Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

PRISMO: predictive skyline query processing over moving objects*#

Nan CHEN†1,2, Li-dan SHOU†‡1, Gang CHEN†1, Yun-jun GAO†1, Jin-xiang DONG1

(1School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)
(2China National Tobacco Corporation Zhejiang Province Corporation, Hangzhou 310001, China)
†E-mail: {cnasd715, should, cg, gaoyj, djx}@zju.edu.cn

Received July 31, 2010; Revision accepted Jan. 5, 2011; Crosschecked Jan. 7, 2012

Abstract: Skyline query is important in the circumstances that require the support of decision making. The existing work on skyline queries is based mainly on the assumption that the datasets are static. Querying skylines over moving objects, however, is also important and requires more attention. In this paper, we propose a framework, namely PRISMO, for processing predictive skyline queries over moving objects that not only contain spatio-temporal information, but also include non-spatial dimensions, such as other dynamic and static attributes. We present two schemes, RBBS (branch-and-bound skyline with rescanning and repacking) and TPBBS (time-parameterized branch-and-bound skyline), each with two alternative methods, to handle predictive skyline computation. The basic TPBBS is further extended to TPBBSE (TPBBS with expansion) to enhance the performance of memory space consumption and CPU time. Our schemes are flexible and thus can process point, range, and subspace predictive skyline queries. Extensive experiments show that our proposed schemes can handle predictive skyline queries effectively, and that TPBBS significantly outperforms RBBS.

Key words: Spatio-temporal database, Moving object, Skyline
doi:10.1631/jzus.C10a0728 **Document code:** A **CLC number:** TP391.4

1 Introduction

Skyline query is an important operation for several applications involving multi-criteria decision making. Given a set of d-dimensional data points, a point p dominates another point q if the coordinate of p on each dimension is as good as or better than that of q, and better on at least one dimension. At the operation of a skyline query, all the points are retrieved that are not dominated by any other points on all d dimensions. For instance, a person who plans to buy a house may want to choose from the database a set of houses that are cheaper and newer than any other houses in price and 'age' attributes, respectively. Fig. 1 illustrates the case in the 2D space. The x-axis represents the price of a house, and the y-axis captures the 'age' of a house, which indicates the number of years since the house was built. House a dominates house b because the former is cheaper and newer, meaning that a is preferable. The most interested houses are $\{a, e, g\}$, called 'skyline'; no other house is better on both dimensions. Skyline query has become an active topic in the database community and received considerable attention in recent years. The existing work on skyline queries, however, is based mainly on the assumption that the dataset is static. For example, in the above instance, the price and 'age' attributes are both static.

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (Nos. 60603044 and 60803003) and the Program for Changjiang Scholars and Innovative Research Team in University (No. IRT0652)
# A preliminary version was presented at the 14th International Conference on Database Systems for Advanced Applications, Apr. 21–23, 2009, Brisbane, Australia

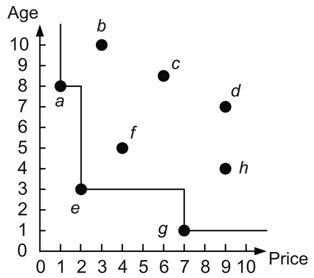

Fig. 1 An example of skyline. Each point represents a house record

The increasing interest in and requirements of numerous emerging applications, such as traffic control, mobile location computing, and meteorology monitoring, promote the development of moving object management. Indexing and querying the current and near-future statuses of moving objects is one of the key challenges that spatio-temporal databases face. In practice, the attributes of a moving object can be classified into three categories. First, a moving object has its spatial attributes in the 2D or 3D space, such as the location and velocity information, which continuously changes with time. Second, a moving object may be bounded by some non-spatial time-parameterized (NSTP) attributes whose values also change with time. Third, a moving object may also have some static attributes that do not change with time. Among the spatial attributes, the distance (from the moving object to a specified point) is probably of much interest, as it usually has an important impact on many practical issues.

In spatio-temporal databases, the existing work usually focuses on queries that are related only to spatial attributes, such as range queries and nearest neighbor (NN) queries. Considering all the above three kinds of attribute and in some special environments, however, there are some requirements for querying the skyline for moving objects at some future time to support multi-criteria optimization. For example, in a big supermarket, a salesman may query a skyline in 10 min for customers, in order to choose the ones for merchandising. Both the salesman and the customers may be on move and be seen as moving objects. The attribute dimensions to be considered here may involve the distances from the customers to the salesman (which is a spatial attribute), the purchase willingness (which may change with time and thus is an NSTP attribute), and the purchasing power (which is a static attribute). Consider another example: in a digital battle, an aid worker may query a skyline at some future time for the injured soldiers, preparing for rescue. The attributes may contain the distances between the aid worker and the injured soldiers, the severity of the injury (which may change with time), and the military rank. Among these dimensions, the distance is a spatial attribute, the severity of the injury is an NSTP attribute, and the military rank is a static attribute. This type of predictive skyline query for moving objects is valuable in many real world applications.

Intuitively, the results of predictive skyline queries change with different query points and different time instances. Fig. 2 illustrates an example of two predictive skyline point queries in the 2D space. It can be seen that the results of predictive skyline queries are different for different query points and at different time instances. Note that the change of distance may make a point change between skyline and non-skyline more than once. In addition, the static and NSTP attribute dimensions are not shown in Fig. 2. The NSTP attributes even make the change of skyline with time more complex.

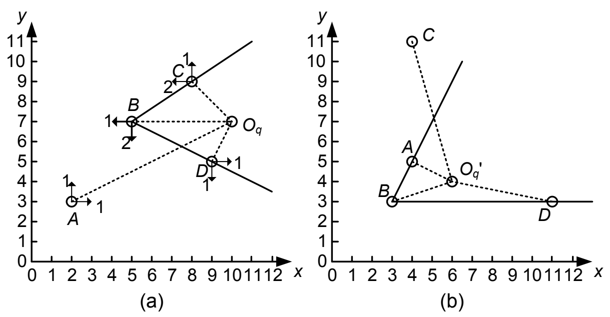

Fig. 2 Skyline for moving objects: (a) skyline at t_1; (b) skyline at t_2 ($=t_1+2$). The dataset contains four moving objects (A, B, C, and D). The query points may also be moving objects. The solid line denotes the skyline; the dashed lines indicate the distances between the query point and the moving objects

Predictive skyline query can also be issued with a 'range', which can be time-parameterized itself. In this case, the query is referred to as a 'predictive skyline range query'. Returning to the battle field example, a team of patrol defense units scattered in a mobile region may issue a predictive skyline query to choose close, threatening, and weak enemy fighter planes or missiles to launch an attack within 10 min. The query region might be a time-parameterized range in this example application.

In some practical applications, a dataset may contain many attributes, but users may choose only and focus on some of these attributes, as a subset of the original attributes. Motivated by this, Tao *et al.* (2006) proposed the problem of subspace skyline, where the skyline is computed for a static dataset on some specified attributes. In moving object datasets, one can also find the requirement for a predictive subspace skyline query. For instance, in the first example of the aid worker, the query may be interested in only the spatial distance and the seriousness of the injury at some time, and interested in the spatial distance and the military rank at another time. Therefore, query for subspace skyline is needed in this case.

In this paper, we identify and solve the problem of predictive skyline queries over moving objects. The proposed solution is implemented in a prototype framework, namely PRISMO (PRedIctive Skyline queries for Moving Objects). In this framework, we present two schemes, called RBBS (branch-and-bound skyline with rescanning and repacking) and TPBBS (time-parameterized branch-and-bound skyline). RBBS is proposed as a brute-force algorithm, while TPBBS is based on a new index structure, namely TPRNS-tree, which is proposed to index moving objects with all three kinds of attributes. The proposed schemes support point, range, and subspace skyline queries. We implement RBBS and TPBBS with two alternative methods separately. We prove that both RBBS and TPBBS are I/O optimal. In addition, we enhance basic TPBBS to TPBBSE for improving the performance of memory space consumption and CPU time, based on the characteristic analysis of TPBBS and the predictive skyline queries over moving objects. Extensive experimental evaluation demonstrates that both schemes can handle predictive skyline queries effectively, and that TPBBS/TPBBSE outperforms RBBS significantly.

We are aware of the work by Huang *et al.* (2006), who proposed a method to continuously monitor skyline queries for moving objects during a period of time, using an event-driven approach. It is totally different, however, from our work for the following reasons. First, the former focuses on the continuous skyline query during a period of time, while ours is for the predictive skyline query at some future time instance. The solution of Huang *et al.* (2006) is first to compute the skyline at the beginning of the continuous query, and then to monitor the change of the skyline for a constant query point. This solution is not applicable for our work, which requires one directly query for different query points at different time. Second, the query region in Huang *et al.* (2006) can only be a point, because it adopts the representation of distances between points as single higher order functions. Nevertheless, the distance between a moving point and a moving range is not a function of time only. In contrast, the query region in our method can also be a range. Third, the former cannot handle continuous skyline queries in subspaces, while in our work, subspace skyline can be computed at a future time. Finally, in the former work, the only dynamic dimension is the distance, while other dimensions are static. In our work, however, all dimensions can be time-parameterized and updated at any time. To the best of our knowledge, this is the first piece of work on predictive skyline queries for moving objects.

2 Related work

2.1 Access methods for moving objects

Traditional indexes for multi-dimensional databases, such as the R-tree (Guttman, 1984) and its variant R*-tree (Beckmann *et al.*, 1990), are designed to support efficient query and update for static objects. Each entry in the R-tree or R*-tree is represented as a minimum bounding rectangle (MBR). In particular, the MBR of a leaf entry denotes the extent of an object, while the MBR of a non-leaf entry tightly bounds all MBRs in its child node. These indexes work well in applications for static objects, but they are not appropriate to index moving objects.

Several index structures have been proposed for moving object indexing. The well-known examples include the time-parameterized R-tree (TPR-tree) (Saltenis *et al.*, 2000) and its variants, such as the TPR*-tree (Tao *et al.*, 2003), the R$^{\mathrm{EXP}}$-tree (Saltenis and Jensen, 2002), and the TPRU-tree (Lin *et al.*, 2003), which are all R-based-trees, indexing only on spatial dimensions. The TPR-tree is an extension to the R*-tree. Moving objects that the TPR-tree indexes are modeled as linear time functions. MBRs of these objects are also time-parameterized. For an MBR, in each dimension, the upper bound is set to move with the maximum speed of the enclosed moving objects, while the lower bound is set to move with the minimum speed of the enclosed moving objects. Therefore, at all times considered, each MBR always bounds the objects that belong to it. MBRs are not tight and are generally larger than strictly needed. To avoid the case where MBRs grow to be too large, whenever a moving object is updated, all MBRs of the nodes along the path to the leaf, which the object belongs to, are recomputed. Based on the same structure of the TPR-tree, the TPR*-tree provides a new set of deletion and insertion operations aimed at minimizing a certain penalty metric function. Considerable work has been done to solve spatial query problems on the TPR-tree and TPR*-tree, such as Benetis *et al.* (2006) for predictive K-nearest neighbor (KNN) and reverse nearest neighbor (RNN) queries.

The TPR-tree and TPR*-tree are two most well-received common access methods for moving objects, due to their high efficiency and high extendability. In addition, with the variants of the TPR-tree, some specialized index structures have also been proposed for special use. Specifically, the $\mathrm{R}^{\mathrm{EXP}}$-tree is designed to handle moving objects with expiration time. The TPRU-tree indexes and queries moving objects with uncertain information, taking the uncertainty into consideration. The QU-trade (Tzoumas *et al.*, 2009) is a framework built on top of the TPR-tree to make the index adaptive to the workload characteristics. Besides the TPR-tree and TPR*-tree, there are some other methods for indexing moving objects. For example, the B^x-tree (Jensen *et al.*, 2004) indexes moving objects by a B^+-tree, using the method of space-filling curve. Chen *et al.* (2008; 2010) proposed some indexes based on the B^x-tree. The work of Chen *et al.* (2008) is suitable for the circumstances in which the update frequencies of moving objects become highly variable, while the work of Chen *et al.* (2010) can balance the update performance and query performance in a self-tuning way.

2.2 Static skyline query

Static skyline query processing has been extensively studied, and a large number of algorithms have been proposed. Borzsonyi *et al.* (2001) first introduced the skyline operator in the database context and developed two skyline computation methods, including block-nested-loop (BNL) and divide-and-conquer (D&C). BNL sequentially scans the whole data file and compares each point with the skyline candidates kept in memory. Only those points not dominated by others are kept as skyline candidates. D&C divides a dataset into several partitions, which are small enough to fit in memory. A partial skyline is computed for each partition. Then, partial skylines are merged and the dominated data points are removed until a complete skyline is obtained. Sort-filter-skyline (SFS) (Chomicki *et al.*, 2003) is based on the same rationale as BNL, but it improves performance by first sorting the data according to a monotone function. Bitmap (Tan *et al.*, 2001) encodes each point p to a bitmap, containing the information of the number of points that have smaller values than p on each dimension. Then, the skyline is obtained using bit operations. In the Index approach (Tan *et al.*, 2001), points are assigned to one of D lists according to the dimension with the smallest value, and are indexed by a B^+-tree. Every list is further divided into batches, in each of which the local skyline is computed and merged into the global skyline. Linear-elimination-sort for skyline (LESS) (Godfrey *et al.*, 2005) combines external sort and skyline search in a multi-pass fashion. It reduces sorting cost, and provides an attractive asymptotic best-case and average case performance, if the number of skyline points is small.

NN (Kossmann *et al.*, 2002) indexes the dataset by an R-tree, and uses the result point of a nearest neighbor query, which is part of the skyline, to partition the space recursively. Branch-and-bound skyline (BBS) (Papadias *et al.*, 2005) is an efficient static skyline query algorithm, sophisticated and widely used. It also indexes the dataset by an R-tree, and uses the nearest neighbor query as the NN algorithm. The difference is that NN requires multiple nearest neighbor queries, but BBS executes only a single retrieval of the tree. BBS maintains a heap of index entries and a result set of skylines. BBS is based on the best-first nearest neighbor (BF-NN) algorithm (Hjaltason and Samet, 1999) and is proved to be I/O optimal. It traverses the R-tree by best-first order and enqueues the traversed entries in ascending order of their mindist (minimum distance). Unlike BF-NN, however, BBS uses L1 norm of all coordinates to compute mindist, and for each traversed entry, BBS compares it with the result set, and enqueues only those entries that are not dominated by any skyline point. NN, BBS, and some other algorithms for static skyline queries all use R-tree as the index structure, because R-tree and its variants have a good characteristic of spatial locality and have been proved to be effective. In the solutions to our problem, we also use R-based-trees.

Recently, other variations of static skyline queries have been proposed in the database literature. Most of them, however, assume static datasets and none query predictive skylines over moving objects. Gao *et al.* (2006) focused on improving the performance of BBS. Lee *et al.* (2007) proposed the methods of answering skying using Z-order curve. Vlachou *et al.* (2008), Cui *et al.* (2009), and Wang *et al.* (2009) focused on skyline queries in distribution or peer-to-peer systems. Yuan *et al.* (2005) and Pei *et al.* (2007) dealt with the problem of skyline cubes. Subspace skyline is also one of these variants of static skyline. For points in a dataset that has an attribute set S with d dimensions, given a subset S_{sub} with d_{sub} dimensions where $d_{\text{sub}} < d$, a subspace skyline query returns the skyline in the dimensions of S_{sub}. Tao *et al.* (2006) proposed the problem of subspace skyline in static environments and dealt with it using an algorithm named SUBSKY. SUBSKY transforms data points with d static dimensions to 1D values and indexes these values by a single conventional B-tree. The result of subspace skyline is achieved by some simple scans of the B-tree. The method of SUBSKY, however, is not appropriate to predictive subspace skyline queries for moving objects. Moving objects are moving with dynamic NSTP attributes, and thus cannot be converted to simple 1D values.

3 Preliminaries and problem statement

Before introducing the algorithms, we give the preliminaries and problem statement.

3.1 Preliminaries

3.1.1 Attributes of moving objects

To reduce the requirement of updates, in most works of indexing moving objects (Saltenis *et al.*, 2000; Saltenis and Jensen, 2002; Lin *et al.*, 2003; Tao *et al.*, 2003; Jensen *et al.*, 2004), moving objects are modeled as linear time functions. A function can be updated whenever a moving object changes its moving status. Specifically, it is subject to update only if the real state of an object deviates considerably from its functional model in the database. And, any complex motion can be seen as the combination of linear functions. Thus, in PRISMO, moving objects are also modeled as linear time functions. Therefore, the position of a moving object at a time instance t in a d-dimensional spatial space is

$$\boldsymbol{O}(t) = (\boldsymbol{x}, \boldsymbol{v}, t_{\text{ref}}) = \boldsymbol{x}(t) = (x_1(t), x_2(t), ..., x_d(t)),$$

where $t \geq t_{\text{ref}}$. Here, $\boldsymbol{x}$ is the position vector of the moving object at a reference timestamp t_{ref}, $\boldsymbol{v} = (v_1, v_2, ..., v_d)$ is a velocity vector. Hence, $\boldsymbol{x}(t) = \boldsymbol{x}(t_{\text{ref}}) + \boldsymbol{v} \cdot (t - t_{\text{ref}})$. Given a query point q in the d-dimensional spatial space, its position at timestamp t is $\boldsymbol{O}_q(t) = (x_{q1}(t), x_{q2}(t), ..., x_{qd}(t))$. Thus, we can obtain the distance from the query point to the moving object:

$$\text{distance}(\boldsymbol{O}(t), \boldsymbol{O}_q(t)) = \left(\sum_{i=1}^{d} (x_i(t) - x_{qi}(t))^2\right)^{\frac{1}{2}}.$$

For a moving object, the NSTP attribute values can be represented as $y_1(t), y_2(t), ..., y_i(t)$, where i is the number of NSTP dimensions. Each single NSTP attribute y_k $(k = 1, 2, \ldots, i)$ is also represented as a linear time function $y_k(t) = y_k(t_{\text{ref}}) + c_k \cdot (t - t_{\text{ref}})$, where c_k is an attribute-specific coefficient. Note that the function of an NSTP attribute can also be updated at any time when needed. Without loss of generality, we assume that $y_k(t)$ is always no less than zero. If $y_k(t)$ is a negative, we can add a positive constant C to $y_k(t)$ for each moving object, $y'_k(t) = y_k(t) + C$, where C is large enough to ensure that $y'_k(t)$ is no less than zero. This transformation does not affect the domination relationship, and hence does not change the query result.

A moving object may also have some static attributes $z_1, z_2, ..., z_j$. For each single static attribute z_k $(k = 1, 2, \ldots, j)$, its value is static and does not change until the next update.

Without loss of generality and for simplicity in presentation, the minimum condition is used on all dimensions to compute the skyline. In a minimum condition, smaller attribute values are preferred in comparison to deterministic dominance between two points. The proposed algorithm can be easily applied to different conditions (e.g., the maximum condition).

3.1.2 Distances

Predictive skyline queries for moving objects require distance computing. The point-to-point distance is defined as the normal Euclidean distance. In addition, to support the algorithm of predictive skyline point query and range query, we need to give the definitions of the 'point-region distance' and 'region-region distance'.

Definition 1 (Point-region distance) The minimum distance between a point and a rectangular region.

Definition 2 (Region-region distance) The minimum distance between two rectangles.

Fig. 3a shows the method of computing the 2D point-region distance. Figs. 3b and 3c show the method of computing the 2D region-region distance.

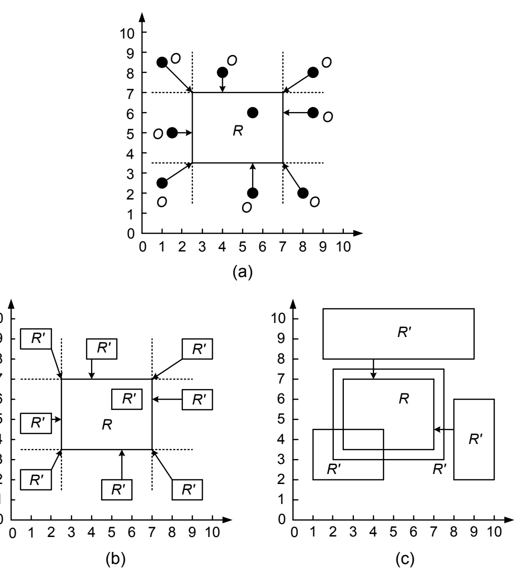

Fig. 3 Minimum distances: (a) point-region distance; (b, c) region-region distance. In (a), each solid circle denotes a case of object O for computing the distance, while R denotes the rectangle. The arrowed line indicates the distance between the point and the rectangle. If O is in R, the distance is zero. In (b) and (c), R and R' denote different cases of two rectangles, and the arrowed line indicates the distance between them. If two rectangles overlap, the distance is zero

3.2 Problem statement

Before introducing the problems, we look at the data representation. Given a set of n moving objects, their data points are in vector form as follows:

$$< x_1, x_2, ..., x_d, v_1, v_2, ..., v_d, y_1, y_2, ..., y_i,$$
$$c_1, c_2, ..., c_i, z_1, z_2, ..., z_j > .$$

Herein, the first d components $x_1, x_2, ..., x_d$ indicate the spatial location at reference time $t_{\rm ref}$, while $v_1, v_2, ..., v_d$ are the respective velocity components on the corresponding spatial dimensions. In this study, we assume the number of spatial dimensions d is 2. The proposed techniques, however, are sufficiently general for a higher dimensional space as well. $y_1, y_2, ..., y_i$ indicate the NSTP attribute values at the timestamp $t_{\rm ref}$, and $c_1, c_2, ..., c_i$ are the corresponding coefficients. The number of NSTP dimensions is i. $z_1, z_2, ..., z_j$ are the values of the j static attributes at $t_{\rm ref}$.

The above vector representation supports the definition of an 'original space'.

Definition 3 (Original space, S) The d-dimensional spatial attributes, the NSTP attribute dimensions whose number is i, and the static attribute dimensions whose number is j, together comprise a D-dimensional space, where $D = d + i + j$. The original space is the space of the original information of moving objects.

The original space, however, is not the one in which predictive skyline queries are to work. Instead, we need to obtain the distance as one dimension. This leads to the definition of a 'target space'.

Definition 4 (Target space, S') The dimension of distance, the NSTP attribute dimensions whose number is i, and the static attribute dimensions whose number is j, together comprise a D'-dimensional space, where $D' = 1 + i + j$. The target space is the one that PRISMO queries. Since $d = 2$, we obtain $D' = (d - 1) + i + j = D - 1$.

We now present the formal definitions for predictive skyline of moving objects.

Definition 5 (Predictive skyline point query) Given the time instance t_q which does not precede the current time and a query object o_q which may also be a moving object, the predictive skyline point query, represented as $Q(o_q, t_q)$, obtains the skyline points in the target space S' at the timestamp t_q, that is, the points not dominated by any other points in the D' dimensions at time t_q.

Definition 6 (Predictive skyline range query) Given a time-parameterized query range r_q which moves and changes size by time, the predictive skyline range query can be represented as follows:

$<x_1^-, x_1^+, x_2^-, x_2^+, ..., x_d^-, x_d^+, v_1^-, v_1^+, v_2^-, v_2^+, ..., v_d^-, v_d^+>$.

Here x_i^+ and x_i^- ($i = 1, 2, \ldots, d$) are the upper and lower bounds of r_q at the reference timestamp t_{ref} on each spatial dimension respectively, and v_i^+ and v_i^- ($i = 1, 2, \ldots, d$) are velocity components of the upper and lower bounds on dimension i respectively. Thus, the query range at time t is $r_q(t) = (x_1^-(t), x_1^+(t), x_2^-(t), x_2^+(t), ..., x_d^-(t), x_d^+(t))$, where $x_i^{-/+}(t) = x_1^{-/+} + v_i^{-/+} \cdot (t - t_{\text{ref}})$. The predictive skyline range query, denoted as $Q(r_q, t_q)$, obtains the skyline points in the target space S' at time instance t_q. Note that the dimension of distance here is the point-rectangle distance from moving objects to the query range.

Definition 7 (Predictive subspace skyline query) Given a subspace of target space S', namely S'_{sub}, the predictive subspace skyline query, denoted as $Q_{\text{sub}}(o_q, t_q, S'_{\text{sub}})$ for point query (or $Q_{\text{sub}}(r_q, t_q, S'_{\text{sub}})$ for range query), retrieves the set of subspace skylines in S'_{sub} at time t_q. Note that the attributes in S'_{sub} can be chosen discretionally from those in S'.

Note that the distance from a moving object to a query point (or a query range) is not a linear function (or a single function). Thus, a moving object may change between skyline and non-skyline several times. Several NSTP attributes, which also change by time, even make the change of skyline more complex and more frequent. In addition, the moving functions, NSTP functions, and static attributes can all be updated at any time. Furthermore, the query point may be different for different queries. For a constant query point, Huang *et al.* (2006) first computed the skyline at the initial time, and then computed for each moving object the interval when it is in the skyline. Obviously, PRISMO is not applicable to this method.

For simplicity in presentation, we use terms 'point query', 'range query', and 'subspace query' to refer to the respective predictive skyline queries in the remainder of this paper.

4 Algorithms for computing predictive skylines

In this section, we propose our solutions to the problem of querying predictive skylines over moving objects.

4.1 RBBS

It is a natural approach to answering predictive skyline queries over moving objects by a good static skyline query algorithm, such as BBS. Before processing BBS, each query has to rescan and recompute the dataset, and rebuild the R-tree. This brute-force solution is called RBBS (BBS with rescanning and repacking).

When a point query $Q(o_q, t_q)$ arrives, RBBS first uses the moving function of the query point o_q to obtain its position o'_q at the query timestamp t_q. Second, RBBS scans the dataset. For each moving object, RBBS computes the distance from o'_q to the position of this object at the query timestamp t_q, and computes the NSTP attribute values at the query time instance t_q. Then, RBBS rebuilds an R-tree on the dimensions of the target space S', including the distance attribute, the NSTP attributes, and the static attributes.

A range query is handled similarly, except that the distance is computed using the point-rectangle distance between moving objects and the query range, as defined in Section 3.1. The solution to a subspace query is also straightforward. The only difference is the dimensionality of the index. To answer a subspace query, RBBS computes only the dimensionality of S'_{sub} and rebuilds an R-tree with these D'_{sub} dimensions.

Now, let us look at the problem of rebuilding the R-tree. The conventional R-tree supports insertions and deletions. Operating in this mode, the R-tree guarantees good space utilization. In RBBS, however, the R-tree is only a snapshot for all moving objects at query time t_q. As an R-tree is rebuilt for each query, it is static and does not ask for any deletion, insertion, or update operation. Therefore, the space utilization of the index should be as high as possible. High space utilization not only reduces the space overhead of the R-tree index, but also improves query performance. Meanwhile, the cost of rebuilding the R-tree should be as low as possible. Rebuilding an R-tree with insertions, however, may ask for very high cost of I/Os, as every insertion may involve several I/Os. Regarding the above issues, the conventional insertion scheme is too costly for rebuilding the R-tree in RBBS.

According to the above characteristic analysis of the R-tree in RBBS, we use a scheme of packing for rebuilding the R-tree. Instead of processing an insertion operation for every object to rebuild an R-tree, using a packing method we first preprocess the dataset of n objects and sort the objects so that the n objects are ordered in $[n/b]$ consecutive groups of b rectangles, where each group of b is intended to be placed in the same leaf level node. Note that the last group may contain fewer than b rectangles. Then, the packing method loads each group to a leaf node and recursively packs the nodes at the next level, from bottom to top, until the root node is created. In this way, the resultant R-tree will have nearly 100% space utilization, and the I/O cost of repacking the index is only the total number of the nodes. Obviously, this building cost is optimal for an R-tree and is much lower than that of the insertion scheme. Note that the packing methods are not appropriate for the environments that ask for dynamic insertions and deletions, but good for static indexes, such as the R-tree of RBBS. Typical packing methods include Hilbert sort (HS) (Kamel and Faloutsos, 1993) and sort-tile-recursive (STR) (Leutenegger *et al.*, 1997). We use these two alternative packing methods to rebuild the R-tree, and compare them in RBBS by experiments.

After repacking the R-tree, in RBBS, a standard static BBS algorithm is processed to obtain the results of skyline points. For a predictive subspace skyline query, BBS is performed on the D'_{sub} R-tree dimensions. Note that the I/O cost of BBS is optimal.

RBBS is straightforward and simple, but the disadvantage is obvious. For each query, RBBS has to rescan the dataset and rebuild the index. This will cause additional rebuilding cost, though the I/O cost is optimal. Therefore, RBBS is a naive brute-force algorithm.

4.2 TPBBS

Regarding the scanning and rebuilding cost of RBBS, we advocate a time-parameterized indexing scheme to process the predictive skyline queries. In this subsection, we propose an indexing and query processing scheme called TPBBS (time-parameterized branch-and-bound skyline), which does not ask for rescanning the dataset or rebuilding the index for each query.

4.2.1 Index structure

As reviewed in Section 2, the TPR-tree is used to index the current and future positions of moving objects. Its dimensions are only in the location space. In the problem of querying predictive skylines over moving objects, however, the target space also includes non-spatial dimensions: the NSTP attribute dimensions and the static attribute dimensions. Therefore, the TPR-tree cannot be directly employed for our use. The R-tree cannot be used either, because its dimensions are static. In addition, note that the result skyline of PRISMO is not the union of the separate skyline of three kinds of attributes. Therefore, to support TPBBS, we propose a unified R-based-tree index structure with all three kinds of attributes, named TPRNS-tree (time-parameterized R-tree with non-spatial dimensions).

In the TPRNS-tree, an index entry is represented by an MBR o_{m} and a velocity bounding rectangle (VBR) o_{v}, where $o_{\text{m}} = \{o^-_{\text{m}1}, o^+_{\text{m}1}, o^-_{\text{m}2}, o^+_{\text{m}2}, ..., o^-_{\text{m}D}, o^+_{\text{m}D}\}$, and $o_{\text{v}} = \{o^-_{\text{v}1}, o^+_{\text{v}1}, o^-_{\text{v}2}, o^+_{\text{v}2}, ..., o^-_{\text{v}(d+i)}, o^+_{\text{v}(d+i)}\}$. Note that different from the TPR-tree and its other variants, in the TPRNS-tree, MBRs and VBRs contain not only spatial dimensions, but also non-spatial dimensions. In addition, the dimensionalities of VBRs and MBRs are also different. There are $D_{\text{v}} = d + i$ dimensions in the VBRs, while $D_{\text{m}} = D = d + i + j$ dimensions in the MBRs. In other words, VBRs include the information of spatial dimensions and the NSTP attributes, while MBRs include the information of all dimensions of the original space S: the spatial, NSTP, and static dimensions. Besides an MBR and a VBR, each entry in the leaf nodes also has a pointer to the moving object, while each entry in the internal nodes has a pointer to the subtree.

In the TPRNS-tree, the lower and upper bounds of the MBRs and VBRs are settled differently for different dimensions. For any dimension of the d spatial and i NSTP attributes, the lower (upper) bound of a VBR is set to the minimum (maximum) speed of the enclosed moving objects in the corresponding MBR, and the lower (upper) bound of the MBR is set to move at this speed. In contrast, for any dimension of the j static attributes, the lower and upper bounds of the MBRs are static values. An MBR bounds all enclosed MBRs at all times not earlier than the current time and it never shrinks. Therefore, the first

$D_\mathrm{v} = d + i$ dimensions of an MBR are not tight, but the bounds of the remaining j dimensions are always tight, which profits the update and query performance.

Similar to the TPR-tree, the insertion algorithm of the TPRNS-tree is based on the one of the R*-tree. The insertion algorithm of the R*-tree aims at minimizing the following penalty metric functions: the area of an MBR, the perimeter of an MBR, the overlap of two MBRs, and the distance between the centers of two MBRs. The insertion algorithm of the TPRNS-tree replaces the four penalty metric functions with their integral counterparts. Specifically, let t_ref be the index creation time, t_c be the current update time, H be a parameter of the tree called horizon, which means how far the queries can 'see' in the future, and $A(t)$ be the value of the penalty metric function at t. The integral is given by

$$I = \int_{t_\mathrm{c}}^{t_\mathrm{c}+H} A(t)\mathrm{d}t.$$

Different from the TPR-tree, however, in the TPRNS-tree, the last j dimensions are static, so there are some constants of time in these four integrals. We use these constants to accelerate the computation of the integral counterparts, and improve the insertion and update performance. Specifically, for the ones of the area and perimeter, we have

$$\begin{aligned} I_\mathrm{area} &= \int_{t_\mathrm{c}}^{t_\mathrm{c}+H} C_1...C_j f_1(t)...f_{d+i}(t)\mathrm{d}t \\ &= C_1...C_j \int_{t_\mathrm{c}}^{t_\mathrm{c}+H} f_1(t)...f_{d+i}(t)\mathrm{d}t, \end{aligned}$$

$$\begin{aligned} & I_\mathrm{perimeter} \\ &= \int_{t_\mathrm{c}}^{t_\mathrm{c}+H} (C_1 + ... + C_j + f_1(t) + ... + f_{d+i}(t))\mathrm{d}t \\ &= (C_1 + ... + C_j)H + \int_{t_\mathrm{c}}^{t_\mathrm{c}+H} (f_1(t) + ... + f_{d+i}(t))\mathrm{d}t. \end{aligned}$$

Here, C_k indicates the value of the MBR on dimension k ($k \in [1, j]$) of the static attributes, while $f_k(t)$ indicates the value of the MBR on dimension k ($k \in [1, d+i]$) of the spatial and NSTP attributes at time t. The integral counterparts of the overlap and distance can also partially use the constants. Therefore, instead of computing the entire integrals of all D dimensions, which is costful, we first carry out part of the integrals, according to the VBRs and the first $d + i$ dimensions of MBRs, and then consider the remaining j constant dimensions of MBRs of the static attributes. This will reduce the computation cost, and accelerate the insertion and update operations of the TPRNS-tree. The operations of split and deletion are performed as they are done in the TPR-tree.

For a moving object, when the moving functions, the functions of NSTP attributes, or the static attributes are changed, an update operation of the TPRNS-tree is performed. Just like most time-parameterized indexes of moving objects, update of a moving object in the TPRNS-tree is implemented as a deletion followed by an insertion. Note that not only the moving function and the NSTP functions, but also the static attributes of a moving object can be updated. The term 'static' here means being constant until the next update.

The TPR-tree is optimized for timestamp queries in interval $[t_\mathrm{c}, t_\mathrm{c} + H]$. A variant of TPR-tree, called TPR*-tree, is optimized for static point interval queries. The TPR*-tree provides a new set of insertion and deletion algorithms, aimed at minimizing another cost function, which is different from the four penalty metric functions above. We also implement an index called TPR*NS-tree, which has the same structure as the TPRNS-tree, but uses the penalty metric function of the TPR*-tree. We compare the TPRNS-tree and TPR*NS-tree in TPBBS by experiments.

4.2.2 Query algorithm

Algorithms proposed for the static skyline are not applicable to predictive skyline queries over moving objects on the TPRNS-tree. This is because first, the result of a predictive skyline query is not constant. It depends on the query time and the query point (or query range). Second, besides static attributes as previous static skyline, PRISMO also contains dynamic attributes: the spatial and NSTP dimensions of the TPRNS-tree, which are time-parameterized. In addition, the target dimension of distance is not a single linear time function. Finally, the dimensions of the TPRNS-tree are in the original space S with the dimensionality of D, while the predictive skyline queries are in the target space S' with D' dimensions. Therefore, we propose the query algorithm of TPBBS, to answer predictive skyline queries over moving objects.

First, let us focus on predictive skyline point queries $Q(o_q, t_q)$. Since these queries contain time-parameterized dimensions, for every accessed moving object in the TPRNS-tree, TPBBS computes its location and NSTP attributes at t_q. For every accessed MBR in the TPRNS-tree, TPBBS expands the spatial and NSTP dimensions to t_q according to the corresponding VBR. The main approach of TPBBS is branch-and-bound on the R-based-tree. Specifically, on the non-spatial dimensions of the TPRNS-tree, i.e., the NSTP and static attributes, the bottom-left corner p of an expanded MBR dominates all points in the MBR. Therefore, if p is dominated by another point o on these dimensions, all points in the MBR are dominated by o and can be discarded safely on these dimensions. On the dynamic spatial dimensions, however, our purpose is to compare the distances between the query point o'_q and the moving objects at the query time instance t_q. We use a pruning technique based on the min-distance for the spatial dimensions. The point-rectangle distance (prd in short) from a query point o'_q to an expanded MBR is as shown in Fig. 3a. Since all objects bounded by the MBR have longer distances to o'_q than prd, if there exists a point o whose distance to o'_q is shorter than prd, all objects bounded by the MBR can be discarded on spatial dimensions.

In TPBBS, we maintain two data structures: one is a skyline set S, which becomes populated with the algorithm; the other is a heap H, which contains and sorts the entries of unexamined moving objects and indexes MBRs, which are not dominated by any point in S. Different from BBS, in TPBBS, the entries in S and H are not the same as in the index. These entries need to be transformed to the so-called 'transition entries', or te for short. In addition, in TPBBS, te of a moving object (leaf entry) and te of an expanded MBR (expanded internal entry) are different. Specifically, a point entry is represented as

$$\mathrm{te}_{\mathrm{point}}(o_q, t_q) = <\mathrm{ppd}(o_q, t_q), \mathrm{NSTP}(t_q), \mathrm{static}> .$$

It contains three kinds of members: (1) the point-to-point distance (ppd for short) between its position and the query point position o'_q at query time instance t_q, (2) the NSTP attribute values at timestamp t_q, and (3) the static attribute values. For an MBR entry, its te is represented as

$$\mathrm{te}_{\mathrm{MBR}}(o_q, t_q) = <\mathrm{prd}(o_q, t_q), \mathrm{NSTP}_{\text{bottom-left}}(t_q), \mathrm{static}_{\text{bottom-left}}, \mathrm{pointer}> .$$

It contains four kinds of members: (1) the minimum point-to-region distance (prd for short) from the query point o'_q to its expanded MBR in the spatial space at query time instance t_q, (2) the NSTP attribute values of its expanded bottom-left corner in the non-spatial space at timestamp t_q, (3) the values of static attributes of its bottom-left corner in the non-spatial space, and (4) a pointer that points to its corresponding non-leaf node in the index for obtaining the detailed information of its children.

The entries in H are sorted by a key, namely the minimum value (mv for short). For an expanded index MBR (or a moving object), its mv is the sum of the coordinates of the first three kinds of members in its $\mathrm{te}_{\mathrm{MBR}}$ (or $\mathrm{te}_{\mathrm{point}}$). Specifically, for a moving object,

$$\mathrm{mv}_{\mathrm{point}}(o_q, t_q) = \mathrm{ppd}(o_q, t_q) + \sum_{k=1}^{i} y^k_{\mathrm{point}}(t_q) + \sum_{k=1}^{j} z^k_{\mathrm{point}},$$

while for an index MBR,

$$\mathrm{mv}_{\mathrm{MBR}}(o_q, t_q) = \mathrm{prd}(o_q, t_q) + \sum_{k=1}^{i} y^k_{\text{bottom-left}}(t_q) + \sum_{k=1}^{j} z^k_{\text{bottom-left}}.$$

The skyline set S contains the final results of the query, and each skyline point inserted to S is from H. The domination comparison is performed between the transition entries in H and S. The domination comparison between a point and an MBR (or between two points), is processed to compare the coordinates of the first three kinds of members in their $\mathrm{te}_{\mathrm{point}}$ (or $\mathrm{te}_{\mathrm{MBR}}$).

Algorithm 1 gives the TPBBS algorithm for point query. For the initialization, the algorithm first computes the query point position o'_q at the query time instance t_q (line 1). Then, for each child entry of the index root, the algorithm expands MBR to t_q (or computes the point position at t_q), generates the transition entries te, computes their mv, and inserts them into H according to their mv (lines 2–10). After the initialization, the algorithm enters a 'while' loop until there is no entry in H. In every loop, we pop the top entry E from the heap H. If E is an internal index MBR, we obtain its expanded $\mathrm{te}_{\mathrm{MBR}}$, and compare it with all $\mathrm{te}_{\mathrm{point}}$ in S. If it is dominated by some point, discard it (lines 13–16); else, we use the pointer in $\mathrm{te}_{\mathrm{MBR}}$ to obtain the information of its children. For every child entry of the internal MBR, we do the same as in the initialization, except that before inserting the transition entry into H, we compare it with all $\mathrm{te}_{\mathrm{point}}$ in S. If it is dominated by some skyline point in S, discard it (lines 17–24). If the popped entry E is a moving object, we compute its $\mathrm{te}_{\mathrm{point}}$ and compare it with all $\mathrm{te}_{\mathrm{point}}$ in S. If it is dominated by some point, discard it; else, insert its $\mathrm{te}_{\mathrm{point}}$ into S (lines 25–30). Note that, during the algorithm, for each candidate entry, its te and mv are computed only once.

Algorithm 1 TPBBS for point query $Q(o_q, t_q)$

```
Input: $R$ is the root node of the TPRNS-tree, $t_q$ is the
    query time instance, and $o_q$ is the query point.
Output: $S$ is the result set of predictive skyline points.
 1: Compute the position $o_q'$ of $o_q$ at $t_q$, $S \leftarrow \varnothing$, $H \leftarrow \varnothing$;
 2: for each child entry $e_i$ of the root node $R$ do
 3:   if $e_i$ is an internal index MBR then
 4:     Expand MBR to $t_q$;
 5:     Form its $\mathrm{te}_{\mathrm{MBR}}$ and compute its $\mathrm{mv}_{\mathrm{MBR}}$;
 6:     Insert $\mathrm{te}_{\mathrm{MBR}}$ into $H$ according to its $\mathrm{mv}_{\mathrm{MBR}}$;
 7:   else
 8:     Compute the position of point at $t_q$;
 9:     Form its $\mathrm{te}_{\mathrm{point}}$ and compute its $\mathrm{mv}_{\mathrm{point}}$;
10:     Insert $\mathrm{te}_{\mathrm{point}}$ into $H$ according to its $\mathrm{mv}_{\mathrm{point}}$;
11: while $H$ is not empty do
12:   Pop the top entry $E$ from $H$;
13:   if $E$ is an internal index MBR then
14:     Retrieve $\mathrm{te}_{\mathrm{MBR}}$ of $E$;
15:     if $\mathrm{te}_{\mathrm{MBR}}$ is dominated by some point in current
        $S$ then
16:       discard $E$;
17:     else
18:       for each child entry $e_i$ of $E$ do
19:         Expand MBR to $t_q$ (or compute the point
            position at $t_q$);
20:         Form its $\mathrm{te}_{\mathrm{MBR}}$ (or $\mathrm{te}_{\mathrm{point}}$) and compute
            its $\mathrm{mv}_{\mathrm{MBR}}$ (or $\mathrm{mv}_{\mathrm{point}}$);
21:         if $\mathrm{te}_{\mathrm{MBR}}$ (or $\mathrm{te}_{\mathrm{point}}$) is dominated by some
            skyline point in current $S$ then
22:           discard $e_i$;
23:         else
24:           Insert $\mathrm{te}_{\mathrm{MBR}}$ (or $\mathrm{te}_{\mathrm{point}}$) into $H$ accord-
              ing to its $\mathrm{mv}_{\mathrm{MBR}}$ (or $\mathrm{mv}_{\mathrm{point}}$);
25:   else
26:     Retrieve $\mathrm{te}_{\mathrm{point}}$ of $E$;
27:     if $\mathrm{te}_{\mathrm{point}}$ is dominated by some skyline point
        in current $S$ then
28:       discard $E$;
29:     else
30:       Insert $\mathrm{te}_{\mathrm{point}}$ into $S$;
31: Return the result set $S$.
```

We now analyze the performance of TPBBS. Considering a predictive skyline query, in the D'-dimensional target space S', we define the line decided by skyline points of the query $Q(o_q, t_q)$ as SKY-LINE(o_q, t_q) (Fig. 4). We can prove that:

Theorem 1 TPBBS visits only the MBRs that overlap SKY-LINE(o_q, t_q) when being expanded to time t_q.

Proof For any MBR expanded to t_q that does not overlap SKY-LINE(o_q, t_q), it is not possible to fall on the left side of SKY-LINE(o_q, t_q), because otherwise, any object in this expanded MBR will dominate some points in SKY-LINE(o_q, t_q), which conflicts with the definition of skyline. Therefore, if no overlap, this expanded MBR must fall on the right side of SKY-LINE(o_q, t_q), and there must exist at least one skyline point of predictive skyline query in SKY-LINE(o_q, t_q) for t_q that dominates this expanded MBR. In these cases, the mv of the skyline point is less than the mv of the expanded MBR. Therefore, the skyline point must be examined before the expanded MBR. It will then become a skyline result, and this MBR will be discarded later. Proved.

For example, in Fig. 4, rectangle M indicates the original MBR in the TPRNS-tree and M' indicates the expanded MBR at time t_q. M' does not overlap SKY-LINE(o_q, t_q), so there must be a skyline point b that dominates M'. The mv of b is $3 + 2 = 5$ and less than that of M', $6 + 5 = 11$. Therefore, b will be processed before M' in TPBBS. This will cause M' to be discarded later. Therefore, TPBBS will visit only the MBRs that overlap SKY-LINE(o_q, t_q) when being expanded to time t_q.

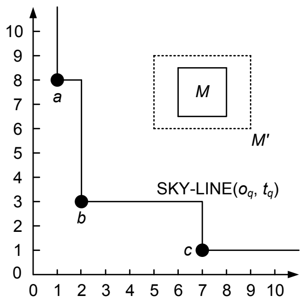

Fig. 4 An example of SKY-LINE(o_q, t_q)

From Theorem 1, and since both points and expanded MBRs are inserted into the heap only once in TPBBS, it can be seen that the query I/O cost of TPBBS is optimal. Therefore, the query I/O is decided by the indexes that TPBBS performs on. In other words, it depends on how many expanded MBRs overlap SKY-LINE(o_q, t_q). We will examine the I/O cost of TPRNS-tree and TPR*NS-tree by experiments.

Now, let us turn to predictive skyline range queries. A range query $Q(r_q, t_q)$ can be processed similarly, except for two differences. First, for the initialization, instead of computing the query point position at the query timestamp t_q, we compute the position and size r'_q of the query range r_q at t_q, according to its moving functions. Second, as the first member in te_{point} and te_{MBR}, the method of computing the distance is different. In particular, for a moving point, this distance is the point-region distance between its position and the query range r'_q at t_q (Fig. 3a). In contrast, for an internal MBR in the TPRNS-tree, this distance is the region-region distance (or rrd in short) between its expanded MBR and the query range r'_q at t_q (Figs. 3b and 3c). These two kinds of distance are used to prune moving objects and MBRs on the spatial dimensions for computing the range queries $Q(r_q, t_q)$. Any moving object or MBR in the expanded MBR must have a larger distance value than this rrd. Therefore, if there exists a point o whose point-region distance to r'_q is shorter than this rrd, all objects or MBRs bounded by the MBR can be discarded on spatial dimensions.

TPBBS can also easily support subspace queries. Given a query $Q_{\text{sub}}(o_q, t_q, S'_{\text{sub}})$ (or $Q_{\text{sub}}(r_q, t_q, S'_{\text{sub}})$), when generating transition entries for points (te_{point}) and MBRs (te_{MBR}) of the TPRNS-tree, we compute only the dimensions in the subspace S'_{sub}. The subsequent computation for their respective mv values (mv_{point} or mv_{MBR}) and domination comparison will also involve these dimensions only. The final result is the answer to the subspace query. Specifically, if the distance dimension is not in S'_{sub}, which means that the query user does not care about the distance attribute, we can discard all location dimensions during the whole query processing, and the distance values can all be ignored.

4.2.3 TPBBSE

Although the I/O cost of TPBBS is optimal, the memory space consumption and CPU time performance of the algorithm can be further improved. In this subsection, we discuss the techniques for improving the space consumption and CPU time performance according to the characteristics of predictive skyline queries over moving objects, and further extend TPBBS to TPBBSE (TPBBS with expansion) using these techniques.

The problem of querying predictive skylines over moving objects involves spatial, NSTP, and static dimensions. To analyze the relationship of predictive skyline points, we denote the skyline points in the subspace of static dimensions only as SK_{sta}, and the whole set of skyline points in the target space S' as SK_{all}.

Theorem 2 $\forall p \in \text{SK}_{\text{sta}}$, $p \in \text{SK}_{\text{all}}$; or p is dominated by another point $q \in \text{SK}_{\text{sta}}$, and q has the same values as p on all static dimensions.

Proof We denote the dataset as S and the value in dimension k of point p as $p.k$. $\forall p \in \text{SK}_{\text{sta}}$, if $\exists q$, for each static dimension k, $p.k = q.k$, obviously $q \in \text{SK}_{\text{sta}}$, q may dominate p in the target space S' of predictive skyline queries, as q may dominate p on the dimensions of distance and NSTP. Otherwise, if such q does not exist, there is no point that can dominate p in the target space S', and hence $p \in \text{SK}_{\text{all}}$. Proved.

Given a dataset S and a predictive skyline point query $Q(o_q, t_q)$ (or a range query $Q(r_q, t_q)$), SK_{sta} is static; i.e., it does not change with the query time or the query point (range). Therefore, we can compute SK_{sta} before queries, and then maintain and use it. When a predictive skyline query arrives, we filter SK_{sta} and discard the points that are dominated by some point in SK_{sta}, according to the query parameters. According to Theorem 2, the filtered set, called S_{fil}, is a part of the final result skyline set. None of the transition entry te_{point} (or te_{MBR}) dominated by S_{fil} can be a skyline point (or contain a skyline point), and thus does not need to be inserted in heap H and examined later. We can use the partial result set S_{fil} as a filtering set to filter unnecessary transition entries. This will help reduce the heap size and improve the CPU time performance.

Meanwhile, note that in TPBBS, a point or MBR transition entry te is checked for domination twice: before it is inserted in heap H, and after it is removed from H. This is because the skyline set S is populated as TPBBS proceeds, and te may be dominated by some skyline points that arrive at S when te is in H. Note that the skyline points arrive at S in ascending order of their mv and are kept in S in this order. When removed from H, te has no need to be checked for domination with the skyline points that have compared with te before te is inserted in H. This will help avoid unnecessary domination comparison and improve the CPU time performance.

We now introduce the query algorithm of TPBBSE. When a predictive skyline point query $Q(o_q, t_q)$ (or a range query $Q(r_q, t_q)$) arrives, TPBBSE is processed similarly with TPBBS in Algorithm 1, except for four differences. First, in addition to the index structure of TPRNS-tree, for a dataset and before the queries, we maintain a skyline set of static dimensions, called $\mathrm{SK_{sta}}$. When a query arrives, we use $\mathrm{SK_{sta}}$ as the input. Second, for the first step (line 1) of Algorithm 1, besides initializing the skyline set S and heap H to empty sets as TPBBS, we check the points in $\mathrm{SK_{sta}}$ that have the same static values on all static dimensions and discard the points that are dominated by other points in $\mathrm{SK_{sta}}$ at t_q. We call this filtered set S_{fil}. To accelerate this filtering step, we maintain $\mathrm{SK_{sta}}$ in a list sorted by the values of static dimensions. During the filtering step of $\mathrm{SK_{sta}}$, we sort the points in S_{fil} in ascending order of their mv at t_q. Third, before a point or MBR transition entry te is inserted in H, TPBBSE compares te not only with the points in S (line 21 in Algorithm 1), but also with the points in S_{fil} whose mv satisfies $\mathrm{mv_{sm}} < \mathrm{mv} < \mathrm{mv_{te}}$. If te is dominated, it is discarded. Here, $\mathrm{mv_{sm}}$ is the largest mv in S and $\mathrm{mv_{te}}$ is the mv of te. The points in S_{fil} with $\mathrm{mv} \leq \mathrm{mv_{sm}}$ have been in S; the points in S_{fil} with $\mathrm{mv} \geq \mathrm{mv_{te}}$ cannot dominate te and hence have no need to compare with te. Fourth, a skyline point s in S has the information whether it is in S_{fil}. Since the entries in S and S_{fil} are both in ascending order of their mv, this information can be easily obtained by a simple check before s is inserted in S. And a point or MBR transition entry te is inserted in H with the information of the current $\mathrm{mv'_{sm}}$, which is the current largest mv in S. Later, when te is removed from H (lines 15 and 27 in Algorithm 1), te is compared only with the points in S whose mv is larger than $\mathrm{mv'_{sm}}$ of te and is not in S_{fil}. Before inserted in H, te must be compared with the points in S whose mv is not larger than $\mathrm{mv'_{sm}}$ of te and the points in S_{fil}. Here, te has no need to compare with these points again. In this way, TPBBSE avoids some unnecessary domination comparison.

For a subspace query $Q_{\text{sub}}(o_q, t_q, S'_{\text{sub}})$ (or $Q_{\text{sub}}(r_q, t_q, S'_{\text{sub}})$), only if S'_{sub} contains all static dimensions, will TPBBSE compute and use S_{fil}. In summary, because of reducing the heap size and avoiding some unnecessary domination comparison, TPBBSE can improve the space consumption and CPU time performance. We will examine it by experiment.

Note that the static attributes of a moving object can be updated. The update of static dimensions may change $\mathrm{SK_{sta}}$. Recomputing $\mathrm{SK_{sta}}$ for each update of static values is, however, too costly. Therefore, the main problem of TPBBSE turns to maintain $\mathrm{SK_{sta}}$. Fig. 5 illustrates an example in a 2D static space. The set of points $\{a, b, c\}$ is the original $\mathrm{SK_{sta}}$. For a moving object, update of its static values is implemented as a deletion followed by an insertion. The deletion of a point p_{del} that is not in $\mathrm{SK_{sta}}$ will not change $\mathrm{SK_{sta}}$. If p_{del} is in $\mathrm{SK_{sta}}$, it is deleted from $\mathrm{SK_{sta}}$ and some points that are dominated by p_{del} may become skyline points in $\mathrm{SK_{sta}}$. For example, in Fig. 5, if the point d is deleted, points f and g will become skyline points. The insertion of a point p_{ins} that is dominated by some point in $\mathrm{SK_{sta}}$ will not change $\mathrm{SK_{sta}}$. If p_{ins} is not dominated by any point in $\mathrm{SK_{sta}}$, p_{ins} will be in $\mathrm{SK_{sta}}$, and some points in $\mathrm{SK_{sta}}$ may be dominated by p_{ins} and should be deleted from $\mathrm{SK_{sta}}$. For example, in Fig. 5, if point d is inserted, it will become a skyline point in $\mathrm{SK_{sta}}$ and point a should be deleted from $\mathrm{SK_{sta}}$.

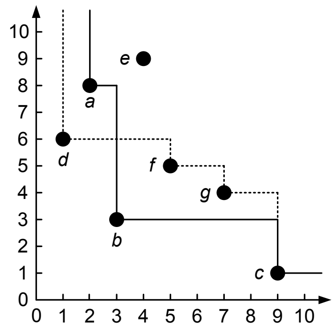

Fig. 5 Change of $\mathrm{SK_{sta}}$

Now, we introduce the solution to maintaining $\mathrm{SK_{sta}}$. In TPBBSE, when an update involving static values arrives, in addition to updating the TPRNS-tree, we do some work to maintain $\mathrm{SK_{sta}}$. For a deleted point p_{del}, we compare it with $\mathrm{SK_{sta}}$. If it is not in $\mathrm{SK}_{\mathrm{sta}}$, nothing needs to be done; else, we delete it from $\mathrm{SK}_{\mathrm{sta}}$. Then, we process a constrained-BBS (C-BBS) algorithm on the static dimensions of the TPRNS-tree. C-BBS is similar to a standard static BBS algorithm except for three differences. First, instead of all dimensions of the index, C-BBS is about only the static dimensions of the TPRNS-tree. Second, C-BBS concerns itself only with the area dominated by p_{del}, and all entries that do not overlap with this area are discarded. Third, the result set is checked for domination with $\mathrm{SK}_{\mathrm{sta}}$. The result points dominated by some point in $\mathrm{SK}_{\mathrm{sta}}$ will be discarded. Obviously, the cost of C-BBS is much lower than that of the standard BBS. For an inserted point p_{ins}, we compare it with $\mathrm{SK}_{\mathrm{sta}}$. If it is dominated by some point in $\mathrm{SK}_{\mathrm{sta}}$, nothing needs to be done; otherwise, we insert p_{ins} in $\mathrm{SK}_{\mathrm{sta}}$ and delete the points dominated by p_{ins} from $\mathrm{SK}_{\mathrm{sta}}$. Note that, for an update, if the insertion point dominates the deletion point on static dimensions, the deletion can be avoided. Also note that, only when the update involves static dimensions, does TPBBSE need to do these. In most applications, however, the static values are seldom updated, and the additional work usually is only to compare the update point with $\mathrm{SK}_{\mathrm{sta}}$. Only for a deletion operation and when the deleted point is in $\mathrm{SK}_{\mathrm{sta}}$, will TPBBSE call a C-BBS, which is not very costly. Therefore, in an overview, the cost of maintaining $\mathrm{SK}_{\mathrm{sta}}$ is rather low.

5 Experiments

We studied the performance of PRISMO via experiments under a variety of settings. We implemented RBBS, TPBBS, and TPBBSE in C++ language, and run the experiments on a personal computer with 2.6 GHz Pentium IV CPU and 1 GB memory, whose operating system is Windows XP Professional. As is commonly done in the study of spatial indexes, the index node size and page size were both set to 4 KB.

5.1 Experimental settings

For the location dimensions in which moving objects move, we used a dataset generator similar to the one in Jensen *et al.* (2004). We used synthetic datasets of moving objects with positions in the space domain of 1000×1000. In most experiments, the datasets were uniform. The initial object positions were generated randomly, so were the moving directions. In other experiments of data distribution, there were some uniform distribution destinations in the space whose number was given. The moving objects moved towards these destinations and chose another when they arrived. The more the destinations, the more uniform the datasets. In the default case, the magnitude of moving speed on each dimension was selected randomly from 0 to 3.

For each NSTP dimension of moving objects, the initial values ranged randomly from 1 to 1000. The coefficients of their dynamic linear functions were chosen randomly from -5 to 5. For the static dimensions of moving objects, the static values ranged from 1 to 10 000. Following the common methodology in the literature of static skyline, we used independent (uniform) and anti-correlated distribution in these static dimensions. In the default case, there were two spatial dimensions, one NSTP attribute dimension, and two static attribute dimensions in the datasets.

The cardinality of the datasets ranged from 1×10^4 to 1×10^6 and the default value was 5×10^5. The horizon (how far the queries can 'see' into the future) in the TPRNS-tree was set to 60 time units, as usually done in the TPR-tree and TPR*-tree. The initial positions were at the indexes creation time, and moving directions of the query points or query ranges were chosen randomly in the space, with the speed ranging from 0 to 3. For each dataset, we executed the same 50 predictive skyline queries and evaluated their average cost. The query time ranged from 1 to the horizon.

5.2 Effect of cardinality

In this set of experiments, we studied the effect of cardinality. We varied the number of moving objects from 1×10^4 to 1×10^6. Recall that, for RBBS, we used two packing methods to repack the index: HS and STR. We first compared these two methods. Table 1 shows the average I/O and CPU time performance for each query at various cardinalities. It can be seen that the performance of RBBS-HS and RBBS-STR was similar. In most cases, RBBS-STR had slightly fewer page accesses and cost slightly more CPU time than RBBS-HS. During the experiment, we observed that RBBS-STR used more page accesses and CPU time for packing the index than RBBS-HS, while fewer page accesses and less CPU

Table 1 Average page accesses and average CPU time of RBBS

Cardinality	Page access (I/Os)				CPU time (s)			
	ID		AD		ID		AD	
	HS	STR	HS	STR	HS	STR	HS	STR
1×10^4	295.5	280.8	305.8	346.5	0.0626	0.0814	0.2190	0.2315
5×10^4	1285.9	1167.1	1520.2	1533.4	0.2311	0.3097	1.0251	1.1096
1×10^5	2440.9	2384.8	3011.8	3180.1	0.4587	0.5925	2.7453	2.0985
2×10^5	4599.1	4586.8	5940.0	6154.5	0.9203	1.2623	5.2152	3.3541
5×10^5	11 061.6	10 913.6	14 399.0	14 355.0	2.3446	3.0975	9.2418	8.0186
1×10^6	21 731.6	21 899.9	28 094.0	27 000.6	5.1484	6.0876	18.5642	15.7659

ID: independent dataset; AD: anti-correlated dataset. HS: Hilbert sort; STR: sort-tile-recursive

time for querying skyline. In addition, the index created by HS had higher space utilization than the one created by STR. For a clearer view, and considering that the number of page accesses is usually a more important metric than CPU time in cost reduction, in the remaining experiments, we used only RBBS-STR to represent the RBBS algorithm.

Recall that, for TPBBS, we use two index structures: TPRNS-tree and TPR*NS-tree. The former uses four cost functions as the TPR-tree, while the latter uses one cost function as the TPR*-tree. We call TPBBS on the TPR*-tree TP*BBS. Fig. 6 shows the average I/O and CPU time performance for each predictive skyline query at various cardinalities, for RBBS-STR, TP*BBS, and TPBBS. As expected, TPBBS and TP*BBS performed much better than RBBS. This is because RBBS has to first rescan the dataset and repack the index of the R-tree with considerable I/O and CPU time cost for each predictive skyline query, and then process a static skyline query. Meanwhile, TPBBS and TP*BBS can directly query without scanning the dataset or rebuilding the index. The difference, however, was not that large. The main reason is that, for each moving object, TPRNS-tree and TPR*NS-tree maintain information of both an MBR and a VBR, while the R-tree of RBBS maintains only an MBR. Therefore, with the same page size, a node of the R-tree in RBBS can contain more entries than the one of the TPRNS-tree and TPR*NS-tree. In addition, the space utilization of the R-tree of RBBS is nearly 100% for static update, while that of the TPRNS-tree and TPR*NS-tree is much less for a dynamic update. Therefore, the number of index nodes of the R-tree of RBBS is smaller than that of the TPRNS-tree and TPR*NS-tree. Furthermore, we used a packing method to minimize the page accesses of rebuilding the R-tree of RBBS. All these help RBBS reduce its total cost of predictive skyline query and shrink the gap between TPBBS/TP*BBS. Through experiments, we observed that for RBBS, because of the high fanout and high utilizaion of the packed R-tree, the maximum heap size of RBBS was smaller than that of TPBBS/TP*BBS. The additional space consumption of the repacking method of RBBS, however, was usually large. Therefore, in summary, the total space consumption of TPBBS/TP*BBS is smaller than that of RBBS.

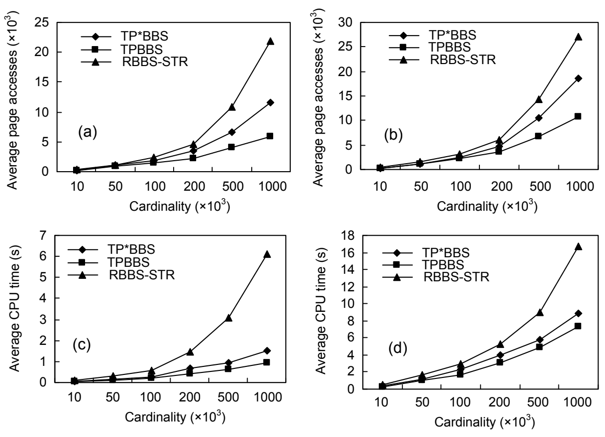

Fig. 6 Page accesses and CPU time vs. cardinality: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset

It can also be seen that TPBBS outperformed TP*BBS. Note that, the TPR-tree is optimized for timestamp queries and uses four cost functions, while the TPR*-tree is optimized for static point interval queries and uses one function. TPRNS-tree of TPBBS is based on the former while TPR*NS-tree of TP*BBS is based on the latter. We can observe that the TPRNS-tree was more suitable for predictive skyline queries, which are also timestamp queries. This means that TPRNS-tree has less expanded MBRs overlapping SKY-LINE(o_q, t_q) at the query time t_q than TPR*NS-tree, and TPRNS-tree has better space locality than TPR*NS-tree for predictive skyline queries. In the following experiments, we use TPRNS-tree instead of TPR*NS-tree.

Now, we compare TPBBS and TPBBSE. Based on the branch-and-bound approach, besides optimal I/O cost, TPBBS can also achieve good memory space consumption and CPU time performance. TPBBSE further improves them, according to the characteristics of predictive skyline queries for moving objects. Through experiments, we observe that, as expected, TPBBS and TPBBSE had the same I/O performance. Table 2 shows the average CPU time and maximum heap size (MHS) for each query at various cardinalities, for TPBBS and TPBBSE. We can see that the memory space consumption of TPBBSE was smaller than that of TPBBS, especially for independent distribution. This is because S_{fil} filters some entries that have no need to be inserted in the heap H. The CPU time performance of TPBBSE was better than that of TPBBS. TPBBSE keeps fewer entries in H and the filtered entries have no need to check for domination later. TPBBSE also avoids some unnecessary domination comparison for entries when they are removed from H. The improvement of CPU time performance, however, was small. This is because some domination comparison between S_{fil} and transition entries consumes additional CPU resource. For a clearer view, from this point forward, we focus on TPBBSE.

5.3 Effect of dimensionality

To study the effect of the dimensionality, we used the datasets with cardinality $N = 5 \times 10^5$ and varied the number of static dimensions from 2 to 5. Consequently, in our experiments, the dimensionality of TPBBSE was from 5 to 8, while that of RBBS was from 4 to 7. Fig. 7 shows the average I/O and CPU time performance for each predictive skyline query at various numbers of static dimensions, for both independent and anti-correlated distributions. As expected, the query cost increased as the dimensionality increased, and the effect was not linear. The higher was the total dimensionality, the faster did the cost increase. This is a common characteristic of the R-based-trees (R-tree and TPR-tree) and all their variants, including the packed R-tree of RBBS and TPRNS-tree of TPBBSE. The effect of dimensionality on CPU time was larger than that on the number of page accesses. This is because with the increase of dimensionality, especially for anti-correlated distribution, the number of the skyline points, the number of times, and cost of domination comparison increase quickly. The effect of dimensionality in TPBBSE was sometimes greater than that in RBBS. This is because the number of dimensions in TPBBSE is more than that in RBBS. RBBS has one dimension of distance, while TPBBSE has two spatial dimensions.

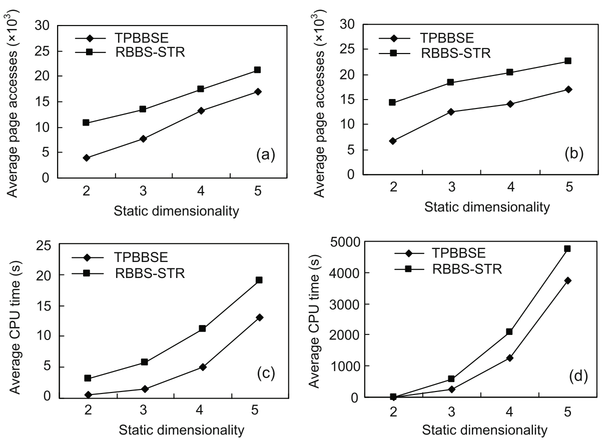

Fig. 7 Page accesses and CPU time vs. dimensionality: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset

5.4 Effect of moving status

In this set of experiments, we studied the effect of moving functions in location dimensions for moving objects. First, we inspected the effect of moving velocities. The default magnitude of moving velocity in each dimension was selected randomly from 0 to 3. In this experiment, however, we increased the velocity in each dimension to 2, 4, 6, and 8 times respectively, and thus the maximum velocity reached 6, 12, 18, and 24 respectively. Fig. 8 shows the average I/O and CPU time performance per predictive skyline query while varying the maximum velocity in each dimension from 3 to 24. The performance of RBBS was almost not affected by the moving velocity. The performance of TPBBSE degraded, however, as the moving velocities increased. This is

Table 2 Average maximum heap size and average CPU time of TPBBS and TPBBSE

Cardinality	Maximum heap size				CPU time (s)			
	ID		AD		ID		AD	
	TPBBS	TPBBSE	TPBBS	TPBBSE	TPBBS	TPBBSE	TPBBS	TPBBSE
1×10^4	210.7	169.6	7695.2	7408.4	0.031	0.029	0.301	0.285
5×10^4	333.8	290.5	19 218.5	18 825.3	0.105	0.091	1.045	0.964
1×10^5	316.4	265.8	26 400.8	25 998.2	0.190	0.163	1.723	1.603
2×10^5	362.9	317.2	34 184.0	33 564.5	0.401	0.362	3.106	2.895
5×10^5	514.6	465.7	44 727.8	44 023.7	0.647	0.586	4.842	4.537
1×10^6	611.7	567.1	64 346.1	63 538.0	0.953	0.872	7.289	6.821

ID: independent dataset; AD: anti-correlated dataset. TPBBS: time-parameterized branch-and-bound skyline; TPBBSE: TPBBS with expansion

because RBBS computes the distances from the moving objects to the query point at the query timestamp t_q when scanning the datasets. It does not affect the cost of the packing or the static skyline queries. For TPBBSE, however, the MBRs in TPRS-tree enlarge by time according to the velocities of objects inside. The faster the objects move, the faster the MBRs enlarge, and the more the MBRs that overlap SKYLINE(o_q, t_q). Therefore, the cost increases with the velocity.

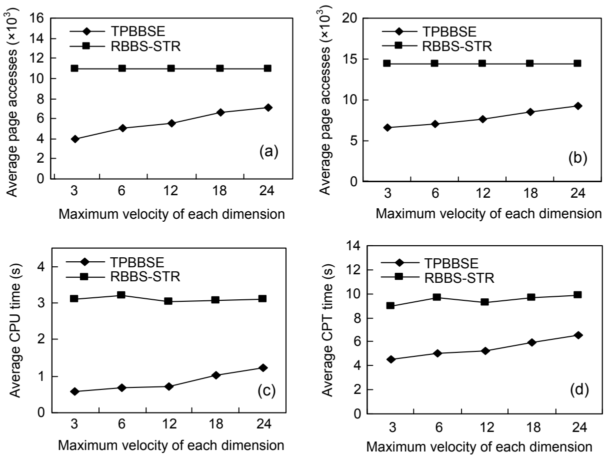

Fig. 8 Page accesses and CPU time vs. velocity: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset

Second, to study the effect of moving objects distribution in location space, we used a method similar to the one in Jensen *et al.* (2004). There are some uniform distributed destinations. The moving objects move towards these destinations and choose another destination when they reach the destinations. The more the destinations, the more uniform the datasets. Fig. 9 shows the average I/O and CPU time performance per predictive skyline query while varying the number of destinations. The distribution of moving objects on spatial dimensions nearly did not affect the query cost for both RBBS and TPBBSE. This is because in an average view, with different query points distributing randomly, the distribution of the moving velocity of objects does not affect the distance domination relationship or the enlargement of MBRs in the TPRNS-tree very much. Another reason is that the distance is only one dimension for predictive skyline queries.

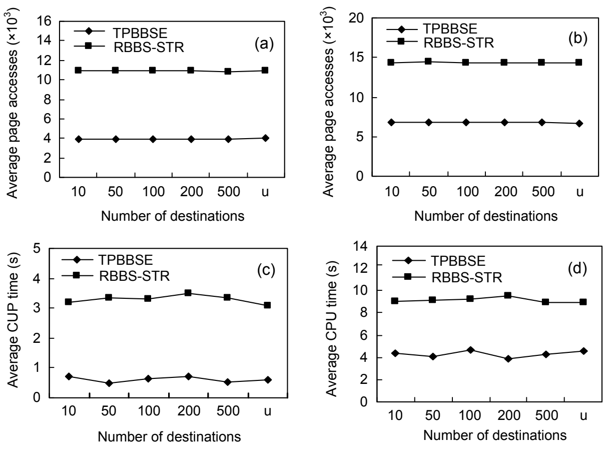

Fig. 9 Page accesses and CPU time vs. the distribution of moving objects: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset. 'u (uniform)' indicates the case where the objects can randomly choose their moving directions

5.5 Range query and subspace query

In this set of experiments, we studied the performance of range queries and subspace queries. Fig. 10

shows the average I/O and CPU time performance per range query with the length of the query varied from 10 to 100 at query time t_q. The performance improved as the length increased. The reason is that a larger query range will contain more moving objects at the query time. The distances from these objects to the query range are zero. These objects become preferable and may dominate more other objects. Therefore, the number of skyline points decreases and the performance improves. Note that, as repacking the R-tree dominates the total query cost, the decrease of the I/O cost is negligible for RBBS.

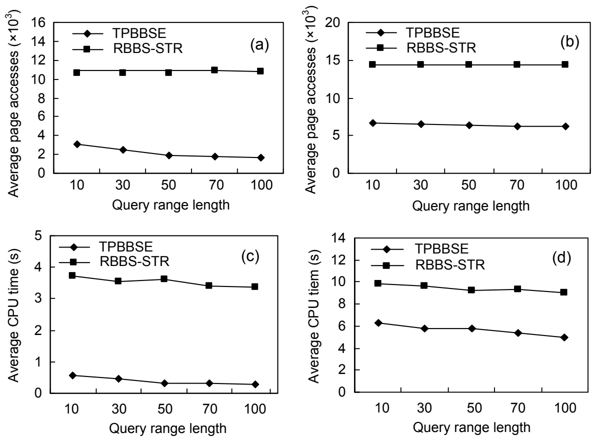

Fig. 10 Page accesses and CPU time vs. query range length: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset

We also studied the performance of subspace queries. Table 3 shows the subspaces we used in these experiments. A, B, and C have three dimensions in target space S', while D has two. C and D both have only one static dimension. Fig. 11 shows the average I/O and CPU time performance of different subspace queries. As expected, the query cost was smaller than the query on all dimensions, and the most important effective issue of performance was the number of dimensions. We can also see that, in TPBBSE, for independent distribution, the effect of the distance dimension was greater than that of the NSTP attribute. And, the effect of the NSTP dimension was greater than that of static attributes. In contrast, for anti-correlated distribution, the most important dimensions were the static ones. In RBBS, due to the repacking of the R-tree per query, the performance improvement was much slighter compared with TPBBSE, and the difference between the effect of different dimensions was small. For anti-correlated distribution, however, the effect of static attributes was greater than the others.

Table 3 Subspaces used in the experiments

Name	Subspace
A	1 NSTP attribute + 2 static attributes
B	distance + 2 static attributes
C	distance + 1 NSTP attribute + 1 static attribute
D	1 NSTP attribute + 1 static attribute

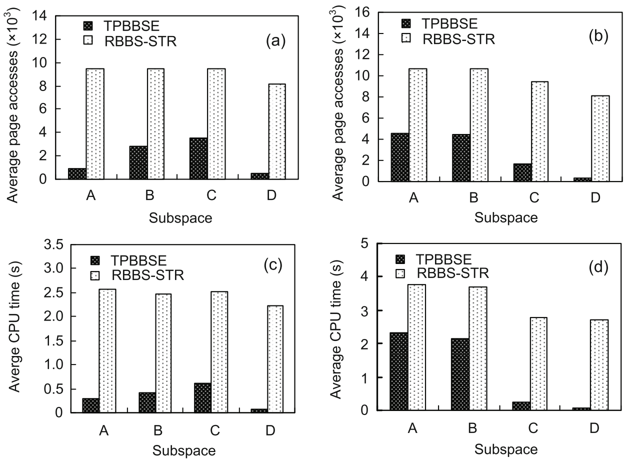

Fig. 11 Page accesses and CPU time in different subspaces: (a) I/Os on an independent dataset; (b) I/Os on an anti-correlated dataset; (c) CPU time on an independent dataset; (d) CPU time on an anti-correlated dataset

6 Conclusions and future work

In this paper, we propose a novel framework called PRISMO for processing predictive skyline queries for moving objects. The domination dimensions over moving objects include the distance, the non-spatial time-parameterized attributes, and static ones. A predictive skyline query returns the skyline of moving objects at some future time. We present two schemes to process predictive skyline queries, namely RBBS and TPBBS. Given a predictive skyline query, RBBS rescans the dataset and rebuilds an R-tree using a packing method, while TPBBS processes the proposed R-tree-based index called TPRNS-tree. Both schemes are I/O optimal, and hence the I/O cost depends on the indexes they use. We implement two packing methods of the R-tree for RBBS, and two indexes for TPBBS. The basic TPBBS is further extended to TPBBSE for reducing the memory space consumption and CPU time, according to the characteristic analysis of predictive skyline queries for moving objects and TPBBS. In addition, we present our solutions to predictive skyline range queries and subspace skyline queries. We evaluate the proposed algorithms through extensive experiments. In future work, we will investigate predictive skyline for moving objects in a future time range.

References

Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B., 1990. The R*-Tree: an Efficient and Robust Access Method for Points and Rectangles. SIGMOD, p.322-331. [doi:10.1145/93597.98741]

Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S., 2006. Nearest neighbor and reverse nearest neighbor queries for moving objects. *VLDB J.*, **15**(3):229-249. [doi:10.1007/s00778-005-0166-4]

Borzsonyi, S., Kossmann, D., Stocker, K., 2001. The Skyline Operator. ICDE, p.421-430. [doi:10.1109/ICDE.2001.914855]

Chen, N., Shou, L.D., Chen, G., Dong, J.X., 2008. Adaptive indexing of moving objects with highly variable update frequencies. *J. Comput. Sci. Technol.*, **23**(6):998-1014. [doi:10.1007/s11390-008-9185-0]

Chen, N., Shou, L.D., Chen, G., Chen, K., Gao, Y.J., 2010. Bs-Tree: a Self-Tuning Index of Moving Objects. DASFAA, p.1-16.

Chomicki, J., Godfrey, P., Gryz, J., Liang, D.M., 2003. Skyline with Presorting. ICDE, p.717-816.

Cui, B., Chen, L.J., Xu, L.H., Lu, H., Song, G.J., Xu, Q.Q., 2009. Efficient skyline computation in structured peer-to-peer systems. *IEEE Trans. Knowl. Data Eng.*, **21**(7):1059-1072. [doi:10.1109/TKDE.2008.235]

Gao, Y.J., Chen, G.C., Chen, L., Chen, C., 2006. An I/O Optimal and Scalable Skyline Query Algorithm. BNCOD, p.127-139.

Godfrey, P., Shipley, R., Gryz, J., 2005. Maximal Vector Computation in Large Data Sets. VLDB, p.229-240.

Guttman, A., 1984. R-Trees: a Dynamic Index Structure for Spatial Searching. SIGMOD, p.47-57. [doi:10.1145/602259.602266]

Hjaltason, G.R., Samet, H., 1999. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, **24**(2):265-318. [doi:10.1145/320248.320255]

Huang, Z.Y., Lu, H., Ooi, B.C., Tung, A.K.H., 2006. Continuous skyline queries for moving objects. *IEEE Trans. Knowl. Data Eng.*, **18**(12):1645-1658. [doi:10.1109/TKDE.2006.185]

Jensen, C.S., Lin, D., Ooi, B.C., 2004. Query and Update Efficient B$^+$-Tree Based Indexing of Moving Objects. VLDB, p.768-779. [doi:10.1016/B978-012088469-8/50068-1]

Kamel, I., Faloutsos, C., 1993. On Packing R-trees. CIKM, p.490-499. [doi:10.1145/170088.170403]

Kossmann, D., Ramsak, F., Rost, S., 2002. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. VLDB, p.275-286.

Lee, K.C.K., Zheng, B.H., Li, H.J., Lee, W.C., 2007. Approaching the Skyline in Z Order. VLDB, p.279-290.

Leutenegger, S.T., Lopez, M.A., Edgington, J., 1997. STR: a Simple and Efficient Algorithm for R-Tree Packing. ICDE, p.497-506. [doi:10.1109/ICDE.1997.582015]

Lin, B., Mokhtar, H., Pelaez-Aguilera, R., Su, J.W., 2003. Querying Moving Objects with Uncertainty. Vehicular Technology Conf., **4**:2783-2787.

Papadias, D., Tao, Y.F., Fu, G., Seeger, B., 2005. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, **30**(1):41-82. [doi:10.1145/1061318.1061320]

Pei, J., Fu, A.W.C., Lin, X.M., Wang, H.X., 2007. Computing Compressed Multidimensional Skyline Cubes Efficiently. ICDE, p.96-105. [doi:10.1109/ICDE.2007.367855]

Saltenis, S., Jensen, C.S., 2002. Indexing of Moving Objects for Location-Based Services. ICDE, p.463-472. [doi:10.1109/ICDE.2002.994759]

Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A., 2000. Indexing the Positions of Continuously Moving Objects. SIGMOD, p.331-342. [doi:10.1145/342009.335427]

Tan, K.L., Eng, P.K., Ooi, B.C., 2001. Efficient Progressive Skyline Computation. VLDB, p.301-310.

Tao, Y.F., Papadias, D., Sun, J.M., 2003. The TPR*-Tree: an Optimized Spatio-Temporal Access Method for Predictive Queries. VLDB, p.790-801. [doi:10.1016/B978-012722442-8/50075-6]

Tao, Y.F., Xiao, X.K., Pei, J., 2006. SUBSKY: Efficient Computation of Skylines in Subspaces. ICDE, p.65. [doi:10.1109/ICDE.2006.149]

Tzoumas, K., Yiu, M.L., Jensen, C.S., 2009. Workload-Aware Indexing of Continuously Moving Objects. PVLDB, p.1186-1197.

Vlachou, A., Doulkeridis, C., Kotidis, Y., 2008. Angle-Based Space Partitioning for Efficient Parallel Skyline Computation. SIGMOD, p.227-238. [doi:10.1145/1376616.1376642]

Wang, S.Y., Vu, Q.H., Ooi, B.C., Tung, A.K.H., Xu, L.Z., 2009. Skyframe: a framework for skyline query processing in peer-to-peer systems. *VLDB J.*, **18**(1):345-362. [doi:10.1007/s00778-008-0104-3]

Yuan, Y.D., Lin, X.M., Liu, Q., Wang, W., Yu, J.X., Zhang, Q., 2005. Efficient Computation of the Skyline Cube. VLDB, p.241-252.