Continuous Skyline Queries for Moving Objects

ABSTRACT

The literature on the skyline algorithms so far mainly deal with queries for static query points over static datasets. With the increasing number of mobile service applications and users, the need for continuous skyline query processing has become more pressing. The continuous skyline operator involves not only static but also dynamic dimensions. In this paper, we examine the spatio-temporal 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 into the connection between data points' spatial positions and their dominance relationship, which provides an indication on where to find changes of skyline and how to update the skyline continuously. Based on the analysis, we propose a kinetic-based data structure and an efficient skyline query processing algorithm. We analyze the space and time costs of the proposed method and conduct an extensive experiment to evaluate the proposal. To the best of our knowledge, this is the first work on continuous skyline query processing.

1. INTRODUCTION

With rapid advances in miniaturization of electronics, wireless communication and positioning technologies, the acquisition and transmission of spatio-temporal data using mobile devices is becoming pervasive. This fuels the demand for location-based services (LBS)[12, 3, 16, 15]. Skyline query retrieves from a given dataset a subset of interesting points that are not dominated by any other point [4]. Skyline query is 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. The skyline query result is based on the current location of the user, which changes continuously when the user moves.

Existing work on skyline queries assumes 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, there are a set of hotels and for each hotel, we have its distance from the beach (*x axis*) and its price (*y axis*). The interesting hotels are all the points not worse than any other point in both distance from the beach and the price. Hotels 2, 4 and 6 are interesting and can be derived by the skyline query, for their distances to the beach and prices are preferable to any other hotels. Note that a point of the minimum value in any dimension is a skyline point – hotels 2 and 6 for example.

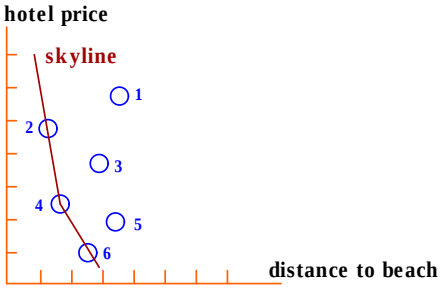
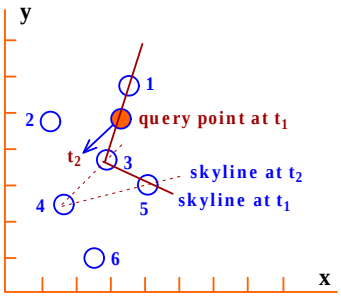


Figure 1: An example of skyline in static scenario

In the above query, the skyline is obtained with respect to a static query point, and in this case, it is the origin of the both axis. Now, let us change the example to the scenario of a tourist walking about to choose a restaurant for dinner. For ease of illustration, we again only consider two factors, namely the distance to the restaurant and the average price of the food. Different from the previous example, the distance from the tourist to a restaurant is not fixed since the tourist is a moving object. Figure 2 shows the changes on the skyline due to the movement. In the figure, positions of the restaurants are drawn in the X-Y plane, while the table shows their prices. A tourist as the query point moves as the arrow indicates from time t_1 to t_2. The skyline, i.e. interesting restaurants, changes with respect to the tourist's position. Skylines at different times are indicated by different line chains. Such problem is common in moving databases [3, 9, 7], and the lack of research in this area motivated our work presented in this paper.

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 spatio-temporal coherence. First, we distinguish the data points that are permanently in the skyline and use them to derive a search bound to constrain the contin-

.



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

Figure 2: An example of skyline in mobile environment

uous skyline query processing. Second, we investigate the connection between data points' spatial locations and their dominance relationship, which provides indication of where to find changes of skyline and update it. Third, to efficiently support processing continuous skyline queries, we propose a kinetic-based data structure and propose associated efficient query processing algorithm.

The paper presents the space and time cost analysis of the proposed method. It also reports on an extensive experimental study, which includes a comparison with an existing method adopted for the application. The results show that the proposed method is efficient with respect to storage space and continuous skyline queries. To the best of our knowledge, this is the first work on continuous skyline queries in mobile environment.

The rest of this paper is organized as follows. In Section 1, we present the preliminaries including our problem statement and a brief review of related work. In Section 3, we carry out a detailed analysis on the problem. In Section 4, we propose our solution which continuously maintains the skyline for moving query points through efficient update. Experimental results are presented in Section 5. Finally, we conclude in Section 6.

2. PRELIMINARIES

2.1 Problem Statement

In LBS, most of the queries are continuous queries [15]. Unlike snapshot queries that are evaluated only once, continuous queries require continuous evaluation as the query results become invalid 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 spatio-temporal coherence of the movement, the skyline will change in a smooth manner. Notwithstanding, updating the skyline of the previous moment will be more efficient than conducting a snapshot query at each moment.

We limit the data and query points moving in a 2D (2-dimensional) space for an intuitive illustration. The 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 X and Y dimension, while p_{ij}'s $(j = 1, ..., m)$ are the static non-spatio attributes. They 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 are moving in 2D plane. Each of them moves in velocity (v_{qx}, v_{qy}), starting from position (x_q, y_q). They pose continuous skyline queries during the movement, which involve both distance and all other static dimensions. Such queries are dynamic due to change in spatial variables. In our solution, we only want to compute the skyline for (x_q, y_q) at the start time 0. Subsequently, continuously query processing is conducted for each user by updating instead of computing a new skyline from scratch each time.

Without loss of generality, we restrict our discussion in what follows to the MIN skyline annotation [4], in which smaller values of distance or attribute p_{ij} are preferred in comparison to determine dominance between two points.

2.2 Related Work

2.2.1 Algorithms for Static Skyline Query

Borzonyi, Kossmann and Stocker [4] for the first time introduced the skyline operator into database systems by extending the SQL SELECT statement with an optional SKYLINE OF clause. Two processing algorithms *Block Nested Loop* (BNL) and *Divide-and-Conquer* (D&C) were proposed. Basically, BNL sequentially scans the whole data file and compares each new point to all skyline candidates kept in memory. Only those points not dominated by others are kept as skyline candidates. The D&C approach divides the whole dataset into several partitions such that each can fit in memory. A local skyline is computed for each partition, and the final skyline is obtained by correctly merging these local skylines.

In [14], Tan, Eng and Ooi proposed two progressive processing algorithms: *Bitmap* and *Index*. In the Bitmap approach, every dimension value of a point *pt* is represented by a few bits, and *pt* itself is transformed into a bit vector by concatenating those bits of all dimensions. In a top-down fashion, vectors of all points form a bit matrix. Whether a given point is in the skyline can be answered without referring to other points, by retrieving some specific bit columns from the matrix and applying bit-wise *and* operation on them. On the other hand, the Index approach uses a novel transformation to map each point into a single dimensional space such that they can be indexed by a B^+-tree. The skyline computation is conducted in several batches, whose number equals that of all distinct values on all dimension in the whole dataset. Within each batch, relevant points are fetched from each partition with the aid of the B^+-tree, and over all those points a local skyline is computed. After each batch the local skyline is merged into the final skyline with unqualified points correctly excluded.

Kossmann, Ramsak and Rost [8] proposed a *Nearest Neighbor* (NN) method to process skyline queries progressively. It first carries out a NN search on the dataset indexed by an R^*-tree, and then inserts the NN point into the skyline. The NN point also determines a region which only contains points dominated by NN and thus can be pruned. The remaining part of the space is partitioned into two parts based on the NN point, and both are inserted into a *to-do* list. Then the algorithm removes a part from the *to-do* list and process it recursively, until the list is empty.

Recently, Papadias, Zhang and Tao [11] proposed a new progressive algorithm named *Branch−and−Bound Skyline*

(BBS) based on the best-first nearest neighbor (BF-NN) algorithm [6]. It first enqueues all the entries of the R*-tree root into a heap sorted on their *mindist*'s to the query point. Then the entry e on the heap top will be dequeued and will be discarded if it is dominated by some skyline point. Otherwise it is either expanded, and enqueue its sub-entries if it is an intermediate entry, or it is inserted into the skyline if it is a point. However, unlike BF-NN, BBS uses L_1 norm to compute *mindist*, and only enqueues those entries that are not dominated by any skyline point. In a slightly different context, Balke, Guntzer and Zheng [1] addressed the skyline operation over web databases where different dimensions are stored in different data sites. Their algorithm first retrieves values in every dimension from remote data sites using sorted access in round-robin on all dimensions, until all dimension values of an object, called the terminating object, have been retrieved. Then all non-skyline objects will be filtered out from all those objects with at least one dimension value retrieved.

2.2.2 KDS and Continuous Queries for Moving Objects

Basch, Guibas and Hershberger [2] proposed a conceptual framework for kinetic data structures (KDS) as a means to maintain continuously evolving attributes of mobile data. The KDS keeps the desired relationship between data by storing all those data in some structures specific to the relationship. The contents in KDS do not change unless the relationship between some data points has been changed. In this way, the data retrieval result based on the desired relationship can be maintained when the data points move continuously.

KDS and its underlying ideas have inspired some database query processing techniques that utilize events to maintain the query result. Mokhtar, Su and Ibarra [9] proposed an event-driven approach to maintain the result of k-NN query on moving objects while time elapses. Their approach starts with a list of all moving objects that are are sorted by their current distance to the query point. Then events indicating when a moving object will change position in the list with its neighbor are computed based on the movement parameters of moving objects. All those events are pushed into a priority queue, which gives priority to events that will happen earlier. The problem of maintaining k-NN query result is transformed into the problem of maintaining the list of moving objects. As time progresses, events are processed and the order of moving objects are maintained, thus making k-NN query result always available in the object list.

Instead of keeping all moving objects in ascending order of distance to query point, Iwerks, Samet and Smith [7] present another event-driven method to maintain continuous k-NN queries on moving objects. Based on the fact that window queries are cheaper to maintain on moving objects than k-NN queries, the authors proposed the Continuous Windowing (CW) k-NN algorithm. The CW k-NN algorithm first gets all those objects within a specific distance d around the query point. And if at least k objects are found, all the final k nearest neighbors must be among these objects and only they need to be checked. Otherwise, the search will be extended outwards with the distance d adjusted. Here events indicating when and which objects will move into the distance d around the query point are computed first, and processed gradually to maintain the query result during the period of query life time.

2.3 Time Parameterized Distance Function

Since the distance between moving query point and data point is involved in the skyline operator in our problem, we therefore present some background about the changing distance in the moving context. 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) and moving with (v_{qx}, v_{qy}), the distance between them can be expressed as a function of time t: $dist(q(t), pt_i(t)) = \sqrt{a \cdot t^2 + b \cdot t + c}$, where a, b and c are constants determined by their starting positions and velocities: $a = (v_{ix} - v_{qx})^2 + (v_{iy} - v_{qy})^2$; $b = 2 \cdot [(x_i - x_q) \cdot (v_{ix} - v_{qx}) + (y_i - y_q) \cdot (v_{iy} - v_{qy})]$; $c = (x_i - x_q)^2 + (y_i - y_q)^2$. For the sake of simplicity, we use function $f_i(t) = a \cdot t^2 + b \cdot t + c$ to denote the square of the distance. When data point pt_i is static, a, b and c are still determined by formulas above with $v_{ix} = v_{iy} = 0$.

2.4 Terminologies

In this subsection, we define the terminologies used in this paper. We use $dist(pt_1, pt_2)$ to represent the Euclidean distance between two points pt_1 and pt_2. 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 $pt_1 \sim pt_2$ that they are incomparable. 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 [2]. In this paper, we use a certificate to ensure the status of a data point 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. An event will trigger a process to update the certificate. The process may result in a change of the skyline.

3. ANALYSIS OF THE CHANGE ON SKYLINE

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

3.1 Skyline on Static Non-spatio Dimensions

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 data points and query points move. This is because they have domineering static non-spatial values, which guarantee that no other objects can dominate them. We denote this type of skyline points as SK_{ns} and the whole set of skyline points as SK_{all}. We call SK_{ns} *static partial skyline*, and SK_{all} *complete skyline*. It is obvious that SK_{ns} is always on the complete skyline as time elapses because its underlying advantage on static non-spatio values does not change as data points and query point move.

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 whole 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}, and thus in query processing efforts can be concentrated on the changing part only, i.e., $SK_{all} - SK_{ns}$. We name the changing part SK_{chg}, and call those points in it *volatile skyline points*. In continuous skyline query processing, only SK_{chg} is needed to be kept tracked for each query. In this manner, we can reduce the overall processing cost.

2. This discrimination reduces the size of data to be sent to clients. Every time when change happens, only SK_{chg} as query result is needed to be transferred, which is beneficial to real mobile applications where clients and servers are usually connected via limited bandwidth.

3. Static partial skyline SK_{ns} also provides indication of search bound for processing a continuous skyline query, as we will discuss next. We use $pt_1 \preceq pt_2$ to represent that pt_1 dominates pt_2 for all m static non-spatio dimensions.

3.2 Search Bound

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 anyone in SK_{ns}. More specifically, it must have advantage in distance to query point since it is dominated with respect to all static dimensions by at least one point in SK_{ns}. This leads to the following Lemma 3.2.1.

LEMMA 3.2.1. *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 disadvantage in both spatial and non-spatio dimensions, pt is dominated by sp at time t so that it is not in the complete skyline. □

Lemma 3.2.1 indicates a search bound for skyline on all dimensions. This can be used to filter out part of unqualified points in query processing: those ones that are farther away than all points in SK_{ns} cannot be in the skyline.

3.3 Change of the Skyline

When the query point q and data points move, their distance relationships may change. It 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 2.3 that the square of distance from each point to 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.

Intuitively, a skyline point s_i in SK_{chg} before time t_x may leave the skyline after that moment. On the other hand, a non-skyline point nsp at time t_x may enter the skyline and become part of SK_{chg} after that moment. 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 must stop dominating it.

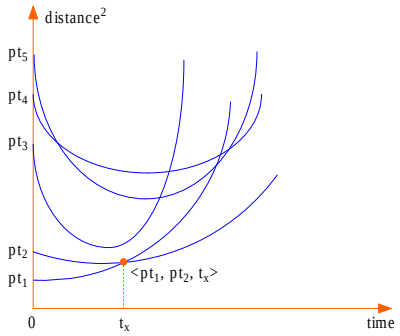


Figure 3: An example of distance functions

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 query than point pt_1 but before that moment the inequality relationship is on the contrary. ¿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. Because before and after the intersection, the only change of comparison is $dist(q, pt_1) < dist(q, pt_2)$ to $dist(q, pt_2) < dist(q, pt_1)$. Apparently if no intersections happen, skyline does not change at all, because the inequality relationship between all points' distances to query point remains unchanged.

Nevertheless, not every intersection will necessarily cause the skyline to change. We need to investigate the conditions in which an intersection $<pt_1, pt_2, t_x>$ causes pt_1 and pt_2 to enter or leave the skyline, since the intersection does not affect any other points not involved in it. Whether an intersection $<pt_1, pt_2, t_x>$ causes skyline to 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 the possibilities.

LEMMA 3.3.1. *An intersection $<pt_1, pt_2, t_x>$ 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. Both pt_1 and pt_2 will still be in the skyline after t_x because they are permanent skyline points.

2. Obviously pt_1 does not leave the skyline. Assume 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 not enter the skyline after t_x.

□

LEMMA 3.3.2. *An intersection $<pt_1, pt_2, t_x>$ 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 $pt_2 \in SK_{ns}$
(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 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 still dominate it. Consequently, pt_2 will enter the skyline after t_x.

2. Obviously pt_2 will not leave skyline after t_x. But if $\forall k, pt_2.p_k \leq pt_1.p_k$ holds, pt_2 will become dominating pt_1 and causes 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 become dominating pt_1 and causes 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 3.3.1, pt_2 itself will not leave the skyline since no other points become dominating it after t_x.

4. Due to the same reasoning as in (1), pt_2 may enter the skyline after t_x. We postpone to later the discussion on whether pt_2 will become dominating pt_1 and make it leave skyline.

□

Table 1 lists all possibilities attached to an intersection. For the possibility that pt_1 comes from SK_{chg} and pt_2 from $\overline{SK_{all}}$, an interesting issue is whether pt_2 can dominate pt_1 after time t_x.

LEMMA 3.3.3. *For an intersection $<pt_1, pt_2, t_x>$ 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-spatio parameter values of pt_1 and pt_2 are not same for every dimensions.*

Table 1: Intersections and possible skyline changes

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

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_1, 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} distance from pt_3 to 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 just be t_x. Therefore three points must have their distance function curves 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 we assume that their static non-spatio parameter values are not same for every dimensions. It leads to pt_2 can not dominate pt_1 after t_k because $pt_1.p_k < pt_2.p_k$ still holds. □

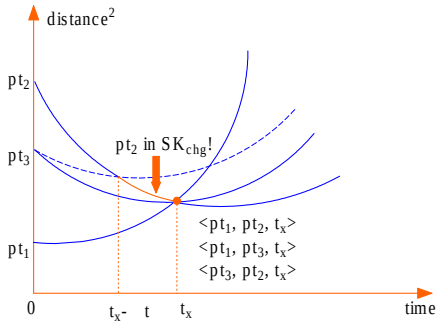


Figure 4: Multiplex intersection example

Figure 4 shows such a scenario, 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.

Let us still refer to the example in Figure 4. According to the strategy above the intersection $<pt_1, pt_2, t_x>$ 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 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 enter SK_{chg} because pt_3 is the only one that used to dominate pt_2 and now it stops dominating due to its distance to 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 happens seldom because it requires that all those points involved to be on the same circle centered at the query point. This situation only happens to the minority data points in usual distributions, 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, $nsp \in SK_{chg}$ if $\forall s_j \in SK_{all}$, we do not have $s_j \prec nsp$.*

Case 1 determines a skyline change, whereas suggests a possibility of change which requires further checking. For s_i and s_j in Case 1, the relationship of their distances to the query point can be described formally in Corollary 3.3.1. Similarly for nsp and s_i in Case 2, we have Corollary 3.3.2.

COROLLARY 3.3.1. *For Case 1, $\exists\ \varepsilon > 0$, $dist(q, s_j) > dist(q, s_i)$ holds for any time $t \in (t-\varepsilon, t)$. But for $t \in (t, t+\varepsilon)$, $dist(q, s_j) < dist(q, s_i)$ holds.*

COROLLARY 3.3.2. *For Case 2, $\exists\ \varepsilon > 0$, $dist(q, nsp) > dist(q, s_i)$ holds for any time $t \in (t-\varepsilon, t)$. But for $t \in (t, t+\varepsilon)$, $dist(q, nsp) < dist(q, s_i)$ holds.*

Corollary 3.3.1 indicates where to find a potential dominating s_j for a volatile skyline point s_i. For a period of time before the change, such s_j must be out of the circle determined by query point q and s_i. We use $Cir(q, s_i)$ to denote the circle whose center is q and radius is the distance from q to s_i, i.e., $dist(q, s_i)$. Corollary 3.3.2 indicates that for the skyline point s_i involved 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 of skyline.

3.4 Continuous Skyline Query Processing

Based on the above observations, we now address the issues of the 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 3.3. This method produces many "false hits" which actually do not cause skyline changes as we have shown in Table 1.

To reduce storing and processing of unnecessary possibilities, we compute and store intersections in an evolving way taking into account the current skyline, rather than compute all intersections at the very beginning. 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 updated skyline. By looking into near future, we ensure that the skyline query result keeps updated, and more information will be obtained later for updating the skyline in farther future.

Given the current skyline points, we only keep those intersections with the likelihood to change the skyline in the future according to Table 1. 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 the sorted order.

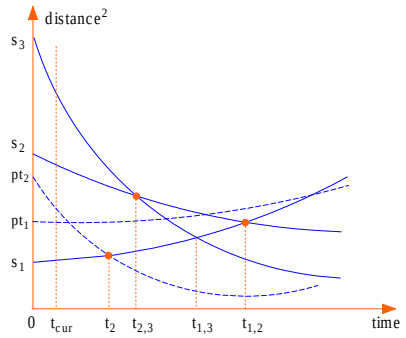


Figure 5: An example of evolving intersections

In Figure 5, for example, s_1, s_2 and s_3 are three adjacent skyline points between time $[0, t_{cur}]$, while pt_1 and pt_2 are two non-skyline points between s_1 and s_2. At an evolving step of time t_{cur}, only those intersections in dot will be computed and stored for future processing. Next evolving step is at time t_2 and pt_2 will enter the skyline if s_1 is the only skyline point dominating it. The further next evolving step is at time $t_{2,3}$. If s_2 is a volatile skyline point and $s_3 \preceq s_2$, intersection $<s_2, s_3, t_{2,3}>$ will cause s_2 to leave the skyline from time $t_{2,3}$ onwards. Otherwise it means an exchange of positions only. Either case causes the order of skyline points to change. For future maintenance, intersection $<s_1, s_3, t_{1,3}>$ will replace $<s_1, s_2, t_{1,2}>$ since now s_1 and s_3 are adjacent. Thus, at time t_{cur} only three intersections will be computed and stored for later processing, less than half of the total intersections in the figure.

4. DATA STRUCTURE AND ALGORITHMS

The analysis presented in Section 3 provides us a basis for adapting the kinetic data structure for supporting continuous skyline query processing. The query results need to

be modified only when some existent skyline points leave or non-skyline points enter the skyline. What we need to do is to decide when such cases happen and then what actions to take. Certificates (see subsection 2.4) can be defined to certify no such instance occur within a period of time, thus ensuring the skyline result does not change within that period of time. If any certificate fails, an update on the skyline is required.

In our method, each skyline point $s_i \in SK_{chg}$ has a certificate corresponding to Case 1 in Section 3.3, since only volatile skyline points may be dominated by other skyline points and leave the skyline. That certificate will fail when another skyline point becomes dominating s_i, and then we must remove s_i from the skyline. Corresponding to Case 2 we also have certificates, though it is not so straightforward as Case 1. At time t, non-skyline point nsp might be dominated by more than one skyline point. For nsp to enter the skyline it must get closer to query point q than all other points dominating it. It is difficult to keep track of all dominators for nsp and predict when nsp will get closer to q than any of them, because those dominators themselves might change as well. We choose another strategy. The earliest time t when nsp gets closer than any of those dominators is recorded. And when that time t comes, conditions in Case 2 will be checked to determined whether nsp will really enter the skyline.

4.1 Implementation Details

Based on the analysis in Section 3, it is obvious that the static partial skyline not only constitutes the unchanging part of the skyline but also provides a search bound for other points that can be in the complete skyline. Besides, the distance from each skyline point (permanent or volatile) provides indication of future changes. Thus, it is beneficial to keep all skyline points in an ordered structure such that the observations from the analysis can be applied efficiently in query processing.

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 we keep an entry of form $(flag, tuple\_id, a, b, c, t_v, t_{skip})$. $flag$ is a boolean variable indicating if the skyline point is in SK_{ns}. $tuple\_id$ is the tuple identifier which can be used to access the record. a, b, c are coefficients of the distance function between this point and query point q, introduced in Section 2.3. t_v is only available to each changing skyline point, indicating its validity time. t_{skip} is the time when the skyline point will exchange its position with its successor in L_{sp}.

Besides L_{sp} for all skyline points, 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. Next we define the certificates and events to be used in our solution.

4.2 Certificates and Events

We define three kinds certificates, 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 distance change resulting from the movement of the query point. Each event is in the form of $(type, time, self, peer)$, where

Table 2: Certificates and events

Cert.	Objective	Event
$s_i s_j$	$\forall s_i \in SK_{chg}, s_j \in SK_{all}$, 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} $\rightarrow dist(q, s_i) < dist(q, s_j)$	$self = s_i$ $peer = s_j$

$type$ 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 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 (in Section 4.4). Here $self$ points to s_i's entry in L_{sp}, whereas $peer$ is the tuple identifier of data point nsp.

Besides, we define another kind of certificate ord_{ij}, which ensures for an existent skyline point s_i that its successor s_j in ordered list 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 3.3, it also involves a position exchange in L_{sp}, i.e. just before s_j dominating it 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 $s_i s_j$ event. If s_i does have an event of $s_i s_j$ type, the event's $time$ value is exactly s_i's validity time t_v. If s_i has no $s_i s_j$ 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 processed. For each current skyline point s_i, we only need to consider the earliest event (s_i becomes invalid or exchanges with its successor), and postpone the further ones after processing the earliest one. This helps preclude unnecessary events to be processed, and thus reducing the processing time.

In our proposed method, the L_{sp} initially contains the current skyline points, and the priority queue contains events that will happen in the nearest future time to cause the

skyline to change. As time elapses, every due event is dequeued from the priority queue and processed based on its *type*. While processing due events and updating the skyline result accordingly, the procedure also creates new events that will happen in later future and that will cause skyline changes. Thus the event queue evolves with due events being dequeued and new events being enqueued, providing correct information for maintaining the skyline by looking into near future. At any time t after all due events are processed, L_{sp} is the correct skyline with respect to the moving query point q's current position.

4.3 Initialization

We partition the dataset D into two groups: the set of static partial skyline points SK_{ns}, and the rest $D' = D - K_{ns}$. We pre-compute SK_{ns} and store it as a system constant. Thus, the initialization produces two outcomes: the changing part, i.e. SK_{chg}, of complete skyline with respect to the starting position of the query point, and the earliest events that will be used later to update the skyline.

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
10. **for each** s_i from $L_{sp}.last.prev$ to $L_{sp}.first$
11. CreateEvents(s_i, q);
// handle last skyline point specially
12. $t_{next} = Q_e.first.time$;
13. $s_{last} = L_{sp}.last$;
14. $t_{nsp} = \infty$; $peer = null$;
15. $C = Cir(q(t_{next}), s_{last}) - Cir(q, s_{last})$
16. **for each** point nsp in C
17. **for each** s_j from s_{last} to $L_{sp}.first$
18. t = time nsp will get closer to q than s_j;
19. **if** $((t \geq s_j.t_v)$ **or** $(t \geq s_j.t_{skip}))$ **continue**;
20. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
21. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
22. **break**;

Figure 6: Initialization framework

To compute SK_{chg} over static dataset for the query point's starting position, in order to use the search bound determined by static partial skyline 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 [10]. In our solution for the static dataset, we use the simplest uniform 2D grid file that divides 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 [13] 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 4.5. Except for the difference on underlying indexing schemas, the initializations for static and moving datasets share the same framework and events creation algorithms.

The initialization framework based on the grid file 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 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. During the search, those cells farther than d_{bnd} are pruned. In line 7 of Figure 6, $mindist(q, cell_i)$ is computed. After all cells are searched or pruned, the algorithm CreateEvents is invoked for each skyline point s_i from outer to inner, to compute all events for all skyline points except for the last one. After the loop, we compute a possible nsp_{ij} event for those points out of the last skyline point s_{last}. That computation 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. The later is represented by $q(t_{next})$. Because only the non-skyline points in that region have chance to get closer to query point than s_{last} and enter the skyline before t_{next}.

Points in a grid cell that is not pruned by the search bound d_{bnd} are sequentially compared to the current skyline points in L_{sp}, which is adjusted with deletion or insertion if necessary.

Algorithm CreateEvents is presented in Figure 7. 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 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 $s_i s_j$ 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 is halted. All events created are enqueued into the event queue.

4.4 Updating the Skyline

In maintaining the skyline, the due events are dequeued and processed according to it type, and new events are computed based on the new position of query point. As in the initialization, the event of points out of the last skyline point is computed in a special way with an estimated search re-

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 $(s_i, s_i.t_v, rep, s_is_j)$ 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_j from s_i to $L_{sp}.first$
15. t = time nsp will get closer to q than s_j;
16. **if** $((t \geq s_j.t_v)$ **or** $(t \geq s_j.t_{skip}))$ **continue**;
17. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
18. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
19. **break**;

Figure 7: Create events

Algorithm updateSkyline(t_{cur})
Input: t_{cur} is the current time
Output:updated L_{sp} and Q_e
1. $s_{last} = L_{sp}.last$;
2. **while** $(Q_e.first.time == t_{cur})$
3. $evt = Q_e.dequeue$;
// call corresponding process for evt
4. Process evt in terms of $evt.type$;
5. **if** $(s_{last} \neq L_{sp}.last)$ **return**;
6. $s_{last} = L_{sp}.last$;
8. $t_{next} = Q_e.first.time$;
9. $C = Cir(q(t_{next}), s_{last}) - Cir(q(t_{cur}), s_{last})$
10. **for each** point nsp in C
11. **for each** s_j from s_{last} to $L_{sp}.first$
12. t = time nsp will get closer to q than s_j;
13. **if** $((t \geq s_j.t_v)$ **or** $(t \geq s_j.t_{skip}))$ **continue**;
14. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
15. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
16. **break**;

Figure 8: Update the skyline

Process s_is_j **event** e
1. $s_i = e.self$; $s_j = e.peer$;
2. Delete s_i from L_{sp};
3. $s_i = s_j.prev$;
4. CreateEvents(s_i, q);

Figure 9: Process s_is_j event

Process nsp_{ij} **event** e
1. $s_i = e.self$; $nsp = e.peer$
2. $dominated$ = FALSE;
// check inner skyline points
3. **for each** s_j in L_{sp} from $s_i.prev$ to first
4. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
5. $dominated$ = TRUE;
6. **break**;
//nsp does not enter the skyline this time
7. **if** ($dominated$)
8. $s_k = s_i.prev$;
9. t = time nsp will get closer to q than s_k;
10. **if** $((t < s_k.t_v)$ **and** $(t < s_k.t_{skip})$
and $(\forall k, s_k.p_k \leq nsp.p_k))$
11. Enqueue (s_k, t, nsp, nsp_{ij}) to Q_e;
12. **else** // nsp enters the skyline
13. Insert nsp into L_{sp} before s_i;
14. $s_j = s_i.prev$;
15. CreateEvents(s_j, q);
16. CreateEvents($s_j.prev$, q);

Figure 10: Process nsp_{ij} event

Process ord_{ij} **event** e
1. $s_i = e.self$; $s_j = e.peer$;
2. Switch s_i and s_j's positions in L_{sp};
3. CreateEvents(s_i, q);
4. CreateEvents(s_j, q);
5. **if** $(s_j.prev \neq null)$
6. CreateEvents($s_j.prev$, q);

Figure 11: Process ord_{ij} event

gion. Algorithm updateSkyline is presented in Figure 8.

The actions to process each kind of events are described respectively in Figures 9 to 11. 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.

4.5 Updating the Moving Plan

If a moving data point mpt_i's moving plan changes, e.g., its velocity and its distance function, will change and intersections with other ones will also be changed as a consequence, which invalidates those events computed based on mpt_i's old distance function curve. Figure 12 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.

To ensure correct distance computation 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. When such an update request comes in, we have to check 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

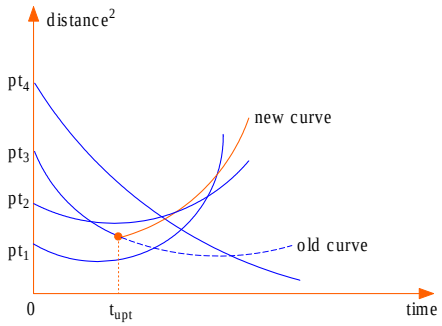


Figure 12: An example of the change of moving plan

need to remove it from the old one and insert it into the new one. If v_x and v_y indicate that mpt_i's speed has been changed, we need to remove old events relevant to mpt_i and compute new ones based on its new speed. This algorithm is presented in Figure 13. 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 12 that right at the time, 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_u executes after updateSkyline(t_u) completes.

4.6 Cost Analysis

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 size of its skyline is $n_{sky} = O((\log N)^{d-1}/(d-1)!)$ as presented in [5]. Since there are m static dimensions involved in skyline operator in our assumption in Section 2.1, the size of skyline on static dimensions is $|SK_{ns}| = O((\log N)^{m-1}/(m-1)!)$, and the size of skyline on all dimensions is $|SK_{all}| = O((\log N)^m/m!)$ at any time. Thus the size of changing part is $|SK_{chg}| = |SK_{all}| - |SK_{ns}| = O((\log N)^{m-1}(\log N - m)/m!)$ at any time. Now we consider the number events at any time. In our method, any s_is_j event or ord_{ij} event is determined by an underlying intersection between two adjacent skyline points' distance function curves. Therefore, the maximum number of events of these two kinds are $|SK_{all}| - 1$, the number of such intersections. For nsp_{ij} events, the worst case is that every non-skyline point is involved in such an event, which means the number of nsp_{ij} events is $N - |SK_{all}|$ at most. By summing up all events, the number of total events in the worst case is $N - 1$.

The IO cost in our method is mainly 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 query can be estimated with a simple probabilistic model. Let the data space be a 2D unit space, and the outer and inner circles have radii r_1 and r_2 respectively. Then the area

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. $\quad Tuple(req.id).cell = cell_2$;
5. $\quad$ remove $req.id$ from $cell_1$ and insert it to $cell_2$;
6. **if** (($req.v_x == Tuple(req.id).v_x$) **and**
   $\quad (req.v_y == Tuple(req.id).v_y)$)
7. $\quad$ **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. $\quad$ **if** ($s_i.tuple\_id == req.id$)
13. $\quad\quad$ Delete all s_i's events;
14. $\quad\quad$ CreatEvents(s_i, q);
15. $\quad\quad$ **return**;
16. $\quad$ Delete all s_i's events with $peer == req.id$;
17. $\quad$ **if** ($dist(q, Tuple(req.id)) \leq dist(q, s_i)$)
18. $\quad\quad$ **break**;
19. $nsp = req.id$;
20. **for each** s_j from s_i to $L_{sp}.first$
21. $\quad t =$ time nsp will get closer to q than s_j;
22. $\quad$ **if** (($t \geq s_j.t_v$) **or** ($t \geq s_j.t_{skip}$)) **continue**;
23. $\quad$ **if** ($\forall k, s_j.p_k \leq nsp.p_k$)
24. $\quad\quad$ Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
25. $\quad\quad$ **break**;

Figure 13: Handle the change of moving plan

of the query circle is $S = \pi \cdot (r_1^2 - r_2^2)$, and the query will access $S \cdot P = \pi \cdot (r_1^2 - r_2^2) \cdot P$ grid cells (pages).

Let us compare the time cost of continuous skyline query to that of using 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 $\frac{\sum_{i=1}^{\mathcal{N}} C_i}{\mathcal{N}}$ respectively. More snapshot queries incur higher total processing cost, while each single snapshot query's cost may 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 $\frac{\sum_{i=1}^{\mathcal{N}'} C_i'}{\mathcal{N}'}$ 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 the next section show that our method even outperforms the privileged snapshot query

method.

5. EXPERIMENTAL EVALUATION

In this section we present the results of our experiments conducted on a desktop PC running on Microsoft Windows XP professional. The PC has a Pentium IV 2.6GHz CPU and 1GB main memory. All experiments were programmed in ANSI C++.

5.1 Experiments on Static Dataset

For the static dataset we mainly explored into the effects of cardinality and non-spatio dimensionality on the performance. We used synthetic datasets of data points with spatial attributes (x and y) and two to five static non-spatio attributes. For each dataset, all data points are distributed independently within the spatial space domain of $10,000 \times 10,000$, and their non-spatio attribute values range independently from 1 to 100,000. 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 experiment results to be reported are the average values of those 100 queries, if not explicitly stated otherwise.

Because currently BBS algorithm is the most efficient one for computing skyline for a static query point [11], we implemented and used it for comparison. At each time instance, the BBS algorithm is invoked to re-compute the skyline in terms of the query point's new position. It is worth noting that BBS can not correctly tell when the skyline changes as our method does.

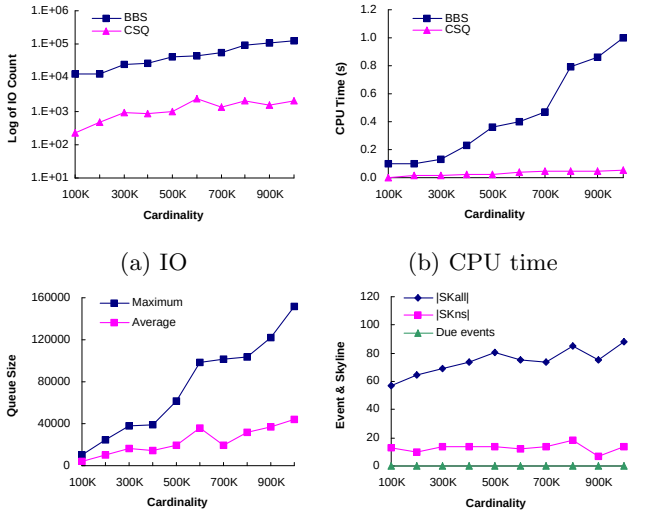
The comparison was carried out on a fair basis. The same set of randomly generated queries are used by both methods on the dataset of the same size. Processing costs, IO count and CPU time, in both methods are amortized over the same number of time units when the skyline changes.

5.1.1 Effect of Cardinality

In this set of experiments, we used datasets of two non-spatio attributes to evaluate the effect of cardinality on our method, by comparing it with the BBS method. For both indices, R*-tree and grid file, we set the data page size to 1K bytes.

Figure 14(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 14(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, which does a totally new search via R*-tree. This processing behavior difference leads to the difference on processing costs.

Figure 14(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



(a) IO (b) CPU time

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

Figure 14: Effect of dataset cardinality

is much smaller compared to the maximum size, and it does not exceed 6% of the cardinality.

Figure 14(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 14(c) and 14(d) we can see that some events may be not processed at all 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.

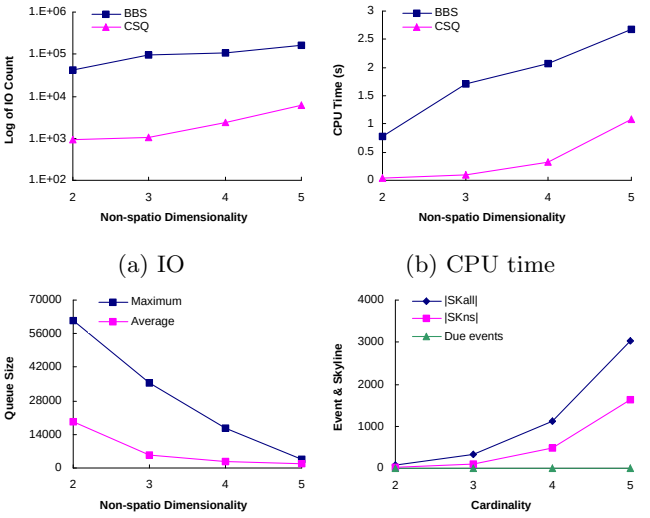
5.1.2 Effect of Non-spatio Dimensionality

In this set of experiments, we used datasets of 500K points with non-spatio dimensionality ranging from two to five to evaluate the effect of non-spatio dimensionality on our solution. The settings are the same as in Section 5.1.1.

Figure 15(a) and 15(b) show the IO and CPU cost respectively. Again our maintenance method outperforms the BBS method. As the non-spatio dimensionality increases the gap of costs keeps steady.

Figure 15(c) shows that the event queue size decreases as the non-spatio 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 may reduce the number of events.

Figure 15(d) shows the effect of non-spatio 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 rapidly as non-spatio dimensionality increases, but the average number of due events at any time unit is drastically much smaller. This indicates that our maintenance strategy still works efficiently.



(a) IO (b) CPU time

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

Figure 15: Effect of non-spatio dimensionality

5.2 Experiments on Moving Dataset

For the moving dataset we mainly explored into the effects of mobility on the performance. We used the dataset of 500K data points with spatial attributes (x and y) and two static non-spatio attributes. Every point in each dataset moves within the 2-dimensional extent with a speed ranging from 10 to 30. Periodically, a number of moving data points send in update requests. Queries are picked up in the same way as on static datasets.

5.2.1 *Effect of Update*

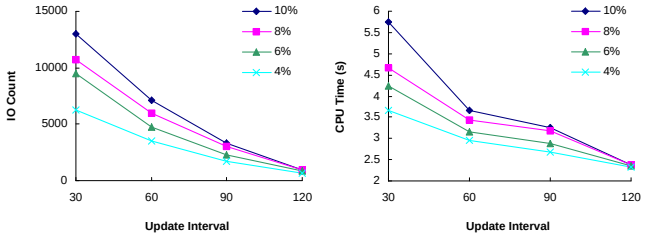
In this set of experiments, the initial speeds of all 500K points were uniformly distributed in the range of 10 to 30. We investigate into two aspects of moving data points update: update interval length and the ratio of points requesting update. We varied the update interval length from 30 to 120 time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

Figure 16(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. While higher update ratio increases update cost at every update time. The similar trend is seen for the CPU time shown in Figure 16(b).

5.2.2 *Effect of Speed Distribution*

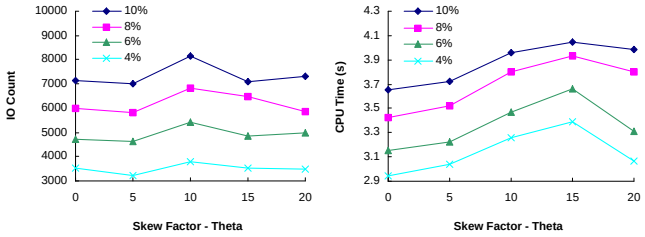
In this set of experiments, we fixed the moving data points update interval to 60, varied the update ratio from 4% to 10% to see the effect of skewed initial speed distributions. The skew factor *theta* ranges from 0, which means a uniform distribution, to 20, which means 80% data points have speed lower than the top 20% largest speeds. Each data point's speed still varies from 10 to 30.

Figure 17(a) shows that when *theta* equals to 15, the IO cost reaches the highest. In Figure 17(b), CPU time increases slowly as *theta* increases from 0 to 15, and then decreases when *theta* equals to 20. As the number of faster moving data points increases the processing cost first also increases. When that number is large enough, however, the processing cost may stop increasing and go back to a lower level. Beside, higher ratio of moving data update still incurs higher processing costs.



(a) IO (b) CPU time

Figure 16: Effect of update



(a) IO (b) CPU time

Figure 17: Effect of speed distribution

6. CONCLUSION

In this paper, we address the problem of continuous skyline query processing. The method, using the kinetic data structure, is based on the analysis that explores the spatio-temporal 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. Our solution is experimentally evaluated to be effective and efficient.

7. REFERENCES

[1] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, 2004.

[2] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *SODA*, 1997.

[3] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. *IDEAS*, 2002.

[4] S. Borzonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[5] C. Buchta. On the average number of maxima in a set of vectors. *Source Information Processing Letters*, 33(2), 1989.

[6] G. Hjaltason and H. Samet. Distance browsing in spatial database. *ACM TODS*, 24(2), 1999.

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

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

[9] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *PODS*, 2002.

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

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

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

[13] Z. Song and N. Roussopoulos. Hashing moving objects. In *MDM*, 2001.

[14] K. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

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

[16] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee. Location-based spatial queries. In *SIGMOD*, 2003.