Indexing the Past, Present, and Anticipated Future Positions of Moving Objects

MINDAUGAS PELANIS, SIMONAS ŠALTENIS, and CHRISTIAN S. JENSEN
Aalborg University, Denmark

With the proliferation of wireless communications and geo-positioning, e-services are envisioned that exploit the positions of a set of continuously moving users to provide context-aware functionality to each individual user. Because advances in disk capacities continue to outperform Moore's Law, it becomes increasingly feasible to store online all the position information obtained from the moving e-service users. With the much slower advances in I/O speeds and many concurrent users, indexing techniques are of the essence in this scenario.

Existing indexing techniques come in two forms. Some techniques capture the position of an object up until the time of the most recent position sample, while other techniques represent an object's position as a constant or linear function of time and capture the position from the current time and into the (near) future. This article offers an indexing technique capable of capturing the positions of moving objects at all points in time. The index substantially modifies partial persistence techniques, which support transaction time, to support valid time for monitoring applications. The performance of a timeslice query is independent of the number of past position samples stored for an object. No existing indices exist with these characteristics.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Continuous variable, indexing, moving object, polyline, querying, trajectory, update

1. INTRODUCTION

Continued advances in hardware technologies combine to provide the enabling foundation for mobile e-services. These advances include the miniaturization and the general improvement in performance of electronics, and it includes

This research was supported in part by a grant from the Nykredit Corporation and by grant 216 from the Danish National Centre for IT Research.

M. Pelanis is currently a Ph.D candidate at Vilnius University, Lithuania. In addition to his primary affiliation, C. S. Jensen is also an adjunct professor in the Department of Technology, Agder University College, Norway.
Authors' address: C. S. Jensen, Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, 9220 Aalborg, Ost, Denmark; email: csj@cs.aau.dk.

the generally improved performance/price ratio. Perhaps most importantly, wireless communication and positioning technologies such as GPS are finding increasingly widespread use. Positioning is important for mobile e-services because these must be context aware, and the positions of the service users are an important aspect of context.

These developments pave the way for a range of qualitatively new types of e-services, which either make little sense or are of limited interest in the traditional context of fixed-location, desktop-based computing. Such services encompass traffic coordination and management, tourist services, safety-related services, and location-based games that merge virtual and physical spaces.

In these e-services, moving objects disclose their positional information (position, speed, velocity, etc.) to the services that, in turn, use this and other information to provide specific functionality. Our focus is on location-enabled services that rely on access to the positions of moving objects. Due to the volumes of data, the data must be assumed to be disk resident; and to obtain adequate query performance, some form of indexing must be employed.

The aim of indexing is to make it possible for multiple users to concurrently and efficiently retrieve desired data from very large databases. Indexing techniques are becoming increasingly important because rapidly increasing volumes of data may be stored, while the improvement in the rate of the transfer of data between disk and main memory cannot keep pace.

In this article, we propose what we believe is the first single index that is able to accurately capture the past, present, and (near) future positions of moving objects. Positions are obtained via sampling. The position of an object in between samples is computed via linear interpolation, and the position since the last sample is given by a linear function.

Previous proposals for indexing moving objects either support only the past positions, up until the most recent position sample (e.g., Beckmann et al. [1990]; Hadjieleftheriou et al. [2002]; Kollios et al. [2001]; Kumar et al. [1998]; Pfoser et al. [2000]; Porkaew et al. [2001]; Tao and Papadias [2001]), or they support only the positions from the current time and into the future (e.g., Agarwal et al. [2000]; Agarwal and Har-Peled [2001]; Jensen et al. [2004]; Patel et al. [2004]; Procopiuc et al. [2002]; Šaltenis et al. [2000]; Šaltenis and Jensen [2002]; Tayeb et al. [1998]; Tao et al. [2003]). No index combines these capabilities with the exception of two recent proposals. A proposal by Papadias et al. [2004] addresses approximate query answering, and a proposal by Lin et al. [2005] indexes broken polylines.

Further, simply using two existing indices, one of each type, does not solve the general indexing problem: for any object, its position for times in between the time of the most recent sample and the current time (CT) cannot be indexed readily with existing techniques. In terms of Figure 1, which shows the trajectories of two objects moving in one-dimensional space, the first type of index supports the solid parts, and the second type supports the dashed parts—no index supports all three parts (solid, dash-dotted, dashed).

In developing such an index, we apply a substantially extended notion of partial persistence to the TPR-tree [Šaltenis et al. 2000]. The TPR-tree supports the querying of the current and anticipated future positions of moving objects,

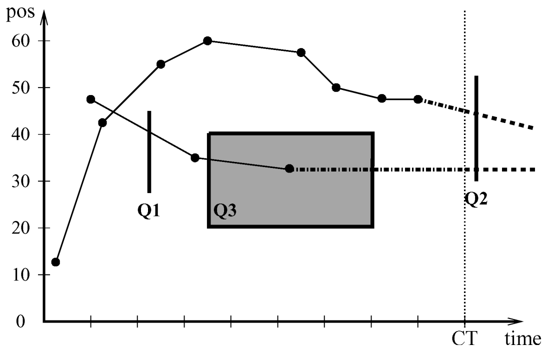


Fig. 1. Querying the positions of moving objects.

the positions represented by linear functions. The resulting index, the R^{PPF}-tree (past, present, and future), captures and supports the efficient querying of the past positions as well. Two key innovations make this possible.

The first is so-called optimized and double time-parameterized bounding rectangles where the latter come in two variants. These are novel kinds of bounding rectangles designed specifically for the partial persistence setting where the conventional bounding rectangles of the TPR-tree are inapplicable.

Second, as the R^{PPF}-tree captures valid time and partial persistence supports transaction time, novel techniques have been developed that allow for the correction of the past in the partial persistence setting. Because position data arrive in time order, it is necessary only to be able to correct the part of the current position prediction that covers the period since the last sample was received. However, even this is quite challenging in the partial persistence framework due to the occurrence of so-called time splits.

All structures and algorithms presented have been implemented for objects moving in one-, two- and three-dimensional space. When explaining the underlying concepts, it is often beneficial to consider only one-dimensional or two-dimensional space, and we most often use two-dimensional terminology.

Performance experiments with the article's proposals study the properties of the different kinds of bounding rectangles, consider update performance, and compare the R^{PPF}-tree with the TPR-tree and an approach that uses two index structures.

While we have chosen moving objects as the concrete motivation for this work, it should be noted that positional information from moving objects is simply a specific instance of the more general sensor data management problem where sensors sample a continuous process. The samples we receive are (position, velocity) pairs for objects moving continuously in from one- to three-dimensional space. We thus expect the indexing technique to be applicable more generally to a broad range of sensor data management settings.

The next section describes the functionality to be supported by the proposed indexing technique. In Section 3, we then consider related work, covering the two classes of existing indices and describing in more detail the TPR-tree. Section 4 considers an alternative approach to solving the indexing problem where separate indices are maintained for the past versus the present and future.

Then a section follows that describes the structure of the R^{PPF}-tree and the algorithms that maintain the index structure under updates and enable querying it. Next, Section 6 presents an analytical study of the cost associated with the correction of the most recently recorded position prediction during updates. Section 7 proceeds to report on empirical studies of performance-related properties of the indexing technique, and Section 8 summarizes and points out future research directions.

2. INDEX FUNCTIONALITY

We proceed to briefly describe the general setting for the indexing problem addressed and then describe the data and queries accommodated by the index.

2.1 Problem Setting

The problem setting aims to concisely capture the general context of the indexing problem. At the core of the problem setting is a set of so-called moving objects that are capable of continuous movement [Sistla et al. 1997]. These objects move in one-, two-, or three-dimensional space.

Next, a set of e-services, with an associated database, are available to the moving objects. The moving objects communicate wirelessly with the services. Further, the moving objects report their movement information, including, and most prominently, their current position and velocity, to the services [Sistla et al. 1997; Wolfson et al. 1998]. This capability is achieved by means of one of a range of geo-location technologies. The services use the database for recording the past, current, and anticipated future movement of each object.

As we are effectively sampling continuous processes, the records of the moving objects' locations are inherently imprecise [Wolfson et al. 1998, 1999]. Different services require movement information with different minimum precisions for them to work. For example, a weather information service requires user positions with very low precision, while an advanced location-based game, where the participants interact with geo-located, virtual objects, requires high precision. Stated in general terms, the highest precision that may be obtained is that offered by the geo-location technology used.

We assume that a required precision is given by the services under consideration. A moving object is aware of the movement information kept for it in the database. An object then issues an update to the database when its actual position deviates by more than the required precision from the position inferred from the positional information in the database (see, e.g., Čivilis et al. [2004, 2005] for recent works that summarize existing contributions).

The workload experienced by the database then consists of a sequence of updates intermixed with queries. The amount of updates is dependent on factors such as the number of objects, the required precision, the agility of the objects, and the service's representation of the objects' movements.

2.2 Data and Queries

The representations used for the moving-object positions and the frequencies of updates needed to maintain a reasonable precision of the moving-object positions are closely related.

Studies of real positional information obtained from GPS receivers installed in cars show that representing positions as linear functions of time reduces the numbers of updates needed to maintain a reasonable precision by as much as a factor of three in comparison to using constant functions [Čivilis et al. 2004, 2005]. Linear functions are thus much better than constant functions.

The use of more complex approximations seems less appropriate for indexing purposes. The information needed to derive linear approximations is readily available which may not be the case for, for example, higher-order functions. Also the use of complex approximations, which are less compact than linear ones, reduces fanout when stored as key values in index nodes.

Thus, because it is important to reduce the number of updates needed, because linear functions are easy to determine, and because linear functions are still simple and compact and incur low computational overhead, we represent the current and (near) future positions of objects as linear functions of time and represent past positions by linear interpolation between consecutive position samples.

When, at time t_u, an object moving in m-dimensional space communicates with the database to update its positional information, it reports its current position ($\bar{x}(t_u) = (x_1(t_u), \ldots, x_m(t_u))$) and its current velocity vector ($\bar{v}(t_u) = (v_1(t_u), \ldots, v_m(t_u))$). If the position of this object is queried at some later time t, but before the next update, the database computes the expected position of the object using linear extrapolation: $\bar{x}(t) = \bar{x}(t_u) + \bar{v}(t_u) \cdot (t - t_u)$. Note that t can be a point that is larger than the current time. In this way, tentative near-future positions of objects can be queried.

If all positional information reported by an object is recorded in the database, the past movement of the object can be reconstructed. Observe though, that in order to achieve a continuous approximation of the past positions of an object, on each update, the velocity vector recorded in the previous update will most likely have to be updated. More specifically, if the previous update occurred at time t'_u and the new update occurs at time t_u, the previous velocity vector $\bar{v}(t'_u)$ should be set to $(\bar{x}(t_u) - \bar{x}(t'_u))/(t_u - t'_u)$, where $\bar{x}(t_u)$ is the just-recorded position of the object.

The bold curve in Figure 2 represents the trajectory of an object moving in one-dimensional space from some time $u1$ to the current time. The object thus starts at position 82 at time $u1$ and moves downward. It first moves quickly, then slows down, then speeds up, and finally slows down. The object could be a car with its position being the distance traveled along a highway.

Times $u1, \ldots, u5$ are update times, and CT is the current time. The figure demonstrates how a polyline approximating the past trajectory of an object is constructed by modifying the reported velocity vector of the object when the next update is processed. For example, consider the situation at time $u3$. The object's position since time $u2$ has so far been given by the linear function that is represented in the figure by the dashed line segment that starts at $u2$ and ends at $u3$. At $u3$, an update occurs because the position given by the function deviates from the real position by more than some threshold. When the update occurs, the position between times $u2$ and $u3$ is changed to become the solid line segment that starts at $u2$ and ends at $u3$, and the position after time $u3$

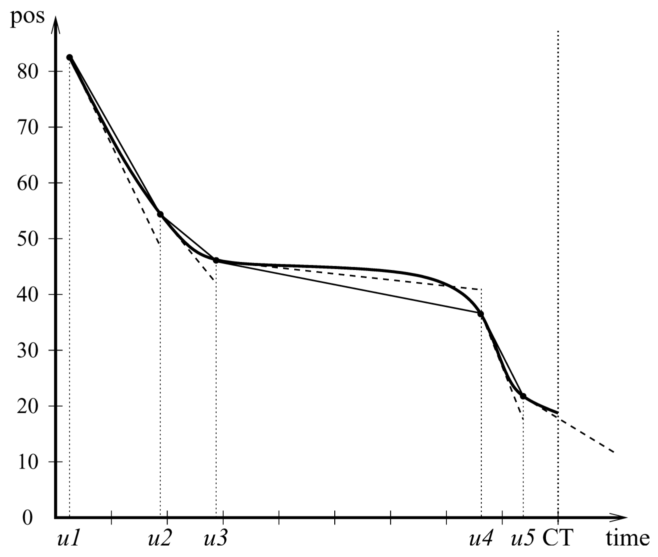


Fig. 2. Trajectory of a one-dimensional moving object.

is given by a new linear function of time, again represented by a dashed line segment in the figure.

Note that the figure is consistent with the update policy adopted—the object updates its position when its actual position deviates by some threshold from the position predicted by the database (shown as dashed line segments). Updates are more frequent when the object is changing its velocity vector. For example, deceleration of the object caused the updates $u2$ and $u3$.

Using the trajectories of two one-dimensional moving objects, Figure 1 exemplifies the two fundamental types of queries we aim to support. Let R be a d-dimensional rectangle and $t^{\vdash}$ and $t^{\dashv}$ ($t^{\vdash} < t^{\dashv}$) be two time points. A *window* query $Q = (R, t^{\vdash}, t^{\dashv})$ specifies the $(m + 1)$-dimensional rectangle obtained by adding the time extent given by $t^{\vdash}$ and $t^{\dashv}$ to R (see $Q3$ in Figure 1). A *timeslice* query ($Q1$ and $Q2$) is a special case of a window query when $t^{\vdash} = t^{\dashv}$. Notice that $t^{\vdash}$ and $t^{\dashv}$ can in principle be any past, present, or future time points. In this article, we focus on timeslice queries.

3. RELATED WORK

We first describe existing indices for indexing the positions of moving objects from some past time until the time of the most recent position sample. We then consider techniques that index the present and anticipated future positions of moving objects. We end by describing the TPR-tree in more detail.

3.1 Past Position Indices

Straightforward use of the R-tree for indexing the evolutions of moving objects has been suggested by several authors. The typical, generic situation is that the evolution of an object is given by a polyline, that is, a sequence of connected line segments. The R-tree is easily capable of indexing line segments, but there also seems to be consensus that the R-tree is not well suited for this problem (e.g., Kollios et al. [2001]; Kolovson and Stonebraker [1991]; Kumar et al. [1998]). Pfoser et al. [2000] suggest two variations of the R-tree for polyline indexing, the TB-tree and the STR-tree. Both attempt, to varying degrees, to group together segments from the same polyline with the goal of answering queries that retrieve object evolutions consisting of multiple segments. They index positions for an object only up to the time of the most recent sample.

Porkaew et al. [2001] suggest allowing one or more line segments of a polyline to extend into the future. These segments then represent predictions. They also suggest using a standard R-tree for the indexing of the segments. When predictions need to be updated, new segments are inserted into the index which implies that multiple segments may exist in the index that specify the position of an object at the same time point. It is suggested that this situation be dealt with in a separate postprocessing step. In our proposal, recorded predictions are adjusted to be consistent when actual position samples are received so that, at any time, a single, accurate position is indexed for each object.

The performance of timeslice queries on an R-tree that indexes line segments decreases as the number of updates grows. The R-tree variant proposed by Cai and Revesz [2000] has this general problem. In our quite different proposal, timeslice performance is unaffected by the number of updates.

Kumar et al. [1998] apply partial persistence to the R-tree, with the objective of supporting the transaction time aspect of temporal data. Transaction time records the history of the current database state which changes only in discrete, step-wise constant fashion and does not involve prediction. As this is quite unlike the continuous valid time aspect of moving objects considered here, the resulting index falls short in meeting our needs. However, the R^{PPF}-tree builds on this work in that it applies partial persistence techniques to an R-tree extension in order to solve a problem that is more challenging to partial persistence.

Two other works [Hadjieleftheriou et al. 2002; Kollios et al. 2001] assume a static database of object evolutions and consider the partitioning of these evolutions into smaller time intervals with the goal being of reducing dead space when the data is indexed with the partially persistent R-tree. When the data set is dynamic as in our case, these solutions are not applicable.

Tao and Papadias [2001] index the same data with both a standard R-tree and a variant of the partially persistent R-tree. The combined index, called the MV3R-tree, is capable of indexing past trajectory data. To address a similar problem, Chaka et al. [2003] propose SETI, a two-level indexing structure which separates indexing of the spatial and temporal dimensions. These indices capture only the positions of an object up until the time of the most recent update and do not support the so-called corrections that are needed to maintain a continuous approximation of past positions as covered in Section 2.2. They thus do not solve the more general problem considered here.

Song and Roussopoulos [2001, 2003] address the problem of tracking and recording positions of moving objects using hashing. The space is subdivided into zones, and their approach works at the granularity of zones. Future queries are not supported because velocities are not recorded.

3.2 Indices Supporting Present and Future Queries

A number of approaches for indexing of the current and predicted future positions of moving points exist that might also be considered as candidates for extensions that additionally index past positions. Tayeb et al. [1998] use PMR-Quadtrees, Kollios et al. [1999] and Papadopoulos et al. [2002] employ the so-called dual data transformation, Agarwal et al. [2000] use the ideas of kinetic data structures [Basch et al. 1997], and Chon et al. [2002] use a space-time grid. While each of these has its strong points, each one also exhibits limitations in relation to our objective of obtaining a practical index that works for objects moving in one, two, and three dimensions.

Sun et al. [2004] propose a method for approximate query answering based on multidimensional histograms. In contrast to our proposal, no individual positions of objects are indexed. Instead, a histogram representing the distribution of the current positions of moving objects is maintained in main memory. The buckets of the histogram corresponding to the recent past are also kept in main memory for some time before being migrated to disk. To answer future queries, instead of applying linear extrapolation using velocity vectors (see Section 2.2), a stochastic method is used to predict the future based on the recent past. Although the discussed approach combines indexing of the past, present, and future positions of moving objects in one data structure, it addresses a different problem than the one considered in this article.

Three proposals for indexing the current and the predicted future positions of objects build on the ideas of the TPR-tree. Procopiuc et al. [2002] propose the STAR-tree. This index seems to be best suited for workloads with infrequent updates. With the objective of enabling efficient deletion of data that is no longer valid, Šaltenis and Jensen [2002] propose the R^{EXP}-tree which extends the TPR-tree to accommodate data with so-called expiration times associated that indicate when the data is no longer considered valid. Tao et al. [2003] adopt assumptions about the query workload that differ slightly from those underlying the TPR-tree. This leads to the use of a new measure when grouping objects into index tree nodes. Due to a number of modifications, the algorithms of the proposed index, called the TPR*-tree, are more complex than the algorithms of the TPR-tree. We have thus chosen to build on the TPR-tree, described next.

Finally, two recent proposals represent the current and future positions of moving objects using linear functions and use the dual data transformation technique for indexing these. Both proposals assume that each linear function is updated within a specified, global maximum duration of time. They also assume a global maximum velocity for all objects. Patel et al. [2004] index the points that result from the dual data transformation by means of two quadtrees, each covering a part of the recent past. Updates apply to the most recent quadtree. As time passes, the old quadtree becomes empty, and a new one is created. The proposal by Jensen et al. [2004] also partitions the recent past, but allows a number of so-called phases that are a multiple of two. Each phase has an associated B-tree. Object positions are represented as one-dimensional points using a transformation technique that involves a space-filling curve. Updates apply to the most recent phase. As time passes, old phases become empty,

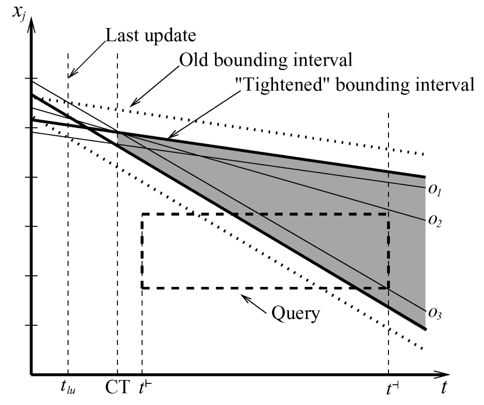


Fig. 3. A bounding interval and a query in the TPR-tree.

and new phases emerge. A recent extension of this approach, the BB^x-tree [Lin et al. 2005], retains the old phases so that past, present, and future positions of moving objects are indexed, but the proposal does not perform corrections of trajectory segments as dicussed in Section 2.2. Thus, if the index is used in an online setting, disconnected trajectories of objects are indexed (see the dashed line segments in Figure 2).

It is unclear whether it is possible to extend these approaches to solve efficiently the problem considered in this article, and we have chosen to build on the TPR-tree that does not rely on the same strict assumptions about the data.

3.3 The TPR-Tree

Based on the R*-tree, the TPR-tree indexes the current and future positions of objects that move in one, two, or three dimensions. While the index employs the basic structure and algorithms of the R*-tree, the indexed objects as well as the bounding rectangles in non-leaf entries are augmented with velocity vectors. This way, bounding rectangles are time-parameterized—they can be computed for different time points. The speeds of the edges of these time-parameterized bounding rectangles (TPBRs) are chosen so that the enclosed moving objects or rectangles remain inside the rectangles at all future times. Section 5.2.2 provides more details on how bounding rectangles of the TPR-tree are computed. Here we present the underlying intuition.

Figure 3 shows the trajectories (thin solid lines) of three moving points in one-dimensional space (i.e., $m = 1$) together with their one-dimensional bounding rectangle, that is, a bounding interval (dotted lines). The coordinates of the bounding interval are set so that the interval is minimum when it is computed (at time t_{lu} in the figure). To see how the speeds of the upper and the lower bounds of the bounding interval are computed, first observe that the objects o_1, o_2, and o_3 all have negative speeds (they move downwards, in the negative direction along the x-axis). Then, the upper bound of the bounding interval is set to move with a speed equal to the maximum speed of the three bounded objects (the speed of object o_1), and the lower bound is set to move with a speed equal to the minimum speed of the three bounded objects (the speed of object o_3). If $m > 1$, the described procedure is repeated for each dimension.

The figure shows that answering a window query in the TPR-tree involves checking for an intersection between an $(m + 1)$-dimensional rectangle and a trapeoid, that is, the query and a bounding rectangle. If the query is a timeslice query, the query is answered in the same way as in the regular R-tree except that, for each encountered TPBR in the tree traversal, its geometry has to be computed for the time of query.

In addition to its use of time-parameterized bounding rectangles, the TPR-tree differs from the R*-tree in how its insertion algorithms group data points into nodes. The R*-tree aims to minimize the areas, overlaps, and margins of bounding rectangles when data objects are inserted into the index. To take into account the temporal evolution of these properties, the TPR-tree uses their integrals over time. The area of the shaded region in Figure 3 illustrates an integral across time of the length of the bounding interval. This use of integrals in the algorithms allows the index to systematically take the objects' velocities as well as their current positions into account when grouping them.

The bounding intervals in Figure 3 illustrate that bounding intervals generally are minimum only at the time they are computed. At later times, a bounding interval is typically larger than the truly minimum bounding interval. To be able to efficiently answer current-time and near-future queries, the TPR-tree algorithms recompute bounding rectangles every time a tree node bounded by a rectangle is updated (compare the bounding intervals at times t_{lu} and CT in the figure). Performance experiments show that this recomputation, which we term tightening, is essential in order to achieve query performance that does not degrade with time [Šaltenis et al. 2000]. Note also that tightening produces time-parameterized bounding rectangles (TPBRs) that are not bounding prior to the current time (see the tightened bounding interval in Figure 3). Thus, a TPR-tree is not valid before CT.

4. SEPARATE INDEXING OF THE PAST AND PRESENT AND FUTURE

This article presents a single index that captures the positions of moving objects at all points in time and supports timeslice queries for all points in time. Here, we briefly consider the desirability of possible alternative designs that involve two separate indices: one for all past times, and one for the present and future times.

As described in Section 3.2, several proposals for the latter type of indexing exist already (e.g., Agarwal et al. [2000]; Agarwal and Har-Peled [2001]; Jensen et al. [2004]; Patel et al. [2004]; Procopiuc et al. [2002]; Šaltenis et al. [2000]; Šaltenis and Jensen [2002]; Tayeb et al. [1998]; Tao et al. [2003]) and can be reused. Thus, if querying only the present and future positions is of interest, one of these existing proposals should simply be used. In what follows, we consequently assume that at least the querying of past positions is to be supported.

For the indexing of only the past positions of objects (i.e., disregarding the present and future positions), we might consider using some type of

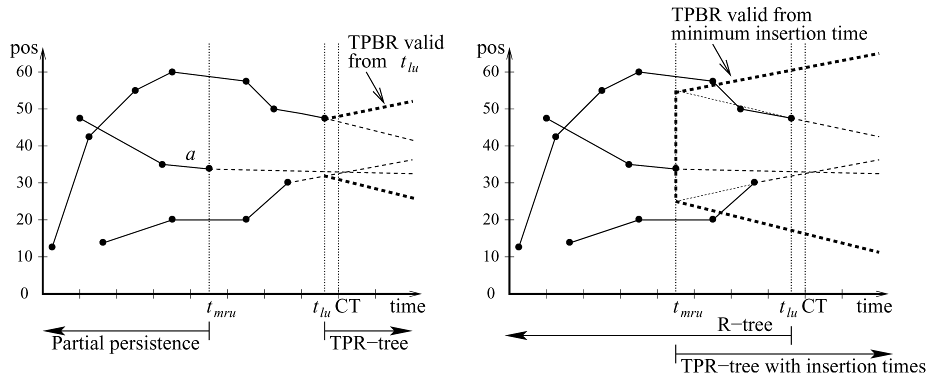


Fig. 4. Indexing the past separately from the present and future.

partially persistent data structure (PP-structure) or using some variant of the R-tree.

The use of a PP-structure is problematic because the history of the position of an object o can only be recorded up until the time of the most recent update of the object, t^o_{mru}. Given a population $\mathcal{O}$ of moving objects, a PP-structure therefore captures the positions of an entire population of moving objects only up until the time t_{mru} of the most recent update of the least recently updated object, that is, time $t_{mru} = \min(\{t^o_{mru} \mid o \in \mathcal{O}\})$.

This is so because insertions and deletions into a PP-structure must occur in chronological order, and we cannot insert the last, open-ended segment of the trajectory of an object into a PP-structure until the final geometry of this segment is known which happens only when it is logically deleted at the next update of the object. Only at this time is it guaranteed that the correct velocity of the last segment can be set (recall Section 2.2).

The left part of Figure 4 visualizes t_{mru}. The final geometry of the last segment (dashed line) of object a is not yet known. Thus the last update of a is not finished; the insertion of a new segment is pending. No operations can be processed by the partially persistent data structure until object a is again updated.

The figure also shows the last update time $t_{lu} = \max(\{t^o_{mru} \mid o \in \mathcal{O}\})$. After this time, an index for the present and future positions of moving objects, such as the TPR-tree, can be used. Observe that when a large number of objects are being indexed, it is likely that $t_{lu} \approx$ CT.

To bridge the time gap shown in the left part of Figure 4, all updates to the database after t_{mru} must be stored in what we term a near-past structure (NP-structure). This data structure can become arbitrarily large, since arbitrarily many updates of each object can occur in between t_{mru} and the current time. Also, this data structure (alternatively, another separate data structure) must provide the functionality of a priority queue on all update times stored in it. This is necessary to enable the migration of updates in chronological order from the NP-structure to the PP-structure.

This analysis demonstrates that, even if only the past positions of objects need to be indexed, a partially persistent index (that disregards the present and future positions) does not provide a straightforward solution that is simpler than this article's more general proposal.

As an alternative to the use of a PP-structure, we may store all past trajectories up to the last update of each object in an R-tree-type index (e.g., Pfoser et al. [2000]). Such an index is built on the line segments that make up trajectories, and it stores complete information up until time t_{mru}.

The last, open-ended parts of the trajectories (from the last update until the current time—dashed line segments in Figure 4) could be stored in a modified TPR-tree or a TPR-tree-type structure. Normally, these indices only capture positions from the current time and forwards. As explained at the end of the previous section, this is because TPBRs of the tree are not valid prior to t_{lu}. A modification of the TPR-tree is therefore needed. The modified TPR-tree must store insertion times in all entries. For a TPBR in such a tree, the insertion time is equal to the minimum of the insertion times of the bounded entries. This is illustrated on the right side of Figure 4 where the three dashed segments are bounded by a TPBR with an insertion time. Note that the TPBR's insertion time, and not CT, is the time when the TPBR is the minimum bounding rectangle of the linear functions corresponding to the bounded segments.

As mentioned in the previous section, tightening of TPBRs—an essential element of the TPR-tree's algorithms—produces TPBRs that are not bounding prior to the current time (or, more precisely, t_{lu}). Thus, tightening cannot be performed in the modified TPR-tree with insertion times, reducing its querying and update performance. Figure 4 exemplifies the larger TPBR (to the right) in the modified TPR-tree in comparison to the corresponding TPBR of the TPR-tree (to the left).

The figure also shows that the time periods covered by the modified TPR-tree and the past R-tree index overlap, meaning that both have to be queried to answer past queries. Distant-past queries do not incur any I/O operations on the modified TPR-tree as the search in the modified TPR-tree stops after examining the insertion times of the entries in the memory-resident root of the TPR-tree. Nevertheless, in the worst case caused by a single object that has not been updated since the beginning of the recorded history, all past queries will incur I/O operations in both indices. If the index has just started recording trajectories, both indices will be of similar size. The heights of both index trees will most likely be the same as the height of a single index tree storing all objects because disk-based index trees have high fanouts. Thus queries will, in most cases, be twice as expensive in the double-tree index as in a single index storing all objects.

A more significant disadvantage of this approach than that of querying two indices is that the timeslice query performance of the R-tree degrades as the amount of data in it increases. No performance guarantees similar to the ones given by partial persistence can be provided for past timeslice queries. Tao and Papadias [2001] clearly demonstrate this through their performance experiments. In Section 7.5, we report on empirical studies of this double-tree index approach that uses an R*-tree and a modified TPR-tree.

5. STRUCTURE AND ALGORITHMS

This section presents the data structures and algorithms associated with the R$^{\mathrm{PPF}}$-tree. As pointed out earlier, we aim to capture and index the actual, real-world positions of the data objects across all of time. In temporal database terms, we consider the valid time of the objects' positions (although we do not allow general updates).

We assume position samples for an object arrive in time order, and we predict the future movement of the object by means of a linear function. When a new position sample arrives, we thus need to correct the prediction that covers the period since the last sample was received, and we need to repredict the future position. This limited need for corrections enables us to support valid time by an extended notion of partial persistence.

Partial persistence was previously proposed by Driscoll et al. [1989] and was also used by Becker et al. [1996] for creating a multiversion B-tree. Partial persistence transforms a linked data structure, termed an *ephemeral* structure, into a corresponding data structure that retains and enables the querying of all past states (in the transaction time sense) of the data being indexed. While the application of partial persistence guarantees that the resulting indexing technique is efficient in a specific sense, it should be noted that the application of partial persistence to an existing indexing technique is not a mechanical process but rather one that involves nontrivial and very significant design decisions.

5.1 Partial Persistence Framework

In explaining partial persistence, we will use terminology that applies to a generalization of R-trees known as grow-post trees [Lomet 1991]. These are balanced trees that store all data entries in their leaf nodes and store bounding predicates and pointers in non-leaf nodes. For example, the TPR-tree uses time-parameterized rectangles as bounding predicates. Non-leaf nodes thus serve to direct search. The algorithms associated with grow-post trees use three fundamental building blocks. Algorithm ChooseSubtree takes an entry that can be inserted into the tree and a tree node as arguments, and it determines the subtree rooted at that node in which the entry should be placed. Algorithm Split handles over-full nodes, thus enabling the tree to grow. Algorithm ComputeBP computes a bounding predicate for a node.

When applied to a grow-post tree, partial persistence guarantees that the current and any previous state of the data set can be queried as if it was stored in a separate, ephemeral tree with a fanout of at least $d \cdot b$—a substantial fraction of b which is the maximum number of entries in a data page in the ephemeral structure. This means that by design, no matter how many past states are accumulated in a partially persistent data structure, past timeslice queries will have a performance similar to the performance of timeslice queries on the ephemeral data structure.

When applying partial persistence, the data and index entries in the ephemeral structure are extended to include two additional fields: *insertion time* and *deletion time*. Thus, an entry contains a data item or a bounding predicate which is managed by the ephemeral algorithms, and an interval timestamp which is managed specially. The timestamps record the times when the corresponding ephemeral entry was inserted and logically deleted.

An entry is *alive* from its insertion time to its deletion time, upon which it becomes *dead*. Nodes are also categorized as being either alive or dead. In an entry of an alive node, the special deletion time ∞ denotes that the entry is alive. In a dead node, which can be produced by a so-called *time split*, the deletion time ∞ indicates that the entry was alive when the node was time split.

The key property of a partially persistent index is that, for each moment in time and for each non-root node, the number of alive entries is either zero or at least $d \cdot b$, where b is the node capacity and $d < 1$ is a predefined constant. This property is called the *weak version condition*, and breaking it causes a *weak version underflow*. This property guarantees that the alive objects at a given time are clustered into a small number of nodes which makes querying efficient.

The weak version condition is one of two invariants maintained by the update algorithms. The other, termed the strong version condition, will be covered shortly. When an insertion or a deletion is performed at time t, the target leaf node is located first. The algorithms of the ephemeral structure are applied while considering only the entries alive at time t. Having located the target leaf node, an insertion operation adds a data entry with timestamp $[t, \infty)$ to that node. The deletion operation just sets the endtime of an alive data entry to t. Before writing the node to disk, algorithm *AssureInvariants*, sketched in Figure 5, ensures that the invariants of partial persistence are maintained.

If the node contains no less than $d \cdot b$ alive entries and no more than b entries, no invariants are broken, and the node can be written to disk. However, if the bounding predicate of the node is changed, this predicate must be updated in the appropriate entry in the parent node. The timestamp of the parent entry is not changed. Note that the node may have a mix of dead and alive entries, denoted by the letters "AD" in Figure 5, but the ephemeral ComputeBP considers only the spatial coordinates of the entries; the timestamps function solely to disregard those entries that are dead inside the time interval of the live parent entry. Such entries may occur if the parent node was time split during the lifetime of the child node, thus producing two parent entries in two parent nodes, pointing to the same child but having disjoint time intervals.

If the node already contains b entries, a *node overflow* occurs. Node overflow as well as weak version underflow are handled by a time split (also called a *version split* [Becker et al. 1996; Kumar et al. 1998]). When a node x is time split at time t, all entries from x alive at t are copied to a new node y, and their timestamps are set to $[t, \infty)$. Node x is considered dead after time t. This is recorded by setting the deletion time of x's parent entry to t. This logical deletion of the parent entry is denoted by the letter "D" in the figure. Note that node x is not written to disk and that its bounding rectangle in the parent entry is not changed.

The new node y produced by the time split may be almost full or almost empty. If so, a few subsequent inserts or deletes would trigger a new time split

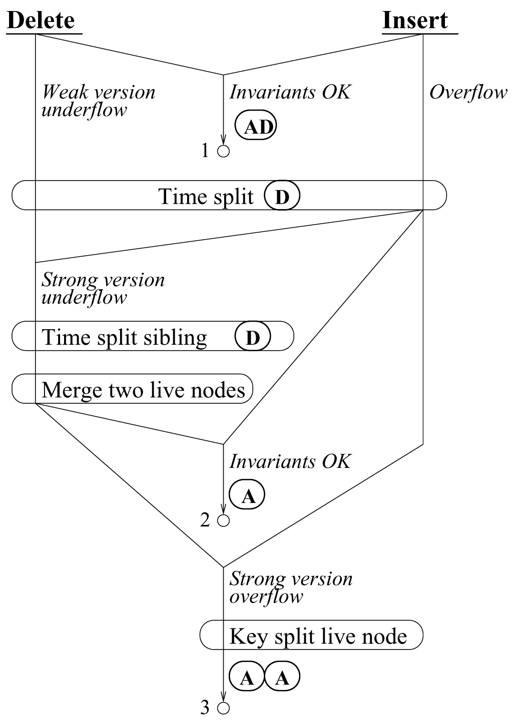


Fig. 5. Algorithm *AssureInvariants*.

which, in the worst case, will result in a space cost of $\Theta(1)$ nodes per operation. To avoid this phenomenon, the number of alive entries in a newly created node must be between $(d + \epsilon)b$ and $(1 - \epsilon)b$, where ϵ is a predetermined constant. This is the *strong version condition*. Note that this constrains ϵ to be less than $(1 - d)/2$. If a time split leads to less than $(d + \epsilon)b$ entries in a node, then a *strong version underflow* occurs, and a newly created node has to be merged with another node of alive entries which is produced by applying a time split to a live sibling. To choose this sibling, the bounding predicate of the entries in y is computed, and ChooseSubtree is used to identify a live sibling of y that is best for the insertion of this bounding predicate.

If, after merging or the initial time split, the node y satisfies all invariants, its bounding predicate is computed, and a new entry with this predicate and timestamp $[t, \infty)$ is inserted in a parent (letter "A" in the figure symbolizes that all bounded entries are alive).

If, after merging or the initial time split, there is more than $(1-\epsilon)b$ entries in a node, a *strong version overflow* occurs, and a *key split* of node y is performed. In a key split, the entries are split according to the ephemeral node splitting algorithm. Two new live entries containing the bounding predicates of the two new nodes are inserted into the parent node, symbolized by the letters "AA" in Figure 5.

As Figure 5 shows, running *AssureInvariants* on a leaf node may result in changed or additional entries in the parent node. More specifically, a live parent entry of a leaf node may get modified (exit 1, Figure 5) or up to two live entries in a parent node may be logically deleted and up to two new live entries may be added (exits 2 and 3). If the parent node is changed, *AssureInvariants* is called on this node with weak version underflow, satisfied invariants, or overflow as a possible outcome.

If needed, the described process is repeated level by level until the root node is reached. If the root node is time-split at time t, a pointer to the new live node together with timestamp $[t, \infty)$ is added to a special root array [Becker et al. 1996] that is stored in main memory.

While partial persistence as described here can be applied fairly easily to the R-tree, its application to the TPR-tree is challenging.

5.2 Computing and Maintaining Time-Parameterized Bounding Rectangles

We first provide the data structure for node entries in the R$^{\text{PPF}}$-tree. We then explain why the TPR-tree procedure for computing time-parameterized bounding rectangles (TPBRs) cannot be used in the R$^{\text{PPF}}$-tree, and we present three alternatives for computing TPBRs.

5.2.1 *Node Entries of the R$^{\text{PPF}}$-Tree.* Leaf entries for m-dimensional point objects consist of an object identifier, a time-parameterized point, and a time interval of validity. Similarly, the non-leaf entries consist of a pointer to a child node, a TPBR, and a time interval of validity. Node entries thus have the following structure:

$$(oid/ptr,\ tpp/tpbr,\ t^{\vdash}, t^{\dashv}).$$

Here $tpp = (\bar{x}; \bar{v}) = (x_1, \ldots, x_m; v_1, \ldots, v_m)$, with the x_i and v_i being the position and velocity coordinates, respectively, of the object at time $t^{\vdash}$ (also denoted $\bar{x}(t^{\vdash})$ and $\bar{v}(t^{\vdash})$). To ensure this, when a node is time split, $\bar{x}$ is recomputed for the entries of the new live node—simple copying is not enough. The structure of *tpbr* and how it is computed is the topic of the rest of Section 5.2. In the following, we consider TPBRs that bound time-parameterized points. The generalization to TPBRs that bound other TPBRs is straightforward.

5.2.2 *TPBRs of the TPR-Tree and Partial Persistence.* When constructing a partially persistent R-tree, the ephemeral algorithms of the R-tree are reused as black boxes in the framework of partial persistence: once the alive entries during a specific time interval are found, their minimum bounding rectangle is computed while ignoring the timestamps.

Partial persistence effectively adds an orthogonal time dimension to an R-tree representing data in m spatial dimensions, rendering it $m + 1$-dimensional. In contrast, the TPR-tree already represents data in $m + 1$ dimensions, and partial persistence should just extend the existing temporal dimension into the past.

The TPBRs of the TPR-tree bound objects that are all alive at the current time by computing the minimum bounding rectangle according to the positions of objects at the current time and by extending it with minimum and maximum speeds in each dimension. The resulting TPBR has $4m$ coordinates:

$$([x_1^{\vdash}, x_1^{\dashv}], \ldots, [x_m^{\vdash}, x_m^{\dashv}]; [v_1^{\vdash}, v_1^{\dashv}], \ldots, [v_m^{\vdash}, v_m^{\dashv}]).$$

Here,

$$\begin{aligned} x_i^{\vdash} &= \min_{o \in Node} \{o.x_i(\mathrm{CT})\}; & v_i^{\vdash} &= \min_{o \in Node} \{o.v_i\}; \\ x_i^{\dashv} &= \max_{o \in Node} \{o.x_i(\mathrm{CT})\}; & v_i^{\dashv} &= \max_{o \in Node} \{o.v_i\}, \end{aligned}$$

where $Node$ is the node, the entries of which are bounded by the computed TPBR.

In the TPR-tree, $x_i^{\vdash}$ and $x_i^{\dashv}$ are additionally recomputed to represent the coordinates of the bounding rectangle at the common reference time (e.g., zero): $x_i^{\dashv} = x_i^{\dashv} - v_i^{\dashv} \cdot \mathrm{CT}$ and $x_i^{\vdash} = x_i^{\vdash} - v_i^{\vdash} \cdot \mathrm{CT}$.

This procedure is perfectly suitable in cases when the node has just been time split, that is, when $t^{\vdash}$ of the TPBR is CT (denoted by "A" in Figure 5). But as Figure 3 illustrates, TPBRs of the TPR-tree are not necessarily valid for time points prior to the current time. Also, they do not take into account different insertion times or finite deletion times of objects. Thus, as they are, they cannot be used after operations that do not lead to time splits ("AD" in Figure 5). It is to address this problem that we investigate different modifications of TPBRs.

Figure 6 shows the evolution of four different types of one-dimensional TPBRs that initially bound three objects. The figure shows how the TPBRs change when two insertions are performed, the second of which causes a time split. The top row of the figure demonstrates how the TPBRs of the TPR-tree can be used in the $\mathrm{R}^{\mathrm{PPF}}$-tree, by changing how $x_i^{\vdash}$ and $x_i^{\dashv}$ of a TPBR are computed:

$$x_i^{\vdash} = \min_{o \in Node} \{o.x_i(t^{\vdash})\}; \qquad x_i^{\dashv} = \max_{o \in Node} \{o.x_i(t^{\vdash})\}.$$

Effectively, the TPBR is computed to be minimum at its insertion-time ($t^{\vdash}$), and the trajectories of the entries are extended to span the time period from $t^{\vdash}$ to infinity.

The figure demonstrates that such a straightforward use of the TPR-tree's TPBRs may result in bounding rectangles that grow fast and are not minimum at any point in time. The bad quality of such bounding rectangles is even more evident when compared with the TPBRs that would result from storing the same data in the TPR-tree. The main reason for this is the inability to do tightening, the process of making the TPBR minimal at the current time. Additionally, bad alive bounding rectangles result in bad dead bounding rectangles being produced in time splits, as exemplified in the top-right picture in the figure.

5.2.3 *Optimized TPBRs.* One may formulate the computation of bounding rectangles as an optimization problem. As described in Section 3.3, one of the heuristics used by the TPR-tree insertion algorithms to group points is the integral of area (or m-dimensional volume in the general case). Assuming that

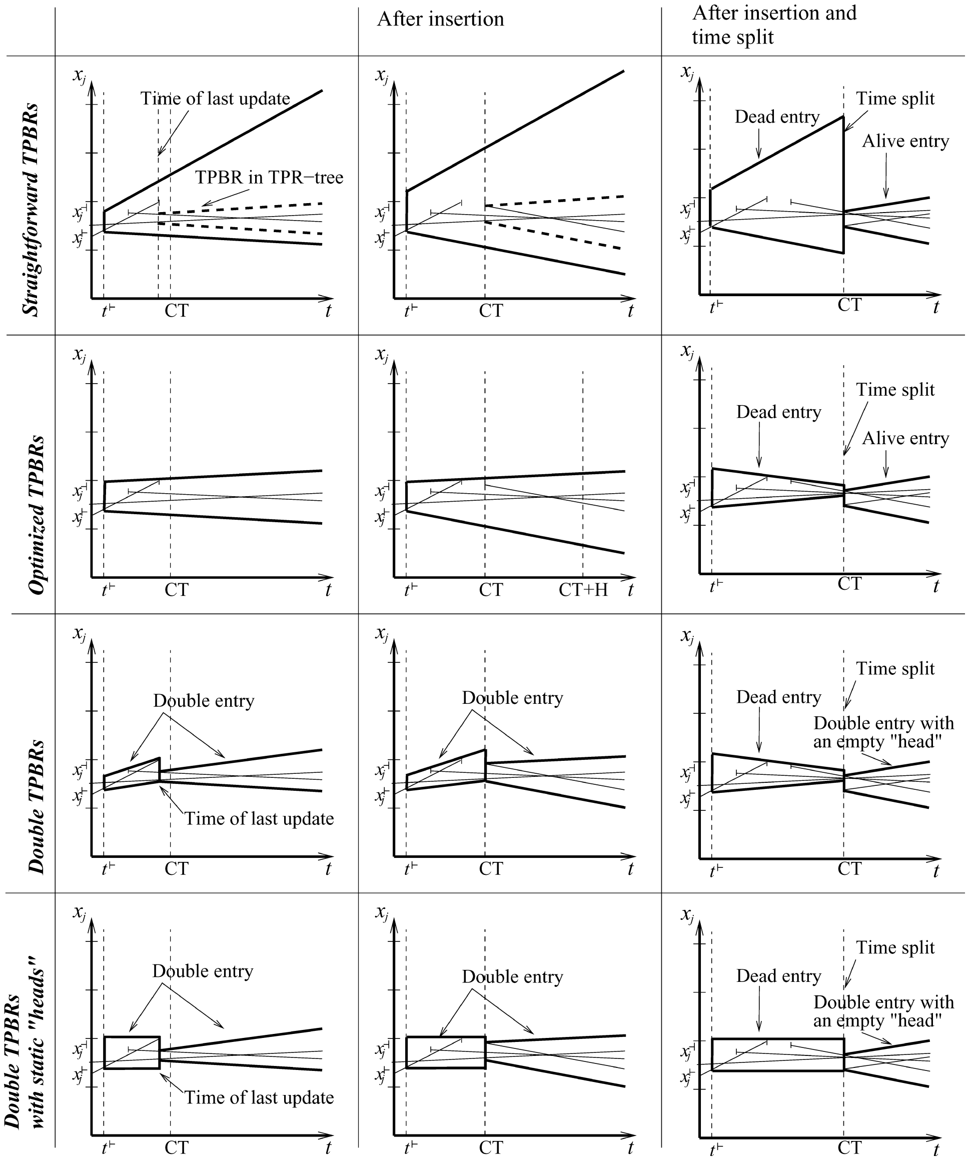


Fig. 6. Evolution of different types of time-parameterized bounding rectangles.

queries are uniformly distributed in time from the current time (CT) and H time units into the future, the integration is done from CT to CT + H, where H is a workload-specific parameter that can be automatically adjusted by observing the index workload [Šaltenis and Jensen 2002].

Similarly, when computing what we term an *optimized* TPBR, its spatial and velocity coordinates should be chosen so as to minimize the integral of the area of the bounding rectangle from $t^{\vdash}$ to CT + H. Figure 7 gives a sketch of an algorithm to compute such an optimal one-dimensional time-parameterized

```
OptimalTPBI(Node)
1  Let $S$ be the set of delimiting points in the $(x, t)$-plane that specifies a set of line segments
   denoting the (finite) trajectories of the one-dimensional moving points in node $Node$.
2  Let $C$ be the convex hull of the points in set $S$.
3  $med \leftarrow (Node.t^{\vdash} + \mathrm{CT} + H)/2$
4  $v = \max_{o \in Node \wedge o.t^{\dashv} = \infty}\{o.v\}$
5  If $C$ is not to the left of the line $t = med$, let $l$ be the line that goes through the upper
   of the two edges of $C$ that cross this line.
6  If no $l$ was found or if the slope of $l$ is smaller than $v$, let $l$ be the line with slope $v$ that
   passes through one of the vertices of $C$, but does not cross its interior.
7  Use $l$ as the trajectory of the upper bound of the bounding interval.
8  Repeat steps 4–7, but with "upper" exchanged with "lower" and "smaller" exchanged
   with "greater."
```

Fig. 7. Algorithm *OptimalTPBI*.

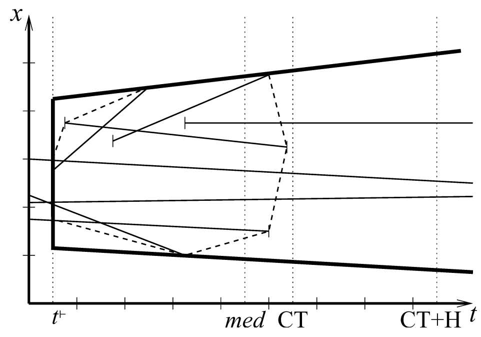


Fig. 8. Optimal TPBR for a set of one-dimensional moving objects.

bounding interval. Figure 8 illustrates how a convex hull is used in the computation. In this figure, the upper bound of the interval has the trajectory corresponding to the line l computed in step 5 of the algorithm and the lower bound has the trajectory corresponding to the line l computed in step 6 (during the repeat caused by step 8). It can be proven that this algorithm computes an optimal—in terms of the integral of the length—bounding interval [Šaltenis and Jensen 2002].

In more dimensions, near-optimal TPBRs can be computed by combining solutions for separate dimensions. This is done calling *OptimalTPBI* for each dimension, but changing how *med* is computed in step 3 of the algorithm. The details can be found elsewhere [Šaltenis and Jensen 2002].

Observe also that, when a node is time split, the TPBR of the old node can be updated to become optimal for a collection of finite trajectories (see "D" in Figure 5 and the rightmost picture of the second row in Figure 6). This represents a refinement of the traditional partial persistence approach where a finite $t^{\dashv}$ is simply set in the bounding predicate of the old node.

Optimized TPBRs are described by the same number of parameters as TPBRs of the TPR-tree. If we assume that a timestamp, a coordinate, or a page id takes up one word, then the size of a non-leaf entry with an optimized TPBR is $4m + 2 + 1$ words: $2m$ spatial coordinates, $2m$ velocity coordinates, two timestamps, and one page id. When $m = 2$, the size is 11 words. Note that live entries can be further compressed by not storing the deletion timestamp, increasing the fanout of nodes with live entries slightly.

The disadvantage of optimized TPBRs is that their computation is complex and takes $O(mn \log n)$ time where n is the number of objects bounded [Šaltenis and Jensen 2002]. In addition, these TPBRS cannot be tightened.

5.2.4 *Double TPBRs.* To allow tightening, we introduce *double* TPBRs. The idea is to divide a TPBR into two parts: a head and a tail. The tail starts at the time of the last update (insertion or deletion), t_{lu}, and extends to infinity. The tail is the regular TPBR of the TPR-tree.

The head bounds the finite segments of trajectories of objects from $t^{\vdash}$ to t_{lu}. This can be done either using an optimized TPBR or TPBRs with static bounds, that is, bounds with zero speeds. Both cases are illustrated in the last two rows of Figure 6. In the first case, the double TPBR is represented as follows: $(tpbr_h, t_{lu}, tpbr_t)$. In the second case, the zero speeds of $tpbr_h$ can be omitted, thus reducing the size of an index entry.

The entry size is further reduced when the bounded node is time split, and the TPBR for the resulting dead node is recomputed (see the rightmost column in Figure 6). In such cases, a double TPBR with a static head becomes a simple static bounding rectangle, and an optimized double TPBR becomes a regular optimized TPBR.

More specifically, using the same assumptions as in the previous section, the size of a live non-leaf entry with a double TPBR is $2 \cdot 4m + 2 + 1$ words. For dead entries, the size can be reduced to $4m + 2 + 1$. For two-dimensional data, the corresponding sizes are 19 and 11 words. A live non-leaf entry with a double TPBR with a static head takes up $2m + 4m + 2 + 1$ words, and its dead counterpart is $2m + 2 + 1$ words long. The corresponding sizes for two-dimensional data are 15 and 7 words.

If the nodes are considered to contain an even mix of live and dead entries, the presented calculations of entry sizes lead to very similar average fanouts of nodes with optimized TPBRs and nodes with double TPBRs with static heads. The average fanout of nodes with optimized double TPBRs is about 33% lower than for the other two types of TPBRs.

Note also that the computation of double TPBRs with static heads, in contrast to the computation of double and single optimized TPBRs, is as simple as the computation of MBRs in the R-tree or TPBRs in the TPR-tree. It involves only computing minimums and maximums of n numbers. For the upper bound:

$$\begin{aligned} tpbr_h.x_i^{\dashv} &= \max_{o \in Node} \{\max(o.x_i(t^{\vdash}), o.x_i(t_{lu}))\}; \\ tpbr_t.x_i^{\dashv} &= \max_{o \in Node \wedge o.t^{\dashv} = \infty} \{o.x_i(t_{lu})\}; \\ tpbr_t.v_i^{\dashv} &= \max_{o \in Node \wedge o.t^{\dashv} = \infty} \{o.v_i\}. \end{aligned}$$

For the lower bound, minimums are used in place of the maximums.

Double TPBRs support tightening, but they also have extra associated costs when compared with the two other kinds of TBPRs. Specifically, their update costs may be higher. An optimized TPBR of a node may not need to be

```
$\textsc{Search}(R, [t^{\vdash}, t^{\dashv}])$
for each root $r$ from the array of roots
  if $[r.t^{\vdash}, r.t^{\dashv}) \cap [t^{\vdash}, t^{\dashv}] \neq \emptyset$ then $\textsc{SearchSubtree}(r, R, [r.t^{\vdash}, r.t^{\dashv}) \cap [t^{\vdash}, t^{\dashv}])$

$\textsc{SearchSubtree}(Node, R, [t^{\vdash}, t^{\dashv}])$
if $Node$ is a leaf node then
  for each $e \in Node$
    if $Intersect(e.tpp, R, [e.t^{\vdash}, e.t^{\dashv}) \cap [t^{\vdash}, t^{\dashv}])$ then output $e$
else
  for each $e \in Node$
    if $Intersect(e.tpbr, R, [e.t^{\vdash}, e.t^{\dashv}) \cap [t^{\vdash}, t^{\dashv}])$ then
      $\textsc{SearchSubtree}(e.ptr, R, [e.t^{\vdash}, e.t^{\dashv}) \cap [t^{\vdash}, t^{\dashv}])$
```

Fig. 9. Algorithms for processing a window query.

updated after the insertion or deletion of an entry from the node (stopping the ascent of procedure *AssureInvariants*). In contrast, a double TPBR, by a virtue of recording the last update time, will always have to be updated unless several updates occur at the same time instant. This updating of t_{lu} ascends to the root because, when double TPBRs are bounded by a parent double TPBR, the t_{lu} of the bounding TPBR is set to the maximum of the t_{lu}'s of the bounded TPBRs.

Section 7 experimentally compares the three types of TPBRs.

5.3 Querying the R$^{\text{PPF}}$-Tree

Querying the R$^{\text{PPF}}$-tree is analogous to querying any other partially persistent data structure. We proceed to detail how both timeslice and window queries, as defined in Section 2.2, are supported. Noting that a timeslice query is a special case of a window query, we describe briefly an algorithm implementing a window query $Q = (R, t^{\vdash}, t^{\dashv})$. In the pseudocode shown in Figure 9 and explained in the following, the closed-open validity interval $[r.t^{\vdash}, r.t^{\dashv})$ of a root r is obtained from the array of roots described in Section 5.1.

Function *Intersect*(*tpbr*, R, *ti*) takes three arguments, a time-parameterized bounding rectangle *tpbr*, a rectangle R, and a time interval *ti*. It returns true if *tpbr* intersects with R at some time during *ti*. As shown in Figure 3 for the case where $m = 1$, computing this function corresponds to checking whether an $(m + 1)$-dimensional rectangle (the spatial query rectangle R extended by the query time interval *ti*) intersects with an $(m + 1)$-dimensional trapezoid (the time-parameterized bounding rectangle *tpbr* extended by the query time interval *ti*). This check is done the same way as in the TPR-tree [Šaltenis et al. 2000]. Note that time-parameterized points in leaf entries can be considered degenerate TPBRs, enabling the same procedure to be used both for leaf and non-leaf entries. When the double TPBRs described in the previous section are used, *Intersect* simply examines the head and the tail of a TPBR separately. The described algorithm works for past, current, and future queries.

Observe that, due to entry copying by time splits, a nondegenerate window query (i.e., $t^{\dashv} > t^{\vdash}$) may return duplicates which then have to be filtered. In addition, due to the copying of non-leaf entries, some nodes may be accessed more than once during the search process. To avoid this, Van den Bercken and Seeger [1996] proposed the reference-point method for multiversion B-trees. As noted by the authors, this method can also be applied to multiversion R-trees, meaning that it is also applicable to the R^{PPF}-tree. Alternatively, the resulting extra I/O operations can be avoided by either employing a main memory buffer large enough to store all of the tree nodes accessed by a query or a hash table of the ids of the pages accessed. In our experiments, we use the hash-table approach. Note that the link algorithm for duplicate avoidance [Van den Bercken and Seeger 1996] cannot be applied to the R^{PPF}-tree although, as we will describe in the next section, the R^{PPF}-tree has links between time-split nodes. The link algorithm requires that bounding regions of nodes do not intersect which is not the case in the R^{PPF}-tree.

5.4 Correction of Last-Recorded Predictions

With the modified procedures for computing TPBRs, the TPR-tree can be made partially persistent, but the resulting trajectory of a moving object (see Figure 2) consists of disconnected and slightly incorrect segments. To obtain accurate, connected trajectories, the velocity of the trajectory segment recorded at the last update has to be modified when a new update is performed. As a result of satisfying the partial persistence invariants, which yields performance guarantees, such a correction is not always trivial. This section describes in detail how such corrections are performed.

5.4.1 *Modifications to the Index Structure.* The last-recorded trajectory segment of an object may be stored in more than one leaf node because the leaf node in question may have been time split a number of times since the previous update. Time splits and copying of information must be tolerated because they are essential in order for multiversion indexes to offer timeslice query performance that is independent of the amounts of updates. The consequence is that, while insertions that start new trajectories and deletions that end trajectories can be performed as described in Section 5.1, updates are much more complex.

To properly correct the last-recorded trajectory segment, all leaf nodes that contain copies of this segment have to be visited. To facilitate this, we maintain two predecessor pointers *pred1* and *pred2* for each node. These record which nodes were split off of which other nodes during time splits. Usually *pred2* is NIL. Whenever a node is time split, *pred1* of the newly created node is set to point back to the original node. If the time split is followed by a merge, *pred1* and *pred2* of the resulting node are set to point to the two nodes that were time split prior to the merge. A key split simply copies the *pred1* and *pred2* of the original node into the split-off node.

With these modifications to the index structure in place, we proceed to describe the update algorithm.

5.4.2 *The* Update *Algorithm.* As input, the update algorithm takes the old entry e_o, to be logically deleted (and corrected), and the new entry e_n, to be inserted. Both $e_o.t^{\dashv}$ and $e_n.t^{\dashv}$ are infinite, $e_n.t^{\vdash}$ is equal to the current time (CT), and $e_o.t^{\vdash}$ is equal to the time of the last update of this object.

```
UPDATE($e_o, e_n$)
1   $Leaf \leftarrow TPRFindLeafForDeletion(Root(\mathrm{CT}), e_o)$
2   $(e_c, PageIds) \leftarrow$ CORRECTLEAVES($Leaf, e_o, e_n$)
3   $e.t^{\dashv} \leftarrow \mathrm{CT}$, where $e \in Leaf$, such that $e.id = e_o.id$        // Logically delete $e_o$
4   $RightNode \leftarrow Leaf$
5   $Assuring \leftarrow true$
6   while $RightNode \neq$ NIL        // Do left-up phase
7     ($NewRightNode$, $PageIds$) $\leftarrow$CORRECTPARENTS($RightNode$, $e_c$, $PageIds$)
8     if $Assuring$ then
        // Here $RightNode$ is a $Leaf$ or its live ancestor
9       $Assuring \leftarrow AssureInvariants(RightNode)$
10    $RightNode \leftarrow NewRightNode$
11  $Node \leftarrow TPRFindLeafForInsertion(Root(\mathrm{CT}), e_n)$        // Insert $e_n$
12  Add $e_n$ to $Node$ with timestamp $[\mathrm{CT}, \infty)$
13  while $AssureInvariants(Node)$ do $Node \leftarrow Parent(Node)$
```

Fig. 10. Algorithm *Update*.

The update algorithm proceeds in the following five phases.

(1) In the *deletion down* phase, the live leaf node that contains e_o is found using the ephemeral TPR-tree deletion algorithm.
(2) In the *left* phase, e_o is logically deleted, after correcting its velocity vector and the velocity vectors of all e_o copies in dead leaf nodes. The velocity correction is done to produce a trajectory segment with an end that connects to the beginning of the segment represented by e_n. The necessary dead leaf nodes are found by following the predecessor pointers.
(3) In the *left-up* phase, the tree is traversed upwards performing *AssureInvariants* as described in Section 5.1. In addition, at each level of the tree, predecessor pointers are used to traverse left (back in history) and to correct bounding rectangles that may have been invalidated by the corrections of the trajectory of e_o at the leaf level.
(4) In the *insertion down* phase, the ephemeral TPR-tree insertion algorithm is used to find the node in which to insert e_n.
(5) In the *insertion up* phase, after inserting e_n, the tree is traversed upwards, again performing *AssureInvariants* as described in Section 5.1.

Figure 10 gives pseudocode for the *Update* algorithm. Before considering the body of this algorithm in more detail, we list the key assumptions used in the pseudocode and in Figures 11 and 14. Specifically, we assume the array of roots mentioned in Section 5.1 has the following structure:

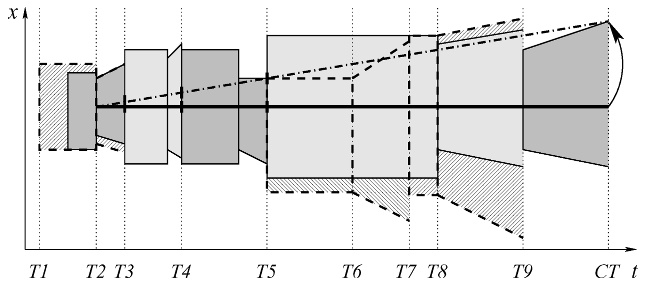
$$(t_1, rp_1, l_1), (t_2, rp_2, l_2), \ldots, (t_r, rp_r, l_r).$$

The i-th element of a root array of r elements contains a root pointer rp_i, the start time of the half-open time interval $[t^{\vdash} = t_i, t^{\dashv} = t_{i+1})$ that specifies the validity of this root (here t_{r+1} is defined as ∞), and the tree level l_i of the root (leaves are at level 0). We assume that the root array can be accessed using the function $Root(t)$, which returns a pointer to the root alive at time t. A root array is shown as part of Figure 12.

For a node Nd, we define $Nd.t^{\vdash}$ to be the minimum of the insertion times of all entries in Nd. If Nd is a root, $Nd.t^{\dashv}$ is also defined and can be retrieved from

```
CorrectLeaves($Leaf, e_o, e_n$)
1   $e_c \leftarrow e_o$
2   $e_c.tpp.\bar{v} \leftarrow (e_n.tpp.\bar{x} - e_o.tpp.\bar{x})/(\mathrm{CT} - e_o.t^{\vdash})$
3   $PageIds \leftarrow \varnothing$
4   while $Leaf \neq$ NIL
5     $PageIds \leftarrow PageIds \cup \{PageId(Leaf)\}$
6     Let $e \in Leaf$ such that
      $e.t^{\dashv} = \infty \wedge e.oid = e_o.oid \wedge e.tpp.\bar{v} = e_o.tpp.\bar{v} \wedge$
      $e.tpp.\bar{x} = e_o.tpp.\bar{x} + e_o.tpp.\bar{v}(e.t^{\vdash} - e_o.t^{\vdash})$
7     $e.tpp.\bar{v} \leftarrow e_c.tpp.\bar{v}$
8     $e.tpp.\bar{x} \leftarrow e_c.tpp.\bar{x} + e_c.tpp.\bar{v}(e.t^{\vdash} - e_c.t^{\vdash})$
9     if $Leaf.t^{\vdash} > e_o.t^{\vdash}$ then
10      Follow a non-NIL predecessor pointer, $Leaf.pred1$ or $Leaf.pred2$, to a predecessor
        node. If both predecessor pointers are non-NIL, only one of the predecessor nodes
        has an entry $e$ that satisfies the conditions in line 6. Make $Leaf$ point to that node.
11    else   // $e$ was inserted during the lifetime of $Leaf$
12      $Leaf \leftarrow$ NIL
13  return $(e_c, PageIds)$
```

Fig. 11. Algorithm *CorrectLeaves*.



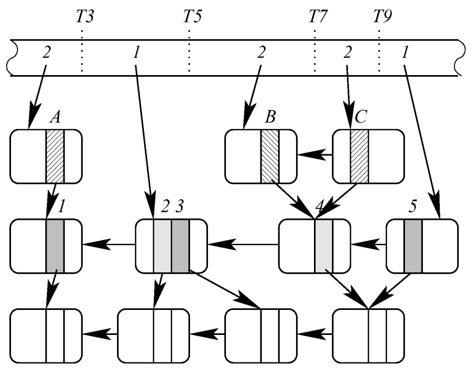


Fig. 12. Correction of a trajectory's last-recorded segment.

the corresponding element of the root array. The function *PageId*(*Nd*) returns a disk page identifier that serves as a pointer to *Nd* in the index structure.

We also assume that algorithm *AssureInvariants*(*Nd*) (Figure 5) returns false when *Nd* is either a root or was not changed by a previous invocation of *AssureInvariants* on *Nd*'s child. Finally, we assume that *TPRFindLeafForDeletion* and *TPRFindLeafForInsertion* are the algorithms of the TPR-tree that find an alive leaf for the insertion or deletion of an entry. Each of these algorithms records the path of alive nodes from the root to the leaf it finds. These paths are later used by the function *Parent*(*Nd*) to return an alive parent of the alive argument node *Nd*.

Having found the alive leaf node where e_o is stored (the *deletion down* phase), the *Update* algorithm proceeds by calling *CorrectLeaves*, which implements the *left* phase of the algorithm. The algorithm *CorrectLeaves* corrects all copies of e_o found at leaf level and collects a set of pointers (*PageIds*) that point to the nodes containing the corrected fragments of the trajectory of e_o. Algorithm *CorrectLeaves* also returns e_c, the corrected variant of e_o. Using *PageIds* and e_c, the algorithm *CorrectParents* traverses the parent level and checks if all entries pointing to nodes in *PageIds* contain e_c during their validity intervals (the *left-up* phase). If needed, TPBRs in these entries are expanded. The procedure is repeated, correcting TPBRs level by level, up the tree (line 7 in Figure 10).

Note that the number of levels for which the algorithm calls *CorrectParents* is independent of how high in the tree *AssureInvariants* has to ascend. In some cases, *AssureInvariants* will stop at the parent of the modified leaf (setting *Assuring* to *false*), while *CorrectParents* has to be called on all levels of the tree as will be explained in Section 5.4.4. In the following, the algorithms *CorrectLeaves* and *CorrectParents* are covered in detail.

5.4.3 *The CorrectLeaves Algorithm.* Algorithm *CorrectLeaves*, presented in Figure 11, is fairly straightforward. It first computes the corrected variant of e_o which, at the outset, differs from e_c only in its velocity vector. Then it traverses the predecessor pointers back from the argument node *Leaf*, finding all copies e of e_o. Note that $e.tpp.\bar{x}$ was recomputed for the time point when the corresponding node was produced by a time split (see Section 5.2.1). This is taken into account in line 6 when finding e and in line 8 when recomputing $tpp.\bar{x}$ using the corrected velocity.

In line 10, where a predecessor pointer is followed, the case where both predecessor pointers of *Leaf* are non-NIL must be handled. In this case, *Leaf* was produced after merging two time-split nodes. Only one of these nodes has a copy of e_o because otherwise there would have existed two copies of e_o in the live TPR-tree at the time of the time split ($Leaf.t^{\vdash}$), and this cannot happen in the TPR-tree. The algorithm stops when a leaf node is found that has a lifetime starting before or at the time when e_o was inserted.

Figure 12 shows an example of the evolution of a part of an index storing one-dimensional data. In the (x, t) space (the top of the figure), the last segment of the trajectory of an object is shown in its old version (bold line) and in its corrected version (dashed line). Double TPBRs with static heads are also shown in non-leaf nodes of the tree. The shadings capture the correspondences between entries at the bottom of the picture and the TPBRs.

In this case, the *CorrectLeaves* algorithm corrects four copies of the trajectory segment produced. These copies are produced by three time-splits that happened at *T3*, *T4*, and *T5*. Four pointers to leaf nodes are returned by *CorrectLeaves* in this example.

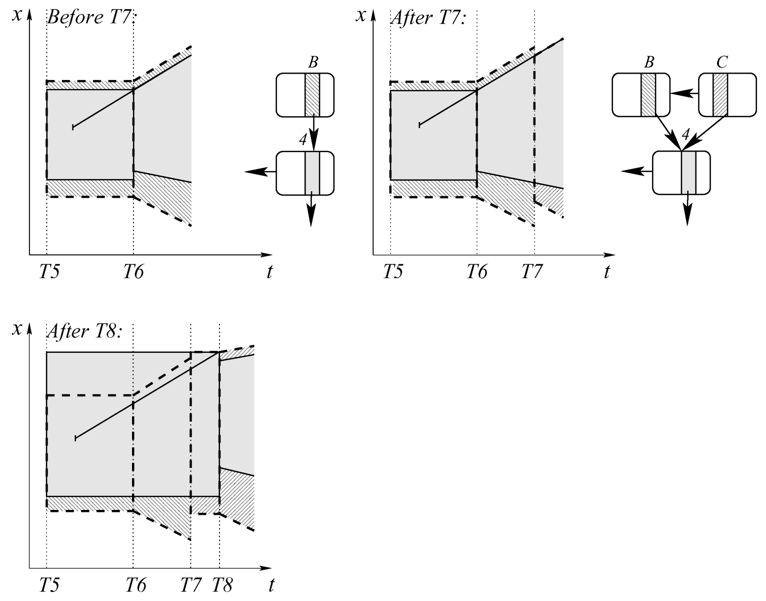


Fig. 13. The sequence of events leading to the anomalous grandparent B.

5.4.4 *Correcting the Non-Leaf Entries.* The pointers to leaf nodes returned by algorithm *CorrectLeaves*, *PageIds*, are passed to algorithm *CorrectParents*, whose task it is to check all parent entries of these nodes. In particular, all parent entries that do not bound the corrected trajectory e_c during their validity time intervals must be corrected. The output of the *CorrectParents* algorithm is another set of pointers to nodes, one level above the nodes identified by *PageIds*. This set will be passed to a subsequent call of *CorrectParents*. Intuitively, the returned set should contain pointers to the nodes that contain entries which were changed in the course of the algorithm. It would seem that there is no need to consider parents of nodes that were not changed. However, Figure 12 shows that this intuition is, in fact, misleading.

Specifically, Figure 12 demonstrates that an ancestor of an entry can be invalidated even if the immediate parent is not invalidated. Consider the entry numbered 4 on level 1 and its corresponding TPBR. It follows from the top of the figure that the TPBR in this entry is not invalidated by the correction of the trajectory segment. Nevertheless, the entry labeled B, the parent of entry 4 which is live from $T5$ to $T7$, is invalidated as its upper bound does not bound entry 4 throughout this time period.

Figure 13 shows the evolution of a small part of Figure 12. Entries 4 and B are shown together with an open-ended segment of a trajectory bounded by entry 4 (note that this segment is different from the one shown in Figure 12). The events of Figure 13 lead to the counterintuitive configuration of TPBRs mentioned previously.

Let us consider this situation in a bit more detail. The first part of the figure shows that, as required by the TPR-tree, entry B bounds entry 4 while they are both alive (from $T5$ to $T7$). From $T6$ to $T7$, nothing happened to either entries B and 4, or to any entries below them. The second part of the figure shows that, at time $T7$, due to activity in other parts of the tree, the node containing entry B was time split, and as a result, the shape of B was frozen. Its child, entry 4, continued to live. Then at time $T8$, the segment of the trajectory shown in the figure was logically deleted. As shown by the third part of the figure, this triggered a recomputation of the TPBR in entry 4. As a result, the TPBR in entry B no longer bounds the TPBR in entry 4. Note though, that all leaf-level entries that were bounded by B and its descendants during the time interval from $T5$ to $T7$ still remain inside B and all its descendants during this time interval.

Summarizing, in the $\mathrm{R}^{\mathrm{PPF}}$-tree, TPBRs in dead entries are only guaranteed to bound their leaf-level descendants (trajectories), not the TPBRs at higher levels. It is worth observing that this property also applies to the partially persistent R-tree.

Returning to the example of Figure 12, entry 5 is the only entry at level 1 that is invalidated by the correction of the trajectory segment at the leaf level. After correcting entry 5 the first call of *CorrectParents* algorithm has to return not only the pointer to the node containing entry 5 but also pointers to all the other nodes on level 1 shown in the figure, including the node with entry 4.

In general, if a node contains at least one entry with a pointer from the set *PageIds*, the pointer to this node is added to the new set of pointers that will be returned and subsequently passed to the next call of *CorrectParents*, on the grand-parent level (line 12 in Figure 14). This way, the algorithm checks the entries on all paths leading from the root(s) to all copies of the corrected trajectory.

5.4.5 *The CorrectParents Algorithm.* Figure 14 contains the pseudocode of algorithm *CorrectParents*. We proceed to explain the algorithm step by step.

As input, the algorithm takes an entry representing a corrected trajectory segment, e_c, a set of pointers to tree nodes, *PageIds*, and a pointer to the rightmost of these nodes, *RightNode*. Here, the rightmost node is the node with the largest $t^{\vdash}$. All nodes identified by *PageIds* are at the same level. As discussed in the previous section, the output of the *CorrectParents* algorithm is another set of pointers to nodes one level above the nodes identified by *PageIds*.

The structure of the *CorrectParents* algorithm is similar to the structure of the *CorrectLeaves* algorithm; it starts from the parent node of *RightNode* (line 5) and uses the predecessor pointers to traverse nodes backwards in time (line 15). If a node has two non-NIL predecessor pointers, at most one of these pointers points to a node that has an entry with a pointer from *PageIds*. If such an entry existed in both predecessor nodes of *Node*, two paths would have existed from the root to the corrected trajectory segment at time *Node*.$t^{\vdash}$. This is impossible, as the TPR-tree is a proper tree.

When checking whether e_c invalidates an entry e (line 11), the validity time of e must be known. If the node to which e belongs (i.e., *Node*) was time split when e was still alive, the deletion time of e was left at infinity. In this case, the true end time of the validity interval of e is equal to the deletion time of *Node*, which is the time of the time split that rendered *Node* dead. When a normal

```
CorrectParents($RightNode, e_c, PageIds$)
1  if $RightNode$ points to a root node then
2    Traverse the root array backwards in time: Start by letting $RightParent$ be a pointer
     to the root just before $RightNode$; then traverse the root array backwards until
     $Level(RightParent) = Level(RightNode) + 1$ or $RightParent.t^{\dashv} \leq e_c.t^{\vdash}$. In the latter
     case or if the beginning of the root array is reached, return (NIL, $\varnothing$).
3    $DelTime \leftarrow RightParent.t^{\dashv}$
4  else
5    $RightParent \leftarrow Parent(RightNode)$
6    $DelTime \leftarrow$ CT
7  $NewPageIds \leftarrow \varnothing$
8  $Node \leftarrow RightParent$
9  while $DelTime > e_c.t^{\vdash}$
10   for each $e \in Node$ such that $e.ptr \in PageIds$
11     If during $[e.t^{\vdash}, \min\{e.t^{\dashv}, DelTime\}]$, $e$ does not contain $e_c$, adjust the position and
       velocity coordinates of $e$ to achieve containment.
12   $NewPageIds \leftarrow NewPageIds \cup \{PageId(Node)\}$
13   if $Node.pred1 \neq$ NIL $\vee$ $Node.pred2 \neq$ NIL then
14     $DelTime \leftarrow Node.t^{\vdash}$
15     Follow a non-NIL predecessor pointer, $Node.pred1$ or $Node.pred2$, that leads to a
       node with an entry $e$ such that $e.ptr \in PageIds$. If both predecessor pointers are
       non-NIL, at most one of them has such an $e$. If there is no such predecessor, abort
       the loop; else make $Node$ point to that predecessor.
16   else      // $Node$ is a root without predecessors
17     Traverse the root array backwards in time: Start by letting $Root$ point to the root
       just before $Node$; continue until $Level(Root) = Level(Node)$ or $Root.t^{\dashv} \leq e_c.t^{\vdash}$. In
       the latter case or if the beginning of the root array is reached, abort the loop; else
       $Node \leftarrow Root$.
18     $DelTime \leftarrow Root.t^{\dashv}$
19 return ($RightParent$, $NewPageIds$)
```

Fig. 14. Algorithm *CorrectParents*.

top-down operation is performed in the R$^{\text{PPF}}$-tree, this true end time is brought down from the parent entry of *Node*. In contrast, in the right-to-left traversal of the *CorrectParents* algorithm, this end time (*DelTime*) is obtained from the node to the right of *Node* (line 14).

The traversal of non-leaf-level nodes using predecessor pointers is more complicated than in the *CorrectLeaves* algorithm because the *CorrectParents* algorithm has to take into account that the index can grow and shrink in height during its evolution. This causes two problems.

The first problem occurs when *RightNode* is already a root but, for the earlier time points, there are nodes at higher levels of the R$^{\text{PPF}}$-tree. This situation is illustrated in Figure 12. To correct entry *B* at level 2 after correcting the entries at level 1, algorithm *CorrectParents* has to start traversing from the rightmost node at level 2 (the node containing entry *C*). This node is not on the path of live nodes followed by the *TPRFindLeafForDeletion* algorithm (line 5 in Figure 14); instead, it is found as the first root to the left of *RightNode* that is one level higher than *RightNode*. This is done traversing the root array (line 2).

The second problem occurs when a right-to-left traversal using predecessor pointers is interrupted because both predecessor pointers are NIL. This can either mean that there are no more nodes at this level of the R$^{\text{PPF}}$-tree, in which case the traversal has to stop, or it can mean that the tree has shrunk

Table I. Symbols Used in the Cost Model

Symbol	Definition
b, b_l	non-leaf and leaf node sizes; maximum numbers of entries
f, f_l	average number of entries alive at a single timestamp in a non-leaf and a leaf node
$u = f/b, u_l = f_l/b_l$	average live utilization of a non-leaf and leaf node
$(1-\epsilon)b, (1-\epsilon)b_l$	number of entries causing a strong version overflow in a non-leaf and a leaf node
k	when key splitting b_s entries, the resulting nodes must have more than kb_s entries
UI	update interval: average time between two updates of an object
s_i	average time between two time splits of a node at level i
c_i	average number of nodes that have to be checked when correcting at level i

in height and then grown again, creating a gap in this level of the R$^{\text{PPF}}$-tree. Figure 12 illustrates the latter case. On level 2, both predecessor pointers of the node containing entry B are NIL. So to check whether entry A needs to be corrected, the algorithm has to use the root array to skip the lower-level root (line 17 in Figure 14).

Naturally, the described correction of the last-recorded fragments of a trajectory costs additional I/O when compared to a normal update operation. The size of this correction overhead will largely depend on the number of entries that store fragments of the trajectory to be corrected. This will in turn depend on the ratio between the average time period between two time splits of leaf nodes and the average time period between two updates of an object. To increase the average time period between time splits, parameter ϵ mentioned in Section 5.1 should be increased. In the next section, we present an analytical cost model that offers theoretical insight into the correction overhead. The performance experiments in Section 7 empirically investigate the effect of ϵ on the update performance.

6. ANALYTICAL EVALUATION OF THE CORRECTION OVERHEAD

Recall that an update of an object involves the logical deletion of the old entry representing the object and the insertion of new entry. As part of the deletion, corrections to split-off copies of the old entry caused by time splits must be performed. We proceed to consider the analytical modeling of the cost of having to perform these corrections. In the worst case, corrections will have to traverse predecessor pointers all the way back to the beginning of the recorded history. To obtain the average cost of a correction, we estimate how many nodes on average have to be accessed at each level of the tree during the correction. This depends on the average interval between time splits at each level. Similar cost modeling of the frequency of time splits was performed by Tao et al. [2002].

Table I summarizes the parameters used in our cost model. For the cost model, we assume that the evolution of an index encompasses two phases. In the *build-up phase*, new objects are more or less gradually inserted into the index. Once all the tracked objects are inserted into the index, the index enters the *stable phase*, where only updates of the existing objects are performed. Each update is a deletion followed by an insertion. We assume that these operations are uniformly distributed among the live leaf nodes and that each live leaf node receives an equal amount of deletions and insertions. In the stable phase, the live fanout of a node, that is, the number of entries alive at the current time, remains constant. We denote this quantity f_l for the leaf nodes and f for the non-leaf nodes. Note that $f < f_l$ because non-leaf entries are larger than leaf entries.

In the stable phase, almost all of structural changes are caused by time splits. After a time split, a leaf node contains f_l live entries. Then each insertion adds one live entry, and a matching deletion converts a live entry into a dead one. Thus, an update increases the number of entries by one, and, if b_l is the maximum number of entries in a leaf node, it takes $b_l - f_l$ updates until the leaf node becomes overfull, causing another time split.

If an object receives an update on average every *UI* time units, a node with f_l live entries receives f_l/UI updates per time unit. Thus, the average time between time splits of a leaf node is:

$$s_0 = \frac{(b_l - f_l)UI}{f_l}. \tag{1}$$

Next, each time split at level $i - 1$ is an "update" at level i, where $i = 0$ corresponds to the leaf level. Such an update renders one live entry dead and creates a new live entry. Thus, analogous to Equation (1), we can express the average time between time splits at level i as follows.

$$s_i = \frac{(b - f)s_{i-1}}{f} = \frac{(b - f)^i(b_l - f_l)UI}{f^i f_l}.$$

This enables us to express the maximum number of times an average leaf-level entry or its ancestor bounding region at a higher level will be copied by time splits in between two updates of a corresponding object:

$$c_i = \left\lceil \frac{UI}{s_i} \right\rceil = \left\lceil \frac{f^i f_l}{(b - f)^i(b_l - f_l)} \right\rceil = \left\lceil \frac{u^i u_l}{(1 - u)^i(1 - u_l)} \right\rceil, \tag{2}$$

where $u = f/b$ and $u_l = f_l/b_l$.

Thus, at level i, c_i nodes, in addition to the live ones, have to be accessed when performing a correction. Consequently, the sole factor determining the cost of correction is u (and u_l), the average *live utilization*. The smaller it is, the fewer nodes have to be accessed during a correction. Notably, when $u \leq 0.5$ and $u_l \leq 0.5$, $c_i = 1$ for all i. This means that, on average, only one predecessor node of each live node on the path from the root to the leaf has to be accessed when correcting. Each access involves reading the node and, if a correction is actually needed, writing the node. Note that when compared to an update algorithm without correction, there may also be additional write I/O operations for the live ancestor nodes of the live leaf node were the updated entry is located. This happens if ancestor bounding regions of the corrected entry exist, are invalidated, and have to be corrected in these ancestor nodes.

Estimating the average live utilization u (u_l) is nontrivial. Its value depends on the build-up phase. In the simplest build-up of the index, all entries are inserted one after another without any intervening updates. The resulting average live fanout will be equal to the average fanout of the corresponding ephemeral index because live nodes will contain only live entries. The analytical model of Yao [1978] shows that, after a series of random insertions, u and u_l in B-trees can be approximated as $\ln 2 \approx 0.69$. Substituting these values into Equation (2), we obtain the following.

$$c_i = \left\lceil \left(\frac{\ln 2}{1 - \ln 2} \right)^{i+1} \right\rceil .$$

Thus, in such a setting, $c_0 = 3$, $c_1 = 6$, and $c_2 = 12$ which translates into large costs of corrections. Fortunately, as argued in the following, under realistic workloads that include some updates, the live utilization of the R$^{\text{PPF}}$-tree is smaller than $\ln 2$.

Initially, observe that the analytical model of Yao [1978] assumes that overfull nodes are split in half and that the resulting two nodes are equallly likely to receive future insertions. In R-trees, as well as the R$^{\text{PPF}}$-tree, an overfull node is key-split into two nodes that do not necessarily have the same number of entries. In addition, the node with more entries may be more likely to receive future insertions than the node with fewer entries. This results in a greater number of key-splits and a reduced live utilization.

Second, observe that even in an ephemeral structure, it is difficult to estimate analytically the utilization when the workload considered contains deletions in addition to insertions. Johnson and Shasha [1993] show that for B-trees, increasing the fraction of deletions in the workload leads to a decrease in the utilization.

In addition, in partially persistent structures, when deletions are present in the workload, nodes that are key-split usually contain less than b_l entries; they may contain as few as $(1-\epsilon)b_l$ entries. This further reduces the live utilization. The larger the value of ϵ is, the smaller the resulting live utilization is. This is confirmed by the performance experiments described in the next section.

Parameter ϵ can be increased only up to some limit within the constraints imposed by the partial persistence framework [Becker et al. 1996]. Consider Figure 5. A strong version underflow—which occurs when the number of entries is less than $(d+\epsilon)b_l$—causes a merge which, in turn, may cause a strong version overflow—which occurs when the number of entries exceeds $(1-\epsilon)b_l$. This is handled by a key-split which, in the worst case, might place as little as $k(1-\epsilon)b_l$ entries in one of the produced nodes, where k is the minimum utilization of the ephemeral index, which is usually equal to 0.4 in R*-trees and TPR-trees. For this to not again cause a strong version underflow, the following should hold:

$$k(1-\epsilon)b_l \geq (d+\epsilon)b_l \quad \Rightarrow \quad \epsilon \leq \frac{k-d}{1+k}. \tag{3}$$

Thus, increasing d, decreases ϵ. Note that in order to obtain efficient time-slice queries, we do not want too small a d [Tao et al. 2002]. Thus, after setting d, we set ϵ to the maximum value possible according to Equation (3). As the next section reports, experiments show that for realistic workloads, $d = 0.2$ yields a live utilization smaller than 0.5 which, in turn, results in cheap corrections.

Summarizing, it is important to note that for the workloads considered in our model, the cost of correction is independent of parameters such as the average frequency of updates, the number of objects, and the index node size. The cost of corrections depends only on the sequence of operations and the parameters d and ϵ. Also, while some corrections may be very expensive, on average they should involve a limited number of I/Os if these parameters are set right. The performance experiments described in the next section validate this claim.

7. PERFORMANCE STUDIES

This section reports on performance experiments with the $\mathrm{R}^{\mathrm{PPF}}$-tree. The generation of workloads for experiments and other settings for the experiments are described first, followed by a presentation of the main results of the experiments.

7.1 Experimental Setting

The $\mathrm{R}^{\mathrm{PPF}}$-tree was implemented in C++, and the experiments were run using generated data. As the $\mathrm{R}^{\mathrm{PPF}}$-tree is the result of applying partial persistence to an adapted TPR-tree, to test the $\mathrm{R}^{\mathrm{PPF}}$-tree, we first used the same experimental settings as were used for the performance experiments with the TPR-tree [Šaltenis et al. 2000]. In addition, we experimented with workloads generated using Brinkhoff's moving object generator [Brinkhoff 2002]. We start by reviewing the settings for the TPR-tree workload generator. Additional details can be found elsewhere [Šaltenis et al. 2000].

In all experiments, the indices were subjected to workloads that intermix queries and update operations. Initially, an index is empty. It is then populated gradually, with entries added when simulated objects report their first positions. After an object enters the simulation, it reports trajectory updates until the end of the workload. The number of objects in the simulation is 100,000, and the simulation is run until 1,000,000 operations, including the initial insertions, are generated.

The workloads simulate objects, such as cars, moving in a network of straight routes connecting 20 destinations in a fully-connected network. The destinations are distributed in the space with dimensions 1000km $\times$ 1000km. Objects move with maximum speeds ranging from 0 to 3km/min. Objects accelerate and decelerate at the beginnings and ends of their routes. This acceleration/deceleration behavior guarantees that objects do not move according to linear functions, making corrections to the last-recorded trajectory segments necessary (see Figure 2).

The workload generation algorithm distributes the updates of the positions of the objects so that the average time interval between two subsequent updates of an object is approximately equal to a given parameter *UI*, which we set to 30 minutes in most experiments.

To test the R^{PPF}-tree in a scenario that corresponds to the tracking cars in a real urban road network, a different set of workloads was generated using Brinkhoff's generator. We slightly modified this generator so that, given a desired number of objects N and a length of the history Hl, for the first tenth of the history, moving objects are gradually introduced, inserting $N/(Hl/10)$ new objects per time unit. After this phase, the number of objects remains constant. Thus, if one object is deleted upon reaching its destination, another object is inserted at the same time unit. We set $N = 25{,}000$, and $Hl = 400$ for most of our experiments.

We use a representation of the road network of Oldenburg, Germany as an input to the generator. The map covers an area of $23{,}572 \times 26{,}915$ space units. This corresponds approximately to a 14.6km × 12.8km area. Assuming that each time unit in the generator corresponds to six seconds, we set the maximum velocity of objects so that it corresponds to a little more than 120km/h (parameter "max. speed div." of the generator is set to 130).

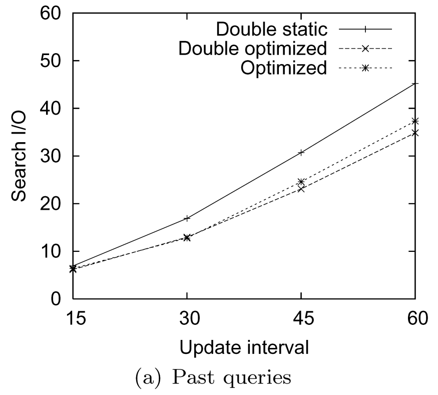
The workload generated by the Brinkhoff generator includes updates of the positions of all objects at each time unit. To generate a more realistic updating pattern, we filter the workload using the so-called vector update policy [Čivilis et al. 2004]. Given a threshold parameter, a linear function of movement is updated only if the distance between the generated position and the position predicted by the function is more than the threshold. Otherwise, the generated position is discarded. Note that this corresponds to the behavior assumed in Figure 2.

Queries are generated intermixed with insertions, deletions, and updates. In the original TPR-tree workloads, one query is generated every 100 update operations. In the Brinkhoff workloads, the filtering step generates 20 queries each time unit. If not stated otherwise, queries are timeslice queries. The spatial part of each query is a randomly placed square occupying 0.25% of the total data space. Equal amounts of past and current/future queries are generated, and future queries have times between the current time CT and CT + $UI/2$. In the Brinkhoff workloads, UI is the current average update interval of object updates as computed by the filtering step. In the experiments, we use TPR-tree workloads, which offer more control of the workload parameters, unless stated otherwise.

We experimented with data page sizes of 4K and 8K, where 8K was the default value. An LRU buffer of 100 pages is used. Unless noted otherwise, d, the minimum fraction of live entries in a node, is set to be 20% of the total capacity of the node. Update and search performance are measured in numbers of I/O operations.

7.2 Bounding Rectangles

In Section 5.2, we proposed three types of time-parameterized bounding rectangles with different properties. Optimized TPBRs and double TPBRs with static heads are the most compact of the three types which is good for fanout and thus query and update performance. But optimized TPBRs cannot be tightened and are complex to compute, both of which adversely affect performance.



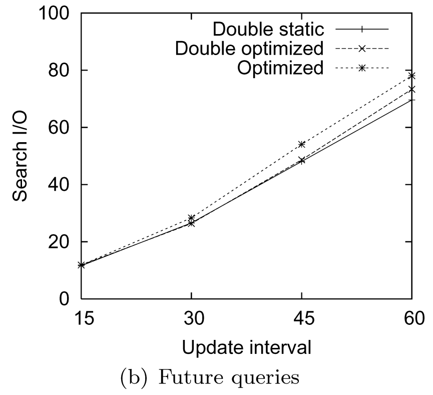
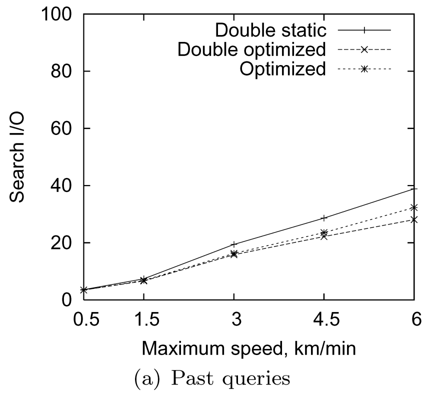


Fig. 15. Past and future query performance for different TPBRs and varying *UI*.



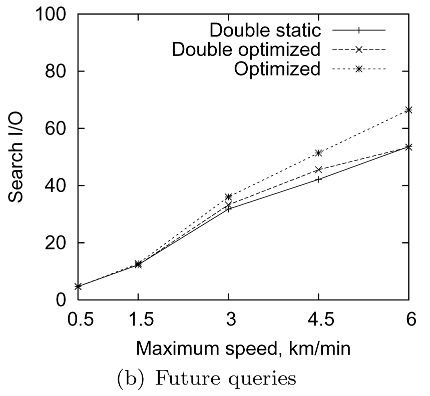
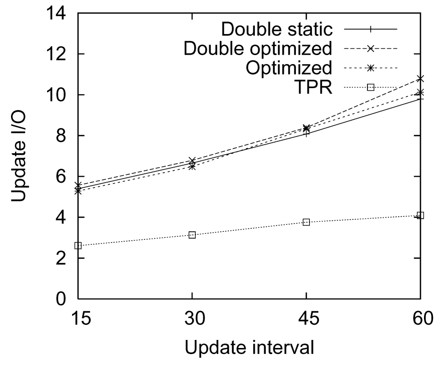


Fig. 16. Past and future query performance for different TPBRs and varying maximum speed.

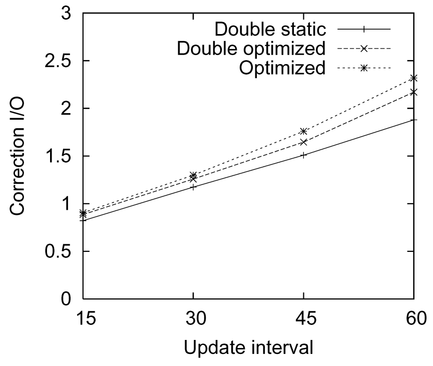
Double optimized TPBRs reduce fanout on average by 33%, but can bound the data better and can be tightened. Like optimized TPBRs, their computation involves complex floating-point geometry and sorting. Finally, double TPBRs with static heads might not bound the data as well as double optimized TPBRs, can be tightened, and are very easy to compute.

To explore how these different types of time-parameterized bounding rectangles affect the query and update performance, we performed two sets of experiments. In one set, we experimented with workloads that vary *UI*, the average interval between two updates of an object. In another set, the maximum speed of the simulated objects was varied. We studied separately the average performance of past queries and the average performance of current/future queries. Figures 15(a) and 15(b) show the results of the experiments with varying *UI*, and Figures 16(a) and 16(b) provide the graphs with varying maximum speed.

The query performance of all three types of bounding rectangles is comparable. When past queries are considered, the performance of double TPBRs with static heads is somewhat worse than the performance of the other two types of bounding rectangles, because rectangles bound parts of the trajectories less accurately than trapezoids. For future queries, the performance of double TPBRs

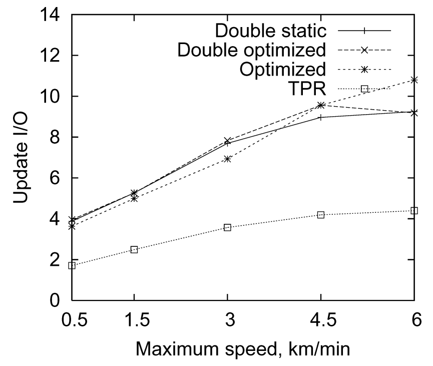


(a) Update performance

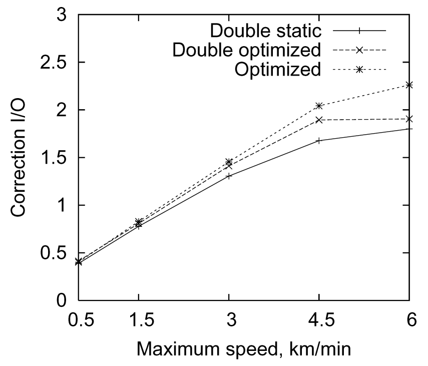


(b) Cost of correction

Fig. 17. Update performance and cost of correction for different TPBRs and varying *UI*.



(a) Update performance



(b) Cost of correction

Fig. 18. Update performance and cost of correction for different TPBRs and varying maximum speed.

does not degrade as fast as the performance of the optimized TPBRs when the update interval or maximum speed increases. This is as expected because longer intervals between updates mean that TPBRs are allowed to expand more before they are recomputed. Similarly, having objects that are faster means that bounding rectangles expand faster. Therefore, double TPBRs, which support tightening, have the advantage for such workloads. The advantages of tightening in this case even outweigh the negative effects of the reduced fanout of non-leaf nodes, which is caused by the larger sizes of entries recording double TPBRs.

In the same set of experiments, the performance of update operations was measured. The results are reported in Figures 17(a) and 18(a) which show that the update costs of indices with different types of TPBRs are very similar. This is as expected because experiments show that the main part of the update cost is the cost of search during deletion.

Next, Figures 17(b) and 18(b) show how many I/O operations on average were performed in relation to correcting the changed trajectory and checking/updating all its ancestors up the tree (in the left and left-up phases of the update algorithm). Notice that when compared with the other two types of

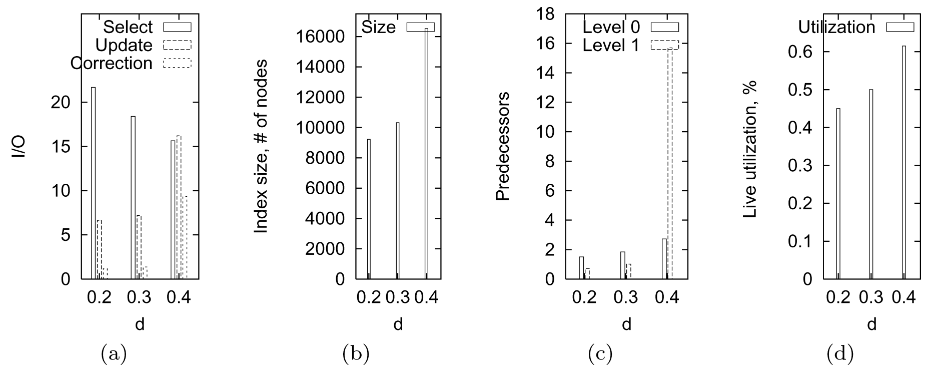


Fig. 19. Characteristics of R^{PPF}-tree for varying d: (a) I/O performance, (b) index size, (c) number of predecessor nodes accessed, and (d) average live utilization of leaf nodes.

TPBRs, the usage of double TPBRs with static heads leads to slightly less I/O operations being spent on the correction part of update. This can be explained by the rectangular heads of double TPBRs being invalidated less frequently by the correction of trajectory segments than the more accurate trapezoids. This, in turn, leads to less corrections of TPBRs.

The findings of the described performance experiments leads to the choice of double TPBRs with static heads as the preferred type of TPBRs as they offer similar I/O performance and are much simpler to compute than the other two types of TPBRs. In the following, we use double TPBRs with static heads as the time-parameterized bounding rectangles of the R^{PPF}-tree.

7.3 Update Performance

The correction of the most recent trajectory segment, as defined in Section 5.4 and studied analytically in Section 6, incurs additional I/O on each update. We performed a number of experiments to explore this overhead and to learn how it is affected by index and workload parameters. Figures 17(b) and 18(b), when compared to Figures 17(a) and 18(a), show that the correction of the last-recorded trajectory segment accounts for less than a fourth of the total update cost.

The analytical study indicates that the correction overhead of updates is dependent on the average number of time splits that occur in between two updates of an object. To vary the number of time splits, we vary d. For each value of d (0.2, 0.3, and 0.4), we chose the largest possible ϵ as explained in Section 6. The frequency of time splits is inversely proportional to ϵ. Thus, this frequency increases with an increasing d.

Figure 19(a) plots the update and query performance against d. As expected, when d increases, the total number of time splits increases (from 8,504 for $d = 0.2$ to 17,948 for $d = 0.4$, not shown in the figure). This results in the decrease by a factor of almost 2.5 in the average number of I/O operations per update as shown in the figure. This decrease is mainly caused by a decrease in the cost of correction of the last-recorded trajectory segment, as also shown in the figure.

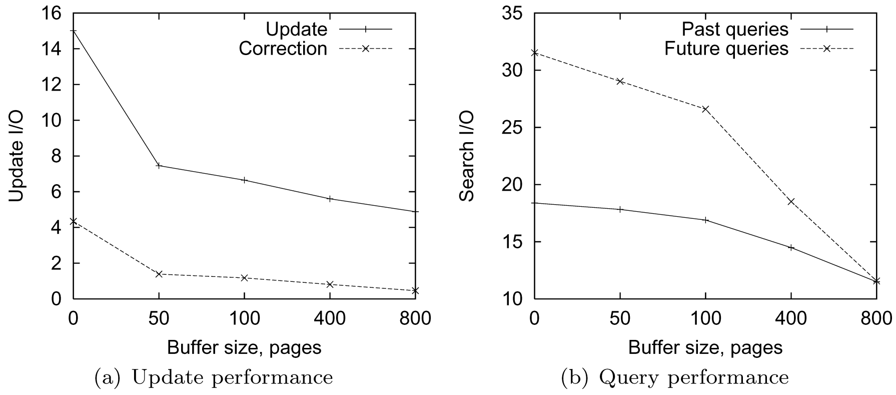


Fig. 20. Update and query performance of R^{PPF}-tree for varying buffer size.

To see how accurately the cost model of Section 6 captures the overhead of corrections, we recorded the average number of predecessor nodes that are accessed on the leaf and pre-leaf levels. The numbers for the same workload as in Figure 19(a) are shown in 19(c). Level 0 is the leaf level and level 1 is the pre-leaf level. Figure 19(d) reports the corresponding average live utilization of nodes at the end of the workload.

The figures show that the cost model correctly predicts that, as soon as the live utilization of nodes exceeds 0.5 (for $d = 0.4$), the number of accessed predecessors grows exponentially with the level number. Note that the cost model predicts that only one predecessor node will be accessed if the live utilization is below 0.5. As the graph shows, on average more that 1.5 predecessor nodes are accessed for the workload of $d = 0.2$. This deviation is probably due to the simplified assumptions of the theoretical model where no other structural changes happen except time splits. In actual workloads, key-splits and merges do actually happen, and they cause additional time splits, increasing the average number of copies of each entry.

While decreasing d leads to improved update performance, timeslice query performance is naturally decreased. This is so because smaller values of d imply a smaller average fanout of the index as seen at the time of query. Nevertheless, the negative effect on query performance is less marked than the positive effect on update performance. Also, as Figure 19(b) shows, smaller values of d result in smaller index sizes.

To learn how much the cost of updates can be reduced by increasing the size of the main memory buffer, we experimented with varying buffer sizes. Figure 20(a) shows the results. As expected, update costs decrease substantially when using a buffer of moderate size compared to using no buffer or using a small buffer. However, substantially larger buffers are not very effective. This is because each update operation includes a number of write I/Os. Next, as expected, the search costs decrease (Figure 20(b)) for increasing buffer sizes, especially for future queries. This is because most of the live nodes become cached in the buffer.

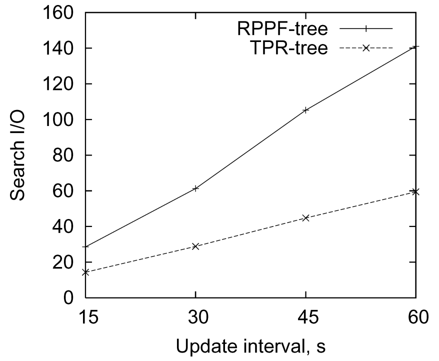


Fig. 21. Query performance of R^{PPF}-tree and TPR-tree for varying *UI*.

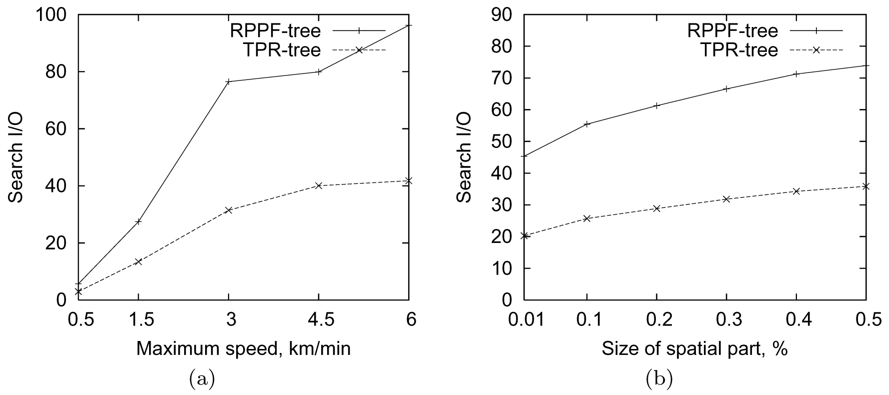


Fig. 22. Query performance of R^{PPF}-tree and TPR-tree for varying maximum speed and varying query size.

7.4 The R^{PPF}-Tree Versus the TPR-Tree

Naturally, recording history in the R^{PPF}-tree does not come for free when both update and query costs are considered. To quantify this cost, we performed the same set of experiments as described earlier but with the TPR-tree. The parameters *UI* and maximum speed were varied. Figures 17(a) and 18(a) show that performing update operations on the TPR-tree is more than twice as fast as doing the same updates on the R^{PPF}-tree.

In a separate set of experiments, we studied the query performance in the R^{PPF}-tree in comparison to the TPR-tree. We ran a number of workloads of updates on both the R^{PPF}-tree and the TPR-tree. To ensure a fair comparison, after the running of a workload, the buffer was cleared and reduced to 50 pages. Ten thousand current/future queries were then run. Figures 21, 22(a), and 22(b) show the results of these experiments.

These experiments report query performance when varying *UI*, the maximum speed, and the spatial extent of the queries (expressed in percents of the total data space). In the experiments covered by Figures 21 and 22(a), the query size is set at 0.2%.

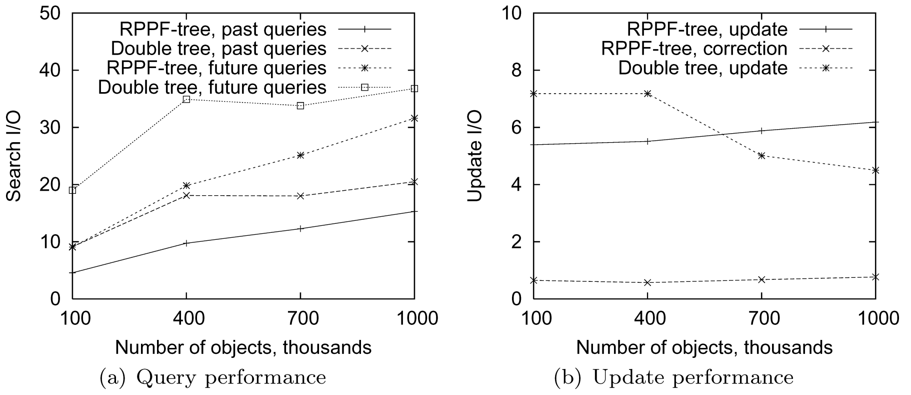


Fig. 23. Query and update performance of R^{PPF}-tree and double tree for a varying number of objects.

Similar to the experiments on the update operations, the presented graphs show that the average numbers of I/O operations per query performed on the R^{PPF}-tree are a little bit more than twice those of the TPR-tree. This is expected because the fanout of the live part of the R^{PPF}-tree is lower than the fanout of the TPR-tree.

7.5 The R^{PPF}-Tree Versus the Double-Tree Index

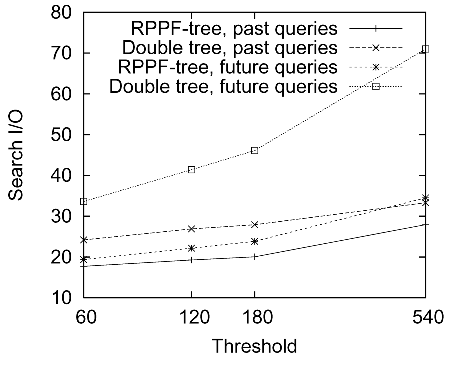
In the last batch of experiments, we compared the R^{PPF}-Tree to the double-tree index described in Section 4. This approach maintains two trees: an R*-tree that indexes the closed, past segments of trajectories and a modified TPR-tree that indexes the open-ended, current linear functions.

In a first set of experiments, we explore how the performance of both indices scale with an increase in the number of indexed objects. The standard workloads were used, but instead of performing one million operations in each workload, we let each object on average make four updates. For a workload with one million objects, this results in four million update operations. Figure 23(a) shows the past and future query performance for both indices. For these settings, the R^{PPF}-tree substantially outperforms the double-tree index.

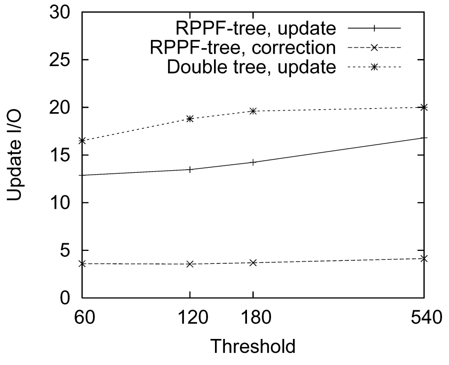
Note that, as for the R^{PPF}-tree, the future queries are more expensive than the past queries in the double-tree index. This is in part explained by the larger numbers of objects that future queries return. The average size of the result of a past query is smaller than the average size of the result of a future query because a subset of the past queries query the build-up phase of the history where the number of objects is small. Similarly, the increases in query costs as the number of objects increases are explained by the matching increases in query result sizes.

Finally, observe that the modified TPR-tree in the double-tree index performs worse than the live part of the R^{PPF}-tree which, as shown earlier, performs worse than a regular TPR-tree. The modified TPR-tree has poor performance because its TPBRs cannot be tightened (see Figure 4).

Figure 23(b) shows how the update performance scales as the number of indexed objects increases. While the update costs of both indices are



(a) Query performance



(b) Update performance

Fig. 24. Query and update performance of R^{PPF}-tree and double tree for varying update thresholds.

comparable, interestingly, the cost of the double-tree index decreases as the number of objects increases from 400K to 1000K. This behavior occurs because the cost of deletion from the modified TPR-tree decreases. To delete an object, the object is first found by searching the tree. Because of the nontight TPBRs in the modified TPR-tree, there is substantial overlap among TPBRs at the current time. Thus, several paths down the tree may end up being traversed. A TPBR is excluded from search if the velocity vector of the searched object is not contained in the velocity bounds of the TPBR. Having more objects means that the objects in an arbitrary node have more similar velocities which yields TPBRs with tighter velocity bounds. This enables the search phase of the deletion to prune more TPBRs when more objects are indexed.

In the next set of experiments, we used the Brinkhoff workloads. Figures 24(a) and 24(b) show search and update performance when the update threshold is varied. The page size in these experiments, was set at 4K. The threshold value of 60 units corresponds to tracking the simulated cars with an accuracy of approximately 37 meters. Note that larger update thresholds correspond to larger average update intervals. The update threshold of 60 resulted in an average update interval of 7.3 time units ($\approx$ 48 seconds), while the threshold value of 540 resulted in an average update interval of 20.9 time units (more than two minutes).

The graphs show similar, though less pronounced, performance trends for the R^{PPF}-tree as the graphs where *UI* is being varied using the workloads generated by the TPR-tree generator (see Figure 17(a)). Again, the double-tree index performs worse, for queries as well as updates.

The Brinkhoff workloads were also used while varying the spatial extent of the queries. The update threshold was set to 60 in these experiments. Figure 25(a) shows the results. Note that the R-tree of the double-tree index is less robust to the increase in the size of the query than the R^{PPF}-tree.

Finally, Figure 25(b) shows the query performance of both indices for window queries. The workloads generated by the TPR-tree generator were used in these experiments. Although partial persistence is not geared for window queries, R^{PPF}-tree shows good performance for windows queries. Notice that,

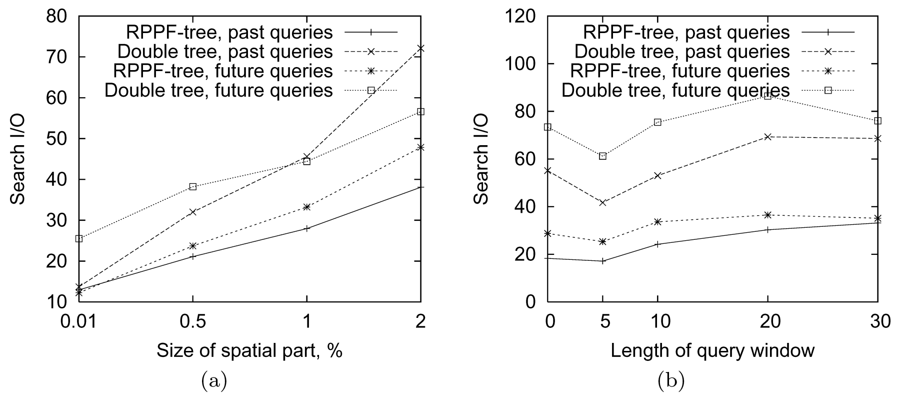


Fig. 25. Query performance of R^{PPF}-tree and double tree for varying query size and varying query window length.

as explained in Section 5.3, we use a hash table of page ids to avoid visiting the same page twice during the execution of a window query.

8. SUMMARY AND RESEARCH DIRECTIONS

With the proliferation of wireless communications and geo-positioning as motivation, this article presents and evaluates an index that is able to capture the past, present, and anticipated future positions of moving objects where continuous polyline representation of the former positions is supported and where the latter positions are represented as linear functions. To the best of the authors knowledge, no other such indices exist.

The index adapts the time-parameterized bounding rectangles of existing R-tree-based indexing techniques for the current and future positions of moving objects to the framework of partial persistence. This includes proposals for several kinds of bounding rectangles, each with different characteristics. It extends the partial persistence framework, which inherently supports transaction time, to support valid time for applications where moving-object position samples, given as linear functions of time, arrive in time order and predict future positions. The index supports objects moving in one, two, and three dimensions, and it is applicable to continuous variables other than geographical position. An analytical study as well as empirical studies with an implementation of the indexing technique are reported that offer insights into the performance characteristics of the index.

As an interesting future research direction, the partial persistence framework presented in this article could be applied to other indexing techniques that use linear functions to capture continuous change. For example, although the recently proposed STRIPES [Patel et al. 2004] and B^x-tree [Jensen et al. 2004] techniques are very different from the TPR-tree based techniques, it may be possible to apply techniques similar to the ones presented in this article to these two in order to extend them to accurately record the history of movement.

As mentioned in Section 3.2, the recently proposed BB^x-tree [Lin et al. 2005] is capable of indexing the past, present, and future positions of moving objects, but the trajectories resulting from online updates of the index consist of disconnected and slightly incorrect line segments. Performing the corrections of trajectory segments described in this article, each update of the BB^x-tree would involve physical deletion and reinsertion of the last-recorded segment in possibly more than one B-tree. It would be interesting to compare the R$^{\mathrm{PPF}}$-tree and the BB^x-tree with corrections with respect to update and search performance.

Another interesting direction would be to pursue different aspects of effective main-memory use in the context of the R$^{\mathrm{PPF}}$-tree. For example, the use of concepts within cache consciousness and obliviousness as well as techniques from the buffer-tree of Arge may be of interest.

Next, while in the performance experiments of this paper, the R$^{\mathrm{PPF}}$-tree was tested using workloads simulating continuously moving objects, the index is also suitable for recording other kinds of continuously changing variables. When such data become available, it would be interesting to explore how the index performs for workloads consisting of sensor data from scenarios where sensor networks track different physical characteristics of the environment such as temperature, barometric pressure, humidity, or ambient light.

REFERENCES

Agarwal, P. K., Arge, L., and Erickson, J. 2000. Indexing moving points. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS)*. 175–186.

Agarwal, P. K. and Har-Peled, S. 2001. Maintaining approximate extent measures of moving points. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 148–157.

Basch, J., Guibas, L., and Hershberger, J. 1997. Data structures for mobile data. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 747–756.

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. 1996. An asymptotically optimal multiversion B-tree. *VLDB J. 5*, 4, 264–275.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 322–331.

Brinkhoff, T. 2002. A framework for generating network-based moving objects. *GeoInformatica 6*, 2, 153–180.

Cai, M. and Revesz, P. Z. 2000. Parametric R-tree: An index structure for moving objects. In *Proceedings of the 10th International Conference on Management of Data (COMAD)*.

Chakka, V. P., Everspaugh, A., and Patel, J. M. 2003. Indexing large trajectory data sets with SETI. In *Online Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*.

Chon, H. D., Agrawal, D., and El Abbadi, A. 2002. Query processing for moving objects with space-time grid storage model. In *Proceedings of the 3rd International Conference on Mobile Data Management (MDM)*. 121–128.

Čivilis, A., Jensen, C. S., Nenortaitė, J., and Pakalnis, S. 2004. Efficient tracking of moving objects with precision guarantees. In *Proceedings of the 1st International Conference on Mobile and Ubiquitous Systems: Networking and Services*. 164–173.

Čivilis, A., Jensen, C. S., and Pakalnis, S. 2005. Techniques for efficient tracking of road-network-based moving objects. *IEEE Trans. Knowl. Data Eng. 17*, 5, 698–712.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. 1989. Making data structures persistent. *J. Comput. Syst. Sci. 38*, 1, 86–124.

Hadjieleftheriou, M., Kollios, G., Tsotras, V. J., and Gunopulos, D. 2002. Efficient indexing of spatiotemporal objects. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*. 251–268.

JENSEN, C. S., LIN, D., AND OOI, B. C. 2004. Query and update efficient B+-tree based indexing of moving objects. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. 768–779.

JOHNSON, T. AND SHASHA, D. 1993. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *J. Comput. Syst. Sci. 47*, 1, 45–76.

KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. J. 1999. On indexing mobile objects. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS)*. 261–272.

KOLLIOS, G., TSOTRAS, V. J., GUNOPULOS, D., DELIS, A., AND HADJIELEFTHERIOU, M. 2001. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. Knowl. Data Eng. 13*, 5, 758–777.

KOLOVSON, C. P. AND STONEBRAKER, M. 1991. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 138–147.

KUMAR, A., TSOTRAS, V. J., AND FALOUTSOS, C. 1998. Designing access methods for bitemporal databases. *IEEE Trans. Knowl. Data Eng. 10*, 1, 1–20.

LIN, D., JENSEN, C. S., OOI, B. C., AND ŠALTENIS, S. 2005. Efficient indexing of the historical, present, and future positions of moving objects. In *Proceedings of the 6th International Conference on Mobile Data Management (MDM)*. 59–66.

LOMET, D. B. 1991. Grow and post index trees: Roles, techniques and future potential. In *Proceedings of the International Symposium on Advances in Spatial Databases*. 183–206.

PAPADOPOULOS, D., KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. J. 2002. Indexing mobile objects on the plane. In *Proceedings of the 5th International Workshop on Mobility in Databases and Distributed Systems (MDDS)*. 693–697.

PATEL, J. M., CHEN, Y., AND CHAKKA, V. P. 2004. STRIPES: An efficient index for predicted trajectories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 637–646.

PFOSER, D., THEODORIDIS, Y., AND JENSEN, C. S. 2000. Novel approaches in query processing for moving object trajectories. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. 395–406.

PORKAEW, K., LASARIDIS, I., AND MEHROTRA, S. 2001. Querying mobile objects in spatio-temporal databases. In *Proceedings of the 8th International Symposium on Advances in Spatial and Temporal Databases (SSTD)*. 59–78.

PROCOPIUC, C. M., AGARWAL, P. K., AND HAR-PELED, S. 2002. STAR-tree: An efficient self-adjusting index for moving objects. In *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments (ALENEX)*. 178–193.

ŠALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LOPEZ, M. A. 2000. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 331–342.

ŠALTENIS, S. AND JENSEN, C. S. 2002. Indexing of moving objects for location-based services. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*. 463–472.

SISTLA, A. P., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. 1997. Modeling and querying moving objects. In *Proceedings of the 13th IEEE International Conference on Data Engineering (ICDE)*. 422–432.

SONG, Z. AND ROUSSOPOULOS, N. 2001. Hashing moving objects. In *Proceedings of the 2nd International Conference on Mobile Data Management (MDM)*. 161–172.

SONG, Z. AND ROUSSOPOULOS, N. 2003. SEB-tree: An approach to index continuously moving objects. In *Proceedings of the 4th International Conference on Mobile Data Management (MDM)*. 340–344.

SUN, J., PAPADIAS, D., TAO, Y., AND LIU, B. 2004. Querying about the past, the present and the future in spatio-temporal databases. In *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*. 202–213.

TAO, Y. AND PAPADIAS, D. 2001. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*. 431–440.

TAO. Y., PAPADIAS, D., AND SUN, J. 2003. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*. 790–801.

Tao, Y., Papadias, D., and Zhang, J. 2002. Cost models for overlapping and multiversion structures. *ACM Trans. Database Syst. 27*, 3, 299–342.

Tayeb, J., Ulusoy, Ö., and Wolfson, O. 1998. A quadtree based dynamic attribute indexing method. *Computer J. 41*, 3, 185–200.

Van den Bercken, J. and Seeger, B. 1996. Query processing techniques for multiversion access methods. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*. 168–179.

Wolfson, O., Xu, B., Chamberlain, S., and Jiang, L. 1998. Moving objects databases: Issues and solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*. 111–122.

Wolfson, O., Chamberlain, S., Dao, S., Jiang, L., and Mendez, G. 1998. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*. 588–596.

Wolfson, O., Sistla, A. P., Chamberlain, S., and Yesha, Y. 1999. Updating and querying databases that track mobile units. *Distrib. Parall. Datab. 7*, 3, 257–387.

Yao, A. C.-C. 1978. On random 2-3 trees. *Acta Inf. 9*, 2, 159–170.

Received July 2004; revised May 2005; accepted August 2005