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

Mindaugas Pelanis, Simonas Šaltenis, and Christian S. Jensen

July 23, 2004

TR-78

A TIMECENTER Technical Report

Title	Indexing the Past, Present and Anticipated Future Positions of Moving Objects Copyright © 2004 Mindaugas Pelanis, Simonas Šaltenis, and Christian S. Jensen. All rights reserved.
Author(s)	Mindaugas Pelanis, Simonas Šaltenis, and Christian S. Jensen
Publication History	July 2004, a TIMECENTER Technical Report

Abstract

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 on-line 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 essence in this scenario.

Past indexing techniques capture the position of an object up until the time of the most recent position sample, or they 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 paper offers an indexing technique capable of capturing the positions of moving objects at all points in time. The index substantially extends partial persistence techniques, which support transaction time, to support valid time for monitoring applications. The performance of a query is independent of the number of past position samples stored for an object. No existing indices exist with these characteristics.

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 the generally improved performance/price ratio. Perhaps most importantly, wireless communications 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 to 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 transfer of data between disk and main memory cannot keep pace.

In this paper, 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., [5, 12, 16, 18, 22, 23, 30]), or they support only the positions from the current time and into the future (e.g., [1, 2, 14, 21, 24, 25, 26, 32, 31]). No index combines these capabilities, with the exception of one recent proposal that addresses approximate query answering [29]. 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 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 [25]. The TPR-tree supports the querying of the current and anticipated future positions of moving objects, the positions being 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.

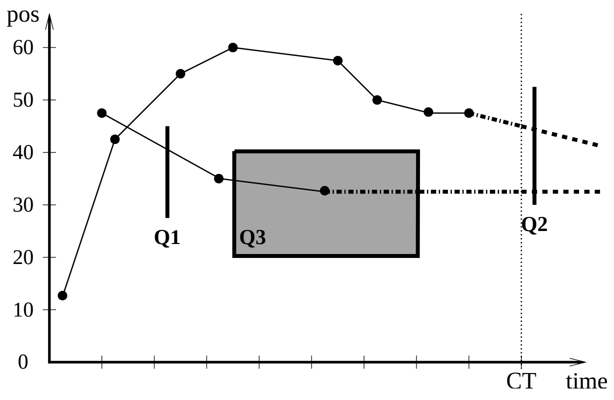

Figure 1: Querying the Positions of Moving Objects

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 samples 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 when 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 either one- or two-dimensional space, and we most often use two-dimensional terminology.

Performance experiments with the paper'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.

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 functionality of 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. Then follows the section that describes structure of the R^{PPF}-tree and the algorithms that maintain the index structure under updates and allow querying it. Section 6 explores performance-related properties of the indexing technique, and Section 7 summarizes and points to research directions. An appendix gives some background information about fundamental design choices underlying the paper's proposal.

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. 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. 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. 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.

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 [10, 11]. 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, e.g., 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.

Figure 2 shows the trajectory of an object moving in one-dimensional space. This could be a car with its position being the distance traveled along a highway. Here $u1, \ldots, u5$ 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 an object when the next update is processed.

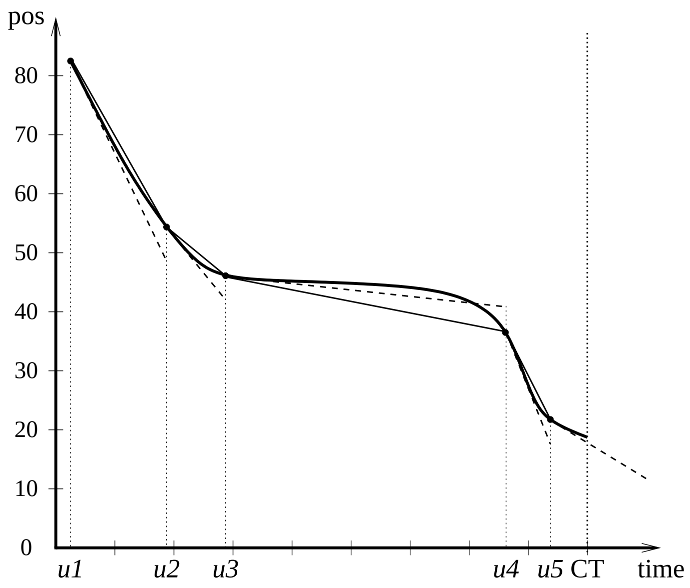

Figure 2: Trajectory of a One-Dimensional Moving Object

Note also 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 in dashed lines). 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 $(d+1)$-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 paper 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, i.e., 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., [16, 17, 18]). Pfoser et al. [22] suggest two variations of the R-tree for polyline indexing: TB-tree and STR-tree. Both attempt, to varying degrees, to group together segments from the same polyline, the goal being to answer 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. [23] suggest to allow one or more line segments of a polyline to extend into the future. These segments then represent predictions. They also suggest to use a standard R-tree for 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 post-processing 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 timeslice performance of an R-tree on line segments decreases as the number of updates grows. The R-tree variant proposed by Cai and Revesz [7] has this general problem. In our quite different proposal, timeslice performance is unaffected by the number of updates.

Kumar et al. [18] apply partial persistence to the R-tree, the objective being to support 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 $\mathrm{R}^{\mathrm{PPF}}$-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 [12, 16] assume a static database of object evolutions and consider the partitioning of these evolutions into smaller time intervals, the goal being to reduce 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 [30] index the same data with both a standard R-tree and a variant of the partially persistent R-tree. The combined index is called the MV3R-tree. It is capable of indexing the past trajectory data. To address similar problem, Chaka et al. [8] propose SETI—a two-level indexing structure which separates indexing of spatial and temporal dimensions. Both SETI and MV3R-tree can index only the positions of an object up until the time of the most recent sample, so the proposals do not solve the more general problem considered here.

Song and Roussopoulos [27, 28] 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 Present/Future Position Indices

A number of approaches for indexing of the current and predicted future positions of moving points exist that may be considered as candidates for extension to also index past positions. Tayeb et al. [32] use PMR-Quadtrees, Kollios et al. [15, 20] employ the so-called dual data transformation, Agarwal et al. [1] use the ideas of so-called kinetic data structures [3], and Chon et al. [9] 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. [29] 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 (cf. Section 2.2), a stochastic method is used to predict the future based on the recent past. The discussed approach, to our knowledge, the only proposal that combines indexing of the past, present, and future positions of moving objects in one data structure. However, it addresses a different problem than the one considered in this paper.

Three proposals for indexing the current and the predicted future positions of objects build on the ideas of the TPR-tree. Procopiuc et al. [24] 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 [26] propose the $\mathrm{R}^{\mathrm{EXP}}$-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. [31] 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. [21] 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. [14] 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, and new phases emerge. It is unclear whether it is possible to extend these approaches to solve the problem considered in this paper, 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 bounding rectangles 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.

Figure 3 shows three moving points in one-dimensional space (i.e., $m = 1$) together with their one-dimensional bounding rectangle (i.e., a bounding interval). The figure shows that answering a window query in the TPR-tree involves the checking for intersection between an $(m+1)$-dimensional rectangle and a trapezoid—a query and a bounding rectangle.

In addition to the use of time parameterized bounding rectangles, the TPR-tree differs from the R*-tree in how its insertion algorithms group points into nodes. The R*-tree aims to minimize the areas, overlaps, and margins of bounding rectangles when 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 a time integral 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

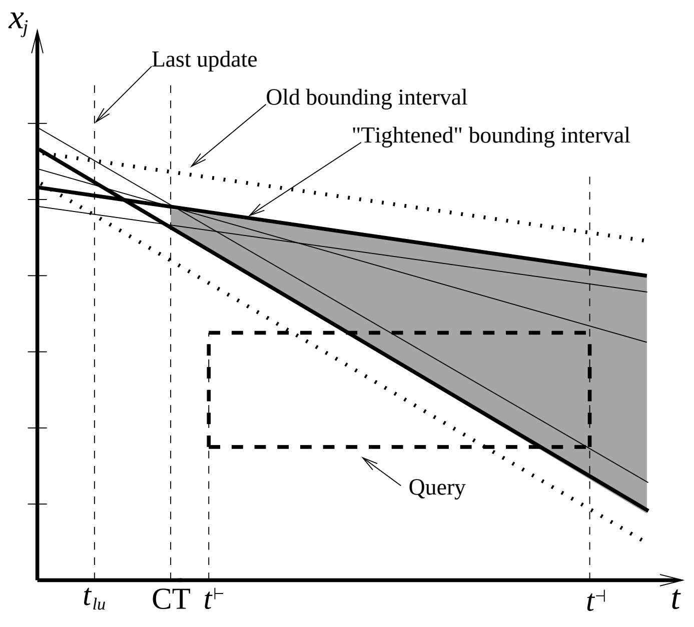

Figure 3: A Bounding Interval and a Query in the TPR-Tree

bounding intervals at times t_{lu} and CT). Performance experiments show that this recomputation, which we term "tightening," is essential in order to achieve query performance that does not degrade with time [25]. Note also that tightening produces time-parameterized bounding rectangles (TPBRs) that are not bounding prior to the current time.

4 Separate Indexing of the Past and Present/Future

This paper 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., [1, 2, 14, 21, 24, 25, 26, 32, 31]) and can be reused. For the former type of indexing, we may consider using some type of 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 the 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 of the most recent update of the least recently updated object, i.e., 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 a trajectory 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).

Thus, all updates to the database after t_{mru} must be stored in some other data structure that we may term a near-past structure (NP-structure). This data structure can become arbitrary large, as arbitrary many updates per each object can happen in-between t_{mru} and the current time. Also, this data structure (or, alternatively, a 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. Depending on the nature of the NP-structure, a separate index to support the future queries may also be needed.

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., [22]). Such an index is built on line segments that make up trajectories.

The last, open-ended parts of the trajectories (from the last update until the current time—recall Figure 2) could be stored in a modified TPR-tree or TPR-tree-type structure (normally, these indices only capture positions from the current time and onwards, so modification is needed). The modified TPR-tree must store insertion times in all entries. For a time-parameterized bounding rectangle (TPBR) in such a tree, the insertion time is equal to the minimum of the insertion times of the bounded entries. 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. Thus, tightening cannot be performed in this modified TPR-tree, reducing its querying and update performance.

As the time periods covered by the modified TPR-tree and the past R-tree index overlap, both have to be queried to answer past queries. Distant-past queries do not incur any I/O operations on the TPR-tree, as the search in the TPR-tree stops after examining the insertion times of the entries in the memory resident root of the TPR-tree. Similarly, no I/Os are done on the past R-tree for queries after the most recent update. Nevertheless, most of the near-past queries will incur I/O operations in both indices. A more significant disadvantage of this approach, than that of querying two indices, is that the 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 queries.

5 Structure and Algorithms

This section presents the data structures and algorithms associated with the R$^{\text{PPF}}$-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 when the last sample was received, and we need to re-predict the future position. This limited need for corrections enables us to support valid time by an extended notion of partial persistence.

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. The application of partial persistence to an existing indexing technique is not a mechanical process, but involves non-trivial and 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 [19]. 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. 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. Split handles over-full nodes, thus enabling the tree to grow. 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, where d is 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 are 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, where $b = k \cdot d$ and where b is the node capacity and k is some 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. 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 end-time of an alive data entry to t. Before writing the node to disk, algorithm *AssureInvariants*, sketched in Figure 4, ensures that the invariants of partial persistence are maintained.

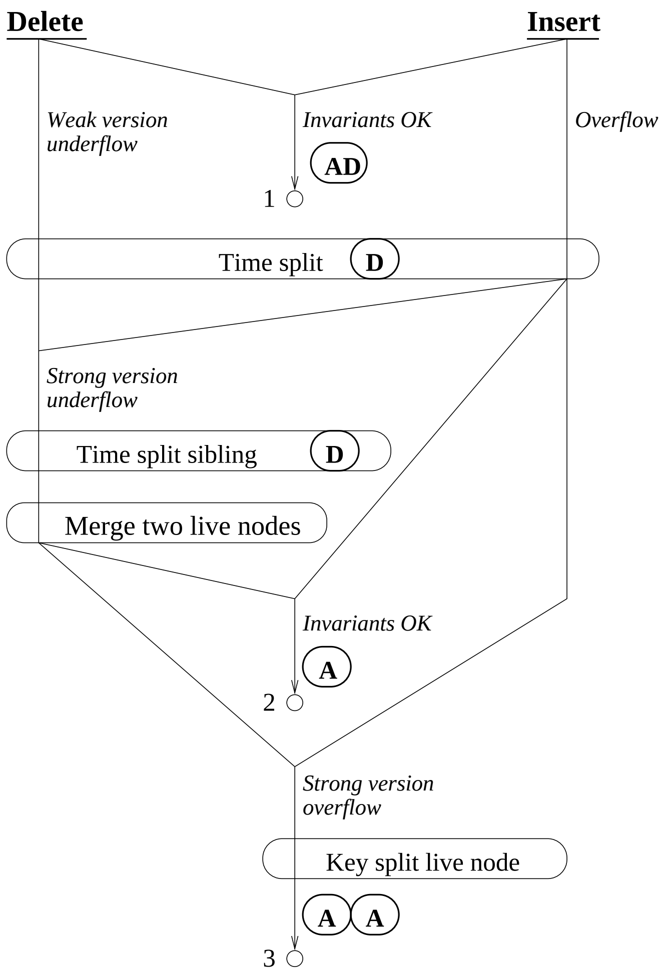

Figure 4: Algorithm *AssureInvariants*

If the node contains no less than d 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 4, 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 life time 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* [4, 18]). 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, 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 $d + e$ and $b - e$, where e is a predetermined constant. This is the *strong version condition*. If a time split leads to less than $d + e$ 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 that 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 $b - e$ 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 4.

As Figure 4 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 4) 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 each of weak version underflow, satisfied invariants, and 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 [4] 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:

$$([x_1^{\vdash}, x_1^{\dashv}], \ldots, [x_d^{\vdash}, x_d^{\dashv}]; [v_1^{\vdash}, v_1^{\dashv}], [v_d^{\vdash}, v_d^{\dashv}])$$

Here,

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

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, i.e., when $t^{\vdash}$ of the TPBR is CT (denoted by "A" in Figure 4). 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 4). It is to address this problem that we investigate different modifications of TPBRs.

Figure 5 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 left column of the figure demonstrates how the TPBRs of the TPR-tree can be used in the 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.

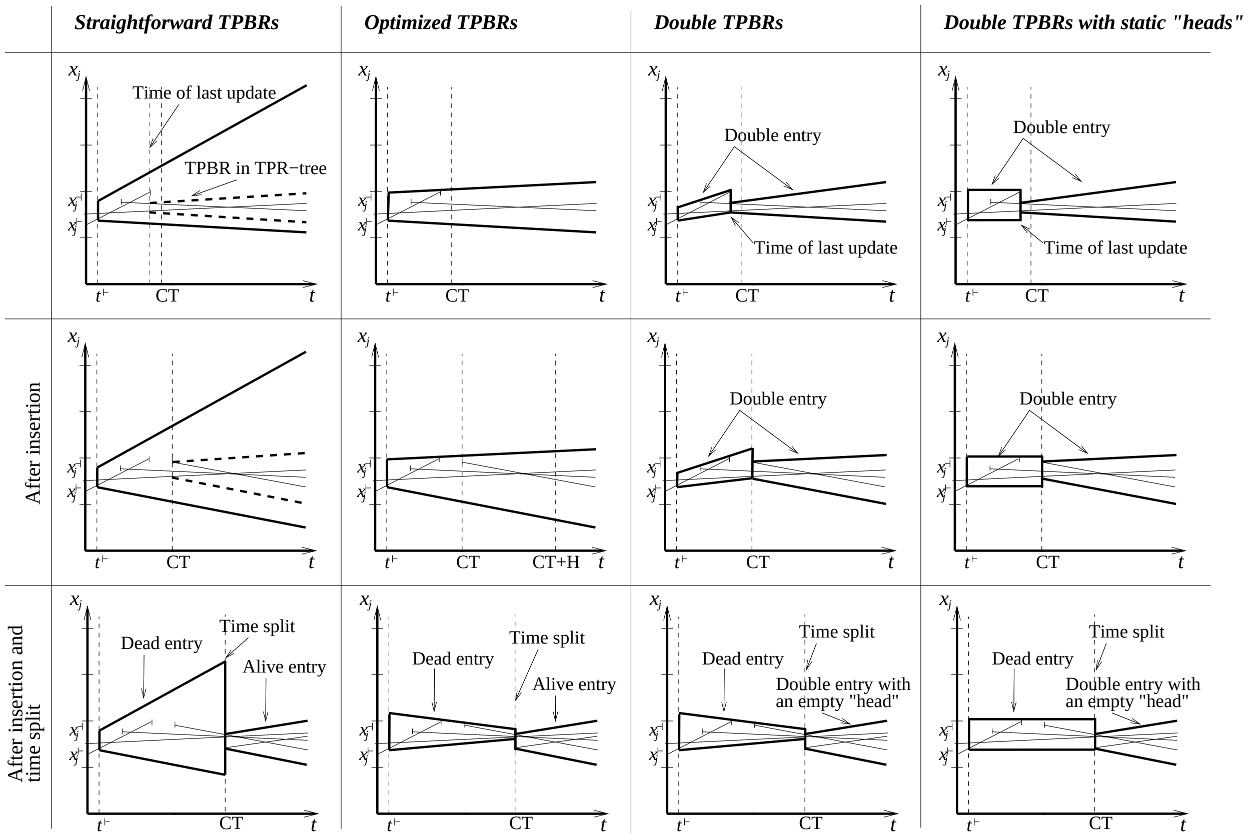

Figure 5: Evolution of Different Types of Time-Parameterized Bounding Rectangles

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 by time splits, as exemplified by the bottom-left 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 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 [26].

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 6 gives a sketch of an algorithm to compute such an optimal one-dimentional time parameterized bounding interval. Figure 7 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 [26].

```
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."
```

Figure 6: Algorithm *OptimalTPBI*

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 [26].

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 4 and the picture at the bottom of the second column in Figure 5). 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.

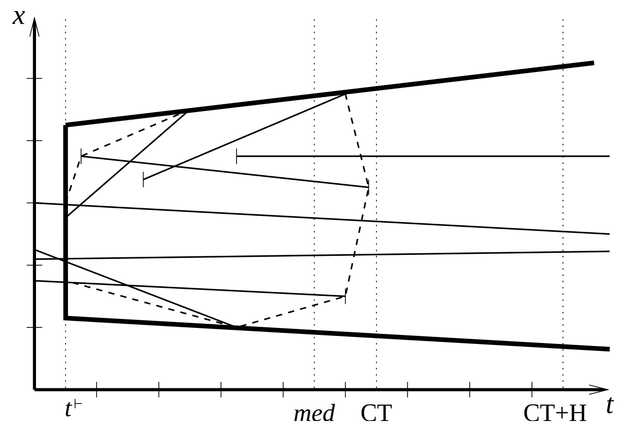

Figure 7: Optimal TPBR for a Set of One-Dimentional Moving Objects

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 fan-out 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 [26]. In addition, these TPBRS cannot be tightened.

5.2.4 Double TPBRs

To allow tightening, we introduce so-called *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 the 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—bounds with zero speeds. Both cases are illustrated in the last two columns of Figure 5. 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 (cf. the last row in Figure 5). 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 subsection, 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 fan-outs of nodes with optimized TPBRs and nodes with double TPBRs with static heads. The average fan-out 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 the other two types of 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 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 6 experimentally compares the three types of TPBRs.

5.3 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.

5.3.1 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 multi-version 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.

5.3.2 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.

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 8 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 9 and 12. 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 10.

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 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 4) 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*.

```
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   $e.t^{\dashv} \leftarrow$ 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
9         $Assuring \leftarrow AssureInvariants(RightNode)$        // Here $RightNode$ is a $Leaf$ or its live ancestor
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 $Node \leftarrow Parent(Node)$
```

Figure 8: Algorithm *Update*

Having found the alive leaf node were 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 8).

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 5.3.4. In the following, the algorithms *CorrectLeaves* and *CorrectParents* are covered in detail.

5.3.3 The *CorrectLeaves* Algorithm

Algorithm *CorrectLeaves*, provided in Figure 9, 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 were 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 life time starting before or at the time when e_o was inserted.

Figure 10 shows an example of the evolution of a part of an index storing one-dimensional data. In the

```
CorrectLeaves($Leaf, e_o, e_n$)
1   $e_c \leftarrow e_o$
2   $e_c.tpp.\bar{v} \leftarrow (e_n.tpp.\bar{x} - e_o.tpp.\bar{x})/(\text{CT} - e_o.t^{\vdash})$
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 ($e_c, PageIds$)
```

Figure 9: Algorithm *CorrectLeaves*

(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.

5.3.4 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 10 shows that this intuition is, in fact, misleading.

Specifically, Figure 10 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 11 shows the evolution of a small part of Figure 10. 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 10). The events of Figure 11 lead to the counter-intuitive configuration of TPBRs mentioned above.

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 neither entries B and 4, nor to any entries below them. The second part of the figure shows that

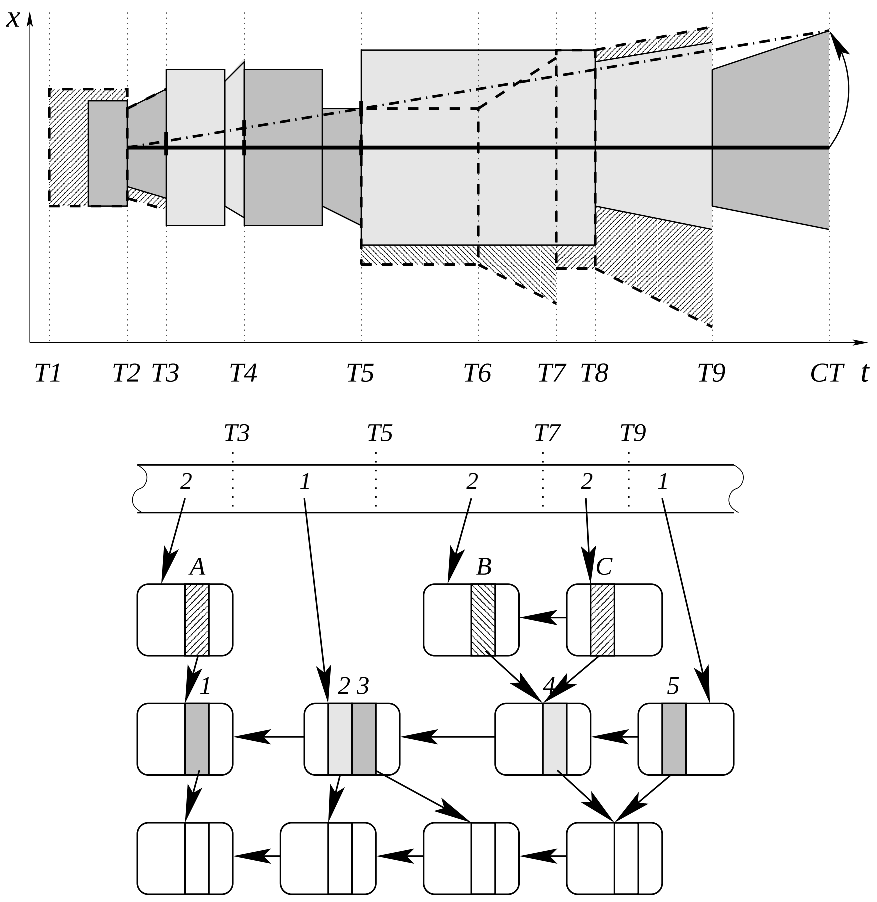

Figure 10: Correction of a Trajectory's Last-Recorded Segment

at time $T7$, due to activity in other parts of the tree, the node containing entry B was time split, resulting in the shape of entry B being 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 R$^{\text{PPF}}$-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 10, 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 12). This way, the algorithm checks the entries on all paths leading from the root(s) to all copies of the corrected trajectory.

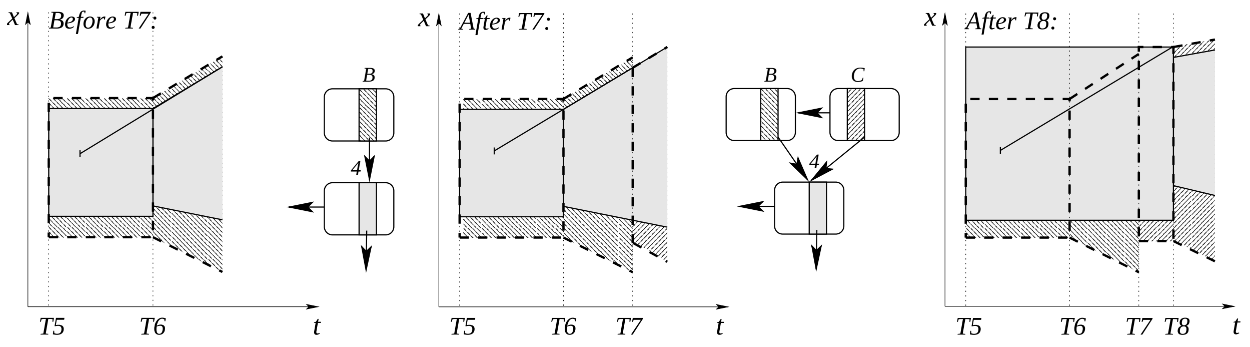

Figure 11: The Sequence of Events Leading to the Anomalous Grandparent B

5.3.5 The *CorrectParents* Algorithm

Figure 12 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*.

```
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 coordi-
       nates 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)
```

Figure 12: Algorithm *CorrectParents*

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 top-down operation is performed in the $\mathrm{R}^{\mathrm{PPF}}$-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 may 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 $\mathrm{R}^{\mathrm{PPF}}$-tree. This situation is illustrated in Figure 10. 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 12); 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 $\mathrm{R}^{\mathrm{PPF}}$-tree, in which case the traversal has to stop, or it can mean that the tree has shrunk in height and then grown again, creating a gap in this level of the $\mathrm{R}^{\mathrm{PPF}}$-tree. Figure 10 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 12).

Naturally, the above-described correction of the last-recorded fragments of a trajectory costs additional I/O, when compared to the normal update operation. The size of this 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. The performance experiments in the next section investigate the effect of ϵ on update performance.

6 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 the presentation of the main results of the experiments.

6.1 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 adapting the partial persistence technique to an adapted TPR-tree, to test the

R$^{\text{PPF}}$-tree, we use the same experimental settings as were used for the performance experiments with the TPR-tree [25]. Here, we review these settings—details can be found elsewhere [25].

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 being 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 routes connecting 20 destinations distributed in the space with dimensions 1000 km $\times$ 1000 km. Objects move with maximum speeds raging from 0 to 3 km/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 (cf. 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 min in most experiments. In addition to updates, 10,000 timeslice queries are issued during a workload. The spatial part of each query is a randomly placed square querying 0.25% of the total data space. Equal amounts of past and current/future queries are generated, with future queries having times between the current time CT and CT $+\ UI/2$.

In all experiments, the data page size is set to 8K, and an LRU buffer of 100 pages is used. Unless noted otherwise, d, the minimum number 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.

6.2 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. Double optimized TPBRs reduce fanout on average by 33%, but may 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 may 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 13 and 14 show the results of the experiments with varying UI, and Figures 15 and 16 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, the reason being that rectangles bound parts of the trajectories less accurately than trapezoids. For future queries, the performance of double TPBRs 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.

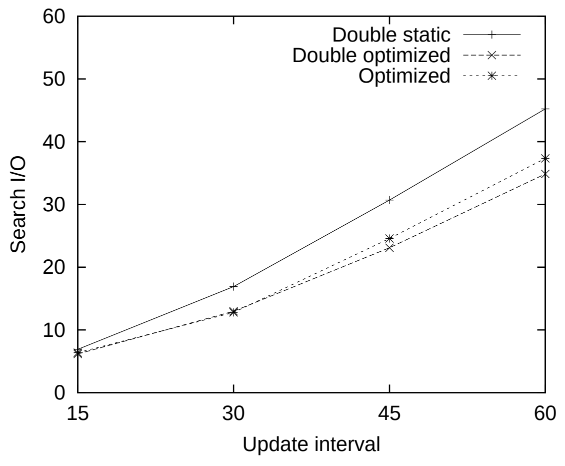

Figure 13: Past Query Performance for Different TPBRs and Varying UI

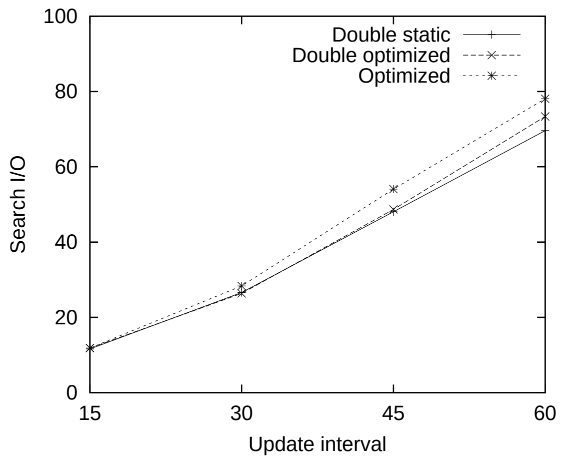

Figure 14: Future Query Performance for Different TPBRs and Varying UI

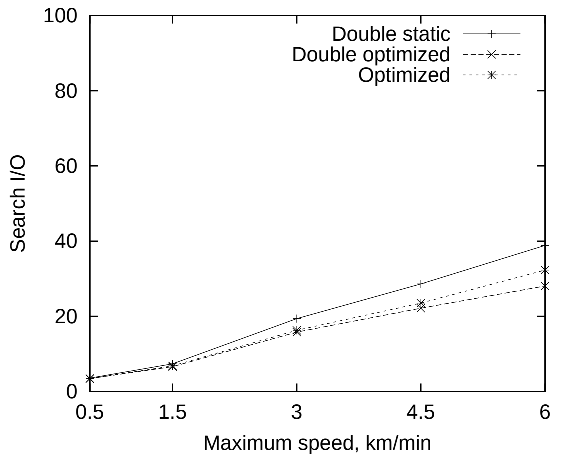

Figure 15: Past Query Performance for Different TPBRs and Varying Maximum Speed

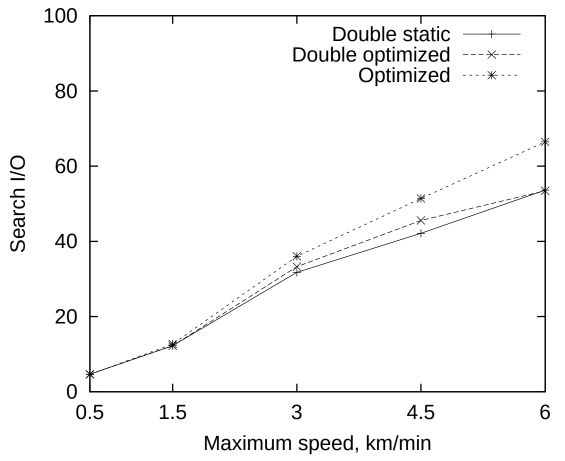

Figure 16: Future Query Performance for Different TPBRs and Varying Maximum Speed

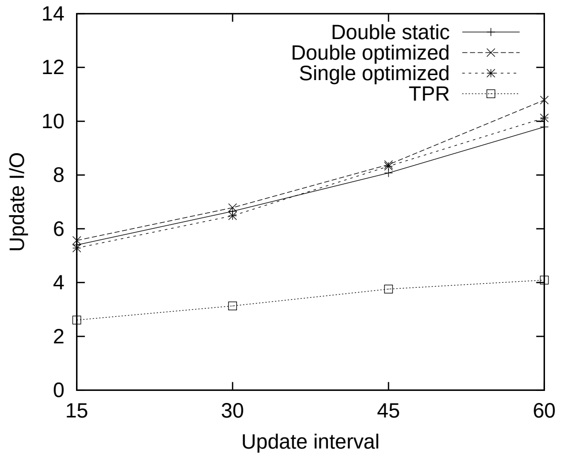

Figure 17: Update Performance for Different TPBRs and the TPR-tree, when Varying UI

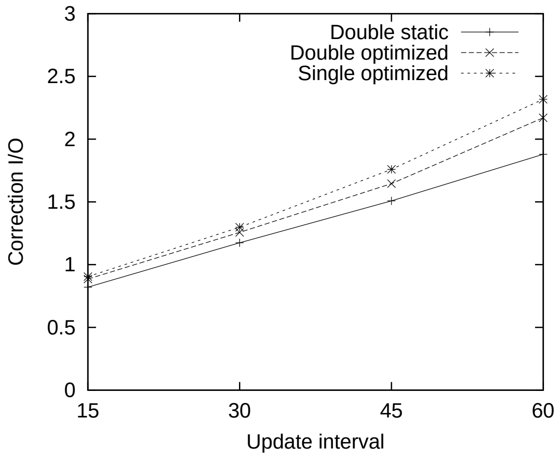

Figure 18: Cost of Correction for Different TPBRs and Varying UI

In the same set of experiments, the performance of update operations was measured. The results are reported in Figures 17 and 19, 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 18 and 20 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 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.

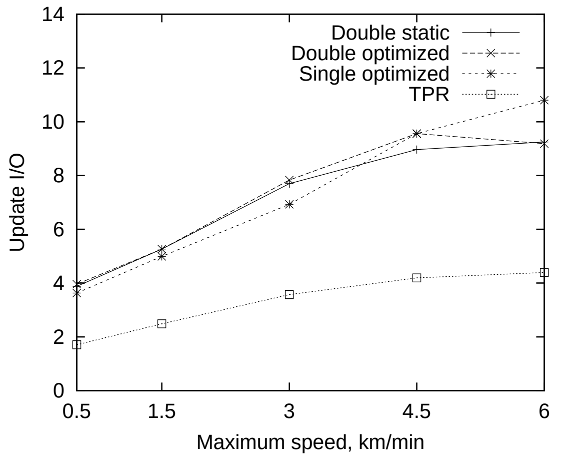

Figure 19: Update Performance for Different TPBRs and the TPR-tree, Varying Maximum Speed

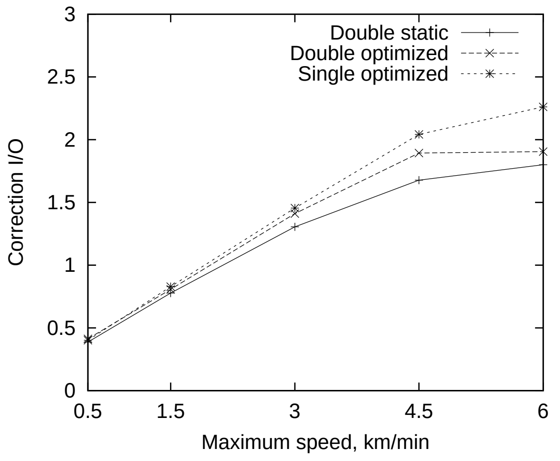

Figure 20: Cost of Correction for Different TPBRs and Varying Maximum Speed

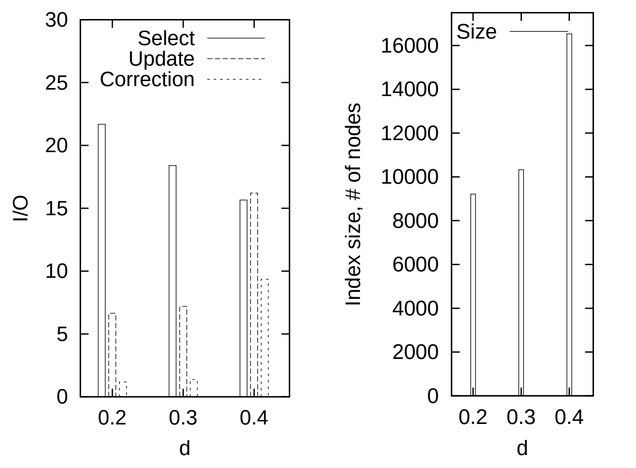

Figure 21: I/O Performance and Index Size of R$^{\mathrm{PPF}}$-Tree for Varying d

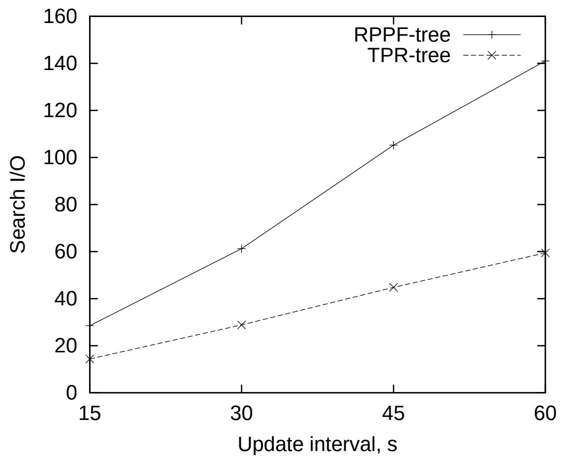

Figure 22: Query Performance of R$^{\mathrm{PPF}}$-Tree and TPR-tree for Varying UI

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$^{\mathrm{PPF}}$-tree.

6.3 Update Performance

As described in Section 5.3, the correction of the most recent trajectory segment 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 18 and 20, when compared to Figures 17 and 19, show that the correction of the last-recorded trajectory segment accounts for less than a fourth of the total update cost.

The update cost overhead is dependent on the average number of time splits that occur between two updates of an object. To vary the number of time splits, in a first set of experiments, we varied d. For each value of d (0.2, 0.3, and 0.4), we chose the largest possible ϵ that allows the invariants of partial persistence to be maintained [4]. As mentioned in Section 5.1, the frequency of time splits is inversely proportional to ϵ. Thus, it increases with an increasing d.

Figure 21 plots the update and query performance against d. As expected, when d decreases, the total number of time splits decreases (from 17,948 for $d = 0.4$ to 8,504 for $d = 0.2\%$). This results in the decrease by a factor of almost 2.5 in the average number of I/O operations per update. Observe that, as the figure shows, this decrease is mainly caused by a decrease in the cost of correction of the last-recorded trajectory segment.

While update performance is improved by decreasing d, timeslice query performance is naturally decreased. This is so because smaller values of d mean smaller average fanout of the index tree as "seen" at the time of query. Nevertheless, the decrease in query performance is smaller than the increase in update performance. Also, as Figure 21 shows, smaller values of d result in smaller index sizes.

6.4 The R$^{\mathrm{PPF}}$-Tree Versus the TPR-Tree

Naturally, recording history in the R$^{\mathrm{PPF}}$-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 and 19 show that performing update operations on the TPR-tree is more than twice as fast as doing the same updates on the $\mathrm{R}^{\mathrm{PPF}}$-tree.

In a separate set of experiments, we explored how much search performance is lost in the $\mathrm{R}^{\mathrm{PPF}}$-tree when compared to the TPR-tree. We ran a number of workloads of updates on both the $\mathrm{R}^{\mathrm{PPF}}$-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 22, 23, and 24 show the results of these experiments, where, in addition to the changing of UI and the maximum speed, we change the size of the query (expressed in percents of the total data space). In the experiments of Figures 22 and 23, the query size is set to 0.2%.

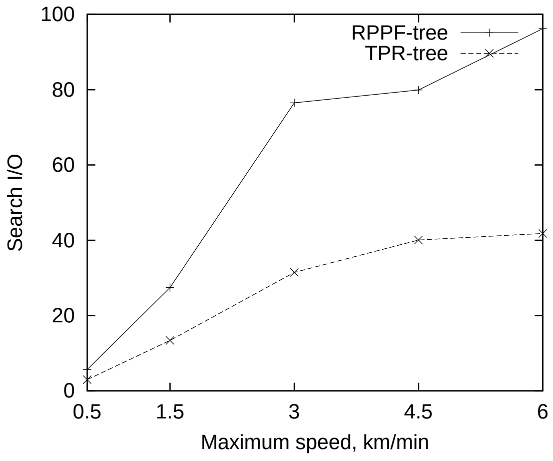

Figure 23: Query Performance of $\mathrm{R}^{\mathrm{PPF}}$-Tree and TPR-tree for Varying Maximum Speed

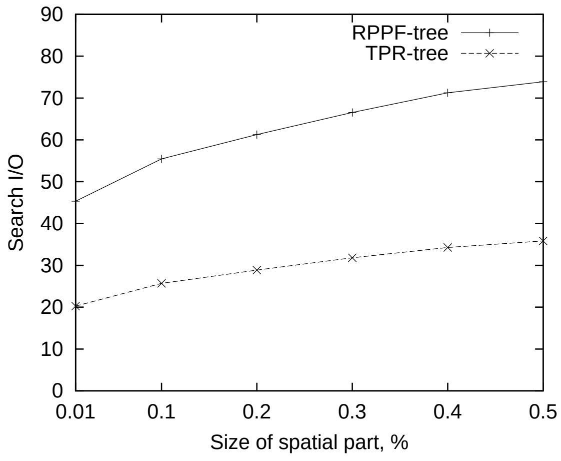

Figure 24: Query Performance of $\mathrm{R}^{\mathrm{PPF}}$-Tree and TPR-tree for Varying Query Size

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 $\mathrm{R}^{\mathrm{PPF}}$-tree are a little bit more than twice those of the TPR-tree. This is as expected because the fanout of the live part of the $\mathrm{R}^{\mathrm{PPF}}$-tree is lower than the fanout of the TPR-tree.

7 Summary and Research Directions

With the proliferation of wireless communications and geo-positioning as motivation, this paper defines what we believe is the first single index that is able to capture the past, present, and anticipated future positions of moving objects, where the latter positions are expressed as linear functions.

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, in the process developing 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. Empirical studies with an implementation of the indexing technique are reported that offer insight into its performance characteristics.

As an interesting future research direction, the partial persistence framework presented in this paper could be applied to other indexing techniques that use linear functions to capture continuous change. For example, although the recently proposed STRIPES [21] and B^x-tree [14] techniques are very different from the TPR-tree based techniques, it may be possible to apply techniques similar to the ones presented in this paper to these two, in order to extend them to accurately record the history of movement.

Next, while in the performance experiments of this paper the R$^{\text{PPF}}$-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 charactersitics of the environment, such as temperature, barometric pressure, humidity, or ambient light.

Acknowledgments

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.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. *Proc. of PODS*, pp. 175–186, 2000.

[2] P. K. Agarwal and S. Har-Peled Maintaining Approximate Extent Measures of Moving Points. *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 148–157, 2001.

[3] J. Basch, L. Guibas, and J. Hershberger. Data Structures for Mobile Data. *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 747–756, 1997.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree *VLDB Journal* 5(4): 264–275, 1996.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. of SIGMOD*, pp. 322–331, 1990.

[6] J. Van den Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. *Proc. of VLDB*, pp. 168–179, 1996.

[7] M. Cai and P. Z. Revesz. Parametric R-Tree: An Index Structure for Moving Objects. *Proc. of COMAD*, 2000.

[8] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets With SETI. *Online Proc. of Biennial Conf. on Innovative Data Systems Research*, 2003.

[9] H. D. Chon, D. Agrawal, and A. El Abbadi. Query Processing for Moving Objects with Space-Time Grid Storage Model. *Proc. of the Intl. Conf. on Mobile Data Management*, pp. 121–128, 2002.

[10] A. Čivilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004, 10 pages, to appear.

[11] A. Čivilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. DB Technical Report TR-5, Department of Computer Science, Aalborg University, February 2004, 23 pages.

[12] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient Indexing of Spatiotemporal Objects. *Proc. of EDBT*, pp. 251–268, 2002.

[13] C. S. Jensen (editor). *Special Issue of the IEEE Data Engineering Bulletin on Indexing of Moving Objects*, 25(2), June 2002.

[14] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. *Proc. of VLDB*, 2004, 12 pages, to appear.

[15] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. *Proc. of PODS*, pp. 261–272, 1999.

[16] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing Animated Objects Using Spatiotemporal Access Methods. *TKDE* 13(5): 758–777, 2001.

[17] C. P. Kolovson and M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. *Proc. of SIGMOD*, pp. 138–147, 1991.

[18] A. Kumar, Vassilis. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *TKDE* 10(1): 1–20, 1998.

[19] D. B. Lomet. Grow and Post Index Trees: Roles, Techniques and Future Potential. *Proc. of the Intl. Symposium on Advances in Spatial Databases*, pp. 183–206, 1991.

[20] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras. Indexing Mobile Objects on the Plane. *Proc. of the Intl. Workshop on Mobility in Databases and Distributed Systems*, pp. 693–697, 2002

[21] J. M. Patel, Y. Chen, V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. *Proc. of SIGMOD*, pp. 637–646, 2004.

[22] D. Pfoser, Y. Theodoridis, and C. S. Jensen. Novel Approaches in Query Processing for Moving Object Trajectories. *Proc. of VLDB*, pp. 395–406, 2000.

[23] K. Porkaew, I. Lasaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. *Proc. of SSTD*, pp. 59–78, 2001.

[24] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. *Proc. of the Intl. Workshop on Algorithm Engineering and Experiments*, pp. 178–193, 2002.

[25] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. *Proc. of SIGMOD*, pp. 331–342, 2000.

[26] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. *Proc. of ICDE*, pp. 463–472, 2002.

[27] Z. Song and N. Roussopoulos. Hashing Moving Objects. *Proc. of the Intl. Conf. on Mobile Data Management*, pp. 161–172, 2001.

[28] Z. Song and N. Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. *Proc. of the Intl. Conf. on Mobile Data Management*, pp. 340–344, 2003.

[29] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present and the Future in Spatio-Temporal Databases. In *Proc. of ICDE*, 2004, pp. 202–213.

[30] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *Proc. of VLDB* pp. 431–440, 2001.

[31] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *Proc. of VLDB* pp. 790–801, 2003.

[32] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3): 185–200, 1998.

[33] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases* 7(3): 257–387, 1999.