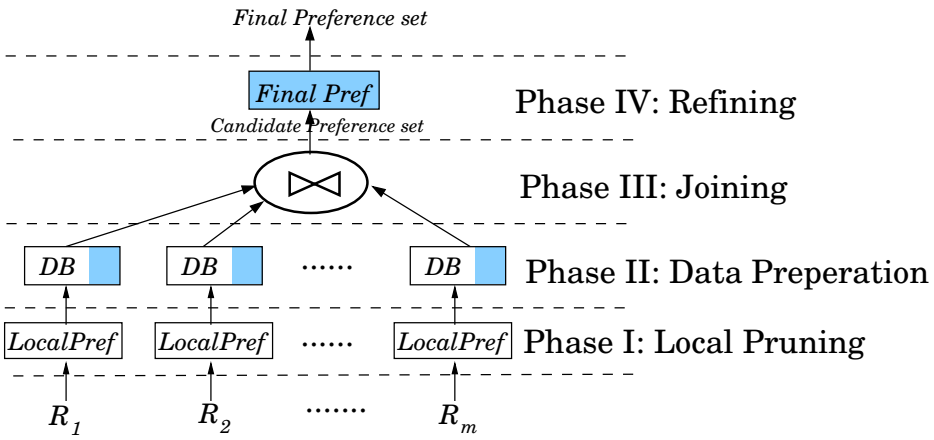



Figure 4.2: Phases of *PrefJoin* Algorithm

minimal computational overhead. The *PrefJoin* algorithm consists of four phases, as depicted in Figure 4.2, namely, *Local Pruning*, *Data Preparation*, *Joining*, and *Refining*. These phases use three preference functions $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ that are chosen carefully based on $\mathcal{P}$. Table 4.2 gives *possible* choices of $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for *skyline*, *k-dominance*, *multi-objective*, *top-k*, and *K-Frequency* preference functions. The choice of break-down for the input preference function $\mathcal{P}$ into $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ is arbitrary, and is the responsibility of the framework users to supply these functions. It is important to note that the break-down of a preference function $\mathcal{P}$ is not unique. For example, for any preference predicate $\mathcal{P}$, a naive and correct way of breaking $\mathcal{P}$ down is to set $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwaise}$ to null, and $\mathcal{P}_{refine}$ to $\mathcal{P}$. The *Local Pruning* phase filters out, using $\mathcal{P}_{local}$, from each input relation, those tuples that are guaranteed not to be in the final preference set. The *Data Preparation* phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phase. The *Joining* phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the *Refining* phase uses $\mathcal{P}_{refine}$ to find the *final* preference set from the output of the joining phase.

Sections 4.4.1 - 4.4.4 discuss the four phases of *PrefJoin* in a generic way that can support a wide variety of preference functions. Section 4.4.5 gives the pseudo-code for the *PrefJoin* algorithm. Section 4.4.6 gives three case studies of *PrefJoin*, namely, *skyline*, *k-dominance*, and *multi-objective* preference functions. Cost analysis

for *PrefJoin* is presented in Appendix 4.7. For simplicity, we limit our presentation of *PrefJoin* to the in-memory version. This is not a limitation for our contribution for two main reasons: (a) *PrefJoin* needs minimal memory requirements. Phase I needs only one in-memory hash bucket at a time, while Phases II and III can work with only two in-memory hash buckets at a time, and (b) memory spilling for hash-based joins is an independent research topic [121, 122]. The ideas for memory spilling can be orthogonally applied to *PrefJoin*, and its competitors (i.e., *GS* [33], and *FlexPref* [34]) without affecting their main operations nor performance trends. Other techniques for main memory join have been proposed [123, 124].

### 4.4.1 Phase I: Local Pruning

Phase I filters out those tuples, from each input relation, that are guaranteed to be not in the final preference answer. The output of Phase I, i.e., the set of non-filtered tuples, is the local preference set $\mathcal{LP}(R_i)$ for each input relation $R_i$, which is defined as the set of tuples such that each tuple $t \in \mathcal{LP}(R_i)$ is a *preferred* tuple over all tuples in $R_i$ with the same join attributes values. For example, Figure 4.1a highlights the tuples in the *local* preference set, for the *Hotels* and *Cruises* relations using the skyline preference method and the *location* attribute as the joining attribute. Phase I has the following two main steps: (a) *Hashing*, where Phase I scans each input relation $R$ and utilizes a hash function $h$, based on the equality join attributes, to hash each tuple $t \in R_i$ to its corresponding hash bucket. For simplicity, we build a *separate* hash bucket $B$ for each value of the equality join attributes. (b) *Local preference computation for each relation* $R_i$, where Phase I employs a preference function $\mathcal{P}_{local}$ over the set of tuples in each hash bucket $B \in R_i$ separately. It is important to note here that $\mathcal{P}_{local}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{local}$ depends on $\mathcal{P}$. For example, per Table 4.2 and as will be detailed in Section 4.4.6, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective* preference functions, $\mathcal{P}_{local}$ would be *skyline*, *skyline*, or *multi-objective*, respectively.

The main idea of Phase I is that any tuple $t \in R_i$ that is not preferred, with respect to $\mathcal{P}_{local}$, over other tuples in $R_i$ with the same value in the equality join attribute, should be filtered out as it has no chance of being preferred in $R_i \bowtie R_j$ with respect to $\mathcal{P}$. As $t$ is not preferred to $\mathcal{P}_{local}$, there must be another tuple $t'$ in the same hash bucket $B$ of $t$ that is better than $t$. This means that when joining $R_i$ with relation $R_j$,

the result of $t' \bowtie r_j$, $r_j \in R_j$, will be preferred over $t \bowtie r_j$. Such early pruning of $t$ saves the overhead of considering $t$ at later steps.

| | $\mathcal{P}_{local}$ | $\mathcal{P}_{pairwise}$ | $\mathcal{P}_{refine}$ |
|---|---|---|---|
| Skyline [2] | *Skyline* | *Skyline* | *null* |
| K-dominance [6] | *Skyline* | *Skyline* | *K-dominance* |
| Multi-objective [5] | *Multi-Objective* | *Multi-Objective* | *Multi-objective* |
| Top-K [1] | *Top-k* | *Sorting* | *Rank-aware join* |
| K-Frequency [7] | *K-Frequency* | *null* | *K-Frequency* |

Table 4.2: Possible setting of $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for some preferences functions.

### 4.4.2 Phase II: Data Preparation

Phase II takes, as input, the local preference set, $\mathcal{LP}(R_i)$, for each relation $R_i$, produced from Phase I and passes it to Phase III along with a set of information, termed *Dominating hash buckets*, $DB(t)$, associated with each tuple $t \in \mathcal{LP}(R_i)$. Such information will be used later in Phase III to avoid producing unnecessary joined tuples. The main idea of Phase II is to associate with each local preference tuple $t$, produced from Phase I, the set of its dominating hash buckets, $DB(t)$, i.e., the set of hash buckets in $R$ that contains tuples preferred over $t$ with respect to preference function $\mathcal{P}_{pairwise}$. Same as $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{pairwise}$ depends on $\mathcal{P}$. For example, per Table 4.2 and as will be detailed in Section 4.4.6, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective*, $\mathcal{P}_{pairwise}$ would be *skyline*, *k-dominance*, or *skyline*, respectively.

For a local preference tuple $t$ of bucket $B$ in relation $R$, $DB(t)$ is computed by comparing $t$ with the first tuple $t'$ of each hash bucket $B'$ in relation $R$, where $B' \neq B$. Three cases may occur:

- *Case 1: $t'$ is preferred over $t$ with respect to $\mathcal{P}_{pairwise}$.* In this case, we add bucket $B'$ to $DB(t)$ as this means that there is a tuple in $B'$ that is preferred over $t$. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in $B$ can be preferred over $t'$, thus no further preference checks are needed for other tuples in $B'$. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare $t$ with the next tuple in $B'$, and act accordingly based on our three cases.

- *Case 2: $t$ is preferred over $t'$ with respect to $\mathcal{P}_{pairwise}$.* In this case, we add bucket $B$ to $DB(t')$ as this means that there is a tuple in $B$ that is preferred over $t'$. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in $B'$ can be preferred over $t$, thus no further preference checks are needed for other tuples in $B'$. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare $t$ with the next tuple in $B'$, and act accordingly based on our three cases.

- *Case 3: Neither $t$ is preferred over $t'$ nor $t'$ is preferred over $t$.* In this case, we do not change neither $DB(t)$ nor $DB(t')$. We proceed with next tuple from $B'$, and act accordingly based on our three cases.

### 4.4.3 Phase III: Joining

Phase III takes, as input, the local preference set $\mathcal{LP}(R_i)$ where each tuple $t{\in}\mathcal{LP}(R_i)$ is associated with a set of dominating hash buckets, $DB(t)$. Phase III uses $DB(t)$ to decide which local preference tuples $t_i \in \mathcal{LP}(R_i)$ and $t_j \in \mathcal{LP}(R_j)$ should be joined together, to produce the *candidate* preference set, denoted as $Candidate_{pref}$. This means that by only consulting the sets of dominating hash buckets, Phase III is able to decide if the joined tuple is a *candidate* preference tuple or not, rather than performing the join operation followed by a preference check. Basically, two tuples $r$ and $s$ from relations $R$ and $S$ that satisfy the equality join condition will be joined together only if $DB(r) \cap DB(s){=}\phi$. Unlike all other phases, this phase does not directly depend on the preference function $\mathcal{P}$, as it always has the same execution regardless of $\mathcal{P}$.

For simplicity, we assume two input relations $R$ and $S$, then we extend our ideas to support arbitrary number of input relations. Consider two local preference tuples $r$ and $s$ from corresponding hash bucket $B$ of relations $R$ and $S$, respectively. Two cases may arise:

- *Case 1: $DB(r) \cap DB(s) \neq \phi$, i.e., the sets of dominating hash buckets for $r$ and $s$ are overlapping.* In this case, we are sure that the joined tuple $t = r \bowtie s$ will not be preferred, hence, we avoid joining $r$ and $s$. To illustrate, consider bucket $B' \in DB(r) \cap DB(s)$. There must be tuple $r' \in B'$, such that $r'$ is preferred over $r$. Similarly, there must be $s' \in B'$, such that $s'$ is preferred over $s$. This means that the tuple $t' = r' \bowtie s'$ must be preferred over $t{=}r \bowtie s$.

- *Case 2: $DB(r) \cap DB(s) = \phi$, i.e., the sets of dominating hash buckets for $r$ and $s$ are disjoint.* In this case, the joined tuple $t = r \bowtie s$ is a *candidate* preference tuple, i.e., it is a preferred tuple by separately considering attributes in relations $R$ and $S$. Hence, we perform the join and produce tuple $t$.

The same idea is generalized to $m$ input relations $R_1$, $R_2$ ,..., $R_m$, by joining the local preference tuples $t_1$, $t_2$, ..., $t_m$ from $R_1$, $R_2$, ..., $R_m$ only if $DB(t_1) \cap DB(t_2) \cap \ldots \cap DB(t_m) = \phi$. Hence, consulting the sets of dominating hash buckets is sufficient to check if the to be joined tuple is a *candidate* answer for the preference query.

### 4.4.4 Phase IV: Refining

Phase IV takes, as input, the *candidate* preference set, $Candidate_{pref}$, produced from the *joining* phase, and finds the final preference answer for the preference function $\mathcal{P}$. Simply, Phase IV employs a preference function $\mathcal{P}_{refine}$ over the set of tuples in the candidate preference set produced from Phase III. Each tuple $t$ that is preferred with respect to $\mathcal{P}_{refine}$ is a final preference tuple for $\mathcal{P}$. It is important to note that $\mathcal{P}_{refine}$ does not have to be the same as $\mathcal{P}$, yet the choice of $\mathcal{P}_{refine}$ depends on $\mathcal{P}$. Specifically, $\mathcal{P}_{refine}$ is chosen to apply the preference computations that could not be pushed before the joining phase. For example, per Table 4.2 and as will be detailed in Section 4.4.6, if $\mathcal{P}$ is a *skyline*, *k-dominance*, or *multi-objective*, $\mathcal{P}_{refine}$ would be *null*, *k-dominance*, or *multi-objective*, respectively.

### 4.4.5 *PrefJoin*: Pseudocode

Algorithm 7 gives the pseudo code of the *PrefJoin* algorithm, presented for two relations $R$ and $S$, for simplicity. The input to the algorithm is the two input relations $R$ and $S$ along with the three preference functions $\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$ and $\mathcal{P}_{refine}$ that are set based on the desired preference function $\mathcal{P}$, i.e., per Table 4.2 for *skyline*, *k-dominance*, and *multi-objective* preference functions. The algorithm starts by executing Phase I, i.e., building the hash buckets and computing the *local* preference sets $\mathcal{LP}(\mathcal{R})$ and $\mathcal{LP}(\mathcal{S})$ for the input relations $R$ and $S$, respectively. This is achieved by applying the preference function $\mathcal{P}_{local}$ over each hash bucket in both input relations (Lines 3 to 10 in Algorithm 7). Then, we proceed to Phase II, where we compute the set of

**Algorithm 7** PrefJoin

1: **Function** PrefJoin(*Relation* $R$, *Relation* $S$,$\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$,$\mathcal{P}_{refine}$)
2: /* Phase I */
3: $\mathcal{LP}(\mathcal{R}) \leftarrow \phi$; $\mathcal{LP}(\mathcal{S}) \leftarrow \phi$
4: Build hash buckets for relations $R$ and $S$
5: **for each** Hash Bucket $B$ in relation $R$ **do**
6: $\mathcal{LP}(\mathcal{R}) \leftarrow \mathcal{LP}(\mathcal{R}) \cup$ ApplyPreferenceFunction ($B$, $\mathcal{P}_{local}$)
7: **end for**
8: **for each** Hash Bucket $B$ in relation $S$ **do**
9: $\mathcal{LP}(\mathcal{S}) \leftarrow \mathcal{LP}(\mathcal{S}) \cup$ ApplyPreferenceFunction ($B$, $\mathcal{P}_{local}$)
10: **end for**
11: /* Phase II*/
12: **for each** local preference tuple $r \in \mathcal{LP}(R)$ **do**
13: $DB(r) \leftarrow$ The set of hash buckets in $R$ that include preferred tuple(s) over $r$ with respect to $\mathcal{P}_{pairwise}$ preference function
14: **end for**
15: **for each** local preference tuple $s \in \mathcal{LP}(S)$ **do**
16: $DB(s) \leftarrow$ The set of hash buckets in $S$ that include preferred tuple(s) over $s$ with respect to $\mathcal{P}_{pairwise}$ preference function
17: **end for**
18: /* Phase III*/
19: $Candidate_{pref} \leftarrow \phi$
20: **for** each pair of tuples ($r$,$s$) where $r$ in hash bucket B in $R$ and $s$ in the corresponding hash bucket $B$ in $S$ **do**
21: **if** $(DB(r) \cap DB(s) = \phi)$ **then**
22: Add $r \bowtie s$ to $Candidate_{pref}$
23: **end if**
24: **end for**
25: /* Phase IV*/
26: **if** $\mathcal{P}_{refine}$ = Null **then**
27: **return** $Candidate_{pref}$
28: **else**
29: **return** ApplyPreferenceFunction ($Candidate_{pref}$,$\mathcal{P}_{refine}$)
30: **end if**

dominating hash buckets $DB(r)$ for each tuple $r \in \mathcal{LP}(\mathcal{R})$ and $DB(s)$ for each tuple $s \in \mathcal{LP}(\mathcal{S})$. $DB(r)$ and $DB(s)$ are computed as the set of hash buckets in $R$ and $S$ that include preferred tuple(s) over $r$ and $s$, respectively, with respect to $\mathcal{P}_{pairwise}$ preference function (Lines from 12 to 17 in Algorithm 7). Then, we proceed to Phase III, as we initialize the candidate preference set, $Candidate_{pref}$ to be empty. We iterate over each pair of tuples ($r$,$s$) where $r$ in hash bucket B in $R$ and $s$ in the corresponding hash bucket $B$ in $S$. In this iteration, we compute $r \bowtie s$, and add its result to $Candidate_{pref}$ only if $DB(r) \cap DB(s) = \phi$ (Lines 19 to 24 in Algorithm 7). Finally, in Phase IV, if $\mathcal{P}_{refine}$ is null, we return $Candidate_{pref}$ as the final answer for preference function $\mathcal{P}$, otherwise, we apply the preference function $\mathcal{P}_{refine}$ over all entries in $Candidate_{pref}$ to produce the final answer for $\mathcal{P}$ (Lines 26 to 30 in Algorithm 7).

### 4.4.6 Case Studies

In this section, we show the strength of *PrefJoin* by giving three diverse case studies, namely, *skyline* [2], *multi-objective* [5] and *k-dominance skyline* [6]. Table 4.2 summarizes the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for these preference functions. For space constraints, we use the SQL query presented in Section 4.1, and relations *Hotels* and *Cruises* presented in Figure 4.1 as a running example. Generally speaking, we set $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ to $\mathcal{P}$.

#### Skyline

The *skyline* preference method returns tuples in a data set that are not dominated by (i.e., not strictly worse than) any other tuple in the data. Formally, given a dataset $\mathcal{D}$ of $l$-dimensional tuples, a *skyline* query finds each tuple $t$, such that $\nexists\ t' \in \mathcal{D}$; s.t., $t'.p_i$ is better than or equal to $t.p_i$, for $1 \leq i \leq l$ and $t'.p_m$ is strictly better than $t.p_m$ for at least one dimension $m$. To support *skyline* queries within *PrefJoin*, we set $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwise}$ to *Skyline*. Interestingly, we set $\mathcal{P}_{refine}$ to null, as each *candidate* preference tuple produced from Phase III is guaranteed to be a *final* preference function. This holds because the each tuple $t_c$=$r \bowtie s$ in the *candidate* preference set could not be dominated. Tuples that are preferred to tuple $r$, over preference attributes in $R$, are stored in hash buckets $DB(r)$. Similarly, those tuples that dominate $s$, over preference attributes in $S$, are only stored in the hash buckets $DB(s)$. Since $DB(r)$ and $DB(s)$

are disjoint, for each *candidate* preference tuple, it is impossible to find a single joined tuple that dominates $t$ in both relations $R$ and $S$. Thus, $r \bowtie s$ can not be dominated for *skyline* preference query, and it is a confirmed final answer. (A formal correctness proof is presented in Section 4.6).

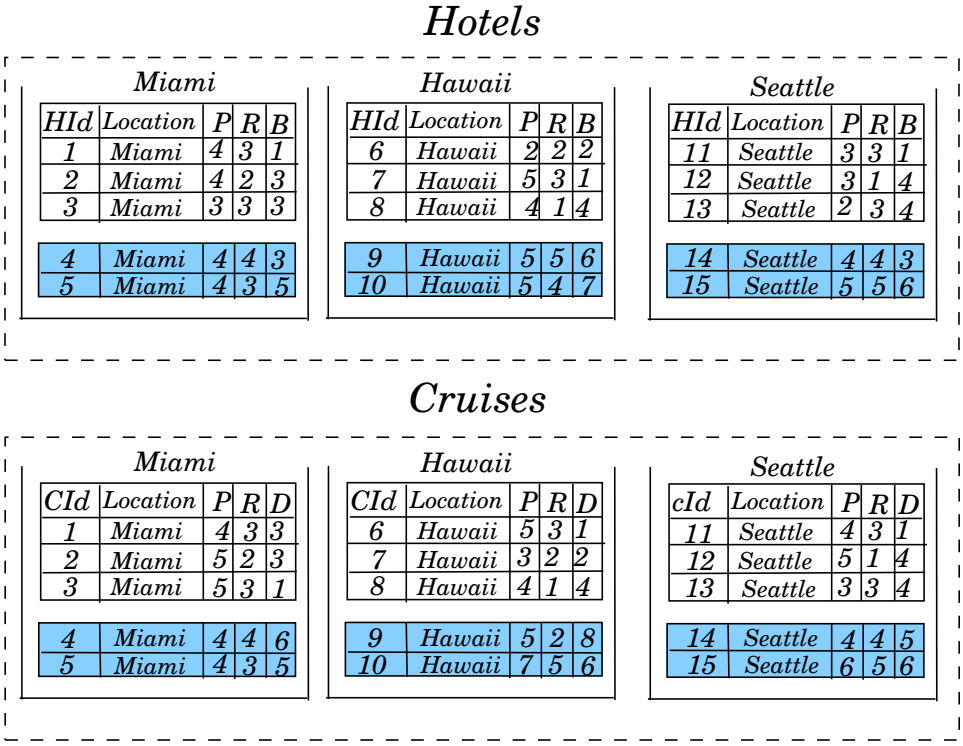



Figure 4.3: Phase I for *Skyline*, *multi-objective*, and *k-dominance* Queries

**Example.** We apply *PrefJoin* algorithm for *skyline* preference function as follow:

*Phase I.* Figure 4.3 shows the hash buckets for the input relations, *Hotels* and *Cruises*, presented in Figure 4.1 where three hash buckets are built as one for each value of the location attribute, i.e., *Miami*, *Hawaii*, and *Seattle*. The set of discarded tuples are highlighted for each bucket in the two input relations. Other tuples represents the local preference $\mathcal{LP}(R)$, which is the output of Phase I. The hotel tuples $h_9$=(9,*Hawaii*,5,5,6) and $h_{10}$ = (10,*Hawaii*,5,4,7) are locally dominated by hotel $h_6$=(6, *Hawaii*,2,2,2), hence, they are not in the local preference set of Hawaii hash bucket. We can see that joining $h_9$ or $h_{10}$ with any cruise in *Hawaii* (i.e., $c_6$ to $c_{10}$) will be

dominated by joining $h_6$ with the same cruise. *Phase II.* Figure 4.4 gives the end result of Phase II after computing $DB(t)$ for all tuples in the *Hotels* and *Cruises* relations where $M$, $H$, and $S$ refer to hash buckets *Miami*, *Hawaii*, and *Seattle*, respectively. To compute the set of dominating hash buckets for hotel $h_1$ = (1,$\underline{M}$,4,3,1), we compare $h_1$ with hotels in $H$ and $S$ hash buckets. For the first $H$ hotel $h_6$ = (6,$\underline{H}$,2,2,2), neither $h_1$ dominates $h_6$ nor $h_6$ dominates $h_1$ (Case 3), so we do not change the set of dominating hash tuples for $h_1$ and $h_6$. Then, we proceed with $h_7$. We find that $h_1$ dominates tuple $h_7$ (Case 2). Hence, we add $M$ to $DB(h_7)$. As no other hotel within $H$ hash bucket can dominate $h_1$, we proceed with hotels from $S$ hash bucket. Since $h_{11}$ dominates $h_1$ (Case 1), we add $S$ to $DB(h_1)$. Then, there is no need to continue checking other hotels in $S$ as none of them can be dominated by $h_1$. *Phase III.* From Figure 4.4, the set of dominating hash buckets for $h_1$ is $DB(h_1)$={S}. Also for cruise $c_1$, we have $DB(c_1)$= {H,S}. Since $DB(h_1) \cap DB(c_1) \neq \phi$, there is no need to join $h_1$, $c_1$. Then, we proceed with the next cruise $c_2$ with $DB(C_2)$={H}. Since $DB(h_1) \cap SB(c_2) = \phi$, we do the actual join and add the results of Phase III. Figure 4.5a gives the *candidate* preference set.

*Phase IV.* No computations are needed in this phase, as each *candidate* preference tuple is guaranteed to be in the final answer.

*Hotels*

*Miami*

| HId | location | P | R | B | DB |
|---|---|---|---|---|---|
| 1 | Miami | 4 | 3 | 1 | S |
| 2 | Miami | 4 | 2 | 3 | H |
| 3 | Miami | 3 | 3 | 3 | H,S |

*Hawaii*

| HId | location | P | R | B | DB |
|---|---|---|---|---|---|
| 6 | Hawaii | 2 | 2 | 2 | – |
| 7 | Hawaii | 5 | 3 | 1 | M,S |
| 8 | Hawaii | 4 | 1 | 4 | S |

*Seattle*

| HId | location | P | R | B | DB |
|---|---|---|---|---|---|
| 11 | Seattle | 3 | 3 | 1 | – |
| 12 | Seattle | 3 | 1 | 4 | – |
| 13 | Seattle | 2 | 3 | 4 | H |

*Cruises*

*Miami*

| CId | location | P | R | D | DB |
|---|---|---|---|---|---|
| 1 | Miami | 4 | 3 | 3 | H,S |
| 2 | Miami | 5 | 2 | 3 | H |
| 3 | Miami | 5 | 3 | 1 | S |

*Hawaii*

| CId | location | P | R | D | DB |
|---|---|---|---|---|---|
| 6 | Hawaii | 5 | 3 | 1 | M,S |
| 7 | Hawaii | 3 | 2 | 2 | – |
| 8 | Hawaii | 4 | 1 | 4 | – |

*Seattle*

| CId | location | P | R | D | DB |
|---|---|---|---|---|---|
| 11 | Seattle | 4 | 3 | 1 | – |
| 12 | Seattle | 5 | 1 | 4 | H |
| 13 | Seattle | 3 | 3 | 4 | H |

Figure 4.4: Phase II for *Skyline*, *multi-objective*, and *k-dominance* Queries

**K-dominance Skyline**

A *k-dominance* preference query [6] redefines the traditional skyline dominance relation to consider only $k$ dimensional subspaces, where $k$ is less than or equal to the total number of preference attributes. Formally, given a dataset $\mathcal{D}$ of $l$-dimensional tuples, a $k$-dominance query finds each tuple $t$, such that $\nexists\ t' \in \mathcal{D}$; $t'.p_i$ is better than or equal to $t.p_i$ for at least $k$ dimensions, and $t'.p_m$ is strictly better than $t.p_m$ for at least one dimension $m$.

Using *k-dominance* for $\mathcal{P}_{local}$ may discard tuples that are needed to eliminate non-preferred tuples in the final answer set, because *k-dominance* dominance relation is not transitive [6]. For example, consider tuples $r_1$=(1,$\underline{B}$,1,2,3), and $r_2$=(2,$\underline{B}$,2,1,4) in hash bucket $B$, and tuple $r'$=(1,$\underline{B'}$,3,1,2) in hash bucket $B'$ in relation $R$, and tuples $s$=(1,$\underline{B}$,1,2,3) in hash bucket $B$, and $s'$=(1,$\underline{B'}$,1,2,3) in hash bucket $B'$. For 2-dominance skyline, $r_1$ 2-dominates $r_2$, hence as described in Section 4.4.1, we remove tuple $r_2$ from $\mathcal{LP}(R)$. In Phase II, we calculate dominating hash buckets: $DB(r_1)$=$\{B'\}$, $DB(r')$=$\phi$, $DB(s)$=$\phi$, $DB(s')$=$\phi$. Hence, Phase III produces *candidate* preference set=$\{r_1 \bowtie s$, $r' \bowtie s'\}$. In Phase IV, As $r' \bowtie s'$ 2-dominates $r_1 \bowtie s$, the *final* preference set is $\{r_2 \bowtie s'\}$. However, $r_2 \bowtie s$ 2-dominates $r' \bowtie s'$, therefore the correct *final* answer should be empty. Therefore, we could not use *k-dominance* for $\mathcal{P}_{local}$, and $\mathcal{P}_{pairwise}$. Thus, to be able to discard tuples in Phase I, we can use any transitive preference function $f$, such that for all possible input relation, the output of the transitive function $f$ must be superset of the *k-dominance* preference. Therefore, we set $P_{local}$ and $\mathcal{P}_{pairwise}$ to *skyline*. Setting $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwise}$ to *skyline* would produce a superset of the answer set, therefore, to eliminate tuples that are dominated with respect to $k-dominance$, we set $\mathcal{P}_{refine}$ to *k-dominance*. (A formal correctness proof is presented in Section 4.6).

**Example.** We apply *PrefJoin* for *k-dominance* preference function as follow: *Phases I,II, and III*. As we are setting $\mathcal{P}_{local}$ and $\mathcal{P}_{pairwise}$ to *skyline*, exactly the same as *skyline* preference function, Phases I,II, and III proceed as presented earlier in Figure 4.3 and 4.4 for the *skyline*. *Phase IV*. The highlighted tuples in Figure 4.5b resembles the *final* preference set for *5-dominance* preference function. As an example, the candidate joined tuple $t_c = h_1 \bowtie c_2$ = (1,$\underline{Miami}$,4,3,1,2, $\underline{Miami}$,5,2,3) cannot be an answer for a five-dominance skyline query as it is 5-dominated by joined tuple $h_6 \bowtie c_7$ = (6,$\underline{Hawaii}$,2,2,2,7,$\underline{Hawaii}$,3,2,2).

| Hid | Cid | Hp | Hr | Hb | Cp | Cr | Cd |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 3 | 1 | 5 | 2 | 3 |
| 2 | 3 | 4 | 2 | 3 | 5 | 3 | 1 |
| 6 | 6 | 2 | 2 | 2 | 2 | 3 | 1 |
| 6 | 7 | 2 | 2 | 2 | 5 | 2 | 2 |
| 6 | 8 | 2 | 2 | 2 | 4 | 1 | 4 |
| 7 | 7 | 5 | 3 | 1 | 3 | 2 | 2 |
| 7 | 8 | 5 | 3 | 1 | 4 | 1 | 4 |
| 8 | 7 | 4 | 1 | 4 | 3 | 2 | 2 |
| 8 | 8 | 4 | 1 | 4 | 4 | 1 | 4 |
| 11 | 11 | 3 | 3 | 1 | 4 | 3 | 1 |
| 11 | 12 | 3 | 3 | 1 | 5 | 1 | 4 |
| 11 | 13 | 3 | 3 | 1 | 3 | 3 | 4 |
| 12 | 11 | 3 | 1 | 4 | 4 | 3 | 1 |
| 12 | 12 | 3 | 1 | 4 | 5 | 1 | 4 |
| 12 | 13 | 3 | 1 | 4 | 3 | 3 | 4 |
| 13 | 11 | 2 | 3 | 4 | 4 | 3 | 1 |

(a) Skyline

| Hid | Cid | Hp | Hr | Hb | Cp | Cr | Cd |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 3 | 1 | 5 | 2 | 3 |
| 2 | 3 | 4 | 2 | 3 | 5 | 3 | 1 |
| 6 | 6 | 2 | 2 | 2 | 2 | 3 | 1 |
| 6 | 7 | 2 | 2 | 2 | 5 | 2 | 2 |
| 6 | 8 | 2 | 2 | 2 | 4 | 1 | 4 |
| 7 | 7 | 5 | 3 | 1 | 3 | 2 | 2 |
| 7 | 8 | 5 | 3 | 1 | 4 | 1 | 4 |
| 8 | 7 | 4 | 1 | 4 | 3 | 2 | 2 |
| 8 | 8 | 4 | 1 | 4 | 4 | 1 | 4 |
| 11 | 11 | 3 | 3 | 1 | 4 | 3 | 1 |
| 11 | 12 | 3 | 3 | 1 | 5 | 1 | 4 |
| 11 | 13 | 3 | 3 | 1 | 3 | 3 | 4 |
| 12 | 11 | 3 | 1 | 4 | 4 | 3 | 1 |
| 12 | 12 | 3 | 1 | 4 | 5 | 1 | 4 |
| 12 | 13 | 3 | 1 | 4 | 3 | 3 | 4 |
| 13 | 11 | 2 | 3 | 4 | 4 | 3 | 1 |

(b) $K$-dominance

| Hid | Cid | Hp+Cp | Hr | Hb | Cr | Cd |
|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 3 | 1 | 2 | 3 |
| 2 | 3 | 4 | 5 | 2 | 3 | 3 | 1 |
| 6 | 6 | 2 | 2 | 2 | 2 | 3 | 1 |
| 6 | 7 | 2 | 5 | 2 | 2 | 2 | 2 |
| 6 | 8 | 2 | 4 | 2 | 2 | 1 | 4 |
| 7 | 7 | 5 | 3 | 3 | 1 | 2 | 2 |
| 7 | 8 | 5 | 4 | 3 | 1 | 1 | 4 |
| 8 | 7 | 4 | 3 | 1 | 4 | 2 | 2 |
| 8 | 8 | 4 | 4 | 1 | 4 | 1 | 4 |
| 11 | 11 | 3 | 4 | 3 | 1 | 3 | 1 |
| 11 | 12 | 3 | 5 | 3 | 1 | 1 | 4 |
| 11 | 13 | 3 | 3 | 3 | 1 | 3 | 4 |
| 12 | 11 | 3 | 4 | 1 | 4 | 3 | 1 |
| 12 | 12 | 3 | 5 | 1 | 4 | 1 | 4 |
| 12 | 13 | 3 | 3 | 1 | 4 | 3 | 4 |
| 13 | 11 | 2 | 4 | 3 | 4 | 3 | 1 |

(c) Multi-objective

Figure 4.5: Example for Phase IV

### Multi-objective

A *multi-objective* preference query [5] combines subsets of preference attributes using monotone scoring functions, and performs a skyline over the new transformed combined attributes. Formally, given a dataset $\mathcal{D}$ of $l$-dimensional tuples, and $n$ monotone objective functions over tuple's attributes $f_1$, $f_2$, ..., $f_n$, a multi-objective query finds the set of tuples that are not dominated with respect to the objective functions. For our motivating example of Figure 4.1, a *multi-objective* query may sum the hotel price and cruise price into a single attribute and performs the skyline over five attributes: *total* price, hotel rating, hotel distance to beach, cruise rating, and cruise days.

To support *Multi-objective* queries within *PrefJoin*, we set the three preference functions: $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ to be *multi-objective* preference function. It is important to note that as we do not have all the input attributes to the objective functions in each input relation, while evaluating the objective functions, we substitute preference attributes from other input relations by a constant value (e.g., zero). (A formal correctness proof is presented in Section 4.6).

**Example.** We modify the SQL query given in Section 4.1, to sum the hotel price

and cruise price into a single attribute. Hence the *multi-objective* preference function have five attributes: *total* price, hotel rating, hotel distance to beach, cruise rating, and cruise days. We apply *PrefJoin* for *multi-objective* preference function as follow:

*Phases I, II, and III.* proceeds as *skyline* query because the given multi-objective function does not include any objective function that combines attributes from the same relation ( Figure 4.3 and Figure 4.4). Figure 4.5b gives the *candidate* preference set, produced from the joining phase for *Hotels* and *Cruises* relations, depicted in Figure 4.1.

*Phase IV.* The highlighted tuples in Figure 4.5c resembles the *final* preference set for the given *multi-objective* preference function. As an example, the candidate joined tuple $t_c = h_1 \bowtie c_2 =$ (1,*Miami*,4,3,1,2,*Miami*,5,2,3) cannot be an answer as it is dominated by joined tuple $h_6 \bowtie c_6 =$ (6,*Hawaii*,2,2,2,6,*Hawaii*,2,3,1).

## 4.5 Join Order

The pseudo code of our *PrefJoin* algorithm, given in Section 4.4.5, highlights the following important observation. The set of dominating hashing buckets, $DB(s)$, is computed completely for each tuple $r$ *in* $R$, where $R$ is the outer input relation to the join operator. Then, for the inner input relation $S$, we only *partially* compute the set of dominating hash buckets, $DB(s)$, for each tuple $s \in S$. The main idea is that for each tuple $s \in S$, we compute only the dominating hash buckets *among* the ones in $DB(r)$, where $r$ is the current tuple under consideration from relation $R$. This observation means that the overall performance of *PrefJoin* can be affected by the join order, i.e., having $R \bowtie S$ versus $S \bowtie R$. It is the objective of this section to estimate the cost of computing the sets of dominating hash buckets for both input relations $R$ and $S$, should each relation be considered as an outer or an inner. Then, we use these costs to decide which join order will be more beneficial to the overall performance of *PrefJoin*. Using this analysis, we guarantee that our approach will always choose the right join order by selecting the inner relation to be the one with fewer local preference tuples.

*Cost of computing $DB(R)$ for the outer relation $R$.* For each tuple $r \in R$, where $r$ is located in hash bucket $B$, the worst case cost of computing $DB(r)$ can be calculated by estimating the cost of comparing $r$ pair wisely with each other local preference tuple in each other hash bucket $B'$ from relation $R$, i.e., $B' \neq B$. Since the cost of comparing

two tuples is proportional to the number of preference attributes, the total cost for computing $DB(r)$ is $Cost(DB(r))= \sum_B (|B|*n)\ \forall B$, s.t., $r \notin B$, where $|B|$ is the cardinality of hash bucket $B$ in relation $R$, and $n$ is the number of preference attributes in relation $R$. Summing up over all tuples in $R$, the total cost for local preference set computation for outer relation $R$ is estimated to be $Cost_{DB}(R) = \sum_r cost(DB(r))$, $\forall r \in \mathcal{LP}(R)$.

*Cost of computing $DB(S)$ for the inner relation $S$.* For each tuple $s \in S$, to be joined with tuple $r \in R$, where $s$ is located in hash bucket $B$, the worst case cost of computing $DB(s)$ can be calculated by estimating the cost of comparing $s$ pair wisely with each other local preference tuple *only* located in hash buckets $B \in DB(r)$. As the cardinality of $DB(r)$ is significantly lower than the number of hash buckets in $S$, having $s$ as an inner relation encounters much lower cost in computing $DB(S)$, than having $S$ as an outer relation. In our running example, the average cardinality of the sets of dominating hash buckets is 0.89, compared to 3 as the number of hash buckets. Then, the total cost for computing $DB(s)$ is $Cost(DB_r(s))= \sum_B (|B|*m)\ \forall B$, s.t., $B \in DB(r)$, where $|B|$ is the cardinality of hash bucket $B$ in relation $S$, and $m$ is the number of preference attributes in relation $S$. Summing up over all tuples in $S$, the total cost for local preference set computation for inner relation $S$ is estimated to be $Cost_{DB}(S) = \sum_s cost(DB(s))$, $\forall s \in \mathcal{LP}(S)$.

Using the above cost estimations, the *PrefJoin* algorithm and pseudo code is slightly modified to perform the cost estimation procedure right away after Phase I. Basically, *PrefJoin* will contrast two costs: (a) The cost of having $R$ as an outer relation plus the cost of having $S$ as an inner relation, and (b) The cost of having $S$ as an outer relation plus the cost of having $r$ as an inner relation. If the first estimated cost is lower, we just proceed with $R$ as an outer relation, otherwise, we swap the two relations to have $S$ as the outer one.

## 4.6 Proof of Correctness

This section proves the correctness of the *PrefJoin* algorithm for the *skyline*, *k-dominance* and *multi-objective* preference methods. For simplicity, we limit the correctness proof to two input relations $R$ and $S$.

### 4.6.1 Correctness of *PrefJoin* for *skyline* queries

The correctness of *PrefJoin* for *skyline* preference function follows from proving that: (1) All skyline tuples in the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a skyline over the joined relation $R \bowtie S$.

**Theorem 3** *Any tuple $r \bowtie s$ that is a skyline over the relation $R \bowtie S$, will be reported by the PrefJoin algorithm.*

**Proof 3** *Assume that there exist a tuple $t{=}r \bowtie s$ that is a skyline over the relation $R \bowtie S$. However, $t$ is not reported by the* PrefJoin *algorithm. Throughout the* PrefJoin *algorithm, if $t$ is not reported, then this means that either tuple $r$ or $s$ (or both) was discarded in* Local Pruning *or* Joining *phases. In* Local Pruning *phase, a tuple $r$ is only discarded if it is not a* local *preference tuple, i.e., there is a tuple $r' \in R$, and in the same hash bucket of $r$, such that $r'$ dominates $r$. Since $r$ and $r'$ are in the same hash bucket, they have the same value for the join attribute, thus $r' \bowtie s$ dominates $r \bowtie s$, which contradicts our assumption that tuple $r \bowtie s$ is a skyline over the relation $R \bowtie S$. The same contradiction holds for $s$. In* Joining *phase, tuple $r \bowtie s$ is not added only if tuple $s$ is dominated in the same bucket $B$ where tuple $r$ is dominated. Hence, there are other tuples $s' \in B'$, and $r' \in B'$. Thus, the joined tuple $r' \bowtie s'$ dominates $r \bowtie s$, which contradicts our assumption. We conclude that the assumption that $t$ is not reported by* PrefJoin *is not possible.*

**Theorem 4** *Any tuple $r \bowtie s$ that is reported by the* PrefJoin *algorithm is actually a skyline over the relation $R \bowtie S$.*

**Proof 4** *Assume tuple $t{=}r \bowtie s$ is reported by the* PrefJoin *algorithm, but there is another tuple $t'{=}r' \bowtie s'$ that dominates $t$. Assume that $r' \in$ bucket $B'$ of relation $R$ and $s' \in$ bucket $B'$ of relation $S$. Using the property of skyline dominance relation, $r'$ must dominate $r$, and $s'$ must dominate $s$. However, the algorithm adds a tuple to $Final_{sky}$, (Line 22 in Algorithm 7), only if $s$ is not dominated in any bucket where $r$ is dominated, including bucket $B'$ which contains tuple $s'$, as we assume that $s'$ dominates $s$. Therefore, $r \bowtie s$ is not added to $Final_{sky}$, and hence not reported by* PrefJoin*, which contradicts our assumption.*

### 4.6.2 Correctness of *PrefJoin* for *k-dominance* queries

The correctness of *PrefJoin* for *k-dominance* preference function follows from proving that: (1) All *k-dominance* tuples over the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a *k-dominance* preference tuple over the joined relation $R \bowtie S$.

**Theorem 5** *All* k-dominance *tuples over the joined relation* $R \bowtie S$ *are reported from the* PrefJoin *algorithm.*

**Proof 5** *As we set* $\mathcal{P}_{local}$*, and* $\mathcal{P}_{pairwise}$ *to* skyline*, from theorems 1 and 2, the* candidate *preference set contains all* skyline *tuples over the preference query. As the answer of* k-dominance *preference function is a subset of the answer of* skyline *preference function[6],* PrefJoin *returns all* k-dominance *preference tuple.*

**Theorem 6** *Any tuple returned from the* PrefJoin *algorithm is a* k-dominance *preference tuple over the joined relation* $R \bowtie S$*.*

**Proof 6** *As we set* $\mathcal{P}_{refine}$ *to* k-dominance *over the* candidate *preference set which contains all tuples in the answer of the* k-dominance *preference set (Theorem 3), each tuple returned from* PrefJoin *is a* k-dominance *preference tuple over the joined relation* $R \bowtie S$*.*

### 4.6.3 Correctness of *PrefJoin* for *multi-objective* queries

The correctness of *PrefJoin* for *multi-objective* preference function follows from proving that: (1) All preference tuples with respect to *multi-objective* query in the joined relation $R \bowtie S$ are reported from the *PrefJoin* algorithm, and (2) Any tuple returned from the *PrefJoin* algorithm is a multi-objective preference tuple over the joined relation $R \bowtie S$.

**Theorem 7** *Any tuple* $t$ *that is a preferred tuple with respect to* multi-objective *query over the relation* $R \bowtie S$*, will be reported by the* PrefJoin *algorithm.*

**Proof 7** *Assume that there exists a tuple* $t$ *that is a preferred tuple with respect to* multi-objective *query over the relation* $R \bowtie S$*. However,* $t$ *is not reported by the* PrefJoin *algorithm. First assume that* $t{=}r \bowtie s$ *is a* preferred *tuple, yet it is not added to the*

candidate *preference set. Throughout the* PrefJoin *algorithm, if t is not added to the* candidate *preference set, then this means that either tuple r or s (or both) was discarded in* Local Pruning *or* Joining *phases. In* Local Pruning *phase, a tuple r is only discarded if it is not a* local *preference tuple, i.e., there is a tuple* $r' \in R$*, and in the same hash bucket of r, such that* $r'$ *dominates r over preference attributes in relation R, and for other attributes in S, we set them to constant values for tuples r and* $r'$ *and objective functions are monotone. Since r and* $r'$ *are in the same hash bucket, they have the same value for the join attribute, thus* $r' \bowtie s$ *dominates* $r \bowtie s$*, which contradicts our assumption that tuple* $r \bowtie s$ *is should be added to the* candidate *preference set. The same contradiction holds for s. In* Joining *phase, tuple* $r \bowtie s$ *is not added only if tuple s is dominated in the same bucket B where tuple r is dominated. Hence, there are other tuples* $s' \in B'$*, and* $r' \in B'$*. Thus, the joined tuple* $r' \bowtie s'$ *dominates* $r \bowtie s$*, which contradicts our assumption. This proves that* $t{=}r \bowtie s$ *is added to* candidate *preference set, if t is a preferred tuple. Then, as we set* $\mathcal{P}_{refine}$ *to* multi-objective*, then t must be reported if it is preferred with respect to* multi-objective *query. We conclude that the assumption that t is not reported by* PrefJoin *is not possible.*

**Theorem 8** *Any tuple returned from the* PrefJoin *algorithm is a* multi-objective *preference tuple over the joined relation* $R \bowtie S$*.*

**Proof 8** *As we set* $\mathcal{P}_{refine}$ *to* multi-objective *over the* candidate *preference set which contains all tuples in the answer of the* multi-objective *preference set (Theorem 5), each tuple returned from* PrefJoin *is a* multi-objective *preference tuple over the joined relation* $R \bowtie S$*.*

## 4.7 Cost Analysis of *PrefJoin*

In this section, we compare the cost of computing preference queries for the proposed algorithm, *PrefJoin*, and *FlexPref* [34]. We limit our discussion only to *skyline* preference.

For two input relations $R$ with cardinality of $|R|$ tuples and relation $S$ with $|S|$ tuples. We denote the skyline tuples in relation $R$ and $S$ as $Sky(R)$, and $Sky(S)$, respectively. For simplicity, we assume a uniform distribution of skyline tuples in each input relation

over $k$ hash buckets, i.e., the cardinality of skyline tuples in hash bucket $B$ of relation $R$ is $\frac{|Sky(R)|}{k}$. However, our analysis is valid for arbitrary distribution. We compare the relevant performance cost of each phase for both *PrefJoin* and *FlexPref* algorithms, as follow:

- *Local Pruning*: *Local Pruning* phase finds the skyline for each input relations, this phase is identical in the both algorithms, hence it would not affect the relevant performance.

- *Data Preparation*: In *PrefJoin* algorithm, we find the dominance relation between the skyline tuples in local preference for each relation. This cost is bounded by $O(|Sky(R)|^2)$ and $O(|Sky(S)|^2)$ for relation $R$ and $S$, respectively. Hence, the total cost of *Data Preparation* is $O(|Sky(R)|^2 + |Sky(S)|^2)$. However, *FlexPref* does not performs any data preparation.

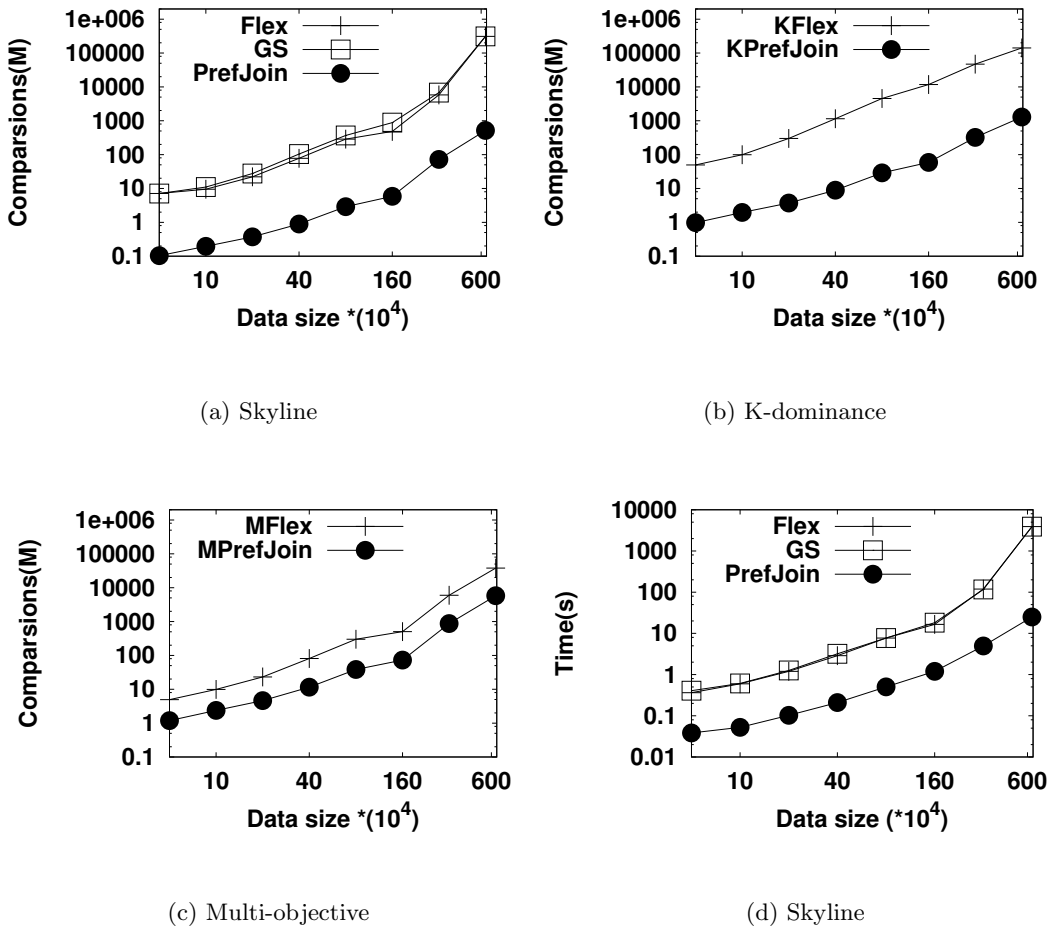
- *Joining*: The cost of joining local preference set for hash bucket $B$ from relation $R$ with the corresponding local preference set in relation $S$, for both approaches, is bounded by $O(\frac{|Sky(R)|}{k}.\frac{|Sky(S)|}{k})$, as the number of tuples is $\frac{|Sky(R)|}{k}$, and $\frac{|Sky(S)|}{k}$ respectively. Hence, the total join cost is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k})$ join operations. The cardinality of the join result is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k})$ tuples.

- *Refining*: *Refining* phase, in *PrefJoin*, is not needed for skyline queries, i.e., $O(1)$, while, the cost to find the skyline over the joined tuples is bounded by $O(\frac{|Sky(R)|.|Sky(S)|}{k}^2)$ comparisons.

From this analysis, we can deduce that our approach performs $O(|Sky(R)|^2 + |Sky(S)|^2)$ comparisons in *Data Preparation* phase to save $O(\frac{|Sky(R)|.|Sky(S)|}{k}^2)$ comparisons.

## 4.8 Experiments

In this section, we analyze the performance of our proposed framework, *PrefJoin*, compared with our two closest related works, the global skyline [33], denoted as *GS* and FlexPref [34], denoted as *Flex*. All our experiments are based on actual implementation




(a) Skyline

(b) K-dominance

(c) Multi-objective

(d) Skyline

Figure 4.6: Scalability

of the three algorithms *PrefJoin*, *GS*, and *Flex*, inside PostgreSQL[103]. Unless mentioned otherwise, our data set is synthetically generated for two input relations $R$ and $S$, where $S$ is the inner relation, the ratio between the cardinality of both relations is 100, i.e., $\frac{|S|}{|R|}$=100, the cardinality of relation $S$ is 1M, the number of groups (i.e., distinct values for the join attribute) is 5K, and the cardinality of the local preference set in each relation is 10% distributed uniformly over all the hash buckets. Also, we assume the set of preference attributes is distributed evenly between the two input relations, with a default of three attributes in each relation. We use the number of comparisons

and wall clock time as our performance measures. All experiments are executed on 2.0 Ghz Intel processor with 1 GB of RAM.

As we will see, *PrefJoin* always outperforms *GS* and *Flex* with at least one order of magnitude. That is why all the experiment figures depicted in this section are plotted with a log scale in terms of the number of comparisons or wall clock time.

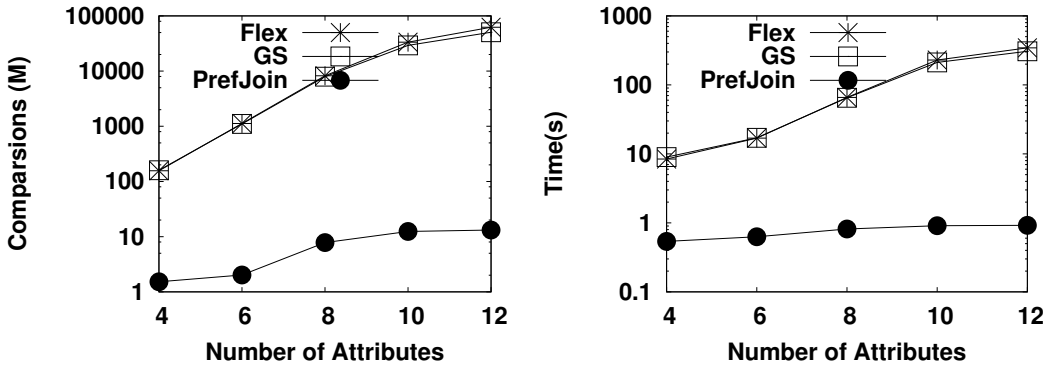
### 4.8.1 Scalability

Figure 4.6 gives the behavior of the three algorithms, when increasing the cardinality of the inner relation $S$ from 50K to 6.4M, while keeping the ratio between relations $R$ and $S$ intact. Figure 4.6a gives the number of comparisons in log scale for a skyline preference function, the order of magnitude difference between our proposed algorithm *PrefJoin* and other algorithms is due to the fact that *PrefJoin* avoids applying the preference function over the joined tuples, and utilizes the dominance relation between tuples in each relation. Similarly, Figures 4.6b and 4.6c give the number of comparisons for *k-dominance* and *multi-objective* respectively. As *GS* is only limited to *skyline*, we run our experiments using *Flex* and *PrefJoin*. The speedup for multi-objective query is smaller than *skyline*, and *k-dominance* preference function, as it requires more computations to be performed in Phase IV because objective preference attributes are not computed until Phase IV.

Figure 4.6d gives the wall clock time in log scale, for *skyline* preference query, which shows that *PrefJoin* has around two orders of magnitude better performance than other algorithms. The wall clock time for *multi-objective* and *k-dominance* shows similar behavior. Based on these experiments, we can conclude that, with the increase of the data size, *PrefJoin* is much more scalable than its competitors, and the performance gain reaches up to three orders of magnitude when the data size exceeds 1M.

### 4.8.2 Number of Preference Attributes

Figure 4.7 studies the effect of the number of preference attributes on the performance of *PrefJoin*, *GS*, and *Flex*, as we increase the number of preference attributes, in each relation, from two to six, i.e., increasing the total number of preference attributes of the output tuples from four to twelve. This directly increases the cardinality of the final




(a) Comparisons (b) Time

Figure 4.7: Number of Preference Attributes

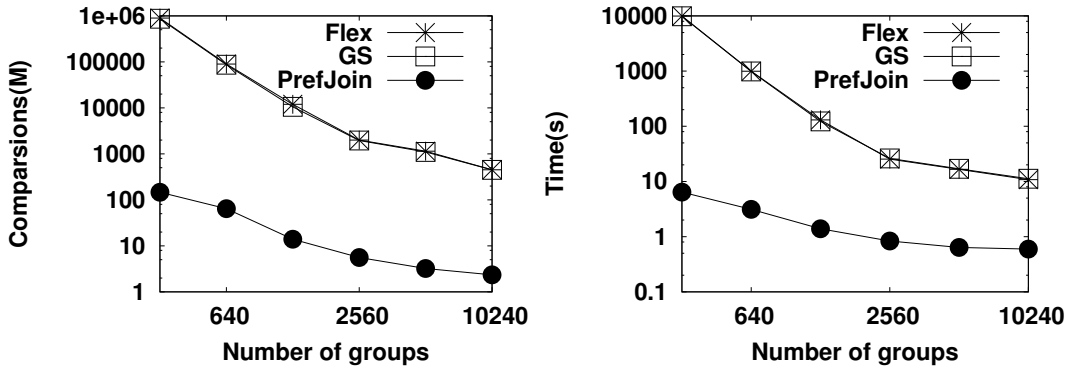
preference set from 2K to 65K. For all algorithms, the number of comparisons and execution time increase with the increase of the number of preference attributes. However, *PrefJoin* exhibits better scalability as it avoids applying the preference function on the preference attributes of the joined tuples.

### 4.8.3 Join Cardinality

This section investigates the effect of the join cardinality on the execution time and number of comparisons. The join cardinality depends on: (a) the number of groups in each relation (i.e., the number of distinct values for the equality join attribute), and (b) the join ratio (i.e., how many tuples in $S$ will be joined with a single tuple from $R$).

#### Number of Groups

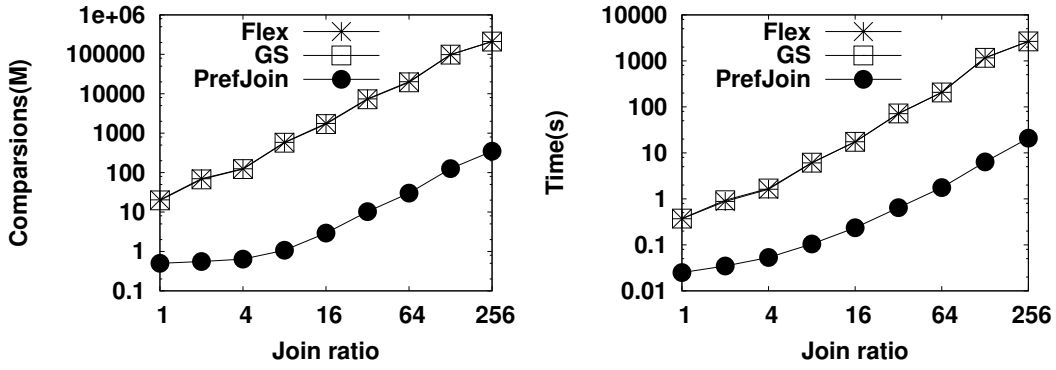
Figure 4.8 studies the effect of increasing the number of groups (i.e., distinct values of the join attribute) from 300 to 10K for input relations, on the total runtime and the comparisons for *PrefJoin*, *Flex* and *GS* algorithms. With the increase of number of groups, the execution time and comparisons for all algorithms decrease exponentially where the size of the final preference set decreases from 260K to 5K. In the mean time, increasing the number of groups increases the cost of computing the sets of dominating hash buckets for local preference tuples, therefore the speedup of *PrefJoin* decreases




(a) Comparisons (b) Time

Figure 4.8: Number of Groups

with respect to *Flex* and *GS* algorithms. Overall, the *PrefJoin* algorithm exhibits at least two orders of magnitude better performance than both *GS* and *Flex* algorithms.



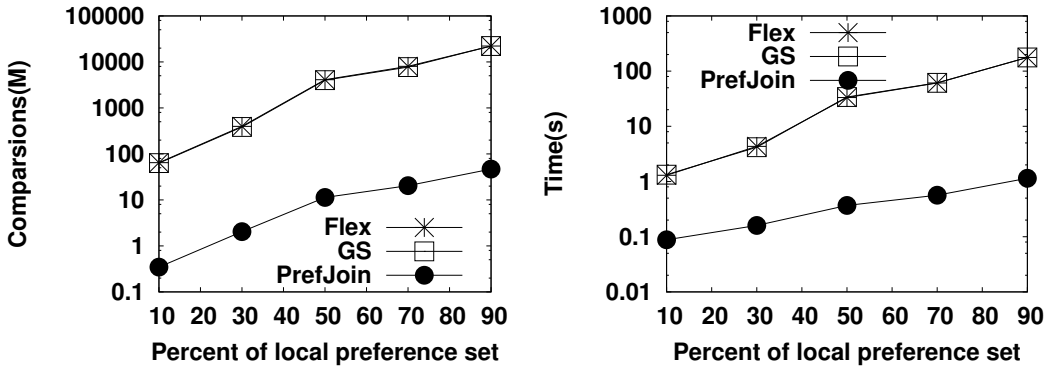

(a) Comparisons (b) Time

Figure 4.9: Join ratio

### Join Ratio

Figure 4.9 increases the join ratio between relations $R$ and $S$, i.e., how many tuples in $S$ will be joined with a single tuple from $R$. The cardinality of $R$ is set to 20K, while the

join ratio is increased from 1:1 to 1:256. With the increase of the join ratio, execution time and comparisons, for all algorithms, increase as the size of the final preference set increases from 1320 to 111K. *PrefJoin*, consistently, has one to two orders of magnitude better performance than both *GS* and *Flex* algorithms, for all join ratios. This is due to: (a) The final preference set size increases; while *PrefJoin* avoids preference comparisons over these tuples, the other algorithms do not, and (b) utilizing the dominance relations in $R$ to avoid unneeded dominance checks in $S$.

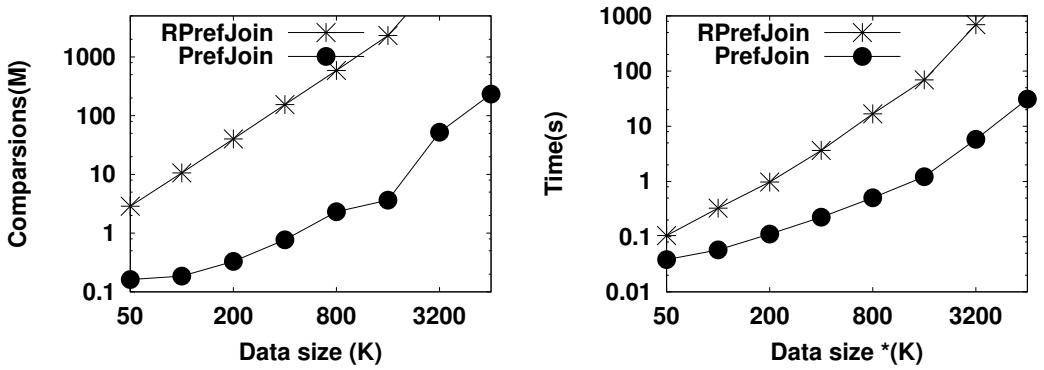



(a) Comparisons (b) Time

Figure 4.10: Percentage of Local preference Sets

### 4.8.4 Percentage of Local Preference Set

Figure 4.10 gives the effect of increasing the percentage of the local preference set for relations $R$ and $S$ from 10% to 90%, on the total runtime and comparisons for *PrefJoin*, *Flex* and *GS*. We set the cardinality of relation $S$ to 200K. Hence, the local preference set for relation $S$ increases from 20K to 180K. With the increase of percentage of local preference set, the execution time and comparisons for all algorithms increase, yet *PrefJoin*, consistently, has at least two orders of magnitude better performance than both *GS* and *Flex* algorithms. This performance is due to the fact that as the final preference sets increase exponentially from 8K to 100K, only *PrefJoin* can utilize the dominance relations from relation $R$ to avoid preference comparisons in $S$.




(a) Comparisons (b) Time

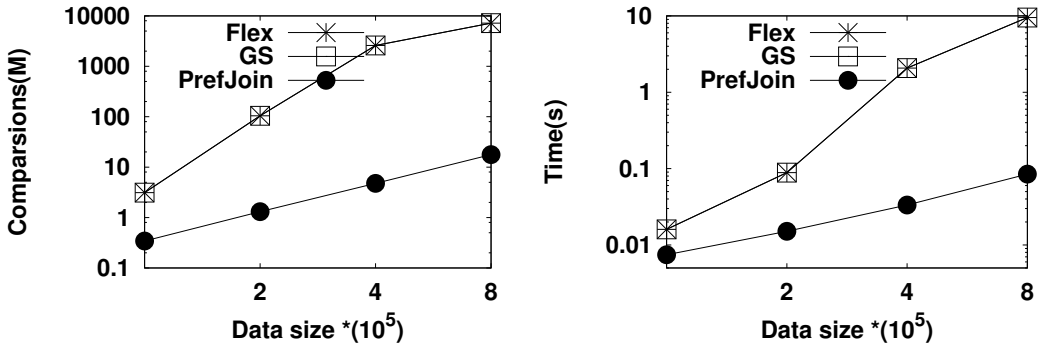
Figure 4.11: Join Order

### 4.8.5 Join Order

Figure 4.11 studies the effect of the join order on the performance of the *PrefJoin* algorithm. We execute the *PrefJoin* algorithm on $R \bowtie S$, i.e., $R$ is an outer relation and $S$ is the inner one, and $S \bowtie R$, termed as *PrefJoin*. As most computations of the set of dominating hash buckets are executed in the outer relation of the join, *PrefJoin* shows an order of magnitude improvements for the running time and comparisons with the increase of the cardinality of relation $S$ from 50K to 6.4M. This confirms the importance of the cost estimations discussed earlier in Section 4.5. Due to accurate cost estimation, *PrefJoin* will decide to use $R$ as the outer relation, and hence achieve an order of magnitude performance improvement in lieu of choosing $S$ as the outer relation.

### 4.8.6 Multiple Input Relations

Figure 4.12 gives the performance of *PrefJoin*, *GS*, and *Flex* for three input relations when increasing the size of each relation from 100K to 800K. With the increase of input size, the number of comparisons and execution time increase for all algorithms. We can see that *PrefJoin* reaches up to four orders of magnitude better performance for the comparisons and three orders of magnitude better for the time than other algorithms. This is mainly because the *candidate* preference set increases exponentially with the




(a) Comparisons

(b) Time

Figure 4.12: Three input relations

input size for *GS* and *Flex* algorithms.

Contrasting this experiment with the similar one given in Figure 4.9 that were designed for only two relations, we can see that the performance gain achieved in *PrefJoin* over other algorithms is even better in case of three relations. This means that *PrefJoin* is better equipped with multiple input relations than other algorithms. This performance is due to two main factors: (1) The progressive output behavior of *PrefJoin* makes it easier to pipeline the result of one join to be the input of another join operator, such behavior is not found in other algorithms, and (2) Increasing the number of joined relations immediately results in increasing the number of preferred tuples which, as we have seen been seen in previous experiments, badly affected other algorithms.

We could not run other experiments for more than three input relations as the cost of executing both *GS* and *Flex* increases dramatically, and that shows us that *PrefJoin* is way preferable for multiple input relations.

## 4.9 Conclusion

This chapter presented *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries where the set of preferred attributes reside in more than one relation. The main idea of *PrefJoin* is to make the join operator

aware of the required preference functionality, and hence inject the ability to early prune those tuples that have no chance of being a preferred tuple. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively. An interesting characteristic of *PrefJoin* is that it aims to join only those tuples that are guaranteed to be an answer, and hence: (a) saves computation costs by not joining unnecessary tuples, and (b) saves expensive preference computations by applying the preference function over those tuples that may needlessly be joined. *PrefJoin* supports a variety of preference function including skyline, multi-objective and $k$-dominance preference queries, by appropriately defining the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for each preference function. The correctness of *PrefJoin* was proved as it returns all preferred tuples and all returned tuples are preferred. Experimental evaluation based on a system implementation of *PrefJoin* and its competitors [34, 33] inside PostgreSQL shows that *PrefJoin* consistently achieves one to three orders of magnitude performance gain over its competitors in various scenarios.

# Chapter 5

# PrefJoin*: An Efficient Preference-aware Join Operator over Imprecise Data

The previous chapter presents an efficient preference-aware join operator, denoted as *PrefJoin*, designed specifically to preference queries over multiple certain relations. In this chapter, we extend this work to address uncertain and incomplete data, denoted as *PrefJoin**. For uncertain data, we employ a user-defined threshold ($H$) and tolerance($\delta$), as presented in Chapter 3, to reduce the burden of computations and to give interesting objects to users. In this chapter, we will only highlight the differences needed to support uncertain and incomplete data.

As the case of preference function over certain data, the state-of-art for evaluating preference for uncertain data assume that the set of preferred attributes are stored in only one relation, waiving on a wide set of queries that include preference computations over multiple imprecise relations. The extended framework supports a variety of preference function including skyline, multi-objective and $k$-dominance uncertain preference queries, and incomplete skyline, discussed in Chapter 2.

## 5.1 Introduction

To help present the changes, consider, for example, a scenario where a user is looking for a vacancy destination where she is interesting in two activities, namely, hiking and climbing in nearby places. User preferences for the hiking are lower cost, better rating, and longer distance and for the climbing are lower risk, better rating, and better view. Figure 5.1 gives information about hiking and climbing, stored in relations *Hiking* and *Climbing*, respectively. The price and locations for hiking and climbing are uncertain. For simplicity, we assume a one dimension uncertain dimension. Assuming minimum rating is better, this user preference request can be represented by the following SQL query:

```
SELECT * from Hiking h, Climbing c
WHERE c.location = h.location
PREFERENCE h.price(min), h.rating(min),
c.price(min), c.rating(min)
```

An naive way to answer this SQL query using existing single-table preference techniques, we first need to join the two input relations (i.e., *Hiking* and *Cruises*), using uncertain joining [125, 126] and then employ the preference framework *UPref*, presented in Chapter 3 on the joined relation based on the location attribute, i.e., `UPref` (*Hiking* $\bowtie_{location}$ *Cruises*). Unfortunately, such an approach is very inefficient as it completely isolates the preference functionality from the join operator. As a result, the join operation will produce too many tuples that have no chance of being in the final answer set. As we will show later in the experiment section, this approach is an order of magnitude less efficient than the proposed framework.

The main goal of *PrefJoin** is to make the join operation aware of the preference functionality over imprecise data, and hence we would be able to early prune those tuples that have no chance of being a preferred tuple before the join operation. The *PrefJoin** framework extend the four phases presented in Sections 4.4.1 to 4.4.4, to be able to handle uncertain and incomplete data. The extended phases are denoted *Local Pruning**, *Data Preparation**, *Joining**, and *Refining**. The *Local Pruning** phase filters out, from each input relation those tuples that are guaranteed not to be in the final

| Id | Loc. | Price | R |
|---|---|---|---|
| 1 | [1.0–4.0] | [0–4] | 6 |
| 2 | [1.5–3.5] | [2–6] | 4 |
| 3 | [1.0–3.0] | [1–4] | 3 |
| 4 | [0.5–3.0] | [3–5] | 2 |
| 5 | [1.5–3.0] | [4–6] | 5 |
| 6 | [5.0–8.0] | [1–5] | 5 |
| 7 | [5.5–7.5] | [3–7] | 3 |
| 8 | [4.0–7.0] | [2–5] | 2 |
| 9 | [5.5–7.0] | [4–6] | 1 |
| 10 | [4.5–7.0] | [5–7] | 4 |
| 11 | [9.5–11.0] | [0–4] | 4 |
| 12 | [8.5–11.0] | [2–6] | 2 |
| 13 | [9.0–11.0] | [1–4] | 1 |
| 14 | [9.5–11.5] | [3–5] | 0 |
| 15 | [9.0–12.0] | [4–6] | 3 |

*Hikings*

| Id | Loc. | Price | R |
|---|---|---|---|
| 1 | [1.0–3.0] | [0–4] | 5 |
| 2 | [1.5–3.5] | [2–6] | 3 |
| 3 | [1.5–3.0] | [1–4] | 2 |
| 4 | [1.0–4.0] | [3–5] | 1 |
| 5 | [0.5–3.0] | [4–6] | 4 |
| 6 | [5.5–7.5] | [1–5] | 6 |
| 7 | [4.5–7.0] | [3–7] | 4 |
| 8 | [5.0–8.0] | [2–5] | 3 |
| 9 | [4.0–7.0] | [4–6] | 2 |
| 10 | [5.5–7.0] | [5–7] | 5 |
| 11 | [8.5–11.0] | [0–4] | 4 |
| 12 | [9.5–11.0] | [2–6] | 2 |
| 13 | [9.0–12.0] | [1–4] | 1 |
| 14 | [9.0–11.0] | [3–5] | 0 |
| 15 | [9.5–11.5] | [4–6] | 3 |

*Cruises*

Figure 5.1: Motivating Example

preference set. For uncertain data, it discards tuples that are dominated utilizing the given threshold ($H$). The *Data Preparation** phase associates meta data with each non-filtered tuple that will be used to optimize the execution of the next phases. Note that we calculate the dominance probability. The *Joining** phase uses that meta data, computed in the previous phase (including the dominance probability), to decide on which tuples should be joined together. Finally, the *Refining** phase finds the *final* preference set from the output of the joining phase. Note that we still does not assume the existence of any index structure, and achieves orders of magnitude performance over the pre-described naive approach. The first three phases only computes or uses the upper bound probability of the objects, while the last phase computes the probability of preference of objects within the user-given tolerance $\delta$. Section 5.2 revises the problem formulation, presented in Section 4.2 to support imprecise data.

## 5.2 Problem Formulation

Without loss of generality, we assume that all dimension values have a total order in which smaller values are better.

**Problem Formulation.** Given: (1) $m$ input imprecise relations $R_1$, $R_2$, ..., $R_m$, (2) Equality join condition over $R_1$ to $R_m$, (3) A set of preference attributes $\mathcal{P}$; such that $\forall\ p \in \mathcal{P}$, $\exists\ i$ s.t $p \in R_i$ and $\forall\ R_i$, $R_i \cap \mathcal{P} \neq \phi$, and (4) A preference method $\mathcal{M}$. A preference query $Q$ finds tuples from $R_1 \bowtie R_2 \bowtie \ldots\ R_m$, that are preferred with respect to $\mathcal{P}$ and $\mathcal{M}$. The join attribute may be uncertain, and the preference attributes may be either certain, uncertain or incomplete.

Applying this formulation to the SQL query given in Section 5.1 gives: (1) Two input relations *Hiking H* and *Climbing C*, (2) The equality join condition is *H.location* = *C.location*, (3) four preference attributes, *H.price*, *H.rating*, *C.price*, *C.rating*, and (4) A skyline preference method. The SQL query finds those tuples, on the form (*HID*, *CID*, *Preference attributes*) that are skylines over the four preference attributes from *H* $\bowtie$ *C*.

## 5.3 PrefJoin*: A Preference-Aware Join Operator over Imprecise Data

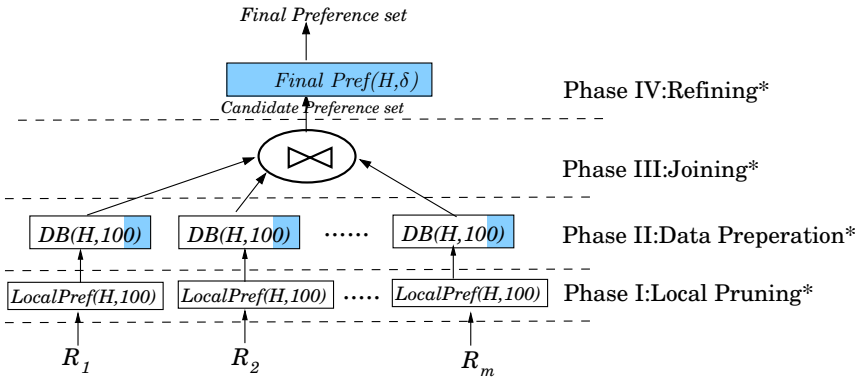



Figure 5.2: Phases of *PrefJoin** Algorithm

In this section, we discuss only the changes needed for *PrefJoin* to realize our proposed framework, *PrefJoin*.PrefJoin** is aware of uncertainty computations, as it prunes uncertain objects using the user-given threshold ($H$). Thus, both by make the join operator aware of the required preference functionality. Therefore, we can early prune more objects in Phase I, and Phase III with minimal computational overhead. The *PrefJoin** framework slightly modifies the four phases of *PrefJoin*, as depicted in Figure 5.2, namely, *Local Pruning**, *Data Preparation**, *Joining**, and *Refining**. These phases use three preference functions $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ that are chosen carefully based on $\mathcal{P}$. We compute an upper bound of object probability during the first three phases, and compute an upper and lower bound on the probability in the last phase. Table 5.1 gives *possible* choices of $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for *uncertain skyline*, *uncertain multi-objective*, *uncertain top-k* preference functions over uncertain dimensions. Also, it shows the settings for *incomplete skyline* over incomplete data. Note that, unlike *PrefJoin*, we need to compute skyline in the refining phase to compute the probability bounds for objects within the user-given tolerance $\delta$. The *Local Pruning** phase filters out, using $\mathcal{P}_{local}$ and the upper bound probability threshold ($H$), from each input relation, those tuples that are guaranteed not to be in the final preference set. The *Data Preparation** phase associates meta data (including upper bound probability) with each non-filtered tuple that will be used to optimize the execution of the next phase. The *Joining** phase uses that meta data, computed in the previous phase, to decide on which tuples should be joined together. Finally, the *Refining** phase uses $\mathcal{P}_{refine}$ to find the *final* preference set from the output of the joining phase.

Sections 5.3.1-5.3.4 discuss the changes in four phases. Section 5.3.5 gives the pseudo-code for the *PrefJoin** algorithm.

### 5.3.1 Phase I: Local Pruning*

As the case for *PrefJoin*, Phase I filters out those tuples, from each input relation, that are guaranteed to be not in the final preference answer. The output of Phase I, i.e., the set of non-filtered tuples, is the local preference set $\mathcal{LP}(R_i)$ for each input relation $R_i$, which is defined as the set of tuples such that each tuple $t \in \mathcal{LP}(R_i)$ is a *preferred* tuple over all tuples in $R_i$ with the same join attributes values.

The main difference between Local pruning* and *local pruning*, presented in Section 4.4.1 is that Local pruning* use *Partitioning* instead of *Hashing*, if the join is on an uncertain attribute. We use partitions as instead of the hash buckets. For *Hikings* and *Climbing* relations, we divide the input relations into three partitions: (1) $[0.0, 4.0]$, (2)$[4.0, 8.0]$, and (3) $[8.0, 12.0]$. Please note that we associate the preference probability calculated by $\mathcal{P}_{local}$ with each non-discarded tuple $t$. If there is no preference attribute over uncertain data, the upper bound probability is set to 100%.

| | $\mathcal{P}_{local}$ | $\mathcal{P}_{pairwise}$ | $\mathcal{P}_{refine}$ |
|---|---|---|---|
| Uncertain Skyline | Uncertain *Skyline* | Uncertain *Skyline* | Uncertain *Skyline* |
| Uncertain Multi-objective | Uncertain *Multi-Objective* | Uncertain *Multi-Objective* | Uncertain *Multi-objective* |
| Uncertain Top-K | Uncertain *Top-k* | Uncertain *Sorting* | Uncertain *Rank-aware join* |
| Incomplete Skyline | *Skyline* | *Skyline* | Incomplete *Skyline* |

Table 5.1: Possible setting of $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for some preferences functions

### 5.3.2 Phase II: Data Preparation*

Similar to *PrefJoin*, Phase II takes, as input, the local preference set, $\mathcal{LP}(R_i)$, for each relation $R_i$, produced from Phase I and passes it to Phase III along with a set of information, termed *Dominating hash buckets*, $DB(t)$, associated with each tuple $t \in \mathcal{LP}(R_i)$. Such information will be used later in Phase III to avoid producing unnecessary joined tuples.

We use the first phase from *UPref*, presented in Chapter 3, to compute the upper bound probability for the objects over preference attributes that contains uncertain data. For completeness, we revise computing the Dominating hash buckets. For each tuple $t$, we maintain the minimum upper bound probability, while comparing it using $\mathcal{P}_{pairwise}$. For a local preference tuple $t$ of bucket $B$ in relation $R$, $DB(t)$ is computed by comparing $t$ with the first tuple $t'$ of each hash bucket $B'$ in relation $R$, where $B' \neq B$. Three cases may occur:

- *Case 1: $t'$ is preferred over $t$ with respect to $\mathcal{P}_{pairwise}$ with a upper bound probability*

*greater than user-defined threshold (H) using the first phase of* UPref . In this case, we add bucket $B'$ to $DB(t)$ as this means that there is a tuple in $B'$ that is preferred over $t$. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in $B$ can be preferred over $t'$, thus no further preference checks are needed for other tuples in $B'$. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare $t$ with the next tuple in $B'$, and act accordingly based on our three cases.

- *Case 2: t is preferred over $t'$ with respect to $\mathcal{P}_{pairwise}$ with a upper bound probability greater than user-defined threshold (H) using the first phase of* UPref. In this case, we add bucket $B$ to $DB(t')$ as this means that there is a tuple in $B$ that is preferred over $t'$. If $\mathcal{P}_{pairwise}$ is transitive, we guarantee that no other tuples in $B'$ can be preferred over $t$, thus no further preference checks are needed for other tuples in $B'$. On the other hand, if $\mathcal{P}_{pairwise}$ is not transitive, we proceed to compare $t$ with the next tuple in $B'$, and act accordingly based on our three cases.

- *Case 3: Neither t is preferred over $t'$ nor $t'$ is preferred over t.* In this case, we do not change neither $DB(t)$ nor $DB(t')$. We proceed with next tuple from $B'$, and act accordingly based on our three cases.

### 5.3.3 Phase III: Joining*

Phase III takes, as input, the local preference set $\mathcal{LP}(R_i)$ where each tuple $t \in \mathcal{LP}(R_i)$ is associated with a set of dominating hash buckets, $DB(t)$, and the upper bound probability, calculated in Phase I and Phase II. Phase III uses $DB(t)$ and the upper bound probability to decide which local preference tuples $t_i \in \mathcal{LP}(R_i)$ and $t_j \in \mathcal{LP}(R_j)$ should be joined together, to produce the *candidate* preference set, denoted as $Candidate_{pref}$. Basically, two tuples $r$ and $s$ from relations $R$ and $S$ that satisfy the equality join condition will be joined together only if $DB(r) \cap DB(s) = \phi$ and the probability of object $r \bowtie s$ exceeds the user-defined threshold ($H$). Unlike all other phases, this phase does not directly depend on the preference function $\mathcal{P}$, as it always has the same execution regardless of $\mathcal{P}$.

For completeness, we detail the needed changes. Consider two local preference tuples $r$ and $s$ from corresponding hash bucket $B$ of relations $R$ and $S$, respectively. Two cases

may arise:

- *Case 1: $DB(r) \cap DB(s) \neq \phi$, i.e., the sets of dominating hash buckets for $r$ and $s$ are overlapping.* In this case, we are sure that the joined tuple $t = r \bowtie s$ will not be preferred, hence, we avoid joining $r$ and $s$. To illustrate, consider bucket $B' \in DB(r) \cap DB(s)$. There must be tuple $r' \in B'$, such that $r'$ is preferred over $r$. Similarly, there must be $s' \in B'$, such that $s'$ is preferred over $s$. This means that the tuple $t' = r' \bowtie s'$ must be preferred over $t{=}r \bowtie s$.

- *Case 2: $DB(r) \cap DB(s) = \phi$, i.e., the sets of dominating hash buckets for $r$ and $s$ are disjoint.* In this case, the joined tuple $t = r \bowtie s$ may be a *candidate* preference tuple, i.e., it is a preferred tuple by separately considering attributes in relations $R$ and $S$. Hence, we perform the join and produce tuple $t$. Then, we compute the join probability $pr_{join}$ of $t$ using [125], if the join attribute is uncertain. Otherwise, $pr_{join}$ is set to 100%. The non-dominance probability of preference for joined tuple $t$ is the product of *r.prob* * $pr_{join}$ * *s.prob*. If this probability is greater than the user-defined threshold, we add tuple $t$ to the candidate preference set, $Candidate_{pref}$. Otherwise, we discard the tuple.

The same idea is easily generalized to $m$ input relations $R_1$, $R_2$ ,..., $R_m$.

### 5.3.4 Phase IV: Refining*

Phase IV takes, as input, the *candidate* preference set, $Candidate_{pref}$, produced from the *joining** phase, and finds the final preference answer for the preference function $\mathcal{P}$, along with its preference probability. Simply, Phase IV employs a preference function $\mathcal{P}_{refine}$ over the set of tuples in the candidate preference set produced from Phase III. Each tuple $t$ that is preferred with respect to $\mathcal{P}_{refine}$ is a final preference tuple for $\mathcal{P}$.

### 5.3.5 *PrefJoin**: Pseudocode

Algorithm 8 gives the pseudo code of the *PrefJoin** algorithm, presented for two relations $R$ and $S$, for simplicity. The input to the algorithm is the two input relations $R$ and $S$ along with the three preference functions $\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$, $\mathcal{P}_{refine}$, user given threshold $H$, and tolerance $\delta$. The changes to *PrefJoin* are underlined in the pseudocode.

**Algorithm 8** PrefJoin*

1: **Function** PrefJoin*(*Relation* $R$, *Relation* $S$,$\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$,$\mathcal{P}_{refine}$, $H$, $\delta$ )
2: /* Phase I */
3: $\mathcal{LP}(\mathcal{R}) \leftarrow \phi$; $\mathcal{LP}(\mathcal{S}) \leftarrow \phi$
4: Build hash buckets(partitions) for relations $R$ and $S$
5: **for each** Hash Bucket(partition) $B$ in relation $R$ **do**
6: $\mathcal{LP}(\mathcal{R}) \leftarrow \mathcal{LP}(\mathcal{R}) \cup$ ApplyPreferenceFunction($B$,$\mathcal{P}_{local}$) ; only the first phases
7: **end for**
8: **for each** Hash Bucket(partition) $B$ in relation $S$ **do**
9: $\mathcal{LP}(\mathcal{S}) \leftarrow \mathcal{LP}(\mathcal{S}) \cup$ ApplyPreferenceFunction ($B$, $\mathcal{P}_{local}$) ; only the first phase
10: **end for**
11: /* Phase II*/
12: **for each** local preference tuple $r \in \mathcal{LP}(R)$ **do**
13: $DB(r) \leftarrow$ The set of hash buckets(partitions) in $R$ that include preferred tuple(s) over $r$ with respect to $\mathcal{P}_{pairwise}$ preference function ; only the first phase
14: **end for**
15: **for each** local preference tuple $s \in \mathcal{LP}(S)$ **do**
16: $DB(s) \leftarrow$ The set of hash buckets in $S$ that include preferred tuple(s) over $s$ with respect to $\mathcal{P}_{pairwise}$ preference function
17: **end for**
18: /* Phase III*/
19: $Candidate_{pref} \leftarrow \phi$
20: **for** each pair of tuples ($r$,$s$) where $r$ in hash bucket(partition) B in $R$ and $s$ in the corresponding hash bucket(partition) $B$ in $S$ **do**
21: **if** $(DB(r) \cap DB(s) = \phi)$ **then**
22: **if** $join_p r(r,s) * r.prob * s.prob > H$ **then**
23: Add $r \bowtie s$ to $Candidate_{pref}$
24: **end if**
25: **end if**
26: **end for**
27: /* Phase IV*/
28: **return** ApplyPreferenceFunction ($Candidate_{pref}$,$\mathcal{P}_{refine}$, $H$, $\delta$)

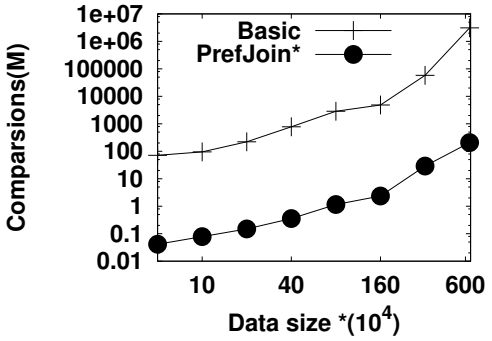
Preference functions $\mathcal{P}_{local}$,$\mathcal{P}_{pairwise}$, $\mathcal{P}_{refine}$ are set based on the desired preference function $\mathcal{P}$, i.e., per Table 5.1. The algorithm starts by executing Phase I, i.e., building the hash buckets, or partitions if the join attribute is uncertain, and computing the *local* preference sets $\mathcal{LP}(\mathcal{R})$ and $\mathcal{LP}(\mathcal{S})$ for the input relations $R$ and $S$, respectively. This is achieved by applying the preference function $\mathcal{P}_{local}$ over each hash bucket or partition in both input relations (Lines 3 to 10 in Algorithm 8). Then, we proceed to Phase II, where we compute the set of dominating hash buckets $DB(r)$ and the upper bound probability for each tuple $r \in \mathcal{LP}(\mathcal{R})$ and $DB(s)$ for each tuple $s \in \mathcal{LP}(\mathcal{S})$.

$DB(r)$ and $DB(s)$ are computed as the set of hash buckets in $R$ and $S$ that include preferred tuple(s) over $r$ and $s$, respectively, with respect to $\mathcal{P}_{pairwise}$ preference function (Lines from 12 to 17 in Algorithm 8). Then, we proceed to Phase III, as we initialize the candidate preference set, $Candidate_{pref}$ to be empty. We iterate over each pair of tuples ($r$,$s$) where $r$ in hash bucket(partition) B in $R$ and $s$ in the corresponding hash bucket(partition) $B$ in $S$. In this iteration, we compute $r \bowtie s$, and add its result to $Candidate_{pref}$ only if $DB(r) \cap DB(s) = \phi$ and if the probability of $t$ tp be preference object exceeds the user-defined threshold $H$(Lines 19 to 26 in Algorithm 8). Finally, in Phase IV, we apply the preference function $\mathcal{P}_{refine}$ over all entries in $Candidate_{pref}$ to produce the final answer for $\mathcal{P}$, using the user-give threshold and tolerance (Lines 27 to 28 in Algorithm 8).

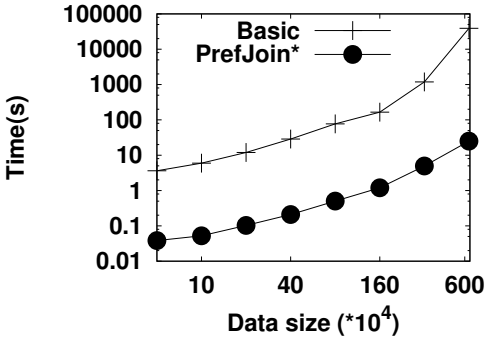
## 5.4 Experiments

In this section, we analyze the performance of our proposed framework, *PrefJoin** in comparison to first computing the join, then applying the preference function over the joined relation. This approach is denoted as *Basic*. Unless mentioned otherwise, our data set is synthetically generated for two input relations $R$ and $S$, where $S$ is the inner relation, the ratio between the cardinality of both relations is 100, i.e., $\frac{|S|}{|R|}$=100, the cardinality of relation $S$ is 1M, the number of groups (i.e., distinct partitions for the join attribute) is 5K, and the cardinality of the uncertain local preference set in each relation is 10% distributed uniformly over all the hash buckets. Each relation contains one uncertain dimension, other than the join attribute, represents a uniform random variable from 0 to 10,000. Also, we assume the set of preference attributes is distributed evenly between the two input relations, with a default of three attributes in each relation. User-defined threshold and tolerance are set to 50%, and 1%, respectively. We use the number of comparisons and wall clock time as our performance measures. We assume the join attribute and preference attributes are uncertain. All experiments are executed on 2.0 Ghz Intel processor with 1 GB of RAM.

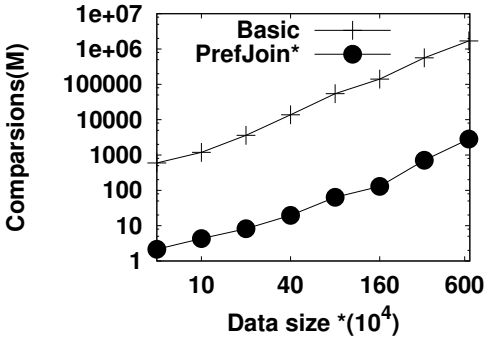
As we will see, *PrefJoin** always outperforms *Basic* with at least one order of magnitude. That is why all the experiment figures depicted in this section are plotted with a log scale in terms of the number of comparisons or wall clock time.



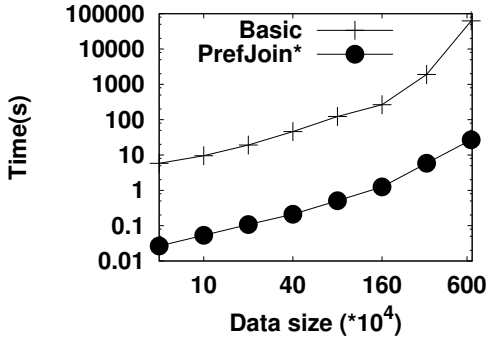

(a) Skyline



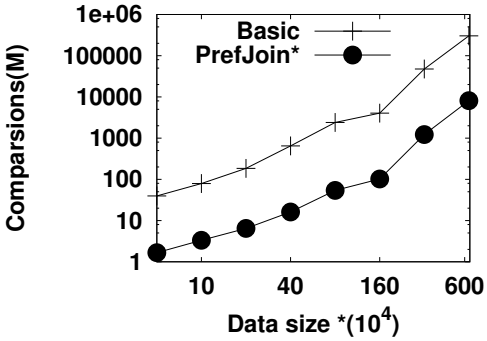

(b) Skyline



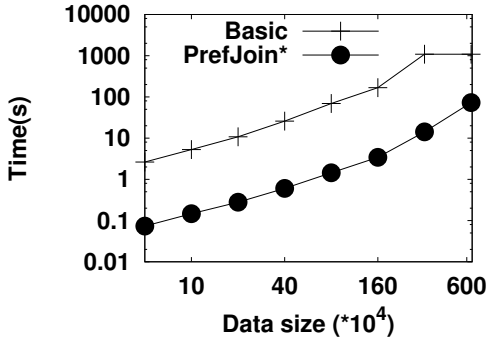

(c) Multi-objective




(d) Multi-objective

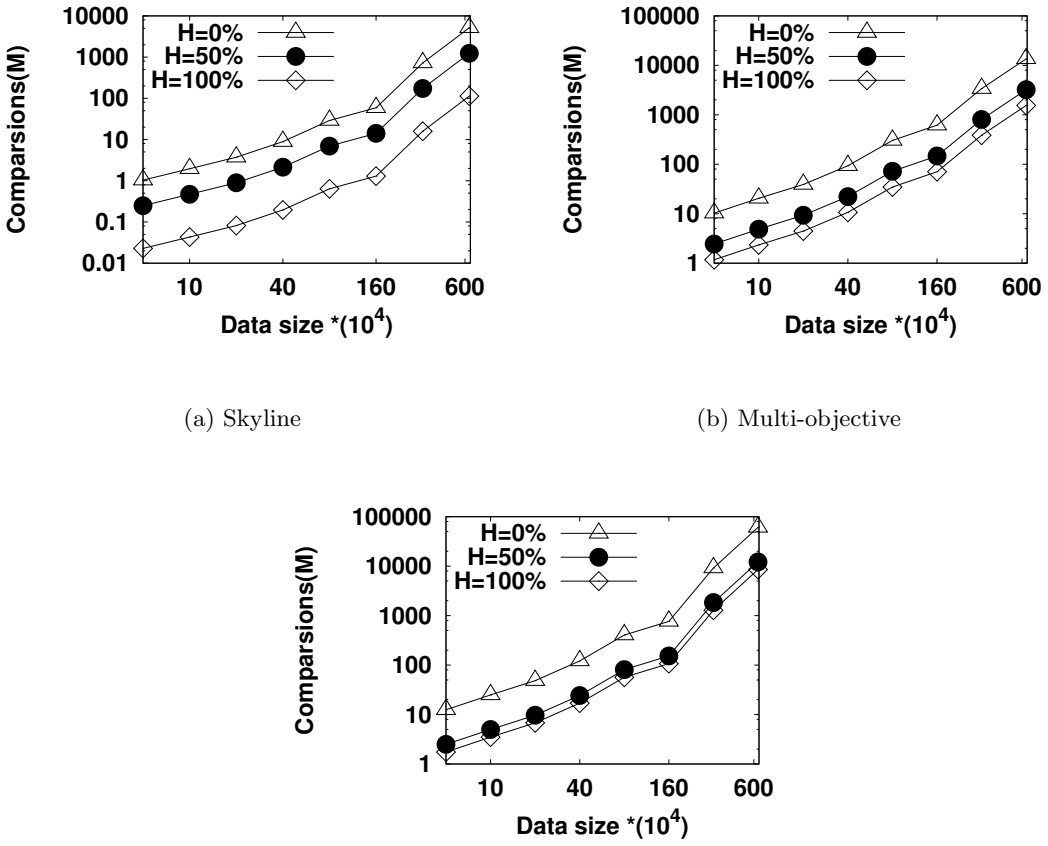



(e) Top-K




(f) Top-K

Figure 5.3: Scalability




(a) Skyline

(b) Multi-objective

(c) Top-K

Figure 5.4: Effect of Threshold on Scalability

### 5.4.1 Scalability

Figure 5.3 gives the behavior of the algorithms, when increasing the cardinality of the inner relation $S$ from 50K to 6.4M, while keeping the ratio between relations $R$ and $S$ intact. Figure 5.3a gives the number of comparisons in log scale for a skyline preference function, the order of magnitude difference between our proposed algorithm *PrefJoin* and other algorithms is due to the fact that *PrefJoin** avoids applying uncertain computations over the joined tuples, and utilizes the dominance relation between tuples in each relation. Figure 5.3b gives the wall clock time in log scale, for *skyline* preference query,

which shows that *PrefJoin** has around two orders of magnitude better performance.

Similarly, Figures 5.3c and 5.3d give the number of comparisons and wall clock time for *k-dominance*, and Figures 5.3e and 5.3f give the number of comparisons and wall clock time for *multi-objective*. The speedup for multi-objective query is smaller than *skyline*, and *k-dominance* preference function, as it requires more computations to be performed in Phase IV because objective preference attributes are not computed until Phase IV.

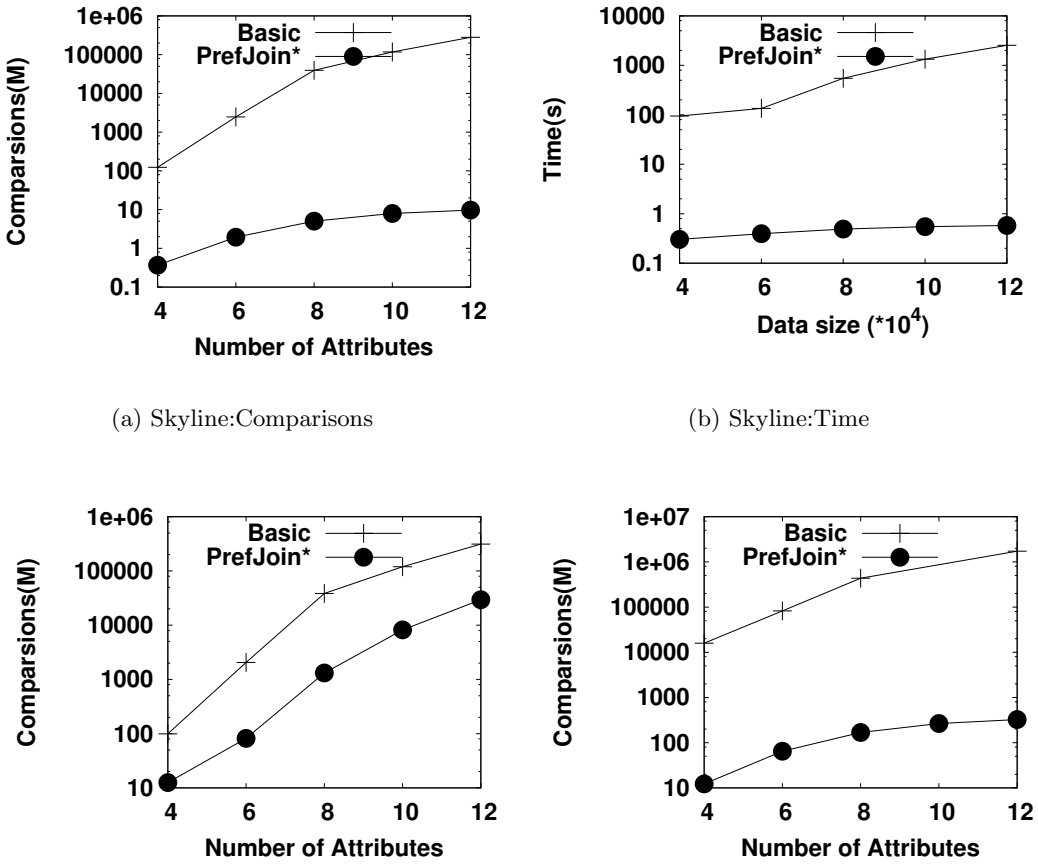
Figure 5.4 gives the effect of threshold for the number of comparisons, when increasing the cardinality of the inner relation $S$ from 50K to 6.4M, while keeping the ratio between relations $R$ and $S$ intact. Based on these experiments, we can conclude that, with the increase of the data size, *PrefJoin** is much more scalable than its competitor, and the performance gain reaches up to three orders of magnitude when the data size exceeds 1M.

### 5.4.2 Number of Preference Attributes

Figure 5.5 studies the effect of the number of preference attributes on the performance of *PrefJoin** and *Basic*, as we increase the number of preference attributes, in each relation, from two to six, i.e., increasing the total number of preference attributes of the output tuples from four to twelve. This directly increases the cardinality of the final preference set from 200 to 6000. For all algorithms, the number of comparisons and execution time increase with the increase of the number of preference attributes. However, *PrefJoin** exhibits better scalability as it avoids applying the preference function on the preference attributes of the joined tuples.

### 5.4.3 Join Cardinality

This section investigates the effect of the join cardinality on the execution time and number of comparisons. The join cardinality depends on: (a) the number of non-overlapping partitions, and (b) the join ratio (i.e., how many tuples in $S$ will be joined with a single tuple from $R$).




(a) Skyline:Comparisons

(b) Skyline:Time

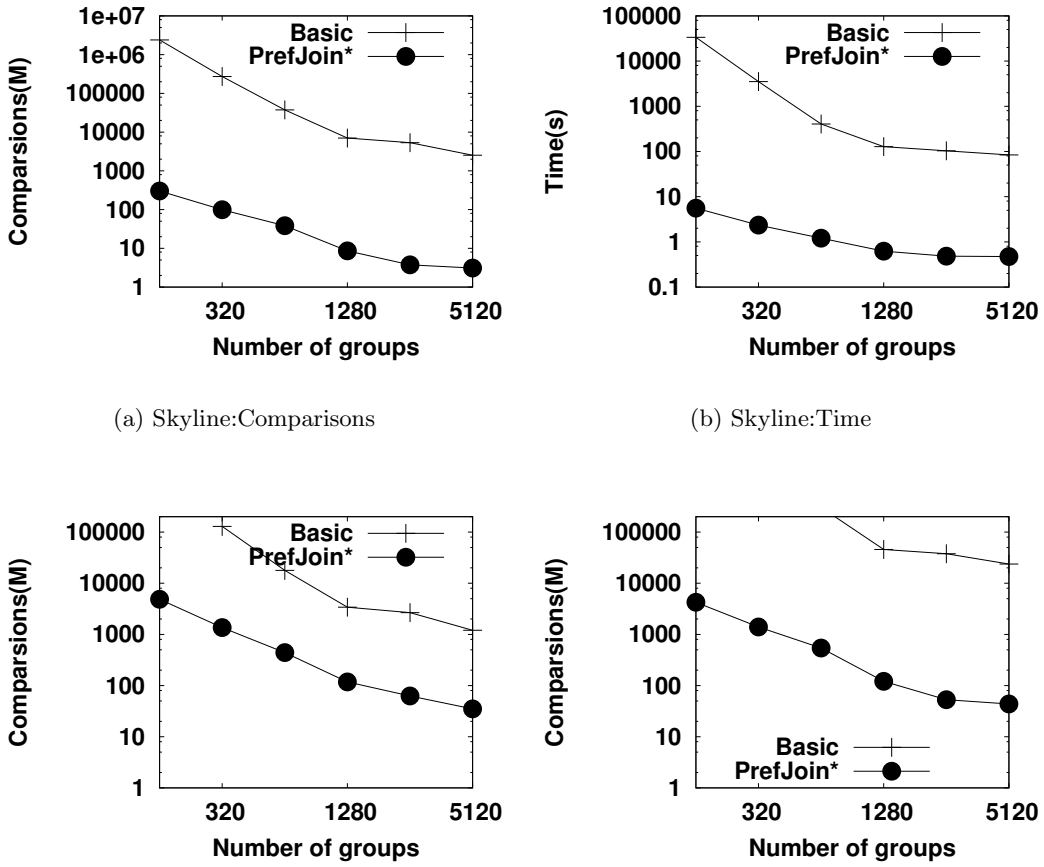
(c) Multi-objective:Comparisons

(d) Top-K:Comparisons

Figure 5.5: Number of Preference Attributes

### Number of Groups

Figure 5.6 studies the effect of increasing the number of non-overlapping partitions on the join attribute from 150 to 5K for input relations, on the total runtime and the comparisons for *PrefJoin**, *Basic* algorithms. With the increase of number of groups, the execution time and comparisons for all algorithms decrease exponentially where the size of the final preference set decreases from 260K to 5K. In the mean time, increasing the number of groups increases the cost of computing the sets of dominating hash buckets for local preference tuples, therefore the speedup of *PrefJoin** decreases with




(a) Skyline:Comparisons

(b) Skyline:Time

(c) Multi-Objective:Comparisons

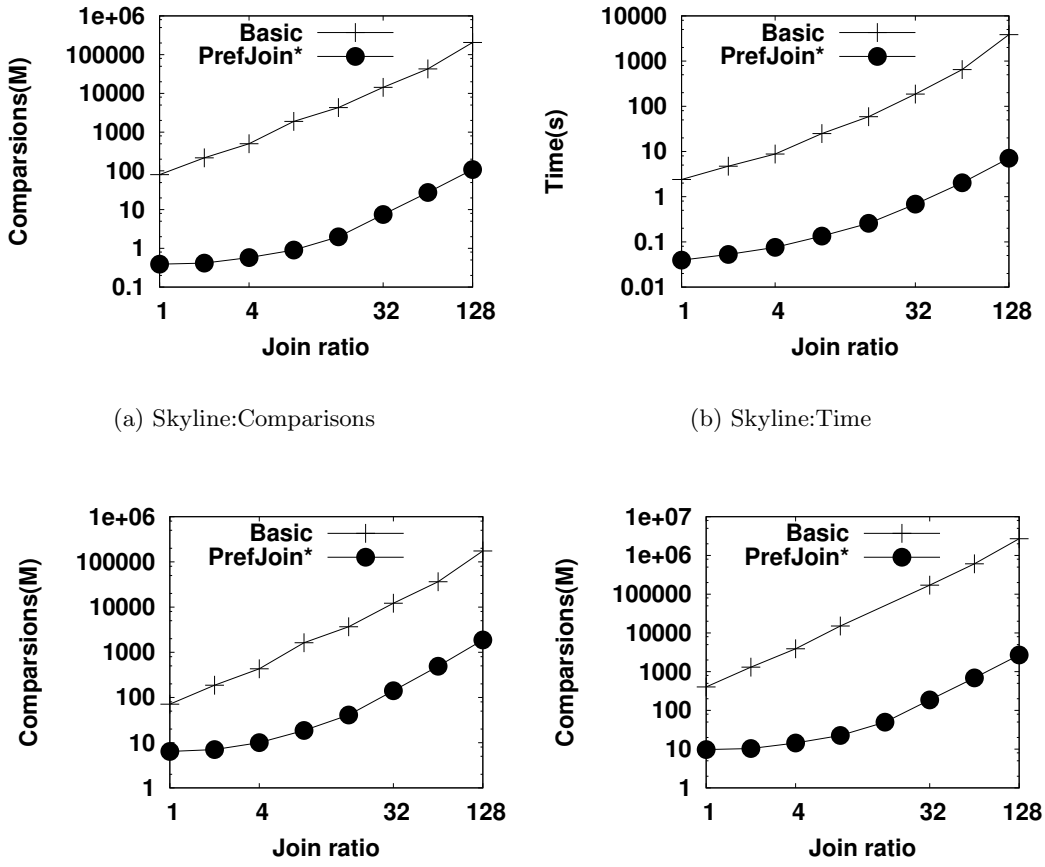
(d) Top-K:Comparisons

Figure 5.6: Number of Groups

respect to *Basic* algorithm. Overall, the *PrefJoin** algorithm exhibits at least two orders of magnitude better performance than *Basic* algorithms.

### Join Ratio

Figure 5.7 increases the join ratio between relations $R$ and $S$, i.e., how many tuples in $S$ will be joined with a single tuple from $R$. The cardinality of $R$ is set to 20K, while the join ratio is increased from 1:1 to 1:256. With the increase of the join ratio, execution time and comparisons increase as the size of the final preference set increases




(a) Skyline:Comparisons

(b) Skyline:Time

(c) Multi-Objective:Comparisons

(d) Top-K:Comparisons

Figure 5.7: Join ratio

from 1320 to 111K. *PrefJoin**, consistently, has one to two orders of magnitude better performance than *Basic* algorithm, for all join ratios. This is due to: (a) The final preference set size increases; while *PrefJoin** avoids preference comparisons over these tuples, the other algorithms do not, and (b) computing the object probabilities in $R$, which avoids unneeded uncertain probability computation in $S$.

## 5.5 Conclusion

This chapter summarizes change needed to *PrefJoin* to be able to support imprecise data. We tightly integrate the preference computations over uncertain and incomplete data with the join operator. An interesting characteristic of *PrefJoin** is that it aims to join only those tuples that are guaranteed to be an answer, and hence: (a) saves computation costs by not joining unnecessary tuples, and (b) saves expensive preference computations by applying the preference function over those tuples that may needlessly be joined. *PrefJoin** supports a variety of preference function including skyline, multi-objective and $k$-dominance preference queries, by appropriately defining the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for each preference function. Experimental evaluation based on a system implementation of *PrefJoin** and its competitor inside PostgreSQL shows that *PrefJoin** consistently achieves one to three orders of magnitude performance gain over its competitors in various scenarios.

# Chapter 6

# Conclusion

Preference queries are essential to find the *relevant* results to users. And with the growing number of applications that generate *imprecise* data, e.g., sensor readings, human reading errors, and data imperfection, it has become essential to support preference queries of various types over imprecise data. We classified *Imprecise* data into two categories: *incomplete* and *uncertain* data. This thesis addresses efficiently extending DBMSs to be preference-aware over imprecise data. The proposed systems are implemented in PostgreSQL.

First, we addressed the problem of skyline queries over *incomplete* data where multi-dimensional data items are missing some values of their dimensions. We showed that with *incomplete* data, the dominance relation among data points may not be transitive, thus, almost all existing techniques for skyline queries are not applicable. We proposed the *ISkyline* algorithm that is designed specifically for *incomplete* data. The *ISkyline* algorithm employs two optimization techniques, namely *virtual points* and *shadow skylines* to exploit the properties of *incomplete*.

Then, we have defined preference queries over uncertain data, and proposed a novel, efficient framework to answer these preference queries. Query answers are probabilistic, where each object is associated with a probability value of being a preferred answer. Users can specify a probability threshold, that each preferred object must exceed, and a tolerance that defines the allowed error margin in probability calculation. We have proposed four methods to bound each object probability for being a preferred object, namely, we have proposed *uncertainty reduction*, *pairwise comparison*, *segmentation*,

and *bound tightening*. Then, we presented a two-phase framework that encapsulates our four proposed methods together using a filter-refine approach. We extended our framework to support multi-dimensional and non-uniform pdf uncertain data.

Then, we proposed *PrefJoin*, an efficient preference-aware join query operator, designed specifically to deal with preference queries where the set of preferred attributes reside in more than one relation. The main idea of *PrefJoin* is to make the join operator aware of the required preference functionality, and hence inject the ability to early prune those tuples that have no chance of being a preferred tuple. *PrefJoin* consists of four main phases: *Local Pruning*, *Data Preparation*, *Joining*, and *Refining* that filter out, from each input relation, those tuples that are guaranteed not to be in the final preference set, associate meta data with each non-filtered tuple that will be used to optimize the execution of the next phases, produce a subset of join result that are relevant for the given preference function, and refine these tuples respectively.

We summarized change needed to extend *PrefJoin* to be able to support imprecise data. We tightly integrate the preference computations over uncertain and incomplete data with the join operator. An interesting characteristic of *PrefJoin** is that it aims to join only those tuples that are guaranteed to be an answer, and hence: (a) saves computation costs by not joining unnecessary tuples, and (b) saves expensive preference computations by applying the preference function over those tuples that may needlessly be joined. *PrefJoin** supports a variety of preference function including skyline, multi-objective and $k$-dominance preference queries, by appropriately defining the $\mathcal{P}_{local}$, $\mathcal{P}_{pairwise}$, and $\mathcal{P}_{refine}$ for each preference function.

The correctness of the proposed frameworks are proved in terms that produce only and all preferred objects. Experimental evaluation based on a system implementation of our proposed frameworks show that they are scalable and efficient. Practically, *PrefJoin* and *PrefJoin** consistently achieve one to three orders of magnitude performance gain over its competitors in various scenarios.

# References


[1] Surajit Chaudhuri and Luis Gravano. Evaluating Top-k Selection Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1999.

[2] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 421–430, April 2001.

[3] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. Efficient Distributed Skylining for Web Information Systems. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 256–273, March 2004.

[4] Mehdi Sharifzadeh and Cyrus Shahabi. The Spatial Skyline Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 751–762, September 2006.

[5] Wolf-Tilo Balke and Ulrich Güntzer. Multi-objective Query Processing for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 936–947, Toronto, Canada, August 2004.

[6] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 503–514, Chicago, IL, June 2006.

[7] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. On High Dimensional Skylines. In *Proceedings of the International*

*Conference on Extending Database Technology, EDBT*, pages 478–495, Munich, Germany, March 2006.

[8] Jongwuk Lee, Gae-won You, and Seung-won Hwang. Personalized Top-K skyline queries in high-dimensional space. *Information Systems*, 34(1):45–61, 2009.

[9] Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. Selecting Stars: The $k$ Most Representative Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[10] Yufei Tao, Ling Ding, Xuemin Lin, and Jian Pei. Distance-based representative skyline. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2009.

[11] Tian Xia, Donghui Zhang, and Yufei Tao. On skylining with flexible dominance relation. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 1397–1399, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Man Lung Yiu and Nikos Mamoulis. Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2007.

[13] Xuemin Lin, Ying Zhang, Wenjie Zhang, and Muhammad Aamir Cheema. Stochastic Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.

[14] Werner Kießling. Foundations of Preferences in Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 311–322, Hong Kong, China, September 2002.

[15] Georgia Koutrika and Yannis E. Ioannidis. Personalization of Queries in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 597–608, Boston, MA, April 2004.

[16] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 259–270, Santa Barbara, CA, June 2001.

[17] Gerhard Köstler Werner Kießling. Preference SQL - Design, Implementation, Experiences. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 990–1001, Hong Kong, China, September 2002.

[18] Jan Chomicki. Database Querying under Changing Preferences. *Annals of Mathematics and Artificial Intelligence*, 2006.

[19] Jan Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems, TODS*, 28(4):427–466, 2003.

[20] Rakesh Agrawal and Edward L. Wimmers. A Framework for Expressing and Combining Preferences. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 297–306, Dallas, TX, May 2000.

[21] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference Queries in Deductive Databases. *New Generation Computing*, 19(1):57–86, 2000.

[22] M. Lacroix and Pierre Lavency. Preferences: Putting More Knowledge into Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 217–225, Brighton, England, September 1987.

[23] Georgia Koutrika and Yannis E. Ioannidis. Personalized Queries under a Generalized Preference Model. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 841–852, Tokyo, Japan, April 2005.

[24] Kostas Stefanidis, Evaggelia Pitoura, and Panos Vassiliadis. Adding Context to Preferences. In *Proceedings of the International Conference on Data Engineering, ICDE*, Istanbul, Turkey, April 2007.

[25] Kostas Stefanidis, Evaggelia Pitoura, and Panos Vassiliadis. A Context-Aware Preference Database System. *International Journal of Pervasive Computing and Communications*, 2006.

[26] Arthur H. van Bunningen, Ling Feng, and Peter M. G. Apers. A Context-Aware Preference Model for Database Querying in an Ambient Intelligent Environment. In *International Conference of Database and Expert Systems Applications, DEXA*, pages 33–43, Krakow, Poland, September 2006.

[27] George Beskales, Mohamed A. Soliman, and Ihab F. Ilyas. Efficient Search for the Top-k Probable Nearest Neighbors in Uncertain Databases. In *VLDB*, 2008.

[28] Reynold Cheng, Jinchuan Chen, Mohamed F. Mokbel, and Chi-Yin Chow. Probabilistic Verifiers: Evaluating Constrained Nearest-Neighbor Queries over Uncertain Data. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2008.

[29] Jian Pei, Bin Jiang, Xuemin Lin, and Yidong Yuan. Probabilistic Skylines on Uncertain Data. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2007.

[30] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. Efficiently Answering Probabilistic Threshold Top-k Queries on Uncertain Data. In *Proceedings of the International Conference on Data Engineering, ICDE*, Cancu, Mexico, April 2008.

[31] Christopher Re, Nilesh Dalvi, and Dan Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In *Proceedings of the International Conference on Data Engineering, ICDE*, Istanbul, Turkey, April 2007.

[32] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Top-k Query Processing in Uncertain Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, Istanbul, Turkey, April 2007.

[33] Wen Jin, Martin Ester, Zengjian Hu, and Jiawei Han. The Multi-Relational Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2007.

[34] Justin J. Levandoski, Mohamed F. Mokbel, and Mohamed E. Khalefa. FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[35] Subi Arumugam, Ravi Jampani, Luis L. Perez, Fei Xu, Christopher Jermaine, and Peter J. Haas. Mcdbr:risk analysis in the database. In *VLDB*, 2010.

[36] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. U-DBMS: a database system for managing constantly-evolving data. In *In VLDB 05: Proceedings of the 31st international conference on Very large data bases*, pages 1271–1274, 2005.

[37] Christoph Koch, Lyublena Antova, Jiewen Huang, Thomas Jansen, Ali Baran, and Sari Maybms. Maybms: A system for managing large uncertain and probabilistic databases. In *Managing and Mining Uncertain Data, chapter 6*. Springer-Verlag, 2008.

[38] Sarvjeet Singh, Chris Mayfield, Rahul Shah, Sunil Prabhakar, Susanne Hambrusch, Jennifer Neville, and Reynold Cheng. Database support for probabilistic attributes and tuples. In *International Conference on Data Engineering*, 2008.

[39] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of ACM*, 22(4):469–476, 1975.

[40] Jirí Matousek. Computing Dominances in $E^n$. *Information Processing Letters*, 38(5):277–278, 1991.

[41] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal Vector Computation in Large Data Sets. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 229–240, Trondheim, Norway, August 2005.

[42] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 717–816, March 2003.

[43] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 275–286, Hong Kong, September 2002.

[44] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems, TODS*, 30(1):41–82, 2005.

[45] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient Progressive Skyline Computation. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 301–310, Rome, Italy, September 2001.

[46] http://movielens.umn.edu.

[47] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient Computation of the Skyline Cube. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 241–252, Trondheim, Norway, August 2005.

[48] Zhenjie Zhang, Xinyu Guo, Hua Lu, Anthony K. H. Tung, and Nan Wang. Discovering Strong Skyline Points in High Dimensional Spaces. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, pages 247–248, Bremen, Germany, November 2005.

[49] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *The VLDB journal*, 16(1):5–28, 2007.

[50] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 467–478, San Diego, CA, June 2003.

[51] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Salsa: computing the skyline without scanning the whole sky. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, 2006.

[52] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 227–238, New York, NY, USA, 2008. ACM.

[53] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 203–214, Baltimore, MD, June 2005.

[54] Jian Pei, Wen Jin, Martin Ester, and Yufei Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 253–264, Trondheim, Norway, August 2005.

[55] Yufei Tao, Xiaokui Xiao, and Jian Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *Proceedings of the International Conference on Data Engineering, ICDE*, Atlanta, GA, April 2006.

[56] Tian Xia and Donghui Zhang. Refreshing the Sky: The Compressed Skycube with Efficient Support for Frequent Updates. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 491–502, Chicago, IL, June 2006.

[57] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 502–513, April 2005.

[58] Yufei Tao and Dimitris Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 18(2):377–391, 2006.

[59] Evangelos Dellis and Bernhard Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302. VLDB Endowment, 2007.

[60] Xiang Lian and Lei Chen. Monochromatic and Bichromatic Reverse Skyline Search over Uncertain Databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 213–226, Vancouver, Canada, June 2008.

[61] Wolf-Tilo Balke, Ulrich Güntzer, and Wolf Siberski. Restricting Skyline Sizes Using Weak Pareto Dominance. *Informatik-Forschung und Entwicklung*, 21(3-4):165–178, 2007.

[62] Zhiyong Huang, Hua Lu, Beng Chin Ooi, and Anthony K.H. Tung. Continuous Skyline Queries for Moving Objects. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 18(12):1645–1658, 2006.

[63] Michael D. Morse, Jignesh M. Patel, and William I. Grosky. Efficient Continuous Skyline Computation. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2006.

[64] Zhiyong Huang, Christian S. Jensen, Hua Lu, and Beng Chin Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2006.

[65] Pin-Kwang Eng, Beng Chin Ooi, and Kian-Lee Tan. Indexing for Progressive Skyline Computation. *Data and Knowledge Engineering*, 46(2):169–201, 2003.

[66] Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, Atlanta, GA, April 2006.

[67] Wen Jin, Jiawei Han, and Martin Ester. Mining Thick Skylines over Large Databases. In *PKDD*, pages 255–266, Pisa, Italy, September 2004.

[68] http://www.basketball-reference.com/.

[69] Yinian Qi, Rohit Jain, Sunil Prabhakar, and Sarvjeet Singh. Threshold Query Optimization for Uncertain Data. In *SIGMOD*, 2010.

[70] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Evaluating Probabilistic Queries over Imprecise Data. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 551–562, San Diego, CA, June 2003.

[71] Christoph Koch and Dan Olteanu. Conditioning Probabilistic Databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2008.

[72] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 369–380, San Jose, CA, February 2002.

[73] Surajit Chaudhuri, Luis Gravano, and Amélie Marian. Optimizing Top-k Selection Queries over Multimedia Repositories. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 16(8):992–1009, 2004.

[74] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid, Hicham G. Elmongui, Rahul Shah, and Jeffrey Scott Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Transaction Database System*, 31(4):1257–1304, 2006.

[75] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Probabilistic Top-k and Ranking-Aggregate Queries. *ACM Trans. Database Syst.*, 33(3), 2008.

[76] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective Query Optimization. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, Santa Barbara, CA, June 2001.

[77] Hyung-Ju Cho and Chin-Wan Chung. An Efficient and Scalable Approach to CNN Queries in a Road Network. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 865–876, Trondheim, Norway, August 2005.

[78] Glenn S. Iwerks, Hanan Samet, and Ken Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 512–523, Berlin, Germany, September 2003.

[79] Mohammad R. Kolahdouzan and Cyrus Shahabi. Alternative Solutions for Continuous K Nearest Neighbor Queries in Spatial Network Databases. *GeoInformatica*, 9(4):321–341, December 2005.

[80] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 634–645, Baltimore, MD, June 2005.

[81] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous Nearest Neighbor Monitoring in Road Networks. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 43–54, Seoul, Korea, September 2006.

[82] Man Lung Yiu, Dimitris Papadias, Nikos Mamoulis, and Yufei Tao. Reverse Nearest Neighbors in Large Graphs. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 18(4):540–553, 2006.

[83] Flip Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 201–212, Dallas, TX, 2000.

[84] Elke Achtert, Christian Bahm, Peer Krager, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Efficient Reverse k-Nearest Neighbor Search in Arbitrary Metric Spaces. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, Chicago, IL, 2006.

[85] Man Lung Yiu and Nikos Mamoulis. Reverse Nearest Neighbors Search in Ad-hoc Subspaces. In *Proceedings of the International Conference on Data Engineering, ICDE*, Atlanta, GA, 2006.

[86] Congjun Yang and King-Ip Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 485–492, Heidelberg, Germany, 2001.

[87] Amit Singh, Hakan Ferhatosmanoglu, and Ali Saman Tosun. High Dimensional Reverse Nearest Neighbor Queries. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, pages 91–98, New Orleans, LA, 2003.

[88] Yufei Tao, Dimitris Papadias, and Xiang Lian. Reverse kNN Search in Arbitrary Dimensionality. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 744–755, Toronto, Canada, 2004.

[89] Justin J. Levandoski, Mohamed F. Mokbel, and Mohamed E. Khalefa. CareDB: A Context and Preference-Aware Location-Based Database System. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2010.

[90] Justin J. Levandoski and Mohamed F. Mokbel. Toward context and preference-aware location-based services. In *MobiDE*, 2009.

[91] Mohamed E. Khalefa, Mohamed F. Mokbel, and Justin J. Levandoski. PrefJoin: An Efficient Preference-aware Join Operator. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.

[92] Justin J. Levandoski, Mohamed F. Mokbel, and Mohamed E. Khalefa. Preference Query Evaluation Over Expensive Attributes. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, 2010.

[93] Robert V. Hogg, Joseph W. McKean, and Allen T. Craig. *Introduction to mathematical statistics*. Prentice Hall, 2004.

[94] Michael Stonebraker and Lawrence A. Rowe. The Design of Postgres. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1986.

[95] Michael Stonebraker, Jeff Anton, and Michael Hirohama. Extendability in POSTGRES. In *IEEE Data Engineering Bulletin*, 1987.

[96] Michael Stonebraker and Greg Kemnitz. The Postgres Next Generation Database Management System. In *Communications of the ACM*, 1991.

[97] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2005.

[98] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Efficient Processing of Skyline Queries with Partially-Ordered Domains. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 190–191, Tokyo, Japan, April 2005.

[99] Ping Wu, Caijie Zhang, Ying Feng, Ben Y. Zhao, Divyakant Agrawal, and Amr El Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, pages 112–130, Munich, Germany, March 2006.

[100] Mikhail J. Atallah and Yinian Qi. Computing all skyline probabilities for uncertain data. In *PODS*, pages 279–287, New York, NY, USA, 2009. ACM.

[101] Wen Jin, Michael D. Morse1, Jignesh M. Patel, Martin Ester, and Zengjian Hu. Evaluating Skylines in the Presence of Equijoins. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[102] Venkatesh Raghavan and Elke A. Rundensteiner. Progressive Result Generation for Multi-Criteria Decision Support Queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2010.

[103] PostgreSQL: http://www.postgresql.org.

[104] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *VLDB Journal*, 13(3):207–221, 2004.

[105] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[106] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for TopX search. In *VLDB*, 2005.

[107] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware Query Optimization. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 203–214, Paris, France, June 2004.

[108] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 346–357, Madison, WI, June 2002.

[109] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Transactions on Database Systems, TODS*, 27(2):153–187, 2002.

[110] Michael J. Carey and Donald Kossmann. On saying "Enough Already!" in SQL. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1997.

[111] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, pages 102–113, Santa Barbara, CA, June 2001.

[112] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: A Framework for Distributed Top-K Query Algorithms. In *VLDB*, 2005.

[113] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 281–290, Rome, Italy, September 2001.

[114] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961. VLDB Endowment, 2002.

[115] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765. VLDB Endowment, 2003.

[116] Dalie Sun, Sai Wu, Jianzhong Li, and A.K.H. Tung. Skyline-join in distributed databases. In *ICDEW*, 2008.

[117] Keping Zhao, Yufei Tao, and Shuigeng Zhou. Efficient top-k processing in large-scaled distributed environments. *Data Knowledge Engineering*, 63(2):315–335, 2007.

[118] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 131–142, New York, NY, USA, 2005. ACM.

[119] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):1–58, 2008.

[120] Hicham G. Elmongui and Walid G. Aref. Skyline-Aware Join Operator. Technical Report CSD TR 08-007, Purdue University, 2008.

[121] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *ICDE*, 2004.

[122] Justin J. Levandoski, Mohamed E. Khalefa, and Mohamed F. Mokbel. On producing high and early result throughput in multi-join query plans. In *TKDE*, 2010.

[123] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems, TODS*, 11(3):239–264, 1986.

[124] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[125] Reynold Cheng, Sarvjeet Singh, Sunil Prabhakar, Rahul Shah, Jeffrey Scott Vitter, and Yuni Xia. Efficient join processing over uncertain data. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, CIKM '06, pages 738–747, New York, NY, USA, 2006. ACM.

[126] T. Ge. Join Queries on Uncertain Data: Semantics and Efficient Processing. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.