Computing Exact Skyline Probabilities for Uncertain Databases

Dongwon Kim, Hyeonseung Im, and Sungwoo Park

Abstract—With the rapid increase in the amount of uncertain data available, probabilistic skyline computation on uncertain databases has become an important research topic. Previous work on probabilistic skyline computation, however, only identifies those objects whose skyline probabilities are higher than a given threshold, or is useful only for 2D data sets. In this paper, we develop a probabilistic skyline algorithm called *PSkyline* which computes exact skyline probabilities of all objects in a given uncertain data set. PSkyline aims to identify blocks of instances with skyline probability zero, and more importantly, to find incomparable groups of instances and dispense with unnecessary dominance tests altogether. To increase the chance of finding such blocks and groups of instances, PSkyline uses a new in-memory tree structure called *Z-tree*. We also develop an online probabilistic skyline algorithm called *O-PSkyline* for uncertain data streams and a top-k probabilistic skyline algorithm called *K-PSkyline* to find top-k objects with the highest skyline probabilities. Experimental results show that all the proposed algorithms scale well to large and high-dimensional uncertain databases.

Index Terms—Skyline computation, skyline probability, uncertain database, data stream

✦

1 INTRODUCTION

WITH the rapid increase in the amount of uncertain data available, query processing on uncertain databases has become an important research topic in the database community. Because of the presence of data uncertainty and the accompanying computational complexity, query processing on uncertain databases calls for different approaches from query processing on conventional databases, as shown in probabilistic range queries [1], [2], probabilistic ranking queries [3], [4], [5], [6], [7], and probabilistic skyline queries [8], [9], [10], [11], [12]. With these advances in query processing, uncertain databases have been successfully adopted in a wide range of real-world applications (see [13] for a survey).

This paper revisits probabilistic skyline computation on uncertain data sets as proposed by Pei et al. [8]. Given a certain data set of tuples, skyline computation returns a subset of tuples that are no worse than, or not dominated by, any other tuples when all dimensions are considered together. We refer to such a subset of tuples as the skyline set. Given an uncertain data set in which the probability distribution of an object is represented by a weighted set of tuples called instances, probabilistic skyline computation finds for each object the probability of its being in the skyline set, or its skyline probability. Pei et al. propose two practical algorithms for *p-skyline computation* which identifies those objects whose skyline probabilities are higher than a given threshold p.

- *The authors are with the Department of Computer Science and Engineering, Pohang University of Science and Technology, San 31 Hyoja-dong, Nam-gu, Pohang, Gyeongbuk, 790-784, Republic of Korea. E-mail: {eastcirclek, genilhs, gla}@postech.ac.kr.*

Although p-skyline computation is useful in identifying those objects that are likely to be superior to all other objects, it has an important limitation: the user should specify a threshold p in order to control the size of the resultant set. This limitation is further exacerbated by the dependency between the distribution of objects in the data set and the size of the resultant set, which makes it difficult to choose an appropriate value of p for retrieving a desired number of objects.

In addition, p-skyline computation is not applicable to uncertain data streams, in which skyline probabilities of all objects vary over time.

Below we illustrate the limitation of p-skyline computation and explain why it is desirable to compute exact skyline probabilities, both on uncertain data sets and on uncertain data streams. We use an NBA database which contains 260,559 records about 1,092 players from the 1999 season to the 2008 season (http://www.basketball-reference.com). We regard each player as an object and his record in each game as an instance of the object. Every instance is assigned a weight that is proportional to the playing time.

1.1 Why Exact Skyline Probabilities?

We wish to identify five players with the highest skyline probabilities when three key attributes are considered—number of points, number of assists, and number of rebounds. (Hence, we consider the NBA database as a 3D uncertain data set.) After trying several values for p, we obtain the five players in Fig. 1a by setting p to 0.31. Note that the skyline probability of each player in the right column is not part of the result of p-skyline computation and has been computed separately for reference only.

Now, we wish to identify five players with the highest skyline probabilities when two additional attributes are considered—number of steals and number of blocks. (Hence, we now consider the NBA database as a 5D uncertain data set.) Unfortunately, we obtain 24 players if we try the same threshold value of $p = 0.31$ again. Since we know neither the

LeBron James	0.420362		LeBron James	0.641458
Kevin Garnett	0.395696		Kevin Garnett	0.640180
Chris Paul	0.367919		Tim Duncan	0.572058
Jason Kidd	0.334187		Dwyane Wade	0.536603
Shaquille O'Neal	0.311626		Chris Paul	0.531069
(a) $d = 3$			(b) $d = 5$	

Fig. 1. Top five players when three and five attributes are considered.

Fig. 2. Change in skyline probabilities of four NBA players $(d = 3)$.

skyline probability of each player nor the distribution of objects in the data set, we need to randomly choose a higher value for p to limit the size of the p-skyline set. After numerous trials, we eventually obtain the five players in Fig. 1b by setting p to a value between 0.516076 and 0.531069. Note that we still do not know which player has the highest skyline probability because the skyline probability of each player needs to be computed separately.

Thus p-skyline computation is useful only if we have a good scheme for analyzing an uncertain data set and choosing a suitable value for p, which can be as difficult as p-skyline computation itself. We can overcome this limitation of p-skyline computation by computing exact skyline probabilities of all objects or top-k objects with the highest skyline probabilities. The recent work by Atallah and Qi [11] presents a probabilistic skyline algorithm for exact skyline probability computation whose worst case time complexity is subquadratic in the size of the data set, but it is useful only for 2D data sets because of a hidden factor exponential in the dimensionality of the data set.

1.2   Why Online Probabilistic Skyline Computation?

While computing exact skyline probabilities is an improvement over p-skyline computation on uncertain data sets, it is not suitable for uncertain data streams in which skyline probabilities of all objects vary over time. For such uncertain data streams, online probabilistic skyline computation is more useful which shows the change in the skyline probability of each object at every time stamp.

Consider again the previous NBA database. If we consider it as an uncertain data set, the skyline probability of each player only reflects his average performance over the period of 10 years from 1999 to 2008. If we consider it as an uncertain data stream, however, online probabilistic skyline computation reveals the fluctuation in the performance of each player. Fig. 2 shows partial results of online probabilistic skyline computation over the same period when the same three attributes as in Fig. 1a are considered. (We use a window size of 200, i.e., the skyline probability is calculated over the past 200 games.) We make the following observations:

- LeBron James and Kevin Garnett, who have similar skyline probabilities over the whole period (LeBron James 0.420362 and Kevin Garnett 0.395696), exhibit very different patterns: LeBron James continually maintains high skyline probabilities while skyline probabilities of Kevin Garnett decrease sharply in the last two years.
- Jason Kidd and Shaquille O'Neal also have similar skyline probabilities over the whole period (Jason Kidd 0.334187 and Shaquille O'Neal 0.311626), but Jason Kidd maintains his performance steadily,
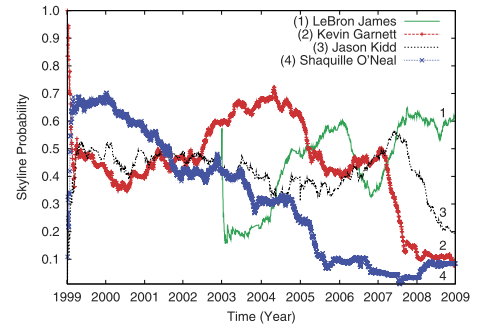
while Shaquille O'Neal is gradually declining in performance.

Thus, by performing online probabilistic skyline computation, we observe the change in the performance of a particular player over a particular period of time, which is new information not yielded by computing exact skyline probabilities only once over the whole period.

1.3   Contributions

The contributions of this paper are threefold.

1.3.1   A Probabilistic Skyline Algorithm PSkyline

We present a probabilistic skyline algorithm *PSkyline* (*P*robabilistic *Skyline*) which computes exact skyline probabilities of all objects in a given uncertain data set. As shown in the recent skyline algorithm by Zhang et al. [14], reducing the number of dominance tests between incomparable tuples is critical to the overall speed of skyline computation on certain data sets. PSkyline extends the same design principle to uncertain data sets—we aim to find incomparable *groups* of instances and dispense with unnecessary dominance tests altogether. PSkyline is a general framework accommodating conventional tree structures such as R-trees [15] and ZBtrees [16], and uses two recursive traversal functions to find incomparable groups of instances.

1.3.2   A New in-Memory Tree Structure Z-Tree

We propose an in-memory tree structure called *Z-tree* which is designed specifically for PSkyline. Ztrees are similar to ZBtrees but exploit Z-addresses [17] in a novel way to find as many incomparable groups of instances as possible. Experimental results show that Z-trees are particularly suitable for probabilistic skyline computation. For example, Z-trees enable us to eliminate up to 99.5 percent of dominance tests on typical uncertain data sets just by finding incomparable groups of instances, and PSkyline with Z-trees runs consistently faster than with R-trees or ZBtrees regardless of the size (up to 20,000 objects with about 4,000,000 instances) and the dimensionality (up to 10) of the data set.

1.3.3   Two Variants of PSkyline—O-PSkyline and K-PSkyline

We also develop two variants of PSkyline for online probabilistic skyline computation and top-k probabilistic skyline computation. *O-PSkyline* (*O*nline *PSkyline*) continuously updates skyline probabilities of all objects in a given uncertain data stream. *K-PSkyline* (top-K *PSkyline*) identifies
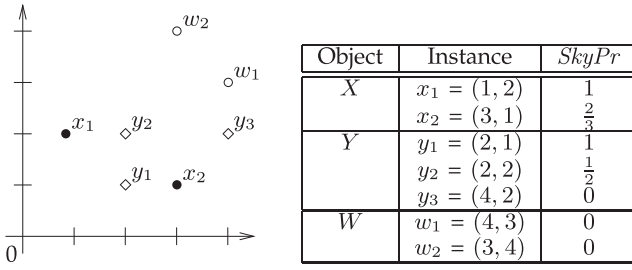
Fig. 3. An uncertain data set and skyline probabilities of instances.

k objects with the highest skyline probabilities from a given uncertain database, without computing skyline probabilities of all objects. Similarly to PSkyline, both O-PSkyline and K-PSkyline maintain tree structures to store instances and avoid unnecessary dominance tests by finding incomparable groups of instances.

1.4 Organization of the Paper

Section 2 introduces probabilistic skyline computation on uncertain databases and Section 3 introduces Z-trees. Section 4 discusses related work and explains the difference between Z-trees and ZBtrees [16]. Sections 5, 6, and 7 present the design of PSkyline, O-PSkyline, and K-PSkyline, respectively. Section 8 presents experimental results and Section 9 concludes.

2 PRELIMINARIES

This section defines probabilistic skyline computation on uncertain data sets and explains its basic properties as observed by Pei et al. [8]. It also defines online probabilistic skyline computation and top-k probabilistic skyline computation.

2.1 Skyline Computation

Given a d-dimensional certain data set, a skyline query retrieves its skyline set consisting of skyline tuples which are not dominated by any other tuples. Under the assumption that smaller values are better, a tuple x dominates another tuple y, written $x \prec y$, if $x[i] \leq y[i]$ holds for $1 \leq i \leq d$ and there exists a dimension k such that $x[k] < y[k]$, where $x[i]$ denotes the ith element of tuple x. We write $x \not\prec y$ to mean that x does not dominate y, $x \prec\!\succ y$ to mean that x and y are incomparable ($x \not\prec y$ and $y \not\prec x$), and $x \preceq y$ to mean that either $x \prec y$ or $x = y$ holds. Then, the skyline set of a certain data set D is defined as

$$\{x \in D \mid y \not\prec x \text{ if } y \in D\}.$$

For example, if we interpret the uncertain data set in Fig. 3 as a certain data set, we obtain the skyline set $\{x_1, y_1\}$.

We can also define dominance relations between tuples and data sets. $x \prec D$ means that x dominates every tuple in D. Conversely $D \prec x$ means that every tuple in D dominates x. We write $D \prec D'$ to mean that every tuple in D dominates all tuples in D'. We also write $D \prec\!\succ D'$ to mean that two data sets D and D' are incomparable.

The dominance relation $\prec$ satisfies transitivity which is the basis for typical optimizations in skyline computation: if $x \prec y$ and $y \prec w$, then $x \prec w$.

2.2 Probabilistic Skyline Computation

A d-dimensional uncertain data set consists of a set of objects each of which denotes a probability distribution over a d-dimensional data space. Our work assumes that probability distributions of all objects are independent and that every object denotes a discrete distribution. These assumptions allow us to represent the probability distribution of each object with a weighted set of tuples called instances. We write $x \in X$ to mean that x is an instance of object X, and write $Pr(x)$ for the weight of instance x. We assume that $\Sigma_{x \in X} Pr(x) = 1$ holds for every object X.

Since an object denotes a probability distribution, it has a certain probability of being in the skyline set. Consider two objects X and Y. The probability that Y dominates an instance $x \in X$, written $Pr[Y \prec x]$, is calculated as follows:

$$Pr[Y \prec x] = \Sigma_{y \in Y, y \prec x} Pr(y). \tag{1}$$

The probability that Y does not dominate x, written $Pr[Y \not\prec x]$, is then calculated as follows:

$$Pr[Y \not\prec x] = 1 - \Sigma_{y \in Y, y \prec x} Pr(y). \tag{2}$$

Now, consider an uncertain data set U and an object $X \in U$. Since an instance x of object X is in the skyline set if no other object dominates x, the probability that x is in the skyline set, written $SkyPr(x)$, is the product of $Pr[Y \not\prec x]$ for every object $Y \in U$ where $Y \neq X$:

$$
\begin{aligned}
SkyPr(x) &= \Pi_{Y \in U, Y \neq X} Pr[Y \not\prec x] \\
&= \Pi_{Y \in U, Y \neq X}(1 - \Sigma_{y \in Y, y \prec x} Pr(y)).
\end{aligned} \tag{3}
$$

Then, the probability that X is in the skyline set, written $SkyPr(X)$, is the expectation of $SkyPr(x)$ for every instance $x \in X$. We calculate $SkyPr(X)$ as follows:

$$
\begin{aligned}
SkyPr(X) &= \Sigma_{x \in X} Pr(x) SkyPr(x) \\
&= \Sigma_{x \in X} Pr(x) \Pi_{Y \in U, Y \neq X}(1 - \Sigma_{y \in Y, y \prec x} Pr(y)).
\end{aligned} \tag{4}
$$

We refer to $SkyPr(x)$ as the skyline probability of instance x, and $SkyPr(X)$ as the skyline probability of object X.

Fig. 3 shows an example of computing skyline probabilities of instances in an uncertain data set. We assume that all instances of an object are assigned the same weight. Then, we obtain skyline probabilities of objects as follows: $SkyPr(X) = \frac{1}{2} \cdot (1 + \frac{2}{3}) = \frac{5}{6}$, $SkyPr(Y) = \frac{1}{3} \cdot (1 + \frac{1}{2} + 0) = \frac{1}{2}$, and $SkyPr(W) = (0 + 0) \cdot \frac{1}{2} = 0$.

2.3 Reducing the Number of Dominance Tests

Equation (3) for calculating $SkyPr(x)$ indicates that we need to find all instances in each object that dominate instance x. Since performing pairwise dominance tests between all instances is too costly, PSkyline tries to reduce the number of dominance tests by performing dominance tests between an instance and a set of instances or between two sets of instances. To be specific, when a set D of instances is involved in dominance tests, PSkyline exploits the minimum and maximum corners, D_{min} and D_{max}, of its (minimum) bounding box:

For every $x \in D$, we have $D_{min} \preceq x \preceq D_{max}$.

We obtain D_{min} and D_{max} by taking the minimum element and the maximum element of all elements in D for each dimension i:

$$D_{min}[i] = \min_{x \in D} x[i]$$
$$D_{max}[i] = \max_{x \in D} x[i].$$

Suppose that we are to compare an instance x and a set D of instances. The following proposition [16], which is based on transitivity of $\prec$, states four sufficient conditions for dispensing with dominance tests between x and all instances in D.

Proposition 2.1.
> If $D_{max} \prec x$, then $D \prec x$.
> If $x \prec D_{min}$, then $x \prec D$.
> If $x \not\prec D_{max}$, then $x \not\prec y$ for any instance $y \in D$.
> If $D_{min} \not\prec x$, then $y \not\prec x$ for any instance $y \in D$.

PSkyline uses the first and third conditions in its implementation.

Now, suppose that we are to compare two sets D and D' of instances. The following proposition [16], also based on transitivity of $\prec$, states two sufficient conditions for dispensing with dominance tests between instances in D and D'.

Proposition 2.2.
> If $D_{min} \not\prec D'_{max}$ and $D'_{min} \not\prec D_{max}$, then $D \prec\!\!\succ D'$.
> If $D_{max} \prec D'_{min}$, then $D \prec D'$.

We exploit the first condition in Proposition 2.2 to find incomparable groups of instances. The second condition allows us to add weights of all instances in D only once and reuse the result for every instance in D'.

Another important technique to reduce the number of dominance tests is to exploit bounding boxes of individual objects. Let us write X_{min} and X_{max} for the minimum and maximum corners of the (minimum) bounding box for object X:

> For every $x \in X$, we have $X_{min} \preceq x \preceq X_{max}$.

The following proposition [8], also based on transitivity of $\prec$, allows us to immediately determine the skyline probability of an instance or an object as either zero or one; PSkyline exploits all three conditions below to reduce the number of dominance tests.

Proposition 2.3.
> If $X_{max} \prec y$, then $SkyPr(y) = 0$.
> If $X_{max} \prec Y_{min}$, then $SkyPr(Y) = 0$.
> If $X_{min} \not\prec Y_{max}$ for every object $X \neq Y$, then $SkyPr(Y) = 1$.

The first condition holds because $Pr[X \not\prec y]$ is zero:

$$Pr[X \not\prec y] = 1 - \Sigma_{x \in X, x \prec y} Pr(x) = 1 - \Sigma_{x \in X} Pr(x) = 0.$$

For example, we obtain $SkyPr(y_3) = 0$ in Fig. 3 because we have $(3, 2) = X_{max} \prec y_3 = (4, 2)$.

The second condition holds because we have $SkyPr(y) = 0$ for every instance $y \in Y$. It allows us not only to immediately determine the skyline probability of Y, but also to eliminate from further consideration all instances of Y even if some instance of Y dominates other instances belonging to different objects. To see why, assume that $y \prec w$, $y \in Y$, and $w \in W$ hold. By transitivity of $\prec$, we have $X_{max} \prec Y_{min} \preceq y \prec w$, from which we deduce $SkyPr(w) = 0$ by the first condition. Thus, even if we discard all instances of Y, we can still deduce $SkyPr(w) = 0$ when we compare X_{max} and w. For example, we obtain $SkyPr(W) = 0$ in Fig. 3 because we have $(3, 2) = X_{max} \prec W_{min} = (3, 3)$.

The third condition holds because no instance of Y is dominated by instances of different objects. For such an object Y, we do not need to compute skyline probabilities of its individual instances. Unlike the second condition, however, it does not allow us to eliminate from further consideration all instances of Y, since some instance of Y may still dominate other instances belonging to different objects.

2.4 Online Probabilistic Skyline Computation

An uncertain data stream S is essentially a sequence of uncertain data sets continually produced over a time interval. We assume that every instance is given a discrete time stamp, starting at 1 and ending at L, and write $S[t_1, t_2]$ for the uncertain data set consisting of all instances over a window covering time stamps t_1 to t_2. Given a window size W, online probabilistic skyline computation on S returns skyline probabilities of all objects for each of uncertain data sets $S[1, W], S[2, W + 1] \ldots, S[L - W + 1, L]$. (For the sake of simplicity, we ignore $S[1, 1]$, $S[1, 2] \ldots, S[1, W - 1]$.)

Our online probabilistic skyline algorithm O-PSkyline reuses the result on an uncertain data set when analyzing the next uncertain data set in the sequence, rather than analyzing successive uncertain data sets independently. We can continue to use (4) to compute the skyline probability of each object, since whenever old instances expire or new instances arrive, O-PSkyline renormalizes the weights of all instances so that $\Sigma_{x \in X} Pr(x) = 1$ holds for every object X. (Every instance has a constant weight, e.g., the playing time in the NBA database, and we use these constant weights to compute the weight $Pr(x)$ of instance x.)

2.5 Top-k Probabilistic Skyline Computation

Given an uncertain data set, top-k probabilistic skyline computation returns k objects with the highest skyline probabilities; for an uncertain data stream, it reports such k objects at each time stamp. To efficiently find such top-k objects, our top-k probabilistic skyline algorithm K-PSkyline exploits an upper bound for the skyline probability of each instance x, written $SkyPr^+(x)$.

Consider an instance x of object X in an uncertain data set U. We choose a set D containing x (and potentially also instances of other objects), e.g., the set of all instances in the node containing x. With respect to D, K-PSkyline computes $SkyPr^+(x)$ for an instance x as follows:

$$\begin{aligned} SkyPr^+(x) &= \Pi_{Y \in U, Y \neq X} Pr[Y \not\prec D_{min}] \\ &\geq \Pi_{Y \in U, Y \neq X} Pr[Y \not\prec x] \quad (5) \\ &= SkyPr(x). \end{aligned}$$

Here, the inequality follows from $D_{min} \preceq x$. Since D_{min} does not belong to any object, we calculate $SkyPr(D_{min})$ as follows:

$$SkyPr(D_{min}) = \Pi_{Y \in U} Pr[Y \not\prec D_{min}]. \quad (6)$$

By rewriting (5) using (6), we obtain $SkyPr^+(x)$ for each instance $x \in D$ with a single division:

$$SkyPr^+(x) = \frac{SkyPr(D_{min})}{Pr[X \not\prec D_{min}]} = \frac{SkyPr(D_{min})}{1 - \Sigma_{w \in X, w \prec D_{min}} Pr(w)}. \quad (7)$$

Note that $Pr[X \not\prec D_{min}]$ is not always equal to one because X may have an instance dominating D_{min}. It can never be

zero, however, because D contains at least one instance of object X, namely x. Then, K-PSkyline calculates an upper bound for the skyline probability of an object X, written $SkyPr^+(X)$, as the expectation of $SkyPr^+(x)$ for every instance $x \in X$:

$$SkyPr^+(X) = \Sigma_{x \in X} Pr(x) SkyPr^+(x). \qquad (8)$$

Using upper bounds for skyline probabilities of instances and objects, K-PSkyline considers only a small number of objects and efficiently computes top-k objects.

3 Z-TREES

PSkyline is a general framework for probabilistic skyline computation which accommodates conventional tree structures such as R-trees [15] and ZBtrees [16]. This section presents a new in-memory tree structure called Z-tree which is designed specifically for PSkyline. Its design is based on Z-addresses [17].

3.1 Z-Addresses

Consider a d-dimensional data space $[0.0, 1.0]^d$. We can divide it into 2^{vd} identical bounding boxes, each with a volume of $\frac{1}{2^{vd}}$, by dividing the unit interval in each dimension into 2^v intervals of equal length. A Z-address is a vd-bit binary number that uniquely identifies one of these bounding boxes.

As an example, consider a tuple x in $[0.0, 1.0]^d$. We use the following sequence of integers as the coordinates of the bounding box to which x belongs:

$$(\lfloor 2^v \cdot x[1] \rfloor, \lfloor 2^v \cdot x[2] \rfloor, \ldots, \lfloor 2^v \cdot x[d] \rfloor).$$

Note that every coordinate is in the range of $[0, 2^v - 1]$ and thus can be represented as a v-bit binary number. Let $\lfloor 2^v \cdot x[i] \rfloor$ be represented as a v-bit binary number $b_{i1}b_{i2} \ldots b_{iv}$. Then, we represent the coordinates of x as a single vd-bit binary number

$$b_{11}b_{12} \ldots b_{1v}\ b_{21}b_{22} \ldots b_{2v}\ \ldots\ b_{d1}b_{d2} \ldots b_{dv}.$$

By interleaving bits in this binary number, we obtain the Z-address of x as another vd-bit binary number

$$b_{11}b_{21} \ldots b_{d1}\ b_{12}b_{22} \ldots b_{d2}\ \ldots\ b_{1v}b_{2v} \ldots b_{dv}.$$

Note that a prefix of the Z-address whose length is a multiple of d is another Z-address: a prefix of length kd (where $k \leq v$) is the Z-address of x that is obtained when the unit interval in each dimension is divided into 2^k intervals. Fig. 4a shows coordinates (in plain font) and Z-addresses (in boldface) associated with $2^{2 \cdot 2}$ bounding boxes in the 2D data space $[0.0, 1.0]^2$ (with $v = 2$ and $d = 2$); Fig. 4b shows Z-addresses associated with $2^{1 \cdot 2}$ bounding boxes in the same data space (with $v = 1$ and $d = 2$).

3.2 Building Z-Trees

The design of Z-trees is based on the following three properties of Z-addresses:

1. Every prefix of a Z-address corresponds to a bounding box containing multiple smaller bounding boxes.
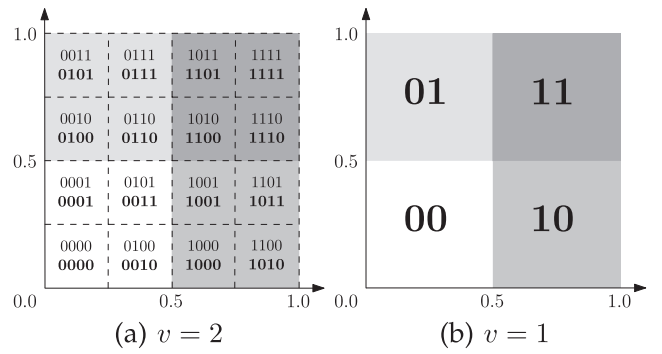

Fig. 4. Z-addresses in $[0.0, 1.0]^2$.

2. If $x \prec y$, then Z-address of $x \leq$ Z-address of y.
3. Every consecutive r bits in a Z-address ($r \leq d$) originate from r different coordinates.

The property 1 states that for a vd-bit Z-address, its prefix of length w corresponds to a bounding box containing 2^{vd-w} smaller bounding boxes (each with a volume of $\frac{1}{2^{vd}}$). It enables us to build a tree of clusters for quickly identifying sparse or dense regions in a data set. Consider a data set D from $[0.0, 1.0]^d$. After computing vd-bit Z-addresses of all tuples, we recursively build a tree of clusters by consuming r bits of Z-addresses at a time (where r is assumed to be a divisor of vd):

1. We begin with a single cluster containing all tuples in D as the root node. Its bounding box is given as $[0.0, 1.0]^d$.
2. At each step, we identify those nodes that contain more than C tuples. For such a node, we use (and discard) the next unused r bits of the Z-address of each tuple to redistribute all tuples among 2^r smaller bounding boxes of the same shape. If a bounding box receives no tuples, we discard it. Then, these new clusters become child nodes, which are sorted according to r bits of Z-addresses used in redistributing tuples. If Z-addresses of tuples in such a node have been exhausted, we recalculate their Z-addresses from their normalized coordinates with respect to the bounding box of the node. Then, we redistribute all tuples in the same way. As a minor optimization, if all tuples in such a node are identical, we do not create its child nodes. Instead, we mark it as a point-mass leaf node containing identical tuples.
3. We repeat this step until no more such nodes remain.

We refer to such a 2^r-way tree of clusters as a Z-tree. Note that bounding boxes of two nodes do not intersect unless one is an ancestor node of the other.

The property 2, which is due to Lee et al. [16] in the context of skyline computation, implies that in a data set sorted according to Z-addresses, a dominating tuple always appears before all those tuples that it dominates. Hence, if x appears before y in a data set sorted according to Z-addresses, y cannot dominate x while x may or may not dominate y. We can easily verify that this property holds in Figs. 4a and 4b. Since every internal node in a Z-tree stores its child nodes in the order of increasing Z-addresses of their tuples, no tuples in a child node can dominate tuples in its left sibling nodes.

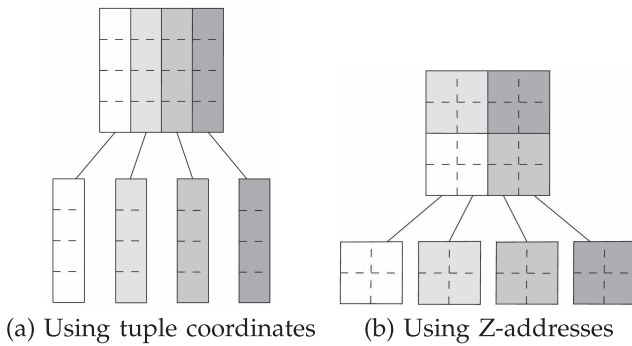(a) Using tuple coordinates   (b) Using Z-addresses

Fig. 5. Structures of trees of tuples.

For our purpose, the most important is the property 3, which is the result of uniformly interleaving bits from tuple coordinates. It prevents Z-trees from generating bounding boxes excessively elongated along a particular dimension. Specifically, every side of a bounding box is twice longer than or as long as its shortest side, thereby resulting in a high chance of placing incomparable groups of tuples among sibling nodes. For example, we find no pairs of incomparable groups in a tree built from tuple coordinates in Fig. 5a, whereas a pair of incomparable groups exist in a Z-tree built from Z-addresses in Fig. 5b.

To our knowledge, no previous skyline algorithm exploits this property of Z-addresses to find incomparable groups of tuples. It is also mainly because of this property that Z-trees lend much better to PSkyline or its variant than popular tree structures such as R-trees [15] and ZBtrees [16], as its overall effectiveness heavily depends on the number of times it finds incomparable groups of instances.

4 RELATED WORK

The problem of skyline computation on certain data sets has been known as the maximal vector problem in the computational geometry community. For generic skyline computation as a database operation [18], typical skyline algorithms strive to reduce the number of dominance tests by using special index structures (as in BBS [19] and ZSearch [16]) or presorting data sets according to a monotone preference function (as in SFS [20] and LESS [21]). More recently, Zhang et al. [14] show that reducing the number of dominance tests between incomparable tuples is critical to the overall speed of skyline computation on certain data sets. Our probabilistic skyline algorithms further develop the same idea by finding incomparable *groups* of instances in uncertain data sets or streams.

Pei et al. [8] are the first to extend skyline computation on certain data sets to probabilistic skyline computation on uncertain data sets. They propose two algorithms, bottom-up and top-down, for p-skyline computation which identifies those objects whose skyline probabilities are higher than a given threshold p. Böhm et al. [22] extend p-skyline computation for uncertain data sets denoting continuous distributions. They propose two algorithms, priority and indexed, which approximate the probability distribution of each object with a constant number of samples.

Note that setting p to zero in p-skyline computation is *not* equivalent to (and indeed much weaker than) computing skyline probabilities of all objects, since it reports an object as soon as its skyline probability becomes known to be higher than p, without actually computing its skyline probability. For example, if all objects have positive skyline probabilities, p-skyline computation with p set to zero terminates almost instantly.

Atallah and Qi [11] present the first work that addresses the problem of computing exact skyline probabilities of all objects. Their main idea is to combine two basic methods of computing skyline probabilities (grid method and weighted dominance counting method) to achieve a probabilistic skyline algorithm that compensates for the weakness of each basic method. Although the worst case time complexity of their algorithm is subquadratic in the size of the data set, its utility is limited to 2D data sets because of a hidden factor exponential in the dimensionality of the data set. (The authors state that for high-dimensional data sets, their algorithm is not a practical solution and a straightforward nested-loop algorithm is better.) In contrast, our algorithm PSkyline lends itself to high-dimensional data sets as well as low-dimensional data sets, especially when its tree structures are Z-trees, whose overall structure is largely independent of the dimensionality of the data set. In fact, the speed of PSkyline with Z-trees is almost irrelevant to the dimensionality of data sets, as we will see in Section 8.

Although the idea of Z-trees is inspired by ZBtrees which are used in the conventional skyline algorithm ZSearch by Lee et al. [16], there are two key differences. The first is that Z-trees exploit the third property of Z-addresses to efficiently find incomparable groups of instances (as well as the second property to prune search space), whereas ZBtrees use only the second property of Z-addresses to visit potential skyline tuples in Z-order and maintain skyline tuples in Z-order. (Lee et al. briefly mention that dominance tests between incomparable groups of tuples can be eliminated, but they do not investigate how to find incomparable groups of tuples.) The second is that because of the first property of Z-addresses, no bounding boxes of sibling nodes intersect with each other in a Z-tree, whereas the same property does not hold in ZBtrees (and also in LSB-trees [23]), in which the bounding box of a node may even completely encompass the bounding box of its sibling node. As a result, PSkyline is more likely to find incomparable groups in Z-trees than in ZBtrees and runs particularly fast in conjunction with Z-trees.

Online skyline computation on certain data streams has been studied by Tao and Papadias [24]. To our knowledge, O-PSkyline is the first online probabilistic skyline algorithm computing *exact* skyline probabilities on uncertain data streams. The work by Zhang et al. [10] considers online p-skyline computation on a special form of uncertain data streams in which every object consists only of a single instance, and thus deals with an inherently different problem. Su et al. [12] also extend p-skyline computation for uncertain data streams and propose its top-k variant. They, however, identify only those objects whose skyline probabilities are higher than p or top-k objects without computing their exact skyline probabilities.

5 PSKYLINE

This section presents our probabilistic skyline algorithm PSkyline. The main novelty of PSkyline lies in the use of two recursive functions for traversing its tree structure and finding incomparable groups of instances. We begin by describing the representation of the tree structure and its invariants. As PSkyline is a fairly complex algorithm (see Figs. 7, 8, and 9), we first give its overview. Then, we explain the details of PSkyline.

5.1 Tree Structures for PSkyline

PSkyline uses the following type definitions for its tree structure:

```
type tuple  =  float[d]
type point  =  instance{ tuple t,
                         float weight,
                         int object,
                         float prob }
             | corner{ tuple t,
                       int object }
type box    =  { tuple min, max }
type tree   =  node{ tree child[],
                     box bound,
                     bool incomparable }
             | leaf{ point data[],
                     box bound,
                     bool incomparable}
```

A tuple of type `tuple` is represented as an array of d floating-point numbers where d is the dimensionality of the uncertain data set. A point of type `point` is either an instance of an object or the maximum corner of the bounding box of an object. An instance of type `instance` stores a tuple `t`, its weight `weight`, and the identifier `object` of the object to which it belongs. Its skyline probability `prob` is uninitialized when a tree structure is built, but is later calculated by PSkyline. A maximum corner of type `corner` stores a tuple `t` and the identifier `object` of the object to which it belongs.

A bounding box of type `box` represents the set of all tuples lying inside a hyperrectangle specified by its two members `min` and `max`. Hence, for example, we may write $x \in B$, $x \prec B$, $B \prec\!\succ B'$ and so on, where B and B' are bounding boxes.

A tree of type `tree` is either an internal node of type `node` or a leaf node of type `leaf`. (We use the terms *tree* and *node* interchangeably.) An internal node has three members: `child`, an array of child nodes; `bound`, a bounding box; `incomparable`, a Boolean flag indicating whether the node can be ignored or not. A leaf node has another member `data` for an array of points, but without the member `child`. `incomparable` is initialized to **false** (Boolean false) in every node.

Given an uncertain data set, PSkyline builds a tree structure of type `tree` and calculates the skyline probability of every instance. We assume that the tree structure satisfies the following two invariants:

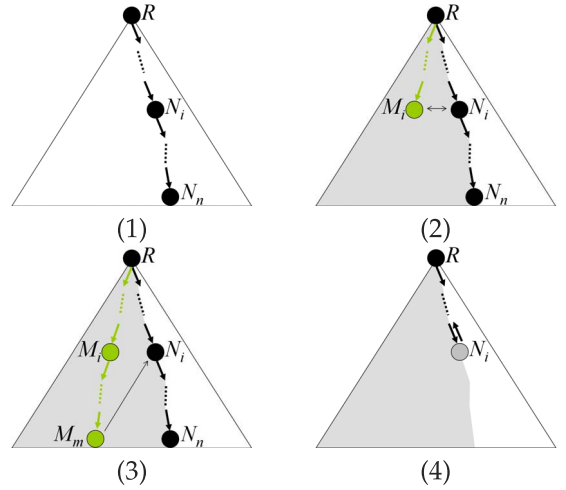- For every internal or leaf node, all points in it lie inside its bounding box.

Fig. 6. $find()$ and $check()$ in PSkyline.

- For every internal node, its child nodes are sorted in such a way that no points in a child node can dominate points in its left sibling nodes.

Both ZBtrees and Z-trees are examples of such tree structures.

From now on, we say that two nodes are incomparable if their bounding boxes are incomparable. We also say that a tuple x dominates a node M if x dominates all tuples in M. By the first invariant, x dominates M if $x \prec M$.bound.min.

5.2 Overview of PSkyline

Given a tree structure R of type `tree`, PSkyline traverses it in depth-first order by invoking a function $find()$ (to "find" the next leaf node). Once $find()$ reaches a leaf node N, PSkyline traverses R again in depth-first order by invoking another function $check()$ (to "check" dominance relations with other leaf nodes). When $check()$ reaches a leaf node M (lying to the left of N), PSkyline compares M and N to update skyline probabilities of instances in N. When the invocation of $check()$ eventually returns to N, PSkyline computes final skyline probabilities of all instances in N. Thus, PSkyline is essentially a nested-loop skyline algorithm where the outer loop locates a leaf node N and the inner loop compares N with other leaf nodes.

Below, we explain how PSkyline reduces the number of dominance tests while traversing R. We use the following scenario in which $find()$ first reaches a leaf node N_n and $check()$ then reaches a leaf node M_m.

Suppose that PSkyline has reached a leaf node N_n, at depth n, after a series of invocations of $find()$ in depth-first order; see 1 in Fig. 6. Upon reaching N_n, PSkyline initializes for each instance x in N_n an array of floating-point numbers that records the sum of weights of those instances in each object that dominate x. (Hence, the array has the same length as the total number of objects.)

Now, PSkyline invokes $check()$ in depth-first order to locate all leaf nodes in R that may contain instances that dominate those instances in N_n. Because of the second invariant on R, it needs to visit only the part of R that lies to the left of the chain of nodes from the root node to N_n. Suppose that PSkyline has reached a node M_i, at depth i; see 2 in Fig. 6

```
Algorithm PSkyline (U)
Input: uncertain data set U
Output: SkyPr(X) for each X ∈ U
Global variables:   R : tree
                    path : tree[]
                    checked : bool[]
                    wsum_x : float[]
1:    compute X_min and X_max for each X ∈ U
2:    U' ← U
3:    for each Y ∈ U do
4:        if ∃X ∈ U such that X_max ≺ Y_min then
5:            SkyPr(Y) ← 0
6:            U' ← U' − {Y}
7:        else if ∄X ∈ U such that X_min ≺ Y_max then
8:            SkyPr(Y) ← 1
9:        end if
10:   end for
11:   D ← (⋃_{X∈U'} X) ∪ {X_max | X ∈ U'}
12:   R ← tree structure built from D
13:   path[0] ← R
14:   find(R, 0)
15:   for each X ∈ U such that SkyPr(X) is unknown do
16:       SkyPr(X) ← Σ_{x∈X} Pr(x)SkyPr(x)
17:   end for
```

Fig. 7. Algorithm PSkyline.

- If $i \le n$ holds and M_i has a corresponding node N_i at the same depth, PSkyline tests if the bounding boxes of M_i and N_i are incomparable. If incomparable, the first invariant on R implies that every pair of leaf nodes from M_i and N_i are also incomparable. In this case, we set the member `incomparable` of M_i to **true** to indicate that it does not need to be compared with N_i again, and terminate the invocation of $check()$ on M_i immediately. Otherwise, we set the member `incomparable` to **false** to indicate that M_i needs to be compared with N_i.

- If $i > n$ holds and M_i has no corresponding node at the same depth, PSkyline compares M_i with N_n instead.

In this way, PSkyline attempts to reduce the number of dominance tests between incomparable groups of instances.

When PSkyline is ready to inspect a leaf node M_m at depth m (where m is not necessarily equal to n), it tests if M_m contains maximum corners for objects. If M_m contains a maximum corner X_{max}, PSkyline traces the path from N_n to the root node and identifies those nodes dominated by X_{max}; see 3 in Fig. 6. If there exists a node N_i at depth i (where $i \le n$) such that X_{max} dominates N_i but not its parent node, the first invariant on R implies that skyline probabilities of all instances in N_i are zero by the first condition in Proposition 2.3. In this case, we terminate the invocation of $find()$ on N_i after canceling all subsequent invocations of $check()$ and $find()$; see 4 in Fig. 6. Otherwise, we perform pairwise dominance tests between instances in M_m and N_n, exploiting Proposition 2.1 to avoid unnecessary dominance tests, to update arrays allocated for instances in N_n. In this way, PSkyline attempts to reduce the number of dominance tests between dominating and dominated instances.

5.3   Probabilistic Skyline Algorithm PSkyline

Fig. 7 shows the pseudocode of PSkyline. It computes the skyline probability of every object in a given uncertain data set U. It uses four global variables (R, $path$, $checked$, $wsum_x$)

```
find(node N, int depth) =    // internal nodes
1:    checked[depth] ← false    // first visit to N
2:    for each N' ∈ N.child[] do    // from left to right
3:        path[depth + 1] ← N'
4:        result ← find(N', depth + 1)
5:        if result = Kill(0) then
6:            SkyPr(x) ← 0 for each instance x ∈ N'.data[]
7:        else if result = Kill(l) then
8:            return Kill(l − 1)
9:    end for
10:   return Okay

find(leaf N, int depth) =    // leaf nodes
11:   checked[depth] ← false    // first visit to N
12:   initialize wsum_x[] for each instance x ∈ N.data[]
13:   result ← check(R, 0, depth)
14:   if result = Kill(l) then
15:       return result
16:   for i from depth − 1 down to 1 do
17:       if checked[i] = false then
18:           checked[i] ← true    // finished invoking check()
                             // on the leftmost leaf node of path[i]
19:       else
20:           break
21:   end for
22:   for each instance x ∈ N.data[] do
23:       dominance tests with instances in N.data[]
24:       update wsum_x[]
25:       calculate SkyPr(x) from wsum_x[]    // Equation (3)
26:   end for
27:   return Okay
```

Fig. 8. Function $find()$ in PSkyline.

and two recursive functions ($find()$ and $check()$). First PSkyline identifies those objects that can be assigned a skyline probability zero or one according to Proposition 2.3 (lines 1-10). Then, it sets R to the tree structure (or equivalently its root node) built from instances of objects and maximum corners for objects (lines 11-12). Finally, it invokes $find()$ and computes skyline probabilities of all remaining objects from skyline probabilities of their instances (lines 13-17).

PSkyline uses $path[i]$ to record the node that $find()$ visits at depth i, and $check[i]$ to indicate whether or not $path[i]$ has been compared with all its left siblings for the incomparability relation. $wsum_x$, an array holding the sum of weights of those instances in each object that dominate x, is initialized when $find()$ visits a leaf node containing x.

Fig. 8 shows the pseudocode of $find()$. An invocation of $find(N, depth)$ satisfies the following invariants:

- N is at depth $depth$ in R.
- $path[depth]$ is set to N.

After computing skyline probabilities of all instances in N, $find()$ returns either Okay or Kill(l) where l is a nonnegative integer:

- Okay means that $find(N, depth)$ has successfully computed skyline probabilities of all instances in N.
- Kill(l) means that the lth ancestor node of N is dominated by the maximum corner for an object and all instances in it have a skyline probability zero. For example, if $find(N, depth)$ returns Kill(1), all instances in the parent node of N have a skyline probability zero. As a special case, if $find(N, depth)$ returns Kill(0), PSkyline assigns a skyline probability zero to all instances in N itself.

```
check(node M, int depth, int ref) =    // internal nodes
 1:    if depth + 1 > ref then
 2:       ref' ← ref
 3:    else
 4:       ref' ← depth + 1
 5:    for each M' ∈ M.child[] do    // from left to right
 6:       if checked[ref'] = false then
                // check() visits M' for the first time
                // after find() visits path[ref'].
 7:          if M' = path[ref'] then
 8:             M'.incomparable ← false
 9:          else if M'.bound ⊀≻ path[ref'].bound then
10:             M'.incomparable ← true
11:          else
12:             M'.incomparable ← false
13:          end if
14:          if M'.incomparable = false then
15:             result ← check(M', depth + 1, ref)
16:             if result ≠ Okay then
17:                return result
18:          end if
19:    end for
20:    return Okay
```

Fig. 9. Function $check()$ in PSkyline.

```
check(leaf M, int depth, int ref) =    // leaf nodes
21:    if M = path[ref] then
22:       return TerminateCheck
23:    if M.bound ⊀≻ path[ref].bound then
24:       return Okay
25:    if X_max ∈ M.data[] for any object X then
                // find the ancestor node of path[ref] that is
                // dominated by maximum corners for objects
26:       dead ← 0
27:       for i from ref down to 1 do
28:          if checked[i] = true then
29:             break
30:          if ∃X_max ∈ M.data[] such that
                   X_max ≺ path[i].bound.min then
31:             dead ← i
32:          else
33:             break
34:       end for
35:       if dead ≥ 1 then
36:          return Kill(ref − dead)
37:    end if
38:    for each instance x ∈ path[ref].data[] do
39:       dominance tests with instances in M.data[]
40:       update wsum_x[]    // Propositions 2.1, 2.2, and 2.3
41:    end for
42:    return Okay
```

Fig. 9 shows the pseudocode of $check()$. An invocation of $check(M, depth, ref)$ satisfies the following invariants:

- M is at depth $depth$ in R.
- $path[ref]$ is set to the leaf node that has been visited by the last active invocation of $find()$.

After comparing $path[ref]$ with leaf nodes in M, $check()$ returns either Okay, TerminateCheck, or Kill(l) where l is a nonnegative integer

- Okay means that $check(M, depth, ref)$ has successfully compared instances in $path[ref]$ and M.
- TerminateCheck has the same meaning as Okay, but it also means that a subsequent invocation of $check()$ has reached $path[ref]$ and all leaf nodes lying to the left of $path[ref]$ have been visited by $check()$. Hence, all previous invocations of $check()$ should be terminated immediately.
- Kill(l) means that the lth ancestor node of $path[ref]$ is dominated by the maximum corner for an object.

When PSkyline completes the invocation of $find()$ on a leaf node N, it computes the correct skyline probability of every instance x in N (lines 22-26 in Fig. 8). This is because of the second invariant on R: no leaf node lying to the right of N contains instances dominating x. Hence, it is easy to obtain a progressive version of PSkyline that determines the skyline probability of every object when $find()$ visits the leaf node containing its last instance. Our experiments in Section 8 use the progressive version of PSkyline.

Our implementation of PSkyline maintains a fixed number of arrays for holding intermediate results of dominance tests, rather than allocating an array for every instance. Note that we no longer need the array $wsum_x$ allocated for instance x after calculating $SkyPr(x)$ in line 25 in Fig. 8. Thus, if leaf nodes contain up to C instances, our implementation of PSkyline allocates only C arrays to hold intermediate results of dominance tests.

5.4 Analysis of PSkyline

The strength of PSkyline, especially on data sets with anticorrelated distributions or high-dimensional data sets, is primarily due to the use of two traversal functions to dispense with dominance tests between incomparable groups of instances. As an extreme case, let us consider an uncertain data set whose n instances are all incomparable with each other. We assume that PSkyline is running on a full k-way tree with height h and k^h leaf nodes such that every leaf node contains the same number $\frac{n}{k^h}$ of instances. Since all instances are incomparable with each other, $find()$ eventually visits every leaf node and performs $\frac{\frac{n}{k^h}(\frac{n}{k^h}-1)}{2}$ pairwise dominance tests between its instances (in line 23 in Fig. 8). Since all sibling nodes are also incomparable with each other, $check()$ tests incomparability only between sibling nodes with the same parent node (in lines 9 and 23 in Fig. 9) and *never performs dominance tests between instances belonging to different leaf nodes* (in line 39 in Fig. 9). Therefore, $check()$ only performs at most $\frac{k(k-1)}{2}$ incomparability tests for each internal or leaf node (where an incomparability test involves two dominance tests). Thus, while traversing the tree, PSkyline performs at most a total of

$$k^h \cdot \frac{\frac{n}{k^h}(\frac{n}{k^h}-1)}{2} + (k^h - 1) \cdot 2 \cdot \frac{k(k-1)}{2}$$
$$= \frac{n(\frac{n}{k^h}-1)}{2} + k^{h+2} - k^{h+1} - k^2 + k$$
$$\approx \frac{n^2}{2k^h} - \frac{n}{2} + k^{h+2}$$

dominance tests between instances. This result indicates that in comparison with a straightforward nested-loop algorithm performing $\frac{n(n-1)}{2}$ dominance tests, the speedup of PSkyline is approximately proportional to the number of leaf nodes in

```
Algorithm O-PSkyline (S)
Input: uncertain data stream S over timestamps 1, · · · , L
Output: SkyPr(X) for each X ∈ S[t, t + W − 1]
        for t = 1, · · · , L − W + 1
Global variables:   R : tree
                    path : tree[]
                    checked : bool[]
1:     R ← tree structure built from S[1, W − 1]
2:     for t from W to L do
3:        for each instance x with timestamp t do
4:           insert x to R
5:        end for
6:        path[0] ← R
7:        find(R, 0)
8:        for each instance x with timestamp t − W do
9:           delete x from R
10:       end for
11:       for each X ∈ S[t − W + 1, t] do
12:          SkyPr(X) ← Σ_{x∈X} Pr(x)SkyPr(x)
13:       end for
14:    end for
```

Fig. 10. Algorithm O-PSkyline.

the tree. (Here, we ignore the cost for building a Z-tree, which is $O(nh)$ and usually much smaller than the cost for traversing the Z-tree.) For comparison, the complexity of the algorithm by Atallah and Qi [11] is approximately $O(n^{2-\frac{1}{d+1}})$ where d is the dimensionality of the data set.

6   ONLINE PSKYLINE

This section presents our online probabilistic skyline algorithm O-PSkyline. It reports skyline probabilities of all objects in each uncertain data set that a given uncertain data stream generates sequentially over a time interval.

O-PSkyline maintains a single uncertain data set over a time window of a specific size W. When the window moves at time t, it updates the uncertain data set by inserting new instances with time stamp t and deleting old instances with time stamp $t − W$. O-PSkyline then incorporates changes of skyline probabilities due to these new and old instances to update the skyline probability of each remaining instance. In this way, O-PSkyline performs dominance tests between any pair of instances at most twice.

Similarly to PSkyline, O-PSkyline stores all instances in a tree structure and traverses it using the two recursive functions $find()$ and $check()$. Fig. 10 shows the pseudo-code of O-PSkyline. At time t, O-PSkyline first inserts new instances with time stamp t to the tree structure R (lines 3-5). After traversing R by invoking $find()$ (line 7), it deletes old instances with time stamp $t − W$ (lines 8-10). Then, it recomputes the skyline probability of every object (lines 11-13). Below, we explain two major differences between PSkyline and O-PSkyline.

First, unlike PSkyline which builds a tree structure only by inserting new instances, O-PSkyline deletes existing instances that expire when the time window moves. O-PSkyline, however, never deletes existing nodes in the tree structure. Suppose that a node N has multiple child nodes. Even after the total number of instances in N becomes smaller than a certain threshold, we do not merge these child nodes because new instances are likely to be inserted to N in the future anyway. Thus, the tree structure R in Fig. 10 never shrinks in the number of nodes in it.

Second, O-PSkyline maintains an array holding inter-mediate results of dominance tests (similar to $wsum_x$ in

```
estimate(node N) =
1:     if N is an internal node then
2:        for each N' ∈ N.child[] do
3:           estimate(N')
4:        end for
5:     else
6:        update wsum_{N_{min}}[]
7:        calculate SkyPr(N_{min}) from wsum_{N_{min}}[]
8:        for each instance x ∈ N.data[] do
9:           SkyPr⁺(x) ← SkyPr(N_{min}) / Pr[X ⊀ N_{min}]   // Equation (7)
10:          SkyPr⁺(X) ← SkyPr⁺(X) + Pr(x)SkyPr⁺(x)
11:       end for
12:    end if
```

Fig. 11. Function $estimate()$ in K-PSkyline.

Fig. 7) for *every* instance. If a leaf node contains at most C instances, PSkyline needs only C such arrays because it finishes calculating $SkyPr(x)$ while $find()$ visits the leaf node containing x. In contrast, O-PSkyline keeps updating $SkyPr(x)$ until x expires from the tree structure, which means that it cannot reuse the same array for multiple instances. In our implementation of O-PSkyline, type `instance` has a new member `wsum` of type **float**[] for such an array.

O-PSkyline performs dominance tests when $find()$ and $check()$ reach leaf nodes. Suppose that at time t, $find()$ reaches a leaf node N. Then, O-PSkyline performs dom-inance tests between old and new instances in N (with time stamps $t − W$ and t, respectively) and existing instances in N. When $check()$ reaches another leaf node M (which lies to the left of N), O-PSkyline performs two kinds of dominance tests: between old and new instances in M and existing instances in N; between existing and new instances in M and new instances in N. In every case above, O-PSkyline updates arrays `wsum` for instances in N only: it computes initial skyline probabilities of new instances in N while adjusting skyline probabilities of existing instances in N.

7   TOP-k PSKYLINE

Our top-k probabilistic skyline algorithm K-PSkyline be-haves in the same way as PSkyline or O-PSkyline, except that it reports only k objects with the highest skyline probabilities. This section explains how K-PSkyline finds such top-k objects from an uncertain data set. Extending O-PSkyline for top-k probabilistic skyline computation is also analogous.

The pseudocode of K-PSkyline is similar to Fig. 7. After building a tree structure R from an uncertain data set U, K-PSkyline computes an upper bound for the skyline probability of each object by calling a function $estimate()$. Then, it finds top-k objects by calling another function $select()$ which exploits these upper bounds. Below, we explain the details of $estimate()$ and $select()$.

Fig. 11 shows the pseudocode of $estimate()$. It traverses the tree structure R to compute an upper bound for the skyline probability of each object using minimum corners of leaf nodes. Upon reaching a leaf node N, it updates $wsum_{N_{min}}[]$ by traversing the left side of the chain of nodes from the root node to N and calculates $SkyPr(N_{min})$. Then, it computes $SkyPr⁺(x)$ of each instance x in N by (7) and updates $SkyPr⁺(X)$ of object X to which x belongs.

Fig. 12 shows the pseudocode of $select()$. It maintains the following invariants at the beginning of the loop (in line 2).

```
select() =
1:   S_init ← U, S_done ← ∅, β ← 0
2:   while true do
3:       S ← up to k objects X ∈ S_init with SkyPr⁺(X) ≥ β
4:       if S is empty then
5:           break
6:       calculate SkyPr(X) for objects X ∈ S by calling find()
7:       S_init ← S_init − S, S_done ← S_done ∪ S
8:       β ← k-th highest SkyPr(X) of objects X ∈ S_done
9:   end while
10:  return top-k objects from S_done
```

Fig. 12. Function $select()$ in K-PSkyline.

- A set S_{init} stores those objects whose skyline probabilities have not been computed yet.
- A set S_{done} stores those objects whose skyline probabilities have been computed.
- β is the kth highest skyline probability $SkyPr(X)$ of objects $X \in S_{done}$.

$select()$ first chooses a set S of candidate objects from S_{init} (line 3). Then, it computes exact skyline probabilities of these candidate objects by calling $find()$ on the tree structure R (line 6). When $find()$ reaches a leaf node N, it calls $check()$ to update arrays wsum for those instances of objects in S that appear in N. $select()$ updates S_{init}, S_{done}, and β at the end of each iteration (lines 7-8) and repeats this procedure until no candidate object remains (lines 4-5).

8 EXPERIMENTS

This section presents experimental results of running our algorithms PSkyline, O-PSkyline, and K-PSkyline. We combine PSkyline with R-trees, ZBtrees, and Z-trees. For PSkyline with R-trees, we do not implement lines 21-22 in Fig. 9 because R-trees do not meet the second invariant in Section 5.1. We combine O-PSkyline with ZBtrees and Z-trees, and K-PSkyline only with Z-trees. For K-PSkyline, we use 5 and 10 for parameter k (top-5 and top-10).

In order to measure the relative performance of our algorithms, we use three algorithms: EX, AQ, and O-EX. EX (*EX*haustive) is an exhaustive algorithm for computing skyline probabilities. Before performing pairwise dominance tests between instances, EX uses the second condition in Proposition 2.3 to eliminate objects with skyline probability zero. (This optimization alone makes EX run much faster than a naive algorithm on typical data sets.) AQ (*A*tallah and *Q*i) is an improvement of the probabilistic skyline algorithm in [11] which additionally implements the same optimization as in EX. O-EX (*O*nline *EX*haustive) is an online exhaustive algorithm for computing skyline probabilities on uncertain data streams. As in O-PSkyline, it incrementally updates skyline probabilities by considering only old instances that expire and new instances that arrive, thereby performing dominance tests between every pair of instances (in the same time window) exactly twice.

We do not compare PSkyline with the two algorithms (bottom-up and top-down) by Pei et al. [8], which are designed only for p-skyline computation and thus incompatible with PSkyline.

Our implementations are all written in C. We run all experiments on Ubuntu Linux with Intel Xeon X3363

Independent   Anti-correlated

(1) $d = 2$

Independent   Anti-correlated

(2) $d = 8$
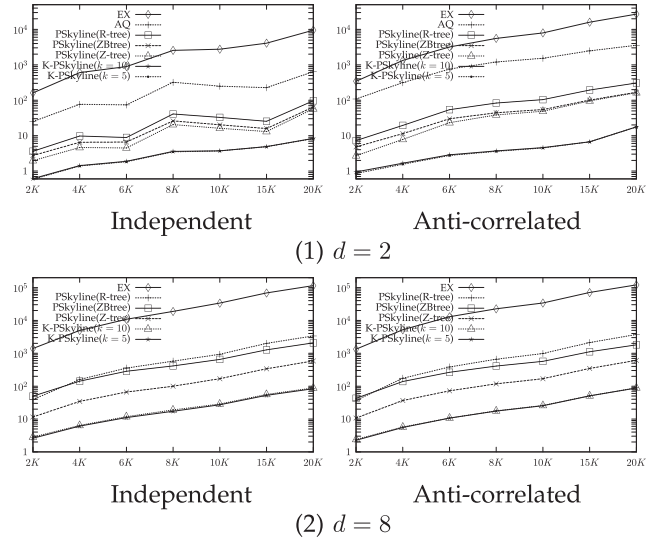
Fig. 13. Elapsed time T (sec) over n.

2.83 GHz and 4 Gigabytes of main memory. As the main performance metric, we use the elapsed time T measured in wall clock seconds. We do not measure the I/O cost because all the algorithms read the entire data only once. All measurements are averaged over five sample runs.

In all experiments, an uncertain data set or stream determines three parameters: dimensionality d, number of objects n, and number of instances s. When reporting the number of objects, we use K for 1,000. For building Z-trees, we allocate 32-bit words for Z-addresses ($v = \lfloor 32/d \rfloor$). We use the following parameters which are experimentally determined: $C = 128$ (maximum number of instances in a node) and $r = 4$ (number of bits consumed at each time).

8.1 Results of Probabilistic Skyline Computation

The experiments use both synthetic data sets and real data sets. For synthetic data sets, we assume that all instances of an object are assigned the same weight (as in [8], [11]). We use the same procedure and parameters described in [8] to create synthetic data sets. First, we generate centers of objects in the data space $[0.0, 1.0]^d$ according to independent distributions or anticorrelated distributions [18]. After generating the center of each object, we create a hyperrectangle whose edge length follows a normal distribution with expectation 0.1 and standard deviation 0.025. Then, we generate instances of the object within the hyperrectangle according to a uniform distribution. The number of instances follows a uniform distribution over interval $[1, 400]$. Hence, a synthetic data set contains $200n$ instances (i.e., $s = 200n$) on average. For real data sets, we use the NBA database ($n = 1,092$ and $s = 260,559$) described in Section 1.

We use two groups of synthetic data sets based on independent and anticorrelated distributions:

- For the first group (Fig. 13), we vary n from $2K$ to $20K$ with

    - $d = 2$ for low-dimensional data sets and
    - $d = 8$ for high-dimensional data sets.

- For the second group (Fig. 14), we vary d from 2 to 10 with $n = 10K$.
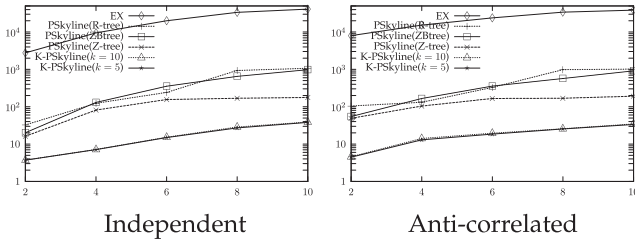
Fig. 14. Elapsed time T (sec) over d ($n = 10K$).

Fig. 15. Elapsed time T (sec) over d ($n = 10K$).

AQ fails to terminate on data sets with $d \geq 4$ because of its hidden factor exponential in d.

For all data sets in Figs. 13 and 14, PSkyline with Z-trees runs consistently faster than all other algorithms for probabilistic skyline computation. On average, it runs 2.60 times faster than PSkyline with ZBtrees, 3.55 times than PSkyline with R-trees, 19.88 times than AQ, and 167.23 times than EX. This result indicates that PSkyline is a significant improvement over previous probabilistic skyline algorithms and that Z-trees are more suitable for PSkyline than R-trees and ZBtrees. We also observe that K-PSkyline is quite efficient for top-k probabilistic skyline computation (on uncertain data sets). For example, it consistently outperforms PSkyline with Z-trees, running 6.68 times ($k = 5$) and 6.45 times ($k = 10$) faster on average.

The fast speed of PSkyline is mainly due to the two traversal functions for finding incomparable groups of instances. Fig. 15 shows the elapsed time on the second group of synthetic data sets for two different versions of PSkyline: the original PSkyline and its variant (denoted with superscript $^-$) without incomparability tests (lines 9 and 23 in Fig. 9). On average, PSkyline with R-trees, ZBtrees, and Z-trees runs 7.86, 4.32, and 7.84 times faster than its variant without incomparability tests, respectively. This result indicates that the two traversal functions of PSkyline are indeed highly effective in finding incomparable groups of instances *regardless of the choice for tree structures.*

We also observe in Fig. 15 that the speed of PSkyline with R-trees and ZBtrees deteriorates with the increase in d, which is due to the increasing incidence of overlapping regions between sibling nodes in R-trees and ZBtrees. The speed of PSkyline with Z-trees, however, remains almost irrelevant to d because the effectiveness of Z-trees is largely independent of d. The following table shows the percentage of dominance tests found unnecessary by incomparability tests of PSkyline with Z-trees, which remains stable across different values of d.

d	2	4	6	8	10
independent	97.7%	96.0%	98.1%	99.2%	99.5%
anti-correlated	99.3%	98.8%	99.0%	99.2%	99.2%

This result also implies that in the case of PSkyline with Z-trees, identifying blocks of instances with skyline probability zero does not contribute to its speed very much.

The following table shows the elapsed time (in seconds) on the NBA database when we consider three different dimensionalities: only number of points and number of rebounds ($d = 2$); also number of assists ($d = 3$); also number of steals and number of blocks ($d = 5$).
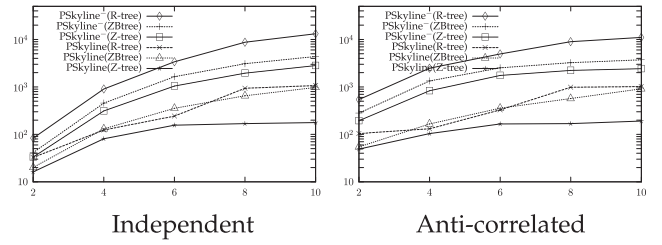
d	EX	AQ	PSkyline			K-PSkyline	
			R-tree	ZBtree	Z-tree	top-10	top-5
2	401.21	1.66	2.19	2.23	0.37	0.89	0.64
3	516.32	3.50	14.98	9.42	1.41	1.09	0.88
5	759.34	196.77	74.82	38.73	17.46	2.47	1.96

All the above observations indicate that the design principle for PSkyline, especially with Z-trees, works well for probabilistic skyline computation on uncertain data sets—by focusing on finding incomparable groups of instances, we achieve a probabilistic skyline algorithm that not only computes skyline probabilities of all objects, but also scales well to both large data sets and high-dimensional data sets.

## 8.2 Results of Online Probabilistic Skyline Computation

The experiments on online probabilistic skyline computation use both synthetic data streams and real data streams. To create synthetic data streams, we use the same procedure described in Section 8.1 except that all instances of each object are given distinct time stamps randomly chosen from interval $[1, 400]$. For real data streams, we reuse the NBA database described in Section 1. (Time stamps range from 1 to 2,087 as NBA games were played for a total of 2,087 days from the 1999 season to the 2008 season.)

We use the following two groups of synthetic data streams based on independent and anticorrelated distributions:

- For the first group (Fig. 16), we vary n from 250 to $3K$ with window size $W = 50$ and

  - $d = 2$ for low-dimensional data streams and
  - $d = 8$ for high-dimensional data streams.
- For the second group (Fig. 17), we vary d from 2 to 10 with window size $W = 50$ and $n = 3K$.

For all data streams in Figs. 16 and 17, O-PSkyline with Z-trees runs consistently faster than all other algorithms for online probabilistic skyline computation. On average, it runs 2.62 times faster than O-PSkyline with ZBtrees and 25.79 times than O-EX. K-PSkyline also consistently outperforms O-PSkyline with Z-trees, running 3.95 times ($k = 5$) and 3.38 times ($k = 10$) faster on average. The comparison excludes independent data streams with $d = 2$, on which O-PSkyline with Z-trees is already fast enough. Overall the results are unsurprising—O-PSkyline is a significant improvement over a naive algorithm, Z-trees are more suitable for O-PSkyline than ZBtrees, and K-PSkyline is quite efficient even on uncertain data streams.

The following table shows the elapsed time (in seconds) on the NBA database with window size $W = 50$.
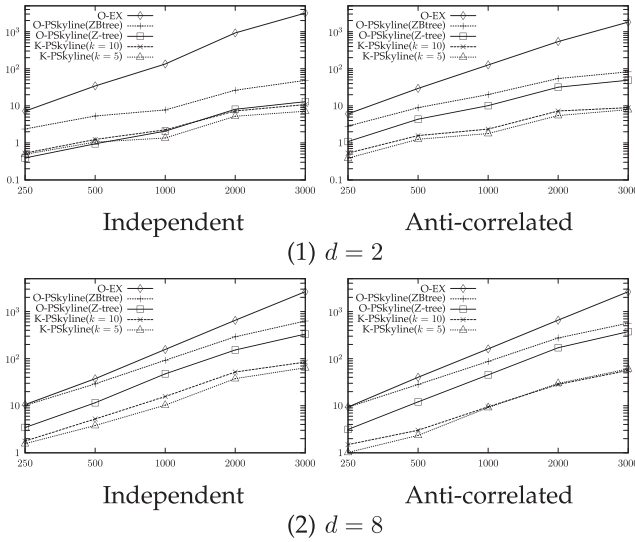
Fig. 16. Elapsed time T (sec) over n ($W = 50$).
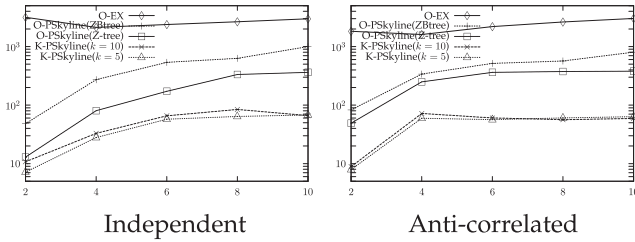
Fig. 17. Elapsed time T (sec) over d ($W = 50$, $n = 3K$).

d	O-EX	O-PSkyline		K-PSkyline	
		ZBtree	Z-tree	top-10	top-5
2	120.45	31.22	16.97	4.07	2.72
3	127.11	60.61	30.94	6.29	4.46
5	137.64	99.16	51.34	12.65	8.63

Although O-PSkyline consistently outperforms O-EX in all experiments, its improvement over O-EX is not so remarkable as the improvement of PSkyline over EX. This is partially due to the cost of recomputing skyline probabilities which is equally incurred by both O-EX and O-PSkyline whenever the time window moves (as in lines 11-13 in Fig. 10). For example, the average speedup of O-PSkyline with Z-trees over O-EX with respect to the total elapsed time is 35.74 on the second group of synthetic data streams. If we ignore the time for recomputing skyline probabilities, however, the average speedup increases to 79.33. Thus, we conclude that the design principle for O-PSkyline, especially with Z-trees, works well for online probabilistic skyline computation on uncertain data streams.

9 CONCLUSION

We have presented a probabilistic skyline algorithm PSkyline which exploits two recursive traversal functions to answer skyline queries on uncertain data sets. PSkyline is an improvement over previous work in [8] and [11] in terms of both functionality and scalability—it computes exact skyline probabilities of all objects and its performance is not limited by the size or the dimensionality of the data set. We have introduced a new in-memory tree structure called Z-tree to find as many incomparable

groups of instances as possible. We have also presented an online probabilistic skyline algorithm O-PSkyline which analyzes uncertain data streams, and a top-k probabilistic skyline algorithm K-PSkyline as the user may be interested only in a small number of objects with high skyline probabilities. The future work includes parallelizing our algorithms for multicore and distributed architectures.

REFERENCES

[1] Y. Tao, R. Cheng, X. Xiao, W.K. Ngai, B. Kao, and S. Prabhakar, "Indexing Multi-Dimensional Uncertain Data with Arbitrary Probability Density Functions," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pp. 922-933, 2005.

[2] J. Chen and R. Cheng, "Efficient Evaluation of Imprecise Location-Dependent Queries," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 586-595, 2007.

[3] M.A. Soliman, I.F. Ilyas, and K.C.-C. Chang, "Top-k Query Processing in Uncertain Databases," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 896-905, 2007.

[4] M. Hua, J. Pei, W. Zhang, and X. Lin, "Ranking Queries on Uncertain Data: A Probabilistic Threshold Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 673-686, 2008.

[5] X. Lian and L. Chen, "Probabilistic Ranked Queries in Uncertain Databases," *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT)*, pp. 511-522, 2008.

[6] K. Yi, F. Li, G. Kollios, and D. Srivastava, "Efficient Processing of Top-k Queries in Uncertain Databases," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 1406-1408, 2008.

[7] X. Lian and L. Chen, "Top-k Dominating Queries in Uncertain Databases," *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT)*, pp. 660-671, 2009.

[8] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic Skylines on Uncertain Data," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pp. 15-26, 2007.

[9] X. Lian and L. Chen, "Monochromatic and Bichromatic Reverse Skyline Search over Uncertain Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 213-226, 2008.

[10] W. Zhang, W. Lin, Y. Zhang, W. Wang, and J.X. Yu, "Probabilistic Skyline Operator over Sliding Windows," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 1060-1071, 2009.

[11] M. Atallah and Y. Qi, "Computing All Skyline Probabilities for Uncertain Data," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS)*, pp. 279-287, 2009.

[12] H. Su, E. Wang, and A. Chen, "Continuous Probabilistic Skyline Queries over Uncertain Data Streams," *Proc. Int'l Conf. Database and Expert Systems Applications (DEXA)*, pp. 105-121, 2010.

[13] C.C. Aggarwal and P.S. Yu, "A Survey of Uncertain Data Algorithms and Applications," *IEEE Trans. Knowledge and Data Eng.*, vol. 21, no. 5, pp. 609-623, May 2009.

[14] S. Zhang, N. Mamoulis, and D.W. Cheung, "Scalable Skyline Computation Using Object-Based Space Partitioning," *Proc. SIGMOD Int'l Conf. Management of Data*, pp. 483-494, 2009.

[15] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 47-57, 1984.

[16] K.C.K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the Skyline in Z Order," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pp. 279-290, 2007.

[17] J.A. Orenstein and T.H. Merrett, "A Class of Data Structures for Associative Searching," *Proc. ACM SIGACT-SIGMOD Symp. Principles of Database Systems (PODS),* pp. 181-190, 1984.

[18] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 421-430, 2001.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive Skyline Computation in Database Systems," *ACM Trans. Database Systems,* vol. 30, no. 1, pp. 41-82, 2005.

[20] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 717-719, 2003.

[21] P. Godfrey, R. Shipley, and J. Gryz, "Maximal Vector Computation in Large Data Sets," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 229-240, 2005.

[22] C. Böhm, F. Fiedler, A. Oswald, C. Plant, and B. Wackersreuther, "Probabilistic Skyline Queries," *Proc. ACM Conf. Information and Knowledge Management,* pp. 651-660, 2009.

[23] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and Efficiency in High Dimensional Nearest Neighbor Search," *Proc. SIGMOD Int'l Conf. Management of Data,* pp. 563-576, 2009.

[24] Y. Tao and D. Papadias, "Maintaining Sliding Window Skylines on Data Streams," *IEEE Trans. Knowledge and Data Eng.,* vol. 18, no. 3, pp. 377-391, Mar. 2006.

Hyeonseung Im is currently working toward the PhD degree in the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH). His research interests include database theory, query optimization, and programming language theory.

Sungwoo Park received the PhD degree in computer science from Carnegie Mellon University (CMU) in 2005. He has been an assistant professor in the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH) since 2006. His research interests include database theory, programming language theory, and logic in computer science.

Dongwon Kim is currently working toward the PhD degree in the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH). His research interests include database theory and query optimization.

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