Progressive Skylining over Web-Accessible Databases

Eric Lo[a,1] Kevin Y. Yip[b,1] King-Ip Lin[c] David W. Cheung[d]

[a]*Department of Computer Science, ETH Zurich*

[b]*Department of Computer Science, Yale University*

[c]*Division of Computer Science, The University of Memphis*

[d]*Department of Computer Science, The University of Hong Kong*

Abstract

Skyline queries return a set of interesting data points that are not dominated on all dimensions by any other point. Most of the existing algorithms focus on skyline computation in centralized databases, and some of them can progressively return skyline points upon identification rather than all in a batch. Processing skyline queries over the web is a more challenging task because in many web applications, the target attributes are stored at different sites and can only be accessed through restricted external interfaces. In this paper, we develop PDS (progressive distributed skylining), a progressive algorithm that evaluates skyline queries efficiently in this setting. The algorithm is also able to estimate the percentage of skyline objects already retrieved, which is useful for users to monitor the progress of long running skyline queries. Our performance study shows that PDS is efficient and robust to different data distributions and achieves its progressive goal with a minimal overhead.

Key words: Distributed DBs, query optimization, information services on the web, web-based information systems

Email addresses: `eric.lo@inf.ethz.ch` (Eric Lo), `yuklap.yip@yale.edu` (Kevin Y. Yip), `linki@msci.memphis.edu` (King-Ip Lin), `dcheung@cs.hku.hk` (David W. Cheung).

[1] The work was done when the authors were with the University of Hong Kong

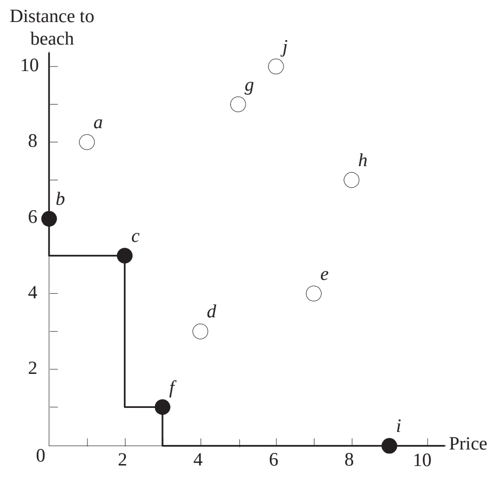


Fig. 1. Example dataset and skyline

1 Introduction

Many database users are interested in the "best" queries – queries that return the objects that best fit a certain set of criteria. However, in many cases – especially in multi-criteria decision making – there is no single best answer to a query. Instead, we have a set of objects that satisfy some basic conditions but none of them is absolutely better than the others.

For example, suppose we want to book a hotel. We may specify the room price (the lower the better) and the distance of the hotel to the beach (the shorter the better) as the evaluation criteria. A hotel is interesting only if there does not exist another available hotel that is not more expensive, but closer to the beach, or one that is not further away from the beach, but costs less. Figure 1 shows ten hotel options, where the two axes represent the two evaluation criteria in arbitrary scales. Following the arguments above, only options b, c, f and i (marked by solid circles) are interesting. We cannot claim any of them to be the best hotel without additional information about the relative weights of the two criteria. However, the best hotel is always in this set for all possible non-negative weights. We call such a set of objects the *skyline* (since the line connecting them resembles a skyline), the objects in the set the *skyline objects*, and the queries that retrieve all skyline objects the *skyline queries* [1].

Skylines are useful for multi-criteria decision making as they represent sets of objects (e.g. solutions to a problem) out of which no better objects can be found. They can be used to filter sub-optimal objects, after which users can focus on the small skyline and select the objects that fit their needs.

As a tremendous amount of data is spread all over the Internet, it is possible that each criterion (represented by an attribute of an object) is stored at a separate site. For our hotel example, the room prices and distances to the beach could be stored at a broker web site and a digital map web site respectively. It is thus important to devise algorithms that can answer this kind of *distributed skyline queries*.

In real situations, it is very common for users to refine their queries several times before getting the desired results. For example, if a user observes the query results that based on the above two criteria contain some hotels that are too far away from the airport, he may refine the query by adding the time to the airport as a new evaluation criterion. A user may get frustrated if he realizes that his query needs to be refined after waiting for a long processing time. This means having a good response time is a crucial requirement for algorithms that answer skyline queries. Rather than returning all skyline objects at the end of the query processing, the algorithms should return skyline objects as long as they are ready since they may give important refinement hints to users, who may decide to terminate the current query and start a refined one immediately. We will devise one such *progressive* algorithm in this paper. We will also minimize the number of remote data access, as the cost of remote data access can be much higher than the computational cost [2,3]. The algorithm can be useful in applications such as interactive search engines.

Pioneering algorithms have been proposed to find skylines in centralized databases in a batch [1], and in a progressive fashion [4–6]. [2] extends skyline queries to the web environment described above. However, as far as we know, there have been no previous works on finding skylines progressively in the distributed environment described above. In this paper we will propose an efficient progressive algorithm for processing skyline queries in such an environment.

The main contributions of this study are: (1) We present a novel algorithm, PDS (progressive distributed skylining), for evaluating skyline queries over web data sources. Internally it uses R*-trees to efficiently determine whether an object belongs to the skyline. (2) We propose a new linear-regression based method to enable faster identification of skyline. (3) We extend PDS to evaluate top-K skyline queries over web databases. (4) We propose ways to estimate the percentage of skyline objects already retrieved, which is useful for users to monitor the progress of long running skyline queries. (5) We show, by various experimental results, that PDS outperforms and is more robust than the distributed skyline algorithms in [2] in terms of both the number of source access and processing time.

The rest of the paper is organized as follows. Section 2 presents the related work on skyline query processing. In Section 3, we review the basic blocking skyline algorithms for the web, which solves a problem similar to ours, but does not return skyline objects progressively. Section 4 describes the details of PDS. In Section 5, we discuss how to extend PDS to evaluate other types of skyline queries and to estimate the query progress. In Section 6, we report the experimental results and finally in Section 7, we conclude the study with some directions for future works.

2 Definition and Related Work

In this section we define the skyline query problem formally. We describe our model for web data access. We also briefly introduce the related work in skyline query processing.

2.1 Definition and model

Given a database, a user is interested in retrieving objects based on a set of evaluation criteria. Each of them is modeled as an attribute of the objects with totally ordered values. For simplicity, we assume each attribute value is a non-negative real number, and smaller values are better. For instance, in our hotel example, a lower price corresponds to a better choice. The algorithms to be discussed in this paper can be modified in trivial ways to handle attributes in which larger values are better.

Given a set of evaluation criteria, we say an object a is *dominated* by another object b (or equivalently, b is said to *dominate* a) if b is not worse than a as evaluated by all criteria, and is strictly better than a as evaluated by at least one criterion. Intuitively, this means b is strictly better than a. Two objects are *incomparable* if both of them are not dominated by the other. A set of objects are *pairwise incomparable* if none of them is dominated by any other. The skyline of a dataset is defined as the set of all data objects that are not *dominated* by any other object in the dataset. It is easy to see that the skyline of a dataset is unique, and the skyline objects are pairwise incomparable.

In this paper, we assume the evaluation criteria (object attributes) are distributed. Each attribute is supplied by a separate data source D_i. Following [3], we also assume each data source provides only two data access methods: *Sorted Access* and *Random Access*. Sorted access is performed through the `getNext()` function, which returns the best attribute value among those that have not been accessed, and the ID of the object that owns the value. If there are multiple objects that own the value, each `getNext()` call will return one of them in an arbitrary order. Referring to Figure 1, if the `getNext()` function of the digital map web site (that stores the distances to the beach) is called multiple times, it will return <i, 0> on the first call, <f, 1> on the second, <d, 3>

on the third, and so on. Random access is performed through the `getScore()` function, which takes an object as input, and returns its attribute value. The call `getScore(h)` to the digital map web site will return the value 7. We use $S(D_i, O, V_{Oi})$ to denote sorted access to data source D_i that returns object O and its attribute value V_{Oi}. The corresponding notation for a random access is $R(D_i, O, V_{Oi})$.

2.2 Related work

Skylines, and other directly related problems like maximum vector [7], contour problem [8] and multi-objective optimization [9], have been well studied for conventional datasets. However, the algorithms do not work when the dataset does not fit into main memory. In relational database context, Borzsonyi et al. [1] first proposed two algorithms, namely *Divide-and-Conquer* (D&C) and *Block Nested Loops* (BNL), to evaluate skyline queries. D&C divides the dataset into several partitions such that each partition can fit into memory. Skyline objects for each individual partition are then computed by a main-memory skyline algorithm. The final skyline is obtained by merging the skyline objects for each partition. BNL compares each object in a nested-loop by keeping a list of candidate skyline objects in main memory. Dominated objects are discarded during the comparisons. The efficiency of BNL is further improved in [10] by sorting the entire dataset according to a monotonic function *a priori*.

Tan et al. [6] presented two progressive skyline algorithms that require preprocessing. The first algorithm encodes the dataset as a collection of bitmaps. Decision on whether an object belongs to the skyline is based on some bit operations. The second algorithm organizes a set of d-dimensional objects into d lists such that an object o is assigned to list i if and only if its value at attribute i is the best among all attributes of o. Each list is indexed by a B-tree, and the skyline is computed by scanning the B-tree until an object that dominates the remaining entries in the B-trees is found. Kossmann et al. [4] observed that the skyline problem is closely related to the nearest neighbor (NN) search problem. They proposed an algorithm that returns skyline objects progressively by applying nearest neighbor search on an R-tree indexed dataset. Papadias et al. [5,11] surveyed most of the secondary-memory algorithms for centralized database and proposed an I/O optimal algorithm that returns skyline progressively. More recently, Chan et al. [12] studied the computation of skyline with partially-ordered domains. They proposed to transform each partially-ordered attribute into a pair of integer attributes (interval) so as to enable the use of index-based techniques as well as avoid the "dimensionality curse" problem [1].

The above methods are designed for centralized databases and do not work well in the distributed web environment because of two reasons. First, they assume attribute values can be accessed at no (or negligible) cost. It is true in a centralized database. However, running these algorithms in the distributed setting may incur substantial amount of unnecessary network access. Second, these centralized query processing techniques rely on indexes that build on all attributes involved in the queries. However, these centralized data structures are not available in a fully decentralized and distributed environment like the Internet.

Very recently, Lin et al. [13] studied ways to support on-line skyline query computation in a stream environment. Still, their work is different from what we are studying because the data values of a object arrived from a data stream are all known *apriori*, where it is not necessary true in the web.

Following the general query model [3] for the web environment, Balke et al. [2] extended the skyline problem for the web in which the attributes of an object are distributed in different web-accessible servers, as we have described in the last section. They proposed two algorithms that compute the skyline in such a distributed environment, which return all skyline objects at the end rather than progressively. Since the problem that they study is similar to the one of the current study, we will discuss them in detail in the next section as an important reference to our new algorithm.

3 Skylining on the Web

3.1 Basic distributed skyline (BDS) algorithm

The basic distributed skyline (BDS) algorithm in [2] contains two phases. The first phase identifies a subset of objects that include all the skyline objects (probably with some other non-skyline objects). The second phase filters out all the non-skyline objects in the subset.

In the first phase, data is retrieved by sorted access only, and each data source is invoked in a round-robin fashion. To illustrate the idea, we extend our hotel example by adding an extra evaluation criterion, time to the airport (say, obtained from the web site of a shuttle bus company), and present it in tabular format in Figure 2. Based on the round-robin scheme, BDS performs sorted access to the database in the order $S(D_1, b, 0)$, $S(D_2, i, 0)$, $S(D_3, e, 1)$, $S(D_1, a, 1)$, $S(D_2, f, 1)$, and so on. The algorithm avoids reading an excessive amount of data by looking for a *terminating object*, which is the first object with all its attribute values retrieved. In Figure 2, the terminating object is

Hotel	Price	Hotel	Distance to the beach	Hotel	Time to the airport
b	0	i	0	e	1
a	1	f	1	c	2
c	2	d	3	j	3
f	3	e	4	f	4
d	4	c	5	b	5
g	5	b	6	g	6
j	6	h	7	h	7
e	7	a	8	i	8
h	8	g	9	d	9
i	9	j	10	a	10

Broker (site D_1) Digital map (site D_2) Bus company (site D_3)

Fig. 2. Example dataset with three attributes distributed at three different sites

f, which is detected after performing 12 sorted access. An important lemma regarding the terminating object is proved in [2], which we rephrase in terms of our notations as follows:

Lemma 1 *When a terminating object T is detected and no more sorted access to each data source D_i can return a value equal to V_{Ti} (the attribute value of T at D_i), all objects not yet encountered cannot be in the skyline.* $\square$

Since attributes are retrieved by sorted access, in order to guarantee that no more attribute values from D_i is equal to V_{Ti}, some extra sorted access is performed on D_i until an attribute value strictly larger than V_{Ti} is returned. If all values are distinct for every attribute, the lemma simply states that phase one is complete when the terminating object is detected, and all objects not yet encountered cannot be in the skyline. In our example, assume attribute values are distinct, then when f is detected as a terminating object, objects g and h have not been encountered by sorted access and thus cannot be skyline objects. Intuitively, this is because they have all attribute values larger than those of f, i.e., they are dominated by f. If duplicate attribute values are allowed, the following extra sorted access has to be performed to avoid missing any skyline object: $S(D_1, d, 4)$, $S(D_2, c, 5)$ and $S(D_3, b, 5)$.

During phase one, for each attribute D_i, a table K_i is created to store all the attribute values (from D_i or not) of the objects whose values at D_i have been retrieved. Figure 3 shows the tables for the three attributes of our extended hotel example (assuming duplicate attribute values are allowed). These tables are passed to phase two, at which the non-skyline objects are filtered by retrieving the missing attribute values through random access (e.g., $R(D_2, b, 6)$) and performing a pairwise comparison between all objects in the tables. Objects found to be dominated by any other are filtered.

Hotel	D_1	D_2	D_3
b	0		5
a	1		
c	2	5	2
f	3	1	4
d	4	3	

K_1

Hotel	D_1	D_2	D_3
i		0	
f	3	1	4
d	4	3	
e		4	1
c	2	5	2

K_2

Hotel	D_1	D_2	D_3
e		4	1
c	2	5	2
j			3
f	3	1	4
b	0		5

K_3

Fig. 3. The tables that store the data obtained from phase one of the BDS algorithm

To reduce the number of comparisons, it is observed in [2] that an object O can only be dominated by other objects that exist in all tables that contain O. For example, since table K_1 contains object b but not e, b cannot be dominated by e. It follows from the fact that the value of e at attribute D_1 must not be smaller than that of b since it is not retrieved before the completion of phase one. Therefore, pairwise comparisons need to be performed between objects that exist in the same table only.

Using the same argument, a missing value for attribute D_i in a table cannot be smaller than the largest D_i value stored in the table. For example, according to table K_1, the value of d at D_3 should be at least 5. This means d is proved to be dominated by f (since d also has larger attribute values than f at D_1 and D_2), without actually getting the value of d at D_3 through a random access. Some network overhead can thus be avoided.

3.2 Improved distributed skyline (IDS) algorithm

It can be seen from Figure 2 that if a terminating object is known *a priori*, it is possible to perform less sorted access in phase one, have fewer objects stored in the K_i tables, and perform less random access and fewer pairwise comparisons in phase two. For example, if object f is predicted to be a terminating object, once $S(D_2, d, 3)$ is performed, the sorted access to D_2 can be stopped according to Lemma 1.

Using the idea, an improved distributed skyline (IDS) algorithm, based on a heuristic to predict the "most probable" terminating object, is proposed in [2]. IDS tries to find an object whose attribute values can be fully retrieved using the smallest number of sorted access. The heuristic works as follows: initially a sorted access is performed on D_1, and the object retrieved is treated as the most probable terminating object. In our example, the object is b. Unlike the original BDS algorithm that performs only sorted access in phase one, the heuristic requires the other attribute values of the most probable terminating object to be retrieved right away through random access, which include $R(D_2, b, 6)$ and $R(D_3, b, 5)$. A score is then computed to estimate the remaining number of sorted access required to reach all attribute values of b, i.e., how much extra sorted access is required if b is used as the terminating object. The score is calculated as the difference between the attribute values of b and the last values retrieved through sorted access. Since no attribute values at D_2 and D_3 have been retrieved through sorted access, the minimum value 0 is used. The score for object b at that moment is thus $(0-0)+(6-0)+(5-0) = 11$.

The algorithm then performs the next sorted access. Instead of doing it in the round-robin fashion, the heuristic randomly selects one of the attributes that the most probable terminating object has not been encountered through sorted access. For object b, the attributes are D_2 and D_3. Suppose D_2 is selected randomly, then the sorted access $S(D_2, i, 0)$ is performed. Again, the other attribute values of i are retrieved through random access $R(D_1, i, 9)$ and $R(D_3, i, 8)$, and the score of it is computed as $(9-0)+(0-0)+(8-0) = 17$. Since the score is larger than that of b, intuitively the "distance" to reach b through sorted access is shorter, so the heuristic keeps b as the most probable terminating object and uses it to decide the next sorted access.

4 Progressive Distributed Skyline Algorithm

In this section we describe the details of our new algorithm PDS. For clarity, we first assume that all the score values are distinct. Therefore, objects returned by sorted access are sorted in a strictly ascending order like Figure 2. Later we will consider the general case where duplicate values are allowed. In Section 4.1, we present the shortcomings of BDS and IDS and the motivations for PDS. Then we show how PDS could identify skyline objects progressively in Section 4.2 and how PDS improves the performance of distributed skylining by rank estimation in Section 4.3. Section 4.4 describes the basic PDS and we generalize it to the case where duplicate values are allowed in Section 4.5. Finally, in Section 4.6, we provide a theoretical evaluation of our work in terms of the desirable features of progressive algorithms listed in [14,4].

4.1 Motivations

Section 3 presented the BDS and IDS algorithms from [2]. While they perform well in various cases, we believe there are grounds for improvement:

(1) In both algorithms, an object is confirmed as a skyline object only after the pairwise comparisons in phase two is completed. In real-time applications this may result in a long response time before the first skyline object is returned.

In fact, it is possible to identify skyline objects much earlier. For example, the two sorted access $S(D_1, b, 0)$ and $S(D_1, a, 1)$ are sufficient to show that b is a skyline object (since no other objects can have a value smaller than b at D_1), so it can be returned right away.

(2) While IDS reduces the number of data access, its success depends on the accuracy of the heuristic to correctly predict the terminating object. Yet the heuristic only works in some specific situations. In some cases, as we are going to show in Section 6, the prediction is not accurate and the number of data access is increased, rather than decreased, due to the extra random access performed in phase one.

Thus, we propose a progressive distributed skylining (PDS) algorithm. PDS does not wait till the end of the algorithm to start returning skyline objects. In fact, if there are no duplicate score values in each data source, PDS determines whether an object is a skyline object immediately after it is first retrieved and outputs it right away if it is, which results in a short response time.

Moreover, we alleviate the problem of the existing heuristic by proposing the use of the object ranks in various attributes for terminating object prediction. We present a linear-regression-based method to estimate the ranks of the object. We argue in Section 4.3 that rank is a more appropriate measure to determine the terminating object. Experiments show that our estimation is robust over various distributions. Moreover, it also works well when data values in different sources are correlated or anti-correlated.

Below we discuss the two main ideas of PDS: progressiveness and rank estimation.

4.2 Enabling progressiveness

To enable progressiveness, we need to determine whether an object belongs to the final skyline as soon as its attribute values are retrieved. Achieving this requires the following lemma:

Lemma 2 *If the sorted access of a data source D_i returns data values in strictly increasing order, an object O retrieved from D_i can only be dominated by objects that are retrieved from D_i before O.*

Proof. By definition, if an object O is dominated by another object P, every attribute value of P must be smaller than or equal to the corresponding attribute value of O. Since D_i returns no duplicate values, the value of P at D_i must be smaller than that of O. This means sorted access to D_i must return P before O. □

Lemma 2 shows that if an object O is retrieved from a data source by sorted access, we only need to test if O is dominated by any object that appears *before* O in the same source. If no such dominating objects exist, O is definitely part of the skyline – as all objects that appear after O cannot dominate O.

Take data source D_3 in Figure 2 as an example. Object $e(7,4,1)$ is first retrieved by invoking the `getNext()` function on D_3 and is identified as part of the skyline because no objects can dominate e on data source D_3 (the attribute values 7 and 4 of e are retrieved by random access [2] if necessary). Next, object $c(2,5,2)$ is retrieved and compared with all objects that appear before it (i.e., object e) and found to be incomparable with e. By Lemma 2, we know that c would not be dominated by objects after it and thus c can be outputted as part of the skyline immediately. Similarly, object $j(6,10,3)$ is next retrieved from D_3. We compare it with e and c and found that j is dominated by e in all dimensions and therefore j is not part of the skyline.

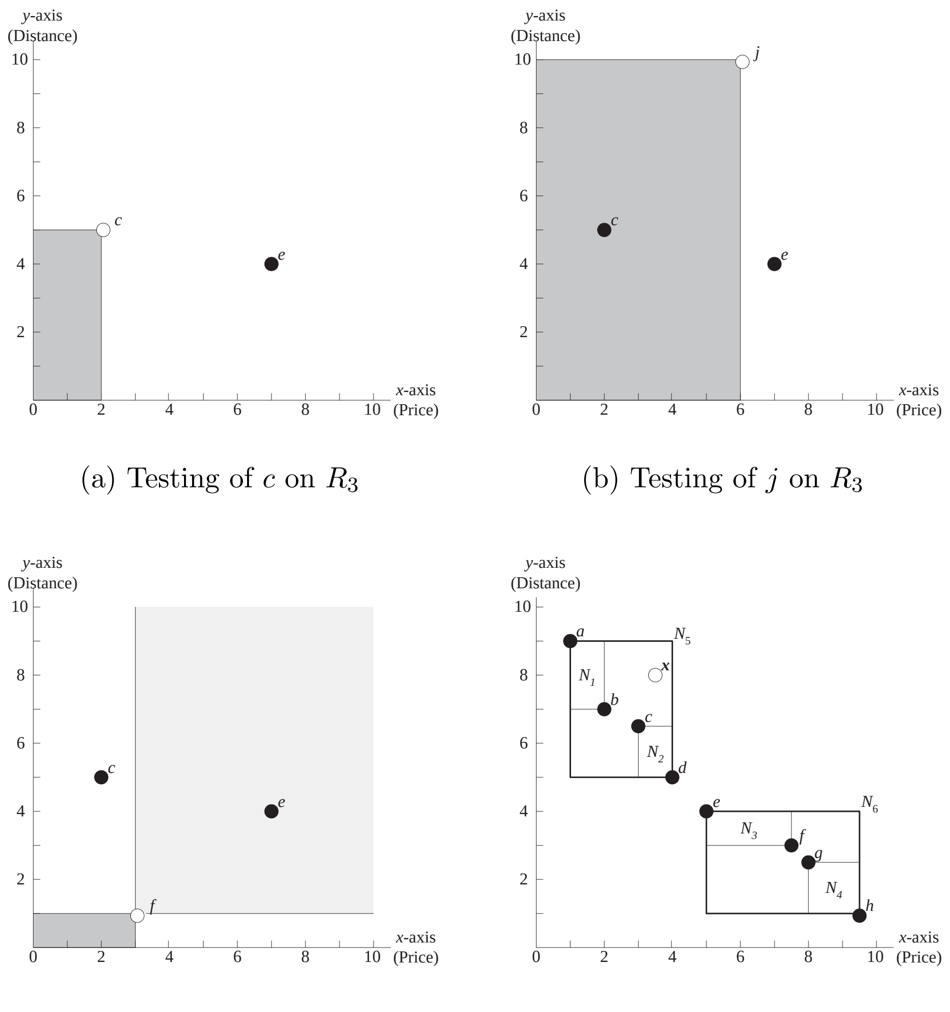
Still, Lemma 2 requires an object O to be compared to every object retrieved before it in the same source, which can be slow if the comparisons are performed in a brute-force manner. We would like to reduce the number of pairwise comparisons. To this end, we propose using a main-memory multidimensional index to speed up this process. We choose R*-tree [15] as the index because of its efficiency and wide acceptance in the skyline literature [4,5]. Our approach differs from the R-tree algorithms that find skyline in centralized databases [4,5] in that the target data set is not ready before the evaluation of the query, which means the tree in our approach is built and updated on-the-fly.

An R*-tree R_i is built for each attribute i involved in the skyline query. R_i contains the skyline objects discovered so far based on sorted access to D_i. As a result, for an object O retrieved from a data source D_i, we can check to see if there is any object in R_i that dominates O. If no such objects can be found, O is identified as a skyline object *immediately* and is inserted into R_i.

In order to demonstrate the concept of R*-tree, let us consider data source D_3 again. Initially, the skyline object e is inserted into R_3. Note that since the score values of the objects retrieved from each source are strictly increasing, we do not need to insert the i-th attribute value of an object into R_i. For instance, the skyline object $e(7,4,1)$ is inserted into R_3 in the form of $e(7,4)$.

To test whether a newly retrieved object O is dominated by any object that has been retrieved before from the same source D_i, a containment query Q_O is constructed, with the origin set as the lower-left corner and O set as the upper-right corner, to see if any object in R_i is contained within Q_O. If the

[2] Following [2,3], here we assume an object O must exist in every data source D_i; however in practice, we could discard O if it does not exist in all data sources.



(a) Testing of c on R_3 (b) Testing of j on R_3

(c) Testing of f on R_3 (d) An example R*-tree

Fig. 4. R*-tree for dominance checking

result of Q_O is empty, no objects retrieved before O dominate it in all the projected dimensions. In this case, O is a skyline object and can be outputted. Conversely, if Q_O is non-empty, there exists at least one object, say S, that is better than O in all projected dimensions. Since S is retrieved before O from data source D_i, this implies that S also has a better value in D_i than O. Therefore, O is dominated by S in all dimensions and does not belong to the final skyline.

Let us revisit data source D_3 as an example. Firstly, recall that object $e(7, 4, 1)$ is a skyline and is thus inserted into R_3 in the form of $e(7, 4)$. To test whether the next retrieved object $c(2, 5, 2)$ is a skyline, a containment query $Q_c[(0, 0)(2, 5)]$ that is based on the projected dimensions D_1 and D_2 of c is executed on R_3 (see Figure 4(a)). Since the result of Q_c is empty, object c is outputted as a skyline object and is inserted into R_3. Next, to see if object $j(6, 10, 3)$ belongs

```
Function isDominated(Object $o$, Attribute $i$, R*-tree $R_i$)
1.   //$R_i$:the R*-tree storing skyline objects appears in attribute $i$
2.   let $N$ as the number of dimensions involved in the skyline
3.   construct a $(N-1)d$ containment query $Q_o$
4.   set the origin as the lower-left corner of $Q_o$ (ignore dimension $i$)
5.   set $o$ as the upper-right corner of $Q_o$ (ignore dimension $i$)
6.   if $R_i$.contain($Q_o$) returns true //$o$ is not a skyline
7.       return true
8.   else //$o$ is a skyline
9.       construct a $(N-1)d$ containment query $Q_o$
10.      set $o$ be the lower-left corner of $Q_o$ (ignore dimension $i$)
11.      set infinity as the upper-right corner of $Q_o$ (ignore dimension $i$)
12.      delete all objects falls in $Q_o$
13.      $R_i$.insert($Q_o$)
13.      return false
End isDominated
```

Fig. 5. Dominance checking by R*-tree

to the skyline, we again execute another containment query $Q_j[(0,0)(6,10)]$ on R_3 (see Figure 4(b)). Query Q_j returns c as an answer, thus object c dominates j at *all* three attributes and thus j can be discarded. Similarly, the containment query $Q_f[(0,0)(3,1)]$ for object $f(3,1,4)$ has an empty result (see Figure 4(c)), therefore f belongs to the skyline and is inserted into R_3. Notice that since $f(3,1)$ dominates $e(7,4)$ in R_3, insertion of f into R_3 should delete objects that are dominated by f in order to make the R*-tree more compact and efficient. In this example, e should be deleted from R_3 after inserting f.

The benefits of using R*-tree to check domination of objects is two-fold. First, the R*-tree built for each attribute is very compact in size since it stores only the skyline objects that have the highest pruning power (e.g., object e is deleted after the insertion of object f into R_3). Second, the containment query operation is very efficient because a containment query Q only visits objects with minimum bounding boxes that overlap with Q in the compact R*-tree. Figure 4(d) shows the node structure of an example R*-tree index. We can see that the dominance checking function does not need to access node N_6 and its children for testing object x.

Summing up the above ideas, we first present an R*-tree based dominance checking function *isDominated* in Figure 5. This function will be employed by the main algorithm PDS later. It returns true if and only if an input object O is dominated by an existing skyline object that appears before O in the same data source.

4.3 Rank estimation

In order for the distributed skyline algorithm to be efficient, it is crucial to find a good terminating object quickly. Here we argue that a good terminating object is characterized by its *rank* in the sorted list of attribute values in each of the sources.

We denote $rank_i(O)$ as the rank of object O in source D_i. We adopt the convention that the first object retrieved from the source has rank 1, the second one has rank 2 and so on. We define $sumrank(O) = \sum_i rank_i(O)$ as the sum of the ranks in all attributes of O. We note the following:

- If T is a terminating object, then the minimum number of access required to locate it by sorted access only is $sumrank(T)$.
- The number of objects that need to be compared increases when $sumrank(T)$ increases, though the increase is not necessarily linear. An increase in the number of objects implies a higher number of random access (to retrieve all the attributes) is required.

The smaller $sumrank(O)$ is, the more preferable O is as the terminating object. Thus we would like PDS to locate an object with the minimum sum of all ranks as the terminating object. This implies that the minimum number of sorted access (and likely, random access) is needed to pinpoint the terminating object.

However, in our model the rank of an object in a data source D_i is not readily available. For instance, a random access supplies the attribute value of an object, but it provides no hints to the number of objects appearing before it. The only way to know the exact rank of O in D_i is via sorted access – if O is retrieved by the k-th sorted access, the rank of O in D_i is k. As a result, we need to estimate the rank of an object in each data source. If one has the information about the distribution of values in a source, one can tailor-make a function to accurately estimate the rank of an object in that source. In this paper, however, we assume such information is not available. Thus we have to devise a method to estimate the rank. While we would like the method to be accurate, we would also like to avoid laborious calculations that take extensive CPU time. We propose using linear regression as our rank estimation method. For each source, PDS has the information about the ranks of the objects already retrieved through sorted access. For a source D_i, we use the information to devise an equation of the form $rank_i = a_i * value_i + b_i$ such that the squared error of the estimated ranks of all retrieved values is minimized, where $value_i$ and $rank_i$ are the value and rank of an object in source D_i. Assume that we have made k sorted access to source D_i, we can obtain a_i and b_i by the following linear regression formulas, which can be found in any standard numerical analysis textbook.

$$a_i = \frac{k \sum rank_i value_i - (\sum value_i)(\sum rank_i)}{k \sum {value_i}^2 - (\sum value_i)^2} \tag{1}$$

$$b_i = \frac{(\sum rank_i)(\sum {value_i}^2) - (\sum value_i)(\sum rank_i value_i)}{k \sum {value_i}^2 - (\sum value_i)^2} \tag{2}$$

By utilizing the retrieved data values in rank estimation, the estimation accuracy is not affected in some adverse situations such as having different value ranges in different attributes, which can severely mislead the estimation method of IDS.

It is easy to determine the values of the coefficients a_i and b_i. Moreover, we can maintain a current value of all the sums in the formulas, and update them incrementally when new values come to get the updated coefficients. As we need at least two values to evaluate the formulas, initially we make two sorted access to each source, and evaluate the initial value of each a_i and b_i.

There are many ways to perform function estimation, and we understand the simple linear least square method may not be always the best one. But in general, it is simple, efficient and quite effective. We will show by experiment that it works very well in various distributions – skewed or not. In more complicated cases where the linear least square does not work well, our method can easily be extended to more advanced methods such as least square fitting to higher order polynomials.

4.4 Algorithm description

The pseudo-code of the main algorithm PDS is presented in Figure 6 and its auxiliary functions are presented in Figure 7. At the beginning, the algorithm performs two sorted access to each source. Consider Figure 2 again. The objects and attribute values returned by the sorted access are: $S(D_1, b, 0)$, $S(D_1, a, 1)$; $S(D_2, i, 0)$, $S(D_2, f, 1)$; $S(D_3, e, 1)$, $S(D_3, c, 2)$. Since the data values are sorted in a strictly increasing order, objects b, i and e are skyline objects from attributes D_1, D_2 and D_3 respectively. By Lemma 2, after retrieving the unknown attribute values of objects b, i and e by random access, they are reported as skyline objects by the `OutputSkyline` function immediately.

Now, the regression coefficients a_i and b_i can be calculated according to the retrieved information. For example, given objects <e,1> (rank 1) and <c,2>

```
Algorithm PDS
1.  $Skyline = \emptyset$ //list of skyline objects
2.  $Pruned = \emptyset$ //list of pruned objects – object that are not skylines
3.  $Cand = \emptyset$ //list of objects that are candidates for terminating object
4.  for each data source $D_i$
5.    perform sorted access and retrieves the two topmost objects $o_{i1}, o_{i2}$
6.    perform necessary random access to $o_{i1}, o_{i2}$
7.    $OutputSkyline(o_{i1}, i)$
8.    initialize regression coefficients $a_i, b_i$
9.    $CheckSkyline(o_{i2}, i)$
10.   add $o_{i1}, o_{i2}$ to $Cand$
11. end for
12. while no objects have all attribute values retrieved by sorted access
13.   $T = PickNextTermObject()$
14.   let $D'$ = set of all data sources of which the attribute values of $T$ have not been retrieved by sorted access
15.   Randomly pick one data source $D'_i$ from $D'$
16.   perform sorted access to $D'_i$, let $o_i$ be the object retrieved
17.   add $o_i$ to $Cand$
18.   update regression coefficients $a_i, b_i$
19.   if $o_i \notin Pruned$ or $Skyline$
20.     $CheckSkyline(o_i, i)$
21. end while
End PDS
```

Fig. 6. The PDS algorithm

(rank 2) from data source D_3, the values of a_3 and b_3 are:

$$a_3 = \frac{2(1 \cdot 1 + 2 \cdot 2) - (1+2)(1+2)}{2(1^2+2^2) - (1+2)^2} = 1$$

$$b_3 = \frac{(1+2)(1^2+2^2) - (1+2)(1 \cdot 1 + 2 \cdot 2)}{2(1^2+2^2) - (1+2)^2} = 0$$

After the initialization of the regression coefficients a_i and b_i, PDS invokes the `CheckSkyline` function to determine whether the second-ranked objects a, f and c belong to the skyline. Essentially, `CheckSkyline` checks if the input object is dominated by any other object by calling the `isDominated` function, which we have presented in the previous section. If the input object is dominated by any other object, it would be discarded; otherwise, it would be reported as a skyline object by the `OutputSkyline` function. After the regression coefficients are updated, one of the skyline objects is selected by the function `PickNextTermObject` as the "most probable" terminating object.

```
Function CheckSkyline(Object $o$, Attribute $i$) //check if $o$ is a skyline point
1.  if isDominated($o$, $i$, $R_i$)
2.      add $o$ to $Pruned$
3.  else
4.      $OutputSkyline(o, i)$
End CheckSkyline

Function PickNextTermObject() //Find the most probable terminating object
1.  for each $o$ in $Cand$
2.      $rank_o = 0$;
3.      for each data source $D_i$
4.          if $o$'s rank in $D_i$ is known
5.              $rank_o$ += $o$'s rank in $D_i$
6.          else
7.              $rank_o$ += $a_i * o_i.value + b_i$
8.  end for
9.  return $o$ with the minimum $rank_o$
End PickNextTermObject()

Function OutputSkyline(Object $o$, Attribute $i$) //report a skyline to user
1.      report $o$ as a skyline
2.      add $o$ to $Skyline$
End OutputSkyline()
```

Fig. 7. The auxiliary functions of PDS

Object	Values	Ranks	Sum of Ranks
b	(0, 6, 5)	$(1, \underline{7}, \underline{5})$	13
a	(1, 8,10)	$(2, \underline{9}, \underline{10})$	21
i	(9, 0, 8)	$(\underline{10}, 1, \underline{8})$	19
f	(3, 1, 4)	$(\underline{4}, 2, \underline{4})$	10
e	(7, 4, 1)	$(\underline{8}, \underline{5}, 1)$	14
c	(2, 5, 2)	$(\underline{3}, \underline{6}, 2)$	11

Table 1
The ranks of objects in the example

Table 1 shows an instance of the ranks of objects after the initialization step. Note that the estimated ranks of the objects are underlined. Referring to the table, object f is chosen as the most probable terminating object T because its sum of ranks, 10, is the smallest.

PDS then iteratively picks an attribute of which the value of the current terminating object has not been retrieved by sorted access in random as the next source to perform sorted access. Assume D_3 is the selected source, then object j will be retrieved by sorted access $S(D_3, j, 3)$. Afterwards, the unknown attribute values of j are retrieved by random access and the `isDominated` function will be invoked to check if j is a skyline object. The process repeats

Hotel	Price	Hotel	Distance to the beach	Hotel	Time to the airport
b	0	i	0	e	1
a	1	f	2	h	2
i	2	j	2	c	3
f	3	c	4	j	3
d	3	e	5	f	4
g	5	b	6	g	5
j	5	h	7	b	7
c	6	a	8	i	8
h	8	g	9	d	8
e	9	d	9	a	10
Broker (site D_1)		Digital map (site D_2)		Bus company (site D_3)	

Fig. 8. A dataset with duplicate values

until there is an object whose attribute values have all been retrieved by sorted access, at which time the whole query processing is complete.

4.5 Handling duplicate values

So far our discussions assume the data values returned by sorted access are distinct and sorted in a strictly increasing order. Essentially, Lemma 2 states that an object O retrieved from data source D_i can be outputted as a skyline if no objects retrieved before O from the same source dominate it. However, the lemma does not applied directly when it is possible to have duplicate attribute values. Consider the dataset in Figure 8. Objects c and j share the same data value 3 in D_3. Assume that after performing two sorted access to each source, PDS regards f as the terminating object and selects data source D_3 to perform the next sorted access. Although object $c(6, 4, 3)$ retrieved from D_3 is strictly better than both $e(9, 5, 1)$ and $h(8, 7, 2)$ on the projected dimensions D_1 and D_2, c is not a skyline object. It is because there is an unencountered object $j(5, 2, 3)$ that is as good as c along dimension D_3 (due to duplicate values) and is better than c along all other dimensions.

This problem can be solved by keeping a buffer B_i for each data source D_i. Whenever an object is retrieved from D_i and is identified as a possible skyline object by the `isDominated` function, it is inserted into B_i instead of reported as a skyline object immediately. Essentially, the buffer B_i is another R*-tree that holds possible skyline objects all with the same value in attribute D_i at a particular instant. The steps of inserting a possible skyline object into the buffer is identical to those of testing dominance of newly retrieved objects and thus can reuse the `isDominated` function. Therefore, for each possible skyline object s that is inserted into the buffer B_i, we check if s is dominated by

```
Function OutputSkyline(Object $o$, Attribute $i$) //handle duplicate values version
1. //$B_i$ : an R*-tree buffer on dimension $i$
2. if value of object $o >$ current value in buffer in dimension $i$
3.     flush all objects in the buffer $B_i$ into Skyline and report them as skyline
4.     clear buffer $B_i$
5. if isDominated($o$,$i$, $B_i$)
6.     insert object $o$ into $B_i$
7. else
8.     discard $o$
End OutputSkyline
```

Fig. 9. Revised OutputSkyline function

any object in the buffer. If this is the case, s will be discarded. Otherwise, all objects in the buffer that are dominated by s will be removed. When an object with a larger value in data source D_i is inserted into B_i, objects that are still in B_i are outputted as skyline and the buffer is cleared. Figure 9 presents the revised `OutputSkyline` function that handles duplicate values.

4.6 Effectiveness evaluation

Now, we give a theoretical evaluation of the effectiveness of PDS according to the criteria listed in [14,4].

1. *Progressiveness*: the first results should be returned to the users almost instantaneously, and the rest of the results should be returned progressively given more time.

2. *Completeness*: given enough time, the algorithm should return the full skyline completely.

3. *Correctness*: the algorithm should return objects that are definitely in the final skyline. That is, the algorithm should not return objects that will be later replaced.

4. *Fairness*: the algorithm should not favor objects that are particularly good in one dimension.

5. *Support user preferences*: users are allowed to control the order of skyline objects returned according to their preferences.

6. *Universality*: the algorithm should be applicable to skyline queries with different dimensionality and dataset, but require no special data structures.

By using the R*-tree approach, PDS returns the first results almost instantly and fulfills criterion (1). Since PDS stops only when it finds a terminating object T with all attribute values retrieved by sorted access, objects that belong to the skyline (i.e., incomparable with T) must have at least one attribute value better than T and which must have been retrieved before T is determined as a terminating object. Thus criterion (2) is also satisfied. Lemma 2 guarantees criterion (3). Since processing queries on the web has to deal with restrictive accessing interfaces (e.g., sorted access and random access only), the order of data retrieved from each source is fixed. Criterion (4) that is suitable for the centralized environment like [14,4] is inapplicable to the problem we are studying. As we will show in the next section, PDS is able to support user preference queries like top-K skyline queries, thus criterion (5) is also fulfilled. Finally, PDS satisfies criterion (6) because it supports skyline queries that involve any number of dimensions, and it is based on a standard index structure (R*-tree) that supports dynamic updates and is independent of the data distribution.

5 Extensions of PDS

Next we propose two interesting extensions of PDS. Section 5.1 discusses how PDS can be applied to processing top-K skyline queries and Section 5.2 shows how PDS can estimate the percentage of skyline objects already retrieved, which is useful for users to monitor the progress of long running skyline queries.

5.1 Evaluation of top-K skyline queries

While skyline queries return the set of all potentially interesting objects, the set can be too large for a human user to process. Therefore one may want to rank the skyline objects according to a preference function f. Consider the running example in Figure 1. A user may be interested in the top-2 skyline hotels that minimize the function $f(\mathit{price}, \mathit{distance}) = \mathit{price} + \mathit{distance}$. The outputted skyline hotels should be <f,4>, <b, 6> in this order (each number is the score of the corresponding hotel according to f). Here, we require that a smaller f value indicates a better choice, and the f value of a non-skyline object is always larger than that of one of the skyline objects. The latter requirement ensures that the desired result is a subset of the skyline.

PDS can easily handle such queries by keeping a priority queue P to keep the top-K objects along the skyline evaluation process. A skyline object s would be inserted into P if s has a lower score than an object o in P. For illustration, consider the above function f. Let bound_i be the value retrieved by the last sorted access to data source D_i, i.e., the lower bound value of objects that have not yet seen in D_i. Originally, PDS terminates when there is an object with all attribute values retrieved by sorted access. Now, PDS terminates when the k-th object in P has a score less than the sum of the lower bounds in all attributes, i.e., $score(k) < \sum_i bound_i$. It is because we know that the scores of all unseen objects must be larger than the score of the k-th object. For the same reason, an object o in P could be outputted as a top-K skyline once $score(o) < \sum_i bound_i$ during the evaluation process. Finally, for an object o with an attribute value newly retrieved by a sorted access, o can be discarded immediately if its score is higher than the score of the k-th object in P, i.e., $score(o) > score(k)$. Note that we could calculate the score of o without retrieving all its unknown attribute values by random access but to use the value $bound_i$ instead.

In fact, the evaluation strategy of top-K skyline by PDS can be extended to the case where only one source supports sorted access (denoted as the *S-Source*) and the rest supports only random access (denoted as the *R-Sources*) [3]. It is a realistic situation because the web abounds sources that support random access only (e.g., the MapQuest site[3]). While it is obvious that all distributed algorithms must scan the S-Source once in order to compute the full skyline under this model, we could evaluate top-k skyline instead of the full skyline by disabling the rank estimation method of PDS and performing sorted access to the S-Source only.

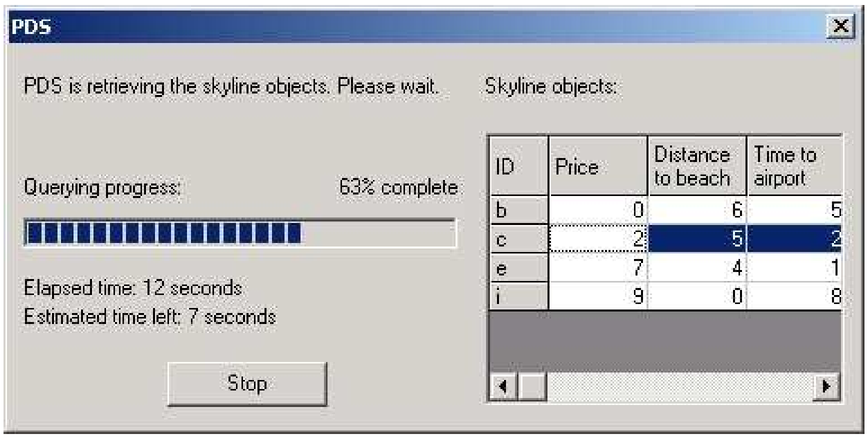
5.2 Progress estimation

A useful application of the terminating object prediction method is to estimate the amount of remaining query time, so that users can determine whether to wait for the completion of the query, or to make a decision based on the current set of results [16,17] (see Figure 10). In PDS, the estimation can be done as follows: suppose object O is the current most probable terminating object with rank (confirmed by sorted access or approximated by linear-regression) R_{oi} on source D_i. Also let R_i be the rank of the last object retrieved from D_i by sorted access. The values $\sum_i R_{oi}$ and $\sum_i R_i$ indicate the estimated total number of sorted access required by the whole query and the number of sorted access already performed respectively. We estimate the completed portion of the query, p, by the following formula:

$$p = \sum_i min(R_{oi}, R_i)/R_{oi}$$

Given p, the estimated remaining query time can be derived from p easily.

[3] http://www.mapquest.com


PDS

PDS is retrieving the skyline objects. Please wait.

Querying progress: 63% complete

Elapsed time: 12 seconds
Estimated time left: 7 seconds

Skyline objects:

ID	Price	Distance to beach	Time to airport
b	0	6	5
c	2	5	2
e	7	4	1
i	9	0	8

Fig. 10. Query progress estimation

The use of $min(R_{oi}, R_i)$ in the numerator is to handle the case $R_i > R_{oi}$, which happens when more than R_{oi} sorted access have been performed at D_i before O is identified as the most probable terminating object. Notice that the formula considers sorted access only. Since in the later stage some values retrieved by sorted access belong to objects that have already been encountered, the number of random access per sorted access is expected to decrease with time. In other words, the formula is a conservative one. In general the value of p has an increasing trend. But decreases could occur locally, due to imprecise estimated ranks or a change of the choice of the most probable terminating object. Smoothing techniques can be applied to eliminate the local fluctuations, such as reporting the average value of p in the last few estimations instead of p.

6 Experimental Evaluation

To evaluate the effectiveness and efficiency of PDS, we evaluated the performance of PDS against three distributed skylining algorithms on both real and synthetic datasets. The first two algorithms are *BDS* (Section 3.1) and its variant *IDS* that is based on distance optimization (Section 3.2). The last algorithm *Optimal* is an optimal version of PDS that has complete knowledge of the target datasets. Optimal determines the best terminating object T by scanning the whole dataset once, and then *calculates* the number of necessary sorted access and random access according to T. Of course Optimal is not applicable in the real situations. It is included to show the best case performance for comparisons.

In Section 6.1, we first evaluate the performance of PDS on a real data set. Then we study the performance of PDS in terms of the number of source access under different settings in Section 6.2. Section 6.3 studies the progres-

	BDS	IDS	PDS	Optimal
Time for first 10 objects (sec)	46	172	<1	n/a
Completion time (sec)	46	172	43	n/a
Source access (K)	35.3	37.1	28.0	24.2

Table 2
Real datasets

Dimension	(a) Uniform dataset				(b) Non-uniform dataset	
	Independent		Anti-correlated		Random	Denormalized
	10K	50K	10K	50K	50K	50K
3	50	64	367	812	61	68
4	167	266	914	2341	234	277
5	430	792	1886	5166	764	751

Table 3
Average numbers of skyline objects

sive behavior of PDS and finally Section 6.4 evaluates the effectiveness of our proposed progress estimation.

The programming language used to implement all algorithms is Java. The R-*trees being built uses a page size of 4Kbytes. Since the size of the main-memory buffer pool gives marginal impact on Skyline algorithms [1], we follow [1] and set the size of the main-memory buffer pool as 10 percent of the size of the database. A Pentium III Xeon 700MHz PC with 1Gbytes RAM is used for all experiments.

6.1 Experiments on real data

To validate PDS on real data distributions, we performed experiments on the *Cover* data set [18]. The real data set is used for predicting forest cover types from cartographic variables and was employed by [19] to experimentally evaluate top-K algorithms over web-accessible data sources. The dataset contains information about various wilderness areas. Specifically, we consider three attributes: Elevation (in meters), aspect (in degrees azimuth), and horizontal distance to roadways (in meters). For the experiments, we extracted a subset of 60,000 objects from the Cover dataset. The skyline results consist of 182 objects. Table 2 shows that PDS returned the first 10 skyline objects almost instantaneously. Further, it returned the full skyline before both PDS and IDS reported the first results. PDS also outperformed BDS and IDS in terms of source access and was close to Optimal.

6.2 Number of source access

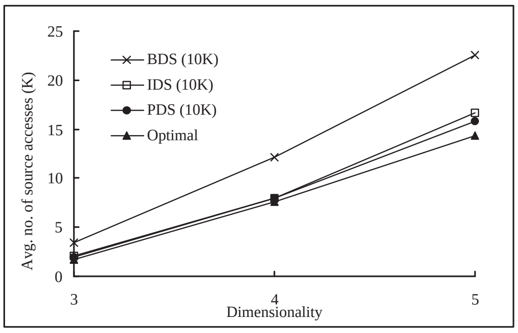
In the coming subsections we present results on synthetic datasets with various sizes and dimensionalities. The values of an attribute were generated according to either uniform, Gaussian or negative exponential distribution in ways to be described later. The attributes were either independent, correlated or anti-correlated [1]. In the first case, the values of each attribute were generated independently according to its own distribution. In the correlated and anti-correlated cases, the values of the first attribute were generated independently, and those of each subsequent attribute were generated as a linear combination of the previous attribute value of the same object and a random value sampled from the corresponding distribution. For the correlated case, the coefficient for the previous attribute value was positive so that the attributes were positively correlated, while in the anti-correlated case, the coefficient was negative so that each attribute had a negative correlation with the previous attribute.

Following the common methodology in the literature, we first study the number of source access for independent and anti-correlated datasets with 3–5 data sources [4], having 10K and 50K objects. Table 3a shows the sizes of the skylines in datasets with different dimensions (# of attributes) and dataset sizes where attribute values are generated according to uniform distributions. We can see the size of the skyline increases with the number of dimensions but not the dataset size. In addition, the size of the skyline in anti-correlated datasets is larger than the corresponding independent datasets.

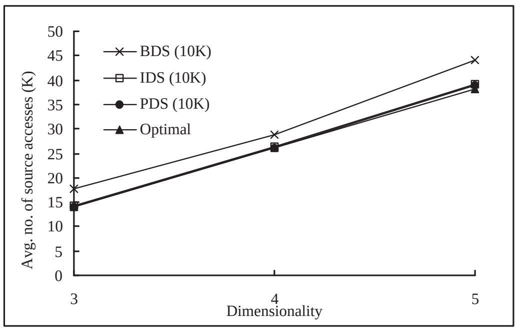
The number of source access versus dataset dimensionality for independent and anti-correlated datasets with 10K objects are shown in Figure 11(a) and Figure 11(b). IDS and PDS made fewer source access than BDS in both kinds of datasets. The results were similar when we scaled up the size of the dataset to 50K objects (see Figure 12(a) and Figure 12(b)). PDS and IDS could make fewer source access because they could reach the terminating object earlier than BDS. The figures also show that the performance of IDS and PDS were close to Optimal, which indicates that PDS and IDS estimated a set of very good terminating objects under uniform distribution. In Figure 11(b) and Figure 12(b), we observe that the improvement of PDS and IDS was less obvious when the dataset was anti-correlated. This is because the best terminating object in anti-correlated datasets usually had value ranks around half the number of objects, in which case the performance of the simple round-robin mechanism of BDS was also close to optimal.

To illustrate the robustness of the algorithms under different situations, we generated another set of datasets with dimensionalities in the range [3,5] and

[4] It is commonly accepted that skyline queries in high dimensional space yields no sensible results [1,4,2]



(a) Independent

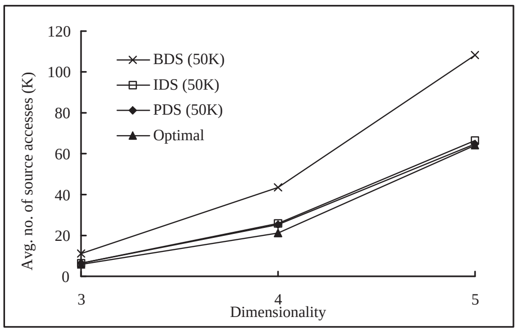


(b) Anti-Correlated

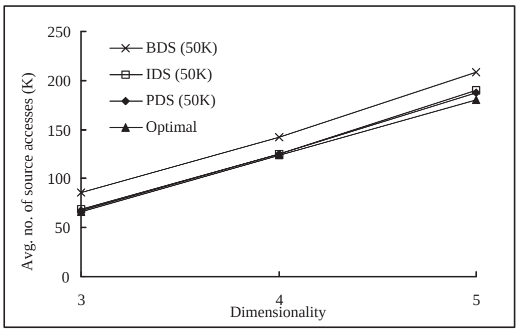
Fig. 11. Average number of source access under uniform dataset (10K objects)

with 50K objects. The distribution of each dimension was randomly picked from the following distributions: Uniform; Normal (Gaussian); and Negative Exponential. Figure 13(a) shows the experimental results. Note that IDS performed significantly worse than BDS in this set of experiments. Since the skyline sizes of the random distribution datasets are similar to that of the independent uniform distribution datasets (see Table 3), the degradation of IDS was solely due to its error in estimating terminating objects. The error was mainly due to the fact that IDS implicitly assumes uniform data distributions, which is not always true in practice. On the other hand, the performance of PDS was still close to Optimal and outperformed both BDS and IDS in the experiments.

We now present the performance of the three algorithms when the values of each attribute are not normalized to the same range in Figure 13(b). We observe that IDS made more source access than PDS because it assumes all attributes are normalized to the same data range, e.g., [0,10]. Again, since the skyline sizes in this setting were close to that of the independent datasets (see Table 3), the degradation of IDS was caused by imprecise prediction of terminating object when the attributes had different value ranges. In contrast, PDS



(a) Independent



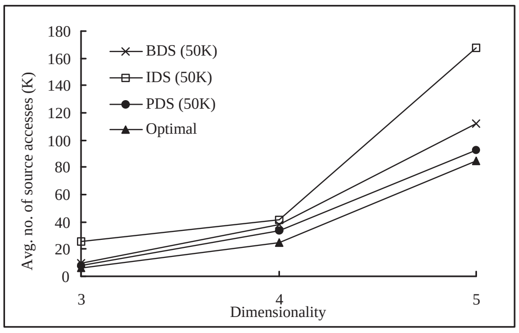
(b) Anti-Correlated

Fig. 12. Average number of source access under uniform dataset (50K objects)

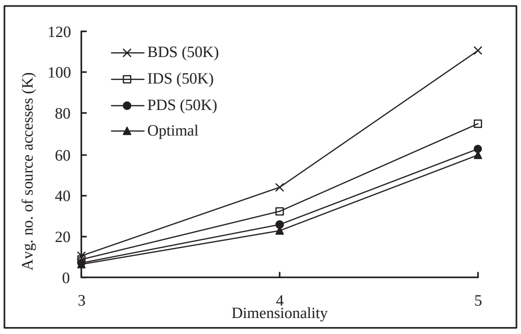
was unaffected by the value ranges because it imposes no such assumptions.

6.3 Progressive behavior

Next we evaluate the speed of the algorithms in returning the skyline points progressively. Figure 14 shows the time for PDS to evaluate a 4-d skyline query verses the percentage of points returned for an anti-correlated dataset with 50K objects. Figure 14 shows that PDS returned the first few skyline objects almost instantaneously. It returned the skyline objects continuously and half of the skyline objects were returned within a minute. Furthermore, PDS outputted the entire skyline before both BDS and IDS reported the first results. The speedup of PDS was brought by the rank estimation and the R*-tree structure. First, the rank estimation of PDS reduced the number of source access effectively and thus the algorithm could be terminated earlier. Second, PDS used R*-tree to reduce the number of comparisons required after the terminating object is found. From the experimental results of both real and synthetic datasets, we assert that the operations of R*-tree in PDS are



(a) Random Distribution



(b) Denormalized Domain

Fig. 13. Average number of source access under various dataset combinations

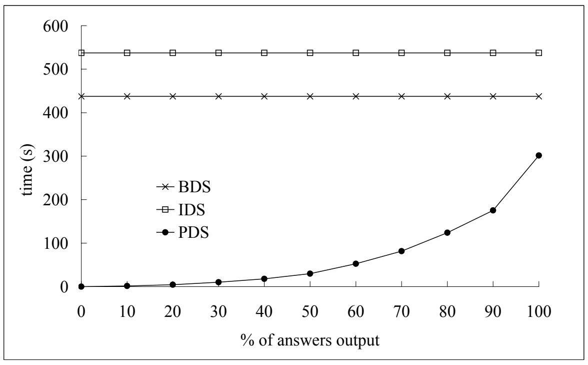


Fig. 14. Progressive behavior (4-d 50k objects, anti-correlated data)

much more efficient than the pairwise comparison approach of BDS and IDS. The running time of IDS was longer than BDS even though it made fewer source access than BDS (see Figure 11(b)) because BDS could discard non-skyline objects in the second phase by using the upper bound information as discussed in Section 3.

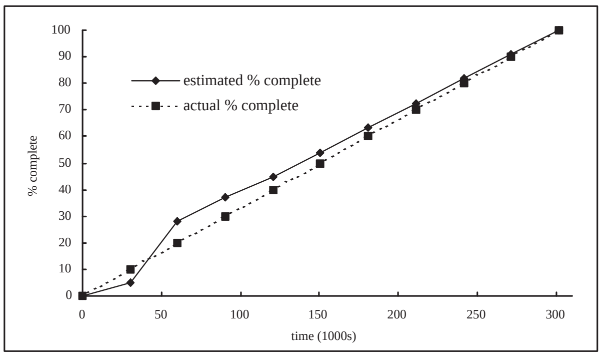


Fig. 15. Estimated completion % vs time (4-d 50k objects, anti-correlated data)

6.4 Effectiveness of progress estimation

Finally, we evaluate the effectiveness of our proposed progress estimation by comparing the estimated completion percentages reported by PDS with the actual percentages. The experimental setting was same as in the previous experiments. In Figure 15, the actual completion percentages of the query over time are shown by the dotted line and the completion percentages estimated by PDS are shown by the solid line. We can see that the estimation of PDS was fairly close to the actual percentages at the beginning and closer in the later stage. The unstable estimation of PDS at the early stage was due to inadequate data for the linear regression method to find a good fit. When more objects were visited by PDS, the regression coefficients continued to be refined and yielded a better result.

7 Conclusion

Skyline queries are important for database applications like decision support and customer information systems. In this paper, we proposed PDS, a novel *progressive* algorithm that computes skyline queries over web-accessible databases. Users execute skyline queries over web data sources by PDS and retrieve the results incrementally. As a result, users could make their decisions in real-time. By making use of a simple estimation based on linear regression and a standard index structure R*-tree, PDS evaluates skyline efficiently in terms of both the number of source access and the computational time over various kinds of data distributions. Furthermore, it does not require any pre-computation (the R*-tree is created on-the-fly) of the remote data and can also be easily extended to evaluate top-K skyline under the web environment. In addition, PDS is user-friendly by supporting a progress estimation such that users know the headway of the query evaluation process and can make early decisions accordingly.

There are several avenues for future work. First, we plan to improve the usability of PDS by allowing users to barter between progressiveness and efficiency. It is because we found that the number of source access could be saved by reporting skyline points less frequently.

Second, distributed query processing on mobile devices has also drawn much attention in recent years [20]. Thus, we plan to evaluate distributed skyline on devices with tight memory requirements (e.g., PDA and cell phone). It would be useful and convenient if users could execute skyline queries through such devices.

Acknowledgment

We would like to thank Benjamin Kao and Nikos Mamoulis for their helpful comments and suggestions. We would also like to thank Marios Hadjieleftheriou for providing his code library for this project.

References

[1] S. Borzsonyi, D. Kossmann, K. Stocker, The skyline operator, in: Proc. of ICDE, 2001, pp. 421–430.

[2] W.-T. Balke, U. Guntzer, J. X. Zheng, Efficient distributed skylining for web information systems, in: Proc. of Extending Database Technology (EDBT), 2004, pp. 256–273.

[3] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over web-accessible databases, in: Proc. of ICDE, 2002, pp. 369–382.

[4] D. Kossmann, F. Ramsak, S. Rost, Shooting stars in the sky: An online algorithm for skyline queries, in: Proc. of VLDB, 2002, pp. 275–286.

[5] D. Papadias, Y. Tao, G. Fu, B. Seeger, An optimal and progressive algorithm for skyline queries, in: Proc. of ACM SIGMOD, 2003, pp. 467–478.

[6] K.-L. Tan, P.-K. Eng, B. C. Ooi, Efficient progressive skyline computation, in: Proc. of VLDB, 2001, pp. 301–310.

[7] J. Matousek, Computing dominances in E^n, Information Processing Letters 38 (5).

[8] D. H. McLain, Drawing contours from arbitrary data points, The Computer Journal 17 (4) (1974) 318–324.

[9] R. Steuer, Multiple criteria optimization, Wiley, 1986.

[10] J. Chomicki, P. Godfrey, J. Gryz, D. Liang, Skyline with presorting, in: Proc. of ICDE, 2003, pp. 717–816.

[11] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems., To appear in ACM Transactions on Databases Systems.

[12] C.-Y. Chan, P.-K. Eng, K.-L. Tan, Stratified computation of skylines with partially-ordered domains, in: To appear in Proc. of ACM SIGMOD, 2005.

[13] X. Lin, Y. Yuan, W. Wang, H. Lu, Stabbing the sky: Efficient skyline computation over sliding windows., in: To appear in ICDE, 2005.

[14] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, P. J. Haas, Interactive data analysis: The control project, IEEE Computer 32 (8) (1999) 51–59.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, in: Proc. of ACM SIGMOD, 1990, pp. 322–331.

[16] G. Luo, J. F. Naughton, C. J. Ellmann, M. W. Watzke, Toward a progress indicator for database queries, in: Proc. of ACM SIGMOD, 2004, pp. 791–802.

[17] S. Chaudhuri, V. R. Narasayya, R. Ramamurthy, Estimating progress of long running sql queries, in: Proc. of ACM SIGMOD, 2004, pp. 803–814.

[18] C. Blake, C. Merz, UCI repository of machine learning databases (1998).
URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`

[19] A. Marian, N. Bruno, L. Gravano, Evaluating top-k queries over web-accessible databases, ACM Transaction on Database System 29 (2) (2004) 319–362.

[20] E. Lo, N. Mamoulis, D. W. Cheung, W. S. Ho, Processing ad-hoc joins on mobile devices, in: Proc. of Database and Expert Systems Applications (DEXA), 2004, pp. 611–621.