Continuous Skyline Queries for Moving Objects

Zhiyong Huang Hua Lu Beng Chin Ooi Anthony K.H. Tung
School of Computing, National University of Singapore
{huangzy, luhua, ooibc, atung}@comp.nus.edu.sg

Abstract

The literature on skyline algorithms has so far dealt mainly with queries of static query points over static datasets. With the increasing number of mobile service applications and users, however, the need for continuous skyline query processing has become more pressing. A continuous skyline query involves not only static dimensions but also the dynamic one. In this paper, we examine the spatiotemporal coherence of the problem and propose a continuous skyline query processing strategy for moving query points. First, we distinguish the data points that are permanently in the skyline and use them to derive a search bound. Second, we investigate the connection between the spatial positions of data points and their dominance relationship, which provides an indication of where to find changes in the skyline and how to maintain the skyline continuously. Based on the analysis, we propose a kinetic-based data structure and an efficient skyline query processing algorithm. We concisely analyze the space and time costs of the proposed method and conduct an extensive experiment to evaluate the method. To the best of our knowledge, this is the first work on continuous skyline query processing.

Keywords: Skyline, continuous query processing, moving object databases.

Regular submission to IEEE TKDE (submitted in July 2005, first revised in December 2005, revised again in July 2006)
Original log number: TKDE-0272-0705

Contact Author:
Hua Lu
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543

Phone: (65)-6874-4774

1 Introduction

With rapid advances in electronics miniaturization, wireless communication and positioning technologies, the acquisition and transmission of spatiotemporal data using mobile devices are becoming pervasive. This fuels the demand for location-based services (LBS) [23, 4, 29, 28]. A skyline query retrieves from a given dataset a subset of interesting points that are not dominated by any other points [6]. Skyline queries are an important operator of LBS. For example, mobile users could be interested in restaurants that are near, reasonable in pricing, and provide good food, service and view. Skyline query results are based on the current location of the user, which changes continuously as the user moves.

The existing work on skyline queries assumes a static setting, where the distances from the query point to the data points do not change. Using the common example in the literature shown in Figure 1, assume there is a set of hotels, and for each hotel, we have its distance to the beach (*x axis*) and its price (*y axis*). The interesting hotels are all the points not worse than any other point in both the distance to the beach and the price. Hotels 2, 4 and 6 are interesting and can be derived by a skyline query, for their distances to the beach and their prices are preferable to those of any other hotels. Note that a point with minimum value in any dimension is a skyline point – hotels 2 and 6 for example. Also, skyline is different from convex hull in that it is not necessarily convex. In this example, hotel 4 makes the skyline not convex.

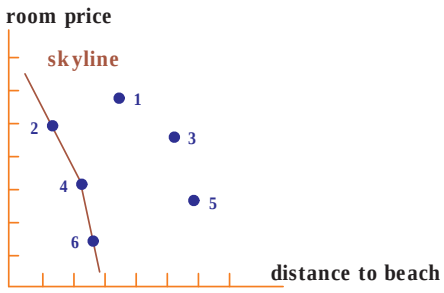
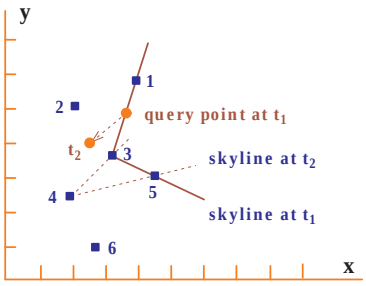


Figure 1. An example of skyline in a static scenario

In the above query, the skyline is obtained with respect to a static query point; in this case, it is the origin of both axes. Now, let us change the example to the scenario of a tourist walking about to choose a restaurant for dinner. We consider three factors in the skyline operation, namely the distance to the

restaurant, the average price of the food and the restaurant rank. Different from the previous example, the distance now is not fixed since the tourist is a moving object. Figure 2 shows the changes in the skyline due to the movement. In the figure, the positions of the restaurants are drawn in the X-Y plane while the table shows their prices and ranks. Lower values are preferred for all three dimensions. A tourist as the query point moves as the arrow indicates from time t_1 to t_2. The skyline – which refers to the interesting restaurants – changes with respect to the tourist's position. Skylines at different times are indicated by different line chains. The situation becomes more complex when all data points can move, which is frequent in real-time applications like e-games and digital war systems. For instance, one player in a field fighting game wants to keep track of those enemies who are close and most dangerous in terms of multiple aspects like energy, weapon, strategy and etc.



Restaurant	Price	Rank
1	6	4
2	5.8	4
3	4	1
4	2.8	3
5	1	2
6	2	4

Figure 2. Skylines in mobile environment

In this paper, we address the problem of continuous skyline query processing, where the skyline query point is a moving object and the skyline changes continuously due to the query point's movement. We solve the problem by exploiting its spatiotemporal coherence. Coherence refers to properties that change in a relevant way from one part to other parts within a scene in computer graphics [8], which is used to build efficient incremental processing for operations such as area filling and face detection. We use spatiotemporal coherence to refer to those spatial properties that do not change abruptly between continuous temporal scenes. The positions and velocities of moving points do not change by leaps between continuous temporal scenes, which enables us to maintain the changing skyline incrementally. First, we distinguish the data points that are permanently in the skyline and use them to derive a search bound to constrain the processing of the continuous skyline query. Second, we investigate the connection be-

tween the spatial locations of data points and their dominance relationship, which provides an indication of where to find changes in the skyline and update it. Third, to efficiently support the processing of continuous skyline queries, we propose a kinetic-based data structure and the associated efficient query processing algorithm. We present concise space and time cost analysis of the proposed method. We also report on an extensive experimental study, which includes a comparison of our proposed method with an existing method adapted for the application. The results show that our proposed method is efficient in terms of storage space, and is especially suited for continuous skyline queries. To the best of our knowledge, this is the first work on continuous skyline queries in the mobile environment.

The rest of this paper is organized as follows. In Section 2, we present the preliminaries including our problem statement and a brief review of related work. In Section 3, we present a detailed analysis of the problem. In Section 4, we propose our solution which continuously maintains the skyline for moving query points through efficient update. The experimental results are presented in Section 5. Section 6 concludes the paper.

2 Preliminaries

2.1 Problem Statement

In LBS, most queries are continuous queries [28]. Unlike snapshot queries that are evaluated only once, continuous queries require continuous evaluation as the query results vary with the change of location and time. Continuous skyline query processing has to re-compute the skyline when the query location and objects move. Due to the spatiotemporal coherence of the movement, the skyline changes in a smooth manner. Notwithstanding this, updating the skyline of the previous moment is more efficient than conducting a snapshot query at each moment.

For intuitive illustration, we limit the data and the moving query points to a two-dimensional (2D) space. Our statement is however sufficiently general for high-dimensional space too. We have a set of n data points in the format $<x_i, y_i, v_{xi}, v_{yi}, p_{i1}, ..., p_{ij}, ..., p_{im}>$ $(i = 1, ..., n)$, where x_i and y_i are positional coordinate values in the space, v_{xi} and v_{yi} are respectively velocity in the X and Y dimensions

while p_{ij}'s $(j = 1, ..., m)$ are static non-spatial attributes, which will not change with time.

For a moving object, x_i and y_i are updated using v_{xi} and v_{yi}. When it is stationary, v_{xi} and v_{yi} are zero. We use $Tuple(i)$ to represent the i-th data tuple in the database. Users move in the 2D plane. Each of them moves in velocity (v_{qx}, v_{qy}), starting from position (x_q, y_q). They pose continuous skyline queries while moving, and the queries involve both distance and all other static dimensions. Such queries are dynamic due to the change in spatial variables. In our solution, we only compute the skyline for (x_q, y_q) at the start time 0. Subsequently, continuously query processing is conducted for each user by updating the skyline instead of computing a new one from scratch each time. Moving points are allowed to change their velocities, which will be addressed in Section 4.2.1. Without loss of generality, we restrict our discussion to what follows the MIN skyline annotation [6], in which smaller values of distance or attribute p_{ij} are preferred in comparison to determine dominance between two points.

2.2 Time Parameterized Distance Function

In our problem, the distance between a moving query point and a data point is involved in the skyline operator. For a moving data point pt_i starting from (x_i, y_i) with velocity (v_{ix}, v_{iy}), and a query point starting from (x_q, y_q) moving with (v_{qx}, v_{qy}), the Euclidean distance between them can be expressed as a function of time t: $dist(q(t), pt_i(t)) = \sqrt{at^2 + bt + c}$, where a, b and c are constants determined by their starting positions and velocities: $a = (v_{ix} - v_{qx})^2 + (v_{iy} - v_{qy})^2$; $b = 2[(x_i - x_q)(v_{ix} - v_{qx}) + (y_i - y_q)(v_{iy} - v_{qy})]$; $c = (x_i - x_q)^2 + (y_i - y_q)^2$. For simplicity, we use function $f_i(t) = at^2 + bt + c$ to denote the square of the distance. When pt_i is static, a, b and c are still determined by the formulas above with $v_{ix} = v_{iy} = 0$. This time parameterized distance function has been used in literature to help processing queries in moving object databases [27, 10, 21].

2.3 Terminologies

For two points pt_1 and pt_2, if $dist(pt_1, q) \leq dist(pt_2, q)$ and $pt_1.p_k \leq pt_2.p_k, \forall k$, and at least one $<$ holds, i.e., $\exists k$, such that $pt_1.p_k < pt_2.p_k$, we say pt_1 dominates pt_2. We say pt_1 and pt_2 are $incomparable$ if pt_1 does not dominate pt_2, and pt_1 is not dominated by pt_2. We use $pt_1 \prec pt_2$ to represent that pt_1

dominates pt_2, and $pt_1 \preceq pt_2$ to represent that pt_1 dominates pt_2 for all static non-spatial dimensions.

In kinetic data structures, a *certificate* is a conjunction of algebraic conditions, which guarantees the correctness of some relationship to be maintained between mobile data objects. Readers are referred to [3] for the formal and detailed description of kinetic data structures (KDS). In this paper, we use a certificate to ensure the status of a data point is valid within a period of time t. For example, a certificate of a point can guarantee it staying in the skyline for a period of time t. Beyond t, its certificate is invalid and an event will trigger a process to update the certificate, which may result in a change in the skyline.

2.4 Related Work

One area with related work concerns skyline queries. Inspired by work on contour problem [15], maximum vectors [14], convex hull [20] and multi-objective optimization [25], Borzonyi, Kossmann and Stocker [6] introduced the skyline operator into relational database context and proposed two processing algorithms: *Block Nested Loop* (BNL) and *Divide-and-Conquer* (D&C). D&C approach partitions the dataset into several parts, processes each part in memory and finally merges all partial skylines together. BNL scans the dataset sequentially and compares each new point to all skyline candidates kept in memory. Chomicki, Godfrey, Gryz and Liang [7] proposed a variant of BNL by pre-sorting the dataset according to some monotone scoring function. Tan, Eng and Ooi [26] proposed two progressive processing algorithms. In *Bitmap* approach, each data point is encoded in a bit string and skyline is computed by some efficient operations on bit matrix of all data points. In *Index* approach, data points are transformed into a single dimensional space and then indexed by B^+-tree which facilitates skyline computation. Kossmann, Ramsak and Rost [13] proposed another progressive processing algorithm *Nearest Neighbor* (NN) based on the depth-first nearest neighbor search [22] via R^*-tree. Papadias and Tao [18, 19] proposed an improved algorithm named $Branch\text{-}and\text{-}Bound\ Skyline$ (BBS) based on the best-first nearest neighbor search [9]. By accessing only nodes that contain skyline points, BBS incurs optimal node access and so far is the most efficient skyline algorithm in static settings. In a slightly different context, Balke, Guntzer and Zheng [2] addressed skyline operation over web databases where different dimensions are stored in different data sites.

Another area with related work is that of kinetic data structures (KDS). Basch, Guibas and Hershberger [3] proposed a conceptual framework for KDS as a means to maintain continuously evolving attributes of mobile data. KDS keeps the relationship of interest between data in some specific structures, and the contents do not change unless the relationship has changed. In this way, data retrieval results based on the relationship of interest can be maintained when the data points move continuously. KDS and its underlying ideas have inspired some unique query processing techniques for moving objects database (MOD). Mokhtar, Su and Ibarra [16] proposed an event-driven approach to maintain the results of k-NN queries on moving objects while time elapses. All moving objects are sorted by their distance to the query point, while events are computed and stored to indicate when and how the order will change. To reduce the points sorted in the KDS, Iwerks, Samet and Smith [10] proposed the Continuous Windowing (CW) k-NN algorithm, which limits search to a smaller region and accesses other points only as needed.

3 The Change of Skyline in Moving Context

In this section, we analyze the change in skyline in continuous query processing. We first point out the search bound that can be used to filter out unqualified data points in determining the skyline for a moving query point. Then we carry out an analysis of the skyline change due to the movement, which reveals some insights for the algorithms in the next section.

3.1 Search Bound

Although in our problem the skyline operator involves both dynamic and static dimensions, some data points could be always in the skyline no matter how the data points and query points move. This is because they have dominating static non-spatial values, which guarantee that no other objects can dominate them. We denote this subset of skyline points as SK_{ns} and the whole set of skyline points as SK_{all}. We call SK_{ns} the *static partial skyline*, and SK_{all} the $complete\ skyline$.

We call points in SK_{ns} *permanent skyline points*. In this way, we distinguish those points always in the complete skyline from the rest of the dataset. The benefit of this discrimination is threefold:

(1) It extracts the unchanging part of a continuous skyline query result from the complete skyline SK_{all}. This allows efforts in query processing to be concentrated on the changing part only, i.e., $SK_{all} - SK_{ns}$. We name that part SK_{chg}, and call those points in it *volatile skyline points*. In continuous skyline query processing, only SK_{chg} needs tracking for each query. In this manner, we can reduce overall processing cost.

(2) The discrimination can reduce the amount of data to be sent to clients. Since SK_{ns} is always in the final skyline result, we need to send it only once from server to client. This benefits mobile applications where clients and servers are usually connected via limited bandwidth.

(3) Static partial skyline SK_{ns} also provides an indication of the search bound for processing a continuous skyline query. Since SK_{ns} is always contained in SK_{all}, for any point not in SK_{ns} to enter SK_{all}, it must be incomparable to any item in SK_{ns}. More specifically, it must have advantage in distance to the query point since it is dominated in all static dimensions by at least one point in SK_{ns}. This leads to Lemma 3.1.1.

Lemma 3.1.1 *At any time t, if sp_f is the farthest point in SK_{ns} to the query point, then any point pt not nearer to the query point than sp_f is not in the complete skyline.*

Proof. Obviously $pt \notin SK_{ns}$, thus $\exists sp \in SK_{ns}$ s.t. $\forall k, sp.p_k \leq pt.p_k$ and at least one inequality holds. From $dist(q, sp) \leq dist(q, sp_f)$ and $dist(q, sp_f) \leq dist(q, pt)$, we get $dist(q, sp) \leq dist(q, pt)$ by transitivity. Because of its disadvantage in both spatial and non-spatial dimensions, pt is dominated by sp at time t so that it is not in the complete skyline. $\diamond$

Lemma 3.1.1 indicates a search bound for the complete skyline. This can be used to filter out unqualified points in query processing: those farther away than all points in SK_{ns} cannot be in the complete skyline. Refer to the example in Figure 2, $SK_{ns} = \{3, 5\}$. At time t_1, $SK_{chg} = \{1\}$ and restaurants 2, 4 and 6 are not in the skyline as they are farther to the query point than restaurant 5, which is the farthest permanent skyline point to the query point.

3.2 Change in the Skyline

When the query point q and data points move, their distance relationships may change. This causes the skyline to change as well. As discussed in Section 3.1, such changes only happen to SK_{chg}, i.e. $SK_{all} - SK_{ns}$. It is also mentioned in Section 2.2 that the square of the distance from each point to the query point can be described as a function of time t. Figure 3 illustrates an example of such functions of several points with respect to the moving query point.

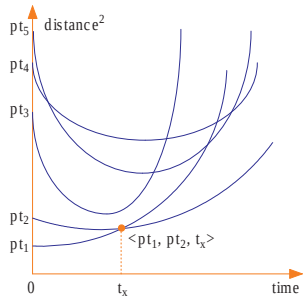


Figure 3. An example of distance function curves

Intuitively, a skyline point s_i in SK_{chg} before time t_x may leave the skyline after t_x. On the other hand, a non-skyline point nsp at time t_x may enter the skyline and become part of SK_{chg} after t_x. For the former, after time t_x, s_i must be dominated by a skyline point s_j in SK_{all}. For the latter, when nsp enters the skyline after time t_x, those points that used to dominate nsp before t_x will stop dominating it. That moment t_x is indicated by an intersection of two distance function curves. We use $<pt_1, pt_2, t_x>$ to represent an intersection shown in Figure 3, where at time t_x point pt_2 is getting closer to the query than point pt_1, opposite to the situation before t_x. From the figure, we can see that such an intersection only alters pt_1 and pt_2's presence in or absence from SK_{chg} if it does cause change. This is because before and after the intersection, the only change of comparison is $dist(q, pt_1) < dist(q, pt_2)$ to $dist(q, pt_2) < dist(q, pt_1)$. If no intersections happen, the skyline does not change at all because the inequality relationship between the distances of all points to the query point remains unchanged. Nevertheless, not every intersection necessarily causes the skyline to change. Whether an intersection $<pt_1, pt_2, t_x>$ causes change is relevant to which set pt_1 and pt_2 belong to just before time t_x, i.e., SK_{ns}, SK_{chg} or $\overline{SK_{all}}$ (neither of the former two, i.e., not in SK_{all}). We have following lemmas to

clearly describe these possibilities.

Lemma 3.2.1 *An intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) has no influence on the skyline if one of the following conditions holds before* t_x*:*

(1) $pt_1 \in SK_{ns}$ *and* $pt_2 \in SK_{ns}$

(2) $pt_1 \in SK_{ns}$ *and* $pt_2 \in SK_{chg}$

(3) $pt_1 \notin SK_{all}$ *and* $pt_2 \in SK_{ns}$

(4) $pt_1 \notin SK_{all}$ *and* $pt_2 \in SK_{chg}$

(5) $pt_1 \notin SK_{all}$ *and* $pt_2 \notin SK_{all}$

Proof. (1) This is obvious according to the definition of permanent skyline points.

(2) Obviously pt_1 does not leave the skyline. Assuming that pt_2 leaves the skyline after t_x, there must be another skyline point s dominating it, i.e., $dist(q, s) < dist(q, pt_2)$ for $t > t_x$ and $\forall k, s.p_k \leq pt_2.p_k$. Since intersection $<pt_1, pt_2, t_x>$ does not change the distance inequality relationship between s and pt_2, $dist(q, s) < dist(q, pt_2)$ also holds for $t < t_x$. Thus s dominates pt_2 before t_x, which contradicts $pt_2 \in SK_{chg}$ before t_x. Therefore pt_2 does not leave the skyline either, and there is no influence on the skyline.

(3) Since $pt_1 \notin SK_{all}$ before t_x, there must be at least one skyline point $s \in SK_{all}$ dominating it. Because $dist(q, s) < dist(q, pt_1)$ does not change after the intersection, s still dominates pt_1 and thus pt_1 will not enter the skyline. Since pt_2 is a permanent skyline point, it will not leave the skyline.

(4) Due to the same reasoning as in (3), pt_1 will not enter the skyline after t_x. Due to the same reasoning in (2), pt_2 itself will not leave the skyline after t_x.

(5) Due to the same reasoning as in (3), neither pt_1 nor pt_2 will enter the skyline after t_x. $\diamond$

Lemma 3.2.2 *An intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) may have influence on the skyline if one of the following conditions holds before* t_x*:*

(1) $pt_1 \in SK_{ns}$ *and* $pt_2 \notin SK_{all}$

(2) $pt_1 \in SK_{chg}$ *and* $pt_2 \in SK_{ns}$

Table 1. Intersections and possible skyline changes

$pt_1 \setminus pt_2$	SK_{ns}	SK_{chg}	$\overline{SK_{all}}$
SK_{ns}	—	—	$\surd$
SK_{chg}	$\surd$	$\surd$	$\surd$
$\overline{SK_{all}}$	—	—	—

(3) $pt_1 \in SK_{chg}$ *and* $pt_2 \in SK_{chg}$

(4) $pt_1 \in SK_{chg}$ *and* $pt_2 \notin SK_{all}$

Proof. (1) Obviously pt_1 will not leave the skyline after t_x. Since $pt_2 \notin SK_{all}$ before t_x there must be at least one skyline point in SK_{all} dominating it. If pt_1 is the only dominating pt_2 before t_x, after t_x, pt_1 will stop dominating pt_2 and no other skyline points will dominate it. Consequently, pt_2 will enter the skyline after t_x.

(2) Obviously pt_2 will not leave the skyline after t_x. But if $\forall k, pt_2.p_k \leq pt_1.p_k$ holds, pt_2 will dominate pt_1 and cause pt_1 to leave the skyline since $dist(q, pt_2) < dist(q, pt_1)$ holds after t_x.

(3) If $\forall k, pt_2.p_k \leq pt_1.p_k$ holds, pt_2 will dominate pt_1 and cause pt_1 to leave the skyline because $dist(q, pt_2) < dist(q, pt_1)$ holds after t_x. Due to the same reasoning as in (2) of Lemma 3.2.1, pt_2 itself will not leave the skyline since no other points will dominate it after t_x.

(4) Due to the same reasoning as in (1), pt_2 may enter the skyline after t_x. $\diamond$

Table 1 lists all possibilities attached to an intersection. For (4) in Lemma 3.2.2, an interesting issue is whether pt_2 can dominate pt_1 after time t_x.

Lemma 3.2.3 *For an intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) in which* $pt_1 \in SK_{chg}$ *and* $pt_2 \notin SK_{all}$ *before* t_x*,* pt_1 *will not be dominated by* pt_2 *and leave the skyline after* t_x*, if no other intersection happens at the same time and the static non-spatial values of* pt_1 *and* pt_2 *are not the same for all dimensions.*

Proof. Assume that pt_1 will be dominated by pt_2 and leave the skyline after t_x, we have $pt_2 \preceq pt_1$. Because pt_2 is not in SK_{all} before t_x, in SK_{all} there must exist at least one pt_3 dominating pt_2, i.e. $pt_3 \prec pt_2$. For simplicity of presentation, we assume that pt_3 is the only one skyline point of such kind. By transitivity, we have $pt_3 \preceq pt_1$. But because pt_1 is in SK_{chg}, the distance from pt_3 to the query

point must be larger than that from pt_1 before t_x; otherwise $pt_3 \prec pt_1$ means pt_1's absence from SK_{chg}. Thus for pt_2 to dominate pt_1 after t_x, it must first become incomparable to pt_3, which requires that an intersection between pt_1 and pt_3 must happen no later than t_x. If the time of intersection is earlier than t_x, however, pt_2 will be in SK_{chg} before t_x. Thus that time must only be t_x. Therefore, their three distance function curves must intersect at the same point, and $<pt_1, pt_2, t_x>$ is not the only intersection at time t_x.

Note that pt_3 cannot be pt_1 in the above proof. Otherwise, before t_x, we have $pt_1 \prec pt_2$. Thus, $\exists k$ such that $pt_1.p_k < pt_2.p_k$ because their static non-spatial attribute values are not the same for all dimensions. This means pt_2 cannot dominate pt_1 even after time t_x. $\diamond$

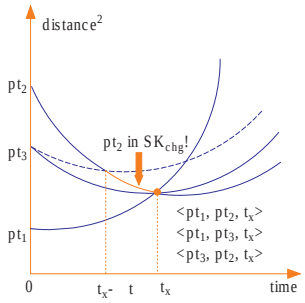


Figure 4. An example of multiplex intersection

Figure 4 shows such a scenario indicated by Lemma 3.2.3, and we call such an intersection *multiplex intersection*. One feasible processing strategy for this situation is to only consider if pt_2 has the chance to enter SK_{chg}. We need to check if pt_1 is the only one that used to dominate pt_2. We ignore the possibility that pt_2 might enter the skyline and start dominating pt_1 at the same time. That possibility is indicated by other intersections at the same time, each of which is to be processed in isolation.

Accordingly, the intersection $<pt_1, pt_2, t_x>$ in Figure 4 will be ignored. After time t_x, both pt_2 and pt_3 are in SK_{all} but pt_1 is not. This result can be achieved as long as the three intersections are correctly processed one by one according to our discussion above, regardless of the order in which they are processed. Now, let us look at the processing of the intersections in the order listed in the figure. First, $<pt_1, pt_2, t_x>$ does not change the skyline because pt_1 does not dominate pt_2 and thus pt_2 will not enter SK_{chg} though it is getting closer to the query point than pt_1. Second, $<pt_1, pt_3, t_x>$ will

cause pt_1 to leave SK_{chg} because pt_1 starts dominating it. Finally, $<pt_3, pt_2, t_x>$ will cause pt_2 to enter SK_{chg} because pt_3 is the only one that used to dominate pt_2 and now it stops dominating the point as its distance to the query point becomes larger. The procedures of other processing orders are similar and thus omitted due to space constraint.

An extreme situation is that many distance function curves are involved in the same multiplex intersection. Our processing strategy can also ensure the correct change as long as each legal intersection is processed correctly in isolation. In fact, this situation is rather special and seldom happens because it requires that all the points involved to be on the same circle centered at the query point. This situation usually happens to minority data points only, and it becomes more infrequent in the moving context.

To summarize the above analysis, we only need to take into account two primitive cases in which the skyline may change.

Case 1 *Just before time t_x, $s_i \in SK_{chg}$ and $\exists s_j \in SK_{all}$ s.t. $s_j \preceq s_i$. At time t_x, an intersection $<s_i, s_j, t_x>$ between their distance function curves happens. Then from time t_x on, $s_i \notin SK_{chg}$ and leaves the skyline because $s_j \prec s_i$, and $s_j \in SK_{all}$ still.*

Case 2 *Just before time t_x, $nsp \notin SK_{all}$ and $\exists s_i \in SK_{all}$ s.t. $s_i \prec nsp$. At time t_x, an intersection $<s_i, nsp, t_x>$ between their distance function curves happens. Then from time t_x on, $nsp \in SK_{chg}$ because $\nexists s_j \in SK_{all}$ s.t. $s_j \prec nsp$.*

Case 1 determines a skyline change, whereas Case 2 suggests a possibility of change which requires further checking. For a period of time before the change in Case 1, s_j must be out of the circle determined by the query point q and s_i. We use $Cir(q, s_i)$ to denote the circle whose center is q and radius is $dist(q, s_i)$. In Case 2, the possible non-skyline point nsp is also out of circle $Cir(q, s_i)$ for a period of time before the change. Namely, the distance from each current skyline point (permanent or volatile) provides indication of future change in the skyline.

3.3 Continuous Skyline Query Processing

We now address the issues of continuous skyline query processing. A naive way is to pre-compute and store all possible intersections of any pair of distance function curves, and then process each one when its time comes according to the discussion in Section 3.2. This method produces many false hits which actually do not cause skyline to change as we have shown in Table 1.

Based on those observations, we compute and store intersections in an evolving way. We only keep those intersections with possibility to change the skyline according to Table 1. Specifically, first, we get the initial skyline and compute some intersections of the distance curves in terms of the current skyline points. Then, when some intersections happen and the skyline is changed, we further compute intersections in terms of the updated skyline. By looking into the near future, we ensure that the skyline query result is kept updated, and more information will be obtained later for updating the skyline further into the future.

Besides, we keep all the current skyline points sorted based on their distance to the query point. At each evolving step, we only compute those possible intersections that involve points between two adjacent skyline points s_i and s_{i+1}, and will happen before s_i and s_{i+1} stop being adjacent. Therefore, we need to keep track of any intersection between two skyline points that are adjacent to each other in sorted order.

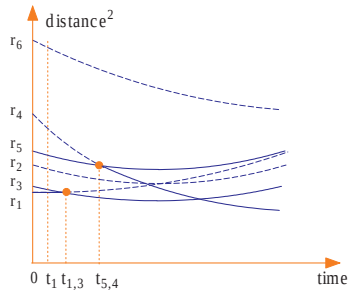


Figure 5. An example of evolving intersections

Figure 5 shows the distance curves of the restaurant example in Figure 2. At time t_1, restaurants r_1, r_3 and r_5 are three adjacent skyline points, and only those two dotted intersections are computed and stored for future processing. Then at time $t_{1,3}$, r_1 will leave the skyline as r_3 becomes to dominate it. Next at

time $t_{5,4}$, r_4 will enter the skyline as its only dominator r_5 stops dominating it. Not all intersections are stored for processing, e.g., the intersection between r_2 and r_4, and that between r_4 and r_1.

Note our method is a kind of sweeping algorithm but with two distinctive features. We have a search bound which renders the search limited in some specific regions instead of the whole data space. The case study in Section 3.2 helps identify result changes and reduce processing in the maintenance. The next section addresses the data structure and relevant algorithms in detail.

4 Data Structure and Algorithms

4.1 Data Structure

We use a bidirectional linked list, named L_{sp} to store all current skyline points, which are sorted in ascending order of their distances to the query point. For each current skyline point s_i, we keep an entry of form $(flag, tuple\_id, a, b, c, t_v, t_{skip})$. $flag$ is a boolean variable indicating if s_i is in SK_{ns}. $tuple\_id$ is the tuple identifier of s_i which can be used to access the record. a, b, c are coefficients of the distance function between s_i and query point q, introduced in Section 2.2. t_v is only available to each changing skyline point, indicating its validity time. t_{skip} is the time when s_i will exchange its position with its successor in L_{sp}. Besides L_{sp}, a global priority queue Q_e is used to hold all events derived from certificates to represent future skyline changes, with preference being given to earlier events.

Based on the analysis in Section 3, we define three kinds of certificates used in the KDS, which are listed in Table 2. The first column is the name of a certificate, the second is what the certificate to guarantee, and the third lists the data points involved in the certificate.

An event occurs when any certificate fails due to the distance change resulting from movement. Each event is in the form of $(type, time, self, peer)$, where $type$ represents the kind of its certificate; $time$ is a future time instance when the event will happen; and $self$ and $peer$ respectively represent skyline point and relevant data point involved in the event.

Certificate $s_i s_j$ ensures for an existent volatile skyline point s_i that any other skyline point s_j with the potential to dominate s_i ($s_j \preceq s_i$) keeps being farther to query point q than s_i, therefore s_i is not

Table 2. Certificates

Cert.	Objective	Data Points
$s_i s_j$	$\forall s_i \in SK_{chg}, s_j \in SK_{all}$, s.t. $s_j \preceq s_i \rightarrow dist(q, s_i) < dist(q, s_j)$	$self = s_i$ $peer = s_j$
nsp_{ij}	$\forall nsp_j \notin SK_{all}, \forall s_i \in SK_{all}$, s.t. $s_i \prec nsp_j \rightarrow$ $dist(q, s_i) \leq dist(q, nsp_j)$	$self = s_i$ $peer = nsp_j$
ord_{ij}	$\forall s_i \in SK_{all}$, s.t. $\exists s_j \in SK_{all} \wedge s_j \not\preceq s_i$ $\wedge s_j = s_i.next$ in L_{sp} $\rightarrow dist(q, s_i) < dist(q, s_j)$	$self = s_i$ $peer = s_j$

dominated by any of them and stays in the skyline. Here $self$ and $peer$ respectively point to s_i's and s_j's entries in L_{sp}.

Certificate nsp_{ij} ensures for a non-skyline point nsp that all those skyline points currently dominating it keeps being closer to query point q than nsp, therefore nps is prevented from entering the skyline. When a certificate of this kind fails at $time$, nsp will get closer to query point q than one skyline point s_i, but whether it will enter the skyline or not depends on whether s_i is the only one that used to dominate it. This will be checked when an event of this kind is being processed. Here $self$ points to s_i's entry in L_{sp}, whereas $peer$ is the tuple identifier of data point nsp.

Certificate ord_{ij} ensures for an existent skyline point s_i that its successor s_j in L_{sp} keeps being farther to query point q than it. This s_j does not have the potential to dominate s_i, otherwise an $s_i s_j$ certificate will be used instead. Here $self$ points to the entry of the predecessor skyline point in the pair, and $peer$ to the successor. Certificate ord_{ij} not only keeps the order of all skyline points in L_{sp}, but also implies a way to simplify event computation and evolvement. For Case 1 described in Section 3.2, it also involves a position exchange in L_{sp}, i.e., just before s_j dominating s_i, s_j must be its successor. And we need to determine if an exchange in L_{sp} really results in $s_i s_j$ event. In this sense, we only need to check for s_i its successor to compute a possible $s_i s_j$ or ord_{ij} event. If s_i does have an $s_i s_j$ or ord_{ij} event, the event's $time$ value is exactly s_i's validity time t_v. If s_i has no such event, its validity time is set to infinity. An event of certificate nsp_{ij} with ***self*** = s_i is supposed to have a time stamp no later than $s_i.t_v$, and those events with a later time are not considered.

Initially, L_{sp} contains the current skyline points, and Q_e contains events that will happen in the nearest future. As time elapses, every due event is dequeued and processed based on its $type$. While processing due events and updating the skyline accordingly, our method also creates new events for future. Thus, Q_e evolves with due events being dequeued and new events being enqueued, providing information for correctly maintaining the skyline. At any time t after all due events are processed, L_{sp} is the correct skyline with respect to the query point q's current position.

4.2 Algorithms

For a given dataset, its SK_{ns} is pre-computed and stored as a system constant. Before maintaining skyline continuously, an initialization is invoked to compute the initial SK_{chg} and the earliest events. To compute SK_{chg} over static dataset for the query point's starting position, in order to use the search bound determined by SK_{ns} and reduce intermediate steps to access data tuples when computing events, we use the grid file to index all data points. Grid file provides a regular partition of space and at most two-disk-access feature for any single record [17]. In our solution for the static dataset, we use a simple uniform 2D grid file dividing the data space into $h \times v$ cells to index D', and the data points within each cell are stored in one disk page.

For the similar reasons we use a hash based method [24] to index moving data points in D'. The data space is also divided into regular cells, with each representing a bucket to hold all those moving data points within its extent. Data points can move across adjacent cells with the velocities in its tuple, which is monitored by a pre-processing layer and declared in an explicit update request to the database. An update request can also change a data point's speed. How to deal with the updates of moving data points to maintain the correct skyline will be addressed in Section 4.2.1. Except for the difference on underlying indexing schemas, the initializations for static and moving datasets share the same framework and events creation algorithm.

The initialization framework is presented in Figure 6. First all permanent skyline points in SK_{ns} are inserted into L_{sp} according to their distance to query point q's starting position. The farthest distance is recorded in variable d_{bnd} as the search bound. Then starting from cell $cell_{org}$ where q's starting position

Algorithm initialization(q)
Input: q is the query point
Output: the skyline for q's starting position
the event queue to be used in maintenance
// load SK_{ns} into skyline list
1. **for each** s_i in Sk_{ns}
2. Compute a, b, c in terms of q;
3. Insert an entry $(1, s_i, a, b, c, \infty, \infty)$ into L_{sp};
// search bound determined by SK_{ns}
4. $d_{bnd} = dist(L_{sp}.last, q)$;
// compute initial skyline
5. Search the grid cell $cell_{org}$ in which q lies;
6. **while** there still exist grid cells unsearched
7. **for each** cell $cell_i$ on next outer surrounding circle
8. **if** ($mindist(q, cell_i) \geq d_{bnd}$)
9. **break;**
10. **else** Search $cell_i$;
// compute events
11. **for each** s_i from $L_{sp}.last.prev$ to $L_{sp}.first$
12. createEvents(s_i, q);
13. handleBound(q, t_{cur});

Figure 6. Initialization framework

lies, all grid cells are searched in a spiral manner that those on an inner surrounding circle are searched before those on an outer one. Cells beyond d_{bnd} are pruned, where $mindist$ is computed as in [22] by regarding a cell as an MBR. Points in a cell not pruned are sequentially compared to the current skyline points in L_{sp}, which is adjusted with deletion or insertion if necessary. After all cells are searched or pruned, algorithm createEvents is invoked for each skyline point s_i from outermost to innermost, to compute all events for all skyline points except the last one s_{last}. Finally, algorithm handleBound is called to compute a possible nsp_{ij} event for those points farther than s_{last}.

Algorithm handleBound is presented in Figure 7. It does not involve all outer non-skyline points of s_{last}'s, instead it is limited to an estimated region. This region C is the difference between the two circles determined by s_{last} and query point q at two different times, the current time and the earliest event time t_{next} in the future. Only those non-skyline points in C have chance to enter the skyline before t_{next}.

Algorithm createEvents is presented in Figure 8. For a given skyline point s_i in L_{sp}, the algorithm first computes the time t when s_i and the next skyline point s_j in L_{sp} will exchange their position in

Algorithm handleBound(q, t_{cur})
Input: q is the query point
Output: upcoming events for $L_{sp}.last$
1. $t_{next} = Q_e.first.time$;
2. $s_{last} = L_{sp}.last$;
3. $C = Cir(q(t_{next}), s_{last}) - Cir(q(t_{cur}), s_{last})$
4. **for each** point nsp in C
5. **for each** s_j from s_{last} to $L_{sp}.first$
6. t = time nsp will get closer to q than s_j;
7. **if** (($t \geq s_j.t_v$) **or** ($t \geq s_j.t_{skip}$)) **continue**;
8. **if** ($\forall k, s_j.p_k \leq nsp.p_k$)
9. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
10. **break**;

Figure 7. Handle bound

the list, i.e. when s_j will get closer to q than s_i. If t is later than s_j's skip time or s_i's validity time, it is ignored. Otherwise, it means an s_is_j event depending on s_j's validity time if $s_i \in SK_{chg}$, or it is a simple order change event. Then for each non-skyline point nsp between $Cir(q, s_i)$ and $Cir(q, s_j)$, the algorithm computes nsp_{ij} event by looping on all skyline points in the inner of nsp. Once an nsp event is derived, the loop on all inner skyline points breaks.

In maintaining the skyline, the due events are dequeued and processed according to its type, and new events are computed based on new positions. As in the initialization, the event of points out of the last skyline point is computed in a special way with an estimated search region by calling handleBound. The actions to process each kind of events are described as follows. For an s_is_j event, s_i is removed from the skyline and new events are computed for s_i's predecessor because its successor skyline point in L_{sp} has been changed. For an nsp_{ij} event, the non-skyline point nsp will be checked against all those skyline points closer to the query point, to see if they will enter the skyline. If not, a possible new nsp event is computed. Otherwise it will be added into the skyline and events will be computed for itself and its predecessor. For an ord_{ij} event the L_{sp} is correctly adjusted by switching s_i and s_j, and events are computed for themselves and their predecessor.

Algorithm createEvents(s_i, q)
Input: s_i is a skyline point in L_{sp}
q is the query point
Output: upcoming events for s_i
1. $peer = null$;
// compute events with next skyline point in L_{sp}
2. $s_j = s_i.next$;
3. t = time s_i and s_j will exchange position;
4. **if** (($t < s_j.t_{skip}$) **and** ($t < s_j.t_v$))
5. **if** (!$s_i.flag$)
6. **if** (($t < s_i.t_v$) **and** ($\forall k, s_j.p_k \leq s_i.p_k$))
7. $s_i.t_v = t$; $peer = s_j$;
8. **else** $s_i.t_{skip} = t$;
// enqueue relevant events
9. **if** ($peer \neq null$)
10. Enqueue ($s_i, s_i.t_v, rep, s_i s_j$) to Q_e;
11. **if** ($s_i.t_{skip} < s_i.t_v$)
12. Enqueue ($s_i, s_i.t_{skip}, s_j, ord_{ij}$) to Q_e;
// compute events involving non-skyline points
13. **for each** point nsp between $Cir(q, s_i)$ and $Cir(q, s_j)$
14. **for each** s_k from s_i to $L_{sp}.first$
15. t = time nsp will get closer to q than s_k;
16. **if** (($t \geq s_k.t_v$) **or** ($t \geq s_k.t_{skip}$)) **continue**;
17. **if** ($\forall k, s_k.p_k \leq nsp.p_k$)
18. Enqueue (s_k, t, nsp, nsp_{ij}) to Q_e;
19. **break**;

Figure 8. Create events

4.2.1 Updating the Moving Plan

A moving data point mpt_i's distance function does not change unless its moving plan changes. When this happens, the intersections of its distance function and other points' will also be changed as a consequence, which invalidates those events computed based on mpt_i's old distance function. Figure 9 shows how a data point's velocity change causes the intersections of the function curves to change. Thus, it may cause the skyline to change in the future.

To ensure correct process with updates, we need to add for each moving object's tuple a field t_{upt} indicating its last update time. We define an update request for any moving data point mpt_i in the form $update(id, x, y, v_x, v_y)$. id is mpt_i's identifier which can be used to locate its tuple directly. x and y represent its current position. v_x and v_y represent its current speed. The algorithm updateMotion in

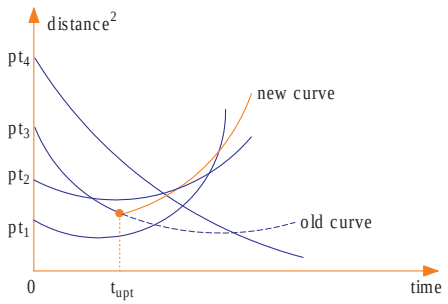


Figure 9. An example of the change of moving plan

Figure 10 is used to process such updates. When an update request comes in, it is first checked if mpt_i has moved to a new cell and if its speed has been changed since the last update. If x and y indicate that mpt_i has moved to a different cell, we need to remove it from the old one and insert it into the new one (line 1-5), which incurs 2 IOs. If v_x and v_y indicate that mpt_i's speed is not changed, the algorithm stops (line 6-7). Otherwise, we need to update the speed record for mpt_i (line 8-10), and adjust relevant events starting from the first skyline points till the first one out of mpt_i (line 17). If mpt_i is a skyline point, then its events will be re-computed and the algorithm stops (line 12-15). Otherwise, the algorithm continues to compute nsp_i events for mpt_i (line 19-24). With the independent distribution assumption, $(|SK_{all}| + 1)/2$ skyline points are expected to be accessed. To facilitate location of events involving a data point efficiently, the priority event queue is implemented using a B$^+$-tree, and each current skyline point s_i has a list of pointers to all those events whose $self$ is s_i.

It also can be seen in Figure 9 that right at the moment t_{upt} when an update request comes in, the skyline does not change abruptly. To keep the skyline correct, the update request is only processed after all due events are processed, i.e., updateMotion(req) at time t_{upt} executes after updateSkyline(t_{upt}) completes.

4.3 Cost Analysis and Discussion

The space cost incurred by our method consists of two components: the space used to keep the skyline and that used to store events. For a d-dimensional dataset with N points subject to independent distribution, the expected size of its skyline is $n_{sky} = O((\ln N)^{d-1})$ [5]. Since there are m static dimensions

Algorithm updateMotion(req)
Input: req is an update request
Output: updated hash index, tuple and Q_e
1. $cell_1 = Tuple(req.id).cell$;
2. $cell_2 = Hash(req.x, req.y)$;
3. **if** ($cell_1 \neq cell_2$)
4. $Tuple(req.id).cell = cell_2$;
5. remove $req.id$ from $cell_1$ and insert it to $cell_2$;
6. **if** (($req.v_x == Tuple(req.id).v_x$) **and** ($req.v_y == Tuple(req.id).v_y$))
7. **return**;
8. $Tuple(req.id).v_x = req.v_x$
9. $Tuple(req.id).v_y = req.v_y$
10. $Tuple(req.id).t_{upt} = t_{cur}$
// Adjust relevant events
11. **for each** s_i in L_{sp} from $L_{sp}.first$
12. **if** ($s_i.tuple\_id == req.id$)
13. Delete all s_i's events;
14. createEvents(s_i, q);
15. **return**;
16. Delete all s_i's events with $peer == req.id$;
17. **if** ($dist(q, Tuple(req.id)) \leq dist(q, s_i)$) **break**;
18. $nsp = req.id$;
19. **for each** s_j from s_i to $L_{sp}.first$
20. t = time nsp will get closer to q than s_j;
21. **if** (($t \geq s_j.t_v$) **or** ($t \geq s_j.t_{skip}$)) **continue**;
22. **if** ($\forall k, s_j.p_k \leq nsp.p_k$)
23. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
24. **break**;

Figure 10. Handle the change of moving plan

involved in skyline operator in our assumption in Section 2.1, the size of skyline on static dimensions is $|SK_{ns}| = O((\ln N)^{m-1})$, and the size of skyline on all dimensions is $|SK_{all}| = O((\ln N)^m)$ at any time. Thus the size of changing part is $|SK_{chg}| = |SK_{all}| - |SK_{ns}| = O((\ln N)^m - (\ln N)^{m-1})$ at any time.

Now we consider the worst-case number of events, i.e., failure of certificates, at any time. In our method, any $s_i s_j$ event or ord_{ij} event is determined by an underlying intersection between two adjacent skyline points' distance function curves. They are *external events* because they affect the skyline result we maintain [3]. Therefore, the maximum number of events of these two kinds is $|SK_{all}|_{max}/2$, since we reduce multiplex intersections into simple ones and store only one at a time. In contrast, nsp_{ij} events

are *internal events* because they are used to adjust internal data structure. As we at most keep one nsp_{ij} event for a non-skyline point at any time, the worst case is that every non-skyline point is involved in such an event, which means the number of nsp_{ij} events is not more than $N - |SK_{all}|_{max}$. By summing up all events, the total number of events in the worst case is $N - |SK_{all}|_{max}/2$. Hence, the ratio of total events to external events is $2N/|SK_{all}|_{max} - 1$. In the worst case where $|SK_{all}|_{max}$ is 1, the upper bound of this ratio is $2N - 1$ which is linear with the number of all points involved. This worst case ratio verifies that our KDS is efficient.

As we store datasets in hard-disk, our method needs to do IO when accessing data points. The main IO cost is incurred by createEvents, which accesses all non-skyline points between the circles of two adjacent skyline points in L_{sp}. This access can be regarded as a special region query over the dataset indexed by grid file, asking for points between two circles with same center but different radiuses. The IO cost of such a query can be estimated with a simple probabilistic model. Let the data space be a 2D unit space (as we use a 2D grid file to index all data points), and the outer and inner circles have radii R_i and r_i respectively when we create events for the ith skyline in L_{sp}. Then the area of the query circle is $S = \pi(R_i^2 - r_i^2)$, and the query will access $SP = \pi(R_i^2 - r_i^2)P$ grid cells (pages), where P is the total number of grid cells. Next we estimate R_i, the distance from q to the i+1th skyline point in L_{sp}. Suppose we do an incremental kNN search for q, if we have met i+1 permanent skyline points, then we must have met the the i+1th skyline point already. With the assumption of independent distribution, $(i + 1)N/|SK_{ns}|$ points are met before the i+1th permanent skyline point. Then in the 2D unit space, we have $\pi R_i^2 = ((i + 1)N/|SK_{ns}|)/N$, which leads to an upper bound of R_i satisfying $R_i^2 = (i+1)/(\pi|SK_{ns}|)$. For r_i, which is the distance from query point q to the ith skyline point, we use a lower bound $min(\sqrt{i/(\pi|SK_{ns}|)}, i/(\sqrt{N} - 1))$ to approximate it. In this way, we get an upper bound of SP.

Let us compare the time cost of continuous skyline query to that of snapshot skyline queries. Assume $\mathcal{N}$ snapshot queries are triggered within a time period $[t_1, t_2]$, and the cost of each is C_i. Then the total and average cost of that method are $\sum_{i=1}^{\mathcal{N}} C_i$ and $\sum_{i=1}^{\mathcal{N}} C_i/\mathcal{N}$ respectively. More snapshot queries incur higher total processing cost, while each single snapshot query's cost is expected to vary little from the

average cost $\mathcal{C}$ because of the static processing fashion. For the same time period, our method computes the initial skyline and events at time t_1, and then updates the skyline only when some certificate fails before t_2. Suppose the number of certificate failures during $[t_1, t_2]$ is $\mathcal{N}'$ (including the initialization), and the cost of each is C'_i, the total and average cost of our method are $\sum_{i=1}^{\mathcal{N}'} C'_i$ and $\sum_{i=1}^{\mathcal{N}'} C'_i/\mathcal{N}'$ respectively. The number of certificate failures $\mathcal{N}'$ is a constant in a fixed time period, therefore the average cost $\mathcal{C}'$ is determined by the total cost only. It makes little sense to compare the total costs of these two methods. If too many snapshot queries are triggered the total cost will be very high, while few snapshot queries deteriorate the result accuracy. To ensure a fair comparison of average costs, we set $\mathcal{N} = \mathcal{N}'$ in our experiment. In other words, we trigger snapshot queries by assuming when the skyline changes is known, which is gained from our method. The experimental study results in next section show that our method even outperforms the privileged snapshot query method.

Our problem formulation assumes a linear movement model for both query point and data points (if they are moving), which is justified by the fact that linear movement model has so far been the most popular one in the literature of moving objects research [1, 12, 16]. This model itself assumes that moving objects hold their current velocities for a period of time, which is also usually considered as a system parameter in typical indexing structures such as TPR-tree [23] and B^x-tree [11]. In most cases, on the other hand, a user can change the speed but seldom changes it every time stamp while still issuing a continuous query. As long as the velocity keeps for a period of time, our method pays off because it saves much computation cost in the result maintenance for future, and it always reports result changes in time, which renders our method beneficial.

4.4 Possible Extensions

It is true that users may issue continuous skyline queries with constraints in SQL WHERE clauses. Our current solution can be adapted to deal with such constraints with some modifications of the kinetic data structures (the certificate) to tender the WHERE clauses. In brief, we first apply the given constraints to SK_{ns} so that an updated SK'_{ns} are gained for further use. Then, in the use of the kinetic data structures, only those data points satisfying the specified constraints will be considered and processed. Thus, our

method is still effective to support the WHERE clauses.

Our current method is focused on processing single continuous skyline query efficiently, whereas it still provides helpful indications for concurrent continuous skyline queries. $|SK_{ns}|$ obviously is the common part for all concurrent queries, which means computation savings can be achieved with $|SK_{ns}|$. Besides, concurrent queries still can share volatile skyline points in some way. These indicate that with proper adaptations our current method can be used to handle this more complex case.

5 Experimental Evaluation

We conducted our experiments on a desktop PC running on MS Windows XP professional. The PC has a Pentium IV 2.6GHz CPU and 1GB memory. All experiments were coded in ANSI C++. The parameters used in the experiments are listed in Table 3. We used both static datasets and moving datasets. For the former, we explored into the effects of cardinality and non-spatial dimensionality on the performance. For the latter, we investigated into the effect of points speed distribution and moving plan update.

Table 3. Parameters used in experiments

Parameter	**Setting**
Dataset cardinality	100K, 200K, ..., 1000K
Dimensionality of non-spatial attributes	$2, 3, 4, 5$
Distribution of non-spatial attributes	Independent, Anti-Correlated
Spatial range	10000×10000
Non-spatial attribute range	[0, 10000]
Point speed range	[10, 30]
Speed Zipf factor	0, 0.5, 1.0, 1.5, 2.0
Update interval	30, 60, 90, 120
Update ratio	4%, 6%, 8%, 10%

5.1 Effect of Cardinality

In this set of experiments, we used synthetic datasets of data points with spatial attributes (x and y) and two non-spatial attributes. For each dataset, all data points are distributed randomly within the spatial space domain of $10,000 \times 10,000$, and their non-spatial attribute values range from 1 to 100,000

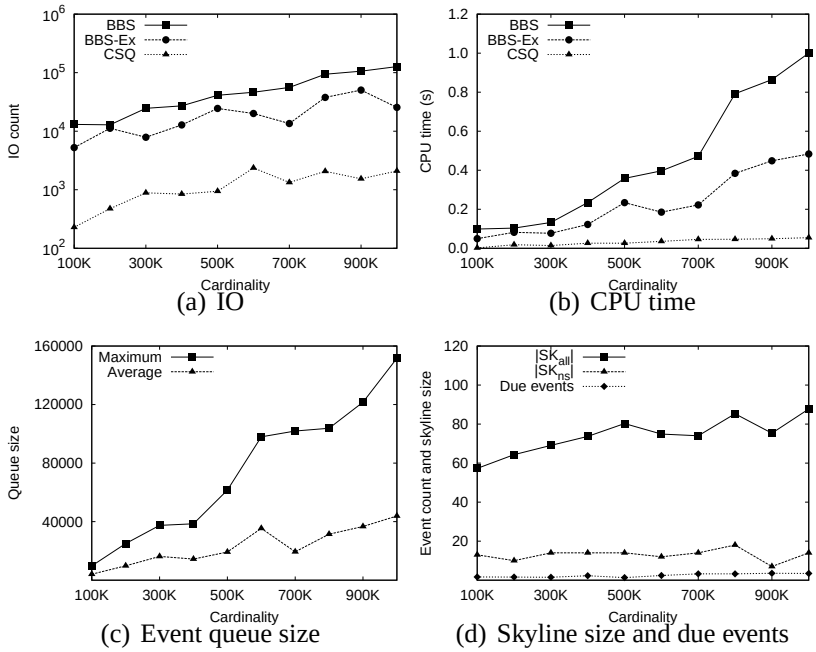
according to either independent or anti-correlated distribution. The cardinality of datasets ranges from 100K to 1M. For each set of data we executed 100 continuous queries moving in random directions. For each query, we randomly generated a point within the data space as the starting position of the moving query point. The speed of each moving query point is also randomly determined and ranges from 10 to 30. Each query ends as soon as the query point moves out of the data space extent. The minimum, maximum and average validity time for all these queries are 1, 475 and 149 units respectively. The experimental results to be reported are the average values on those 100 queries.

Since BBS algorithm is the most efficient method for computing skyline in static settings (both dataset and query point are static) [18], we adapted it for comparison in our experiments. At each time instance, the BBS algorithm is invoked to re-compute the skyline in terms of the query point's new position. Besides, we extended BBS algorithm to exploit the pre-computed static partial skyline points SK_{ns} for pruning, i.e., SK_{ns} is used in every call of BBS algorithm to prune more unqualified tree nodes and data points. In the result reports that follows, we use "BBS-Ex" to denote this method, in contrast to the pure BBS method. It is worth noting that both BBS based methods cannot correctly tell when the skyline changes as our method does.

The comparison was carried out on a fair basis. The same set of randomly generated queries are used by all methods on the same series of datasets. Processing costs, IO count and CPU time, in all methods are amortized over the same number of time units when the skyline changes. For both kinds of indices, R*-tree and grid file, we set the data page size to 1K bytes.

5.1.1 Datasets of Independent Non-spatial Attribute Values

Figure 11(a) shows that as cardinality increases the logarithm of IO count of our maintenance method grows steadily, and nearly 2 orders of magnitude less than that of BBS. Figure 11(b) shows that as cardinality increases the CPU time cost of our maintenance solution grows steadily, in a rate much less than that of BBS. At each time instance, our maintenance solution does not need to search the whole dataset again to re-compute the skyline from scratch, instead it mainly involves event processing which consists less computation of distance and comparison of attribute values than BBS based methods,



(a) IO (b) CPU time

(c) Event queue size (d) Skyline size and due events

Figure 11. Effect of cardinality of independent datasets

which do a totally new search via R*-tree. This processing behavior difference leads to the difference on processing costs. The improvement gained by BBS-Ex compared to pure BBS indicates that SK_{ns} does help pruning, nevertheless BBS-Ex cannot tell the skyline changes either.

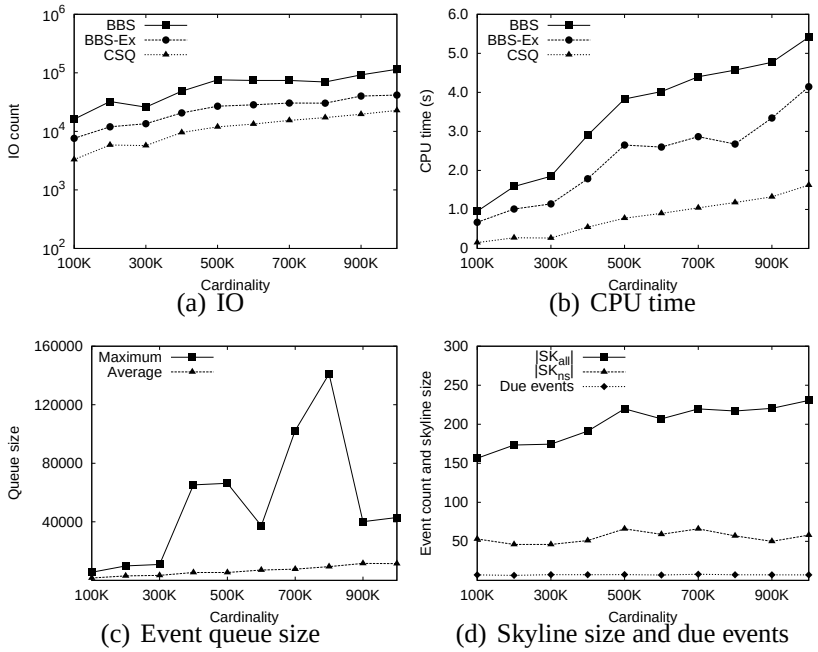
Figure 11(c) shows the effect of cardinality on event queue size at any time unit. The maximum size is gained throughout all 100 queries. It can be seen that the queue event size increases as the cardinality increases, the average queue size is much smaller compared to the maximum size, and it does not exceed 6% of the cardinality.

Figure 11(d) shows the effect of cardinality on skyline size and the number of events being processed at any time unit. It can be seen that complete skyline size roughly increases as cardinality increases, but the average number of due events at any time unit of skyline change never exceeds 4, which indicates the efficiency of our maintenance strategy.

By comparing Figure 11(c) and 11(d) we can see that some events are not processed before the query ends. In a real application, we can take advantage of this observation to further reduce the queue size. The lifetime of a query can be estimated in a specific scenario, e.g., in 2 hours or this afternoon, and any event whose due time later than it will be prevented from being enqueued.

5.1.2 Datasets of Anti-Correlated Non-spatial Attribute Values

We also carried out experiments on datasets whose two non-spatial attributes are anti-correlated. We used the method in [6] to generate such datasets. Figure 12 shows our continuous skyline query processing still outperforms both BBS based methods. The higher cost than that on uniform datasets is attributed to the increase of skyline size of anti-correlated datasets. The anti-correlation between non-spatial attributes also makes the events number increases less unsteadily, as the dominance relationship of data points is more irregular compared to the independent datasets.



(a) IO (b) CPU time

(c) Event queue size (d) Skyline size and due events

Figure 12. Effect of cardinality of anti-correlated datasets

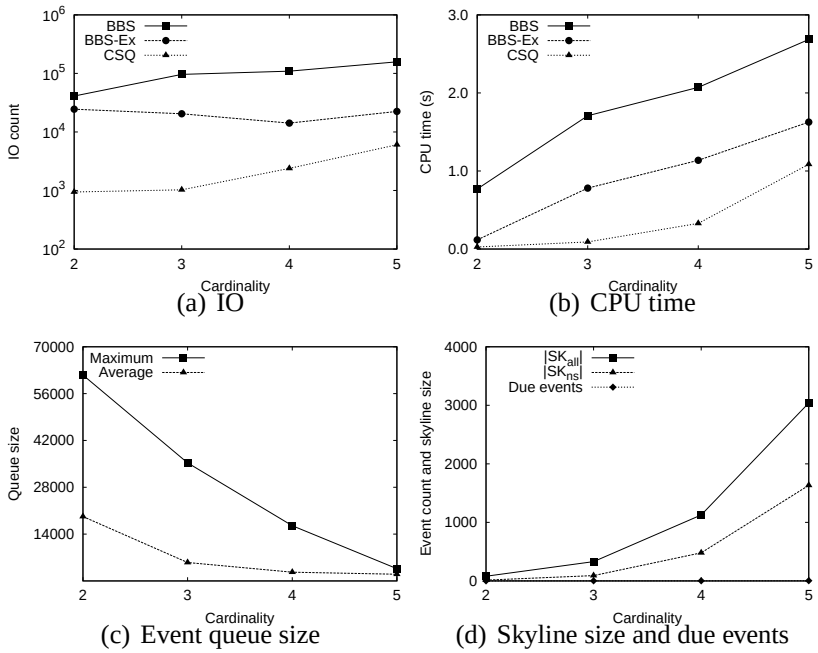
5.2 Effect of Non-spatial Dimensionality

In this set of experiments, we used datasets of 500K points with non-spatial dimensionality ranging from two to five to evaluate the effect of non-spatial dimensionality on our solution. Values on those non-spatial dimensions are of independent distribution. Other settings are the same as in Section 5.1. Datasets with anti-correlated non-spatial values incur similar performance trends, except that every single cost is higher than its counterpart on the independent datasets. Hence we omit those figures here. Figure 13(a)

and 13(b) show the IO and CPU cost respectively. Again our maintenance method outperforms the BBS based methods, and BBS-Ex is better than pure BBS.

Figure 13(c) shows that the event queue size decreases as the non-spatial dimensionality increases. The probability that one volatile skyline point will be dominated by others is lower when more dimensions are involved, because all dimensions are independent in our dataset. This reduces the number of events.

Figure 13(d) shows the effect of non-spatial dimensionality on skyline size and the number of events being processed at any time unit. It can be seen that both static partial skyline and complete skyline size increases as non-spatial dimensionality increases, but the average number of due events at any time unit is still drastically smaller. This indicates that our continuous query processing method still works efficiently.



(a) IO (b) CPU time

(c) Event queue size (d) Skyline size and due events

Figure 13. Effect of non-spatial dimensionality

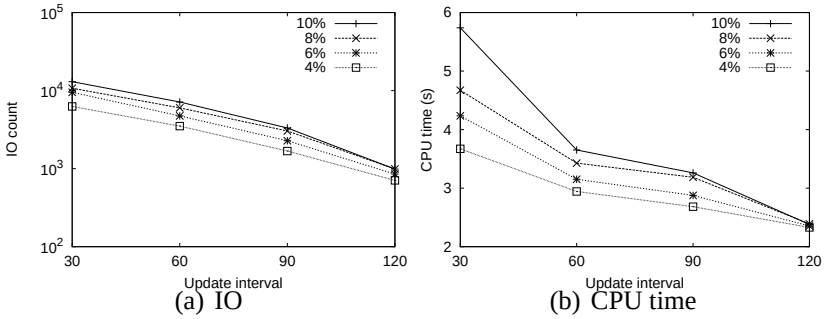
5.3 Effect of Movement Update

In this set of experiments, we used the dataset of 500K data points with spatial attributes (x and y) and two static non-spatial attributes. Every point in each dataset moves within the 2D extent with a speed

ranging from 10 to 30. The hash mechanism is based on the same grid file used for static datasets, with each cell as a bucket containing moving data points. Periodically, a number of moving data points send in update requests. Queries are picked up in the same way as in Section 5.1.

In this set of experiments, the initial speeds of all 500K points were uniformly distributed in the range of 10 to 30. We mainly explore into two aspects of moving data points update: update interval length and the ratio of points requesting update. We varied the update interval length from 30 to 120 time units and update ratio from 4% to 10%.

Figure 14(a) shows the IO count decreases as the update interval increases, and higher ratio of moving data update incurs more IO counts. Longer update interval reduces the amortized update cost which involves changing tuple and recomputing events, and weakens the effects of different update ratios. While higher update ratio increases update cost at every update time. The similar trend is seen for the CPU time shown in Figure 14(b).



(a) IO (b) CPU time

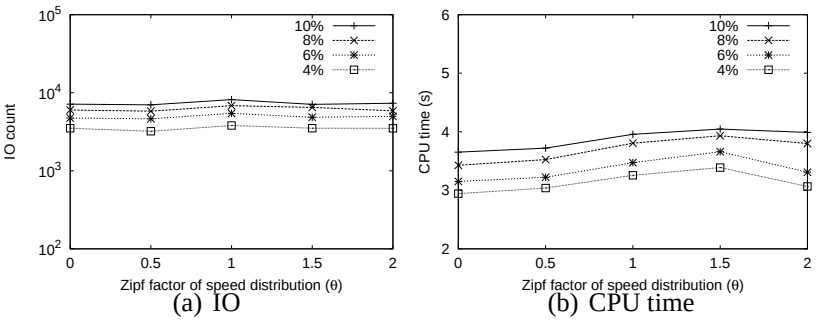
Figure 14. Effect of update

5.4 Effect of Speed Distribution

In this set of experiments, we fixed the moving data points update interval to 60, varied the update ratio from 4% to 10% to see the effect of initial speed distributions. The Zipf factor θ of speed distribution varies from 0, which is a uniform distribution, to 2, which is a skewed distribution where 80% data points move slowly and the 20% move fast. Other settings are the same as in Section 5.3.

Figure 15(a) shows that the IO cost of the proposed method is not too sensitive to skewness on speed. In Figure 15(b), CPU time increases slowly as θ increases from 0 to 1.5, and then decreases when θ

increases from 1.5 to 2. For the same θ, a higher ratio of mobility data set incurs a higher processing cost. The experiments show that our method performs well for the different distributions of moving speed.



(a) IO (b) CPU time

Figure 15. Effect of speed distribution

6 Conclusion

In this paper, we have addressed the problem of continuous skyline query processing. The method, using the kinetic data structure, is based on the analysis that exploits the spatiotemporal coherence of the problem. Our solution does not need to compute the skyline from scratch at every time instance. Instead, the possible change from one time to another is predicted and processed accordingly, thus making the skyline query result updated and available continuously. The experimental studies conducted using different datasets and parameters demonstrate that the proposed method is robust and efficient. To the best of our knowledge, this is the first work on skyline queries in the moving context.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *PODS*, pages 175–186, 2000.

[2] W.-T. Balke, U. Guentzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[3] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *ACM SODA*, pages 747–756, 1997.

[4] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002.

[5] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of ACM*, 25(4):536–543, 1978.

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

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

[8] D. Hearn and M. P. Baker. *Computer Graphics C Version*. Prentice-Hall International, Inc., New Jersey, 1997.

[9] G. Hjaltason and H. Samet. Distance browsing in spatial database. *ACM TODS*, 24(2):265–318, 1999.

[10] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.

[11] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.

[12] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *PODS*, pages 261–272, 1999.

[13] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[14] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4):469–476, 1975.

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

[16] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *PODS*, pages 188–198, 2002.

[17] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, 1984.

[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 30(1):41–82, 2005.

[20] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[21] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[22] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[23] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[24] Z. Song and N. Roussopoulos. Hashing moving objects. In *MDM*, pages 161–172, 2001.

[25] R. E. Steuer. *Multiple criteria optimization*. Wiley, New York, 1986.

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

[27] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, pages 334–345, 2002.

[28] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *SSDBM*, pages 317–326, 2004.

[29] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD Conference*, pages 443–454, 2003.