Skyline Query Processing for Incomplete Data

Mohamed E. Khalefa Mohamed F. Mokbel Justin J. Levandoski

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{khalefa,mokbel,justin@cs.umn.edu}

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

I. Introduction

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

Due to the importance of skyline queries, several research efforts have been dedicated to develop efficient skyline query processors (e.g., see [2], [3], [4], [5], [6], [7]). Almost all of these algorithms rely mainly on two implicit assumptions: (1) Data are complete, i.e., all dimensions are available for all data items. Such an assumption of completeness is not practical in many cases. For example, consider the movie rating application [1] with hundreds of users rating thousands of movies. It is highly unlikely that every single user will rate all movies. Instead, a user will rate only the movies that interest her. As a result, each movie will be represented as a D-dimensional point with several *blank* (i.e., *incomplete*) dimensions. Another example is from the hotel application where some hotels may not disclose some of their properties. These undisclosed properties are represented as *incomplete* entries within the hotel multi-dimensional point representation. (2) With the exception of [2], all skyline algorithms assume transitivity in the dominance relation, i.e., if data item p_i dominates p_j while p_j dominates p_k, then p_i dominates p_k. Using the transitivity property, skyline query processing algorithms exploit various ways of data pruning and indexing. Unfortunately, as will be seen in this paper, the transitive dominance relation is not applicable to the case of *incomplete* data.

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

We start by introducing two variations of traditional skyline algorithms to accommodate the existence of *incomplete* data, namely, the *Replacement*, and the *Bucket* algorithms. Then, we introduce the *ISkyline* algorithm as a specialized algorithm for

This work is supported in part by the Grant-in-Aid of Research, Artistry, and Scholarship, University of Minnesota, DTC Digital Technology Initiative Program, University of Minnesota, and DTC Intelligent Storage Consortium (DISC), University of Minnesota.

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

- We define the *dominance* relation for *incomplete* data and we show that the transitive dominance relation does not hold for *incomplete* data.
- We introduce two new algorithms, namely, *Replacement* and *Bucket* algorithms that utilize existing skyline algorithms to accommodate *incomplete* data.
- We introduce the *ISkyline* algorithm as a novel algorithm designed specifically for efficient skyline computation over *incomplete* data.
- We provide a proof of correctness for the *ISkyline* algorithm.
- We give experimental evidence that the *ISkyline* algorithm is efficient, scalable, and clearly outperforms the variations of traditional skyline algorithms.

The rest of this paper is organized as follows: Section II highlights related work. Preliminary discussion and problem formulation are given in Section III. Section IV provides two variations of traditional skyline algorithms for *incomplete* data. The *ISkyline* algorithm is introduced in Section V while its proof of correctness is given in Section VI. Section VII gives experimental evidence for the efficiency of our algorithms. Finally, Section VIII concludes this paper.

II. Related Work

The term *skyline queries* has been coined out in the database literature [8] to refer to the secondary storage version of the maximal vector set problem [9], [10]. Since then, several algorithms have been proposed for skyline queries that include no preprocessing solutions (e.g., [8], [11]), presorting solutions (e.g., [3]), and index-based solutions (e.g., [4], [5], [6]). Due to its practicality, several research efforts have been dedicated to developing various *skyline* algorithms for various environments, e.g., partially-ordered domains [12], high-dimensional data [2], [13], [7], skyline cube [7], [14], sliding window [15], [16], continuous skyline computations [17], [18], mobile ad-hoc networks [19], spatial skylines [20], web information systems [21], and data mining [22]. Unfortunately, all these algorithms consider only the case of complete data with no direct extension of considering the case of *incomplete* data where the dominance relation is not transitive.

The closest work to ours is the k-dominant skyline problem [2] in which a point p is considered to dominate point q if only a subset of size k of the dimensions in p dominates the corresponding dimensions in q. Under this definition, the dominance relation turns to be non-transitive, which is the case also for *incomplete* data. The k-dominant skyline algorithm overcomes the non-transitive property by discarding only those points that are dominated in all dimensions while keeping those points that are only dominated in k dimensions. As the k-dominant skyline algorithm considers only the case of complete data, applying it directly to the case of *incomplete* data misses the opportunity to make use of *incomplete* subspaces Thus, applying the k-dominant algorithm directly to the case of *incomplete* data would result in prohibitive costs that can be avoided with the knowledge of *incomplete* dimensions.

III. Preliminaries

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

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

A. *Dominance Relation for* Incomplete *Data*

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

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

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

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

B. Bitmap Representation

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

Definition 2: Two points P and Q are comparable if and only if the bitwise-and of their bitmaps is not zero.

IV. Using Traditional Skyline Algorithms for Incomplete Data

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

The *Replacement* Algorithm. The main idea of the *Replacement* algorithm is to replace any *incomplete* dimension in a data item by $-\infty$. By doing so, all *incomplete* dimensions are transformed to *complete* dimensions. Then, we can apply any traditional skyline algorithm to get the set $S_{sky-\infty}$ of

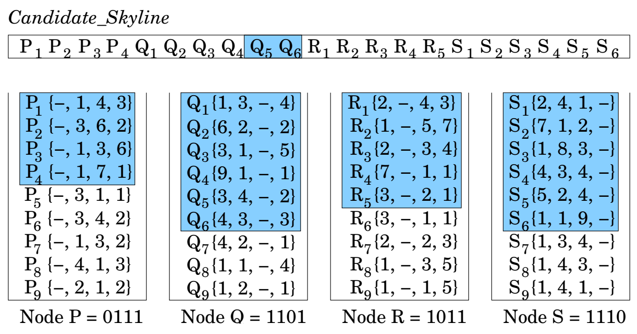


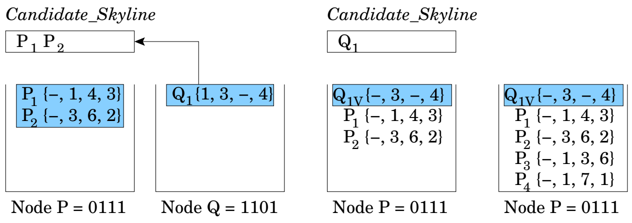
Fig. 1. The Bucket algorithm

current skylines. Then, for all points in $S_{sky-\infty}$, we *replace* the $-\infty$ values by an *incomplete* dimension, i.e., return to the original form. Finally, we perform an exhaustive pairwise comparison between all *incomplete* points that are in $S_{sky-\infty}$ to find the actual skyline points. The *replacement* algorithm greatly improves upon the naive method as the exhaustive pairwise comparison is done only for those points in $S_{sky-\infty}$ rather than all input data points.

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

The *Bucket* Algorithm. The main idea of the *Bucket* algorithm is to divide all incoming points into distinct buckets where all points in each bucket have the same bitmap representation. By doing so, the transitive dominance relation would hold among all points within the same bucket. Then, we can apply a traditional skyline algorithm over all points within each bucket by ignoring the *incomplete* dimensions. We would call the set of skylines for each bucket as a *local* skyline. Finally, we collect the points from all *local* skyline sets and include them in one list, termed *candidate* skyline, list in which we perform an exhaustive pairwise comparison among all points to get the query answer. The correctness of the *Bucket* algorithm comes from the fact that for a point to be a skyline, it has first to be a *local* skyline among all points in its bucket. Also, if a point P_i is a *local* skyline in bucket P, it needs to be compared only against all *local* skyline of other *comparable* buckets.

Figure 1 gives an example of the *Bucket* algorithm in which 36 points are divided into four buckets, P, Q, R, and S based on their bitmaps. For each bucket, we compute the *local*



(a) $P_1 P_2$ are skylines, Q_1 arrives (b) Inserting Q_{1V} in P (c) P_3 P_4 arrive

Fig. 2. Virtual point insertion

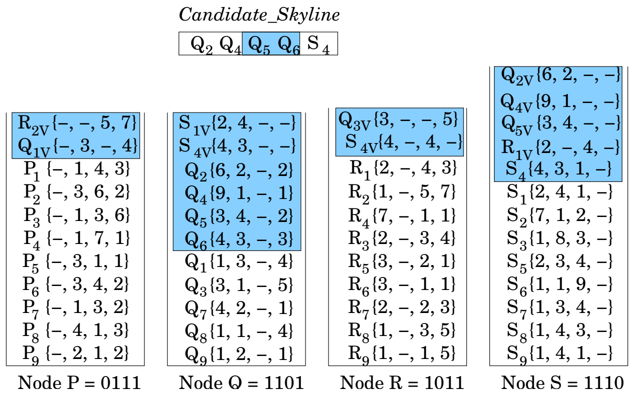


Fig. 3. Final effect of Virtual points

skylines separately, depicted by shaded rectangle in Figure 1. Overall, we have 21 *local* skyline points that are considered as *candidate* skylines in which we perform an exhaustive pairwise comparison to conclude that Q_5 and Q_6 are the skylines over all 36 points.

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

V. Efficient Skyline Computation for Incomplete Data

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

A. Virtual Points and Shadow Skylines

Virtual Points. The main purpose of *virtual points* is to reduce the number of points in the *candidate* skyline list. The main idea is that a point X in a bucket N_i can be used to reduce the number of *local* skylines in a bucket N_j where $i \neq j$. By doing so, the number of *local* skyline at each bucket can be reduced, and hence, the number of *candidate* skylines can be reduced significantly. Figure 2 illustrates the idea of *virtual points* when applied to the example in Figure 1. In this case, we compute the *local* skyline for each bucket and the *candidate* skyline online while we read the input data, as no pre-processing is assumed. The current *local* skyline for node P is P_1 and P_2; these points are also inserted into the *candidate* skyline list. Figure 2a shows the time instance in which we read Q_1 as a skyline point for node Q. In this case, we compare Q_1 against all points in the *candidate* skyline list that have *comparable* bitmaps to that of Q_1, i.e., P_1 and P_2. Since Q_1 dominates both points, we remove P_1 and P_2 from the *candidate* list while keeping only Q_1. Notice that, up to now, this scenario is also applicable to the *Bucket* algorithm.

However, the *ISkyline* algorithm distinguishes itself as it creates a *virtual point* Q_{1v} out of Q_1. Q_{1v} will be inserted in node P to reduce the number of *local* skylines. The main idea is that for an incoming point P_x to be a *local* skyline in P, it must not be dominated by Q_{1v}. Q_{1v} is formed by considering only the common dimensions in the bitmaps of nodes P and Q. Figure 2b gives an example where Q_{1v} is formed as $(-, 3, -, 4)$ and inserted into P. Notice that currently, the *local* skyline of P includes only Q_{1v}. Figure 2c gives the process of reading input points P_3, P_4, and P_5. Since all points are dominated by the *virtual point* Q_{1v}, we ignore P_3 , P_4 and P_5 by neither storing them as *local* skylines nor propagating them to be *candidate* skylines. By doing so, we significantly reduce the size of the *candidate* skyline list. For example, compare the *local* skyline list of P in Figure 2c that includes only one point, Q_{1v}, to that of Figure 1 that includes four points P_1, P_2, P_3, and P_4. Moreover, as Q_{1v} is a *virtual point*, it is not propagated to the *candidate* skyline list. So, in the *ISkyline* algorithm (Figure 2), none of the points in P becomes *candidates*, while in the *Bucket* algorithm (Figure 1), four points from P are *candidates*.

Figure 3 gives the end result of *local* skylines at each bucket and the *candidate_skyline* list after reading all the input data and employing the *virtual point* concept. It can be seen that employing the *virtual point* concept reduces the size of the *candidate_skyline* list to 5 instead of 26, as in the *Bucket* algorithm.

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

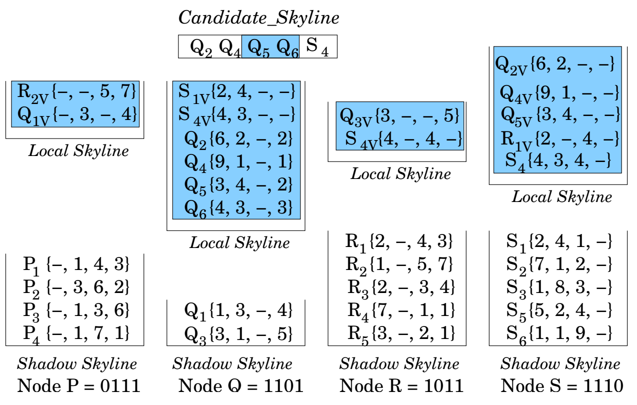


Fig. 4. Final effect of Shadow Skyline

skyline as Y may help later in dominating *candidate* skylines. For example, in Figure 3, although P_3 is not stored in the *local* skyline list of P, it dominates Q_4 from the *candidate* list. To see how this scenario may take place, consider the case of Figure 2c in which P_3 is not stored in the *local* skyline list as it is dominated by Q_{1v}. Whenever Q_4 arrives, Q_4 will not be compared to P_3 as P_3 is not a *local* skyline. Thus, Q_4 will be stored as a *candidate* skyline although it is dominated by P_3. This scenario does not affect the correctness of the algorithm, however, it causes an overhead of not being able to discard any dominated points. Thus, for bucket P, we store all points P_1 to P_9, instead of storing only P_1 to P_4 as in the *Bucket* algorithm. It is important to note that even with this side effect, *virtual points* still perform better than the *Bucket* algorithm as the savings in the size of *candidate* and *local* skylines is much more powerful than the drawback of storing and comparing all points.

The *ISkyline* algorithm introduces the concept of *shadow skylines* that works together with *virtual points* to alleviate the problem of storing and comparing all input data. The main idea is that we do not need to store all points in each bucket, instead, we only need to store the skyline set of points not found in the local skyline list. For example, in bucket P, instead of storing all points P_1 to P_9, we need to store only P_1 to P_4 as these are the skyline points of P_1 to P_9. In this case, we will call P_1 to P_4 as the *shadow* skyline of P. The *shadow skyline* of a bucket N is the set of points that are real skylines among all points in N minus those points that are stored in the *local* skyline of N. For example, consider bucket Q in Figure 1, where the original skyline set includes points Q_1 to Q_6. However, with *virtual points* (Figure 3), only points Q_2, Q_4, Q_5, and Q_6 are stored in the *local* skyline set of Q. Thus, the *shadow* skyline of Q includes Q_1 and Q_3. Figure 4 gives the list of *local* and *shadow* skylines of each bucket. Points that are not shown in the figure are discarded by the *ISkyline* algorithm. By doing so, the storage increase of the *ISkyline* algorithm over the *Bucket* algorithm is limited only to the *virtual points*. Moreover, as we will show in the algorithm details, the *candidate* skylines need to be compared only against points in the *shadow* skyline rather than all points.

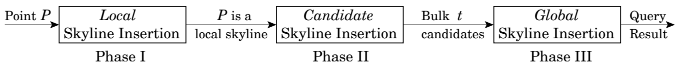


Fig. 5. Phases of the ISkyline Algorithms

B. *The* ISkyline *Algorithm*

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

Data structure. With each bucket node N associated with the bitmap $N.B$, we store three pieces of information: (1) The *local_skyline* list that may contain both real and *virtual* points as shown in Figure 4, (2) The *shadow_skyline* list that contains only real data points as shown in Figure 4, and (3) A flag *updated* that is turned on only when the *shadow_skyline* list is modified. Such flag significantly prunes the search space by avoiding looking at unmodified buckets. It is important to note that the number of buckets we maintain is the same as the number of distinct bitmaps of all input data. To access bucket nodes by their bitmaps, we maintain a hash table with the entry $<$ *bitmap*, *node pointer*$>$ that associates each available bitmap with one bucket node. Finally, we maintain two lists, *candidate_skyline* and *global_skyline* that maintain current *candidate* skylines and the query result, respectively.

Figure 5 gives an overview of the *ISkyline* algorithm that reads data sequentially from an input file with no assumptions about index availability or data preprocessing. The main idea is that each input point P may pass through *up to* three phases (depicted by rectangles in Figure 5). In Phase I, for each point P in node N, we check if P needs to be (a) stored in the *local_skyline* list of N, (b) stored in the *shadow_skyline* list of N, or (c) completely discarded. Only those points that are stored in the *local_skyline* list go onto Phase II. For each point P in Phase II, we check if P needs to be stored in the *candidate_skyline* list. This phase will also determine whether *virtual points* should be inserted in other node buckets based on the comparison of P with other points in the *candidate_skyline* list. Once we have t points in the *candidate_skyline* list, we move to Phase III where we update the list of points in the *global_skyline* list, i.e., the current query answer.

Algorithm 1 gives the pseudo code of the *ISkyline* algorithm. The input to the algorithm is: (1) a set S of data points and (2) the tuning parameter t. The output of the algorithm is the set of *global* skylines. The algorithm starts by initializing the *global_skyline* and *candidate_skyline* lists. Then, for every input point $P \in S$, we either retrieve its corresponding node N from the hash table or create N if it does not exist (Lines 5 to 6 in Algorithm 1). Then, we start in Phase I where we attempt to insert P into the *local_skyline* list (Line 7 in Algorithm 1). If P ends up to be a *local* skyline of N, we start

Algorithm 1 Skyline Computation for Incomplete Data

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

Algorithm 2 Phase I: *Local* Skyline Insertion

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

Phase II where we attempt to add P to the *candidate_skyline* list (Line 9 in Algorithm 1). Whenever the number of points in the *candidate_skyline* list exceeds t, we move to Phase III where we update the list of *global skylines* and clear the *candidate_skyline* list (Lines 11 to 12 in Algorithm 1). Finally, once we reach to the end of input, we update the list of *global skylines* and conclude the algorithm (Line 16 in Algorithm 1). In the rest of this section, we will discuss in details the three phases of the *ISkyline* algorithm.

1) Phase I: Local *Skyline Insertion:* Algorithm 2 gives the pseudo code of Phase I in which, for each point P, we either store it in the *local_skyline* list, store it in the *shadow_skyline* list, or discard it. Basically, we check if P is not dominated by any point in the *local_skyline* list. If this is the case, we decide to store P in the *local_skyline* list, update the entries in both the *local_skyline* and *shadow_skyline* lists accordingly, and return *true* to indicate that P is a *skyline* for node N (Lines 3 to 5 in Algorithm 2). It is important to note that we do not remove *virtual points* from the *local_skyline* even those *virtual points* are dominated by P. The main idea is that those virtual points may dominate other points that cannot be dominated by P. For example, in Figure 4, although point S_4 dominates the virtual point R_{1v}, we did not remove R_{1v}. By doing so, R_{1V} later dominated point S_3 which is not dominated by S_4. Thus, we intentionally do not remove *virtual points* as they could help in reducing the search space for *local* and *candidate* skylines. On the other hand, if P ends up to be dominated by some point in the *local_skyline* list, we check if P is dominated only by *virtual points*. If this is the case, we decide to insert P in the *shadow_skyline* list of N, set the *updated* flag of N to be *true* to indicate a change in the *shadow_skyline* list, update the list of *shadow* skylines accordingly, and return *false* (Lines 6 to 11 in Algorithm 2). It is important to note that by being dominated by *virtual points* only, P is considered to be a *skyline* among all current points of the same bitmap. That is why we keep P in the *shadow* list. Finally, If P was dominated by at least one real point from the *local_skyline* list of N, we simply discard P and return *false*.

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

Inserting a *virtual point* P into a node N is mainly performed in three steps: (1) All *real* points in the *local_skyline* list of N that are dominated by P are moved to the *shadow_skyline* list of N. For example, consider Figure 2a; when we insert Q_1 as a *virtual point* in P, we find that Q_1 dominates both P_1 and P_2, thus, we move P_1 and P_2 to the *shadow_skyline* list as depicted in Figure 4. (2) All *virtual* points in the *local_skyline* list of N that are dominated by P and have complete dimensions that are a superset of, or same as P's complete dimensions are removed. The main idea is that if two *virtual points* P_{iv} and P_{jv} have the same bitmap and P_{iv} dominates P_{jv}, then there is no need to store P_{jv} as any point that will be dominated by P_{jv} will also be dominated by P_{iv}. Similar arguments hold for the case of P_{jv} having a superset bitmap of P_{iv}. (3) We insert a *virtual point* P_v in the *local_skyline* list of N. P_v is created by copying the values from P for *only* the common dimensions of P and N bitmap while having *incomplete* in other dimensions.

3) Phase III: Global *Skyline Insertion:* Algorithm 4 gives the pseudo code of Phase III in which we propagate qualified points from being *candidate* skylines to be *global* skylines. At the same time, we *validate* existing *global* skyline points

Algorithm 3 Phase II: *Candidate* Skyline Insertion

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

against newly incoming points that were read since the last computation of the *global* skyline. The algorithm mainly has four steps: (1) Checking existing *candidate* and *global* points against each other for the dominance relation to remove any points that are dominated in any of these two lists (Lines 2 to 5 in Algorithm 4). The main idea of this step is to early prune those dominated points as there is no point in processing them further with the following expensive steps. (2) All remaining points in the *global_skyline* list are compared against all *shadow_skyline* lists of *comparable* but not equal nodes with a true *updated* flag. If at least one point in the compared *shadow_skyline* lists dominates a point P in the *global_skyline* list, we immediately delete P from the *global* skylines (Lines 6 to 10 in Algorithm 4). Notice the importance of the *updated* flag as an *optimization* technique that avoids comparing with *shadow_skyline* lists that did not change recently. Also, it is important to note that we do *not* need to compare *global* skyline points against the *local_skyline* list of comparable nodes as any *real* point in the *local_skyline* list is also stored in the *candidate_sklyine* list and hence it has been considered through the first step. (3) This step aims to process remaining points in the *candidate_skyline* list in the same way as points in the *global_skyline* list are processed in the second step with the *exception* that points in the *candidate_skyline* list are compared against *all* comparable nodes regardless of the status of the *updated* flag (Lines 11 to 15 in Algorithm 4). The main idea for ignoring the *updated* flag is that points in the *candidate_skyline* list have recently arrived, and thus, are not compared yet with points in the *shadow_skyline* lists. (4) Finally, the *global_skyline* list (i.e., the current query answer) is formed by combining all remaining *candidate* and *global* skylines together. Also, we reset all *updated* flags to *false* to indicate that the current answer is up to date. (Lines 16 to 17 in Algorithm 4). It is important to note that throughout Algorithm 4, deleting a point from either the *candidate* or the *global* lists indicates that the point is stored in the *shadow_skyline* list of its corresponding node.

VI. Proof of Correctness

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

Algorithm 4 Phase III: *Global* Skyline Insertion

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

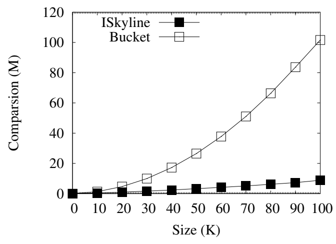
Theorem 1: Any point P that is a skyline over all input data items, will be reported by the *ISkyline* algorithm

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

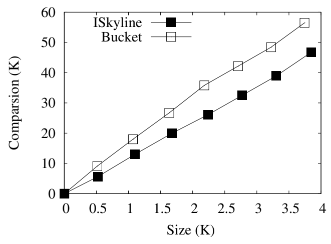
■

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

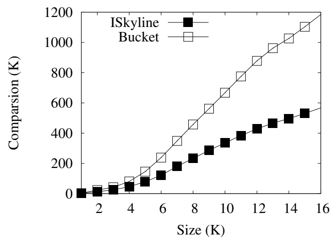
Proof: Assume that there exists a point P that is reported from the *ISkyline* algorithm, however, there exist another real point Q in the input data set that dominates P, i.e., P is not a true skyline. As point P is reported as a result, it is stored in the *global_skyline* list. On the other hand,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



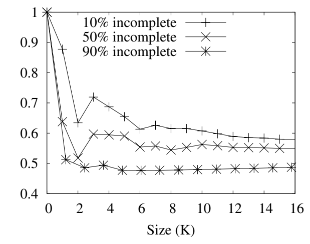
(a) Synthetic



(b) MoviesLens



(c) NBA



(d) NBA (*incomplete* ratio)

Fig. 6. Scalability

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

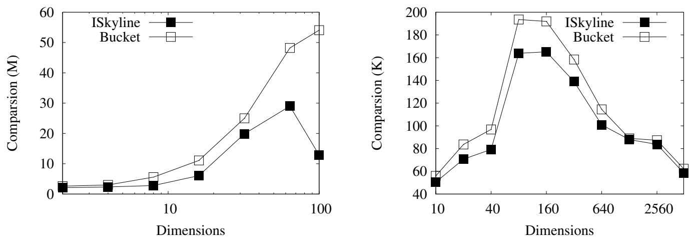
VII. Experimental Results

This section experimentally evaluates the performance of the proposed algorithms. As this is the first attempt for skyline query processing in *incomplete* data, we could not compare with any previous technique. Also, initial experiments show that our proposed *Replacement* algorithm performs severely worse than our proposed *Bucket* and *ISkyline* algorithms. This is mainly due to the fact that the skyline set $S_{-\infty}$ in the *Replacement* algorithm is of a very large size. So, in this section, we focus only in the performance analysis and comparison of both the *Bucket* and *ISkyline* algorithms. Our test bed includes three data sets: (1) MovieLens [1]. This is a real data set of 3900 points where each point is of 6000 dimensions that represent the user ratings (6000 users) of 3900 movies. There is only about 1 Million reviews, which means that this data set is 95% *incomplete*, i.e., only 5% of the ratings are available. (2) NBA [23]. This is a real data set containing records for 16,000 NBA players where each record has 17 dimensions representing various statistics about basketball skills. The NBA data is rather complete, however, we explicitly remove values in order to test the performance of our algorithms. Removed values represent missing statistics about the players for some years. Unless mentioned otherwise, the default incomplete percentage for the NBA dataset is 20%. (3) Synthetic. We generated a 20% incomplete synthetic data set of 100,000 points, each with 100 dimensions.

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

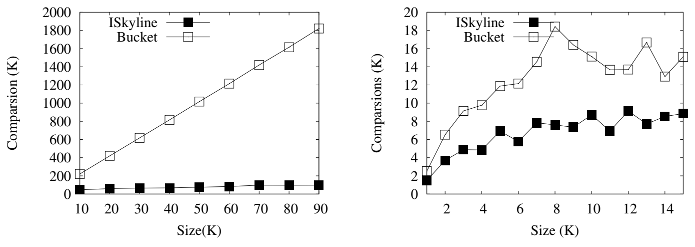
A. Scalability

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



(a) Synthetic dataset (b) Movies

Fig. 7. Data Dimensionality



(a) Synthetic dataset (b) NBA

Fig. 8. Incremental behavior

mance between *ISkyline* and *Bucket* in MovieLens indicates the number of buckets that get benefit from *virtual points* and *shadow_skyline*. For NBA data (Figure 6c), similar to other data sets, *ISkyline* steadily outperforms *Bucket*. Notice that in NBA data set, the number of *comparable* buckets would be between the Synthetic and the MovieLens data. So, the superiority of *ISkyline* over *Bucket* is more than the case of MovieLens but less than the case of Synthetic.

B. Ratio of Completeness.

Figure 6d gives the effect of the ratio of *incomplete* entries on the performance of *ISkyline* over *Bucket*. We plot the number of comparisons incurred by *ISkyline* as a ratio from that of *Bucket*. We also plot three entries of *ISkyline* with *incomplete* ratios of 10%, 50%, and 90%. It is clear that with the increase of the ratio of *incomplete* data, *ISkyline* would be a better enhancement over *bucket*, i.e., the ratio of the number of comparisons is decreased. The main reason for this is that with more *incomplete* data, *virtual points* can reduce the number of *local* skylines at each bucket and the overhead needed to update the list of *global* skylines. Such role of *virtual points* becomes more clear with the increase of the *incompleteness* ratio. This experiment reflects the fact that *ISkyline* is designed specifically with the *incompleteness* problem in mind while *Bucket* uses an adaptation of existing skyline algorithms to accommodate *incomplete* data. It is important to note that the performance ratio between *ISkyline* and *Bucket* is stable with the increase of data size.

C. Data Dimensionality

Figure 7 gives the effect of increasing the dimensionality (represented by a log scale) on the performance of both *ISkyline* and *Bucket* for Synthetic and MovieLens datasets. As in previous experiments, *ISkyline* steadily outperforms *Bucket* for up to 100 dimensions in the Synthetic data and 5000 dimensions in MovieLens data. In both data sets, the number of required comparisons by *ISkyline* rises up for medium dimensions (i.e., 100-500 dimensions) and then goes down for higher dimensions. The main reason is that the performance depends mainly on the *comparability* of data items. If most data items are comparable with each other, the performance will be worse. With few dimensions, high ratio of the *incomplete* data will be removed from the input as a data item may include only *incomplete* dimensions. Thus, the number of *comparable* of points would be less. With the increase of dimensions, the *comparability* ratio increases till we reach to a peak point. Then, with the increase of dimensions, the number of possible buckets would increase, and hence the data items would have different buckets with different bitmaps, reducing the comparability ratio.

D. Incremental behavior

Figure 8 gives the *incremental* behavior of both *ISkyline* and *Bucket*. We measure the number of comparisons needed to “refresh” the query answer after adding 1,000 new data items for both synthetic and NBA data. Due to its *incremental* properties, managed by the *updated* flag, *ISkyline* clearly outperforms *Bucket* in all cases. This indicates that *ISkyline* smartly avoids reevaluation and redundant processing that are done by *Bucket* to maintain the current answer of *global* skylines up to date. It is important to note also that for large data sizes, e.g., more than 50K in Figure 8a, adding 1K of data by the *ISkyline* would have the same cost regardless of the current data size, while in *Bucket*, the cost will be increased linearly with the increase of data size. This is mainly due to the fact that the metadata stored as *virtual points*, *shadow skylines*, and *updated* flag aid *ISkyline* to focus only on the new 1K additions of data rather than reconsidering all data as in the case of *Bucket*.

VIII. Conclusion

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

REFERENCES

[1] "http://movielens.umn.edu."

[2] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang, "Finding k-Dominant Skylines in High Dimensional Space," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2006.

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2003.

[4] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems, TODS*, vol. 30, no. 1, pp. 41–82, 2005.

[6] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient Progressive Skyline Computation," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2001.

[7] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang, "Efficient Computation of the Skyline Cube," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2005.

[8] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2001.

[9] H. T. Kung, F. Luccio, and F. P. Preparata, "On Finding the Maxima of a Set of Vectors," *Journal of ACM*, vol. 22, no. 4, pp. 469–476, 1975.

[10] J. Matousek, "Computing Dominances in E^n," *Information Processing Letters*, vol. 38, no. 5, pp. 277–278, 1991.

[11] P. Godfrey, R. Shipley, and J. Gryz, "Maximal Vector Computation in Large Data Sets," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2005.

[12] C. Y. Chan, P.-K. Eng, and K.-L. Tan, "Stratified Computation of Skylines with Partially-Ordered Domains," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2005.

[13] J. Pei, W. Jin, M. Ester, and Y. Tao, "Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2005.

[14] Y. Tao, X. Xiao, and J. Pei, "SUBSKY: Efficient Computation of Skylines in Subspaces," in *Proceedings of the International Conference on Data Engineering, ICDE*, Atlanta, GA, Apr. 2006.

[15] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the Sky: Efficient Skyline Computation over Sliding Windows," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2005.

[16] Y. Tao and D. Papadias, "Maintaining Sliding Window Skylines on Data Streams," *IEEE Transactions on Knowledge and Data Engineering, TKDE*, vol. 18, no. 2, pp. 377–391, 2006.

[17] Z. Huang, H. Lu, B. C. Ooi, and A. K. Tung, "Continuous Skyline Queries for Moving Objects," *IEEE Transactions on Knowledge and Data Engineering, TKDE*, vol. 18, no. 12, pp. 1645–1658, 2006.

[18] M. D. Morse, J. M. Patel, and W. I. Grosky, "Efficient Continuous Skyline Computation," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2006.

[19] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi, "Skyline Queries Against Mobile Lightweight Devices in MANETs," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2006.

[20] M. Sharifzadeh and C. Shahabi, "The Spatial Skyline Queries," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[21] W.-T. Balke, U. Güntzer, and J. X. Zheng, "Efficient Distributed Skylining for Web Information Systems," in *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2004.

[22] W. Jin, J. Han, and M. Ester, "Mining Thick Skylines over Large Databases," in *European Conference on Principles and Practice of Knowledge Discovery in Databases, PKDD*, 2004.

[23] "http://www.basketball-reference.com/."