Adaptive Processing for Distributed Skyline Queries over Uncertain Data

Xu Zhou, Kenli Li, *Member, IEEE*, Yantao Zhou, and Keqin Li, *Fellow, IEEE*

Abstract—Query processing over uncertain data has gained growing attention, because it is necessary to deal with uncertain data in many real-life applications. In this paper, we investigate skyline queries over uncertain data in distributed environments (DSUD query) whose research is only in an early stage. The state-of-the-art algorithm, called e-DSUD algorithm, is designed for processing this query. It has the desirable characteristics of progressiveness and minimum bandwidth consumption. However, it still needs to be perfected in three aspects. (1) Progressiveness. Each time it only returns one query result at most. (2) Efficiency. There are a significant amount of redundant I/O cost and numerous iterations which causes a long total query time. (3) Universality. It is restricted to the case where local skyline tuples are incomparability. To address these concerns, we first present a detailed analysis of the e-DSUD algorithm and then develop an improved framework for the DSUD query, namely IDSUD. Based on the new framework, we propose an adaptive algorithm, called ADSUD, for the DSUD query. In the algorithm, we redefine the approximate global skyline probability and choose local representative tuples due to minimum probabilistic bounding rectangle adaptively. Furthermore, we design a progressive pruning method and apply the reuse mechanism to improve its efficiency. The results of extensive experiments verify the better overall performance of our algorithm than the e-DSUD algorithm.

Index Terms—Data management, distributed database, skyline query, uncertain data

1 Introduction

1.1 Motivation

Recently skyline query processing has become a topic issue in database management research. The popularity of skyline queries mainly depends on their widespread use in real-world applications, such as multi-criteria data analysis, data mining, and decision making [1]. In those applications, data uncertainty arises inherently for several reasons, such as incomplete survey results, data measure and collection approaches, and so on [2]. Furthermore, with the development of big data and cloud computing, data storage has trended to become increasingly distributed [2]. In practice, more and more applications need to collect data from multiple data sources which have distributed and decentralized control. Since the data uncertainty and distributed data storage have been the inherent characteristics of many applications, skyline queries over uncertain data in distributed environments (DSUD query) will be paid growing attention [3], [4].

DSUD query processing is a vital research topic with many potential real-life applications. Consider the stock market application scenario as an example [3]. Stock traders always want to mine the stocks with the best investment potential. For this purpose, they need to access history deals stored in multiple distributed databases. The databases are usually scattered over different places like New York, London, Shanghai, Tokyo, etc. Moreover, the historical deal is uncertain for network or human beings. So the uncertainty of each deal must be taken into consideration. In this case, a DSUD query is more suitable because it allows better local data management, smaller update cost, and higher tolerance to machine failures [4].

We take the uncertain dataset shown in Fig. 1 as an example of a P-skyline query. The property value and probability of each deal in Fig. 1 is depicted in Table 1. Assume a tuple is recorded by two attributes (price, volume). Suppose a tuple with a lower price and a lower volume will be a good choice. Let the probabilistic threshold be equal to 0.4. According to the definition of skyline probability in [1], since there is no tuple dominating the tuple d_1, d_2, and d_6, it holds that the skyline probabilities of the above tuples are equal to their existential probabilities, respectively. For tuple d_3 which is dominated by tuple d_1, its skyline probability is computed by $0.6 \times (1-0.3) = 0.42$. Similarly, the skyline probabilities of d_4 and d_5 are equal to $0.8 \times (1-0.3) \times (1-0.4) = 0.336$ and $0.3 \times (1-0.4) = 0.18$, respectively. Since the skyline probabilities of d_2, d_3, and d_6 are larger than the probability threshold, we retrieve them as our skyline query results.

There have been abundant research achievements about skyline queries over uncertain data, but most of them are primarily focused on the assumption of a single and centralized storage database [1], [5], [6], [7]. Such approaches assume a sole relation as input, and lack adaptations or optimizations specific to distributed computing environments [8]. Furthermore, many attempts have been made for distributed skyline query over precise data. They resulted

- Z. Zhou, K. Li, and Y. Zhou are with the College of Information Science and Engineering, Hunan University, and the National Supercomputing Center, Changsha, Hunan 410082, China.
E-mail: zhouxu2006@126.com, {lkl, yantao_z}@hnu.edu.cn.
- K. Li is with the College of Information Science and Engineering, Hunan University, and the National Supercomputing Center, Changsha, Hunan 410082, China, and the Department of Computer Science, State University of New York, New Paltz, New York 12561.
E-mail: lik@newpaltz.edu.

Manuscript received 30 Jan. 2015; revised 16 July 2015; accepted 10 Aug. 2015. Date of publication 2 Sept. 2015; date of current version 6 Jan. 2016.
Recommended for acceptance by D. Olteanu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2475764

1041-4347 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

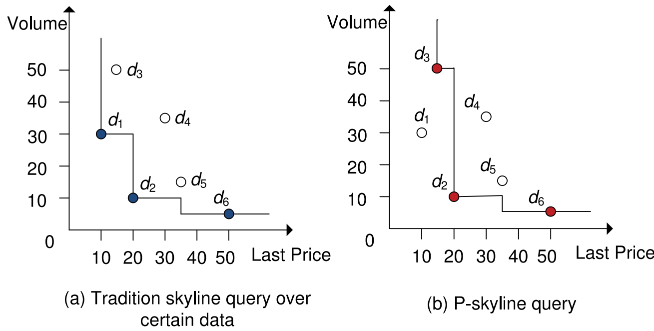


Fig. 1. Illustration of P-skyline query.

in a lot of famous approaches [4] discussed in Section 2. Nevertheless, most of these approaches are inapplicable to our problems because their research basis is the certain database. On the contrary, we carry on the distributed skyline query research on uncertain data in this paper.

Note that skyline queries over uncertain data in distributed environments have been studied quite extensively but separately. Ding and Jin [3] first formulated the DSUD query. Their pioneering work has desirable features of minimize bandwidth consumption and progressiveness. However, it still needs to be improved from three respects of progressiveness, efficiency, and universality (See Section 2.3 for details).

1.2 Our Contributions

As mentioned above, the famous activities about skyline queries in distributed environments are mainly researched over certain data [3], [4]. Moreover, the existing algorithms about our problem also have several aspects which need to be perfected. Therefore it leads to an urgent requirement for researching on DSUD query. Motivated by these concerns, we make the key contributions as follows.

- We review the work about DSUD query and summarize the objectives of the algorithms about it.
- We propose an improved framework for the DSUD query where a Query-Routing phase is used to prune unqualified local sites.
- We present an adaptive (ADSUD) algorithm for the DSUD query on the basis of the IDSUD framework. In the algorithm, we redefine the approximate global skyline probability to improve its universality, and give a definition of the MPBR to boost its progressiveness. Moreover, we design an improved PR-tree (IPR-tree) to organize the local data sets and apply the reuse mechanism to accelerate the DSUD query.
- We conduct a comprehensive evaluation of our ADSUD algorithm. The experimental results illustrate that ADSUD has much better efficiency and progressiveness comparing to e-DSUD.

The rest of the paper is organized as follows. Section 2 surveys previous work related to ours. Section 3 gives a definition of the DSUD query and presents the objectives of DSUD query algorithms. Section 4 introduces our improved distributed uncertain skyline query (IDSUD) framework. Section 5 describes our proposed algorithm, namely ADSUD, and analyzes its performance. Section 6 evaluates the performance of our proposed algorithm with experiments. Section 7 concludes this paper with directions for future work.

TABLE 1
Deal's Property Value and Probability

Deal	d_1	d_2	d_3	d_4	d_5	d_6
Price	10	20	15	30	35	50
Volume	30	10	50	35	15	5
Probability	0.3	0.4	0.6	0.8	0.3	0.5

2 Related Work

In this section, we survey the related work about the DSUD query.

2.1 Probabilistic Skyline Queries

The first study about probabilistic skyline query, named P-skyline, was reported by Pei et al. [1]. The P-Skyline query is developed to return skyline tuples whose skyline probabilities are larger than a specified probability threshold. This pioneering work inspires a number of follow-up studies [5], [6], [7], [8], [9], [10]. Dongwon et al. [5] proposed an in-memory tree structure, Z-tree, to find as many incomparable groups of instances as possible. Then a probabilistic skyline query algorithm, called P-Skyline, is designed. Atallah and Qi [6] focused on the cases where low probability events cannot be ignored. In such situations, it is necessary to compute skyline probabilities of all data items. In addition to conventional skyline operator over uncertain data, numerous skyline query variants have also been studied. Lian and Chen [7] formalized the probabilistic reverse skyline query over uncertain data in both monochromatic and bichromatic cases. Zhang et al. [8] focused on the problem of skyline query over sliding windows on uncertain data stream. Recently, Lian and Chen [9] formulated a novel and important query, namely probabilistic group subspace skyline query, that is useful in applications like sensor data analysis.

Although skyline queries over uncertain data have been researched widely, they are mainly based on the P-skyline semantics. The answer obtained via a P-Skyline query usually depends on the probabilistic threshold. When a small threshold is specified, it includes skyline tuples undesirably dominating each other; if a larger probabilistic threshold is arranged, it may contain much fewer skyline tuples [2]. To address the above concern, Liu et al. studied a new uncertain skyline query, called U-Skyline query. It reports a set of tuples that has the highest probability [2].

Notice that, all the above works are studied on a centralized database. So they can not be applied to compute distributed skyline queries over uncertain data directly.

2.2 Distributed Skyline Queries

Skyline queries in distributed environments were first researched in [10]. It supports the web information which is vertically partitioned into lists. Thereafter, in the literature, abundant research achievements have been gained to address distributed skyline queries [4].

Wang et al. and Chen et al. both researched skyline queries in structured P2P networks, namely BATON networks, where each peer is responsible for a partial region of data space [4]. Hose et al. proposed an approach to skyline processing in unstructured P2P networks that uses routing indexes to identify relevant peers. Cui et al. studied skyline

query processing in a distributed environment, where the querying peer can directly communicate with all peers [4]. Rocha-Junior et al. [11] proposed a grid-based approach for distributed skyline processing (AGiDS), which assumes that each peer maintains a grid-based data summary structure for describing its data distribution. Zhu et al. [12] proposed a feedback-based distributed skyline query algorithm (FDS) to support arbitrary horizontal partitioning. It aims at minimize the network bandwidth consumption. Chen et al. [13] paid attention on a constrained skyline query in distributed environments. In order to help to decide execution order between different local sites, they first presented a partition algorithm. Then they introduced heuristics for selecting a given number of multiple filtering points.

Trimponias et al. [14] studied skyline queries over distributed web information systems where the attributes of each tuple are provided by different sources. The proposed algorithm, namely PDS, is restricted to the case where each server maintains exactly one dimension. Taking the concern into account, Trimponias et al. [14] proposed an approach to skyline query on distributed vertical decompositions of arbitrary dimensionality. In [15], Vlachou et al. proposed a new angel-based partitioning scheme by using the hyperspherical coordinates of the data points. Based on a distributed data stream model, Sun et al. [16] designed an algorithm (BOCS) for skyline queries over data streams, which collects data streams from different servers and then processes skyline query on them. Skyline queries in wireless sensor networks (WSNs) were studied in [17]. It proposed an algorithm that is carried out in k iterations. In each iteration, it computes the skyline set of a partial dataset.

The MapReduce framework has an obvious advantage in processing skyline queries over big data sets due to its characteristics of ease of programming, scalability, and fault-tolerance [18]. In [19], Zhang et al. presented a preliminary approach for skyline queries in MapReduce. The authors developed three skyline algorithms under the MapReduce framework, called MR-BNL, MR-SFS, and MR-Bitmap. In [20], Tao et al. studied the minimal algorithms for MapReduce. Also, they introduced an approach for processing skyline queries over two-dimensional data sets, which needs to sort the data sets. In [21], Park et al. proposed an effective parallel algorithm, SKY-MR, for processing the skyline queries. In the SKY-MR algorithm, a sky-quadtree with a sample of the entire data set is designed and utilized in the data partition and local pruning. In [22], Chen et al. introduced the angle data partitioning approach and proposed the MR-Angle algorithm for skyline queries in MapReuce. Since the existing works on skyline computation in MapReduce still run significant parts serially, in [23], Mullesgaard et al. designed a grid partitioning scheme to divide the data space into partitions, and employed a bitstring to represent them. Furthermore, they proposed the MR-GRMRS algorithm for skyline queries, which applies multiple reducers for computing global skyline unprecedentedly.

Although there have been abundant activities for skyline queries in distributed environments, they can merely be applied to process precise data sets. Since they do not take the uncertain scenarios into consideration, the above algorithms are not applicable to handle the DSUD queries in this paper. Moreover, the DSUD queries are without the property of additivity, and hence, it is a great challenge to study them under MapReduce. Due to the huge advantage of MapReduce in processing large-scale data sets, we will take the DSUD queries under MapReduce as our next work.

2.3 Distributed Skyline Queries over Uncertain Data

The notation of the distributed skyline query over uncertain data was first proposed by Ding and Jin [3]. They proposed the DSUD algorithm and its enhance algorithm e-DSUD. In the e-DSUD algorithm, each local site computes its local skylines and sorts them by their local skyline probabilities first. The remaining part of the e-DSUD algorithm is an iterative process. In an iteration, each local site sends the representative tuple, which has the maximum local skyline probability, to the server H. Then, H collects the local representative tuples and computes the global candidate skylines. Afterwards, H selects the global candidate skyline, which has the maximum approximate global skyline probability, and sends it to local sites (except the local site that the skyline is from). Finally, each local site refreshes its local skyline set and helps computing the local skyline probability of the tuple from H.

Although the e-DSUD algorithm is both communication- and computation-efficient, it still has room for development in the following aspects.

1) Progressiveness. Although the e-DSUD algorithm can return the query results progressively, it can only return one query result after an iteration at most, and sometime it can not get any answer at all.
2) Efficiency. The e-DSUD algorithm could retrieve the query results with minimum bandwidth consumption. However, it does not take total query time, which is also a key performance metric to measure the distributed algorithms, into account. It needs $H(d, N)$ iterations for getting all the query results at least, where $H(d, N)$ is the expected number of skyline tuples and is computed by

$$H(d, N) \approx \sum_{n=0}^{N} \frac{(\ln n)^{d-1}}{d!} \times P(n), \tag{1}$$

where $P(n)$ denotes the probability of having exactly n tuples [3]. Moreover, for calculating local skyline probability of each global candidate skyline from server H, it needs to visit each local PR-tree for $H(d, n)$ times at least, which incurs redundant I/O and CPU costs.

From the analysis above, the two issues become the bottlenecks of the e-DSUD algorithm, because they cause total query time too long when processing on massive databases.
3) Universality. The e-DSUD algorithm is restricted to the case where the local skyline tuples are incomparable. But, generally speaking, some of the local skyline tuples are undesirably dominating each other [2]. In this case, it is unsuitable to choose a representative tuple based on its approximate global skyline probability at server H.

At the beginning of e-DSUD, each local site computes its local skylines, respectively. Given two local skyline tuples

t_1 and t_2 from local site S_i and $t_1 \succ t_2$, the local skyline probability of tuple t_2 is

$$Pr_{LSky}(t_2) = Pr(t_2) \times (1-Pr(t_1)) \times \prod_{\substack{t \succ t_2 \\ t \in UDB_i - t_1}} (1-Pr(t)). \quad (2)$$

At the first iteration, a prior queue L at the server H is initialized with multiple representative tuples from local sites. Assume that tuples t_1 and t_1' are received from different local sites S_i and S_j separately, and $t_1 \succ t_1'$. At the end of this iteration, the local skyline probability of tuple t_1' is refreshed as

$$Pr_{LSky}(t_1') = Pr_{LSky}(t_1') \times (1-Pr(t_1)) \times \prod_{\substack{t \succ t_1' \\ t \in L - t_1}} (1-Pr(t)). \quad (3)$$

At the start of the second iteration, assume that tuple t_2 is sent to H and $t_2 \succ t_1'$. The approximate global probability of tuple t_1' can be computed by

$$\begin{aligned} Pr'_{GSky}(t_1') = {} & Pr_{LSky}(t_1') \times Pr_{LSky}(t_2) \times \frac{(1-Pr(t_2))}{Pr(t_2)} \\ & \times \prod_{\substack{t \succ t_1' \\ t \in L \cap UDB_x - t_2}} \left[Pr_{LSky}(t) \times \frac{(1-Pr(t))}{Pr(t)} \right], \end{aligned} \quad (4)$$

where $1 \leq x \leq m, x \neq i, j$.

By introducing Equation (2) and Equation (3) to Equation (4), we have

$$\begin{aligned} Pr'_{GSky}(t_1') = {} & Pr_{LSky}(t_1') \times (1-Pr(t_1))^2 \times (1-Pr(t_2)) \\ & \times \prod_{\substack{t \in L \\ t \succ t_1}} (1-Pr(t)) \times \prod_{\substack{t \succ t_2 \\ t \in UDB_i - t_1}} (1-Pr(t)) \\ & \times \prod_{\substack{t \succ t_1' \\ t \in UDB_x \cap L - t_1}}^{m} \left[Pr_{LSky}(t) \times \frac{(1-Pr(t))}{Pr(t)} \right]. \end{aligned} \quad (5)$$

From Equation (5), we notice that the non-existential probability of t_1, which is equal to $1-Pr(t_1)$, is multiplied for two times. In this case, the approximate global skyline probabilities of some tuples are less than its real values. Therefore, it is inapplicable to select representative tuples of H according to their approximate global skyline probabilities when local skyline tuples have dominant relationship.

As mentioned above, all the works reviewed are unapplicable to compute the DSUD query directly. Furthermore, as analyzed in Section 2.3, the existing algorithm for DSUD queries can also be refined in three aspects. Hence, it is significative to do further study on the DSUD query.

3 Preliminaries

In this section, we give a definition of the DSUD query and introduce the objectives of the DSUD query algorithms.

3.1 The DSUD Query

Definition 3.1 (Distributed Skyline Query over Uncertain Data, DSUD Query [3]). *Assume a probabilistic threshold α $(0 \leq \alpha \leq 1)$ and a distributed site S_k for $1 \leq k \leq m$. Each site S_k processes a local skyline query over an uncertain database UDB_k, which stores N_k tuples and $\bigcup_{k=1}^{m} UDB_k = UDB$. There is also a server H to compute the global uncertain skyline answers.*

A distributed skyline query over uncertain data is to report those tuples whose global skyline probabilities are not smaller than a given threshold α at server H.

The local skyline probability of tuple t over the database UDB_i can be transformed into the following:

$$Pr_{LSky}(t, UDB_i) = \begin{cases} Pr(t) \times \prod_{t' \in UDB_i, t' \succ t} (1-Pr(t')), & t \in UDB_i; \\ \prod_{t' \in UDB_i, t' \succ t} (1-Pr(t')), & \text{otherwise.} \end{cases}$$

where m represents the number of local sites and $1 \leq i \leq m$.

Based on the traditional skyline definition, assume each local uncertain database is independent, the global skyline probability of tuple $t \in UDB_i$ is computed by

$$Pr_{GSky}(t) = Pr_{LSky}(t, UDB_i) \times \prod_{k=1}^{m} Pr_{LSky}(t, UDB_k),$$

for $k \neq i$.

The DSUD query returns the tuples whose global skyline probabilities are not less than α.

3.2 Objectives of DSUD Query Algorithms

As mentioned in [4], the main objective of distributed skyline processing is minimizing total query time. There are several main issues that affect the total query time.

- Bandwidth consumption. To reduce the network transfer time, the effective approach is to minimize the bandwidth consumption, which is measured by the number of tuples transmitted over the network.
- Local processing time. It is defined as the time caused by processing local queries. To minimize the total query time, it can be achieved by making use of efficient local indexing and adopting state-of-the-art centralized algorithms at each local site.

Moreover, in [3] and [13], they all take progressiveness as their important objects. Since the total query time may be very long, especially processing large databases, a good distributed skyline processing algorithm should return some early query results as soon as possible and produce a majority of the other results well before the end of query procedure.

The e-DSUD algorithm is progressive and with minimum bandwidth consumption [3]. However, as mentioned in Section 2.3, it does not take the total query time into consideration. Moreover, its progressiveness also has room to be improved.

4 The IDSUD Framework

In this paper, our goals are minimizing the total query time and performing better progressiveness for the DSUD query. In order to achieve these goals, motivated by the DSUD framework in [3], we propose an improved framework, IDSUD, for processing the DSDU query. There are some differences between the DSUD in [3] and our IDSUD frameworks. First, a Query-Routing phase is introduced into

IDSUD. Moreover we design a two level pruning strategy, which are site pruning in Query-Routing phase and progressively pruning strategy inside each local site or server. Second, we design an improved PR-tree (IPR-tree) to boost the DSUD query. Finally, in the To-Server phase of DSUD, each local site chooses only one representative tuple with the largest local skyline probability each time. In our IDSUD, we will use a new local tuple choosing strategy, called MPBR (See Section 5.2), to select multiple representative tuples. Specifically, the IDSUD framework is depicted in Algorithm 1.

Algorithm 1. IDSUD_Framework

Input: Local uncertain database UDB_k at local site S_k for $1 \leq k \leq m$, a probabilistic threshold α;
Output: A set *UGSky* containing all the global skylines.
1: **for** each local site S_k **do**
2: computes its local skyline set $ULSky_k$
3: sorts tuples within $ULSky_k$
4: **while** $ULSky_k$ is not empty **do**
5: **for** each local site S_k **do**
6: sends its partial abstract information to server H
7: server H prunes unqualified local sites and sends request information to the rest local sites
8: **for** each rest local site S_k **do**
9: sends its representative tuples to H
10: server H unites the representative tuples from different local sites and computes a global candidate skyline set $UGSky_{can}$
11: server H selects and feeds back its representative tuples
12: **for** each local site S_k **do**
13: computes the local skyline probability of each tuple from server H
14: refreshes its local skyline set $ULSky_k$
15: server H computes each candidate tuple's global skyline probability and adds qualified tuples to *UGSky*
16: return *UGSky*

At the beginning of IDSUD, Lines 1-3 are Local-Computation phase. In this phase, each local site S_k processes skyline query using a centralized algorithm, namely LUSQ, in Section 5.3.3. To accelerate the local skyline queries, the IPR-tree is used to organize the local data sets. The tuples within each $ULSky_k$ are sorted in decreasing order of their local skyline probabilities, respectively. At the same time, the centralized server H initializes a set *UGSky*, which includes all the global skylines lastly. The rest of the IDSUD is carried out in iterations. After introducing a new phase, called Query-Routing phase, into the IDSDU framework, each iteration consists of five phases as follows.

- Query-Routing phase (Lines 5-7). This phase is used to decide which local site can contribute to the final global skyline set. Each local site S_k generates its MPBR (See Section 5.2) and sends its partial abstract information to H. Then H prunes unqualified local sites based on Lemmas 5.2 and 5.4. Moreover, if a global skyline gaining in present iteration is from local site S_k, then S_k will be requested to attend the Query-Routing phase of the next iteration. This method is efficient to reduce bandwidth consumption.
- Local-To-Server phase (Lines 8-9). Each local site, which has not been pruned in the Query-Routing phase, sends the representative tuples within its MPBR to the server H.
- Server-Computation phase (Line 10). The server H unions all the local representative tuples, and computes the global candidate skylines by the GUSQ algorithm in Section 5.3.4. After gaining global candidate skyline tuples and adding them to a prior queue L, we arrange the tuples within it in decreasing order of their new approximate global skyline probabilities in Section 5.1. In this phase, the H also maintains a MPCS (Defined in Section 5.2), then Lemmas 5.3 could also be used to prune unqualified tuples.
- Server-Feedback phase (Line 11). The server H first generates a MPBR over the set $UGSky_{can}$. Then the representative tuples within its MPBR are broadcasted to the corresponding local sites (except the local site where the tuple comes from). These tuples are delivered to local sites for both gaining their global skyline probabilities and refreshing local skyline set $ULSky_k$.
- Local-Pruning phase (Lines 12-14). Each local site receives the global candidate skyline tuples from the server H, and maintains a MPCS for pruning the global candidate tuples form H. The unqualified ones could be expunged according to Lemmas 5.3. For the left tuples, we compute their local skyline probabilities and send them to H, respectively. Moreover, we refresh each $ULSky_k$ by pruning unqualified tuples.

In [3], it uses window query operators over each local PR-tree, respectively, for gaining local skyline probabilities of the global candidate skylines from H. Being different from the method in [3], we introduce the reuse mechanism [24] into IDSUD. In the Local-Computation phase, we store the information of nodes having been visited in a reuse heap PH_k^r at each local site S_k. We only need to execute window query operators over PH_k^r instead of accessing the local IPR-tree PR_k for many times. This method avoids unnecessary access of disk which is an effective way of reducing the total query time.

For ensuring the progressiveness of our approach, those qualified skyline tuples within *UGSky* after each iteration will be returned as a part of the final DSUD query results. Furthermore, in this iteration, while a global skyline answer is from local site S_k, it will be requested to attend the next iteration.

5 The ADSUD Algorithm

Based on the IDSUD framework in Section 4, the key components of the ADSUD are described in detail in this section. Our algorithm, ADSUD, is adaptive for the MPBR helps each local site to select the multiple representative tuples adaptively.

5.1 Local Sorting Strategies

At each local site, we sort tuples in decreasing order of their local skyline probabilities, in a way similar to the e-DSUD algorithm [3], which ranks tuples within a priority queue L at server H by the approximate global skyline probabilities.

The old definition of approximate global skyline probability in [3] is an effective way of choosing the most dominant tuple. However, as analyzed in Section 2.3, this way is not suitable when the local skyline answers have dominant relationship. Hence in this section we give a new definition of approximate global skyline probability.

Given a set *UGPrune* which includes unqualified tuples at the server H, for any tuple $t \in UDB_i$, the global skyline probability is

$$\begin{aligned} Pr_{GSky}(t) &= Pr_{LSky}(t, UDB_i) \times \prod Pr_{LSky}(t, UDB_k) \\ &\le Pr_{LSky}(t, UDB_i) \times \prod_{t' \succ t} (1 - Pr(t')) \\ &\le Pr_{LSky}(t, UDB_i) \times \prod_{t' \succ t} \Bigg[(1 - Pr(t')) \\ &\quad \times \frac{Pr_{LSky}(t', UDB_k)}{Pr(t')} \times \prod_{t'' \succ t'} \frac{1}{1 - Pr(t'')} \Bigg], \end{aligned}$$

where $t' \in UDB_k \cap L$, $t'' \in UDB_i \cap (UGSky \cup UGPrune)$, and $1 \le k \le m, k \ne i$. Here, our new approximate global skyline probability of tuple t is

$$\begin{aligned} Pr'_{NewGSky}(t) &= Pr_{LSky}(t, UDB_i) \times \prod_{t' \succ t} \Bigg[(1 - Pr(t')) \\ &\quad \times \frac{Pr_{LSky}(t', UDB_k)}{Pr(t')} \times \prod_{t'' \succ t'} \frac{1}{1 - Pr(t'')} \Bigg]. \end{aligned}$$

In the new definition above, when computing $Pr'_{NewGSky}(t)$, we find out tuple t' which dominates t from the priority queue L. Tuple t' and t are from different local sites. For improving universality of our ADUSP algorithm, we consider the relationship between t' and tuples t'', and adjust the old definition in [3].

In the Server-Feedback phase, server H first computes $Pr'_{NewGSky}(t)$ of each tuple within L. If $Pr'_{NewGSky}(t) \le \alpha$, tuple t is not a global skyline and is deleted from L. The left tuples within L are arranged in decreasing order of their new approximate global skyline probabilities.

5.2 Minimum Probabilistic Bounding Rectangle

A good distributed skyline algorithm should be able to transfer unqualified skyline points as less as possible [3], [4]. Hence a local skyline selection strategy plays an important role in a distributed skyline query algorithm.

In this section, we give a definition of the MPBR. Our MPBR is different from the MBR of R-tree in two aspects. First, the MBR is usually created by clustering the near points while our MPBR is generated according to the probability threshold. Second, the MBR is utilized in the Local-Computation phase for improving the pruning capacity. Our MPBR is used to help choosing the local multiple representative tuples which will be sent to the server in the Local-To-Server phase, and helps to gain the abstracted information for site pruning in the Query-Routing phase.

Definition 5.1 (Minimum Probabilistic Bounding Rectangle (MPBR)). *A MPBR* BR *consists of the minimum set of tuples that satisfy the condition*

$$Pr_{nonexist}(BR) = \prod_{t_j \in BR} (1 - Pr(t_j)) < \alpha.$$

Let t_{min} and t_{max} be the minimum and maximum corner of a BR. We have

$$\begin{aligned} t_{min} &= (min_{t_i \in BR} t_i.x_1, min_{t_i \in BR} t_i.x_2, \ldots, min_{t_i \in BR} t_i.x_d), \\ t_{max} &= (max_{t_i \in BR} t_i.x_1, max_{t_i \in BR} t_i.x_2, \ldots, max_{t_i \in BR} t_i.x_d), \end{aligned}$$

and $Pr(t_{min}) = max_{t_i \in BR} Pr(t_i)$, $Pr(t_{max}) = min_{t_i \in BR} Pr(t_i)$.

Note that, we could use two-tuples, (t_{min}, t_{max}), to denote a MPBR.

Suppose that site S_1 gets a local skyline set $\{a, b, c, d, e\}$ and $\alpha = 0.3$, we choose points a, b and c to generate a MPBR for $(1 - Pr(a)) \times (1 - Pr(b)) \times (1 - Pr(c)) = (1 - 0.5) \times (1 - 0.4) \times (1 - 0.3) = 0.21 \le 0.3$.

Definition 5.2 (MPBR-dominance). *Given two MPBRs $BR = (t_{min}, t_{max})$ and $BR' = (t'_{min}, t'_{max})$, we have $BR \succ BR'$, if $t_{max} \succ t'_{min}$.*

Lemma 5.1. *Given a MPBR $BR = (t_{min}, t_{max})$ and a tuple t, if $t_{max} \succ t$, then t can be safely pruned.*

Proof. Since $t' \succ t_{max}$, for any tuple $t' \in BR$, if $t_{max} \succ t$, we have $t' \succ t$ according to the transmit property of dominant relationship. Therefore $BR \succ t$. Because $Pr_{nonexist}(BR) \le \alpha$, we get $Pr_{GSky}(t) \le Pr(t) \times Pr_{nonexist}(BR) < Pr(t) \times \alpha$. Considering $0 \le Pr(t) \le 1$, we have $Pr_{GSky}(t) < \alpha$. Hence tuple t is not a local skyline answer and can be safely pruned. □

Based on Lemma 5.1 we also design a new lemma used to local site pruning in the Query-Routing phase of the IDSUD.

Lemma 5.2. *Given two MPBRs $BR = (t_{min}, t_{max})$ and $BR' = (t'_{min}, t'_{max})$, if $BR \succ BR'$, then the tuples contained in BR' can be safely pruned.*

Proof. Since $BR \succ BR'$, we have $t_{max} \succ t'_{min}$ due to Definition 5.2. For each tuple $t \in BR'$, we have $t_{max} \succ t$, which means tuple t can be safely pruned according to Lemma 5.1. Since t represents any tuple within BR', it holds that the tuples within BR' are not local skyline results. Hence, the lemma holds. □

Note that it only takes a single MPBR into account in the lemmas above. For the purpose of getting much higher pruning power, we design some new lemmas, which take a set of MPBRs into account, instead of a single one.

Definition 5.3 (MPBR Constrained Space (MPCS)). *For a MPBR set $BRS = \{1 \le i \le |BRS| | BR_i = (t^i_{min}, t^i_{max})\}$, its MPCS consists of the union of all the regions which are dominated by t^i_{max}.*

Based on the property of MPCS, we get following lemmas.

Lemma 5.3. *Given a MPCS CS of a MPBR set $BRS = \{1 \le i \le |BRS| | BR_i = (t^i_{min}, t^i_{max})\}$ and a tuple t, if $t \in CS$, the tuple t can be safely pruned.*

Proof. Assume a MPBR $BR_i = (t^i_{min}, t^i_{max})$ and $BR_i \in BRS$, since $t \in CS$, from Definition 5.3, we have that tuple t is

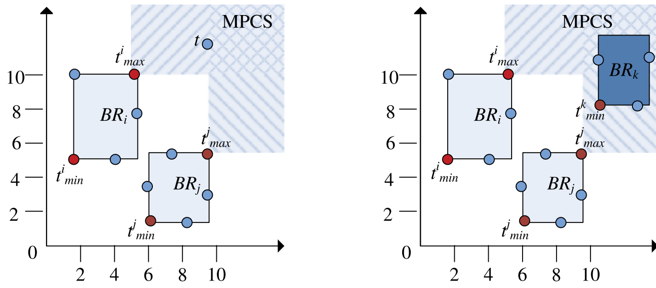


Fig. 2. Illustration of MPCS.

dominated by BR_i. It holds that $t^i_{max} \succ t$. Therefore tuple t is not a query result according to Lemma 5.1 and this lemma holds. □

Based on Lemma 5.3, if a global candidate tuple t, which is fed back from server H, falls in one of the local site's MPCS, then t is not a global skyline. Moreover, suppose server H receives a local skyline tuple t, it is not a global skyline answer if it falls into the MPCS at server H.

On the basis of Lemma 5.3, we also propose a new lemma for local site pruning in the Query-Routing phase of IDSUD in Section 4.

Lemma 5.4. *Given a MPCS* CS *and a MPBR* $BR = (t_{min}, t_{max})$, *if* $BR \in CS$, *the tuples contained in* BR *can be safely pruned.*

Proof. Since $BR \in CS$, for any tuple $t \in BR$, we have $t \in CS$. Therefore tuple t is not a local skyline according to Lemma 5.3. Since tuple t represents any tuple contained in BR, this lemma holds. □

Fig. 2 shows an example of MPCS. As shown in Fig. 4, if a tuple t falls into a MPCS, it is not a skyline answer according to Lemma 5.3. Taking a MPBR BR_k into consideration, when it falls into a MPCS, tuples within it can be pruned safely due to Lemma 5.4.

Assume a MPBR BR_i is received from local site S_i, if BR_i falls in the MPCS of server H, then BR_i is unqualified based on Lemma 5.4.

5.3 The Local Algorithms

To minimize the total query time, it is partially achieved by minimizing the individual local processing time, which is accomplished by adopting state-of-the-art centralized algorithms and making use of efficient local indexing at each local site. Therefore, in this section, we first introduce the reuse technology which is an effective way of reducing the I/O cost. Then based on the PR-tree introduced in [3], we present a more efficient index, namely IPR-tree. Last, two new local centralized algorithms, LUSQ and GUSQ are designed.

5.3.1 The Reuse Mechanism

The reusing technology in [24] is an effective way for reducing the redundant I/O operations. Recently, it has been successfully used in two famous skyline query variants, which are monochromatic and bichromatic mutual skyline queries [24] and reverse skyline query over certain data [25]. Therefore, for minimizing the local processing time at each local site, we utilize the reusing technology to boost the performance of the DSUD query.

As illustrated in [24] and [25], when processing reverse skyline query, it usually computes skyline answers first. Then for each skyline answer, the traditional approaches usually apply the window query over the R-tree to check wether it is a reverse skyline result. This incurs redundant I/O cost because of visiting the same nodes for many times. In the effective way of applying the reuse mechanism, it stores the nodes having been examined into a reuse heap.

5.3.2 The IPR-Tree

In order to improve the query efficiency, indices are built on uncertain database as usual [3], [4]. It could sharply reduce the processing time through using index technology. Therefore, in this section we adjust the PR-tree proposed in [3] and introduce an improved PR-tree, called IPR-tree.

According to the definition of skyline probability in Section 3.1, the skyline probability of each tuple depends on two issues, which are its existential probability and non-existential probability of each entry dominates it. Therefore, in our IPR-tree, each intermediate entry contains non-existential probability of its child entries instead of the minimum probability in the PR-tree [3]. The non-existential probability of an intermediate entry PE is computed by

$$Pr_{nonexist}(PE) = \begin{cases} \prod_{PE_i \in Child(PE)} (1 - Pr(PE_i)), & \text{for a leaf entry;} \\ \prod_{PE_i \in Child(PE)} Pr_{nonexist}(PE_i), & \text{otherwise.} \end{cases}$$

Fig. 3 shows a general example of an IPR-tree. Let the capacity of a minimum bounding rectangle (MBR) be equal to 3. The existential probability of tuples a, b, and c are 0.4, 0.5, 0.6, respectively. Hence, the maximize probability of entry PE_3 is equal to $max(Pr(a), Pr(b), Pr(c)) = 0.6$. The non-existential probability of PE_3 is computed by

$$Pr_{nonexist}(PE_3) = (1 - Pr(a)) \times (1 - Pr(b)) \times (1 - Pr(c)) = 0.12.$$

Consider PE_1 to be an intermediate entry. It stores the probabilities of $max(Pr_{max}(PE_3), Pr_{max}(PE_4)) = 0.6$ and its non-existential probability which is equal to $Pr_{nonexist}(PE_3) \times Pr_{nonexist}(PE_4) = 0.04032$.

5.3.3 LUSQ Algorithm

At each local site S_k, we organize the local uncertain databases UDB_k by the IPR-tree in Section 5.3.2. Then we design a local site uncertain skyline query (LUSQ) algorithm to compute skylines at each local site. In LUSQ, we use a progressive pruning strategy to reduce the search space.

Given a probability threshold α $(0 \leq \alpha \leq 1)$, based on the IPR-tree, we have the following lemmas.

Lemma 5.5. *Given an entity E, if $Pr_{max}(E) < \alpha$, then tuples within E can be safely pruned.*

Proof. Taking each tuple $t \in E$ into account, we have $Pr(t) \leq Pr_{max}(E)$. Since $Pr_{max}(E) < \alpha$, we get $Pr(t) < \alpha$. Due to Equation (2) in Section 3.1, it is obvious that $Pr_{LSky}(t_i) < Pr(t) < \alpha$. Since t_i represents any tuple within E, this lemma holds. □

Lemma 5.6. *Given two entries E_i and E_j, if $E_i \succ E_j$ and $Pr_{max}(E_j) \times Pr_{nonexist}(E_i) < \alpha$, the tuples contained in E_j can be safely pruned.*

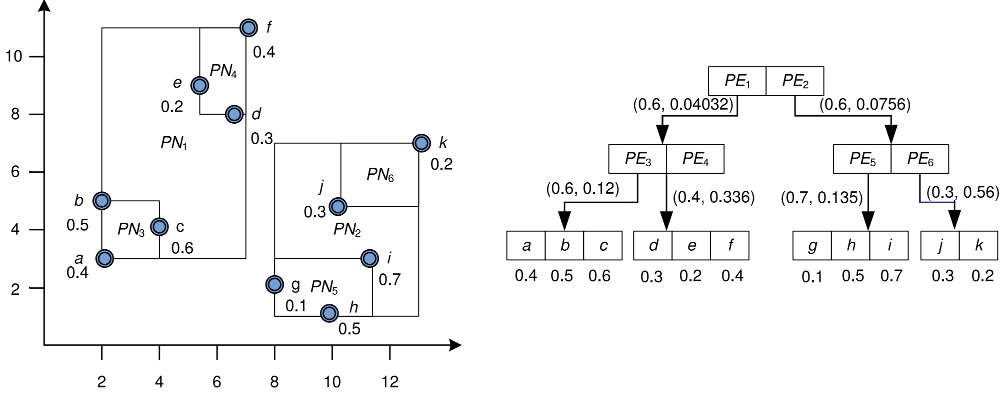


Fig. 3. Example of an IPR-tree.

Proof. Since $\exists t \in E_j$, it is obvious that $Pr(t) \leq Pr_{max}(E_j)$. For $Pr_{max}(E_j) \times Pr_{nonexist}(E_i) < \alpha$ and $E_i \succ E_j$, we have

$$\begin{aligned} Pr_{LSky}(t) &\leq Pr_{max}(E_j) \times (1 - Pr(E_i)) \\ &\leq Pr_{max}(E_j) \times Pr_{nonexist}(E_i) < \alpha. \end{aligned}$$

Consider t represents any tuple in E_j and $Pr_{LSky}(t) < \alpha$. we can get the lemma obviously. □

For the sake of pruning unqualified tuples as early as possible, we introduce a progressive pruning strategy. In general, given a local candidate skyline tuple set $ULSky^k_{can}$ at local site S_k, with accessing entries in order of their distance to the origin, the upper bound of local non-dominated probability of entry E, $Pr_{UBLDom}(E)$, can be computed by

$$Pr_{UBLDom}(E) = \prod_{t_u \succ E, t_u \in ULSky^k_{can}} (1 - Pr(t_u)).$$

Therefore, the upper bound of local skyline probability of E is equal to

$$Pr_{UBLSky}(E) = Pr_{max}(E) \times Pr_{UBLDom}(E).$$

Lemma 5.7. *Given an entry E, in case $Pr_{UBLSky}(E) < \alpha$, tuples within E can be safely pruned.*

Proof. For $\exists t \in E$, we have

$$\begin{aligned} Pr_{LSky}(t) &\leq Pr(t) \times \prod_{t_u \succ E, t_u \in ULSky^i_{can}} (1 - Pr(t_u)) \\ &\leq Pr_{max}(E) \times \prod_{t_u \succ E, t_u \in ULSky^i_{can}} (1 - Pr(t_u)) \\ &= Pr_{UBLSky}(E). \end{aligned}$$

Since $Pr_{UBLSky}(E) < \alpha$, it holds that $Pr_{LSky}(t) < \alpha$. Hence tuple t is not a skyline. Considering tuple t represents any tuple within E, the lemma holds. □

Given a set of examined unqualified skyline points, $ULSky^k_{Ref}$, which has contribution to prune the unexamined points, we have the following observation.

Observation 5.1. Given a tuple t_i, any tuple t_j dominated by t_i must be at least dominated by the same tuples.

This observation is clear. For any tuple t_v that dominates t_i, since $t_i \succ t_j$, it holds that $t_v \succ t_j$ due to the transitivity of dominant relationship.

Lemma 5.8. *Consider any visited tuple t which is unqualified. If $(1 - Pr(t_u)) \times Pr_{UBLDom}(t_u) < \alpha$, then t_u may have contribution to prune the unexamined points. Therefore, we add it to $ULSky^i_{Ref}$.*

Proof. Given an unexamined tuple t_v that is dominated by t_u, the local skyline probability of t_v is equal to

$$\begin{aligned} Pr_{LSky}(t_v) &= Pr(t_v) \times \prod_{t \in UDB_i, t \succ t_v} (1 - Pr(t)) \\ &\leq Pr(t_v) \times (1 - Pr(t_u)) \times \prod_{t \succ t_u, t \in UDB_i} (1 - Pr(t)) \\ &= Pr(t_v) \times (1 - Pr(t_u)) \times Pr_{UBLDom}(t_u). \end{aligned}$$

Since $(1 - Pr(t_u)) \times Pr_{UBLSky}(t_u) < \alpha$, we have $Pr_{LSky}(t_v) < \alpha$. Therefore, the lemma holds. □

With respect to $ULSky^k_{can}$ and $ULocal^k_{Ref}$, we have

$$\begin{aligned} Pr'_{UBLDom}(E) &= \prod_{\substack{t \succ E \\ t \in ULSky^k_{can}}} (1 - Pr(t)) \times \prod_{\substack{t \succ E \\ t \in ULSky^k_{Ref}}} (1 - Pr(t)) \\ &= Pr_{UBLDom}(E) \times \prod_{\substack{t \succ E \\ t \in ULSky^k_{Ref}}} (1 - Pr(t)). \end{aligned}$$

Considering $Pr'_{UBLDom}(E)$, we have the following lemma.

Lemma 5.9. *Given an entry E, it can be safely pruned if $Pr(E) \times Pr'_{UBLDom}(E) < \alpha$.*

The proof of Lemma 5.9 is similar to Lemma 5.7.

Based on the above lemmas, the LUSQ algorithm is proposed to compute the local skyline set efficiently. The LUSQ employs the IPR-tree to organize the data set. The pseudocode of this LUSQ algorithm is depicted in Algorithm 2.

It is composed of two phases, which are pruning phase and refining phase. In the pruning phase, it identifies the unqualified skyline tuples on the basis of Lemmas 5.5-5.9. And a reuse min-heap PH^r_k is defined besides a min-heap PH_k. It is used to store entries that have been visited and play a key role in decreasing redundant I/O operators. According to Observation 5.1 and Lemma 5.8, we add the

unqualified tuple t to the set $ULSky^k_{Ref}$, if t has contribution to prune other unvisited tuples. Since our pruning strategies can guarantee that all the pruned tuples are not query results, the candidate set $ULSky^k_{can}$ after pruning would contain all the query results. However, there are also some tuples that should be unqualified query results still exist in $ULSky^k_{can}$. Therefore, in the refining phase, for each tuple t within $ULSky^k_{can}$, we use the window query operator over PH^r_k to find out all the tuples that dominate it and compute its local skyline probability incrementally. Since PH^r_k stores all the index information of each tuple at different levels in its entirety (See Theorem 5.10), it is not required to visit the IPR-tree IPR_k for more times but to reuse the information stored in PH^r_k.

Algorithm 2. LUSQ_Algorithm

Input: An uncertain local data set UDB_k, an IPR-tree IPR_k, and a probability threshold α.
Output: A set $ULSky_k$ containing all the local skylines.

1: Initialize a min-heap $PH_k = \emptyset$, a reuse heap $PH^r_k = \emptyset$, a candidate skyline set $ULSky^k_{can} = \emptyset$, and a refine set $ULSky^k_{Ref} = \emptyset$
2: Insert the entries in the root of IPR into PH_k and PH^r_k respectively
3: **while** PH_k is not empty **do**
4: Remove top entry E from PH_k
5: **if** E is a leaf node **then**
6: **if** $Pr(E) < \alpha$ **then**
7: **if** $(Pr_{UBLSky}(E) < \alpha) \wedge ((1-Pr(E)) \times Pr_{UBLSky}(E) \leq \alpha)$ **then**
8: Add E to $ULSky^k_{Ref}$ /* by Lemmas 5.5, 5.7 and 5.8 */
9: **else**
10: **if** $(Pr(E) \times Pr_{UBLDom}(E) < \alpha) \vee (Pr(E) \times Pr'_{UBLDom}(E) < \alpha)$ **then**
11: **if** $(1-Pr(E)) \times PrUBLSky(E) \leq \alpha$ **then**
12: Add E to $ULSky^k_{Ref}$ /* by Lemmas 5.7, 5.8 and 5.9 */
13: **else**
14: Make E as false alarm
15: **else**
16: Add E to $ULSky^k_{Can}$
17: **else**
18: Remove top entry E from PH^r_k
19: **for** each children E_i of E **do**
20: Insert E_i into PH^r_k
21: **if** $(Pr_{max}(E_i) < \alpha) \| (Pr_{max}(E_i) \times Pr_{UBLSky}(E_i) < \alpha) \| (Pr_{max}(E_i) \times Pr'_{UBLSky}(E_i) < \alpha)$ **then**
22: Make E_i as false alarm /* by Lemmas 5.5 and 5.7 */
23: **else**
24: Insert E_i into PH_k
25: Prune all unqualified entries in PH_k by Lemma 5.6
26: Refine $ULSky^k_{can}$ by invoking the window query over PH^r_k and add the local skylines to $ULSky_k$
27: Return $ULSky_k$

Complexity. The LUSQ algorithm includes two phases which are the pruning phase and refining phase. In the prune phase, suppose that the height of the IPR-tree IPR_k is $\hbar$ and the average access cost of visiting a node is ∂. The node access cost by LUSQ is $\hbar\partial H(d, N/m)$ at most. In the refining phase, window query operators over the reuse heap are used to compute each candidate tuple's accurate local skyline probability. Assume that the average cost of a window query operator is δ. The total cost of refining all the candidates is $\delta H(d, N/m)$. Therefore, the total cost of LUSQ is $O((\hbar\partial+\delta)H(d, N/m))$. Due to Equation (1), we have the complexity of the LUSQ as

$$O\left((\hbar\partial+\delta) \times \sum_{n=0}^{N/m} \frac{1}{d!}(\ln n)^{d-1} \times P(n)\right).$$

5.3.4 The GUSQ Algorithm

In the Server-Computation phase, H unites all the representative tuples from local sites and stores the ones, which cannot be pruned by Lemmas 5.1 and 5.3, in an uncertain database UDB_0. Then, we design a global skyline query algorithm, called GUSQ algorithm, to gain global candidate skylines.

Algorithm 3. GUSQ_Algorithm

Input: An uncertain local data set UDB_0 containing the tuples from local sites and a probability threshold α.
Output: A set $UGSky_{can}$ containing all the global candidate skylines.

1: Sort the left tuples within UDB_0 in decreasing order of the minimum attribute value of each tuple
2: **for** each tuple $t \in UDB_0$ **do**
3: Access the tuples within UDB_0 which are sorted before t, and compute the local skyline probability of the tuple t
4: **if** $Pr_{LSky}(t) \geq \alpha$ **then**
5: Add t to a prior queue L
6: Return $UGSky_{can}$

In the GUSQ algorithm, it prunes the unqualified tuples within UDB_0 and gains a global candidate skyline set $UGSky_{can}$. We choose the function $min^d_{i=1}(t.a_i)$ as the monotonic function to sort the tuples within UDB_0 and use $\sum^d_{i=1} t.a_i$ as the tie breaker. Therefore, we can get $Pr_{LSky}(t)$ for each left tuple $t \in UDB_0$ through accessing the tuples before it [2].

Complexity. In the Server-Computation phase, to get the local skyline probability of each tuple $t \in UDB_0$, it is necessary to access all the tuples sorted before t. In the worst case, the complexity of GUSQ algorithm is computed by $1+2+\cdots+(|UDB_0|-1) = \frac{1}{2}(|UDB_0|^2 - |UDB_0|)$. The complexity of GUSQ algorithm is $O(|UDB_0|^2)$. It is obviously that $|UDB_0| = m \times H(d, N/m)$. Therefore, according to Equation (1) in [3], the complexity of GUSQ algorithm is

$$O\left(m^2 \times \left(\sum_{n=0}^{N/m} \frac{1}{d!}(\ln n)^{d-1} \times P(n)\right)^2\right).$$

Furthermore, to get the accurate global skylines, it is necessary to refine tuples within $UGSky_{can}$ though computing their global skyline probabilities. In order to reduce the bandwidth cost and get good progressiveness, we sort the tuples within $UGSky_{can}$ in decreasing order of their new approximate global skyline probabilities defined in Section 5.1. Then we generate a MPBR over $UGSky_{can}$ and

TABLE 2
Performance Metrics

Metric	Measurement
I/O cost	Average number of nodes accesses (AN)
Bandwidth cost	Number of tuples transmitted over the network (NT)
Total query time	Total time between submitting the quest and returning all the answers (QT)
Progressiveness	Bandwidth cost as a function of the number of reported skyline points

feed back the representative tuples within it to local sites for gaining their global skyline probabilities.

Discussion. In a distributed system which contains only one server, the server computation phase may be the bottleneck of our ADSUD. In this case, we could expand the traditional distributed system by using a server cluster. The local sites are divided into several groups, and all the local sites within a group connect with the same server. In the server-computation phase, all the servers within the server cluster work cooperatively to calculate the global candidate skylines and compute their accurate global skyline probabilities.

5.4 Performance Analysis of ADSUD Algorithm

In this section, we certify that the ADSUD algorithm is I/O efficient and correct.

Theorem 5.10. *The reuse min-heap* PH_k^r *can ensure the integrity of each local PR-tree* IPR_k *for* $1 \leq k \leq m$.

For clarity and readability, the detailed proofs of the theorems in this section are moved to the supplement.

Theorem 5.11. *Compared to the e-DSUD algorithm proposed in [3], our ADSUD algorithm is more I/O efficient.*

Theorem 5.12. *Any tuple contained in UGSky after the execution of ADSUQ algorithm is guaranteed to be a final global skyline answer.*

6 Performance Evaluation

In this section, we verify the efficiency and progressiveness of our proposed ADSUD algorithm. Our experiments use both the synthetic datasets and a real-life data set. Since ADSUD is motivated by e-DSUD in [3], we consider e-DSUD as the baseline algorithm, and compare ADSUD with it.

In [3], it only takes bandwidth consumption and progressiveness as its performance measurements. In the following experiments, we compare ADSUD against e-DSUD with considering not only the bandwidth cost, progressiveness, but also the total query time and the I/O cost. The main performance metrics are summarized in Table 2.

Besides, we analyze the four metrics shown in Table 2in the following aspects: (1) the number of local sites m; (2) the dimensionality of the datasets d; (3) the probability threshold α; (4) the total cardinality of datasets N.

The experiments were performed on a cluster composed of 10 servers. Each server has two Intel Xeon E52667 3.3 GHz CPUs (each CPU contains 8 cores), and 64 GB main memory, and runs the 64bit Microsoft Windows 7 operating system. All algorithms were implemented in VC++ and the page sizes of PR-tree and IPR-tree are both 4,096 Bytes. The reported results are an average of 50 queries on the condition of the same system parameters.

TABLE 3
System Parameters

Parameter	Values
The number of local site (m)	20, **40**, 60, 80, 100
Dimensionality(d)	2, **3**, 4, 5
Probability threshold (α)	**0.3**, 0.5, 0.7, 0.9
Cardinality (N)	**2,000 K**, 4,000 K, 6,000 K, 8,000 K, 10,000 K

6.1 Experiments on Synthetic Datasets

In order to study the scalability of both ADSUD and e-DSUD, we first do experiments on the synthetic datasets with two popular distributions: Independent (Ind) and Anti-correlated (Ant). Specifically, for the Ind data set, all attribute values are generated independently using a uniform distribution; for the Ant data set, if a point has a small coordinate on one dimension, then it tends to have a large coordinate on at least another dimension [12]. Similar to [3], we use uniform distribution to randomly generate an existential probability of each tuple to make them be uncertain.

In this section, we investigate the performance of our ADSUD by comparing it with e-DSUD under different parameters. The parameters of the experiments are listed in Table 3, and the default parameter values are given in bold. Note that, in each experiment, only one parameter varies and the others are fixed to their default values.

6.1.1 ADSUD Performance versus Number of Local Sites m

In the first set of experiments, we examine the effect of the number of local sites m on the performance of the DSUD query. The number of local sites m varies from 20 to 100 by a step of 20 and the other parameters kept to their default values.

Moreover, in our experiments, to simulate 20, 40, 80, 60, 100 computing nodes, it decomposes a server to 2, 4, 6, 8, and 10 virtual computing nodes, respectively.

Figs. 4a and 4b depict that QT and AN of the two algorithms, e-DSUD and our ADSUD, get smaller as m grows. This is because the size of each local database decreases with m grows. It is obvious that it needs less AN and local query time for a smaller local database, which reduces the QT in turn. As shown in Fig. 4c, the NT raises with m increases. Since in the Server-Feedback phase, the tuples selected from the server H are broadcasted to the corresponding local sites (except the local site where the tuple comes from). Thus, it is necessary to make $m{-}1$ copies of each representative tuple and send them to different local sites, respectively. Due to Equation (1), the total number of final skyline tuples, $H(d,N)$, is fixed according to d and N. Therefore, the total number of tuples delivered from the server H is $(m{-}1)H(d,N)$ at least. It is easy to draw a conclusion that the larger the number of local sites m, the more the network bandwidth cost needs.

Fig. 4 illustrates that the performance of our ADSUD is much better than that of e-DSUD as m grows. For Ind

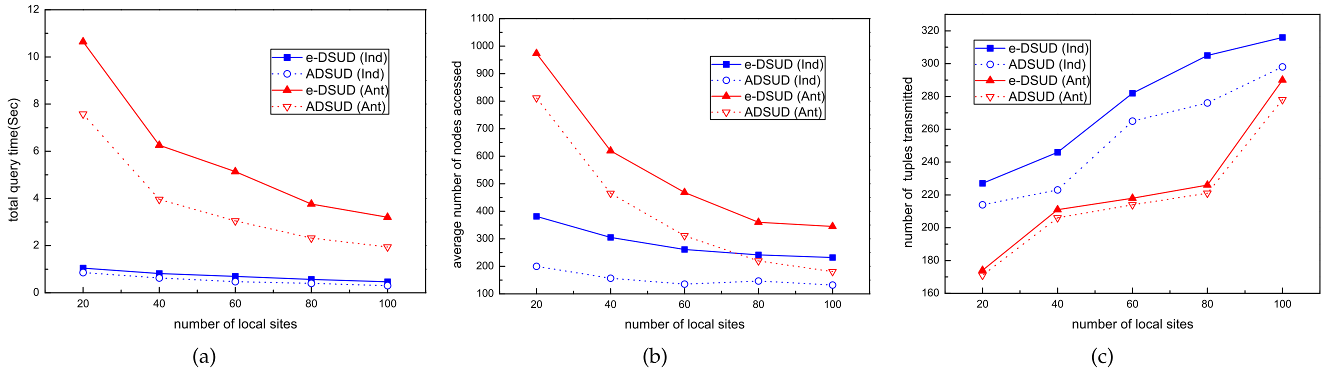


Fig. 4. Performance versus number of local sites m. (a) QT. (b) AN. (c) NT.

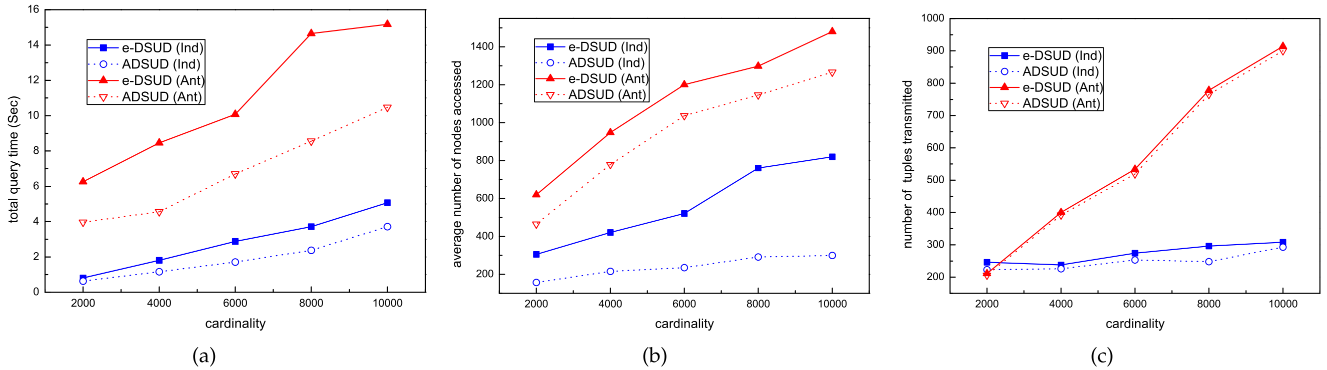


Fig. 5. Performance versus cardinality N. (a) QT. (b) AN. (c) NT.

datasets, ADSUD can decrease 35.13 percent QT, 48.56 percent AN, and 9.51 percent NT, separately. Considering the Ant datasets, our ADSUD algorithm can cut down 40.66 percent QT, 47.47 percent AN, and 4.14 percent NT at most.

Obviously, the performance benefit of NT is much less than that of QT or AN. This is because in e-DSUD each local site delivers only one representative tuple each time, while in the Local-To-Server phase of ADSUD, each local site delivers multiple representative tuples to the server H. Moreover, in the Server-Feedback phase at each iteration, H chooses several global candidate tuples and delivers it to local sites. For each global candidate tuple within H, it needs to generate $m-1$ copies and broadcast them to local sites, separately. In some cases, the NT of our ADSUD may a little larger than that of e-DSUD. It is worth to notice that, in most cases, the NT of our ADSUD is less than that of e-DSUD. Moreover, our ADSUD always outperforms e-DSUD with considering the overall performance in terms of QT, AN, and NT.

6.1.2 ADSUD Performance versus Cardinality N

In the second set of experiments, we examine the effect of the cardinality N on the performance of the DSUD query. The cardinality N varies from 2,000 to 10,000 K by a step of 2,000 K and the other parameters kept to their default values.

Fig. 5 shows the results by varying the cardinality N. Obviously, the QT, AN, and NT of our ADSUD and e-DSUD grow when N gets larger. This is expected, because the local query time and communication time increase as N becomes larger, and the size of the global skylines ascend as the growth of N which makes AN and NT grow. In addition, our ADSUD outperforms e-DSUD in all cases with the growth of N. For Ind datasets, ADSUD can decrease 40.67 percent QT, 63.53 percent AN, and 16.21 percent NT, separately. Considering the Ant datasets, our ADSUD can cut down 46.11 percent QT, 24.91 percent AN, and 2.81 percent NT at most.

6.1.3 ADSUD Performance versus Dimensionality d

In the third set of experiments, we study the performance of ADSUD with d varying from 2 to 5 by a step of 1, and the other parameters are kept to their default values.

The efficiency of the algorithms under various d is depicted in Tables 4 and 5, where QT, AN, and NT are reported, respectively. As expected, the performance of the two algorithms, e-DSUD and our ADSUD, both reduce sharply with the growth of d. It has two reasons. The first one is in a high-dimensional space, each tuple has a low probability of being dominated by other ones, which makes the final skyline set become larger. More query results lead to higher bandwidth cost. The second one is the poor performance of the R-tree in high dimensions, which incurs the algorithms to visit more entries of local PR-tree/IPR-tree.

TABLE 4
Experimental Results versus Dimensionality d (Ind)

d	e-DSUD			ADSUD		
	QT(sec)	AN	NT	QT(sec)	AN	NT
2	0.339	77	68	0.250	38	62
3	0.813	305	246	0.627	157	223
4	3.634	1,308	998	2.240	201	692
5	6.988	7,225	3,464	4.323	411	3,431

TABLE 5
Experimental Results versus Dimensionality d (Ant)

d	e-DSUD			ADSUD		
	QT(sec)	AN	NT	QT(sec)	AN	NT
2	0.126	24	46	0.088	21	40
3	6.258	619	211	3.960	465	206
4	20.715	4,304	2,496	9.029	767	2,478
5	118.567	29,946	14,005	64.648	859	13,932

Obviously, the performance of ADSUD is also much better than that of e-DSUD. Especially, for Ind datasets, it could reduce 38.36 percent QT, 94.30 percent AN, and 9.35 percent NT by utilizing our ADSUD. Taking the Ant datasets into account, applying the ADSUD can decrease 56.41 percent QT, 97.01 percent AN, and 13.04 percent NT.

6.1.4 *ADSUD Performance versus Threshold* α

In the fourth set of experiments, we explore the impact of threshold α on the performance of the algorithms. Specifically, α varies from 0.3 to 0.9 by a step of 0.2, and the other parameters are kept to their default values.

Fig. 6 illustrates the experimental results when we vary α from 0.3 to 0.9, under the Ind and Ant datasets. As α increases, the performance of the two algorithms both become better. The reason is the size of the global skylines is sensitive to the probability threshold α. According to the definition of P-skyline query in Section 2, the larger the probability threshold, the smaller the global skyline sets. So the NT reduces as α raises. Moreover, with the growth of α, we can prune much more unqualified skyline tuples whose global skyline probabilities are less than it. Since larger α gains better pruning ability, the QT and AN are both cut down in turn. In addition, comparing to the e-DSUD, for Ind datasets, our ADSUD reduces 30.51 percent QT, 48.56 percent AN, and 29.21 percent NT in the best case. Consider the Ant datasets. Our algorithm can decrease 42.37 percent QT, 24.91 percent AN, and 10.81 percent NT at most separately.

6.2 Experiments on Real Datasets

In this section, we evaluate our proposed algorithm over a real dataset, Household (Hous) [26] (available at http://www.ipums.org/). It includes 127,000 tuples about the percentage of an American family's annual income. We consider four attributes, which are the expenditures of gas, electricity, water, and heating.

Figs. 7 and 8 show that the results on the real data set are consistent with the ones obtained from the experiments on the synthesis datasets. The performance of ADSUD is much better than e-DSUD on the real data set. As m increases, comparing to the e-DSUD, our ADSUD can reduce 61.76 percent QT, 90.23 percent AN, and 4.37 percent NT at most separately. Comparing to the e-DSUD, for processing the real dataset, our ADSUD decreases 58.93 percent QT, 82.40 percent AN, and 14.57 percent NT with the growth of α.

6.3 Progressiveness Performance

In this set of experiments, we evaluate the progressiveness of our proposed algorithm over synthetic datasets and the real dataset (Hous) in Section 6.2.

We use the synthetic datasets, which are under Ind and Ant distributions with $d = 5$ and the other parameters equal to their default values in Table 4.

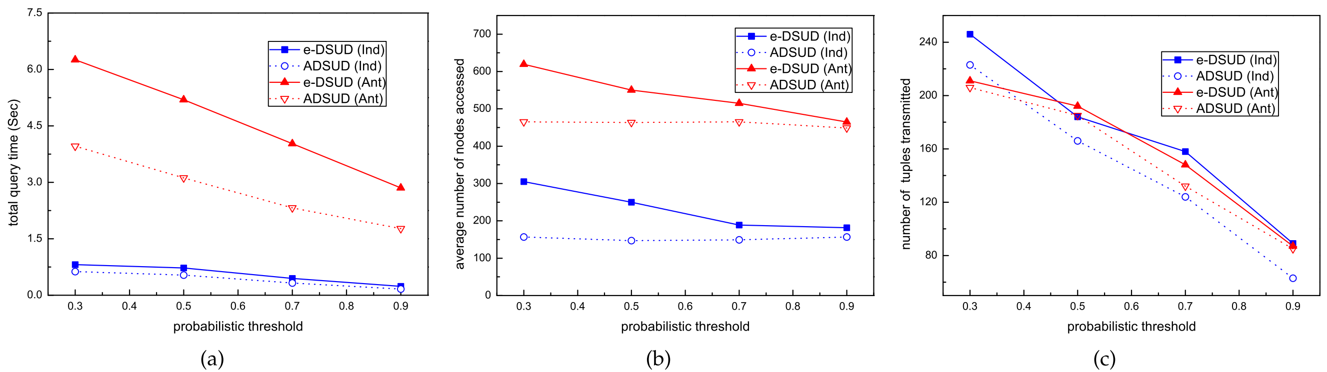


Fig. 6. Performance versus probabilistic threshold α. (a) QT. (b) AN. (c) NT.

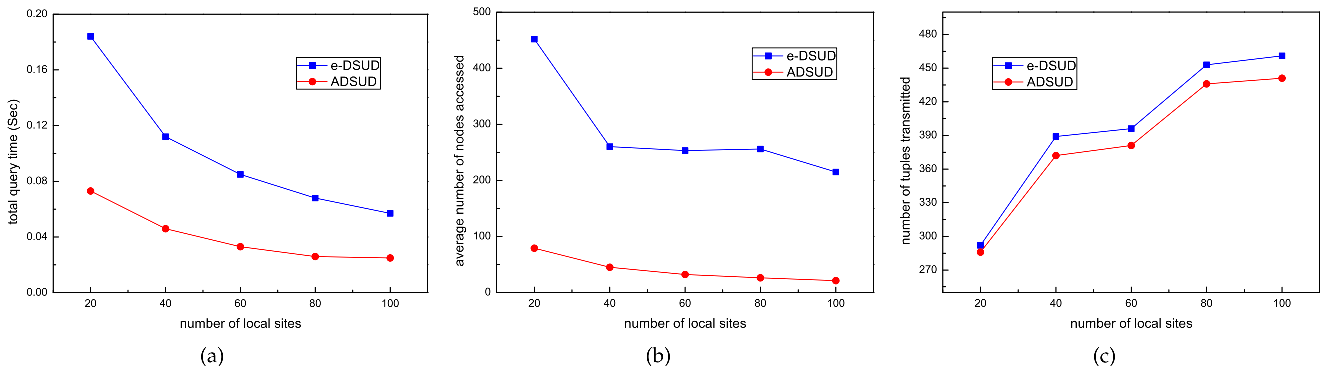


Fig. 7. Performance on real dataset (Hous) versus number of local sites m. (a) QT. (b) AN. (c) NT.

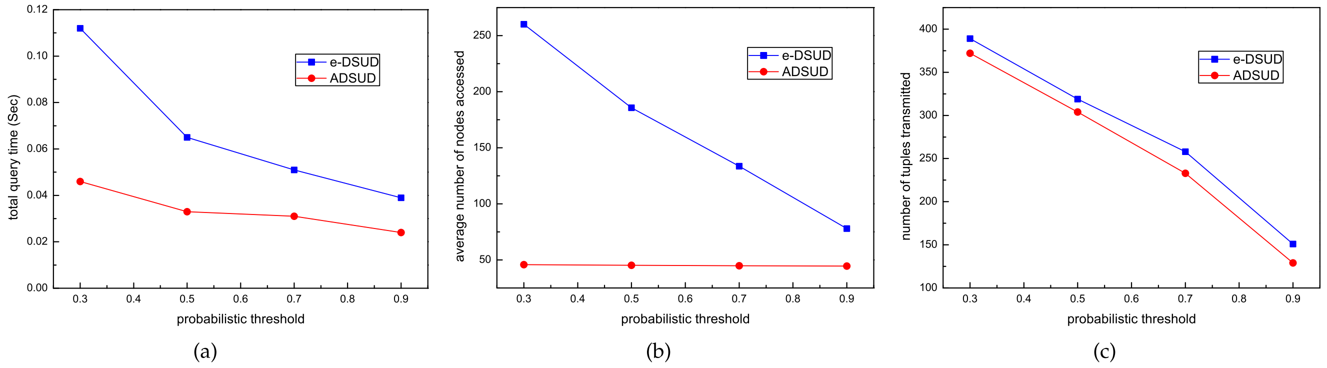


Fig. 8. Performance on real dataset (Hous) versus probabilistic threshold α. (a) QT. (b) AN. (c) NT.

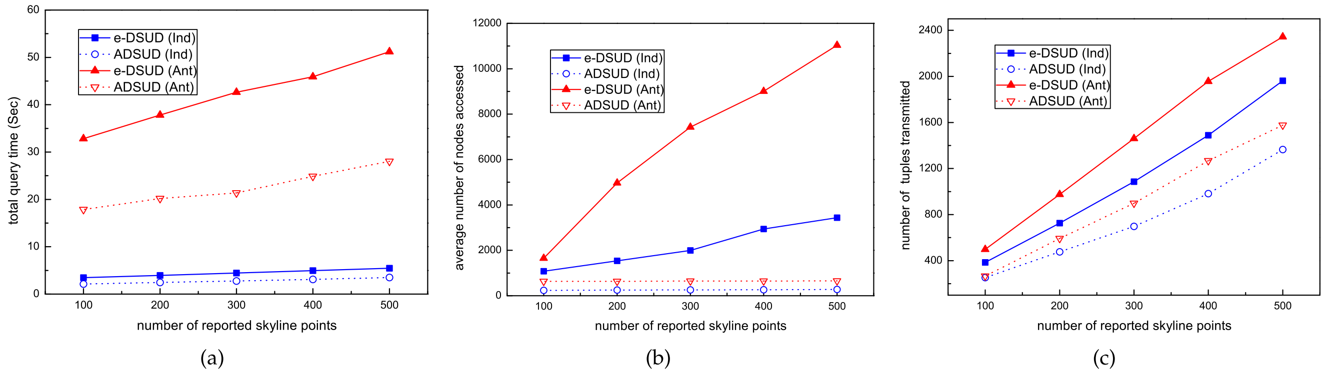


Fig. 9. Progressiveness comparison versus synthetic datasets. (a) QT. (b) AN. (c) NT.

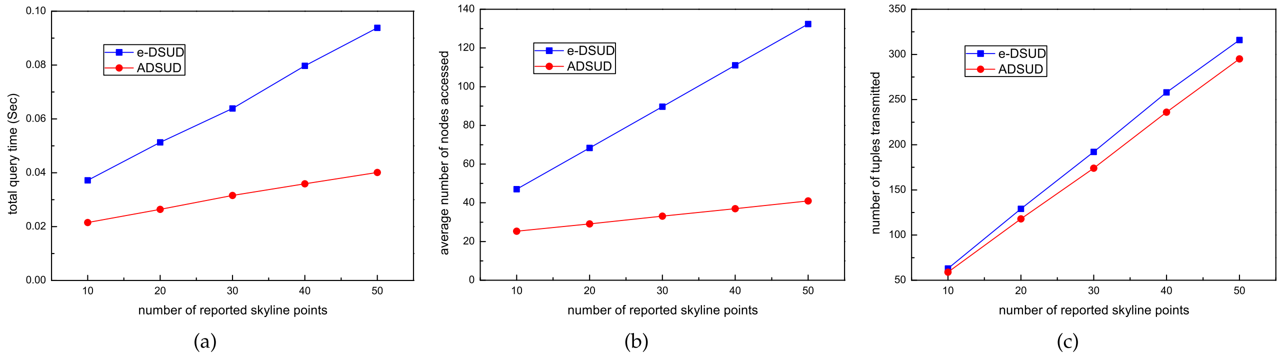


Fig. 10. Progressiveness comparison versus real dataset (Hous). (a) QT. (b) AN. (c) NT.

We analyze progressiveness of ADSUD through evaluating the QT, AN, and NT with varying the number of reported skyline points. The accumulated QT, AN, and NT of ADSUD and e-DSUD over Ind and Ant distributions are reported as the number of reported skyline tuples grows.

From Figs. 9 and 10, we note that when it returns the same number of query results, ADSUD always needs less QT, AN, and NT, comparing to e-DSUD. For Ind datasets, our ADSUD reduces 38.89 percent QT, 94.31 percent AN, and 35.76 percent NT in the best case. Consider the Ant datasets. Our algorithm can cut down 49.85 percent QT, 97.01 percent AN, and 46.59 percent NT at most separately. Besides, for the real dataset, the ADSUD could also decrease 58.93 percent QT, 82.40 percent AN, and 9.38 percent NT. It also shows that with respect to the progressiveness, ADSUD has more steady performance than e-DSUD.

The two phenomenons above indicate ADSUD has much better progressiveness. This is expected because in the Server-Feedback phase of ADSUD, it chooses multiple global candidate skyline tuples from H and sends them to local sites each time. Therefore after each iteration, it can return several query results. However, in e-DSUD, after each iteration, it can return only one query answer and sometimes we can not get any answer.

7 Conclusion

In this paper, we focus on the P-skyline query in distributed environment, namely DSUD query. In order to accelerate the DSUD query, we propose an improved DSUD framework and design an ADSUD algorithm. In ADSUD, several efficient technologies, including an IPR-tree and the reuse technology, are employed. Moreover, we define the MPBR for collecting the global abstract information and selecting local representative tuples. Extensive experiments have been conducted to clarify the effectiveness and the

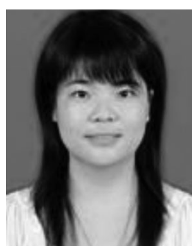
efficiency of our algorithms. Considering MapReduce possess tremendous advantages in extracting, processing, and analysis of big datasets, the DSUD queries under MapReduce can be an extension work to be done as our future work.

ACKNOWLEDGMENTS

The authors are grateful to the three anonymous reviewers. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61202109, and 61472126), and the International Science & Technology Cooperation Program of China (Grant No. 2015DFA11240). Kenli Li is the corresponding author.

REFERENCES

[1] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 15–26.
[2] X. Liu, D.-N. Yang, M. Ye, and W.-C. Lee, "U-skyline: A new skyline query for uncertain databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 945–960, Apr. 2013.
[3] X. Ding and H. Jin, "Efficient and progressive algorithms for distributed skyline queries over uncertain data," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 8, pp. 1448–1462, Aug. 2012.
[4] K. Hose and A. Vlachou, "A survey of skyline processing in highly distributed environments," *VLDB J. Int. J. Very Large Data Bases*, vol. 21, no. 3, pp. 359–384, 2012.
[5] K. Dongwon, I. Hyeonseung, and P. Sungwoo, "Computing exact skyline probabilities for uncertain databases," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 12, pp. 2113–2126, Dec. 2012.
[6] M. J. Atallah and Y. Qi, "Computing all skyline probabilities for uncertain data," in *Proc. 28th ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2009, pp. 279–287.
[7] X. Lian and L. Chen, "Reverse skyline search in uncertain databases," *ACM Trans. Database Syst.*, vol. 35, no. 1, p. 3, 2010.
[8] W. Zhang, X. Lin, Y. Zhang, W. Wang, G. Zhu, and J. X. Yu, "Probabilistic skyline operator over sliding windows," *Inform. Syst.*, vol. 38, no. 8, pp. 1212–1233, 2013.
[9] X. Lian and L. Chen, "Efficient processing of probabilistic group subspace skyline queries in uncertain databases," *Inform. Syst.*, vol. 38, no. 3, pp. 265–285, 2013.
[10] W.-T. Balke, U. Güntzer, and J. X. Zheng, "Efficient distributed skylining for web information systems," in *Proc. Adv. Database Technol.*, 2004, pp. 256–273.
[11] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "Agids: A grid-based strategy for distributed skyline query processing," in *Data Management in Grid and Peer-to-Peer Systems*. New York, NY, USA: Springer, 2009, pp. 12–23.
[12] L. Zhu, Y. Tao, and S. Zhou, "Distributed skyline retrieval with low bandwidth consumption," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 3, pp. 384–400, Mar. 2009.
[13] L. Chen, B. Cui, and H. Lu, "Constrained skyline query processing against distributed data sites," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 2, pp. 204–217, Feb. 2011.
[14] G. Trimponias, I. Bartolini, D. Papadias, and Y. Yang, "Skyline processing on distributed vertical decompositions," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 850–862, Apr. 2013.
[15] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Angle-based space partitioning for efficient parallel skyline computation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 227–238.
[16] S. Sun, Z. Huang, H. Zhong, D. Dai, H. Liu, and J. Li, "Efficient monitoring of skyline queries over distributed data streams," *Knowl. Inform. Syst.*, vol. 25, no. 3, pp. 575–606, 2010.
[17] B. Chen and W. Liang, "Progressive skyline query processing in wireless sensor networks," in *Proc. 5th Int. Conf. Mobile Ad-hoc Sensor Netw.*, 2009, pp. 17–24.
[18] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in mapreduce," *VLDB J.*, vol. 23, no. 3, pp. 355–380, 2014.
[19] B. Zhang, S. Zhou, and J. Guan, "Adapting skyline computation to the mapreduce framework: Algorithms and experiments," in *Database Systems for Adanced Applications*. New York, NY, USA: Springer, 2011, pp. 403–414.
[20] Y. Tao, W. Lin, and X. Xiao, "Minimal mapreduce algorithms," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 529–540.
[21] Y. Park, J.-K. Min, and K. Shim, "Parallel computation of skyline and reverse skyline queries using mapreduce," in *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 2002–2013, 2013.
[22] L. Chen, K. Hwang, and J. Wu, "Mapreduce skyline query processing with a new angular partitioning approach," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2012, pp. 2262–2270.
[23] K. Mullesgaard, J. L. Pederseny, H. Lu, and Y. Zhou, "Efficient skyline computation in mapreduce," in *Proc. 17th Int. Conf. Extending Database Technol.*, 2014, pp. 37–48.
[24] T. Jiang, Y. Gao, B. Zhang, D. Lin, and Q. Li, "Monochromatic and bichromatic mutual skyline queries," *Expert Syst. Appl.*, vol. 41, no. 4, pp. 1885–1900, 2014.
[25] Y. Gao, Q. Liu, B. Zheng, and G. Chen, "On efficient reverse skyline query processing," *Expert Syst. Appl.*, vol. 41, no. 7, pp. 3237–3249, 2014.
[26] Y. Tao, X. Xiao, and J. Pei, "Efficient skyline and top-k retrieval in subspaces," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 8, pp. 1072–1088, Aug. 2007.



Xu Zhou received the master's degree from the Department of Information Science and Engineering, Hunan University, in 2009. She is currently working toward the PhD degree in the Department of Information Science and Engineering, Hunan University, Changsha, China. Her research interests include data management.

Kenli Li received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a full professor of computer science and technology at Hunan University. His major research includes parallel computing and cloud computing. He has published more than 130 papers in international conferences and journals, such as the *IEEE Transactions on Computers* and *IEEE Transactions on Parallel and Distributed Systems*. He is an outstanding member of CCF and a member of the IEEE.

Yantao Zhou received the PhD degree in information and electrical engineering from the Wuhan Naval University of Engineering, China, in 2009. He is currently a professor of electric and information engineering at Hunan University in Changsha. His major research contains parallel computing and distributed data management.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and distributed computing. He has published more than 350 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Cloud Computing*. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.