Ph.D. DISSERTATION

Efficient Parallel Processing of Skyline Queries for Big Data

빅데이터의 효율적인 스카이라인 질의 처리를 위한 병렬처리 알고리즘

BY

Park, Yoonjae

Feburary 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Efficient Parallel Processing of Skyline Queries for Big Data

빅데이터의 효율적인 스카이라인 질의 처리를 위한 병렬처리 알고리즘

BY

Park, Yoonjae

Feburary 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Efficient Parallel Processing of Skyline Queries for Big Data

빅데이터의 효율적인 스카이라인 질의 처리를 위한 병렬처리 알고리즘

지도교수 심 규 석
이 논문을 공학박사 학위논문으로 제출함

2017년 2월

서울대학교 대학원

전기 컴퓨터 공학부

박 윤 재

박윤재의 공학박사 학위 논문을 인준함

2017년 2월

위 원 장: 김 형 주 (인)

부위원장: 심 규 석 (인)

위 원: 이 상 구 (인)

위 원: 홍 성 수 (인)

위 원: 민 준 기 (인)

Abstract

The skyline operator and its variants such as dynamic skyline, reverse skyline and probabilistic skyline operators have attracted considerable attention recently due to its broad applications. However, computing a skyline is challenging today since we have to deal with big data. For data-intensive applications, the MapReduce framework has been widely used recently.

In this dissertation, we propose the efficient parallel algorithms for processing skyline, dynamic skyline, reverse skyline and probabilistic skyline queries using MapReduce. For the skyline, dynamic skyline and reverse skyline queries, we first build quadtree-based histograms to prune out non-skyline points. We next partition data based on the regions divided by the histograms and compute candidate skyline points for each partition using MapReduce. Finally, in every partition, we check whether each skyline candidate point is actually a skyline point or not using MapReduce. For the probabilistic skyline query, we first introduce three filtering techniques to prune out points that are not probabilistic skyline points. Then, we build a quadtree-based histogram and split data into partitions according to the regions divided by the quadtree. We finally compute the probabilistic skyline points for each partition using MapReduce. We also develop the workload balancing methods to make the estimated execution times of all available machines to be similar. We did experiments to compare our algorithms with the state-of-the-art algorithms using MapReduce and confirmed the effectiveness as well as the scalability of our proposed skyline algorithms.

keywords: Skyline queries, reverse skyline queries, probabilistic skyline queries, parallel algorithms, MapReduce algorithms

student number: 2011-20842

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

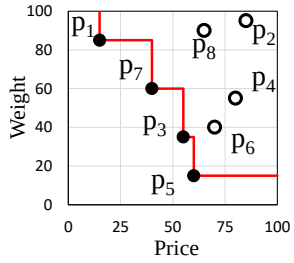
1.1 Motivation

The skyline operator [10] and its variants such as dynamic skyline [38], reverse skyline [16] and probabilistic skyline [41] operators have recently attracted considerable attention due to their broad applications including product or restaurant recommendations [30, 31], preference-based marketing [16, 55], review evaluations with user ratings [28], querying wireless sensor networks [53] and graph analysis [61].

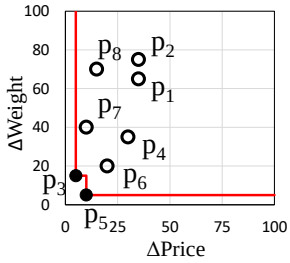
Given a d-dimensional data set $\mathbb{D} = \{p_1, p_2, \ldots, p_{|\mathbb{D}|}\}$, the *skyline* of $\mathbb{D}$ is composed of the points, called *skyline points*, which are not dominated by any other point. A point p_i is said to *dominate* another point p_j if p_i is not greater than p_j in all dimensions and p_i is smaller than p_j in at least a single dimension. For example, consider a laptop database $\mathbb{D}$ with $price$ and $weight$ attributes in Figure 1.1(a). We plot every laptop in $\mathbb{D}$ into a 2-dimensional space where the horizontal axis indicates the $price$, and the vertical axis represents the $weight$ in Figure 1.1(b). The laptop $p_7 = \langle 40, 60 \rangle$ dominates another laptop $p_8 = \langle 65, 90 \rangle$ according to the definition of the dominance relationship. A user who would like to buy a cheap and light-weight laptop can consider the laptops in the skyline $\{p_1, p_3, p_5, p_7\}$, since there always exists a better laptop in the skyline for any laptop which is not contained in the skyline. Each laptop in the

ID	Price	Weight
p_1	15	85
p_2	85	95
p_3	55	35
p_4	80	55
p_5	60	15
p_6	70	40
p_7	40	60
p_8	65	90

(a) The laptop data set $\mathbb{D}$



(b) The skyline of $\mathbb{D}$



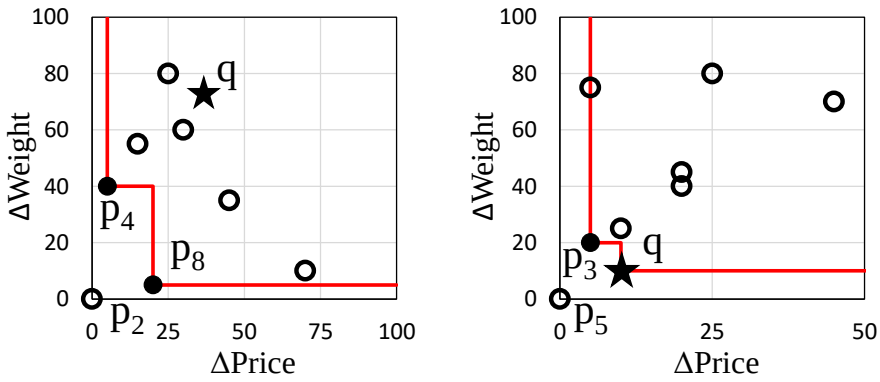
(c) The dynamic skyline of $\mathbb{D}$ with respect to $q = \langle 50, 20 \rangle$

Figure 1.1: An example of a skyline and a dynamic skyline

skyline is represented by a black circle in Figure 1.1(b).

When the dominance relationship is defined with respect to a user's query point, the skyline becomes *a dynamic skyline*. Given a query point, the dynamic skyline is a set of all points that are not dynamically dominated by any other point with respect to the query point. A point p_i dynamically dominates another point p_j with respect to a query point q if (1) p_i has closer values to q than p_j in all dimensions and (2) p_i has strictly closer values to q than p_j in at least a single dimension. For example, when a user wants to find a laptop whose price and weight are close to 50 and 20 respectively, the dynamic skyline of $\mathbb{D}$ with respect to a query point $\langle 50, 20 \rangle$ is the set of candidate laptops to be purchased. Note that, after transforming all data points to the points consisting of coordinate-wise distances to the query point $\langle 50, 20 \rangle$, the dynamic skyline is equal to the skyline of the transformed points. We plot the transformed points in Figure 1.1(c) and denote the dynamic skyline with respect to $\langle 50, 20 \rangle$ (i.e., $\{p_3, p_5\}$) by black circles. The user will purchase the laptop p_3 or p_5 since they are not dynamically dominated by other laptops with respect to the user's preference.

While the skyline and the dynamic skyline queries retrieve the products that users like, a *reverse skyline query* focuses on companies' perspective. Suppose that each point p_i represents a user who purchased a laptop with its price and weight shown in Figure 1.1(a) and a company wants to estimate the sales of a new laptop to be



(a) The dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_2\}$ with respect to p_2

(b) The dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_5\}$ with respect to p_5

Figure 1.2: The dynamic skylines with respect to p_2 and p_5

manufactured whose price and weight will be 50 and 25 respectively. We denote the new laptop by a query point $q = \langle 50, 25 \rangle$. If the query point q belongs to the dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_i\}$ with respect to a point p_i, we assume that the user p_i finds the laptop q interesting. The reverse skyline of $\mathbb{D}$ with respect to q is defined as the set of every point $p_i \in \mathbb{D}$ such that q belongs to the dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_i\}$ with respect to p_i. In other words, the reverse skyline of $\mathbb{D}$ with respect to q is the set of all customers who will be interested in q. For instance, reconsider the data set $\mathbb{D}$ in Figure 1.1(a). Given a query point $q = \langle 50, 25 \rangle$, we plot the dynamic skylines with respect to p_2 and p_5 in Figures 1.2(a) and 1.2(b), respectively. Since q does not belong to the dynamic skyline with respect to p_2 (i.e., $\{p_4, p_8\}$), the point p_2 is not contained in the *reverse skyline* with respect to q. However, the point p_5 is in the reverse skyline with respect to q because the dynamic skyline with respect to p_5 (i.e., $\{p_3, q\}$) contains q.

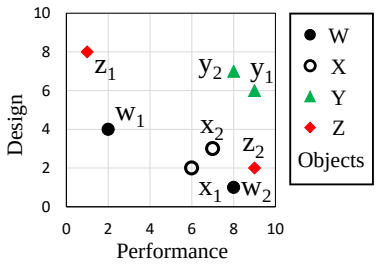
Since there has been an increased growth recently in a number of applications such as social network [1], data integration [19] and sensor data management [17] that naturally produce large volumes of *probabilistic/uncertain data*, another variant of the skyline query, called a *probabilistic skyline query*, was introduced in [41] for uncer-

tain objects. The uncertainty is inherent due to various factors such as data randomness and incompleteness, limitations of measuring equipments and so on. An *uncertain object* can be described by the discrete or continuous uncertainty model. In the discrete model, an object U is modeled as a set of instances and denoted by $U = \{u_1, u_2, \ldots, u_{|U|}\}$ where u_i is a d-dimensional point with its existence probability. In the continuous model, an object U is modeled as an *uncertainty region* with its probabilistic distribution function (pdf). Given a set of uncertain objects $\mathbb{D}$ represented by the discrete model, a *possible world* is a set of instances from objects in $\mathbb{D}$ where at most a single instance may be selected from each object. The *skyline probability* of an instance is the probability that it appears in a possible world and is not dominated by every instance of the other objects in the possible world. Then, the skyline probability of an object is the sum of the skyline probabilities of its all instances. Similarly, for the continuous model, we define the *skyline probability* of an object by using its uncertainty region and pdf. Given a *probability threshold* T_p, regardless of the uncertainty models used, the probabilistic skyline is the set of uncertain objects whose skyline probabilities are at least T_p.

The probabilistic skyline query has many practical applications. Consider an online shopping website such as Amazon. In the website, users can purchase a laptop and rate the laptop they bought. Users can evaluate the performance and design of the laptop they bought where scores are between 1(good) and 10(bad). Each laptop gets different reviews from the users since the viewpoints vary from person to person. We can model each laptop as a discrete uncertain object and each review of the laptop can be viewed as an instance of the objects. For example, the ratings of laptops W, X, Y and Z are represented by the discrete uncertain model in Figure 1.3(a). The instance w_1 of the laptop W represents that 50% of users rated the performance and design of laptop W as 2 and 4, respectively. We plot each instance in Figure 1.3(b). The probabilistic skyline can be used to find out which laptop is probabilistically better than others. Given a minimum probability threshold $T_p = 0.5$, the laptops W and Z are

Object	Instances	Performance	Design	Probability
W	w_1	2	4	0.5
	w_2	8	1	0.4
X	x_1	6	2	0.2
	x_2	7	3	0.2
Y	y_1	9	6	0.8
	y_2	8	7	0.2
Z	z_1	1	8	0.5
	z_2	9	2	0.5

(a) The rating of laptops



(b) Uncertain objects

Figure 1.3: An example of a probabilistic skyline

probabilistic skyline objects which are probabilistically better than the other laptops.

In the following, we present some relevant applications which require the skyline query and its variant queries.

Recommending products: The skyline, dynamic skyline and probabilistic skyline queries help customers to reduce the number of candidates to consider as shown in the previous examples. Without considering all items in the database, the customers can make decisions quickly among the items in the skyline only. By utilizing the rapidly growing social web which has been a source of vast amount of data concerning user preferences in the form of ratings, the dynamic skyline queries is useful for personalized recommendation based on the user preference.

Marketing based on user preferences: The reverse skyline can be utilized to estimate the sales of a new product to be manufactured based on the preferences of the customers. It can be used to decide the specification of the new product or the location of a new store. For example, among the several possible location for the new store, the reverse skyline returns the estimated number of customers who are potentially interested in the new store. Thus, we can select the location which maximizes the estimated number of customers.

Making decisions with sensor networks: The modern world is full of devices with sensors and processors. Such deployments of computational resources enable us to measure, collect and process large data from billions of connected devices serv-

ing many applications. Since a common characteristics of such sensors is that every measured value is associated with some measurement error, the probabilistic skyline is useful on sensor networks. For example, consider a large number of devices equipped with sensors to measure NO_2 and SO_2 concentrations in the air and deployed in a wide area to monitor the air pollution. To determine the locations of a new air purifier to reduce the air pollution, we can consider the locations of the devices whose pairs of measured NO_2 and SO_2 values are in the probabilistic skyline.

Computing the skyline and its variants becomes more challenging problems today as there is an increasing trend of applications which expect to deal with vast amounts of data that usually do not fit in the main memory of one machine. For frequent pattern mining [45, 49] and graph mining [39], the skyline operator is used as a primitive operator and computes skyline of patterns and sub-graphs represented by multiple features. Since the numbers of patterns and sub-graphs increase exponentially as the sizes of graph and pattern increase, the patterns and sub-graphs can be considered as a big data. For such data-intensive applications, Google's MapReduce [15] and its open-source equivalent Hadoop [4] have been considered as a *de facto* standard. MapReduce is a powerful and widely used tool that provides easy development of scalable parallel applications such as large-scale graph processing, text processing and machine learning to process big data on large clusters of commodity machines. At Google, more than 10,000 distinct programs have been implemented using MapReduce [15]. Thus, we develop the parallel skyline, dynamic skyline, reverse skyline and probabilistic skyline algorithms using the MapReduce framework.

1.2 Contributions of This Dissertation

In this dissertation, we propose efficient query processing algorithms for skyline and its variant queries discussed in our motivating applications. Our contributions are as follows:

- We first study the optimization of skyline query processing. We propose an efficient parallel skyline computation algorithm which consists of three phases. In the first phase, we build a new histogram which is an extension of quadtrees [20] to effectively prune out non-skyline points in advance. In the second phase, we split data into partitions based on the regions divided by our proposed histograms and compute candidate skyline points for each partition independently using MapReduce. Finally, we check whether each candidate point is actually a skyline point in every region independently by another MapReduce phase. Although our proposed algorithms are devised for the MapReduce framework, they can be also applied to other frameworks such as MPI [27] and multi-cores. Since the dynamic skyline can be obtained by calculating the skyline after transforming the coordinates of data points with respect to a given query point, we can utilize our parallel skyline computation algorithm to compute the dynamic skyline.

- We next investigate the reverse skyline query processing. To the best of our knowledge, no existing work has addressed computing the reverse skyline query using MapReduce. We analyze the characteristics of the reverse skylines theoretically to prune non-reverse skyline points. Based on the properties of the reverse skylines, we develop the novel parallel algorithm consisting of three phases. In the first phase, we build a variant of quadtree which is used for pruning non-reverse skyline points by utilizing the characteristics. In the second phase, by using MapReduce, we compute the local reverse skyline points in each partition split by the histogram. In the last phase, we compute the global reverse skyline points in every region independently and simultaneously by using MapReduce.

- We finally present the efficient algorithm for computing the probabilistic skyline query for both continuous and discrete uncertain models. To prune out non-probabilistic skyline objects in advance, we develop three filtering methods. The

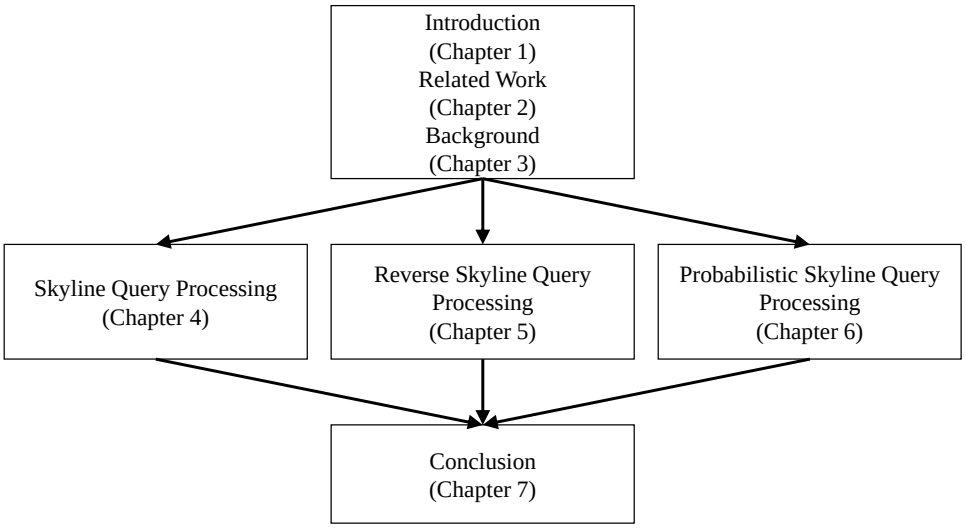


Figure 1.4: Dissertation overview

proposed algorithms are composed of only two phases. In the first phase, we build a variant of a quadtree. In the second phase, by utilizing the proposed filtering methods, we efficiently compute the probabilistic skyline in each partition according to the space split by the variant of a quadtree. To balance the workload and reduce the transmission overhead, we also propose a workload balancing technique for the second phase.

Since the skyline queries studied in this dissertation are widely required in many applications such as product or restaurant recommendations, review evaluations with user ratings, querying wireless sensor networks and graph analysis, our proposed algorithm will enhance the performance of those applications practically.

1.3 Dissertation Overview

The remaining chapters of this dissertation is organized as follows. In Chapter 2, we review the previous works on skyline query processing. In Chapter 3, we provide the background to this dissertation which is commonly utilized in this dissertation. In

Chapter 4, we study the problem of skyline query processing and expend it to the dynamic skyline query processing. In Chapter 5, we investigate the problem of finding reverse skyline of a given query point. In Chapter 6, we examine the problem of probabilistic skyline query processing. We finally present our conclusions and future work in Chapter 7. The diagram in Figure 1.4 outlines how the chapters of this dissertation are related to each other.

Chapter 2

Related Work

In this chapter, we present related work on skyline and its variant query processing. We first discuss the algorithms for computing skylines and next study reverse skyline and probabilistic skyline query processing techniques. In Table 2.1, we summarized the related works on skyline, reverse skyline and probabilisitic skyline query processing.

2.1 Skyline Queries

After skyline processing was introduced in [10], several serial algorithms for computing skylines were introduced in [6, 7, 13, 26, 29, 38, 47] to improve the performance of skyline processing. The work in [10] presents skyline algorithms such as block-nest-loop (BNL) and divide-and-conquer (D&C). The sort-filter skyline (SFS) algorithm [13] improves BNL using presorted data set according to the scores computed by a monotone function. Linear-elimination-sort for skyline (LESS) [22] is an optimized algorithm of SFS by eliminating a portion of a data set during sorting. Tan et al. [47] first proposed the progressive techniques, called Bitmap and Index that progressively return skyline points as they are identified unlike most existing algorithms that require at least one pass over the dataset to return the first skyline point. By exploiting R*-tree [8], an improved algorithm, called NN, based on the nearest neighbor search was

Queries	Serial algorithms	Parallel algorithms
Skyline query	[6], [7], [13], [26], [29], [38], [47]	[2], [25], [37], [40], [56], [57]
Reverse skyline query	[16],[21], [55], [59]	[53]
Probabilistic skyline query	[5], [33], [41]	[18]

Table 2.1: The related works on skyline and its variant queries

presented in [26]. Papadias et al. [38] improved NN using a branch-and-bound method (BBS) and also introduced the dynamic skyline query.

After the dynamic skyline was introduced in [38], variants of the skyline queries such as the reverse skylines [16], probabilistic skylines [33, 41], top-k frequent skylines [12], spatial skylines [43], continuous skylines [5, 48, 58], and stochastic skylines [34] have been also introduced. Although many existing serial algorithms utilize centralized indexing structures such as B^+-trees and R*-trees to check whether a point belongs to the skyline or not, such algorithms are not suitable to be parallelized using MapReduce since MapReduce does not provide the functionality for building and accessing distributed indexing structures.

Although we focus on computing the skyline using MapReduce, we still need a serial skyline algorithm to calculate the local skyline for each partition. Thus, among the serial skyline algorithms [6, 10, 13, 29] without using centralized indexes, we adopt the state-of-the-art algorithm *BSkyTree-P* [29]. *BSkyTree-P* calculates a skyline in a divide-and-conquer manner. To split the data space into 2^d partitions, *BSkyTree-P* first selects a pivot point which reduces the number of checking dominance relationships between point pairs from different partitions. Then, every point dominated by the pivot point is removed and *BSkyTree-P* recursively divides the partitions into sub-partitions until each partition contains at most one point. It next merges the partitions and computes the local skyline of the merged partition repeatedly until there is a single partition and then the global skyline is obtained.

Recently, skyline processing algorithms in distributed environments such as MapReduce [37, 40, 56, 57] and other distributed systems [2, 23, 25, 60] have been proposed. Among the above works, we next illustrate *MR-GPMRS* [37], *MR-BNL* [56], and *PPF-PGPS* [57] briefly since they are the most relevant works to ours. We also present 1/2-step algorithms [2] and *PPPS* [25] since they are related to ours although the works in [2] and [25] are not proposed for the MapReduce framework.

While *MR-GPMRS* [37] consists of the partitioning and global skyline phases only, *MR-BNL* [56] and *PPF-PGPS* [57] are composed of the partitioning, local skyline and global skyline phases. In the partitioning phase, the space is split into partitions by using angle-based partitioning [52] in *PPF-PGPS* or grid partitioning in *MR-GPMRS* and *MR-BNL*. In contrast to *MR-GPMRS* using two phases, *MR-BNL* and *PPF-PGPS* compute the local skyline for each partition in the additional local skyline phase. The benefit of the additional phase is that the overheads of computing the skyline as well as distributing the points via the network in the global skyline phase are reduced since the number of local skyline points in each partition is much less than that of all points in the partition. Then, in the global skyline phase, *MR-GPMRS*, *MR-BNL* and *PPF-PGPS* compute the global skyline.

In the global skyline phase, all points in the partitions should be distributed into groups so that each point p and all other points dominated by p are contained in the same group. However, since *MR-GPMRS* with two phases does not have the local skyline phase, it does not get the benefit of decreasing the network overhead of the global skyline phase from the filtering by the local skyline phase. The machines participating in the MapReduce framework could not be fully utilized by *MR-BNL* and *PPF-PGPS* since *MR-BNL* and *PPF-PGPS* use a single machine to compute the global skyline. Thus, the performances of *MR-BNL* and *PPF-PGPS* deteriorate when there are a large number of local skyline points. On the contrary, our proposed parallel skyline algorithm is optimized by utilizing the available machines as many as possible and performing additional pruning technique in the local skyline phase.

Although the works in [2] and [25] are not proposed for MapReduce, we present them here since they can be processed with MapReduce. The 1-step and 2-step algorithms in [2] split the data space into $\lceil m^{1/(d-1)} \rceil^d$ and m^d grid partitions, respectively, where m is the number of machines. They next prune the partitions with no skyline point by dominance relationships of the grid partitions and compute the global skyline for every unpruned partition in parallel.

The algorithm *PPPS* in [25] for multi-core machines utilizes the angle-based space partitioning [52]. *PPPS* recursively splits each partition into two partitions until the number of the partitions becomes the desired number of CPU cores c. The local skyline is next computed for every partition in parallel. Finally, *PPPS* performs a bottom-up merge in $O(\log(c))$ iterations until there remains a single partition only where *PPPS* merges the local skylines simultaneously in every group of two partitions in each iteration. Since *PPPS* can utilize $c/2^i$ cores only in the i-th merging iteration, multi-cores are not fully utilized for parallel merging of the local skylines.

2.2 Reverse Skyline Queries

The reverse skyline query was introduced in [16] where a branch-and-bound reverse skyline (BBRS) algorithm and an enhanced reverse skyline using skyline approximation (RSSA) algorithm are proposed.

BBRS is an extension of BBS algorithm [38] for the skyline query. Given a query point q, BBRS first computes the superset of the reverse skyline which can be simply computed than the reverse skyline. Then, it verifies that each point p in the superset of the reverse skyline is a reverse skyline point of q by invoking a window query. The window query checks whether the rectangular area centered at p contains other point in the data set or not. The point p is a reverse skyline point only and only if there exists no other point in the rectangular area. BBRS utilizes R*-tree to speed up the window queries.

RSSA is an improved version of BBRS by utilizing the dynamic skyline to prune away non-reverse skyline points. RSSA first computes the dynamic skyline of each point in the data set before computing the reverse skyline. Given a query point q, RSSA computes the superset of the reverse skyline of q and check whether each point in the superset is pruned by dominance relationships with the dynamic skyline points calculated before. If p is not pruned by the dnyamic skyline points, p is checked whether a reverse skyline point or not by the window query. The number of invoking the window queries is reduced by the pruning based on the dynamic skyline points and thus RSSA shows better performance than BBRS.

Recently, the RSQ algorithm [21] is proposed to reduce the number of traversing R*-tree by utilizing a technique. However, since BBRS, RSSA and RSQ utilize the centralized index structure R*-tree, it is hard to parallelize the algorithms by using the MapReduce framework.

Many reverse skyline variants such as the bichromatic reverse skyline queries [55], reverse skyline queries on data stream [59] and the reverse skyline queries in the wireless sensor networks [53] are introduced. In [53], a skyband-based approach to process reverse skyline queries energy-efficiently in wireless sensor networks is proposed. Note that the wireless sensor network is similar to the distributed computation framework but it has limitated network bandwidth and battery power. Thus, the wireless sensor network is not suitable to run a MapReduce job since it requires heavy network traffic and computation power. To the best of our knowledge, there is no parallel reverse skyline algorithm using MapReduce.

2.3 Probabilistic Skyline Queries

The uncertainty inherently arises in the real-world from diverse applications. Due to the importance of supporting applications dealing with uncertain data, the techniques for processing uncertain queries such as probabilistic top-K [44] and similarity

join [35] queries have been proposed. Refer to [54] for the summary of processing uncertain queries.

The serial algorithms for probabilistic skyline processing over uncertain data have been introduced in [5, 41]. The skyline probabilities of all objects in the discrete model are computed without considering the minimum probability threshold in [5]. Skyline computation with the minimum probability threshold is considered in [41] for both discrete and continuous models, but every instance of each object has the same existence probability. To parallelize such serial algorithms, we need two MapReduce phases. The first phase splits data into partitions randomly and computes the partial skyline probabilities of every object in each partition independently. The second phase computes the skyline probability of each object by collecting its partial skyline probabilities from different partitions. However, the performances of the algorithms simply extended from the serial algorithms degrade since they do not utilize the filtering techniques based on the probabilistic threshold. In this dissertation, we address a generalized problem of both [5] and [41], and we compute the probabilistic skylines with the minimum probability threshold for the discrete and continuous models.

Recently, as shown in Section 2.1, parallel skyline processing algorithms with MapReduce for certain data (i.e., non-probabilistic data) were presented. We can develop the parallel algorithms for uncertain data by simply performing one of the algorithms for certain data for every possible world. However, since there are exponential number of possible worlds (i.e., $O(2^{|\mathbb{D}|})$) where $|\mathbb{D}|$ is the number of uncertain objects in the data set), naive extensions of such algorithms to uncertain data are very inefficient and impractical.

The most relevant work to ours is the MapReduce algorithm *PSMR* [18], but *PSMR* can compute the probabilistic skylines only for the case where each uncertain object has a single instance in the discrete model. The algorithm *PSMR* in [18] works with two MapReduce phases as follows. It first computes the local candidate and affect sets in the first phase. The candidate set contains possible probabilistic skyline objects and

the affect set includes the probabilistic non-skyline objects required to compute the skyline probabilities of the objects in the candidate set. In the second phase, *PSMR* first divides the union of the candidate and affect sets into several partitions each of which is allocated to a different machine. After broadcasting the candidate set to every machine, each machine computes the partial skyline probabilities of all broadcast candidate objects by using the objects in its allocated partition. Then, they gather all partial skyline probabilities of each object from different machines into one of the machines to calculate the skyline probabilities of all candidate objects in parallel. To the best of our knowledge, the case of allowing multiple instances to each object for processing probabilistic skylines using MapReduce has not addressed yet.

Chapter 3

Background

In this chapter, we provide the technical background commonly used in this dissertation. We first present the definitions of skyline, dynamic skyline, reverse skyline and probabilistic skyline queries. We next present the overview of the MapReduce framework.

3.1 Skyline and Its Variants

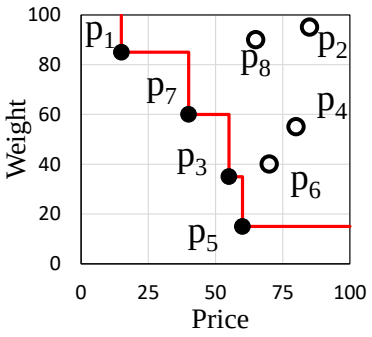
Consider a d-dimensional data set $\mathbb{D} = \{p_1, p_2, \ldots, p_{|\mathbb{D}|}\}$. A point p_i is represented by $\langle p_i(1), p_i(2), \cdots, p_i(d)\rangle$ where $p_i(k)$ is the k-th coordinate of p_i. A point p_i *dominates* another point p_j, denoted as $p_i \prec p_j$, if the two conditions hold: (1) for every k with $1 \leq k \leq d$, we have $p_i(k) \leq p_j(k)$ and (2) there exists k with $1 \leq k \leq d$ such that $p_i(k) < p_j(k)$ holds. The skyline is defined as follows.

Definition 3.1.1 (Skyline) *The* skyline *of* $\mathbb{D}$*, represented by* $SL(\mathbb{D})$*, is a subset of* $\mathbb{D}$ *where every point in* $SL(\mathbb{D})$ *is not dominated by every other point in* $\mathbb{D}$*. In other words,* $SL(\mathbb{D}) = \{p_i \in \mathbb{D} \mid \nexists p_j (\neq p_i) \in \mathbb{D}$ *such that* $p_j \prec p_i\}$*. The points in* $SL(\mathbb{D})$ *are called* skyline points *of* $\mathbb{D}$*.*

Example 3.1.2 *Consider a data set* $\mathbb{D}$ *representing laptops with two attributes of*

ID	Price	Weight
p_1	15	85
p_2	85	95
p_3	55	35
p_4	80	55
p_5	60	15
p_6	70	40
p_7	40	60
p_8	65	90

(a) The laptop data set $\mathbb{D}$



(b) The skyline of $\mathbb{D}$

Figure 3.1: An example of a skyline

price and weight in Figure 3.1(a). In Figure 3.1(b), we plot every point in $\mathbb{D}$ *into a 2-dimensional space where the horizontal axis indicates the price and the vertical axis represents the weight. The point* $p_3 = \langle 55, 35 \rangle$ *dominates* $p_6 = \langle 70, 40 \rangle$ *since we have* $p_3(1) = 55 < p_6(1) = 70$ *and* $p_3(2) = 35 < p_6(2) = 40$*. Since* p_1 *is not dominated by the other points in* $\mathbb{D}$*,* p_1 *is a skyline point (i.e.,* $\in SL(\mathbb{D})$*). The skyline of* $\mathbb{D}$ *is* $SL(\mathbb{D}) = \{p_1, p_3, p_5, p_7\}$*. We plot every point in* $SL(\mathbb{D})$ *with a black circle in Figure 3.1(b).* ∎

Given a query point q, we say that a point p_i *dynamically dominates* another point p_j with respect to q, denoted as $p_i \prec_q p_j$, if and only if (1) $|p_i(k) - q(k)| \leq |p_j(k) - q(k)|$ for all k with $1 \leq k \leq d$ and (2) there exists k with $1 \leq k \leq d$ such that $|p_i(k) - q(k)| < |p_j(k) - q(k)|$. The dynamic skyline is defined based on the dynamical dominance relationships.

Definition 3.1.3 (Dynamic skyline) *Given a query point* q *and a data set* $\mathbb{D}$*, the* dynamic skyline*, represented by* $DSL(q, \mathbb{D})$*, is the set of points that are not dynamically dominated by any other point with respect to* q*. In other words,* $DSL(q, \mathbb{D}) = \{p_i \in \mathbb{D} \mid \nexists p_j (\neq p_i) \in \mathbb{D}$ *such that* $p_j \prec_q p_i\}$.

In the above definition, dynamic skyline $DSL(q, \mathbb{D})$ is equivalent to the traditional

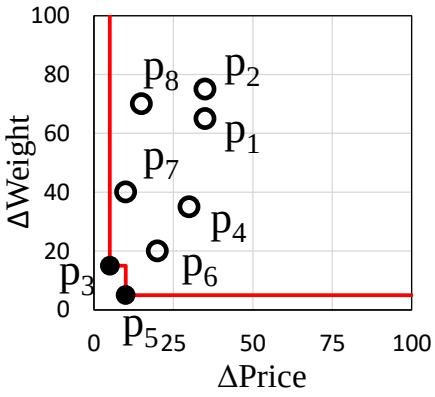


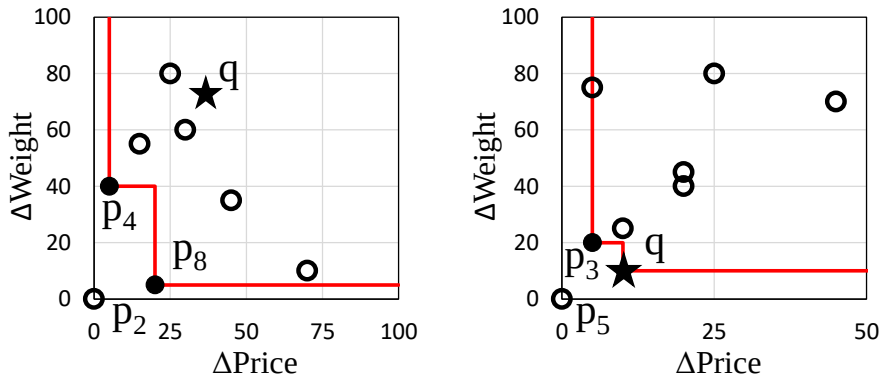
Figure 3.2: The dynamic skyline of $\mathbb{D}$ with respect to $q = \langle 50, 20 \rangle$

skyline obtained by transforming every point p_i in $\mathbb{D}$ into a new space where the query point q becomes the origin and for every k with $1 \leq k \leq d$ (i.e., $|p_i(k) - q(k)|$ becomes the k-dimensional coordinate value in the new space).

Example 3.1.4 *Let us consider the data set $\mathbb{D}$ in Figure 3.1(a) and a query point $q = \langle 50, 20 \rangle$. The point $p_3 = \langle 55, 35 \rangle$ dynamically dominates the point $p_7 = \langle 40, 60 \rangle$ with respect to q since we have $|p_3(1) - q(1)| = 5 < |p_7(1) - q(1)| = 10$ and $|p_3(2) - q(2)| = 15 < |p_7(2) - q(2)| = 30$. In Figure 3.2, each point $p_i = \langle p_i(1), p_i(2) \rangle$ in $\mathbb{D}$ is converted to $p'_i = \langle |p_i(1) - q(1)|, |p_i(2) - q(2)| \rangle$. For instance, $p_1 = \langle 15, 85 \rangle$ is mapped to $p'_1 = \langle |15-50|, |85-20| \rangle = \langle 35, 65 \rangle$. Since the dynamic skyline $DSL(q, \mathbb{D})$ is the same as the skyline of the points mapped to the new space whose origin is the query point q, $DSL(q, \mathbb{D})$ becomes $\{p_3, p_5\}$. We represent the dynamic skyline point p_3 by a black circle in Figure 3.2.* ∎

Based on the definition of the dynamic skyline, the notion of the *reverse skyline* is proposed.

Definition 3.1.5 (Reverse skyline) *Given a d-dimensional data set $\mathbb{D}$ and a query point q, the* reverse skyline*, represented by $RSL(q, \mathbb{D})$, is the set of every point p_i*



(a) The dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_2\}$ with respect to p_2

(b) The dynamic skyline of $\mathbb{D} \cup \{q\} - \{p_5\}$ with respect to p_5

Figure 3.3: The dynamic skylines with respect to p_2 and p_5

in $\mathbb{D}$ *satisfying* $q \in DSL(p_i, \mathbb{D} \cup \{q\} - \{p_i\})$*(i.e., the query point* q *is contained in the dynamic skyline with respect to* p_i*).*

Example 3.1.6 *Consider the data* $\mathbb{D}$ *in Figure 3.1(a). Figure 3.3(a) illustrates the dynamic skyline with respect to* p_2*. Since* q *does not belong to the dynamic skyline with respect to* p_2*,* p_2 *is not a reverse skyline point (i.e.,* $p_2 \notin RSL(q, \mathbb{D})$*). However, since* q *is in the dynamic skyline with respect to* p_5 *illustrated in Figure 3.3(b),* p_5 *is in the reverse skyline with respect to* q*.* ∎

We next introduce the definition of the probabilistic skyline [41] by the popular *possible worlds* semantics [3, 14]. Each object in uncertain data can be modeled by a set of instances with their existence probabilities (i.e., discrete model) or an uncertainty region with its pdf (i.e., continuous model).

The discrete model: Given a set of uncertain objects $\mathbb{D}$, an object $U \in \mathbb{D}$ is modeled as a set of instances and denoted by $U = \{u_1, u_2, \cdots, u_{|U|}\}$ where u_i is associated with an existence probability $P(u_i)$ such that $\sum_{u_i \in U} P(u_i) \leq 1$. A *possible world* is a materialized set of instances from objects. Since all instances of U are mu-

tually exclusive, multiple instances of U cannot belong to a possible world simultaneously. The probability that an instance $u_i \in U$ appears in a possible world is $P(u_i)$ and the probability that any instance of an object U does not appear is $1 - \sum_{u_i \in U} P(u_i)$.

When a possible world contains an instance $u_i \in U$, if any instance v_j of every other object $V \in \mathbb{D}$ dominating u_i does not exist in the possible world, u_i is a skyline instance in the possible world. Since such a probability is $\prod_{V \in \mathbb{D}, V \neq U} (1 - \sum_{v_j \in V, v_j \prec u_i} P(v_j))$, the skyline probability of u_i, denoted by $P_{sky}(u_i)$, can be written as follows [5]:

$$P_{sky}(u_i) = P(u_i) \times \prod_{V \in \mathbb{D}, V \neq U} (1 - \sum_{v_j \in V, v_j \prec u_i} P(v_j)). \tag{3.1}$$

We define the skyline probability of an object U, denoted by $P_{sky}(U)$, as the sum of the skyline probabilities of all its instances (i.e., $P_{sky}(U) = \sum_{u_i \in U} P_{sky}(u_i)$).

The continuous model: An uncertain object $U \in \mathbb{D}$ is modeled as an *uncertainty region* $U.R$ with its probabilistic distribution function $U.f(\cdot)$ [9, 32, 41]. We assume that each uncertainty region is a hyper-rectangle as in [32]. The probability that an instance of U is located at a point u in $U.R$ is $U.f(u)$ where $\int_{U.R} U.f(u)\mathrm{d}u = 1$.

Given an object $U \in \mathbb{D}$, $P_{sky}(U)$ is defined in [41] as:

$$P_{sky}(U) = \int_{U.R} U.f(u) \prod_{V \in \mathbb{D}, V \neq U} \left(1 - \int_{V.R} V.f(v)\mathbb{I}(v \prec u)\mathrm{d}v\right)\mathrm{d}u \tag{3.2}$$

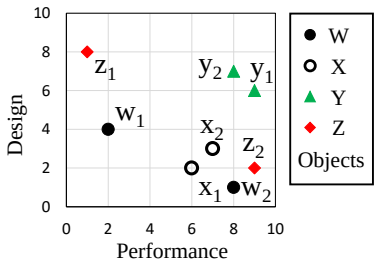
where $\mathbb{I}(v \prec u)$ is an indicator function which returns 1 if v dominates u, and 0 otherwise.

Definition 3.1.7 (Probabilistic skyline) *For a set of uncertain objects $\mathbb{D}$ and a probability threshold T_p, the probabilistic skyline, denoted by $pSL(\mathbb{D}, T_p)$, is the set of all objects whose skyline probabilities are at least T_p. That is, $pSL(\mathbb{D}, T_p) = \{U \in \mathbb{D} \mid P_{sky}(U) \geq T_p\}$.*

Example 3.1.8 *Consider a set of objects $\mathbb{D} = \{W, X, Y, Z\}$ with the discrete model in Figure 3.4(a). In Figure 3.4(b), we plot every instance in $\mathbb{D}$ into a 2-dimensional*

Object	Instances	Performance	Design	Probability
W	w_1	2	4	0.5
	w_2	8	1	0.4
X	x_1	6	2	0.2
	x_2	7	3	0.2
Y	y_1	9	6	0.8
	y_2	8	7	0.2
Z	z_1	1	8	0.5
	z_2	9	2	0.5

(a) The rating of laptops



(b) Uncertain objects

Figure 3.4: An example of a probabilistic skyline

space. Since y_1 *is dominated by* w_1, w_2, x_1, x_2 *and* z_2, *the skyline probability of* y_1 *computed by Equation* (3.1) *is* $P_{sky}(y_1) = P(y_1)(1 - P(w_1) - P(w_2))(1 - P(x_1) - P(x_2))(1 - P(z_2)) = 0.024$. *Similarly,* $P_{sky}(y_2) = 0.012$. *The skyline probability of* Y *is* $P_{sky}(Y) = P_{sky}(y_1) + P_{sky}(y_2) = 0.036$. *Furthermore, we have* $P_{sky}(W) = 0.9$, $P_{sky}(X) = 0.4$ *and* $P_{sky}(Z) = 0.74$. *When* T_p *is* 0.5, $pSL(\mathbb{D}, T_p)$ *is* $\{W, Z\}$. ∎

3.2 MapReduce Framework

MapReduce [15] or its open-source equivalent Hadoop [4] is a widely used framework for data-intensive parallel computation in shared-nothing clusters of machines. In Hadoop, data is represented as key-value pairs. Hadoop divides the input data to a MapReduce job into fixed-size pieces called *chunks* and spawns a *mapper task* for each chunk. The mapper task invokes a *map* function for each key-value pair in the chunk and the map function may output several key-value pairs. The key-value pairs emitted by all map functions are grouped by keys in the *shuffling phase* and passed to *reducer tasks* to generate the final output. Users can control which key goes to which reducer task by modifying a *Partitioner* class. For each distinct key, the reduce task invokes a *reduce* function with the key and the list of all values sharing the key as input. A *reduce* may generate several key-value pairs. Each mapper (or reducer) task can execute a *setup* function before invoking map (or reduce) functions and a *cleanup*

function after executing all map (or reduce) functions. Hadoop executes the main function on a single master machine and we may pre-process the input data or post-process the output in the main function.

Chapter 4

Parallel Skyline Query Processing

4.1 SKY-MR: Our Skyline Computation Algorithm

To calculate the skyline, a brute-force algorithm compares each point with every other points. If we know a non-skyline point earlier, we do not have to compare the point with every other points. Knowing data distributions allows us to prune such non-skyline points earlier. To identify the data distribution, many histograms such as MHIST [42] and V-Opt [24] have been proposed. However, these histograms partition the data space into buckets with arbitrary regions and thus it is unsuitable and difficult to identify the dominance relationship between buckets effectively. Thus, we utilize a variant of the quadtree [20], called the *sky-quadtree* since we can easily identify the dominance relationship between the regions of the quadtree as shown later in this chapter.

We present our parallel algorithm *SKY-MR* to discover the skyline $SL(\mathbb{D})$ in a given data set $\mathbb{D}$ by utilizing the *sky-quadtree*. The pseudocode of *SKY-MR* is shown in Figure 4.1. *SKY-MR* consists of the following three phases.

(1) **Sky-quadtree building phase:** To filter out non-skyline points effectively earlier, we propose a new histogram, called the *sky-quadtree*, to represent data distributions. To speed up, we build a *sky-quadtree* with a sample of D where each leaf node with non-skyline sample points only is marked as "pruned".

Function SKY-MR($\mathbb{D}$, ρ, d, δ)

$\mathbb{D}$: a data set, ρ: the split threshold,

d: the dimension, δ : local skyline threshold

begin

1. $sample$ = ReservoirSampling($\mathbb{D}$);
2. $sky\text{-}quadtree$ = SKY-QTREE($sample$, ρ, d);
3. Broadcast $sky\text{-}quadtree$;
4. $Local\text{-}SL$ = RunMapReduce(L-SKY-MR);
5. **if** $Local\text{-}SL$.size $\geq \delta$ **then**
6. Broadcast *non-empty leaf node ids*;
7. SL = RunMapReduce(G-SKY-MR);
8. **else** SL = G-SKY($Local\text{-}SL$);
9. **return** SL;

end

Figure 4.1: The SKY-MR algorithm

(2) **Local skyline phase:** We partition the data D based on the regions divided by the *sky-quadtree* and compute the local skyline for the region of every unpruned leaf node independently using MapReduce by calling *L-SKY-MR*.

(3) **Global skyline phase:** We calculate the global skyline using MapReduce from the local skyline points in every unpruned leaf node by calling *G-SKY-MR*. When the number of local skyline points is small, we run the serial algorithm *G-SKY* in a single machine to speed up.

We next present the details of the above three phases.

4.1.1 SKY-QTREE: The Sky-Quadtree Building Algorithm

A *sky-quadtree* is an extension of quadtrees [20] which subdivide the d-dimensional space recursively into sub-regions. In a *sky-quadtree*, internal nodes have exactly 2^d children and each leaf node has at most a predefined number of points ρ called the

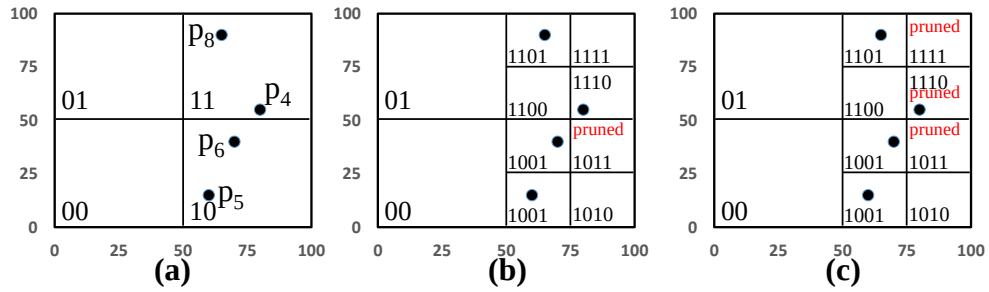


Figure 4.2: An example of sky-quadtree building

split threshold. We denote the region of a node n as $region(n)$. An id is assigned to each node based on its location in *sky-quadtrees*. In a d-dimensional space, the id of a node n with depth k is represented by $id(n) = a_1 a_2 \cdots a_{k \cdot d}$ which consists of the first $(k-1) \cdot d$ bits coming from its parent node and the remaining d bits $a_{(k-1)\cdot d+1} a_{(k-1)\cdot d+2} \cdots a_{k \cdot d}$ where $a_{(k-1)\cdot d+i} = 0$ (or $a_{(k-1)\cdot d+i} = 1$) if the i-th dimensional range of the $region(n)$ is the first half (or the second half) of its parent's i-th dimensional range. Similarly, we let $node(id)$ represent the node with an id id. We can decompose $id(n)$ into d number of bit strings $sub(id(n), i)$s (for $1 \leq i \leq d$) s.t. $sub(id(n), i) = a_i a_{(i+d)} a_{(i+2\cdot d)} \cdots a_{(i+(k-1)\cdot d)}$.

Given a pair of bit strings $a = a_1 a_2 \cdots a_p$ and $b = b_1 b_2 \cdots b_q$, we say that $a = b$ if $a_i = b_i$ for all i = 1, 2, $\cdots$, $min(p, q)$, and $a < b$ if there exists an integer j, with $1 \leq j \leq min(p, q)$, s.t. $a_i = b_i$ for all $i = 1, 2, \cdots, j-1$ and $a_j < b_j$. Similarly, we write $a > b$ if there exists an integer j, with $1 \leq j \leq min(p, q)$, s.t. $a_i = b_i$ for $i = 1, 2, \cdots, j-1$ and $a_j > b_j$.

Definition 4.1.1 *Given a pair of leaf nodes n_i and n_j in a sky-quadtree, if every point in $region(n_i)$ dominates all points in $region(n_j)$, n_i* dominates *n_j and we represent it by $n_i \prec n_j$. If every point in $region(n_i)$ does not dominate all points in $region(n_j)$, n_i* does not dominate *n_j and we denote it by $n_i \not\prec n_j$.*

Dominance relationships by node ids: Based on the following proposition, we

can efficiently identify the nodes dominated by another non-empty leaf node in a *sky-quadtree* by utilizing node ids.

Proposition 4.1.2 *Given a pair of nodes n_i and n_j in a sky-quadtree, n_i dominates n_j if $sub(id(n_i), k) < sub(id(n_j), k)$ for all $k = 1, 2, \cdots, d$. Similarly, n_i does not dominate n_j (i.e., $n_i \not\prec n_j$) if there exists k such that $sub(id(n_i), k) > sub(id(n_j), k)$ with $k = 1, 2, \cdots, d$.*

Proof: Let $region(n) = \langle [n(1)^-, n(1)^+), \cdots, [n(d)^-, n(d)^+) \rangle$ where $[n(k)^-, n(k)^+)$ is the range of the k-th dimension of n's covering region. If $sub(id(n_i), k) < sub(id(n_j), k)$, both $[n_i(k)^-, n_i(k)^+)$ and $[n_j(k)^-, n_j(k)^+)$ are disjoint and $n_i(k)^+ \leq n_j(k)^-$. Thus, for a pair of points p_i in $region(n_i)$ and p_j in $region(n_j)$, we have $p_i(k) < n_i(k)^+ \leq n_j(k)^- \leq p_j(k)$. If $sub(id(n_i), k) < sub(id(n_j), k)$ holds for all k=1, 2, $\cdots$, d, we have $p_i \prec p_j$ and $n_i \prec n_j$. Similarly, if there exists k such that $sub(id(n_i), k) > sub(id(n_j), k)$, we have $p_i(k) > p_j(k)$ and $n_i \not\prec n_j$ ∎

Building a sky-quadtree: In order to quickly build a *sky-quadtree*, we utilize a random sample obtained from D by reservoir sampling [51]. Since we use a sample only, we may prune fewer non-skyline points than using D. However, the use of sampling does not affect the correctness of our skyline computation algorithm *SKY-MR* because all skyline points exist in unpruned leaf nodes.

The procedure *SKY-QTREE* (in line 2 of Figure 4.1) builds a *sky-quadtree* by inserting a sample into the root node and recursively splits each node n to 2^d child nodes whenever the number of points in n exceeds the split threshold ρ. When splitting a node n, we insert each point p in $region(n)$ into its child node n_i into which p is inserted. If the last d-bit string of n_i's id is $00 \cdots 0$ (i.e., the first half in every dimension), we mark n_j whose last d-bit string of its id is $11 \cdots 1$ (i.e., the second half in every dimension) as "pruned" and skip all remaining points belonging to n_j. After all points are inserted into child nodes, we recursively split each unpruned child node. When we cannot split any more, starting from the root node, we traverse the *sky-quadtree* to mark every node dominated by a non-empty leaf node as "pruned".

Example 4.1.3 *Consider the data D in Figure 3.1(a) and the split threshold ρ=1. Suppose that a sample $\{p_4, p_6, p_7, p_8\}$ is inserted into the root node. In Figure 4.2(a), the root node is subdivided since it has more than ρ points. The id of the root node's child node in the top-left corner is 01 since the region covers the first and second halves of the root node's first and second dimensions respectively. The node id 1011 can be decomposed into $sub(\mathbf{10}\mathbf{1}1, 1) = 11$ and $sub(1\mathbf{01}\mathbf{1}, 2) = 01$. Additionally, for the node id 1000, $sub(\mathbf{10}\mathbf{0}0, 1) = 10$ and $sub(1\mathbf{00}\mathbf{0}, 2) = 00$. Because $sub(\mathbf{10}\mathbf{0}0, 1) < sub(\mathbf{10}\mathbf{1}1, 1)$ and $sub(1\mathbf{00}\mathbf{0}, 2) < sub(1\mathbf{01}\mathbf{1}, 2)$, we have $node(1000) \prec node(1011)$. Thus, we mark $node(1011)$ as "pruned" in Figure 4.2(b). In addition, since $node(1000) \prec node(1110)$ and $node(1000) \prec node(1111)$ hold, both nodes are marked as "pruned". The final sky-quadtree obtained is presented in Figure 4.2(c).* ∎

Finding a proper sample size: Since we construct a *sky-quadtree* from a sample $S \subset D$, the number of data points of D located in the region of each leaf node varies a lot. Because we split a node of a *sky-quadtree* if it contains more than the split threshold ρ sample points, the number of sample points in each leaf node is upper-bounded by ρ. However, there is no upper-bound for the number of actual data points of D located in each leaf node. It can be problematic if a node contains much more points than estimated number since it can cause skewness of the workloads of machines in the local and global skyline phases. Thus, we next study how to choose the sample size $|S|$ to find the *probabilistic* upper-bound of the number of data points in each leaf node.

For a leaf node n of a quadtree, we denote the number of sample and data points in the region of n by $S(n)$ and $D(n)$. Since we utilize the random sampling, we can estimate $|D(n)|$ as $|\hat{D}(n)| = |S(n)| \times |D|/|S|$. We want to guarantee that the probability of the actual partition size $|\hat{D}(n)|$ being much greater than the estimated size (i.e., $\gamma \times |\hat{D}(n)|$ for $\gamma > 1$) is less than a threshold δ. When $|D(n)|$ is small, it does not affect the workloads of the our algorithm *SKY-MR* even though $|D(n)|$ is much larger than $|\hat{D}(n)|$. Thus, we focus on the case when $|D(n)| \geq \omega$ for a user-defined param-

eter ω. In other word, given a user-defined parameters γ and δ, we want to estimate the number of sample size such that $Pr[\gamma \times |S(n)| \times |D|/|S| < |D(n)|] < \delta$ when $D(n) \geq \omega$. We have the following lemma which can be used to find a proper sample size.

Lemma 4.1.4 *Given a leaf node n, a maximum probability threshold δ, a partition size threshold ω, a endurable size ratio $\gamma > 1$ and a sample $S \subset D$, if $|S| \geq -2 \ln \delta \frac{D \cdot \gamma^2}{\omega(1-\gamma)^2}$, we have $Pr[\gamma \times |S(n)| \times |D|/|S| < |D(n)|] < \delta$ for $|D(n)| \geq \omega$.*

Proof: Let X_j be a random variable that is 1 if j-th point in S belongs to the region of n and 0 otherwise. Since we do uniform random sampling, $X_1, \cdots, X_{|S|}$ are independent Bernoulli trials with $P(X_j = 1) = |D(n)|/|D|$. The number of points in S belonging to $S(n)$ is $X = \sum_j X_j$ and the expected value of X is $\mu = E[X] = |S| \cdot |D(n)|/|D|$. Then, we have $Pr[\gamma \cdot |\hat{D}(n)| < |D(n)|] = P[X \times |D|/|S| < |D(n)|/\gamma]$ since we have $|\hat{D}(n)| = |S(n)| \times |D|/|S| = X \times |D|/|S|$.

Chernoff bounds state that we have $P[X < (1-\epsilon)\mu] < \exp(-\mu\epsilon^2/2)$ for $0 < \epsilon \leq 1$. Rewriting the probability to conform to the Chernoff bounds, we get $P[X < (1-(1-\frac{|S| \cdot |D(n)|}{\gamma \cdot \mu \cdot |D|}))\mu] = Pr[X < (1-(1-\frac{1}{\gamma}))\mu]$ since $\mu = |S| \cdot |D(n)|/|D|$ holds. Then, by applying the Chernoff bounds, we obtain $Pr[X < (1-(1-1/\gamma))\mu] < \exp(-\mu(1-1/\gamma)^2/2)$. Thus, $Pr[X < (1-(1-1/\gamma))\mu]$ is upper bounded by $\exp(-|S| \cdot |D(n)|/|D| \cdot (1-1/\gamma)^2/2)$. In other words, the maximum probability threshold δ is at most $\exp(-|S| \cdot |D(n)|/|D| \cdot (1-1/\gamma)^2/2)$. Solving it for $|S|$, we get $|S| \geq -2 \ln \delta \frac{D \cdot \gamma^2}{D(n)(1-\gamma)^2}$. Since we assume that $|D(n)| \geq \omega$, $|S|$ should be larger than $-2 \ln \delta \frac{D \cdot \gamma^2}{\omega(1-\gamma)^2}$. ∎

4.1.2 L-SKY-MR: The Local Skyline Computation Algorithm

We next present the parallel algorithm *L-SKY-MR* that calculates the local skyline independently for every unpruned leaf node in the *sky-quadtree*. The *sky-quadtree* Q is first broadcast to all map functions. Each map function is next called with a point p

in D. If the point p is in the region of an unpruned leaf node n_p of Q, we output the key-value pair $\langle n_p, p\rangle$. Otherwise, we do nothing.

In the shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct leaf node, and a reduce function is called with each node n and its point list L. Each reduce function computes the local skyline in L (i.e., $SL(L)$) and outputs $\langle n,p\rangle$ for every local skyline point p. It also produces an artificial d-dimensional point referred to as the *virtual max point* of the node n which is denoted by vp_n where $vp_n(k) = max_{p \in SL(L)} p(k)$ with $1 \leq k \leq d$. Every virtual max point of each unpruned leaf node is output to the file VIRTUAL in the Hadoop distributed file system(HDFS). The virtual max point will be used to reduce the number of checking dominance relationships by the following proposition.

Proposition 4.1.5 *If a point p does not dominate the virtual max point of a leaf node n (i.e., vp_n) in a sky-quadtree, p does not dominate every local skyline point in $region(n)$.*

Proof: We will prove the contrapositive: if p dominates a local skyline point in $region(n)$, we have $p \prec vp_n$. Since the point p dominates a local skyline point p_l in $region(n)$, we have $p(k) \leq p_l(k)$ for every k with $1 \leq k \leq d$ and there exists k such that $p(k) < p_l(k)$. By the definition of the virtual max point, $p_l(k) \leq vp_n(k)$ holds for every k. Thus, we also have $p(k) \leq vp_n(k)$ for every k and there exists k such that $p(k) < vp_n(k)$. In other words, $p \prec vp_n$. ∎

Example 4.1.6 *Consider the points p_i=$\langle 5, 30\rangle$, p_j=$\langle 10,\ 20\rangle$ and p_k=$\langle 15, 10\rangle$. Assume $\{p_j,\ p_k\}$ is the local skyline of an unpruned leaf node n. The virtual max point vp_n is $\langle 15, 20\rangle$. Since $p_i \not\prec vp_n$, p_i does not dominate every local skyline point in n due to Proposition 4.1.5 and we do not need to check whether the points p_j and p_k are dominated by p.* ∎

In addition, each reduce function selects a single local skyline point, called a *sky-filter point*, for each dimension which has the minimum value on the dimension. The

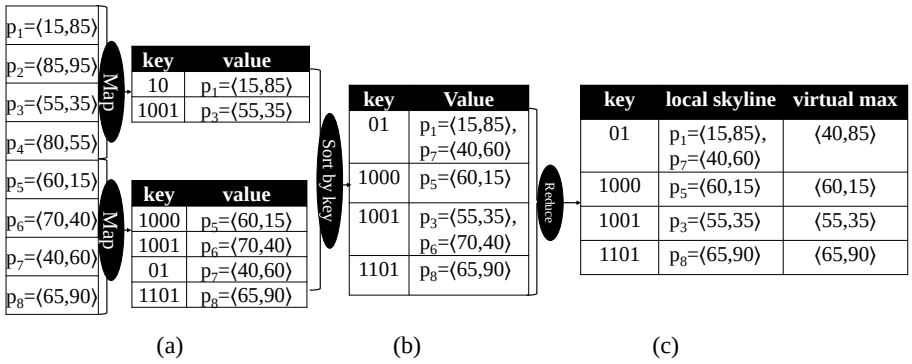


Figure 4.3: The data flow in the local skyline phase of SKY-MR

local skyline points dominated by such selected sky-filter points will be filtered out in the next global skyline phase. All sky-filter points are stored to the file called SKY-FILTER in HDFS.

Example 4.1.7 *Consider the sky-quadtree in Example 4.1.3. Figures 4.3(a)–(c) show the data flow in the local skyline phase of* SKY-MR. *After the sky-quadtree is broadcast to all map functions, each map function is invoked with a point p in D as illustrated in Figure 4.3(a). For instance, $\langle 10,\ p_1\rangle$ is emitted since p_1 is contained in the unpruned leaf node, $node(10)$. In Figure 4.3(a), the key-value pairs emitted from all map functions are shown. The key-value pairs grouped by each distinct key are provided in Figure 4.3(b). Each reduce function finally outputs the local skyline of a node and the virtual max point as well as sky-filter points. Consider $node(10)$ whose skyline points are $\{\langle 15, 85\rangle, \langle 40, 60\rangle\}$. The reduce function with $node(10)$ outputs $\langle 15, 85\rangle$ and $\langle 40, 60\rangle$ as sky-filter points. It also outputs $\langle 40, 85\rangle$ as a virtual max point. The points output by all reduce functions are illustrated in Figure 4.3(c).* ∎

Discussion: We can utilize R*-trees instead of our *sky-quadtrees*. However, since R*-trees are optimized to reduce the amount of "dead space" (empty area) covered by their nodes, a large portion of uncovered space tends to be generated in R*-trees. Furthermore, generating an R*-tree from a sample increases uncovered space even

Function G-SKY-MR.map(n_i, p)

n_i: a node id, p: a point belongs to the node with id = key

begin

1. $nodes$ = LoadNonEmptyNodes();
2. **if** DominatedByFilterPoints(p) **then return**;
3. output(n_i, $(+, p)$);
4. **for** each node id n_j ($\neq n_i$) in $nodes$ **do**
5. **if** IsNeeded(n_i, n_j) **then**
6. **if** $p \prec vp_{n_j}$ **then** output(n_j, $(*, p)$);

end

Figure 4.4: The map function of the G-SKY-MR algorithm

more. Since every point belonging to the uncovered space in an R*-tree cannot be pruned, using an R*-tree instead of a *sky-quadtree* produces a lot of unpruned points resulting in a significant increase of execution times in the next phase. In addition, it is difficult to compute local skyline and global skyline in each node of an R*-tree independently because the regions represented by nodes in an R*-tree are overlapped with each other.

4.1.3 G-SKY-MR: The Global Skyline Computation Algorithm

The procedure *G-SKY-MR* computes the global skyline in every non-empty unpruned leaf node independently using MapReduce. In the map function called with each local point, the point is emitted to every other unpruned leaf node in which it may dominate at least a point in the node. Since it is straightforward to implement the serial algorithm *G-SKY*, we omit the details of *G-SKY* here.

The pseudocode of *G-SKY-MR.map* is shown in Figure 4.4. In *G-SKY-MR*, a map function with each local skyline point p discards the point p if p is dominated by a sky-filter point chosen in the previous phase. Otherwise, the pair $\langle n_i, (+, p)\rangle$ is emit-

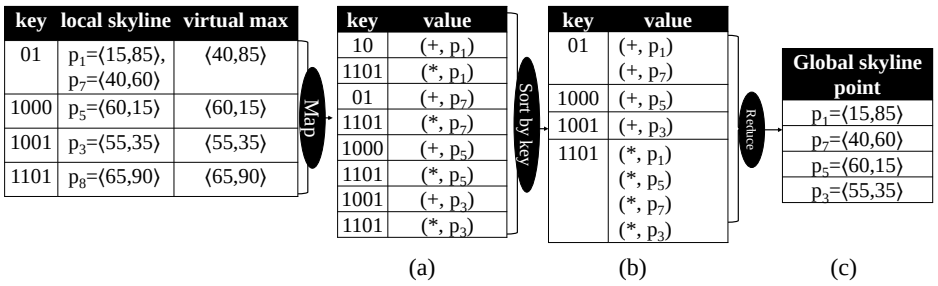


Figure 4.5: The data flow in the global skyline phase of SKY-MR

ted where n_i is the leaf node containing p and the symbol '+' represents that p is in $region(n_i)$ (in lines 1-3 of *G-SKY-MR.map*).

If a local skyline point p_j of node n_j is dominated by at least a local skyline point of the other nodes, p_j cannot be a global skyline point. However, if every point p is sent to all other nodes except n_i, the communication overhead is very expensive. By Definition 4.1.1, when $n_i \not\prec n_j$, every point in n_i cannot dominate the points in n_j. The procedure *IsNeeded*(n_i, n_j) easily identifies such case (i.e., $n_i \not\prec n_j$) using ids of two nodes based on Proposition 4.1.2. If $n_i \not\prec n_j$, *IsNeeded*(n_i, n_j) returns false. Otherwise, *IsNeeded*(n_i, n_j) returns true and we output the pair $\langle n_j, (*, p)\rangle$ where '*' indicates that the point p is not in $region(n_j)$ but p may dominate at least a point in $region(n_j)$. However, if p does not dominate vp_{n_j}, we do not emit the pair $\langle n_j, (*, p)\rangle$ due to Proposition 4.1.5 (in lines 4-6).

Each reduce function called with a node n_i next computes the global skyline points by checking whether each of n_i's local skyline points annotated with '+' is dominated by a local skyline point associated with '*' which comes from the other nodes.

Example 4.1.8 *The behavior of* G-SKY-MR *is illustrated in Figures 4.5(a)–(c). Every map function is called with each local skyline point. For example, the map function with* p_1 *emits* $\langle$*01,*(+,p_1)$\rangle$ *since the point* p_1 *is in* $region(node(01))$. *In addition,* $\langle$*1101,*(*,p_1)$\rangle$ *is emitted since* p_1 *dominates the virtual max point of* $node(1101)$. *However, in the map function invoked with* p_5, $\langle$*1101,*(*,p_5)$\rangle$ *is not emitted because* p_5 *does*

Parameter	Range	Default
Number of samples (s)	$100 \sim 8,000$	400
Split threshold (ρ)	$10 \sim 60$	20
Number of points (n)	$10^7 \sim 4 \times 10^9$	10^8
Number of dimensions (d)	$2 \sim 10$	6
Number of machines (t)	$5 \sim 20$	10

Table 4.1: Parameters used for the skyline algorithms

not dominate the virtual max point $\langle 65,90 \rangle$ in $node(1101)$. Figure 4.5(a) shows the key-value pairs emitted by all map functions. The key-value pairs after the shuffling phase are shown in Figure 4.5(b). Each reduce function computes the global skyline of its associated node. After all reduce functions are finished, we obtain the skyline in Figure 4.5(c). ∎

Extending to dynamic skylines: We first convert each point p_i in D to a point p'_i using a query point q where $p'_i(k)=|p_i(k) - q(k)|$ for $k=1, \cdots, d$, as presented in Section 3.1. Then, we calculate the dynamic skyline with respect to q by computing skyline points among the converted points. Extending *SKY-MR* to handle the dynamic skylines is straightforward since at the first and second phases, each point in D can be easily transformed into a new space whose origin is the query point q. Due to lack of space, we do not present the details of dynamic skyline processing using MapReduce.

4.2 Experiment

We empirically evaluated the performance of our proposed algorithms using the parameters as summarized in Table 4.1. All experiments on MapReduce were performed on the cluster of 20 nodes of Intel(R) Core(TM) i3 CPU 3.3GHz machines with 4GB of main memory running Linux. The implementations of all algorithms were complied by Javac 1.6. We used Hadoop 1.0.3 for MapReduce [4]. The execution times in the graphs shown in this section are plotted in log scale. We ran all algorithms five times

Algorithm	Description
SKY-MR-S/M	*SKY-MR-S* utilizes the serial algorithm *G-SKY*. *SKY-MR-M* utilizes *G-SKY-MR*.
SKY-MR	*SKY-MR* adaptively selects *G-SKY-MR* or *G-SKY* with respect to the number of local skyline points. If it is less than 7×10^5, *G-SKY* is selected.
MR-BNL	The state-of-the-art using MapReduce in [56].
PPPS-MR	The MapReduce implementation of PPPS in [25]. We set the sample size (s) to 1,000 which shows the best performance.
GRID-MR-1/2	The MapReduce implementations of the 1-step and 2-step algorithms in [2].
SKY-SC	The serial implementation of *SKY-MR*.
BBS	The state-of-the-art for a single core in [38].
SKY-MC	The implementation of *SKY-MR* for multi-cores.
PPPS	The state-of-the-art for multi-cores in [25].
SKY-MP	The implementation of *SKY-MR* using MPI.
GRID-1/2	The implementations of the 1-step and 2-step algorithms using MPI in [2].
SKY-SPARK	The implementation of *SKY-MR* using Spark.

Table 4.2: Implemented skyline algorithms

and measured the average execution times. We do not plot the execution times of some algorithms when they did not finish within 8 hours or they did not work due to some reasons such as out of memory.

Implemented algorithms: The MapReduce algorithms implemented for skyline are presented in Table 4.2. Furthermore, we also implemented the variants of *SKY-MR* for other environments such as using a single-core machine, multi-core machines, message passing interface (MPI) library [27] and Spark [46] to see the effectiveness of our proposed algorithms compared to the existing algorithms[2, 25, 38] in such

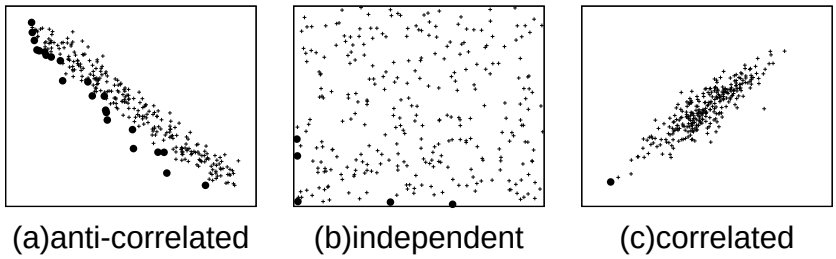


Figure 4.6: Examples of data sets

environments.

Data sets: We built three synthetic data sets which were randomly generated by *correlated*, *independent* and *anti-correlated* distributions. The three types of data sets are typically used to evaluate the performance of skyline algorithms [10]. Figure 4.6 shows the examples of such data sets where skyline points are represented by small bold circles. The sizes of resulting synthetic data sets are varied from 392MB to 153GB depending on the number of points (n) as well as the number of dimensions (d).

4.2.1 Performance Results for Skylines

Default values of s and ρ: To find the proper values of s and ρ, we ran *SKY-MR* with varying s from 100 to $8,000$ and ρ from 10 to 60. According to the Lemma 4.1.4, s should be larger than $-2 \ln \delta \frac{D \cdot \gamma^2}{\omega(1-\gamma)^2}$. By letting $\delta = 0.99$, $\omega = 10^5$ and $\gamma = 2$, we have $s \geq 80.4$. Thus, we have $Pr[2 \times |S(n)| \times |D|/|S| < |D(n)|] < 0.99$ for leaf nodes with $|D(n)| \geq 10^5$.

The average execution times of *SKY-MR* for all data sets are shown in Figure 4.7. Since the best performance of *SKY-MR* is obtained with $s = 400$ and $\rho = 20$, we set $s = 400$ and $\rho = 20$ as the default values. When the sample size s decreases, since the samples do not reflect the data distribution precisely, the number of pruned points decreases and *SKY-MR* becomes inefficient. In *SKY-MR*, virtual max points, sky-filter points and local skyline points of an unpruned node are sent to the other unpruned

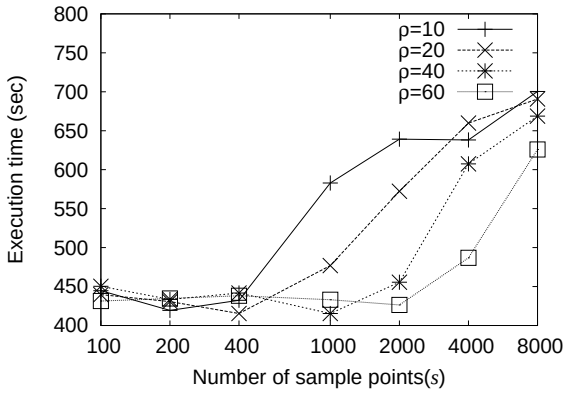


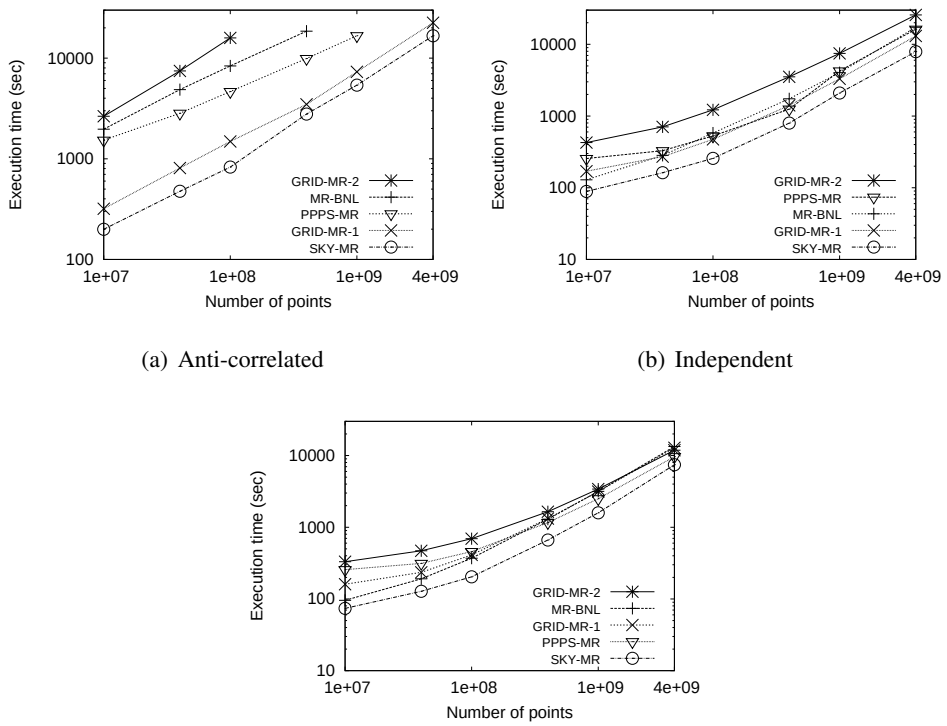
Figure 4.7: SKY-MR with varying s and ρ

leaf nodes. Thus, as the sample size s increases, the number of unpruned leaf nodes of a *sky-quadtree* which receives such points from other unpruned nodes increases and *SKY-MR* becomes inefficient due to high network costs. Decreasing ρ has also a similar effect of increasing the sample size s.

Varying n: We varied n from 10^7 to 4×10^9 and plot the running times of the algorithms in Figure 4.8. *SKY-MR* is always better than *SKY-MR-S/M* since it switches to *SKY-MR-S* or *SKY-MR-M* adaptively based on the number of local skyline points. Thus, we do not report the performance of *SKY-MR-S/M* in the rest of the paper.

Since the number of skyline points of the anti-correlated data sets is generally larger than those of the independent data sets and the correlated data sets, the algorithms with the anti-correlated data sets take generally more execution time than those of the other data sets.

GRID-MR-2 is always the worst performer due to the high cost of computing the relaxed skyline grids from t^d grids (e.g., when $t = 10$ and $d = 6$, we have $t^d = 10^6$ number of grids). *MR-BNL* performs better than *GRID-MR-2*, but it is still slower than *SKY-MR* because *MR-BNL* calculates the global skyline in a single machine only. *GRID-MR-1* performs poorly because it broadcasts all points p in each relaxed skyline grid to every other grid containing points which may be dominated by p. Since *SKY-*



(a) Anti-correlated

(b) Independent

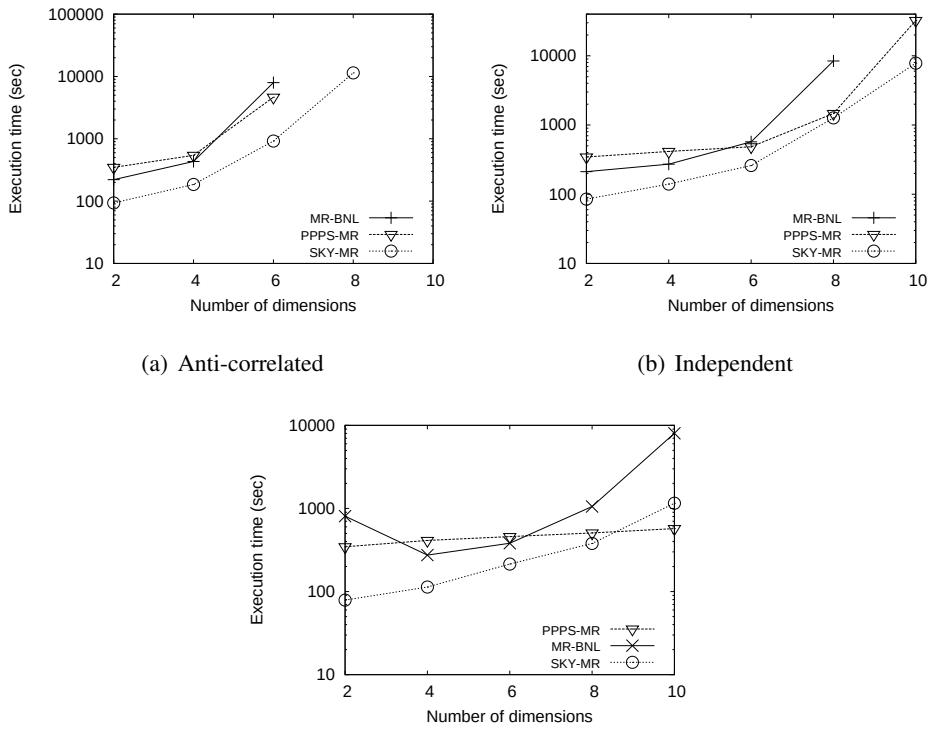
(c) Correlated

Figure 4.8: Varying the number of points (n) for skyline processing

MR filters out non-skyline points effectively using the *sky-quadtree*, it shows the best performance.

Varying d: With varying d from 2 to 10, we plot the execution times of the algorithms except *GRID-MR-1/2* in Figure 4.9 because they show similar patterns with varying n.

The execution times of all algorithms increase gradually with increasing d since checking the dominance relationship between two points becomes more expensive with large values of d. Furthermore, when $d = 2$, *MR-BNL* becomes slow because *MR-BNL* utilizes only 4 ($= 2^d$) machines out of 10 machines. For the independent and anti-correlated data sets, *PPPS-MR* becomes slow since the last two partitions are merged in a single machine. However, *PPPS-MR* becomes fast for the correlated data



(a) Anti-correlated

(b) Independent

(c) Correlated

Figure 4.9: Varying the number of dimensions (d) for skyline processing

sets, since there are a small number of local skyline points and merging them can be done quickly. The graphs confirm that *SKY-MR* is generally the best performer.

Varying t: We show the relative speed of the tested algorithms averaged over all data sets in Figure 4.10. That is, for each algorithm, we plot its running time with 5 machines divided by its running time with t machines. For example, if the running times of *SKY-MR* with 5 and 20 machines are T_5 and T_{20} respectively, we plot the ratio T_5/T_{20} for t=20. In an ideal case, if the number of machines increases by 4 times from 5 to 20, the speed will be 4 times faster. We also plot the ideal speedup curve in the graphs of Figure 4.10. For the relative speed, our proposed algorithm *SKY-MR* shows the best scalability since *SKY-MR* effectively prunes data by partitioning with the *sky-quadtrees* and utilizes the virtual max points and sky-filter points to reduce the

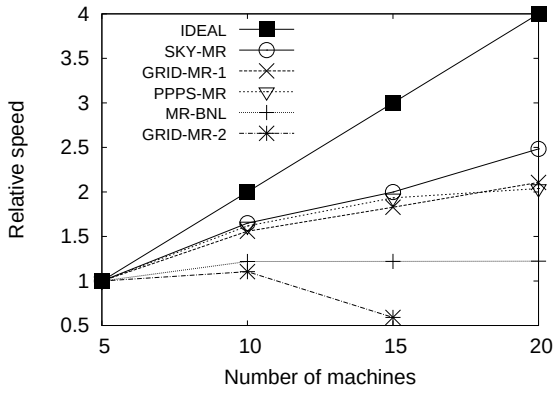


Figure 4.10: Relative speed with varying the number of machines (t)

SKY-MR	F and V	F	V	NONE
Correlated	218	229	234	242
Independent	260	281	283	283
Anti-correlated	840	1146	952	1207

Table 4.3: Effects of the virtual max points (V) and sky-filter points (F) (sec)

unnecessary comparisons.

The effects of the virtual max and sky-filter points: To evaluate the performance improvements by utilizing the virtual max points and sky-filter points, we report the execution times of *SKY-MR* using virtual max point and the sky-filter points (F and V), *SKY-MR* using only the sky-filter points (F), *SKY-MR* using only the virtual max points(V), and *SKY-MR* without both points (NONE) in Table 4.3. For the correlated data set, the performance of *SKY-MR* is improved mainly by the sky-filter points, since most of the local skyline points are removed by sky-filter points in the third phase. For the anti-correlated data set, since the virtual max points reduce unnecessary comparisons between the local skyline points by Proposition 4.1.5, the running time decreases. Since *SKY-MR* utilizes both sky-filter points and virtual max points, it runs faster than the other algorithms.

Distribution	Algorithm	10^5	10^6	10^7	10^8	10^9
Anti-correlated	*SKY-SC*	1.5	91.3	949.8	8906.8	-
	BBS	39.9	603.2	16334.5	-	-
	SQL	410	1168	-	-	-
Independent	*SKY-SC*	0.6	1.6	10.1	70.4	-
	BBS	0.3	1.9	12.6	2275.8	-
	SQL	117	3386	-	-	-
Correlated	*SKY-SC*	0.2	0.5	3.3	41.5	-
	BBS	0.1	0.2	1.3	11.6	-
	SQL	6.6	113	-	-	-

Table 4.4: Varying n on a single core machine (sec)

4.2.2 Performance Results in Other Environments

We finally present the experimental results by comparing the performance of our ported algorithms to other environments with the existing state-of-the-art algorithms in such environments. We did experiments with varying n and d, but reported only the experimental results with varying n.

Single core machine: We compared our serial algorithm *SKY-SC* to the state-of-the-art serial algorithm *BBS* [38] which utilizes an R*-tree on a single core machine. We report the average execution times with varying n from 10^5 to 10^9 in Table 4.4. We do not include the construction time of R*-trees for *BBS*, but we include the construction time of sky-quadtrees for *SKY-SC* in Table 4.4. Whenever any algorithm did not finish due to lack of memory, we do not show the running time in Table 4.4.

BBS finds skyline points progressively in increasing order of their distances to the origin. When the number of skyline points is small (i.e., correlated data), most of minimum bounding rectangles (MBRs) of R*-trees are pruned by the skyline points found at the beginning of *BBS* and thus *BBS* shows slightly better performance than *SKY-SC*. However, when the number of skyline points is large (i.e., independent or anti-correlated data), many MBRs are not pruned by the skyline points. Since *SKY-SC*

Distribution	Algorithm	10^7	4×10^7	10^8	4×10^8	10^9
Anti-correlated	*SKY-MC*	116.8	380.2	877.4	-	-
	PPPS	2201.5	10887	21736	-	-
Independent	*SKY-MC*	11.3	37.7	81.4	290.8	666.0
	PPPS	14.9	66.5	193.1	1171.3	-
Correlated	*SKY-MC*	5.1	17.8	44.5	170.3	410.8
	PPPS	4.9	20.5	50.2	216.3	512.2

Table 4.5: Varying n on a multi-core machine (sec)

filters out non-skyline points effectively using the *sky-quadtree* as well as virtual max points and sky-filter points, when the number of skyline points becomes large, *SKY-SC* performs much better than *BBS*.

Multi-core machine: We evaluated our *SKY-MC* and *PPPS* [25] devised for multi-core machines. Experiments were performed on a 32-core machine of Intel(R) Xeon(TM) E7 CPU 2.67GHz with 128GB of main memory running Linux. We show the average execution times with varying n from 10^7 to 10^9 in Table 4.5. Whenever any algorithm did not finish due to lack of memory, we do not show the running time in the table. As shown in Table 4.5, *SKY-MC* is much better than *PPPS* for all cases even if our work is originally developed for MapReduce. The reason is that *SKY-MC* filters out non-skyline points effectively using the *sky-quadtree* as well as virtual max points and sky-filter points.

MPI: We compared our *SKY-MP* with *GRID1* and *GRID2* proposed in [2]. We used MPICH2 [27] for the implementations of MP-model and GMP-model. We report the average execution times with varying n in Table 4.6. Whenever any algorithm did not finish within 8 hours, we do not show the running time in the table. Similar to the experiments with multi-core machines, our *SKY-MP* performs better than the others due to effective pruning.

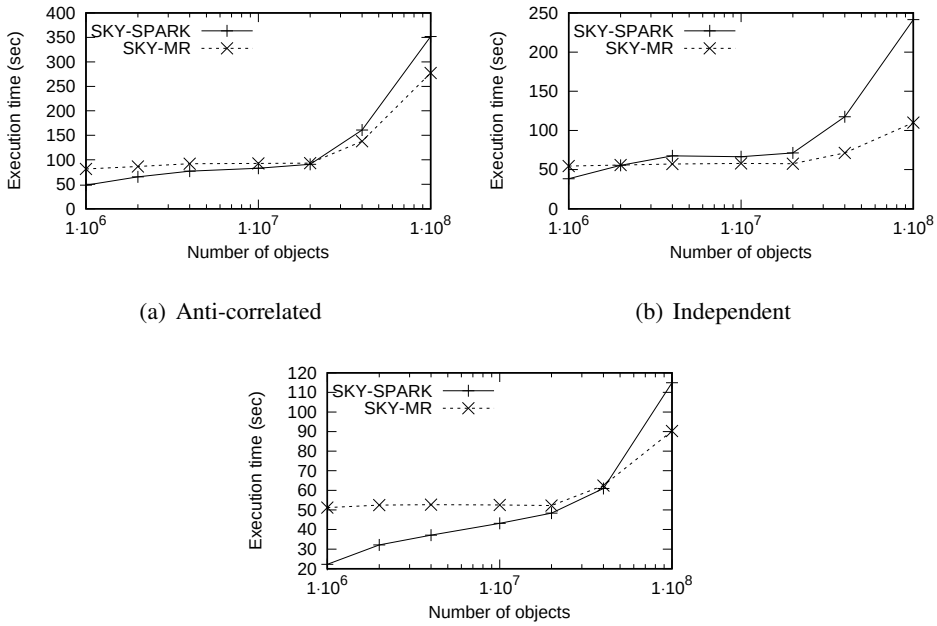
Distribution	Algorithm	10^7	4×10^7	10^8	4×10^8	10^9
Anti-correlated	*SKY-MP*	190	665	1478	4513	9509
	GRID1	226	699	1586	4651	9698
	GRID2	5134	20955	-	-	-
Independent	*SKY-MP*	12	53	183	669	1583
	GRID1	20	69	142	785	1801
	GRID2	641	860	1139	2300	4250
Correlated	*SKY-MP*	4.9	16	51	365	669
	GRID1	5.4	28	66	442	1126
	GRID2	237	447	642	1250	2322

Table 4.6: Varying n on MPI (sec)

Spark: Spark [46] is especially useful for parallel processing of distributed data with iterative algorithms. Spark uses Resilient Distributed Datasets (RDDs) which are a collection of elements partitioned across the nodes of a cluster and can be operated on in parallel. Since RDDs can be kept in memory, algorithms can iterate over RDD data many times very efficiently. It is generally known that Spark is useful for iterative algorithms (e.g., K-means) and interactive data analysis. However, due to its in-memory style data structure, MapRedure is still a promising parallel framework for the applications in which the intermediate results become much larger than input data and/or each machine requires a large (but not disjoint) portion of the intermediate result to generate a final result.

We implemented our *SKY-MR* using Spark, called *SKY-SPARK*, and compared it with the *SKY-MR* using the MapReduce framework. In Figure 4.11, we plot the execution times of *SKY-MR* and *SKY-SPARK* with varying the number of points n from 10^6 to 10^8. Due to the overhead of invoking MapReduce jobs, *SKY-MR* always takes more time than 50 seconds. However, the execution time of *SKY-SPARK* can be less than 50 seconds when the size of data is small. As the number of points increases, *SKY-MR* becomes faster than *SKY-SPARK* since skyline computation requires more larger

intermediate results (i.e., local skylines) to compute the global skyline.



(a) Anti-correlated

(b) Independent

(c) Correlated

Figure 4.11: Varying the number of points (n) on Spark

Chapter 5

Parallel Reverse Skyline Query Processing

5.1 RSKY-MR: Our Reverse Skyline Computation Algorithm

To develop efficient reverse skyline algorithm, we first study the characteristics of the reverse skyline query. Then, we present out MapReduce reverse skyline algorithm called *RSKY-MR*.

Given a query point q, we divide the data D into 2^d orthants with respect to a query point q as illustrated in Figure 5.1 to filter out non-reverse skyline points efficiently. The set of all data points located in an orthant o is denoted as D_o. For each orthant o represented by the region $\langle [o(1)^-, o(1)^+], \cdots, [o(d)^-, o(d)^+] \rangle$, the id, denoted by $a_1 a_2 \cdots a_d$, is assigned where $a_i = 0$ if $[o(i)^-, o(i)^+] = [-\infty, q(i)]$ and $a_i = 1$ if $[o(i)^-, o(i)^+] = [q(i), \infty]$. For instance, the orthant 01 represents the region $\langle [-\infty, 50], [25, \infty] \rangle$ in Figure 5.1.

Lemma 5.1.1 *For p_i, $p_j \in D$, if p_j is located at an orthant o and p_j dynamically dominates a query point q with respect to p_i (i.e. $p_j \prec_{p_i} q$), then p_i is also in the same orthant o.*

Proof: When $p_j \prec_{p_i} q$, we have $|q(k) - p_i(k)| \geq |p_j(k) - p_i(k)|$ for all $k = 1, \cdots, d$. Squaring both sides and rearranging terms, the above condition becomes equivalent

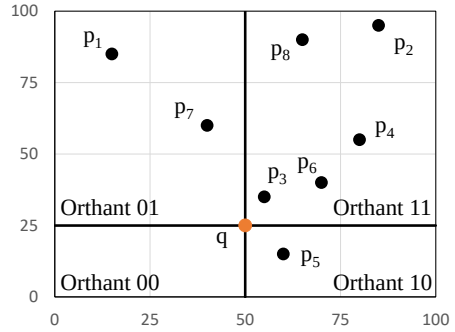


Figure 5.1: The space split with respect to $q = \langle 50, 25 \rangle$

to $0 \geq (p_j(k) - p_i(k))^2 - (q(k) - p_i(k))^2$. Then, we get $0 \geq (p_j(k) + q(k) - 2p_i(k)) \cdot (p_j(k) - q(k)) = -2 \cdot (p_i(k) - q(k))(p_j(k) - q(k)) + (p_j(k) - q(k))^2$. Since $2 \cdot (p_i(k) - q(k)) \cdot (p_j(k) - q(k)) \geq (p_j(k) - q(k))^2 \geq 0$ for all k=1,$\cdots$,d, $(p_i(k) - q(k))$ and $(p_j(k) - q(k))$ have the same sign. Thus, p_i and p_j are in the same orthant. ∎

Note that $p_i \notin RSL(q, D)$ if there exists a point $p_j \in D$ such that $p_j \prec_{p_i} q$. Since every point dynamically dominating q with respect to p_i is always located in the same orthants in which p_i is located by Lemma 5.1.1, our brute-force algorithm *BR-RSKY-MR* calculates the reverse skyline of each orthant independently and merges all reverse skylines.

To compute $RSL(q, D)$ efficiently, we next devise the algorithm *RSKY-MR* with the following three phases. *RSKY-MR* utilizes *rsky-quadtrees* which are a variant of *sky-quadtrees*. The pseudocode of *RSKY-MR* is presented in Figure 5.2.

(1) **Rsky-quadtree building phase:** By running *RSKY-QTREE*, we build an *rsky-quadtree* associated with each orthant from a sample obtained by reservoir sampling [51].

(2) **Local reverse skyline phase:** For each unpruned leaf node of every *rsky-quadtree*, we compute candidate reverse skyline points in parallel by invoking *L-RSKY-MR*. In addition, the local dynamic skyline of the midpoints between every point p ($\in D$) and q is selected to prune non-reverse skyline points in the next phase.

Function RSKY-MR(D, q, ρ, d, δ)

D: a dataset, q: a query point, ρ: the split threshold,

d: the dimension, δ: strong reverse skyline threshold

begin

1. $sample$ = ReservoirSampling(D);
2. $rsky\text{-}quadtrees$=RSKY-QTREE($sample$, ρ, d);
3. Broadcast q and $rsky\text{-}quadtrees$;
4. $Local\text{-}RSL$ = RunMapReduce(L-RSKY-MR);
5. **if** $Local\text{-}RSL$.size $\geq t$
6. **then** Broadcast q, *non-empty leaf node ids*;
7. RSL = RunMapReduce(G-RSKY-MR);
8. **else** RSL = G-RSKY($Local\text{-}RSL$);
9. return RSL;

end

Figure 5.2: The RSKY-MR algorithm

(3) **Global reverse skyline phase:** We check in parallel whether each candidate reverse skyline point is actually a global reverse skyline point. Similar to *SKY-MR*, depending on the number of candidate reverse skyline points produced in the previous phase, the global reverse skyline is computed on a single machine by calling *G-RSKY* or on multiple machines by invoking *G-RSKY-MR*.

5.1.1 RSKY-QTREE: The Rsky-Quadtree Building Algorithm

For effective pruning with *rsky-quadtrees*, we adopt the idea of midpoints introduced in [16, 53]. The midpoint between a point p and a query point q is defined as $mid(p, q)$ = $\langle (p(1)+q(1))/2, \cdots, (p(d)+q(d))/2 \rangle$. Since $|(p(i)+q(i))/2-q(i)| \leq |p(i)-q(i)|$ holds for each dimension i, the following is trivially true.

Proposition 5.1.2 *The midpoint $mid(p, q)$ always dynamically dominates p with respect to q.*

We develop the following lemmas to identify efficiently whether a point in D is a global reverse skyline point.

Lemma 5.1.3 *Given an orthant o and a query point q, $p_i \in D_o$ is not in the reverse skyline of D_o with respect to q, if and only if there exists another point $p_j \in D_o$ s.t. $mid(p_j, q) \prec_q p_i$.*

Proof: ($\Rightarrow$:) When $p_i \notin RSL(q, D_o)$, $q \notin DSL(p_i, D_o)$ holds and there exists a point $p_j (\in D_o)$ s.t. $p_j \prec_{p_i} q$. Since $|p_j(k) - p_i(k)| \leq |q(k) - p_i(k)|$ for all k=1, $\cdots$, d, we can derive $(p_j(k) - q(k))^2 \leq 2 \cdot (p_i(k) - q(k))(p_j(k) - q(k))$, as shown in the proof of Lemma 5.1.1. Since p_i and p_j are in the same orthant by Lemma 5.1.1, $|p_j(k)-q(k)|/2 = |(p_j(k)+q(k))/2-q(k)| \leq |p_i(k)-q(k)|$. Similarly, we can derive $|(p_j(k) + q(k))/2 - q(k)| < |p_i(k) - q(k)|$ for at least a single dimension k. Thus, by the definition of the midpoints, there exists $p_j \in D_o$ such that $mid(p_j, q) \prec_q p_i$.

($\Leftarrow$:) We have $|(p_j(k) + q(k))/2 - q(k)| = |p_j(k) - q(k)|/2 \leq |p_i(k) - q(k)|$ for all k=1, $\cdots$, d, when $mid(p_j, q) \prec_q p_i$. By multiplying $2(p_j(k) - q(k))$ to both sides, we get

$$(p_j(k) - q(k))^2 \leq 2(p_j(k) - q(j))(p_i(k) - q(k)). \tag{5.1}$$

Since $|p_j(k)-p_i(k)| = |p_j(k)-q(k)-p_i(k)+q(k)|$, we have $(p_j(k)-p_i(k))^2 = (p_j(k) - q(k))^2 + (p_i(k) - q(k))^2 - 2(p_j(k) - q(k))(p_i(k) - q(k))$. By replacing $(p_j(k)-q(k))^2$ with $2(p_j(k)-q(j))(p_i(k)-q(k))$ and using the above inequality (5.1), we obtain $(p_j(k)-p_i(k))^2 \leq (p_i(k)-q(k))^2$ and thus $|p_j(k)-p_i(k))| \leq |p_i(k)-q(k)|$ holds for every dimension k. Similarly, we can show that $|p_j(k) - p_i(k))| < |p_i(k) - q(k)|$ for at least a dimension k. Consequently, $p_j \prec_{p_i} q$ and q cannot be a dynamic skyline point with respect to p_i. In other words, if $mid(p_j, q) \prec_q p_i, p_i \notin RSL(q, D_o)$.

∎

Lemma 5.1.4 *For two points p_i, $p_j \in D_o$, if $p_j \prec_q p_i$, we have $p_i \notin RSL(q, D_o)$.*

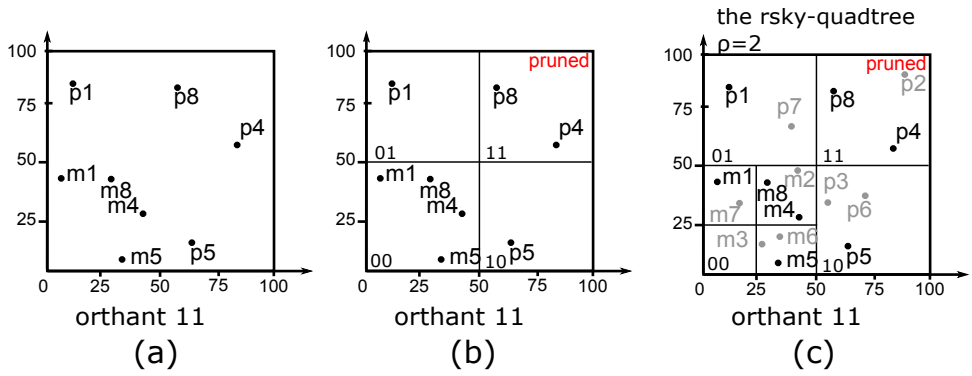


Figure 5.3: An example of rsky-quadtree building

Proof: Since $mid(p_j, q) \prec_q p_j$ by Proposition 5.1.2, $mid(p_j, q) \prec_q p_i$ holds. Thus, $p_i \notin RSL(q, D_o)$ due to Lemma 5.1.3. ∎

We develop the procedure *RSKY-QTREE* to build *rsky-quadtrees*. The main differences from *SKY-QTREE* presented in Section 4.1.1 are as follows: (1) Given a query point q and a data set D, an *rsky-quadtree* associated with each orthant o is built by inserting sample points $p \in D_o (\subset D)$ and their midpoints. (2) In an *rsky-quadtree*, every node n is marked as "$pruned$" if there exists a point $p \in D_o$ dynamically dominating the node n since all points belonging to the node n cannot be in the reverse skyline. (3) In an *rsky-quadtree*, every node n is also marked as "$pruned$" if there exist at least two points p_i, $p_j \in D_o$ whose $mid(p_i, q)$ and $mid(p_j, q)$ dynamically dominate the node n. Since $mid(p, q)$ always dynamically dominates p according to Proposition 5.1.2, we need at least two midpoints to prune a node of an *rsky-quadtree*.

Example 5.1.5 *Consider the data D in Figure 3.1(a) with a query point $q = \langle 0, 0 \rangle$ and the split threshold $\rho = 2$. Dividing D into 4 orthants with respect to q results in a non-empty orthant o with id=11 only. Assume $\{p_1, p_4, p_5, p_8\}$ is a sample of D. All sample points and their midpoints are inserted into the root node as shown in Figure 5.3(a) where m_i represents $mid(p_i, q)$. We recursively subdivide the data space starting from the root node until the number of points and midpoints in each*

unpruned leaf node of the rsky-quadtree is at most ρ. Since there are multiple midpoints dynamically dominating the node with id=11 (i.e., m_1, m_4, m_5 and m_8), it is marked as "pruned" as illustrated in Figure 5.3(b). The rsky-quadtree constructed from the sample is shown in Figure 5.3(c). ∎

5.1.2 Computations of Reverse Skylines using Rsky-Quadtrees

To illustrate how to compute the reverse skylines using *rsky-quadtrees*, we utilize the following definitions.

Definition 5.1.6 *For a leaf node n, let $L_p(n) = \{p \in D |\ p$ is located in $region(n)\}$, $L_m(n) = \{mid(p,q) | p \in D$ s.t. $mid(p,q)$ is located in $region(n)\}$ and $L(n) = L_p(n) \cup L_m(n)$. The* strong reverse skyline *$SRSL(q, L(n))$ of $L(n)$ with respect to q is $\{p_j \in L_p(n) \mid p_j \in RSL(q, L_p(n))$ and $\nexists m (\neq mid(p_i, q)) \in L_m(n)$ s.t. $m \prec_q p_j\}$.*

A reverse skyline point p is a strong reverse skyline point of the node containing p since p is not dominated by the midpoints of all other points in D according to Lemma 5.1.3. Thus, if we can eliminate all non-reverse skyline points from $SRSL(q, L(n))$ of every node n in *rsky-quadtrees*, we can obtain the reverse skyline.

To eliminate non-reverse skyline points in each node n, we need the local dynamic skyline midpoints $DSL(q,\ L_m(n))$ of every other node. For example, consider the points p_i, p_j, $p_k \in D_o$. If $mid(p_k, q) \prec_q mid(p_j, q)$ and $mid(p_j, q) \prec_q p_i$, we have $mid(p_k, q) \prec_q p_i$ and $p_i \notin RSL(q, D_o)$ by Lemma 5.1.3. Thus, if $mid(p_k, q) \prec_q mid(p_j, q)$, we only need $mid(p_k, q)$ to check whether p_i is a reverse skyline point or not.

The local dynamic skyline midpoints themselves are not sufficient, however, to eliminate all non-reverse skyline points from the strong reverse skylines. For instance, consider the point p_i in Figure 5.4. Although p_i is a strong reverse skyline point, $p_i \notin RSL(q, D)$ because $m_j (= mid(p_j, q)) \prec_q p_i$ holds. Since $m_i (= mid(p_i, q)) \prec_q m_j$ in $node(00)$, m_j is not a local dynamic skyline midpoint. Thus, if we only use the local

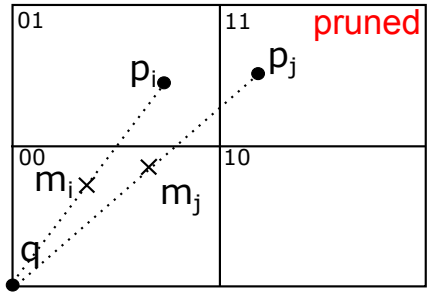


Figure 5.4: Points and their midpoints in an orthant

dynamic skyline midpoints blindly, we cannot eliminate p_i correctly. However, since m_i is a local dynamic skyline midpoint, we can annotate m_i with a special symbol, representing that m_j dynamically dominates p_i, in order to utilize m_i to prune p_i. We call such annotated midpoints the *verification midpoints* as defined below:

Definition 5.1.7 *Given a query point q and the set of midpoints $L_m(n)$ of an unpruned leaf node n in an* rsky-quadtree*, consider a point p such that $mid(p,q) \in DSL(q, L_m(n))$. The midpoint $mid(p,q)$ is a* verification midpoint *if there exists $m_j \in L_m(n)$ such that $mid(p,q) \prec_q m_j \prec_q p$.*

Lemma 5.1.8 *Given a query point q and an rsky-quadtree r of an orthant o, $p \in D_o$ is a reverse skyline point if and only if (1) p is in $SRSL(q, L(n))$ of an unpruned leaf node n in r,(2) $mid(p,q)$ is not a verification midpoint, and (3) for every unpruned leaf node n' in r, there does not exist $m(\neq mid(p,q)) \in DSL(q, L_m(n'))$ such that $m \prec_q p$.*

Proof: ($\Rightarrow$:) We prove the contrapositive: when one of the three conditions is not satisfied, $p \notin RSL(q, D)$.

When the condition (1) is not satisfied, by Definition 5.1.6, there is $m(\neq mid(p,q)) \in L_m(n)$ or $p_i(\neq p) \in L_p(n)$ s.t. $m \prec_q p$ or $mid(p_i, q) \prec_q p$. If the condition (2) is not satisfied, there is a midpoint m s.t. $mid(p,q) \prec_q m \prec_q p$ by Definition 5.1.7. When the condition (3) is not satisfied, there is $m(\neq mid(p,q)) \in DSL(q, L_m)$ s.t.

$m \prec_q p$. Thus, whenever one of the three conditions is not satisfied, there exists a midpoint $m \neq mid(p, q)$ s.t $m \prec_q p$ and $p \notin RSL(q, D_o)$ according to Lemma 5.1.3. Therefore, $p \in RSL(q, D_o)$ implies that all three conditions are satisfied.

($\Leftarrow$:) For the purpose of contradiction, suppose $p \notin RSL(q, D_o)$. Based on Lemma 5.1.3, there exists a midpoint $m \neq mid(p, q)$ s.t. $m \prec_q p$ for a point p contained in the orthant o. Without loss of generality, assume that m is in an unpruned leaf node. (Otherwise, let m be m_u where m_u ($\neq mid(p, q)$) is a midpoint which is in an unpruned leaf node and dynamically dominates m. The midpoint m_u always exists by the properties of the *rsky-quadtrees* and we have $m_u \prec_q p$).

When p and m are located in the same node n of r, $p \notin SRSL(q, L(n))$ since $m \prec_q p$. It contradicts the condition (1) resulting that p and m should be located in different nodes of r. In the unpruned leaf node n_m containing m, if $m \in DSL(q, L_m(n_m))$, it contradicts the condition (3). Therefore, there exists another midpoint $m' \in DSL(q, L_m(n_m))$ s.t. $m' \prec_q m$. If $m' \neq mid(p, q)$, it also contradicts the condition (3) since $m' \prec_q m \prec_q p$. This implies that $m' = mid(p, q)$. Since m and $mid(p, q)$ are located in the same unpruned leaf node and $mid(p, q) \prec_q m \prec_q p$, $mid(p, q)$ is a verification midpoint by Definition 5.1.7. It contradicts the condition (2). Therefore, if all three conditions hold, p is a reverse skyline point.∎

We next define the *reverse virtual max point* of each leaf node of an *rsky-quadtree* and provide the property of the reverse virtual max points.

Definition 5.1.9 *The reverse virtual max point of each leaf node n of an rsky-quadtree, denoted by rvp_n, is defined as the point whose k-th dimensional value $rvp_n(k)$ is $max_{p_i \in SRSL(q,L(n))} |p_i(k) - q(k)|$ for every k=1,2,$\cdots$,d.*

Proposition 5.1.10 *If a midpoint m does not dynamically dominate the reverse virtual max point of a leaf node n in an rsky-quadtree, m does not dynamically dominate every strong reverse skyline point in $region(n)$.*

We omit the proof of Proposition 5.1.10 because it is similar to that of Proposition

Function L-RSKY-MR.map(key, p)

key: null, p: a point

begin

1. $rsky\text{-}qtrees$ = LoadTrees(), q = LoadQuery();
2. $O(p)$ = FindOrthants(p, q);
3. **for** each $o \in O(p)$ **do**
4. n_p = GetNode(p, $rsky\text{-}qtrees[o]$);
5. **if** $n_p.pruned$ == **false then**
6. emit((o,n_p),("P",p));
7. n_m = GetNode(mid(p, q), $rsky\text{-}qtrees[o]$);
8. **if** $n_m.pruned$ == **false then**
9. emit((o,n_m),("M",mid(p,q)));

end

Figure 5.5: The map function of the L-RSKY-MR algorithm

4.1.5 in Section 4.1.2.

5.1.3 L-RSKY-MR: The Local Reverse Skyline Computation Algorithm

Based on Lemmas 5.1.1, 5.1.3 and 5.1.8, the procedure *L-RSKY-MR* computes the strong reverse skyline and local dynamic skyline midpoints in every unpruned leaf node of all *rsky-quadtrees*. The pseudocodes of map and reduce functions are shown in Figures 5.5 and 5.6, respectively.

Each map function is called with a point p in D. To check whether a point p is a reverse skyline point or not, we examine only the points in each orthant containing p by Lemma 5.1.1. Thus, in the map function called with p, we examine each orthant o containing p independently. Note that if a point $p \in D_o$ is in the pruned leaf node of the *rsky-quadtree*, p is not a global reverse skyline point due to Lemma 5.1.3 since there exists a midpoint of another point in D_o which dynamically dominates p with respect to q. For each orthant o, we perform the following two steps: (1) We check whether p

Function L-RSKY-MR.reduce(key, L)

key: (orthant id o, a node id n), L: a list of points and midpoints

begin

1. q = LoadQuery();
2. $SRSL$ = StrongReverseSkyline(q,L);
3. output(key, $SRSL$);
4. $L_m = \{m \in L | m$ has symbol "M"$\}$;
5. DSL = DynamicSkyline(q,L_m);
6. **for** each midpoint m in DSL **do**
7. **if** IsVerificationMidpoint(m, L_m) **then**
8. output (key, ("V",m);
9. **else** output (key, ("M",m);
10. append(RSKY-FILTER, FilterPoint(DSL);
11. append(RVIRTUAL, VirtualMax($SRSL$);

end

Figure 5.6: The reduce function of the L-RSKY-MR algorithm

belongs to an unpruned leaf node of o's *rsky-quadtree*. If it does, we emit $\langle (o, n_p)$,("P", $p)\rangle$ where "P" represents that p is a point (in lines 2-6 of *L-RSKY-MR.map*). (2) If $mid(p, q)$ belongs to an unpruned leaf node n_m, we output $\langle (o, n_m)$, ("M", $mid(p, q))\rangle$ where "M" denotes that m is a midpoint (in lines 7-9).

After the shuffling phase groups the output of the map functions according to each distinct unpruned leaf node, a reduce function is called with each distinct group. For each distinct group (o, n), the list $L(n)$, which is $L_p(n) \cup L_m(n)$ as defined in Definition 5.1.6, is generated.

Consider the reduce function called with a distinct group (o, n) and the input value list $L(n)$. The reduce function computes the strong reverse skyline (i.e., $SRSL(q, L(n))$) and the local dynamic skyline of midpoints (i.e., $DLS(q, L_m(n))$) according to Lemma 5.1.8 (in lines 1-5 of *L-RSKY-MR.reduce*). For every strong reverse skyline point, the reduce

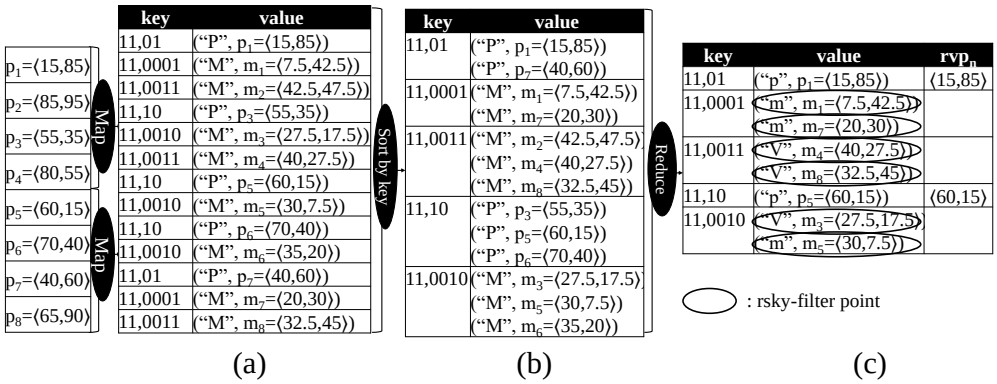


Figure 5.7: The data flow in the local reverse skyline phase of RSKY-MR

function outputs the key-value pair $\langle (o, n), (\text{“P”}, p)\rangle$ (in line 3). In addition, the reduce function emits $\langle (o, n), (\text{“V”}, v)\rangle$ for every verification midpoint v in $DSL(q, L_m(n))$ defined in Definition 5.1.7 (in lines 6-8). For every midpoint m in $DSL(q, L_m(n))$ which is not a verification midpoint, the reduce function outputs $\langle (o, n), (\text{“M”}, m)\rangle$(in line 9).

Similar to *L-SKY-MR*, for each dimension, the reduce function chooses a single midpoint, called an *rsky-filter midpoint*, in $DSL(q, L_m(n))$ which has the minimum value of the dimension. The reduce function also computes the reverse virtual max point of the leaf node n. Finally, the reduce function outputs the rsky-filter midpoints and the reverse virtual max point to the files called RSKY-FILTER and RVIRTUAL in HDFS respectively (in lines 10-11).

Example 5.1.11 *Consider the rsky-quadtree in Example 5.1.5. In the local reverse skyline phase, a map function is invoked with each point $p \in D$. For instance, the map function with p_5 outputs $\langle (11, 10), (\text{“P”}, p_5)\rangle$ because p_5 belongs to the unpruned leaf node, $node(10)$, in the orthant with id=11. In addition, since $mid(p_5, q)$ belongs to an unpruned leaf node, $node(0010)$, in the same orthant, the map function also outputs $\langle (11, 0010), (\text{“M”}, mid(p_5, q))\rangle$. The key-value pairs output by all map functions are shown in Figure 5.7(a). The output of the shuffling phase is presented in Figure 5.7(b).*

Function G-RSKY-MR.map(key, p)

key: (an orthant id o, a node id n), p: (a point or a midpoint p, $mark$)

begin

1. q = LoadQuery();
2. **if** IsPoint(p) **then**
3. **if** DynamicDominatedByFilterPoints(p,q,o) **then return**;
4. emit(key, ("P", p));
5. **if** IsMidpoint(p) **then**
6. $rsky\text{-}quadtrees$ = LoadTrees()
7. $nodes$ = LoadNonEmptyNodes(o);
8. **for** each node id n_i in $nodes$
9. **if** IsNeeded(n, n_i) **then**
10. output((o,n_i), (mark, p));

end

Figure 5.8: The map function of the G-RSKY-MR algorithm

For each distinct key (o, n), a reduce function is called with the list of points and midpoints in $region(n)$. For instance, the reduce function invoked with the key (11,10) receives $\{p_3, p_5, p_6\}$ as input value list and outputs $\langle$(11,10),(p_5,"P")$\rangle$ since p_5 is a strong reverse skyline point. The reduce function next calculates the verification midpoints, reverse virtual max point and rsky-filter points. In $node(0011)$, m_4 is annotated with "V" since m_2 is in $region(\ n)$ and m_2 dominates p_4. In $node(0011)$, m_4 and m_8 are selected as the rsky-filter midpoints since $m_8(1) = 32.5$ and $m_4(2) = 27.5$ are the minimum value on the first and second dimensions respectively. In Figure 5.7(c), we have shown the output emitted by all reduce functions where the rsky-filter midpoints are circled. ∎

5.1.4 G-RSKY-MR: The Global Reverse Skyline Computation Algorithm

The parallel algorithm *G-RSKY-MR* finds the global reverse skyline points independently in each non-empty unpruned leaf node by Propositions 4.1.2, 5.1.10 and Lemma 5.1.8. We omit the details of the serial algorithm *G-RSKY* due to space limitations.

For every strong reverse skyline point p, we check whether (1) p is not dynamically dominated by a local dynamic skyline midpoint m (i.e., $m \not\prec_q p$) and (2) p's midpoint is not one of the verification midpoints. If both conditions are satisfied, p is a global reverse skyline point due to Lemma 5.1.8. To check the condition (1), we examine whether $m \prec_q p$ for every midpoint m contained in all unpruned leaf nodes n_i. However, we do not need to check whether $m \prec_q p$ if there is k such that $sub(id(n_i), k) > sub(id(n_j), k)$ where n_j is the node containing p and $sub(id(n), k)$ is the k-th substring of n's id defined in Section 4.1.1. The reason is that we have $m \not\prec_q p$ for every point p in n_j according to Proposition 4.1.2. In addition, if $m \not\prec_q rvp_{n_j}$ (i.e., the reverse virtual max point of n_j), since $m \not\prec_q p$ for every strong reverse skyline point p belonging to n_j by Proposition 5.1.10, we do not need to check $m \prec_q p$ either.

The pseudocode of *G-RSKY-MR* is presented in Figure 5.8. Each map function is invoked with a strong reverse skyline point (i.e., annotated with "P") or a local dynamic skyline midpoint (i.e., annotated with "M" or "V") which were generated at the previous phase. Consider a map function called with a strong reverse skyline point p in an unpruned leaf node n in an orthant o. Note that p is not a global reverse skyline point if p is dominated by another point's midpoint by Lemma 5.1.3. Thus, the map function emits $\langle (o, n)$, ("P",$p)\rangle$ if p is not dynamically dominated by rsky-filter midpoints (in lines 1-4 of *G-RSKY-MR.map*).

For the map function called with a local dynamic skyline midpoint m contained in an unpruned leaf node n_m in an orthant o, the map function should find out all unpruned leaf nodes n requiring m to check whether n's strong reverse skyline points are the global reverse skyline points. If $n_m \not\prec n$ or $m \not\prec_q rvp_n$, n does not require m to check n's strong reverse skyline points by Propositions 4.1.2 and 5.1.10. For each

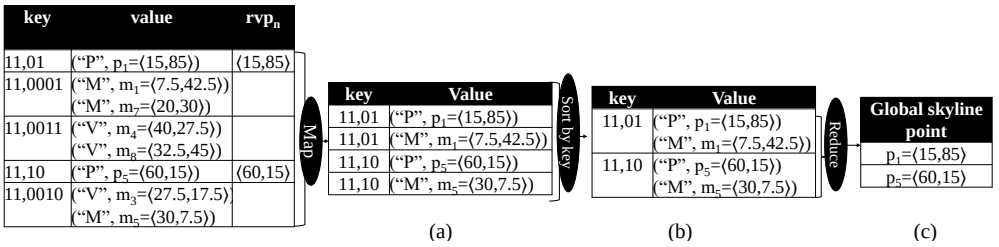


Figure 5.9: The data flow in the global reverse skyline phase of RSKY-MR

unpruned leaf node n which requires m, the map function outputs $\langle(o, n)$, ("V",$m)\rangle$ if m is a verification midpoint. Otherwise, it outputs $\langle(o, n)$, ("M",$m)\rangle$ (in lines 5-10).

The key-value pairs emitted by map functions are grouped according to each distinct unpruned leaf node in the shuffling phase and a reduce function is called with each distinct group. Each reduce function checks whether the strong reverse skyline points in a node are the global reverse skyline points based on Lemma 5.1.8. If a strong reverse skyline point p is dynamically dominated by the midpoints coming from the other nodes or p's midpoint is annotated with "V", p cannot be a reverse skyline point. Finally, the reduce function outputs the global skyline points.

Example 5.1.12 *Assume a map function is called with each point in the output of the local reverse skyline phase in Example 5.1.11. Since* m_1 *in* $node(0001)$ *dynamically dominates the reverse virtual max point of* $node(01)$*, the map function with* m_1 *emits* $\langle(11,$**01**$)$*, ("M",* $m_1)\rangle$*. However, since* m_3 *does not dynamically dominate the reverse virtual max point of* $node(1000)$*, we do not emit* $\langle(11, \mathbf{1000})$*, ("V",* $m_3)\rangle$*. Figures 5.9(a) and 5.9(b) show the output of all map functions and the result of the shuffling phase respectively.*

For every unpruned leaf node, a reduce function is called to see whether each strong reverse skyline point is actually a global reverse skyline point. For example, the input value list of the reduce function with the key (11, 01) is $\{p_1, m_1\}$*. Since* m_1 *is not a verification midpoint,* p_1 *is a global reverse skyline point. After every reduce function is finished,* $\{p_1, p_5\}$ *becomes the reverse skyline as in Figure 5.9(g).* ∎

Parameter	Range	Default
No. of samples (s)	$100 \sim 8,000$	1,000
Split threshold (ρ)	$10 \sim 60$	40
No. of points (n)	$10^7 \sim 4 \times 10^9$	10^8
No. of dimensions (d)	$2 \sim 10$	6
No. of machines (t)	$5 \sim 20$	10

Table 5.1: Parameters used for the reverse skyline algorithms

5.2 Experiment

We empirically evaluated the performance of our proposed algorithms using the parameters as summarized in Table 5.1. All experiments on MapReduce were performed on the cluster of 20 nodes of Intel(R) Core(TM) i3 CPU 3.3GHz machines with 4GB of main memory running Linux. The implementations of all algorithms were complied by Javac 1.6. We used Hadoop 1.0.3 for MapReduce [4]. The execution times in the graphs shown in this section are plotted in log scale.

5.2.1 Performance Results for Reverse Skylines

We next present the experimental results of the reverse skyline algorithms with randomly generated query points.

Data sets: We built three synthetic data sets which were randomly generated by *correlated*, *independent* and *anti-correlated* distributions. The three types of data sets are typically used to evaluate the performance of skyline algorithms [10]. The sizes of resulting synthetic data sets are varied from 392MB to 153GB depending on the number of points (n) as well as the number of dimensions (d).

Implemented algorithms: The MapReduce algorithms implemented for the reverse skyline are presented in Table 5.2. We ran all algorithms five times and measured the average execution times. We do not plot the execution times of some algorithms when they did not finish within 8 hours or they did not work due to some reasons such

Algorithm	Description
RSKY-MR-S/M	*RSKY-MR-S* utilizes *G-RSKY*. *RSKY-MR-M* utilizes *G-RSKY-MR*.
RSKY-MR	*RSKY-MR* adaptively selects *G-RSKY-MR* or *G-RSKY* with respect to the number of strong reverse skyline points. If it is less than 10^4, *G-RSKY* is selected.
BR-RSKY-MR	Our brute-force algorithm without using *rsky-quadtrees* in Section 5.1

Table 5.2: Implemented reverse skyline algorithms

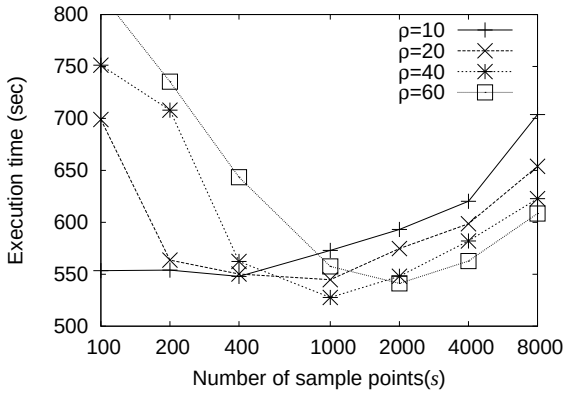
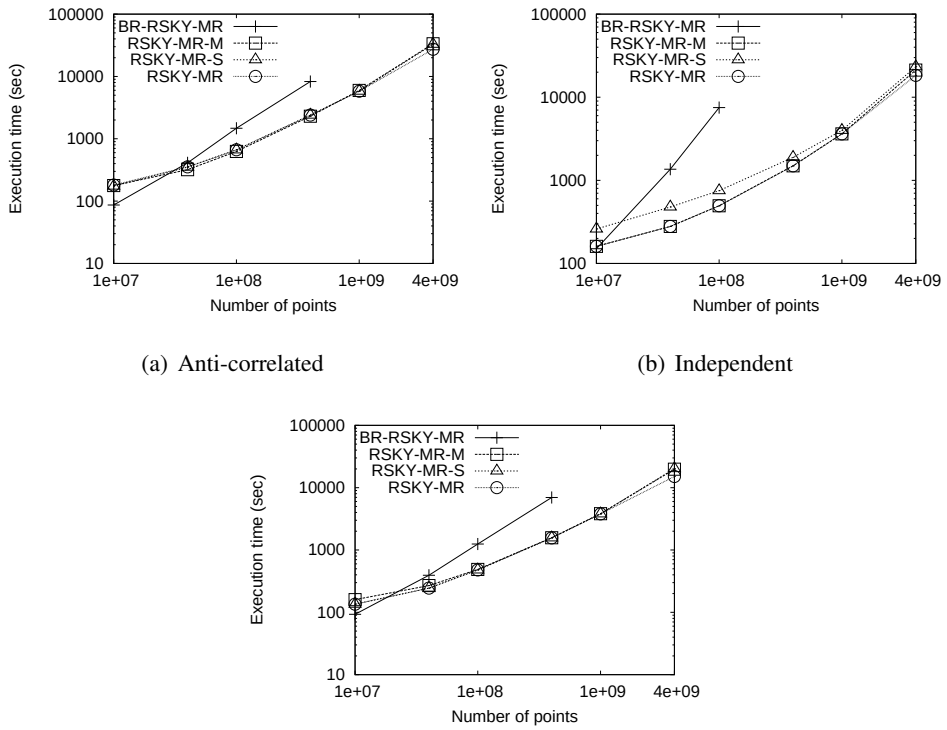


Figure 5.10: RSKY-MR with varying s and ρ

as out of memory.

Default values of s and ρ: To choose the proper values of s and ρ, we varied s from 100 to $8,000$ and ρ from 10 to 60. Figure 5.10 presents the average execution time over all data sets. We utilize $s = 1,000$ and $\rho = 40$ as the default values since *RSKY-MR* shows the best performance with those values. Note that small and large values of s make *RSKY-MR* inefficient, as we mentioned in Section 4.2.1.

Varying n: We varied n from 10^7 to 4×10^9 and plot the execution times in Figure 5.11. Similar to the skyline experimental results with varying n, the performance of every algorithm on the anti-correlated data set is worse than that of itself on the other data sets. When there is a skew in data such that a lot of points belong to an orthant,



(a) Anti-correlated

(b) Independent

(c) Correlated

Figure 5.11: Varying the number of points (n) for reverse skyline processing

BR-RSKY-MR shows the worst performance since *BR-RSKY-MR* computes the reverse skyline in each orthant independently.

Even though our *RSKY-MR* also computes the reverse skyline in every orthant independently, *RSKY-MR* performs well due to the effective use of rsky-quadtrees. Furthermore, *RSKY-MR-M* shows better performance than *RSKY-MR-S* due to its parallelization of the third phase when the number of strong reverse skyline points is large. As we expected, the performance of *RSKY-MR-S* is better than that of *RSKY-MR-M* only for small correlated data sets. Since *RSKY-MR* selects *RSKY-MR-M* or *RSKY-MR-S* adaptively depending on the number of strong reverse skyline points, *RSKY-MR* always shows the best performance.

Varying t and d: In order to evaluate the speedup of our algorithms, we varied

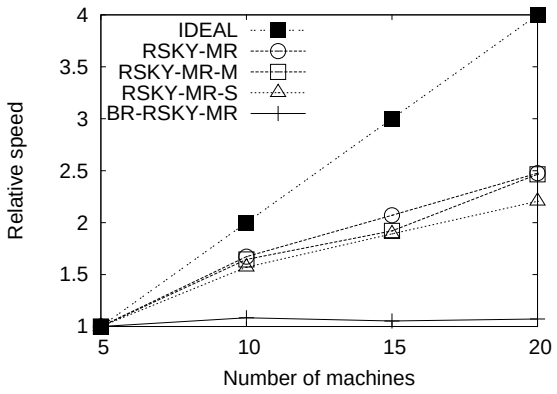


Figure 5.12: Relative Speed with varying the number of machines (t) for reverse skyline processing

the number of machines t from 5 to 20. In Figure 5.12, we show the relative speed of each algorithm. For the relative speed with varying t, when the number of strong reverse skyline points is small (i.e., the correlated data set), *RSKY-MR-S* has the better scalability than *RSKY-MR-M*. Over all cases, *RSKY-MR* shows the best scalability. Furthermore, the graphs varying d have almost the same trends with those with varying n in Figure 5.11. Thus, we do not provide the graphs for the experiments with varying d.

Chapter 6

Parallel Probabilistic Skyline Query Processing

6.1 Early Pruning Techniques

If we know that an object cannot be a probabilistic skyline object, we can avoid computing its skyline probability. Thus, we introduce three filtering techniques called *upper-bound filtering*, *zero-probability filtering* and *dominance-power filtering*. We next present the details of each filtering technique.

6.1.1 Upper-bound Filtering

The following propositions address that the skyline probability of an object U by considering a sample $\mathbb{S}$ of the objects in $\mathbb{D}$ only is an *upper bound* of $P_{sky}(U)$ for both discrete and continuous models.

Proposition 6.1.1 *Consider an object U in the discrete model. For an instance $u_i \in U$, the value of $P_{sky}(u_i)$ computed by Equation* (3.1) *with $V' \subseteq V$ and $\mathbb{S} \subset \mathbb{D}$ instead is the* upper bound *of $P_{sky}(u_i)$. The sum of the upper bounds of $P_{sky}(u_i)$s with all $u_i \in U$ is the* upper bound *of $P_{sky}(U)$.*

Proposition 6.1.2 *Consider an object U modeled by its uncertainty region $U.R$ with a pdf $U.f(\cdot)$. The value of $P_{sky}(U)$ computed by Equation* (3.2) *with $\mathbb{S} \subset \mathbb{D}$ and a*

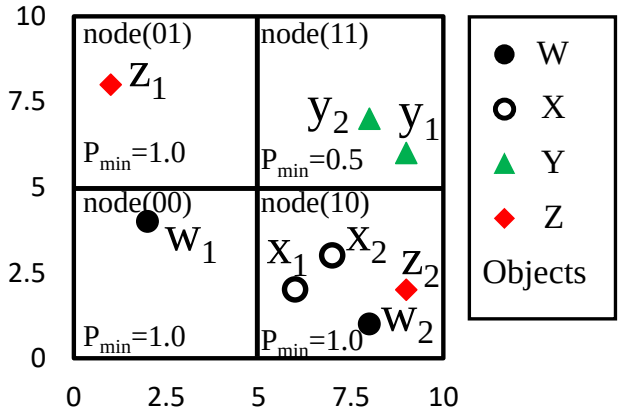


Figure 6.1: A PSQtree

sub-region $V.R'$ *of* $V.R$ *becomes the* upper bound *of* $P_{sky}(U)$.

By keeping the upper bounds of the skyline probabilities of all instances in each object, we can identify probabilistic non-skyline objects. As shown in Figure 6.1, all instances in $\langle[5, 10), [5, 10)\rangle$ are dominated by w_1. Note that $P_{sky}(y_1)$=$P(y_1)\prod_{V\in\mathbb{D},V\neq Y}$ (1-$\sum_{v_j\in V,v_j\prec y_1} P(v_j)$)=0.024 by Equation (3.1). Due to Proposition 6.1.1, we have $P_{sky}(y_1) \leq P(y_1)$(1-$P(w_1)$)=0.4 which is obtained by using $\mathbb{S} = \{W\}$ and $V' = \{w_1\}$. Similarly, the upper bound of $P_{sky}(y_2)$ becomes 0.1. Thus, the upper bound of $P_{sky}(Y)$ becomes 0.5 by adding the upper bounds of $P_{sky}(y_1)$ and $P_{sky}(y_2)$. If T_p is 0.6, since $P_{sky}(Y) \leq 0.5 < T_p$, Y is a non-skyline object.

We now present how to compute the upper bound of the skyline probability of every object in each partition for our upper-bound filtering. Let $R.min$ be the point whose k-th coordinate is the minimum in the k-th dimension for a rectangular region R.

Definition 6.1.3 *For an instance* u_i *of an object* $U \in \mathbb{D}$*, a set of objects* $\mathbb{S} \subset \mathbb{D}$ *and a rectangular region* $R(u_i)$ *including* u_i*, we define*

$$\beta(U, \mathbb{S}, R(u_i)) = \frac{\prod_{V \in \mathbb{S}} (1 - \sum_{v_j \in V, v_j \prec R(u_i).min} P(v_j))}{1 - \sum_{u_k \prec R(u_i).min, u_k \in U} P(u_k)} \tag{6.1}$$
$$\text{and } up(u_i, U, \mathbb{S}, R(u_i)) = P(u_i) \times \beta(U, \mathbb{S}, R(u_i)).$$

The upper bound of the skyline probability of an instance can be computed by utilizing the following lemma.

Lemma 6.1.4 *Consider an instance u_i of an object $U \in \mathbb{D}$ and a rectangular region $R(u_i)$ which contains u_i. For a set of objects $\mathbb{S} \subset \mathbb{D}$, we have $P_{sky}(u_i) \leq up(u_i, U, \mathbb{S}, R(u_i))$.*

Proof: Let D_{u_i} and D_R be the sets of the instances in $\mathbb{D}$ which dominate u_i and $R(u_i).min$, respectively. Since every instance dominating $R(u_i).min$ also dominates u_i, we have $D_R \subseteq D_{u_i}$. We derive $P_{sky}(u_i) \leq up(u_i, U, \mathbb{S}, R(u_i))$ as follows:

$$P_{sky}(u_i) \leq P(u_i) \prod_{V \in \mathbb{S}, V \neq U} \left(1 - \sum_{v_j \in V \cap D_R} P(v_j)\right) \text{ (by Proposition 6.1.1)}$$
$$= P(u_i) \prod_{V \in \mathbb{S}, V \neq U} \left(1 - \sum_{v_j \in V \cap D_R} P(v_j)\right) \frac{1 - \sum_{u_i \in U \cap D_R} P(u_i)}{1 - \sum_{u_i \in U \cap D_R} P(u_i)}$$
$$\leq P(u_i) \frac{\prod_{V \in \mathbb{S}} \left(1 - \sum_{v_j \in V \cap D_R} P(v_j)\right)}{1 - \sum_{u_k \in U \cap D_R} P(u_k)}$$
$$= P(u_i) \frac{\prod_{V \in \mathbb{S}} (1 - \sum_{v_j \in V, v_j \prec R(u_i).min} P(v_j))}{1 - \sum_{u_k \prec R(u_i).min, u_k \in U} P(u_k)} = up(u_i, U, \mathbb{S}, R(u_i))$$
(since $\{v_j \in V | v_j \prec R(u_i).min\} = V \cap D_R$) ∎

Corollary 6.1.5 *Consider an object $U \in \mathbb{D}$ and let $R(u_i)$ be a rectangular region containing an instance u_i of U. For a set of objects $\mathbb{S} \subset \mathbb{D}$, if we have $\sum_{u_i \in U} up(u_i, U, \mathbb{S}, R(u_i)) < T_p$, U is not a probabilistic skyline object.*

By Corollary 6.1.5, we do not compute the skyline probability of an uncertain object U if we have $\sum_{u_i \in U} up(u_i, U, \mathbb{S}, R(u_i)) < T_p$. We call such pruning the *upper-bound filtering*. We can prune even further when every object has a single instance only as follows.

Lemma 6.1.6 *When every object in $\mathbb{D}$ has a single instance, consider an instance u of an object U and a rectangular region $R(u)$ containing u. For a set of objects $\mathbb{S} \subset \mathbb{D}$, if we have $up(u, U, \mathbb{S}, R(u)) = P(u) \times \beta(U, \mathbb{S}, R(u)) < T_p$, U is not a skyline object. Furthermore, if $\beta(U, \mathbb{S}, R(u)) < T_p$ also holds, there is no object in the probabilistic skyline whose instance is dominated by u.*

Proof: Since every object has a single instance, we have $P_{sky}(U) = P_{sky}(u) \leq up(u, U, \mathbb{S}, R(u)) \leq T_p$ by Lemma 6.1.4 and U is not a skyline object due to Corollary 6.1.5.

We next prove the second case of when $\beta(U, \mathbb{S}, R(u)) < T_p$ by contradiction. Assume that there is a skyline object W whose instance w is dominated by u (i.e., $u \prec w$). By Proposition 6.1.1, we have $P_{sky}(W) \leq P(w) \prod_{V \in \mathbb{S}} (1 - \sum_{v \in V, v \prec w} P(v))$. Since $R(u)$ contains u, $R(u).min \prec u \prec w$ holds and hence $R(u).min \prec w$. Furthermore, because every instance v such that $v \prec R(u).min$ also dominates w, we have $\{v \in V | v \prec R(u).min\} \subseteq \{v \in V | v \prec w\}$ and get

$$P(w) \prod_{V \in \mathbb{S}} (1\text{-} \sum_{v \in V, v \prec w} P(v)) \leq P(w) \prod_{V \in \mathbb{S}} (1\text{-} \sum_{v \in V, v \prec R(u).min} P(v)).$$

When U has a single instance u which does not dominate $R(u).min$, we have $\sum_{u \prec R(u).min, u \in U} P(u) = 0$ resulting that

$$\beta(U, \mathbb{S}, R(u)) = \prod_{V \in \mathbb{S}} (1 - \sum_{v \in V, v \prec R(u).min} P(v))$$

from Definition 6.1.3. Thus, we have $P_{sky}(W) \leq P(w) \times \beta(U, \mathbb{S}, R(u))$. Now assume that $\beta(U, \mathbb{S}, R(u)) < T_p$ holds. Then, we obtain $P_{sky}(W) < T_p$. It contradicts to the assumption that W is a skyline object. ∎

The following corollary shows that the Lemma used by *PSMR* [18] is a special case of our Lemma 6.1.6 since $\beta(U, \mathbb{S}, R(u)) = \prod_{V \in \mathbb{S}}(1 - \sum_{v_j \in V, v_j \prec u} P(v_j))$ when $R(u)$ degenerates to the minimum bounding rectangle containing only a single instance u.

Corollary 6.1.7 *When every object in $\mathbb{D}$ has a single instance, consider an instance u of an object U and a subset $\mathbb{S} \subset \mathbb{D}$. If $P(u) \prod_{V \in \mathbb{S}}(1 - \sum_{v_j \in V, v_j \prec u} P(v_j)) < T_p$, U is not a skyline object. Furthermore, when $\beta(U, \mathbb{S}, R) = \prod_{V \in \mathbb{S}}(1 - \sum_{v_j \in V, v_j \prec u} P(v_j)) < T_p$ also holds, there is no object in the probabilistic skyline whose instance is dominated by u.*

The continuous model: We define $up_{pdf}(u, U, \mathbb{S}, R(u))$ by replacing the summations in Definition 6.1.3 with integrations over all points contained in $V.R$ for every object $V \in \mathbb{S}$.

Definition 6.1.8 *For an object $U \in \mathbb{D}$ with its uncertainty region $U.R$, a point u located in $U.R$, a subset $\mathbb{S} \subset \mathbb{D}$ and a rectangular region $R(u)$ which contains u, $up_{pdf}(u, U, \mathbb{S}, R(u))$ is defined as follows:*

$$up_{pdf}(u, U, \mathbb{S}, R(u)) = U.f(u) \frac{\prod_{V \in \mathbb{S}}(1\text{-} \int_{V.R} V.f(v) \mathbb{I}(v \prec R(u).min) dv)}{1\text{-} \int_{U.R} U.f(w) \mathbb{I}(w \prec R(u).min) dw}$$

where $R(u).min$ is the point whose k-th coordinate is the minimum in the k-th dimension for $R(u)$.

The following lemma states the condition of when an object is not a probabilistic skyline object. Since the proof is similar to that of Lemma 6.1.4, we omit it.

Lemma 6.1.9 *Consider the skyline threshold T_p, an object $U \in \mathbb{D}$ and a point u in $U.R$. Let $R(u)$ be a rectangular region containing u. For an object $U \in \mathbb{D}$ and a subset $\mathbb{S} \subset \mathbb{D}$, when $\int_{U.R} up_{pdf}(u, U, \mathbb{S}, R(u)) du < T_p$, U is not a skyline object.*

6.1.2 Zero-probability Filtering

Recall that the skyline probability of $u_i \in U$ is $P_{sky}(u_i) = P(u_i) \prod_{V \in \mathbb{D}, V \neq U}(1 - \sum_{v_j \in V, v_j \prec u_i} P(v_j))$. When $P_{sky}(u_i) = 0$, there exists an object V such that $\sum_{v_j \in V, v_j \prec u_i}$

$P(v_j) = 1$. Thus, an instance of V dominating u_i always appears in every possible world and u_i cannot contribute to computing the skyline probability of every other object.

Lemma 6.1.10 *Consider an instance u_i of an object $U \in \mathbb{D}$ and a rectangular region $R(u_i)$ containing u_i. For a subset $\mathbb{S} \subset \mathbb{D}$, when $\prod_{V \in \mathbb{S}}(1\text{-}\sum_{v_j \in V, v_j \prec R(u_i).min} P(v_j))=0$, the skyline probability of u_i is zero and we can delete u_i from U.*

Proof: By Lemma 6.1.4, we have $P_{sky}(u_i) \leq up(u_i, U, \mathbb{S}, R(u_i)) = P(u_i) \times \beta(U, \mathbb{S}, R(u_i))$. If $\prod_{V \in \mathbb{S}}(1 - \sum_{v_j \in V, v_j \prec R(u_i).min} P(v_j)) = 0$ (i.e., the numerator in Equation (6.1) of $\beta(U, \mathbb{S}, R(\ u_i))$ is zero), we have $0 \leq P_{sky}(u_i) \leq up(u_i, U, \mathbb{S}, R(u_i)) = 0$. ∎

We refer to the pruning technique based on Lemma 6.1.10 as the *zero-probability filtering*.

The continuous model: When $up_{pdf}(u, U, \mathbb{S}, R(u)) = 0$ holds for all $u \in U.R$, we have $\int_{U.R} up_{pdf}(u, U, \mathbb{S}, R(u))\mathrm{d}u = 0$ and U is not a skyline object by Lemma 6.1.9. Thus, we can delete U.

6.1.3 Dominance-Power Filtering

We maintain a small number of objects with the high dominating power and use them for checking the dominance relationship to handle large data.

Definition 6.1.11 *Consider a d-dimensional space $\langle[0, b(1)), \cdots, [0,\ b(d))\rangle$ where $[0,\ b(k))$ is its range of the k-th dimension. The* dominating power *of an instance v_j=$\langle v_j(1), \cdots, v_j(d)\rangle$, denoted by $DP(v_j)$, is $\prod_{k=1}^{d}(b(k) - v_j(k))$. Furthermore, the dominance power of an object V, denoted by $DP(V)$ is $\sum_{v_j \in V}(P(v_j) \times DP(v_j))$.*

As the existence probability of an instance v_j of an object V increases, the skyline probability of u_i of another object U dominated by v_j decreases. In addition, the number of instances of other objects dominated by v_j tends to be larger as the dominating power $DP(v_j)$ grows. Thus, we utilize $DP(V)$ to estimate the dominating power

of V. We refer to the set of top-K objects with the largest dominating powers as a *dominating object set* F.

For an object U with a dominating object set F, if we have $\sum_{u_i \in U} P(u_i) \prod_{V \in F, V \neq U} (1 - \sum_{v_j \in V, v_j \prec u_i} P(v_j)) < T_p$, U is not a probabilistic skyline object in $\mathbb{D}$ and thus we do not compute its skyline probability. We call the strategy the *dominance-power filtering*.

To maintain the K objects with the largest dominating powers and identify non-skyline objects at the same time, we invoke the procedure *DP-Filter* which utilizes a min-heap H to store the K dominating objects. For an object U, if the value of $P_{sky}(U)$ which is computed by considering H instead of $\mathbb{D}$ is less than T_p, *DP-Filter* returns FALSE to indicate that U is not a probabilistic skyline object due to Proposition 6.1.1. Otherwise, it returns TRUE. In this case, we also update H by inserting U. In other words, if the number of objects in H is less than K, we insert the object U into H. When the number of objects in H is K and the dominance power of U is larger than that of the object O with the minimum dominance power in H, we delete O from H and insert U to H.

The continuous model: Consider a d-dimensional space $\langle [0,b(1)), \cdots, [0,b(d)) \rangle$. The dominance power of an object U in the continuous model, represented by $DP_{pdf}(U)$, is defined as $\int_{U.R} U.f(u) \prod_{k=1}^{d} (b(k) - u(k)) \mathrm{d}u$. We keep top-$K$ objects with the highest dominating powers as the dominating object set F. The only change is to utilize Proposition 6.1.2 instead of Proposition 6.1.1. If $\int_{U.R} U.f(u) \prod_{V \in F, V \neq U} (1 - \int_{V.R} V.f(v) \mathbb{I}(v \prec u) \mathrm{d}v) \mathrm{d}u < T_p$ holds for an object U, *DP-Filter* returns FALSE. Otherwise, it returns TRUE and update H with U.

6.2 Utilization of a PS-QTREE for Pruning

To divide the data space into several sub-spaces, we develop a variant of *sky-quadtrees* in [40], called the *PS-QTREE*.

6.2.1 Generating a PS-QTREE

We recursively divide d-dimensional space into equi-sized 2^d sub-spaces, each of which is associated with a node in a *PS-QTREE*, until the number of points in each sub-space does not exceed the *split threshold*, denoted by ρ. We refer to the region represented by a node n as $n.region = \langle [n(1)^-, n(1)^+), \cdots, [n(d)^-, n(d)^+) \rangle$ where $[n(k)^-, n(k)^+)$ is the k-th dimensional range. We also define $n.min$ ($n.max$) as the $n.region$'s closest (farthest) corner of a leaf node n from the origin. Each node n is assigned with an id according to the method in [40] and the node with an id "i" is represented by $node(i)$. To build a *PS-QTREE* quickly, we utilize a random sample $\mathbb{S}$ of the objects in $\mathbb{D}$. Figure 6.1 shows an example of a *PS-QTREE* produced by the subset $\mathbb{S} = \{W, Z\}$ of $\mathbb{D}$ in Figure 3.4(a)(a).

6.2.2 Exploiting a PS-QTREE for Filtering

In this section, we show how the filtering techniques presented previously can be exploited by using a *PS-QTREE*.

Definition 6.2.1 *Consider a dataset $\mathbb{D}$, and a leaf node n of a PS-QTREE built by a sample $\mathbb{S} \subset \mathbb{D}$. We define $n.P_{min}(\mathbb{S}) = \prod_{V \in \mathbb{S}}(1 - \sum_{v_j \in V, v_j \prec n.min} P(v_j))$ for the discrete model and $n.P_{min}(\mathbb{S}) = \prod_{V \in \mathbb{S}}(1 - \int_{V.R} V.f(v)\mathbb{I}(v \prec n.min)dv)$ for the continuous model.*

By traversing the *PS-QTREE*, we set $n.P_{min}(\mathbb{S})$ in each leaf node n where $\mathbb{S}$ is the sample used to build the *PS-QTREE* and initially $n.P_{min}(\mathbb{S})$=1. In each leaf node n, we scan every object $V \in \mathbb{S}$ to check whether $n.min$ is dominated by an instance v_j of V and compute the sum of $P(v_j)$ of every instance v_j dominating $n.min$. We next update $n.P_{min}(\mathbb{S})$ by multiplying $(1 - \sum_{v_j \in V, v_j \prec n.min} P(v_j))$ to itself according to Definition 6.2.1. For the continuous model, we generate the points in $V.R$ for each object $V \in \mathbb{S}$ by following $V.f(\cdot)$ and build a *PS-QTREE* by using the generated points.

Upper-bound filtering: We can utilize $n.P_{min}(\mathbb{S})$ for the upper-bound filtering due to the following corollary. The proof of the corollary is analogous to that of Lemmas 6.1.4 and 6.1.9 by letting $R(u_i).min = n(u_i).min$.

Corollary 6.2.2 *For a PS-QTREE T generated by a sample $\mathbb{S} \subset \mathbb{D}$ and an instance u_i of an object U, let $n(u_i)$ be the leaf node of T whose region contains u_i. Depending on an uncertainty model, the skyline probability of U (i.e., $P_{sky}(U)$) is upper bounded by $up_T(U, \mathbb{S})$ where*

$$up_T(U,\mathbb{S}) = \begin{cases} \sum_{u_i \in U} \frac{P(u_i) \times n(u_i).P_{min}(\mathbb{S})}{1 - \sum_{u_k \prec n(u_i).min, u_k \in U} P(u_k)} \\ \\ \int_{U.R} \frac{U.f(u_i) \times n(u_i).P_{min}(\mathbb{S})}{1 - \int_{U.R} U.f(w) \mathbb{I}(w \prec n(u_i).min) dw} \mathrm{d}u_i. \end{cases}$$

Zero-probability filtering: We also use $n.P_{min}(\mathbb{S})$ for the zero-probability filtering by the following corollary whose proof is similar to that of Lemma 6.1.10.

Corollary 6.2.3 *For a leaf node n of a PS-QTREE built by a sample $\mathbb{S} \subset \mathbb{D}$, when $n.P_{min}(\mathbb{S})=0$, the skyline probability of every instance in the $n.region$ is zero and thus we can delete the instances of all objects in the $n.region$ from $\mathbb{D}$.*

To build a *PS-QTREE*, the procedure *GenQtree* is called with a sample $\mathbb{S}$ of the objects in $\mathbb{D}$. We omit the pseudocode of *GenQtree* since it is straightforward.

6.2.3 Partitioning Objects by a PS-QTREE

For an object $U \in \mathbb{D}$, if we distribute its instances to several partitions, we need an additional aggregation phase to compute the skyline probability of U by summing the skyline probabilities of its instances in multiple partitions. To guarantee that the skyline probability of each object can be computed without an extra MapReduce phase, we allocate all instances of each object U to a single partition by utilizing $U.max$ defined as follows.

Definition 6.2.4 *For the discrete model, the* max *and* min *points of an object U, represented by $U.max$ and $U.min$, are defined as $U.max(k) = max_{u_i \in U} u_i(k)$ and $U.min(k) = min_{u_i \in U} u_i(k)$, respectively, for $k = 1, \ldots, d$. For the continuous model, where U is modeled by an uncertainty region $U.R$ with pdf, $U.max$ ($U.min$) is the farthest (closest) corner point in $U.R$ from the origin.*

Let $\mathcal{M}(\mathbb{D}, n_\ell)$ be the set of objects whose max points belong to a leaf node n_ℓ of a *PS-QTREE*. We need to identify all the other objects required to compute the skyline probability of every object $U \in \mathcal{M}(\mathbb{D}, n_\ell)$. To do this efficiently, we use the dominance relationship between a pair of leaf nodes.

Definition 6.2.5 *For a pair of nodes n_1 and n_2 in a PS-QTREE, if $n_1.min(k) < n_2.max(k)$ for $k = 1, \cdots, d$, we say n_1* weakly dominates *n_2 and represent it by $n_1 \preceq n_2$.*

Consider the *PS-QTREE* in Figure 6.1. The min point of $node(00)$ (i.e., $node(00).min$) is $\langle 0, 0 \rangle$ and the max point of $node(11)$ (i.e., $node(11).max$) is $\langle 100, 100 \rangle$. Since $node(00).min(1) < node(11).max(1)$ and $node(00).min(2) < node(11).max(2)$, $node(00) \preceq node(11)$. We also have $node(00).min(k) < node(01).max(k)$ for every k and $node(00) \preceq node(01)$. However, since $node(01).min(2) \geq node(10).max(2)$, $node(01)$ does not weakly dominate $node(10)$.

For each leaf node n_ℓ, Lemma 6.2.6 shows that the exact skyline probabilities of the objects in $\mathcal{M}(\mathbb{D}, n_\ell)$ can be computed by considering only the instances located in the region of every leaf node which weakly dominates n_ℓ in both discrete and continuous models.

Lemma 6.2.6 *Consider a dataset $\mathbb{D}$, a leaf node n_ℓ of a PS-QTREE and an object $U \in \mathcal{M}(\mathbb{D}, n_\ell)$. In the discrete model, for each instance u_i of U, if an instance v_j of another object $V \in \mathbb{D}$ is contained in the region of a leaf node n such that $n \not\preceq n_\ell$, v_j does not dominate u_i. In the continuous model, if $V.R.min$ does not dominate $n_\ell.max$ for another object $V \in \mathbb{D}$, V does not affect the skyline probability of U.*

Proof: Since the object U is in $\mathcal{M}(\mathbb{D}, n_\ell)$, $U.max$ is contained in $n_\ell.region$. Consider the discrete model first. For an instance $u_i \in U$, $u_i(k) \leq U.max(k) < n_\ell.max(k)$ holds for $k = 1, \cdots, d$. Since $n \not\preceq n_\ell$, there exists a value k such that $n_\ell.max(k) \leq n.min(k)$. Because v_j is contained in $n.region$, we have $n.min(k) \leq v_j$. Thus, we have $u_i(k) < n_\ell.max(k) \leq n.min(k) \leq v_j$ and v_j does not dominate u_i. Similarly, we can prove the case of the continuous model. ∎

According to Lemma 6.2.6, we define the set of instances of an object $V \notin \mathcal{M}(\mathbb{D}, n_\ell)$ required to compute the skyline probability of every object U in $\mathcal{M}(\mathbb{D}, n_\ell)$.

Definition 6.2.7 *For a leaf node n_ℓ, let $\mathcal{I}_w(\mathbb{D}, n_\ell)$ be all instances of an object in $\mathbb{D} - \mathcal{M}(\mathbb{D}, n_\ell)$ which are in a leaf node n satisfying $n \preceq n_\ell$. In other words, $\mathcal{I}_w(\mathbb{D}, n_\ell) = \{v_j \in V | V \notin \mathcal{M}(\mathbb{D}, n_\ell) \wedge n(v_j) \preceq n_\ell\}$.*

Consider the dataset $\mathbb{D}$ in Figure 3.4(a) and the *PS-QTREE* in Figure 6.1. $\mathcal{I}_w(\mathbb{D}, node(10))$ is $\{w_1, w_2, z_2\}$ since $node(00)$ and $node(10)$ weakly dominate $node(10)$ as well as $\mathcal{M}(\mathbb{D}, node(10))$=$\{X\}$.

6.3 PS-QPF-MR: Our Algorithm with Quadtree Partitiong and Filtering

In this section, we develop the algorithms with a single MapReduce phase by distributing the objects based on the space split by a *PS-QTREE*.

We first present the MapReduce algorithm *PS-QP-MR* (Probabilistic Skyline algorithm by Quadtree Partitioning) which utilizes a *PS-QTREE*. Then, we provide the MapReduce algorithm *PS-QPF-MR* which enhances *PS-QP-MR* by applying the filtering techniques described in Section 6.1.

PS-QP-MR: We build a *PS-QTREE* with a sample $\mathbb{S}$ of data $\mathbb{D}$ in a single machine by calling *GenQtree* introduced in Section 6.2. We next split $\mathbb{D}$ using MapReduce into partitions each of which corresponds to a leaf node n_ℓ of the *PS-QTREE* and contains

Function PS-QPF-MR($\mathbb{D}$, T_p, ρ)

$\mathbb{D}$: uncertain dataset, T_p: probability threshold, ρ: split threshold

begin

1. $\mathbb{S}$ = Sample($\mathbb{D}$);
2. *PSQtree* = GenQtree($\mathbb{S}$, ρ);
3. Broadcast *PSQtree*; Broadcast T_p;
4. pSL = RunMapReduce(PS-QPFC-MR, $\mathbb{D}$);
5. **return** pSL;

end

Figure 6.2: The PS-QPF-MR algorithm

the objects in $\mathcal{M}(\mathbb{D}, n_\ell)$ as well as the instances in $\mathcal{I}_w(\mathbb{D}, n_\ell)$ (by Definition 6.2.7). We then compute the skyline probability of each object U in $\mathcal{M}(\mathbb{D}, n_\ell)$ and output U if U is a probabilistic skyline object.

PS-QPF-MR: The only difference of *PS-QPF-MR* from *PS-QP-MR* is to check whether each object U is a skyline candidate object or not by using the three filtering techniques and to compute the skyline probabilities of only skyline candidate objects. We present the pseudocode of *PS-QPF-MR* in Figure 6.2.

Setup function: Before map functions are called, the setup function of each mapper task initializes a min-heap H and loads a *PS-QTREE* to share them across the map functions. The min-heap H maintains the dominating object set F for the dominance-power filtering introduced in Section 6.1.3.

Map function: The pseudocode of the map function is shown in Figure 6.3. The map function invoked with an object U loads the probability threshold T_p (line 1 of PS-QPFC-MR.map). We apply the zero-probability, upper-bound and dominance-power filtering techniques by invoking *ZeroProb*, *UpperBound* and *DP-Filter*, respectively (lines 2-6). We refer to U' as the object after pruning U's instances by *ZeroProb*. If the upper bound of the $P_{sky}(U')$ computed by *UpperBound* is at least T_p, *DP-Filter* is

Function PS-QPFC-MR.map(U)

U: an uncertain object

begin

1. T_p = LoadThreshold();
2. U' = ZeroProb(U, $PSQtree$);
3. $upper$ = UpperBound(U', $PSQtree$);
4. $cand$ = **FALSE**;
5. **if** $upper \geq T_p$ **then**
6. $cand$ = DP-Filter(U', T_p, H);
7. **if** $cand$ **then** emit($n(U'.max)$, (U', 'C'));
8. **for** each leaf node n_ℓ in $PSQtree$ **do**
9. **if** $cand =$ **True** and $n_\ell = n(U'.max)$ **then** continue;
10. I = NewList();
11. **for** each u_i in U' **do**
12. **if** $n(u_i) \preceq n_\ell$ **then**
13. I.add(u_i));
14. emit(n_ℓ, (I, 'W', $cand$))

end

Figure 6.3: The map function of the PS-QPFC-MR algorithm

invoked to check whether U' is a candidate object or not. If U' is a candidate object (i.e., $cand$ =TRUE), the map function emits the key-value pair $\langle n(U'.max)$, (U', 'C')$\rangle$ where $n(U'.max)$ is the leaf node containing $U'.max$ and 'C' represents that U' is a skyline candidate contained in $\mathcal{M}(\mathbb{D}, n(U'.max))$ (line 7).

For each leaf node n_ℓ, we emit each instance u_i of U' which is required to compute the exact skyline probabilities of objects in $\mathcal{M}(\mathbb{D}, n_\ell)$ (i.e., $u_i \in \mathcal{I}(\mathbb{D}, n_\ell)$) (lines 8-14). For an instance $u_i \in U'$, if $n(u_i) \not\preceq n_\ell$, u_i does not dominate the instances of the objects in $\mathcal{M}(\mathbb{D}, n_\ell)$ by Lemma 6.2.6. Thus, if $n(u_i) \preceq n_\ell$, the map function puts u_i into the list I. After every instance of U' is evaluated for n_ℓ, the map function outputs

Function PS-QPFC-MR.reduce(n_ℓ, L)

begin

1. (L_C, L_W^T, L_W^F) = SplitList(L);
2. T_p = LoadThreshold();
3. **for** each object U in L_C **do**
4. $\quad skyline\_prob$ = SkylineProb(U, L_C, L_W^T, L_W^F);
5. $\quad$ **if** $skyline\_prob \geq T_p$ **then**
6. $\quad\quad$ emit(U, $skyline\_prob$);

end

Figure 6.4: The reduce function of the PS-QPFC-MR algorithm

the key-value pair $\langle n_\ell, (I,$ 'W', $cand)\rangle$ where 'W' denotes that the instances are in $\mathcal{I}(\mathbb{D}, n_\ell)$ and $cand$ represents that U' is a candidate object or not (line 14). Note that when U' is a candidate object and $n_\ell = n(U'.max)$, we do nothing (line 9) since it is already sent in line 7.

Reduce function: In the shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct leaf node, and a reduce function is called with each node n_ℓ and a value list L. The pseudocode of the reduce function is presented in Figure 6.4. The value list L is split into L_C, L_W^T and L_W^F where L_C is $\mathcal{M}(\mathbb{D}, n_\ell)$, L_W^T is the subset of $\mathcal{I}_w(\mathbb{D}, n_\ell)$ whose instances are marked with $cand$ = TRUE, and L_W^F is $\mathcal{I}_w(\mathbb{D}, n_\ell) - L_W^T$ (line 1 of PS-QPFC-MR.reduce).

To split L into three partitions L_C, L_W^T and L_W^F effectively, we exploit the functionality of *secondary sorting* [36] provided by the MapReduce framework which arranges the elements in L with a specific ordering such that all elements belonging to L_C always appear first, all elements belonging to L_W^T are located next and the elements belonging to L_W^F are placed last.

Once all elements in L_C are loaded into main memory, the reduce function computes the skyline probability of every object U in L_C by invoking *SkylineProb* (lines

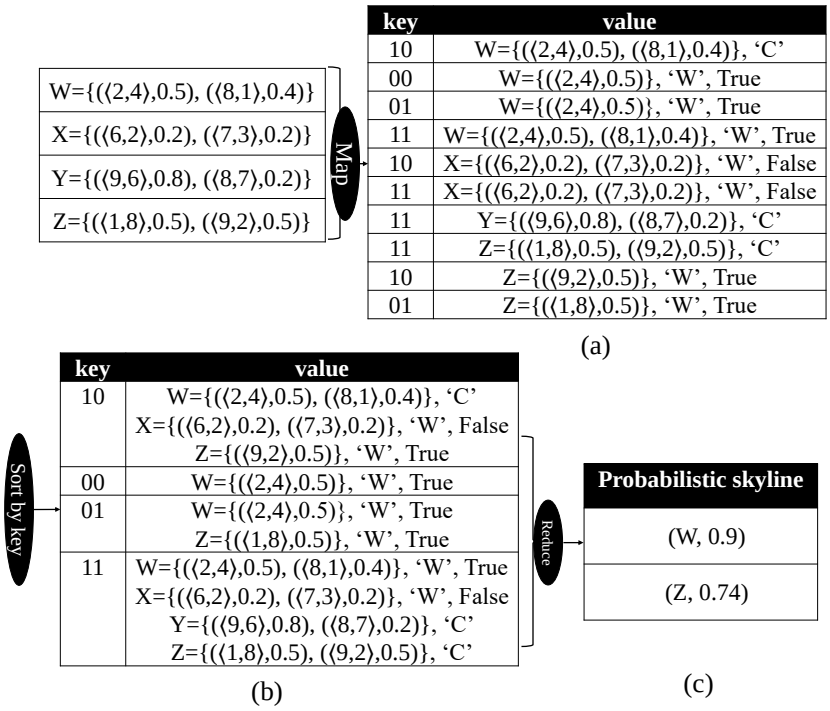


Figure 6.5: The steps of *PS-QPFC-MR*

2-4). Since we keep only the elements of L_C in main memory, we require $O(|L_C|) = O(|\mathcal{M}(\mathbb{D}, n_\ell)|)$ memory.

To discover non-skyline objects earlier, we first compute $P_{sky}(U)$ with other objects in L_C since L_C is already in main memory. Then, $P_{sky}(U)$ is updated with L_W^T and next updated with L_W^F. The reason why L_W^F is read in last is that all instances in L_W^F tend to have less dominance power than the instances in L_W^T since they belong to non-skyline objects (i.e., $cand$ = FALSE).

Let $\mathbb{O}$ be the set of objects whose instances were used to compute $P_{sky}(U)$ up to now. Note that, by Proposition 6.1.1 with $\mathbb{S} = \mathbb{O}$, the skyline probability of U computed by using $\mathbb{O}$ becomes an upper bound of $P_{sky}(U)$. Thus, whenever the skyline probability of U updated currently is less than T_p, *SkylineProb* returns zero to indicate that U is a non-skyline object. Otherwise, we output U with $P_{sky}(U)$ (lines 5-6).

Example 6.3.1 *Consider the data* $\mathbb{D}$ *and the* PS-QTREE *in Figure* 3.4(a) *with the probability threshold* T_p=0.5. *Figures* 6.5(a)-(d) *show the data flow in* PS-QPF-MR. *After the* PS-QTREE *is broadcast to all map functions, each map function is called with an uncertain object as illustrated in Figure* 6.5(a). *Consider the map function called with* X. *Since the upper bound of the skyline probability of* X *is* $node(10).P_{min} \cdot P(x_1) + node(10).P_{min} \cdot P(x_2) = 0.4 < T_p = 0.5$, X *is not a skyline candidate object (due to Corollary 6.1.5). Note that every instance of* X *is contained in the region of* $node(10)$. *The map function emits the key-value pairs* $\langle 10, (\{(\langle 6, 2\rangle, 0.2), (\langle 7, 3\rangle, 0.2)\}$,*"W",* $False)\rangle$ *and* $\langle 11, (\{(\langle 6, 2\rangle, 0.2), (\langle 7, 3\rangle, 0.2)\}$,*"W",* $False)\rangle$ *since* $node(10)$ *weakly dominates* $node(10)$ *itself and* $node(11)$. *Figure* 6.5(a) *shows the key-value pairs emitted by all map functions. The key-value pairs grouped by each distinct key are provided in Figure* 6.5(b). *As shown in Figure* 6.5(c), *the probabilistic skyline objects* W *and* Z *are output by the reduce functions called with* $node(10)$ *and* $node(11)$, *respectively.* ∎

The continuous model: We utilize the Monte Carlo integration [11] to calculate the skyline probabilities of objects. We sample points u from $U.R$ uniformly and $P_{sky}(U)$ in Equation (3.2) is calculated as the average value of $|U.R| \times U.f(u) \prod_{V \in \mathbb{D}, V \neq U} P_{LS}(u, V)$ where $P_{LS}(u, V) = 1 - \int_{V.R} V.f(v)\mathbb{I}(v \prec u)\mathrm{d}v$. The integral to calculate $P_{LS}(u, V)$ is also computed by the Monte Carlo integration.

The pseudocode of *PS-QPFC-MR* is the same as that of *PS-QPFC-MR* for the discrete model except that it utilizes the filtering techniques for the continuous model and the lines 10-14 of the map function in Figure 6.3 are replaced by the lines below. Due to Lemma 6.2.6, when $U.R.min \prec n_\ell.max$ holds, we send U to the reduce function of n_ℓ.

10. **if** $U.R.min \prec n_\ell.max$ **then**
11. emit(n_ℓ, (U, 'W', $cand$));

6.3.1 Optimizations of PS-QPF-MR

When a map function is invoked with an object V, each instance $v_j \in V$ is transmitted to the reduce function corresponding to every leaf node n_ℓ dominated weakly by $n(v_j)$ (i.e., the leaf node whose region includes v_j). To minimize the number of transmissions by all map functions, we can actually cluster the leaf nodes of a *PS-QTREE* into several groups such that a single reduce function processes all leaf nodes of each group with the main memory available in each machine.

When we cluster the leaf nodes, we should balance workloads for all reduce functions too. Let a group G_i be a set of leaf nodes $\{n_{i_1}, \ldots, n_{i_{|G_i|}}\}$. The input of a reduce function with a group G_i consists of the objects whose max points are in the region of a leaf node $n_{i_k} \in G_i$ and the instances v_j such that $n(v_j)$ weakly dominates a leaf node $n_{i_k} \in G_i$. Thus, we estimate the number of the objects as well as the number of the instances in each group by utilizing the sample used to build a *PS-QTREE* and force the input size of every reduce function to be similar for workload balancing.

Reducing Network Overhead by Clustering

Let $\mathcal{G}$ be a set of groups $\{G_1, \cdots, G_{|\mathcal{G}|}\}$ where G_i is a group of leaf nodes $\{n_{i_1}, \ldots, n_{i_{|G_i|}}\}$. Then, let $\mathcal{M}(\mathbb{D}, G_i) = \bigcup_{n_{i_k} \in G_i} \mathcal{M}(\mathbb{D}, n_{i_k})$ and $\mathcal{I}_w(\mathbb{D}, G_i) = \bigcup_{n_{i_k} \in G_i} \mathcal{I}_w(\mathbb{D}, n_{i_k})$. The reduce function called for a group G_i computes the skyline probability of every object in $\mathcal{M}(\mathbb{D}, G_i)$ by using the other objects in $\mathcal{M}(\mathbb{D}, G_i)$ and all instances in $\mathcal{I}_w(\mathbb{D}, G_i)$.

As mentioned in Section 6.3, the reduce function called for each leaf node n_ℓ requires $O(|\mathcal{M}(\mathbb{D}, n_\ell)|)$ memory only since we utilize the secondary sorting. Let the size of main memory be $s(mem)$ and the average size of an object be $s(obj)$. When we group leaf nodes, since each reduce function for a group G_i requires $O(|\mathcal{M}(\mathbb{D}, G_i)|)$ memory, we should have $|\mathcal{M}(\mathbb{D}, G_i)| \cdot s(obj) \leq s(mem)$ so that $\mathcal{M}(\mathbb{D}, G_i)$ can be kept in the main memory. In addition, since the number of transmissions by all map functions is $\sum_{G_i \in \mathcal{G}} (|\mathcal{I}_w(\mathbb{D}, G_i)| + |\mathcal{M}(\mathbb{D}, G_i)|)$ and $\sum_{G_i \in \mathcal{G}} |\mathcal{M}(\mathbb{D}, G_i)|$ is a constant

regardless of leaf node grouping, we should minimize $\sum_{G_i \in \mathcal{G}} |\mathcal{I}_w(\mathbb{D}, G_i)|$ to reduce the number of transmissions. Therefore, our leaf node grouping problem can be formulated as follows:

Definition 6.3.2 [Leaf node grouping problem] *Let the average size of an object be $s(obj)$, the size of main memory assigned to each reduce function be $s(mem)$ and N=$\{n_1, \cdots, n_{|N|}\}$ be the set of all leaf nodes in a* PS-QTREE. *Assume $|\mathcal{M}(\mathbb{D}, n_\ell)| \cdot s(obj) \leq s(mem)$ for every $n_\ell \in N$. The problem is to find a set of disjoint groups $\mathcal{G}$=$\{G_1, \cdots, G_{|\mathcal{G}|}\}$ such that $G_1 \cup \cdots \cup G_{|\mathcal{G}|} = N$, $|\mathcal{M}(\mathbb{D}, G_i)| \cdot s(obj) \leq s(mem)$ for all i=$1, \ldots, |\mathcal{G}|$ and $\sum_{G_i} |\mathcal{I}_w(\mathbb{D}, G_i)|$ is minimized.*

Since this problem can be reduced from the well-known NP-Complete *bin packing problem* [50] by setting $|\mathcal{I}_w(\mathbb{D}, G_i)|$ =1 for every group G_i, it is NP-Complete and thus we devise a greedy algorithm. Let $\tilde{\mathcal{G}}$ be the set of groups created so far in our algorithm. It takes each leaf node n_ℓ of a *PS-QTREE* one by one and inserts n_ℓ into the group $G_i \in \tilde{\mathcal{G}}$ which can accommodate n_ℓ (i.e., $|\mathcal{M}(\mathbb{D}, G_i \cup \{n_\ell\})| \cdot s(obj) \leq s(mem)$) with the minimum of ($|\mathcal{I}_w(\mathbb{D}, G_i \cup \{n_\ell\})| - |\mathcal{I}_w(\mathbb{D}, G_i)|$). If there is no group to accommodate n_ℓ, we create an empty group G_j, put n_ℓ into G_j and insert G_j into $\tilde{\mathcal{G}}$.

To apply our heuristics, we need $|\mathcal{M}(\mathbb{D}, G_i)|$ and $|\mathcal{I}_w(\mathbb{D}, G_i)|$. With the sample $\mathbb{S}$ to build the *PS-QTREE*, by assuming that $|\mathcal{M}(\mathbb{D}, G_i)|$ and $|\mathcal{I}_w(\mathbb{D}, G_i)|$ are proportional to $|\mathcal{M}(\mathbb{S}, G_i)|$ and $|\mathcal{I}_w(\mathbb{S}, G_i)|$ respectively, we estimate them as $|\hat{\mathcal{M}}(\mathbb{D}, G_i)|$ $= |\mathcal{M}(\mathbb{S}, G_i)| \cdot \frac{|\mathbb{D}|}{|\mathbb{S}|}$ and $|\hat{\mathcal{I}}_w(\mathbb{D}, G_i)| = |\mathcal{I}_w(\mathbb{S}, G_i)| \cdot \frac{|\mathbb{D}|}{|\mathbb{S}|}$.

Workload Balancing of Reduce Functions

After applying leaf node grouping, $|\mathcal{M}(\mathbb{D}, G_i)|$ of every group $G_i \in \mathcal{G}$ becomes similar and the sum of $|\mathcal{I}_w(\mathbb{D}, G_i)|$s over all groups $G_i \in \mathcal{G}$ is minimized. However, since the sizes of $\mathcal{I}_w(\mathbb{D}, G_i)$s may be skewed, the execution times of reduce functions can be quite different. Let the input of the reduce function for a group G_i be $X(G_i)$ which actually consists of $\mathcal{M}(\mathbb{D}, G_i)$ and $\mathcal{I}_w(\mathbb{D}, G_i)$. We balance the workloads of

reduce functions for the groups G_i with large $|\mathcal{I}_w(\mathbb{D}, G_i)|$ by splitting $\mathcal{I}_w(\mathbb{D}, G_i)$ into m_{G_i} disjoint partitions $\{\mathcal{I}_w(\mathbb{D}, G_i, 1), \cdots, \mathcal{I}_w(\mathbb{D}, G_i, m_{G_i})\}$ such that every instance of the each object is in the same partition. With respect to $X(G_i)$, we next generate a set $\mathcal{X}(G_i)$=$\{X_1(G_i), \cdots, X_{m_{G_i}}(G_i)\}$ where $X_k(G_i)$ is composed of $\mathcal{M}(\mathbb{D}, G_i)$ and a partition $\mathcal{I}_w(\mathbb{D}, G_i, k)$, and invoke a reduce function with $X_k(G_i)$ to calculate partial skyline probability of each instance u of an object U in $\mathcal{M}(\mathbb{D}, G_i)$. Then, the skyline probability of U is computed in the main function by collecting all partial skyline probabilities of every instance u∈U.

The skyline probability of each instance u of every object $U \in \mathcal{M}(\mathbb{D}, G_i)$ can be computed by using the reduce functions each of whose input is $X_k(G_i) \in \mathcal{X}(G_i)$. Given a set of partitions $I(G_i) = \{\mathcal{I}_w(\mathbb{D}, G_i, 1), \cdots, \mathcal{I}_w(\mathbb{D}, G_i, m_{G_i})\}$ of $\mathcal{I}_w(\mathbb{D}, G_i)$, let $S(\mathbb{D}, G_i, k)$ be the set of objects whose instances are contained in the k-th partition $\mathcal{I}_w(\mathbb{D}, G_i, k) \in I(G_i)$ and $P(\mathbb{D}, G_i, k)$ be the probability that every instance v_j of an object V in $S(\mathbb{D}, G_i, k)$ which dominates u does not exist in a possible world (i.e., $P(\mathbb{D}, G_i, k) = \prod_{V \in S(\mathbb{D}, G_i, k)}(1 - \sum_{v_j \in V, v_j \prec u} P(v_j))$). For the instance u, since every object V such that there exists an instance $v_j \in V$ dominating u is contained in one of $\mathcal{M}(\mathbb{D}, G_i)$, $S(\mathbb{D}, G_i, 1), \cdots, S(\mathbb{D}, G_i, m_{G_i}-1)$ and $S(\mathbb{D}, G_i, m_{G_i})$, the skyline probability of u can be computed as follows:

$$P_{sky}(u) = P(u) \times \prod_{V \in \mathbb{D}, V \neq U} (1 - \sum_{v_j \in V, v_j \prec u} P(v_j))$$

$$= P(u) \times \prod_{V \in \mathcal{M}(\mathbb{D}, G_i), V \neq U} (1 - \sum_{v_j \in V, v_j \prec u} P(v_j)) \times \prod_{k=1}^{m_{G_i}} P(\mathbb{D}, G_i, k)$$

While the reduce function invoked with $X_k(G_i)$ computes $P(\mathbb{D}, G_i, k)$ using $\mathcal{I}_w(\mathbb{D}, G_i, k)$, one of the reduce functions calculates $P(u) \times \prod_{V \in \mathcal{M}(\mathbb{D}, G_i), V \neq U}(1 - \sum_{v_j \in V, v_j \prec u} P(v_j))$. Then, we can compute the skyline probability of u by using the above equation.

After leaf node grouping, the number of reduce functions processed by each ma-

chine is either $\lfloor |\mathcal{G}|/t \rfloor$ or $\lfloor |\mathcal{G}|/t \rfloor + 1$ where t is the number of machines. Thus, we set the number of reduce function calls to $\lceil \frac{|\mathcal{G}|}{t} \rceil \cdot t$ which is at least $|\mathcal{G}|$ and the smallest multiple of t so that each machine processes the same number (i.e., $\lceil |\mathcal{G}|/t \rceil$) of reduce functions. To do this, our workload balancing problem is defined as follows:

Definition 6.3.3 [Workload balancing problem] *Given a set of groups* $\mathcal{G} = \{G_1, \cdots, G_{|\mathcal{G}|}\}$ *which is the result of the leaf node grouping problem and a number of machines* t*, the problem is to find* $\mathcal{X}(G_i) = \{X_1(G_i), \cdots, X_{m_{G_i}}(G_i)\}$ *such that* (1) $\mathcal{I}_w(\mathbb{D}, G_i)$ *is split into disjoint partitions* $I(G_i) = \{\mathcal{I}_w(\mathbb{D}, G_i, 1), \cdots, \mathcal{I}_w(\mathbb{D}, G_i, m_{G_i})\}$ *for each group* $G_i \in \mathcal{G}$*,* (2) $X_k(G_i)$ *is composed of* $\mathcal{M}(\mathbb{D}, G_i)$ *as well as* $\mathcal{I}_w(\mathbb{D}, G_i, k) \in I(G_i)$*,* (3) $\lceil \frac{|\mathcal{G}|}{t} \rceil \cdot t = \sum_{G_i} m_{G_i}$ *and* (4) $\max_{G_i \in \mathcal{G}, X_k(G_i) \in \mathcal{X}(G_i)} |\mathcal{I}_w(\mathbb{D}, G_i, k)|$ *is minimized. Note that* $\sum_{G_i} m_{G_i}$ *is the total number of reduce functions utilized by all groups.*

We next present the greedy algorithm *GreedyWorkload* for the workload balancing problem. Let $\tilde{m}_{G_i}$ be the number of partitions in $\mathcal{X}(G_i)$ for each group G_i. Initially, $\tilde{m}_{G_i}$=1. At each step of *GreedyWorkload*, we repeatedly select the group G_i with the maximum $|\mathcal{I}_w(\mathbb{D}, G_i)|/\tilde{m}_{G_i}$ and increase $\tilde{m}_{G_i}$ by one until $\sum_{G_i} \tilde{m}_{G_i} = \lceil \frac{|\mathcal{G}|}{t} \rceil \cdot t$. As we did in leaf node grouping previously, we estimate $|\mathcal{I}_w(\mathbb{D}, G_i)|$ by utilizing a sample $\mathbb{S}$ of the objects in $\mathbb{D}$.

After *GreedyWorkload* terminates, for every group G_i, we split $\mathcal{I}_w(\mathbb{D}, G_i)$ into $\{\mathcal{I}_w(\mathbb{D}, G_i, 1), \cdots, \mathcal{I}_w(\mathbb{D}, G_i, \tilde{m}_{G_i})\}$. We broadcast $\mathcal{G}$ and $\tilde{m}_{G_i}$ of every $G_i \in \mathcal{G}$ to all map functions. To make the size of every partition similar, when a map function is called with an object whose instances belong to $\mathcal{I}_w(\mathbb{D}, G_i)$, the map function chooses a random number k between 1 and $\tilde{m}_{G_i}$ and sends the instances to the reduce function handling $X_k(G_i)$.

Lemma 6.3.4 *When* $\mathcal{I}_w(\mathbb{D}, G_i)$ *can be split into equi-sized partitions for every* $G_i \in \mathcal{G}$*, the procedure* GreedyWorkload *finds an optimal solution for the workload balancing problem.*

Proof: Due to the space limitation, we omit the proof. ∎

Since we may not split $\mathcal{I}_w(\mathbb{D}, G_i)$ into equi-sized partitions such that every instance of each object lies in the same partition, *GreedyWorkload* does not guarantee the optimality.

6.3.2 Sample Size and Split Threshold of a PSQtree

In leaf node grouping, although we require $s(obj) \cdot |\mathcal{M}(\mathbb{D}, G_i)| \leq s(mem)$ for every group G_i, the reduce function handling G_i may suffer from the lack of memory space since we estimate $|\mathcal{M}(\mathbb{D}, G_i)|$ approximately by using a sample $\mathbb{S}$ of $\mathbb{D}$. To avoid such deficiency, we enforce $s(obj) \cdot |\hat{\mathcal{M}}(\mathbb{D}, G_i)| \leq \alpha \cdot s(mem)$ (e.g., α=0.8) where $|\hat{\mathcal{M}}(\mathbb{D}, G_i)|$ is the estimate of $|\mathcal{M}(\mathbb{D}, G_i)|$. We refer to it as the *memory utilization heuristics*.

Finding a proper sample size: We study how to choose the sample size to estimate $|\mathcal{M}(\mathbb{D}, G_i)|$ accurately. When $s(obj) \cdot |\hat{\mathcal{M}}(\mathbb{D}, G_i)| < \alpha \cdot s(mem)$ but $s(mem) < s(obj) \cdot |\mathcal{M}(\mathbb{D}, G_i)|$, it is problematic. Thus, we want the probability that $|\hat{\mathcal{M}}(\mathbb{D}, G_i)| < \alpha \cdot |\mathcal{M}(\mathbb{D}, G_i)|$ is less than a threshold δ.

Lemma 6.3.5 *Given a group G_i, a threshold δ and a sample $\mathbb{S} \subset \mathbb{D}$, if*

$$|\mathbb{S}| \geq \frac{-2 \cdot |\mathbb{D}| \cdot \ln \delta}{(1-\alpha)^2 \cdot |\mathcal{M}(\mathbb{D}, G_i)|}$$

holds, we have $P[|\hat{\mathcal{M}}(\mathbb{D}, G_i)| < \alpha \cdot |\mathcal{M}(\mathbb{D}, G_i)|] < \delta$.

Proof: Let X_j be a random variable that is 1 if j-th object in $\mathbb{S}$ belongs to $\mathcal{M}(\mathbb{D}, G_i)$ and 0 otherwise. Since we do uniform random sampling, $X_1, \cdots, X_{|\mathbb{S}|}$ are independent Bernoulli trials with $P(X_j = 1) = |\mathcal{M}(\mathbb{D}, G_i)|/|\mathbb{D}|$. The number of objects in $\mathbb{S}$ belonging to $\mathcal{M}(\mathbb{D}, G_i)$ is $X = \sum_j X_j$ and the expected value of X is $\mu = E[X] = |\mathbb{S}| \cdot |\mathcal{M}(\mathbb{D}, G_i)|/|\mathbb{D}|$. We have $P[|\hat{\mathcal{M}}(\mathbb{D}, G_i)| < \alpha \cdot |\mathcal{M}(\mathbb{D}, G_i)|] = P[X \cdot |\mathbb{D}|/|\mathbb{S}| < \alpha \cdot |\mathcal{M}(\mathbb{D}, G_i)|]$ since $|\hat{\mathcal{M}}(\mathbb{D}, G_i)|$ is $X \cdot |\mathbb{D}|/|\mathbb{S}|$.

Chernoff bounds state that for $0 < \epsilon \leq 1$, we have $P[X < (1\text{-}\epsilon)\mu] < \exp(-\mu\epsilon^2/2)$. Rewriting the probability to conform to the Chernoff bounds, we get $P[X < (1-(1-\frac{\alpha\cdot|\mathbb{S}|\cdot|\mathcal{M}(\mathbb{D},G_i)|}{|\mathbb{D}|\mu}))\mu] < \delta$. Then, we obtain $\exp(-\frac{\mu}{2}(1-\frac{\alpha\cdot|\mathbb{S}|\cdot|\mathcal{M}(\mathbb{D},G_i)|}{|\mathbb{D}|\mu})^2) \leq \delta$ by applying the Chernoff bounds. Substituting $\mu = |\mathbb{S}| \cdot |\mathcal{M}(\mathbb{D}, G_i)|/|\mathbb{D}|$ and solving it for $|\mathbb{S}|$, we obtain the lower bound of $|\mathbb{S}|$. ∎

To compute the above bound for every problematic group G_i satisfying $s(obj) \cdot |\mathcal{M}(\mathbb{D}, G_i)| > s(mem)$, by letting $|\mathcal{M}(\mathbb{D}, G_i)| = \frac{s(mem)}{s(obj)}$, we have $|\mathbb{S}| \geq \frac{-2\cdot|\mathbb{D}|\cdot\ln\delta\cdot s(obj)}{(1-\alpha)^2\cdot s(mem)}$ since the lower bound of $|\mathbb{S}|$ is maximized when $|\mathcal{M}(\mathbb{D}, G_i)|$ is minimized.

Setting the split threshold ρ: When we build a *PS-QTREE* with a sample $\mathbb{S}$, we split a node n if the number of instances in n exceeds the split threshold ρ. To apply leaf node grouping with the *memory utilization heuristics*, we should guarantee that $s(obj)\cdot|\mathcal{M}(\mathbb{D}, n_\ell)|\leq\alpha\cdot s(mem)$ for each n_ℓ since every group G_i contains at least a single leaf node.

After the *PS-QTREE* is generated, we assume that the number of instances of objects appearing in each leaf node n_ℓ is at most $\rho \cdot |\mathbb{D}|/|\mathbb{S}|$. Let n_I be the average number of instances in each object. Then, under the assumption of uniform distribution, we have $|\mathcal{M}(\mathbb{D}, n_\ell)| \leq \rho/n_I \cdot |\mathbb{D}|/|\mathbb{S}|$. Thus, we set $\rho = \alpha \cdot s(mem) \cdot n_I \cdot |\mathbb{S}|/(s(obj) \cdot |\mathbb{D}|)$ obtained by finding the minimum ρ satisfying $s(obj) \cdot \rho/n_I \cdot |\mathbb{D}|/|\mathbb{S}| \leq \alpha \cdot s(mem)$.

6.4 PS-BRF-MR: Our Algorithm with Random Partitioning and Filtering

In this section, we present the MapReduce algorithm *PS-BRF-MR* which utilizes random partitioning as well as the filtering techniques in Section 6.1. We refer to the brute-force algorithm based on random partitioning without such filtering techniques as *PS-BR-MR*. Due to the space limitation, we omit the detailed pseudocodes for both algorithms.

Generally, random partitioning is not suitable to the continuous model since all

objects required to compute the skyline probability of an object U by performing the integration in Equation (3.2) cannot be in the same partition containing U. To apply random partitioning to the continuous model, we adapt a specific integration method, Monte Carlo integration [11], which is based on sample points (refer to [11] for details). Thus, for each object U, the partial values required to compute the integration for the skyline probabilities are computed using the sample points selected in each partition. Then, we calculate the skyline probability of U by integrating the partial values of all partitions.

PS-BRF-MR: When a dataset $\mathbb{D}$ is split into disjoint partitions, $P_1, \ldots, P_m$, to calculate the skyline probability of an instance $u_i \in U$, we compute its *k-th local* skyline probability $P_{LS}(u_i, U, k)$ in every partition P_k.

Definition 6.4.1 *For disjoint partitions $P_1, \ldots, P_m$ of a dataset $\mathbb{D}$ and an instance $u_i \in U$, the k-th* local skyline probability *of u_i, denoted by $P_{LS}(u_i, U, k)$, is*

$$\prod_{V \in P_k, V \neq U} (1 - \sum_{v_j \in V, v_j \prec u_i} P(v_j)).$$

By Equation (3.1), we obtain

$$P_{sky}(u_i) = P(u_i) \prod_{k=1}^{m} P_{LS}(u_i, U, k). \tag{6.2}$$

The algorithm *PS-BRF-MR* consists of two MapReduce phases. In the first MapReduce phase, the filtering techniques presented in Section 6.1 are applied to identify the non-skyline objects so that we can compute the skyline probabilities for the skyline *candidate* objects only. In the second MapReduce phase, we gather every local skyline probability of each instance to compute the skyline probabilities of all objects. *PS-BRF-MR* consists of the following three phases:

(1) PS-QTREE building phase: We build a *PS-QTREE* with a sample $\mathbb{S} \subset \mathbb{D}$ by calling the procedure *GenQtree* in Section 6.2.2. Recall that it is done without using MapReduce.

(2) Local skyline probability phase: After broadcasting a *PS-QTREE* and T_p, each map function checks if each object is a *candidate* by the filtering methods in Section 6.1.

We divide the data objects $\mathbb{D}$ into disjoint partitions, $P_1, \ldots, P_m$. For every partition-pair (P_i, P_j) with $1 \leq i \leq j \leq m$, we compute the local skyline probabilities of the instances in P_i and P_j in parallel. For each partition-pair (P_i, P_j), when $i = j$, for every instance u of each candidate object U in P_i, we compute the i-th local skyline probability $P_{LS}(u, U, i)$ defined in Definition 6.4.1. When $i < j$, we compute $P_{LS}(u, U, j)$ for every u of U in P_i by considering the instances $v \in V$ in P_j and calculate $P_{LS}(v, V, i)$ for every v of each candidate object V in P_j by considering the instances u in P_i. To reduce the number of comparisons, we compare the skyline candidate objects with other skyline candidate objects first and then compare them to non-skyline objects by using the secondary sorting illustrated in Section 6.3.

(3) Global skyline phase: We gather the local skyline probabilities computed in the previous phase and calculate the exact skyline probabilities of the instances of every skyline candidate object using Equation (6.2). For a candidate object U, if $P_{sky}(U) = \sum_{u \in U} P(u) \prod_{i=1}^{m} P_{LS}\,(u, U, i) \geq T_p$, we output U as a skyline object.

The continuous model: For the continuous model, we use a specific integration method, Monte Carlo integral [11] which samples points u from the uncertainty region $U.R$ uniformly. In the local skyline probability phase, for each partition-pair (P_i, P_j), when $P_i = P_j$, it calculates $\prod_{V \in P_i, U \neq V} P_{LS}(u, V)$ for all $U \in P_i$ where $P_{LS}(u, V)$ is $1 - \int_{V.R} V.f(v)\mathbb{I}(\, v \prec u)\mathrm{d}v$. If $P_i \neq P_j$, we compute $\prod_{V \in P_j} P_{LS}\,(u, V)$ for $U \in P_i$ and $\prod_{U \in P_i} P_{LS}(v, U)$ for $V \in P_j$. In the global skyline phase, we compute $P_{sky}(U)$ by utilizing the $\prod_{V \in P_i, U \neq V} P_{LS}(\, u, V)$ obtained in the previous phase since $P_{sky}(U)$ is the average value of $|U.R| \times U.f(u) \prod_{V \in \mathbb{D}, V \neq U} P_{LS}(u, V)$ by using Monte Carlo integration as in Section 6.3.

Parameter	Range	Default
Number of samples ($\|\mathbb{S}\|$)	$1000 \sim$ 10,000	1000 for *PS-QPF-MR* 2000 for *PS-QP-MR* 10000 for *PS-BRF-MR*
Number of dominating objects ($\|F\|$)	$50 \sim 5000$	100 for *PS-QPF-MR* 1000 for *PS-BRF-MR*
Number of objects ($\|\mathbb{D}\|$)	$10^5 \sim 10^8$	10^7
Number of dimensions (d)	$2 \sim 8$	4
Probability threshold (T_p)	$0.1 \sim 0.6$	0.3
Number of inst. per object (ℓ)	$1 \sim 400$	40
Number of machines (t)	$10 \sim 200$	25

Table 6.1: Parameters used for the probabilistic skyline algorithms

6.5 Experiments

Experiments were done mainly on a cluster of 50 machines with Intel i3 3.3GHz CPU and 4GB of memory running Linux. We also used Amazon's EC2 Infrastructure as a Service (IaaS) cloud to show the scalability of *PS-QPF-MR* up to 200 machines with Intel Xeon 2.5GHz CPU and 3.75GB of memory. The implementations of all algorithms were compiled by Javac 1.6. We used Hadoop 1.2.1 for MapReduce. The execution times in the graphs are plotted in log scale.

Datasets: We generated the synthetic datasets with correlated, independent and anti-correlated distributions, referred to as *COR*, *IND* and *ANTI* respectively, since they are typically used to evaluate the performance of skyline algorithms [10, 40, 41]. For a d-dimensional space, we generated the center c of each object using the three distributions where each dimension has a domain of $[1, 10000)$. In the discrete model, for each object U, we selected the number of U's instances using the uniform distribution in the range $[1, \ell]$, where ℓ is 40 by default. Each instance was generated inside the rectangle centered at c whose edge size is uniformly distributed in the range $[1, 200]$. The ratio of the objects U with $\sum_{u_i \in U} P(u_i) = 1$ to all objects in the dataset was

Algorithm	Description
PS-QP-MR	The algorithm with quadtree partitioning
PS-QPF-MR	The algorithm with quadtree partitioning and filtering
PS-BR-MR	The algorithm with random partitioning
PS-BRF-MR	The algorithm with random partitioning and filtering
PSMR	The state-of-the-art algorithm in [18]

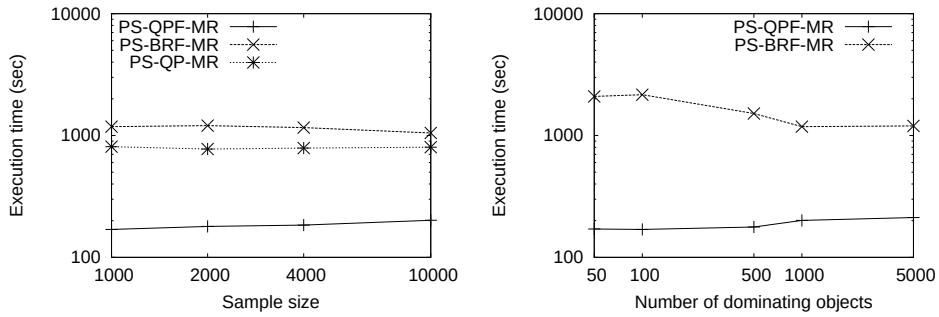
Table 6.2: Implemented probabilistic skyline algorithms

set to 0.5. In the continuous model, for each object U, we selected the length of k-th dimension of $U.R$ in $[1, 200]$, and assumed $U.f(\cdot)$ is the uniform distribution. The sizes of resulting synthetic datasets are varied from 88MB to 86GB depending on the number of points ($|\mathbb{D}|$), the number of dimensions (d) and the number of instances per each object (ℓ). We also varied the probability threshold T_p from 0.1 to 0.6 to produce diverse probabilistic skyline queries. We set $T_p = 0.3$ as the default value. The parameters used by our algorithms are summarized in Table 6.1.

Implemented algorithms: The MapReduce algorithms implemented for the probabilistic skyline are presented in Table 6.2. We do not plot the execution times of some algorithms when they did not finish within 6 hours or they did not work due to some reasons such as out of memory.

Default value of m: In the random partitioning algorithms (i.e., *PS-BR-MR* and *PS-BRF-MR*), we split data $\mathbb{D}$ into m partitions. Since such algorithms split all pairs of objects into $m(m+1)/2$ partition-pairs, we set m to the minimum natural number satisfying $m(m+1)/2 \geq t$ so that each machine can process at least a single partition-pair.

Default values of $|\mathbb{S}|$ and $|F|$: By the discussion in Section 6.3.2, the sample size $|\mathbb{S}|$ should be at least 700 objects since $|\mathbb{S}| \geq \frac{-2 \cdot |\mathbb{D}| \cdot \ln \delta \cdot s(obj)}{(1-\alpha)^2 \cdot s(mem)} = 700$ with $s(mem) =$ 4GB, $s(obj) =$ 1KB, $|\mathbb{D}| = 10^7$, $\alpha = 0.8$ and $\delta = 0.01$. Thus, to find the proper sizes of a sample $\mathbb{S}$ and a dominating object set F (i.e., $|\mathbb{S}|$ and $|F|$), we ran our algo-



(a) Varying $|\mathbb{S}|$

(b) Varying $|F|$

Figure 6.6: Selection of $|\mathbb{S}|$ and $|F|$

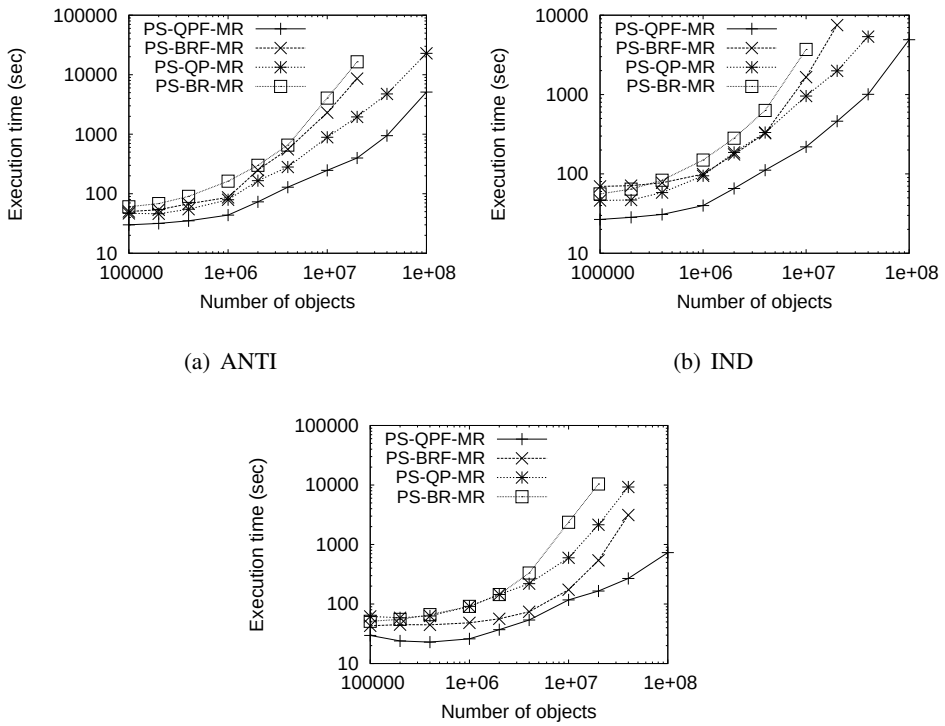
rithms with varying $|\mathbb{S}|$ from $1,000$ to $10,000$ and $|F|$ from 50 to $5,000$, respectively. The average execution times over all datasets with varying $|\mathbb{S}|$ and $|F|$ are shown in Figures 6.6(a) and 6.6(b), respectively. Since *PS-BR-MR* does not utilize a *PS-QTREE* and a dominating object set F, we do not plot its execution times in Figure 6.6.

Although more objects are filtered by the upper-bound and dominance-power filtering as $|\mathbb{S}|$ and $|F|$ increase, the costs for computing upper bounds and maintaining dominating object set increase. Consequently, we set the default values of $|\mathbb{S}|$ and $|F|$ with which each algorithm show the best performance. For instance, the best performance of *PS-QPF-MR* is obtained with $|\mathbb{S}| = 1000$ and $|F| = 100$.

6.5.1 Performance Results for Probabilistic Skylines

We presented the experiment results with the discrete model first and the continuous model next.

Varying $|\mathbb{D}|$: We plotted the running times of the tested algorithms with varying the number of objects $|\mathbb{D}|$ from 10^5 to 10^8 with each dataset in Figures 6.7(a), (b) and (c), respectively. *PS-QPF-MR* with *COR* is faster than that with the other datasets since most of instances are dominated by a few instances in *COR* and the three filtering methods can identify non-skyline objects effectively. The best performance is shown



(a) ANTI

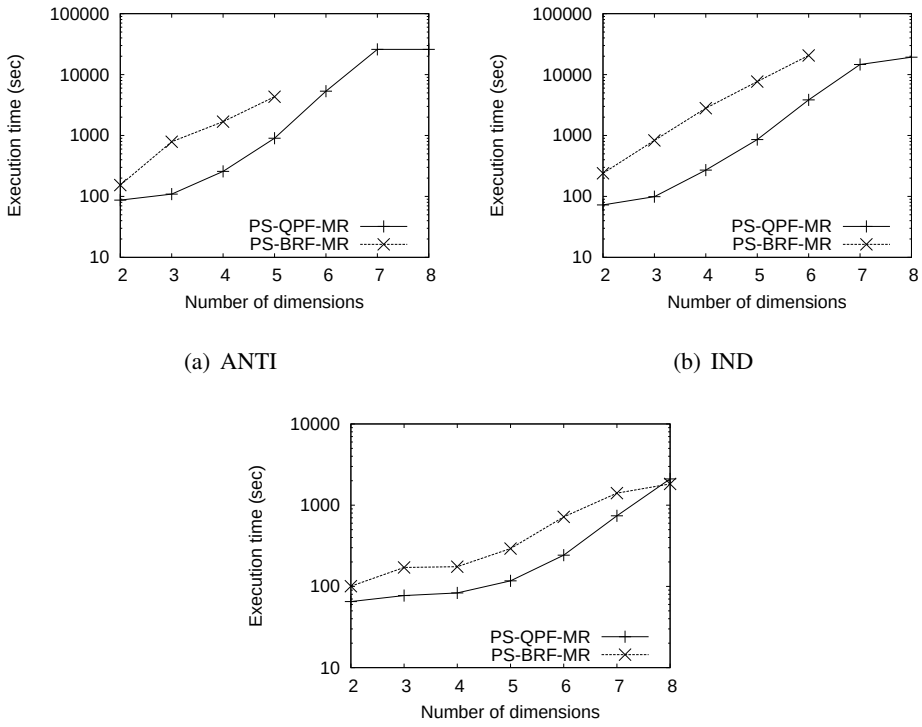
(b) IND

(c) COR

Figure 6.7: Varying the number of objects ($|\mathbb{D}|$)

by *PS-QPF-MR* which utilizes the three filtering methods and the quadtree partitioning. *PS-QPF-MR* is also found to be at least 1.7 times faster than *PS-BRF-MR*. Since *PS-QPF-MR* and *PS-BRF-MR* are always faster than *PS-QP-MR* and *PS-BR-MR*, respectively, due to the three filtering methods, we showed only the execution times of *PS-QPF-MR* and *PS-BRF-MR* in the rest of the paper.

Varying d: The execution times with varying the number of dimensions d from 2 to 8 were reported in Figure 6.8. Since the time complexity of checking the dominance relationship between instances is $O(d)$, the execution times of both algorithms become larger as d grows. We found that *PS-QPF-MR* is 4.4 times faster than *PS-BRF-MR* on the average since quadtree partitioning is very effective. However, *PS-BRF-MR*



(a) ANTI

(b) IND

(c) COR

Figure 6.8: Varying the number of dimensions (d)

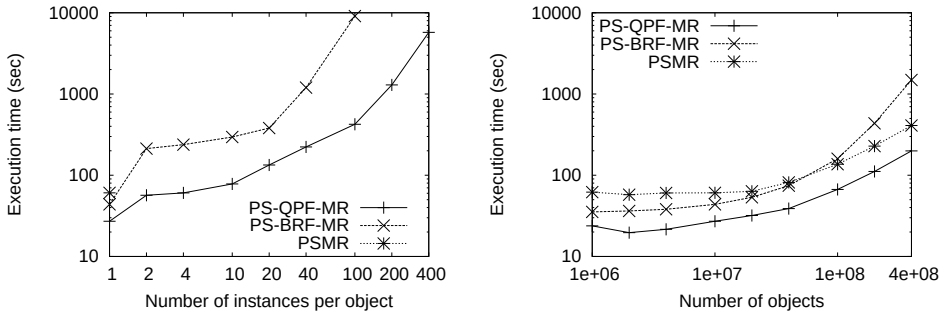
performs fast for *COR* with high dimension since there are a small number of candidate objects and merging their skyline probabilities can be done quickly.

Varying T_p: We showed the execution time, number of candidate objects and number of skyline objects on average with varying T_p from 0.1 to 0.6 in Table 6.3. Since all filtering methods are applied before data partitioning, the average numbers of candidate objects by both algorithms are the same. With increasing T_p, since the numbers of candidate and skyline objects decrease, the execution times decrease. *PS-QPF-MR* is up to 7.9 times faster than *PS-BRF-MR*.

Varying ℓ: We evaluated both algorithms with changing the number of instances per object ℓ from 1 to 400. We also tested the state-of-the-art algorithm *PSMR* for the

T_p	*PS-QPF-MR*	*PS-BRF-MR*	# of candidate objects	# of skyline objects
0.1	400	1905	259009	1057
0.2	223	1469	204964	509
0.3	164	1452	165907	329
0.4	161	1267	140129	234
0.5	151	1164	121678	172
0.6	151	1115	106530	127

Table 6.3: Varying the probability threshold (T_p)



(a) Varying ℓ

(b) Varying $|\mathbb{D}|$ when $\ell = 1$

Figure 6.9: Varying ℓ and $|\mathbb{D}|$ when $\ell = 1$

specific case where each object has only a single instance. We showed the average execution times over all datasets in Figure 6.9(a). Since *PSMR* is only applicable when ℓ=1, we plotted the execution time of *PSMR* only when ℓ=1. Although *PS-BRF-MR* is worse than *PSMR* with large datasets, *PS-QPF-MR* is 2.1 times faster than *PSMR*. We also reported the execution times of all algorithms with varying $|\mathbb{D}|$ from 10^5 to 4×10^8 when ℓ=1 in Figure 6.9(b). We found that *PS-QPF-MR* is 2.3 times faster than *PSMR* on the average.

Varying t: With increasing the number of machines t up to 50 in our cluster, we presented the execution times with the default-sized datasets ($|\mathbb{D}|$=10^7) and large datasets ($|\mathbb{D}|$=10^8) in Table 6.4. For the large datasets, since *PS-BRF-MR* finished

Algorithm	t	10	20	25	30	40	50
	IND	401	242	212	197	167	162
PS-QPF-MR	*COR*	177	89	85	79	78	64
($\|\mathbb{D}\| = 10^7$)	*ANTI*	429	228	196	175	152	135
	IND	4373	2089	1872	1469	1177	912
PS-BRF-MR	*COR*	361	205	179	160	130	117
($\|\mathbb{D}\| = 10^7$)	*ANTI*	4893	2409	2307	1664	1338	959
	IND	8107	4555	3569	2698	2268	1811
PS-QPF-MR	*COR*	1119	627	541	471	398	351
($\|\mathbb{D}\| = 10^8$)	*ANTI*	8442	3874	3738	3002	2206	1987

Table 6.4: Varying t with our cluster (sec)

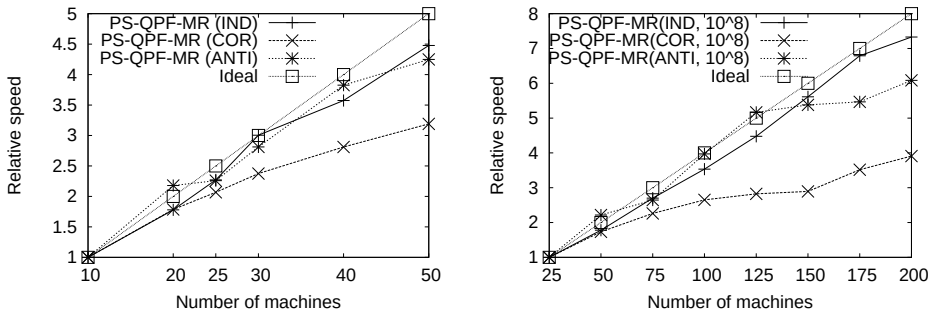
t	25	50	75	100	125	150	175	200
IND	8783	4936	3252	2485	1961	1565	1293	1198
COR	1234	712	546	466	437	426	351	316
ANTI	12655	5713	4783	3186	2451	2352	2315	2080

Table 6.5: Varying t on Amazon EC2 with $|\mathbb{D}|$=10^8 (sec)

within 6 hours only when $t = 40, 50$ with COR, we reported execution times and *relative speedup* to 10 machines of *PS-QPF-MR* only in Table 6.4 and Figure 6.10(a), respectively.

To show the scalability of *PS-QPF-MR*, we also tested with large datasets ($|\mathbb{D}|$=10^8) on Amazon EC2 Infrastructure consisting of 200 machines and showed the execution time as well as *relative speedup* to 25 machines in Table 6.5 and Figure 6.10(b), respectively.

In both experiments using large datasets, *PS-QPF-MR* shows linear speedup with *IND* and *ANTI*, but sub-linear speedup with *COR*. It is because the number of probabilistic skyline objects in the correlated data is very small and the benefit of using a large number of machines is marginal.



(a) With our cluster

(b) With Amazon EC2

Figure 6.10: Relative speedups with $|\mathbb{D}| = 10^8$

Filtering technique	IND	COR	ANTI
Zero-probability (# of inst.)	12806654	172962353	4352818
Upper-bound (# of obj.)	882787	8614581	490691
Dominance-power (# of obj.)	9773641	9986907	9605070

Table 6.6: Filtered objects per filtering technique

The effects of filtering techniques: We first presented the number of instances removed for zero-filtering technique and the numbers of non-skyline objects detected not to compute their skyline probabilities for each of the other filtering techniques in Table 6.6. We found that dominance-power filtering detects more non-skyline objects than upper-bound filtering. In Table 6.7, we showed the execution times of *PS-QPF-MR* by applying dominance-power filtering only (D) or all filtering techniques (ALL). When all filtering techniques were used, we applied them in the order of zero-probability filtering, upper-bound filtering and dominance-power filtering. Applying all filtering

Dataset	*IND*	*COR*	*ANTI*	Average
PS-QPF-MR (ALL)	212	85	196	164
PS-QPF-MR (D)	226	123	207	185

Table 6.7: Effects of the filtering techniques (sec)

PS-QPF-MR	L and W	L	NONE
Execution time (sec)	164	301	329
# of transmitted instances ($\times 10^6$)	454	454	894

Table 6.8: Effects of optimization techniques

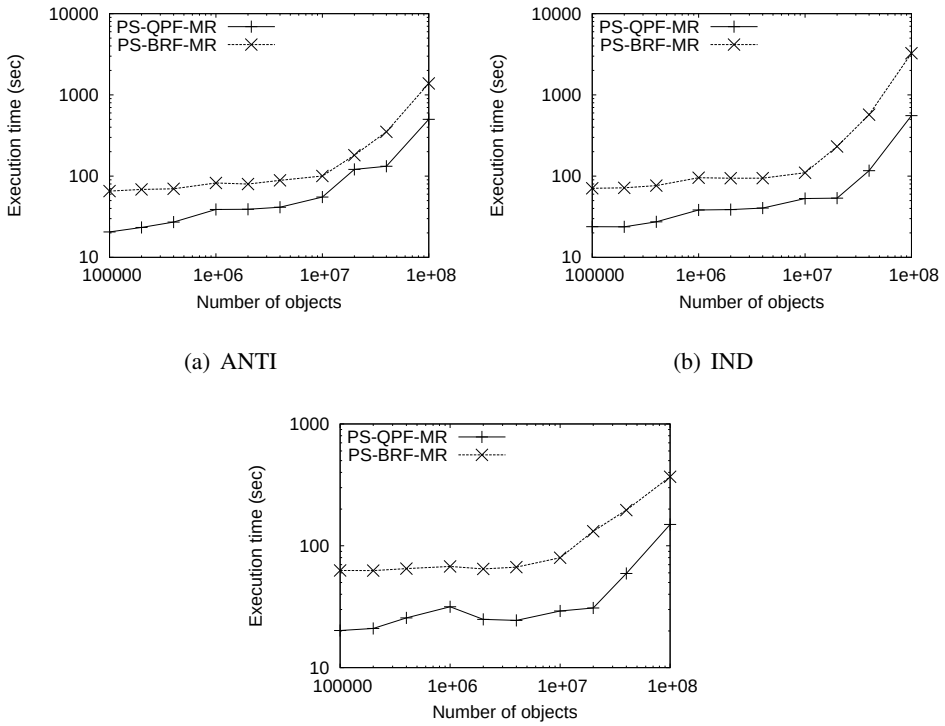
$\|\mathbb{D}\| = 10^7$	Algorithm	IND	COR	ANTI
Execution time (sec)	*PS-QPF-MR*	164	82	198
	PS-BRF-MR	473	143	470
# of dominance comparisons ($\times 10^6$)	*PS-QPF-MR*	59579	2037	58834
	PS-BRF-MR	72844	3432	70369

Table 6.9: Effect of quadtree partitioning using EC2

techniques is faster than applying dominance-power filtering only in *PS-QPF-MR*.

The effects of optimization techniques: In Table 6.8, we reported the average execution times and average number of transmitted instances by *PS-QPF-MR* without leaf node grouping and workload balancing (NONE), *PS-QPF-MR* with leaf node grouping only (L) and *PS-QPF-MR* with both methods (L and W). *PS-QPF-MR* with leaf node grouping (L) has 49% of transmitted instances than *PS-QPF-MR* without both methods (NONE). Since the workload balancing technique splits the instances required to compute the skyline probabilities of objects in each group in order to utilize all machines available, *PS-QPF-MR* with L and W is the most efficient as shown in Table 6.8.

The effect of quadtree partitioning: To show the effectiveness of quadtree partitioning, we experimented with datasets of $|\mathbb{D}| = 10^7$ using 200 machines on Amazon EC2 and presented the execution times as well as the numbers of checking dominance relationships between instances of objects by both algorithms in Table 6.9. While *PS-QPF-MR* has 1.37 times smaller number of dominance relationship comparisons than *PS-BRF-MR*, *PS-QPF-MR* is 2.33 times faster than *PS-QPF-MR*, on average. Since



(a) ANTI

(b) IND

(c) COR

Figure 6.11: Varying the number of objects ($|\mathbb{D}|$) for the continuous model

PS-QPF-MR has a single MapReduce phase but *PS-BRF-MR* consists of two MapReduce phases, the performance gain in terms of execution time is higher than that in terms of dominance relationship comparisons for *PS-QPF-MR*.

The continuous model: We set the default values of $(|\mathbb{S}|, |F|)$ to $(10000, 2000)$ and $(2000, 1000)$ for *PS-QPF-MR* and *PS-BRF-MR*, respectively, since they performed the best with those values. We omit the experimental results with varying $|\mathbb{S}|$ and $|F|$ since they show similar patterns with those for the discrete model. In Figure 6.11, we plotted the execution times of both algorithms with varying $|\mathbb{D}|$. We found that *PS-QPF-MR* runs up to 7.72 times faster and is 2.37 times faster on the average than *PS-BRF-MR*.

Chapter 7

Conclusion

We introduced efficient parallel algorithms for the skyline, dynamic skyline, reverse skyline and probabilistic skyline queries.

We first study the optimization of skyline query processing. We propose an efficient parallel skyline computation algorithm which consists of three phases. In the first phase, we build a new histogram which is an extension of quadtrees to effectively prune out non-skyline points in advance. In the second phase, we split data into partitions based on the regions divided by our proposed histograms and compute candidate skyline points for each partition independently using MapReduce. Finally, we check whether each candidate point is actually a skyline point in every region independently by another MapReduce phase. Although our proposed algorithms are devised for the MapReduce framework, they can be also applied to other frameworks such as MPI and multi-cores. Since the dynamic skyline can be obtained by calculating the skyline after transforming the coordinates of data points with respect to a given query point, we can utilize our parallel skyline computation algorithm to compute the dynamic skyline.

Second, we investigate the reverse skyline query processing. To the best of our knowledge, no existing work has addressed computing the reverse skyline query using MapReduce. We analyze the characteristics of the reverse skylines theoretically to prune non-reverse skyline points. Based on the properties of the reverse skylines, we

develop the novel parallel algorithm consisting of three phases. In the first phase, we build a variant of quadtree which is used for pruning non-reverse skyline points by utilizing the characteristics. In the second phase, by using MapReduce, we compute the local reverse skyline points in each partition split by the histogram. In the last phase, we compute the global reverse skyline points in every region independently and simultaneously by using MapReduce.

We finally present the efficient algorithm for computing the probabilistic skyline query for both continuous and discrete uncertain models. To prune out non-probabilistic skyline objects in advance, we develop three filtering methods. The proposed algorithms are composed of only two phases. In the first phase, we build a variant of a quadtree. In the second phase, by utilizing the proposed filtering methods, we efficiently compute the probabilistic skyline in each partition according to the space split by the variant of a quadtree. To balance the workload and reduce the transmission overhead, we also propose a workload balancing technique for the second phase.

For each type of skyline queries, we performed extensive experiments and confirmed the effectiveness and scalability of our proposed algorithms. We believe that our algorithms proposed in this dissertation can be applied practically in many important applications and enhance the performance of skyline query processing.

Bibliography

[1] E. Ada and C. Ré. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 30(2):15–22, 2007.

[2] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[3] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.

[4] Apache. Hadoop. http://hadoop.apache.org.

[5] M. J. Atallah and Y. Qi. Computing all skyline probabilities for uncertain data. In *PODS*, 2009.

[6] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, page 405, 2006.

[7] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Trans. Database Syst.*, 33(4):31, 2008.

[8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[9] C. Böhm, F. Fiedler, A. Oswald, C. Plant, and B. Wackersreuther. Probabilistic skyline queries. In *CIKM*, pages 651–660. ACM, 2009.

[10] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[11] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 7:1–49, 1998.

[12] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, 2006.

[13] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[14] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 523–544, 2007.

[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, 2008.

[16] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.

[17] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR*, pages 317–328, 2005.

[18] L. Ding, G. Wang, J. Xin, and Y. Yuan. Efficient probabilistic skyline query processing in mapreduce. In *BigData Congress*, pages 203–210, 2013.

[19] X. L. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, pages 469–500, 2009.

[20] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1), 1974.

[21] Y. Gao, Q. Liu, B. Zheng, and G. Chen. On efficient reverse skyline query processing. *Expert Systems with Applications*, 41(7):3237–3249, 2014.

[22] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[23] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, 2006.

[24] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[25] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–96, 2011.

[26] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, 2002.

[27] A. N. Laboratory. Mpich2. http://www.mpich.org/.

[28] T. Lappas and D. Gunopulos. Efficient confident search in large review corpora. In *ECML/PKDD (2)*, 2010.

[29] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Information Systems*, 39:1–21, 2014.

[30] J. Lee, S. won Hwang, Z. Nie, and J.-R. Wen. Navigation system for product search. In *ICDE*, 2010.

[31] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. Preference query evaluation over expensive attributes. In *CIKM*, 2010.

[32] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, pages 213–226, 2008.

[33] X. Lian and L. Chen. Reverse skyline search in uncertain databases. *ACM Trans. Database Syst.*, 35(1), 2010.

[34] X. Lin, Y. Zhang, W. Zhang, and M. A. Cheema. Stochastic skyline operator. In *ICDE*, pages 721–732, 2011.

[35] V. Ljosa and A. K. Singh. Top-k spatial joins of probabilistic objects. In *ICDE*, pages 566–575, 2008.

[36] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *VLDB*, 5(8):704–715, 2012.

[37] K. Mullesgaard, J. L. Pedersen, H. Lu, and Y. Zhou. Efficient skyline computation in mapreduce. In *EDBT*, pages 37–48, 2014.

[38] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, 2003.

[39] A. N. Papadopoulos, A. Lyritsis, and Y. Manolopoulos. Skygraph: an algorithm for important subgraph discovery in relational graphs. *Data Mining and Knowledge Discovery*, 17(1):57–76, 2008.

[40] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *VLDB*, 6(14):2002–2013, 2013.

[41] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, pages 15–26, 2007.

[42] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[43] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.

[44] M. A. Soliman, I. F. Ilyas, and K.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.

[45] A. Soulet, C. Raïssi, M. Plantevit, and B. Cremilleux. Mining dominant patterns in the sky. In *2011 IEEE 11th International Conference on Data Mining*, pages 655–664. IEEE, 2011.

[46] Spark. Spark. http://spark.apache.org.

[47] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[48] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *TKDE*, 18(2):377–391, 2006.

[49] M. Van Leeuwen and A. Ukkonen. Discovering skylines of subgroup sets. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 272–287. Springer, 2013.

[50] V. V. Vazirani. *Approximation algorithms*. springer, 2001.

[51] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[52] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, pages 227–238, 2008.

[53] G. Wang, J. Xin, L. Chen, and Y. Liu. Energy-efficient reverse skyline query processing over wireless sensor networks. *TKDE*, 24(7), 2012.

[54] Y. Wang, X. Li, X. Li, and Y. Wang. A survey of queries over uncertain data. *Knowledge and information systems*, 37(3):485–530, 2013.

[55] X. Wu, Y. Tao, R. C.-W. Wong, L. Ding, and J. X. Yu. Finding the influence set through skylines. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 1030–1041. ACM, 2009.

[56] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. In *DASFAA*, pages 403–414, 2011.

[57] J. Zhang, X. Jiang, W. S. Ku, and X. Qin. Efficient parallel skyline evaluation using mapreduce. *IEEE Trans. Parallel Distrib. Syst.*, 27(7):1996–2009, 2016.

[58] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, pages 1060–1071, 2009.

[59] L. Zhu, C. Li, and H. Chen. Efficient computation of reverse skyline on data stream. In *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, volume 1, pages 735–739. IEEE, 2009.

[60] L. Zhu, Y. Tao, and S. Zhou. Distributed skyline retrieval with low bandwidth consumption. *TKDE*, 21(3):384, 2009.

[61] L. Zou, L. Chen, M. T. Özsu, and D. Zhao. Dynamic skyline queries in large graphs. In *DASFAA*, 2010.

초 록

스카이라인 질의와 스카이라인에서 파생된 동적 스카이라인, 역 스카이라인 그리고 확률적 스카이라인 질의들은 다양한 응용이 가능하기 때문에 최근에 많은 연구가 진행되어 왔다. 스카이라인 질의들은 큰 데이터를 처리해야하는 경우가 많기 때문에 효율적인 스카이라인 질의 처리는 중요한 문제이다. 큰 데이터를 처리해야하는 경우를 위해 맵리듀스 프레임워크가 제안되었고, 따라서 본 논문에서는 스카이라인, 동적 스카이라인, 역 스카이라인, 확률적 스카이라인 질의 처리를 위한 효율적인 맵리듀스 알고리즘을 개발한다.

스카이라인, 동적 스카이라인, 역 스카이라인에 대해서는 질의 결과에 포함될 수 없는 데이터를 빠르게 제거하기 위해서 쿼드트리에 기반한 히스토그램을 생성한다. 그리고 히스토그램에 따라 데이터를 여러 파티션으로 나누고 각 파티션에 있는 데이터만을 이용하여 스카이라인이 될 수 있는 후보 데이터를 맵리듀스를 이용하여 병렬적으로 뽑아낸다. 그 후에 다시 맵리듀스를 사용하여 병렬적으로 후보 데이터 중 실제 스카이라인을 찾아낸다. 확률적 스카이라인의 효율적인 처리를 위해 먼저 세가지 필터링 기법을 제안하였다. 이 필터링 기법을 활용할 수 있도록 쿼드트리에 기반한 히스토그램을 생성한다. 쿼드트리의 영역에 따라 데이터를 파티션하고 각 파티션마다 확률적 스카이라인 점들을 찾아낸다. 각 컴퓨터의 수행시간을 비슷하게 맞추기 위해서 부하균형 기법도 제안하였다. 다양한 실험을 통해 제안한 알고리즘의 성능들이 최신 관련 연구 보다 좋음을 확인하였고, 사용하는 컴퓨터의 수를 늘림에 따라 성능이 확장성을 갖고 있음을 확인하였다.

주요어: 스카이라인 질의, 역 스카이라인 질의, 확률적 스카이라인 질의, 맵리듀스
학번: 2007-2007