Indexing the Positions of Continuously Moving Objects

Simonas Šaltenis† Christian S. Jensen† Scott T. Leutenegger‡ Mario A. Lopez‡

† Department of Computer Science, Aalborg University, Denmark

‡ Department of Mathematics and Computer Science, University of Denver, Colorado, USA

Abstract

The coming years will witness dramatic advances in wireless communications as well as positioning technologies. As a result, tracking the changing positions of objects capable of continuous movement is becoming increasingly feasible and necessary. The present paper proposes a novel, R*-tree based indexing technique that supports the efficient querying of the current and projected future positions of such moving objects. The technique is capable of indexing objects moving in one-, two-, and three-dimensional space. Update algorithms enable the index to accommodate a dynamic data set, where objects may appear and disappear, and where changes occur in the anticipated positions of existing objects. A comprehensive performance study is reported.

1 Introduction

The rapid and continued advances in positioning systems, e.g., GPS, wireless communication technologies, and electronics in general promise to render it increasingly feasible to track and record the changing positions of objects capable of continuous movement.

In a recent interview with Danish newspaper Børsen, Michael Hawley from MIT's Media Lab described how he was online when he ran the Boston Marathon this year [19]. Prior to the race, he swallowed several capsules, which in conjunction with other hardware enabled the monitoring of his position, body temperature, and pulse during the race. This scenario demonstrates the potential for putting bodies, and, more generally, objects that move, online. Achieving this may enable a multitude of applications. It becomes possible to detect the signs of an impending medical emergency in a person early and warn the person or alert a medical service. It becomes possible to have equipment recognize its user; and the equipment may alert its owner in the case of unauthorized use or theft.

Industry leaders in the mobile phone market expect more than 500 million mobile phone users by year 2002 (compared to 300 million Internet users) and 1 billion by year 2004, and they expect mobile phones to evolve into wireless Internet terminals [14, 25]. Rendering such terminals location aware may substantially improve the quality of the services offered to them [12, 25]. In addition, the cost of providing location awareness is expected to be relatively low. These factors combine to promise the presence of substantial numbers of location aware, on-line objects capable of continuous movement.

Applications such as process monitoring do not depend on positioning technologies. In these, the position of a moving point object could for example be a pair of temperature and pressure values. Yet other applications include vehicle navigation, tracking, and monitoring, where the positions of air, sea, or land-based equipment such as airplanes, fishing boats and freighters, and cars and trucks are of interest. It is diverse applications such as these that warrant the study of the indexing of objects that move.

Continuous movement poses new challenges to database technology. In conventional databases, data is assumed to remain constant unless it is explicitly modified. Capturing continuous movement with this assumption would entail either performing very frequent updates or recording outdated, inaccurate data, neither of which are attractive alternatives.

A different tack must be adopted. The continuous movement should be captured directly, so that the mere advance of time does not necessitate explicit updates [27]. Put differently, rather than storing simple positions, functions of time that express the objects' positions should be stored. Then updates are necessary only when the parameters of the functions change. We use one linear function per object, with the parameters of a function being the position and velocity vector of the object at the time the function is reported to the database.

Two different, although related, indexing problems must be solved in order to support applications involving continuous movement. One problem is the indexing of the current and anticipated future positions of moving objects. The other problem is the indexing of the histories, or trajectories, of the positions of moving objects. We focus on the former problem. One approach to solving the latter problem (while simultaneously solving the first) is to render the solution to the first problem partially persistent [6, 15].

We propose an indexing technique, the time-parameterized R-tree (the TPR-tree, for short), which efficiently indexes the current and anticipated future positions of moving point objects (or "moving points," for short). The technique naturally extends the R*-tree [5].

Several distinctions may be made among the possible approaches to the indexing of the future linear trajectories of moving points. First, approaches may differ according to the space that they index. Assuming the objects move in d-dimensional space ($d = 1, 2, 3$), their future trajectories may be indexed as lines in ($d + 1$)-dimensional space [26]. As an alternative, one may map the trajectories to points in a higher-dimensional space which are then indexed [13]. Queries must subsequently also be transformed to counter the data transformation. Yet another alternative is to index data in its native, d-dimensional space, which is possible by parameterizing the index structure using velocity vectors and thus enabling the index to be "viewed" as of any future time. The TPR-tree adopts this latter alternative. This absence of transformations yields a quite intuitive indexing technique.

A second distinction is whether the index partitions the data (e.g., as do R-trees) or the embedding space (e.g., as do Quadtrees). When indexing the data in its native space, an index based on data partitioning seems to be more suitable. On the other hand, if trajectories are indexed as lines in ($d+1$)-dimensional space, a data partitioning access method that does not employ clipping may introduce substantial overlap.

Third, indices may differ in the degrees of data replication they entail. Replication may improve query performance, but may also adversely affect update performance. The TPR-tree does not employ replication.

Fourth, we may distinguish approaches according to whether or not they require periodic index rebuilding. Some approaches (e.g., [26]) employ individual indices that are only functional for a certain time period. In these approaches, a new index must be provided before its predecessor is no longer functional. Other approaches may employ an index that in principle remains functional indefinitely [13], but which may be optimized for some specific time horizon and perhaps deteriorates as time progresses. The TPR-tree belongs to this latter category.

In the TPR-tree, the bounding rectangles in the tree are functions of time, as are the moving points being indexed. Intuitively, the bounding rectangles are capable of continuously following the enclosed data points or other rectangles as these move. Like the R-trees, the new index is capable of indexing points in one-, two-, and three-dimensional space. In addition, the principles at play in the new index are extendible to non-point objects.

The next section presents the problem being addressed, by describing the data to be indexed, the queries to be supported, and problem parameters. In addition, related research is covered. Section 3 describes the tree structure and algorithms. It is assumed that the reader has some familiarity with the R*-tree. To ease the exposition, one-dimensional data is generally assumed, and the general n-dimensional case is only considered when the inclusion of additional dimensions introduces new issues. Section 4 reports on performance experiments, and Section 5 summarizes and offers research directions.

2 Problem Statement and Related Work

We describe the data being indexed, the queries being supported, the problem parameters, and related work in turn.

2.1 Problem Setting

An object's position at some time t is given by $\bar{x}(t) = (x_1(t), x_2(t), \ldots, x_d(t))$, where it is assumed that the times t are not before the current time. This position is modeled as a linear function of time, which is specified by two parameters. The first is a position for the object at some specified time t_{ref}, $\bar{x}(t_{ref})$, which we term the reference position. The second parameter is a velocity vector for the object, $\bar{v} = (v_1, v_2, \ldots, v_d)$. Thus, $\bar{x}(t) = \bar{x}(t_{ref}) + \bar{v}(t - t_{ref})$. An object's movement is observed at some time, t_{obs}. The first parameter, $\bar{x}(t_{ref})$, may be the object's position at this time, or it may be the position that the object would have at some other, chosen reference time, given the velocity vector $\bar{v}$ observed at t_{obs} and the position $\bar{x}(t_{obs})$ observed at t_{obs}.

Modeling the positions of moving objects as functions of time not only enables us to make tentative future predictions, but also solves the problem of the frequent updates that would otherwise be required to approximate continuous movement in a traditional setting. For example, objects may report their positions and velocity vectors when their actual positions deviate from what they have previously reported by some threshold. The choice of the update frequency then depends on the type of movement, the desired accuracy, and the technical limitations [28, 20, 17].

As will be illustrated in the following and explained in Section 3, the reference position and the velocity are used not only when recording the future trajectories of moving points, but also for representing the coordinates of the bounding rectangles in the index as functions of time.

As an example, consider Figure 1. The top left diagram shows the positions and velocity vectors of 7 point objects at time 0.

Assume we create an R-tree at time 0. The top right diagram shows one possible assignment of the objects to minimum bounding rectangles (MBRs) assuming a maximum of three objects per node. Previous work has shown that attempting to minimize the quantities known as overlap, dead space, and perimeter leads to an index with good query performance [11, 18], and so the chosen assignment appears to be well chosen. However, although it is good for queries at

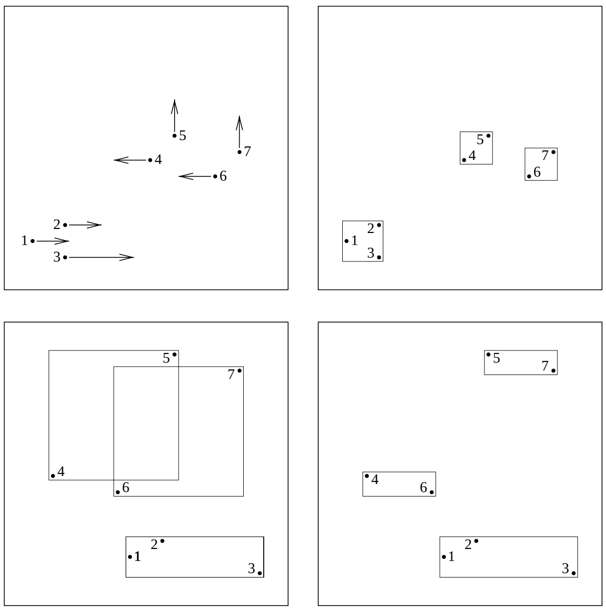

Figure 1: Moving Points and Resulting Leaf-Level MBRs

the present time, the movement of the objects may adversely affect this assignment.

The bottom left diagram shows the locations of the objects and the MBRs at time 3, assuming that MBRs grow to stay valid. The grown MBRs adversely affect query performance; and as time increases, the MBRs will continue to grow, leading to further deterioration. Even though the objects belonging to the same MBR (e.g., objects 4 and 5) were originally close, the different directions of their movement cause their positions to diverge rapidly and hence the MBRs to grow.

From the perspective of queries at time 3, it would have been better to assign objects to MBRs as illustrated by the bottom right diagram. Note that at time 0, this assignment will yield worse query performance than the original assignment. Thus, the assignment of objects to MBRs must take into consideration when most queries will arrive.

The MBRs in this example illustrate the kind of time-parameterized bounding rectangles supported by the TPR-tree. The algorithms presented in Section 3, which are responsible for the assignment of objects to bounding rectangles and thus control the structure and quality of the index, attempt to take observations such as those illustrated by this example into consideration.

2.2 Query Types

The queries supported by the index retrieve all points with positions within specified regions. We distinguish between three kinds, based on the regions they specify. In the sequel, a d-dimensional rectangle R is specified by its d projections $[a_1^{\vdash}, a_1^{\dashv}], \dots [a_d^{\vdash}, a_d^{\dashv}]$, $a_j^{\vdash} \leq a_j^{\dashv}$, into the d coordinate axes. Let R, R_1, and R_2 be three d-dimensional rectangles and t, $t^{\vdash} < t^{\dashv}$, three time values that are not less than the current time.

Type 1 timeslice query: $Q = (R, t)$ specifies a hyper-rectangle R located at time point t.

Type 2 window query: $Q = (R, t^{\vdash}, t^{\dashv})$ specifies a hyper-rectangle R that covers the interval $[t^{\vdash}, t^{\dashv}]$. Stated differently, this query retrieves points with trajectories in ($\bar{x}$, t)-space crossing the ($d + 1$)-dimensional hyper-rectangle ($[a_1^{\vdash}, a_1^{\dashv}]$, $[a_2^{\vdash}, a_2^{\dashv}]$, ... , $[a_d^{\vdash}, a_d^{\dashv}]$, $[t^{\vdash}, t^{\dashv}]$).

Type 3 moving query: $Q = (R_1, R_2, t^{\vdash}, t^{\dashv})$ specifies the ($d + 1$)-dimensional trapezoid obtained by connecting R_1 at time $t^{\vdash}$ to R_2 at time $t^{\dashv}$.

The second type of query generalizes the first, and is itself a special case of the third type. To illustrate the query types, consider the one-dimensional data set in Figure 2, which represents temperatures measured at different locations. Here, queries $Q0$ and $Q1$ are timeslice queries, $Q2$ is a window query, and $Q3$ is a moving query.

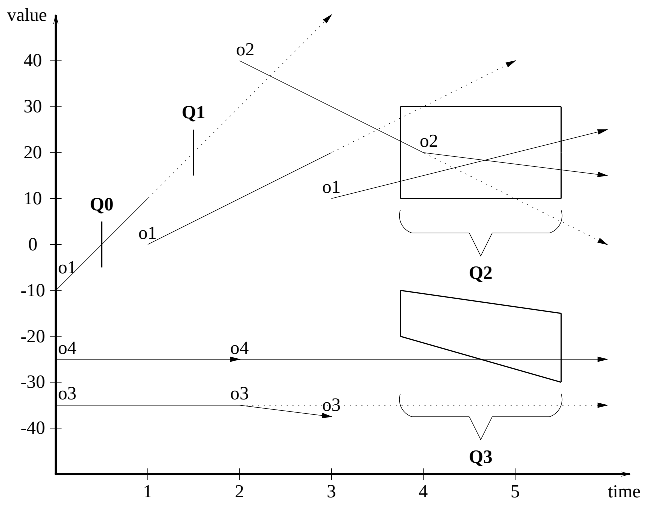

Figure 2: Query Examples for One-Dimensional Data

Let $iss(Q)$ denote the time when a query Q is issued. The two parameters, reference position and velocity vector, of an object as seen by a query Q depend on $iss(Q)$, because objects update their parameters as time goes. Consider object $o1$: its movement is described by one trajectory for queries with $iss(Q) < 1$, another trajectory for queries with $1 \leq iss(Q) < 3$, and a third trajectory for queries with $3 \leq iss(Q)$. For example, the answer to query $Q1$ is $o1$, if $iss(Q1) < 1$, and no object qualifies for this query if $iss(Q1) \geq 1$.

This example illustrates that queries far in the future are likely to be of little value, because the positions as predicted at query time become less and less accurate as queries move into the future, and because updates not known at query time may occur. Therefore, real-world applications may be expected to issue queries that are concentrated in some limited time window extending from the current time.

2.3 Problem Parameters

The values of three problem parameters affect the indexing problem and the qualities of a TPR-tree. Figure 3 illustrates these parameters, which will be used throughout the paper.

- *Querying window* (*W*): how far queries can "look" into the future. Thus, $iss(Q) \leq t \leq iss(Q) + W$, for Type 1 queries, and $iss(Q) \leq t^{\vdash} \leq t^{\dashv} \leq iss(Q) + W$ for queries of Types 2 and 3.
- *Index usage time* (*U*): the time interval during which an index will be used for querying. Thus, $t_l \leq iss(Q) \leq t_l + U$, where t_l is the time when index is created/loaded.
- *Time horizon* (*H*): the length of the time interval from which the times t, $t^{\vdash}$, and $t^{\dashv}$ specified in queries are drawn. The time horizon for an index is the index usage time plus the querying window.

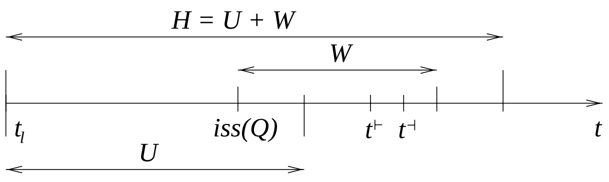

Figure 3: Time Horizon *H*, Index Usage Time *U*, and Querying Window *W*

Thus, a newly created index must support queries that reach H time units into the future. While the utility of parameter *U* (and *H*) is more clearcut for static data sets and bulkloading, we shall see in Section 3.4 that this parameter is also useful in a dynamic setting where index updates are allowed. Specifically, although a TPR-tree is functional at all times after its creation, using different values for parameter *U* during insertions affects the search properties of the tree.

2.4 Previous Work

Related work on the indexing of the current and future positions of moving objects has concentrated mostly on points moving in one-dimensional space.

Tayeb et al. [26] use PMR-Quadtrees [22] for indexing the future linear trajectories of one-dimensional moving point objects as line segments in (x, t)-space. The segments span the time interval that starts at the current time and extends *H* time units into the future. A tree expires after *U* time units, and a new tree must be made available for querying. This approach introduces substantial data replication in the index—a line segment is usually stored in several nodes.

Kollios et al. [13] employ the dual data transformation where a line $x = x(t_{ref}) + v(t - t_{ref})$ is transformed to the point $(x(t_{ref}), v)$, enabling the use of regular spatial indices. It is argued that indices based on Kd-trees are well suited for this problem because these best accommodate the shapes of the (transformed) queries on the data. Kollios et al. suggest, but do not investigate in any detail, how this approach may be extended to two and higher dimensions. Kollios et al. also propose two other methods that achieve better query performance at the cost of data replication. These methods do not seem to apply to more than one dimension.

Next, Kollios et al. provide theoretical lower bounds for this indexing problem, assuming a static data set and $H = \infty$. Allowing the index to use linear space, the types of queries discussed in Section 2 can be answered in $O(n^{(2d-1)/2d} + k)$ time. Here d is the number of dimensions of the space where the objects move, n is the number of data blocks, and k is the size in blocks of a query answer. To achieve this bound, an external memory version of partition trees may be used [1]. It is argued that, although having good asymptotic performance bounds, partition trees are not practical due to the large constant factors involved.

Basch et al. [4] propose so-called kinetic main-memory data structures for mobile objects. The idea is to schedule future events that update a data structure so that necessary invariants hold. Agarwal et al. [2] apply these ideas to external range trees [3]. Their approach may possibly be applicable to R-trees or time-parameterized R-trees where events would fix MBRs, although it is unclear how to contend with future queries that arrive in non-chronological order. Agarwal et al. address non-chronological queries using partial persistence techniques and also show how to combine kinetic range trees with partition trees to achieve a trade-off between the number of kinetic events and query performance.

The problem of indexing moving points is related to the problem of indexing now-relative temporal data. The GR-tree [7] is an R-tree based index for now-relative bitemporal data. Combined valid and transaction time intervals with end-times related to the continuously progressing current time result in regions that grow, albeit in a restricted way. The idea in this index is to accommodate growing data regions by introducing bounding regions that also grow. Specifically, bounding regions are time-parameterized, and their extents are computed each time a query is asked.

The R$^{\mathrm{ST}}$-tree [24] is the spatiotemporal index that indexes the histories of the positions of objects. Positions are assumed to remain constant in-between explicit index updates, and their histories are captured by associating valid and transaction time intervals, which may be now-relative, with them. The continuity thus stems from the temporal aspects rather than the spatial, and the techniques employed in this index are more akin to those employed in the GR-tree than those employed here.

Finally, Pfoser et al. [21] consider the separate, but related problem of indexing the past trajectories of moving points, which are represented as polylines (connected line segments).

3 Structure and Algorithms

This section presents the structure and algorithms of the TPR-tree. The notion of a time-parameterized bounding rectangle is defined. It is shown how the tree is queried, and dynamic update algorithms are presented that tailor the tree to a specific time horizon *H*. In the following, we use the term bounding interval for a one-dimensional bounding rectangle and the term bounding rectangle for any *d*-dimensional hyper-rectangle.

3.1 Index Structure

The TPR-tree is a balanced, multi-way tree with the structure of an R-tree. Entries in leaf nodes are pairs of the position of a moving point and a pointer to the moving point, and entries in internal nodes are pairs of a pointer to a subtree and a rectangle that bounds the positions of all moving points or other bounding rectangles in that subtree.

As suggested in Section 2, the position of a moving point is represented by a reference position and a corresponding velocity vector—(x, v) in the one-dimensional case, where $x = x(t_{ref})$. We let t_{ref} be equal to the index creation time, t_l. Other possibilities include setting t_{ref} to some constant value, e.g., 0, or using different t_{ref} values in different nodes.

To bound a group of *d*-dimensional moving points, *d*-dimensional bounding rectangles are used that are also time-parameterized, i.e., their coordinates are functions of time. A time-parameterized bounding rectangle bounds all enclosed points or rectangles at all times not earlier than the current time.

A tradeoff exists between how tightly a bounding rectangle bounds the enclosed moving points or rectangles across time and the storage needed to capture the bounding rectangle. It would be ideal to employ time-parameterized bounding rectangles that are *always minimum*, but the storage cost appears to be excessive. In the general case, doing so deteriorates to enumerating all the enclosed moving points or rectangles. This is exemplified by Figure 4, where a node consists of two one-dimensional points A and B moving towards each other. Each of these points plays the role of lower (resp. upper) bound of the minimum bounding interval at some time. Examples with this property may be constructed for any number of points.

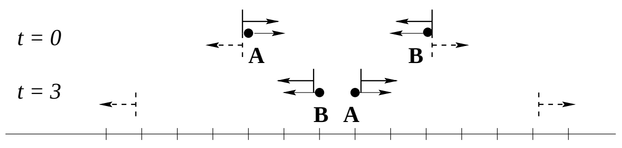

Figure 4: Conservative (Dashed) Versus Always Minimum (Solid) Bounding Intervals

Instead of using true, always minimum bounding rectangles, the TPR-tree employs what we term *conservative* bounding rectangles, which are minimum at some time point, but possibly (and most likely!) not at later times. In the one-dimensional case, the lower bound of a conservative interval is set to move with the minimum speed of the enclosed points, while the upper bound is set to move with the maximum speed of the enclosed points (speeds are negative or positive, depending on the direction). This ensures that conservative bounding intervals are indeed bounding for all times considered.

Figure 4 illustrates conservative bounding intervals. The left hand side of the conservative interval in the figure starts at the position of object A at time 0 and moves left at the speed of object B, and the right hand side of the interval starts at object B at time 0 and moves right at the speed of object A. Conservative bounding intervals never shrink. At best, when all of the enclosed points have the same velocity vector, a conservative bounding interval has constant size, although it may move.

Following the representation of moving points, we let $t_{ref} = t_l$ and capture a one-dimensional time-parameterized bounding interval $[x^{\vdash}(t), x^{\dashv}(t)] = [x^{\vdash}(t_l) + v^{\vdash}(t - t_l), x^{\dashv}(t_l) + v^{\dashv}(t - t_l)]$ as $(x^{\vdash}, x^{\dashv}, v^{\vdash}, v^{\dashv})$, where

$$\begin{aligned} x^{\vdash} &= x^{\vdash}(t_l) = \min_i\{o_i.x^{\vdash}(t_l)\} \\ x^{\dashv} &= x^{\dashv}(t_l) = \max_i\{(o_i.x^{\dashv}(t_l)\} \\ v^{\vdash} &= \min_i\{o_i.v^{\vdash}\} \\ v^{\dashv} &= \max_i\{o_i.v^{\dashv}\} \end{aligned}$$

Here, the o_i range over the bounding intervals to be enclosed. If instead the bounding interval being defined is to bound moving points, the o_i range over these points, $o_i.x^{\vdash}(t_l)$ and $o_i.x^{\dashv}(t_l)$ are replaced by $o_i.x(t_l)$, and $o_i.v^{\vdash}$ and $o_i.v^{\dashv}$ are replaced by $o_i.v$.

The rectangles defined above are termed load-time bounding rectangles and are bounding for all times not before t_l. Because the rectangles never shrink, but may actually grow too much, it is desirable to be able to adjust them occasionally. Specifically, as the index is only queried for times greater or equal to the current time, it is possible and probably attractive to adjust the bounding rectangles every time any of the moving points or rectangles that they bound are updated. The following formulas specify the adjustments to the bounding rectangles that may be made during updates.

$$\begin{aligned} x^{\vdash} &= \min_i\{o_i.x^{\vdash}(t_{upd})\} - v^{\vdash}(t_{upd} - t_l) \\ x^{\dashv} &= \max_i\{o_i.x^{\dashv}(t_{upd})\} - v^{\dashv}(t_{upd} - t_l) \end{aligned}$$

Here, t_{upd} is the time of the update, and the formulas may be restricted to apply to the bounding of points rather than intervals, as before. Each formula involves five terms, which may differ by orders of magnitude. Special care must be taken to manage the rounding errors that may occur in the finite-precision floating-point arithmetic (e.g., IEEE standard 754) used for implementing the formulas [8].

We call these rectangles update-time bounding rectangles. The two types of bounding rectangles are shown in Figure 5.

The bold top and bottom lines capture the load-time, time-parameterized bounding interval for the four moving objects represented by the four lines. At time t_{upd}, a more narrow and thus better update-time bounding interval is introduced that is bounding from t_{upd} and onwards.

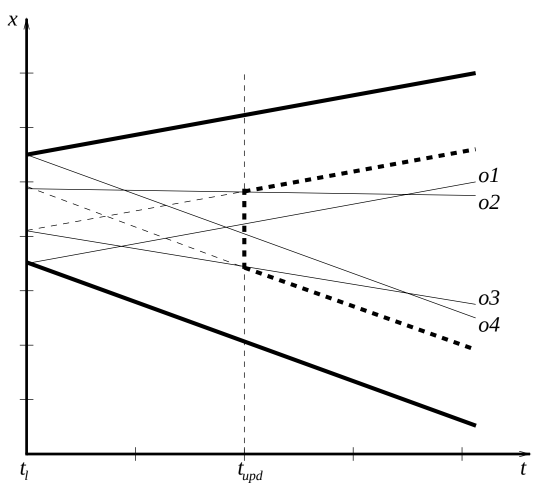

Figure 5: Load-Time (Bold) and Update-Time (Dashed) Bounding Intervals for Four Moving Points

It is worth noticing that the sole use of load-time bounding rectangles corresponds to simply bounding the $2d$-dimensional points that result from the dual transformation of the linear trajectories, as proposed by Kollios et al. [13]. Update-time bounding rectangles go beyond this approach.

3.2 Querying

With the definition of bounding rectangles in place, we show how the three types of queries presented in Section 2 are answered using the TPR-tree.

Answering a timeslice query proceeds as for the regular R-tree, the only difference being that all bounding rectangles are computed for the time t^q specified in the query before intersection is checked. Thus, a bounding interval specified by $(x^{\vdash}, x^{\dashv}, v^{\vdash}, v^{\dashv})$ satisfies a query $(([a^{\vdash}, a^{\dashv}]),\ t^q)$ if and only if $a^{\vdash} \leq x^{\dashv} + v^{\dashv}(t^q - t_l) \wedge a^{\dashv} \geq x^{\vdash} + v^{\vdash}(t^q - t_l)$.

To answer window queries and moving queries, we need to be able to check if, in $(\bar{x}, t)$-space, the trapezoid of a query (cf. Figure 6) intersects with the trapezoid formed by the part of the trajectory of a bounding rectangle that is between the start and end times of the query. With one spatial dimension, this is relatively simple. For more dimensions, generic polyhedron-polyhedron intersection tests may be used [9], but due to the restricted nature of this problem, a simpler and more efficient algorithm may be devised.

Specifically, we provide an algorithm for checking if a d-dimensional time-parameterized bounding rectangle R given by parameters $(x_1^{\vdash}, x_1^{\dashv}, x_2^{\vdash}, x_2^{\dashv}, \ldots, x_d^{\vdash}, x_d^{\dashv}, v_1^{\vdash}, v_1^{\dashv}, v_2^{\vdash}, v_2^{\dashv}, \ldots, v_d^{\vdash}, v_d^{\dashv})$ intersects a moving query $Q = (([a_1^{\vdash}, a_1^{\dashv}], [a_2^{\vdash}, a_2^{\dashv}], \ldots, [a_d^{\vdash}, a_d^{\dashv}], [w_1^{\vdash}, w_1^{\dashv}], [w_2^{\vdash}, w_2^{\dashv}], \ldots, [w_d^{\vdash}, w_d^{\dashv}]),$

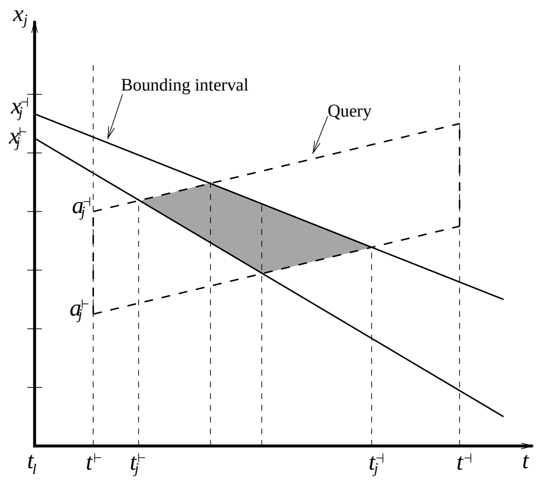

Figure 6: Intersection of a Bounding Interval and a Query

$t^{\vdash}, t^{\dashv})$. This formulation of a moving query as a time-parameterized rectangle with starting and ending times is more convenient than the definition given in Section 2.2. The velocities w are obtained by subtracting R_2 from R_1 in the earlier definition and then normalizing them with the length of interval $[t^{\vdash}, t^{\dashv}]$.

The algorithm is based on the observation that for two moving rectangles to intersect, there has to be a time point when their extents intersect in each dimension. Thus, for each dimension j ($j = 1, 2, \ldots, d$), the algorithm computes the time interval $I_j = [t_j^{\vdash}, t_j^{\dashv}] \subset [t^{\vdash}, t^{\dashv}]$ when the extents of the rectangles intersect in that dimension. If $I = \bigcap_{j=1}^{d} I_j = \emptyset$, the moving rectangles do not intersect and an empty result is returned; otherwise, the algorithm provides the time interval I when the rectangles intersect. The intervals for each dimension are computed according to the following formulas.

$$I_j = \begin{cases} \emptyset & \textbf{if } a_j^{\vdash} > x_j^{\dashv}(t^{\vdash}) \wedge a_j^{\vdash}(t^{\dashv}) > x_j^{\dashv}(t^{\dashv}) \vee \\ & \quad a_j^{\dashv} < x_j^{\vdash}(t^{\vdash}) \wedge a_j^{\dashv}(t^{\dashv}) < x_j^{\vdash}(t^{\dashv}) \\ [t_j^{\vdash}, t_j^{\dashv}] & \textbf{otherwise} \end{cases}$$

The first disjunct in the condition expresses that Q is above R and the second means that Q is below R. Formulas for $t_j^{\vdash}$ and $t_j^{\dashv}$ follow.

$$t_j^{\vdash} \quad = \quad \begin{cases} t^{\vdash} + \frac{x_j^{\dashv}(t^{\vdash}) - a_j^{\vdash}}{w_j^{\vdash} - v_j^{\dashv}} & \textbf{if } a_j^{\vdash} > x_j^{\dashv}(t^{\vdash}) \\ t^{\vdash} + \frac{x_j^{\vdash}(t^{\vdash}) - a_j^{\dashv}}{w_j^{\dashv} - v_j^{\vdash}} & \textbf{if } a_j^{\dashv} < x_j^{\vdash}(t^{\vdash}) \\ t^{\vdash} & \textbf{otherwise} \end{cases}$$

Here, the first condition states that Q is above R at $t^{\vdash}$, and the second states that Q is below R at $t^{\vdash}$.

$$t_j^{\dashv} = \begin{cases} t^{\vdash} + \frac{x_j^{\dashv}(t^{\vdash}) - a_j^{\vdash}}{w_j^{\vdash} - v_j^{\dashv}} & \textbf{if } a_j^{\vdash}(t^{\dashv}) > x_j^{\dashv}(t^{\dashv}) \\ t^{\vdash} + \frac{x_j^{\vdash}(t^{\vdash}) - a_j^{\dashv}}{w_j^{\dashv} - v_j^{\vdash}} & \textbf{if } a_j^{\dashv}(t^{\dashv}) < x_j^{\vdash}(t^{\dashv}) \\ t^{\dashv} & \textbf{otherwise} \end{cases}$$

In this formula, the first condition states that Q is above R at $t^{\dashv}$, and the second states that Q is below R at $t^{\dashv}$.

To see how $t_j^{\vdash}$ and $t_j^{\dashv}$ are computed, consider the case where Q is below R at $t^{\dashv}$. Then Q must not be below R at $t^{\vdash}$, as otherwise Q is always below R and there is no intersection (the case of no intersection is already accounted for). This means that the line $a_j^{\dashv} + w_j^{\dashv}(t - t^{\vdash})$ intersects the line $x_j^{\vdash}(t^{\vdash}) + v_j^{\vdash}(t - t^{\vdash})$ within the time interval $[t^{\vdash}, t^{\dashv}]$. Solving for t gives the desired intersection time ($t_j^{\dashv}$).

Figure 6 exemplifies a moving query, a bounding rectangle, and their intersection time interval in one dimension.

3.3 Heuristics for Tree Organization

As a precursor to designing the insertion algorithms for the TPR-tree, we discuss how to group moving objects into nodes so that the tree most efficiently supports timeslice queries when assuming a time horizon *H*. The objective is to identify principles, or heuristics, that apply to both dynamic insertions and bulkloading, and to any number of dimensions. The goal is to obtain a versatile index.

It is clear that when *H* is close to zero, the tree may simply use existing R-tree insertion and bulkloading algorithms. The movement of the point objects and the growth of the bounding rectangles become irrelevant—only their initial positions and extents matter. In contrast, when *H* is large, grouping the moving points according to their velocity vectors is of essence. It is desirable that the bounding rectangles are as small as possible at all times in $[t_l, t_l + H]$, the interval during which the result of the operation (insertion or bulkloading) may be visible to queries (t_l is thus the time of an insertion or the index creation time). An important aspect in achieving this is to keep the growth rates of the bounding rectangles, and thus the values of their "velocity extents," low. (In one-dimensional space, the velocity extent of a bounding interval is equal to $v^{\dashv} - v^{\vdash}$.)

This leads to the following general approach. The insertion and bulkloading algorithms of the R*-tree, which we consider extending to moving points, aim to minimize objective functions such as the areas of the bounding rectangles, their margins (perimeters), and the overlap among the bounding rectangles. In our context, these functions are time dependent, and we should consider their evolution in $[t_l, t_l + H]$. Specifically, given an objective function $A(t)$, the following integral should be minimized.

$$\int_{t_l}^{t_l+H} A(t)dt \qquad (1)$$

If $A(t)$ is area, the integral computes the area (volume) of the trapezoid that represents part of the trajectory of a bounding rectangle in $(\bar{x}, t)$-space (see Figure 6).

We use the integral in Formula 1 in the dynamic update algorithms, described next, and in the bulkloading algorithms, described elsewhere [23].

3.4 Insertion and Deletion

The insertion algorithm of the R*-tree employs functions that compute the area of a bounding rectangle, the intersection of two bounding rectangles, the margin of a bounding rectangle (when splitting a node), and the distance between the centers of two bounding rectangles (used when doing forced reinsertions) [5]. The TPR-tree's insertion algorithm is the same as that of the R*-tree, with one exception: instead of the functions mentioned here, integrals as in Formula 1 of those functions are used.

Computing the integrals of the area, margin, and distance are relatively straightforward [23]. The algorithm that computes the integral of the intersection of two time-parameterized rectangles is an extension of the algorithm for checking if such rectangles overlap (see Section 3.2). At each time point when the rectangles intersect, the intersection region is a rectangle and, in each dimensions, the upper (lower) bound of this rectangle is defined by the upper (lower) bound of one of the two intersecting rectangles.

The algorithm thus divides the time interval returned by the overlap-checking algorithm into consecutive time intervals so that, during each of these, the intersection is defined by a time-parameterized rectangle. The intersection area integral is then computed as a sum of area integrals. Figure 6 illustrates the subdivision of the intersection time interval into three smaller intervals for the one-dimensional case. The algorithm is given elsewhere [23].

In Section 2.3, parameter $H = U + W$ was introduced. This parameter is most intuitive in a static setting, and for static data. In a dynamic setting, *W* remains a component of *H*, which is the length of the time period where integrals are computed in the insertion algorithm. How large the other component of *H* should be depends on the update frequency. If this is high, the effect of an insertion on the tree will not persist long and, thus, *H* should not exceed *W* by much. The experimental studies in Section 4 aim at determining what is a good range of values for *H* in terms of the update frequency.

The introduction of the integrals is the most important step in rendering the R*-tree insertion algorithm suitable for the TPR-tree, but one more aspect of the R*-tree algorithm must be revisited. The R*-tree split algorithm selects one distribution of entries between two nodes from a set of candidate distributions, which are generated based on sortings of point positions along each of the coordinate axes. In the TPR-tree split algorithm, moving point (or rectangle) positions at different time points are used when sorting. With load-time bounding rectangles, positions at t_l are used, and with update-time bounding rectangles, positions at the current time are used.

Finally, in addition to sortings along the spatial dimensions, the split algorithm is extended to consider also sortings along the velocity dimensions, i.e., sortings obtained by sorting on the coordinates of the velocity vectors. The rationale is that distributing the moving points based on the velocity dimensions may result in bounding rectangles with smaller "velocity extents" and which consequently grow more slowly.

Deletions in the TPR-tree are performed as in the R*-tree. If a node gets underfull, it is eliminated and its entries are reinserted.

4 Performance Experiments

In this section we report on performance experiments with the TPR-tree. The generation of two- and three-dimensional moving point data and the settings for the experiments are described first, followed by the presentation of the results of the experiments.

4.1 Experimental Setup and Workload Generation

The implementation of the TPR-tree used in the experiments is based on the Generalized Search Tree Package, GiST [10]. The page size (and tree node size) is set to 4k bytes, which results in 204 and 146 entries per leaf-node for two- and three-dimensional data, respectively. A page buffer of 200k bytes, i.e., 50 pages, is used [16], where the root of a tree is pinned and the least-recently-used page replacement policy is employed. The nodes that are modified during an index operation are marked as "dirty" in the buffer and are written to disk at the end of the operation or when they otherwise have to be removed from the buffer.

The performance studies are based on workloads that intermix queries and update operations on the index, thus simulating index usage across a period of time. In addition, each workload initially bulkloads the index. An efficient bulkloading algorithm developed for the TPR-tree is used [23]. This algorithm is based on the heuristic of minimizing area integrals and has *H* as a parameter. We proceed to describe how the updates, queries, and initial bulkloading data are generated.

Because moving objects with positions and velocities that are uniformly distributed seems to be rather unrealistic, we attempt to generate more realistic (and skewed) two-dimensional data by simulating a scenario where the objects, e.g., cars, move in a network of routes, e.g., roads, connecting a number of destinations, e.g., cities. In addition to simulating cars moving between cities, the scenario is also motivated by the fact that usually, even if there is no underlying infrastructure, moving objects tend to have destinations.

With the exception of one experiment, the simulated objects in the scenario move in a region of space with dimensions 1000×1000 kilometers. A number *ND* of destinations are distributed uniformly in this space and serve as the vertices in a fully connected graph of routes. In most of the experiments, *ND* $= 20$. This corresponds to 380 one-way routes. The number of points is $N = 100,000$ for all but one experiment. No objects disappear, and no new objects appear for the duration of a simulation.

For the generation of the initial data set that is bulkloaded, objects are placed at random positions on routes. The objects are assigned with equal probability to one of three groups of points with maximum speeds of 0.75, 1.5, and 3 km$/$min (45, 90, and 180 km$/$h). During the first sixth of a route, objects accelerate from zero speed to their maximum speeds; during the middle two thirds, they travel at their maximum speeds; and during the last one sixth of a route, they decelerate. When an object reaches its destination, a new destination is assigned to it at random.

The workload generation algorithm distributes the updates of an object's movement so that updates are performed during the acceleration and deceleration stretches of a route. The number of updates is chosen so that the total average time interval between two updates is approximately equal to a given parameter *UI*, which is fixed at 60 in most of the experiments.

In addition to using data from the above-described simulation, some experiments also use workloads with two- and three-dimensional uniform data. In these workloads, the initial positions of objects are uniformly distributed in space. The directions of the velocity vectors are assigned randomly, both initially and on each update. The speeds (lengths of velocity vectors) are uniformly distributed between 0 and 3 km$/$min. The time interval between successive updates is uniformly distributed between 0 and 2*UI*.

To generate workloads, the above-described scenarios are run for 600 time units (minutes). For *UI* $= 60$, this results in approximately one million update operations.

In addition to updates, workloads include queries. Each time unit, four queries are generated (2400 in total). Timeslice, window, and moving queries are generated with probabilities 0.6, 0.2, and 0.2. The temporal parts of queries are generated randomly in an interval of length *W* and starting at the current time. The spatial part of each query is a square occupying a fraction *QS* of the space ($QS = 0.25\%$ in most of the experiments). The spatial parts of timeslice and window queries have random locations. For moving queries, the center of a query follows the trajectory of one of the points currently in the index.

The workload generation parameters that are varied in the experiments are given in Table 1. Standard values, used if a parameter is not varied in an experiment, are given in boldface.

4.2 Investigating the Insertion Algorithm

As mentioned in Section 3.4, the TPR-tree insertion algorithm depends on the parameter *H*, which is equal to *W* plus some duration that is dependent on the frequency of updates. How the frequency of updates affects the choice of a value

Parameter	Description	Values Used
ND	Number of destinations [cardinal number]	0, 2, 10, **20**, 40, 160
N	Number of points [cardinal number]	**100,000**, 300,000, 500,000, 700,000, 900,000
UI	Update interval length [time units]	**60**, 120
W	Querying window size [time units]	0, 20, **40**, 80, 160, 320
QS	Query size [% of the data space]	0.1, **0.25**, 0.5, 1, 2

Table 1: Workload Parameters

for H was explored in two sets of experiments, for data with $UI = 60$ and for data with $UI = 120$. Workloads with uniform data were run using the TPR-tree. Different values of H were tried out in each set of experiments.

Figure 7 shows the results for $UI = 60$. Curves are shown for experiments with different querying windows W. The leftmost point of each curve corresponds to a setting of $H = 0$.

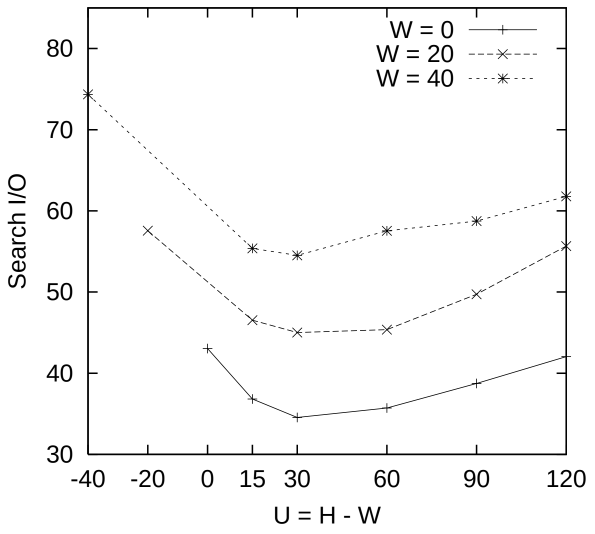

Figure 7: Search Performance For $UI = 60$ and Varying Settings of H

The experiments demonstrate a pattern, namely that the best values of H lie between $UI/2 + W$ and $UI + W$. This is not surprising. In $UI/2$ time units, approximately half of the entries of each leaf node in the tree are updated, and after UI time units, almost all entries are updated. The leaf-node bounding rectangles, the characteristics of which we integrate using H, survive approximately similar time durations. In the subsequent studies, we use $H = UI/2 + W$.

4.3 Comparing the TPR-Tree To Its Alternatives

A set of experiments with varying workloads were performed in order to compare the relative performance of the R-tree, the TPR-tree with load-time bounding rectangles, and the TPR-tree with update-time bounding rectangles.

For the former, the regular R*-tree is used to store fragments of trajectories of points in $(\bar{x}, t)$-space. For this to work correctly, the inserted trajectory fragment for a moving point should start at the insertion time and should span H time units, where H is at least equal to the maximum possible period between two successive updates of the point. Not meeting this requirement, the R-tree may return incorrect query results because its bounding rectangles "expire" after H time units. In our simulation-generated workloads, the slowest moving points on routes spanning from one side of the data space to the other may not be updated for as much as 600 time units. For the R-tree we, thus, set $H = 600$, which is the duration of the simulation.

Figure 8 shows the average number of I/O operations per query for the three indices when the number of destinations in the simulation is varied. Decreasing the number of destinations adds skew to the distribution of the object positions and their velocity vectors. Thus, uniform data is an extreme case.

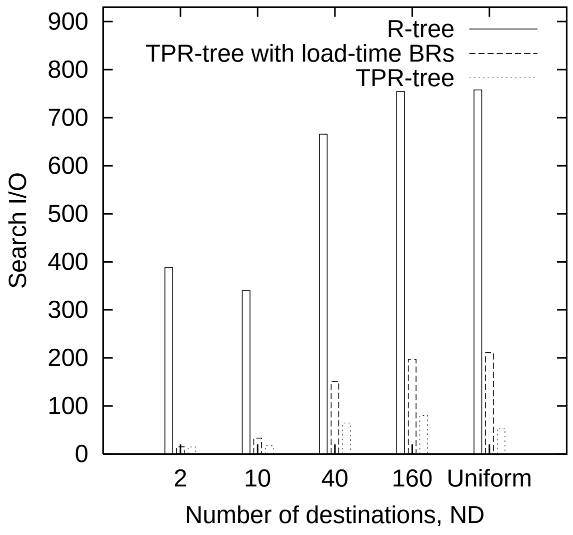

Figure 8: Search Performance For Varying Numbers of Destinations and Uniform Data

As shown, increased skew leads to a decrease in the numbers of I/Os for all three approaches, especially for the TPR-tree. This is expected because when there are more objects with similar velocities, it is easier to pack them into bounding rectangles that have small velocity extents and also are relatively narrow in the spatial dimensions.

The figure demonstrates that the TPR-tree is an order of magnitude better than the R-tree. The utility of update-time bounding rectangles can also be seen, although it should be noted that tightening of bounding rectangles increases the update cost. For example, for a workload with 10 destinations, the use of update-time bounding rectangles decreases the average number of I/Os for searches from 33 to 17, while update cost changes from 1.3 to 1.6 I/Os. For uniform data, the change is from 211 to 54, for searches, and from 2 to 3.5, for updates.

Figure 9 explores the effect of the length of the querying window, *W*, on querying performance. The relatively constant performance of the TPR-tree may be explained by noting that the data in this experiment is skewed ($ND = 20$), with groups of points having similar velocity vectors. Results would be different for uniform data (cf. Figure 7). The relatively constant performance of the R-tree can be explained by viewing the three-dimensional minimum bounding rectangles used in this tree as two-dimensional bounding rectangles that do not change over time. That is why queries issued at different future times have similar performance.

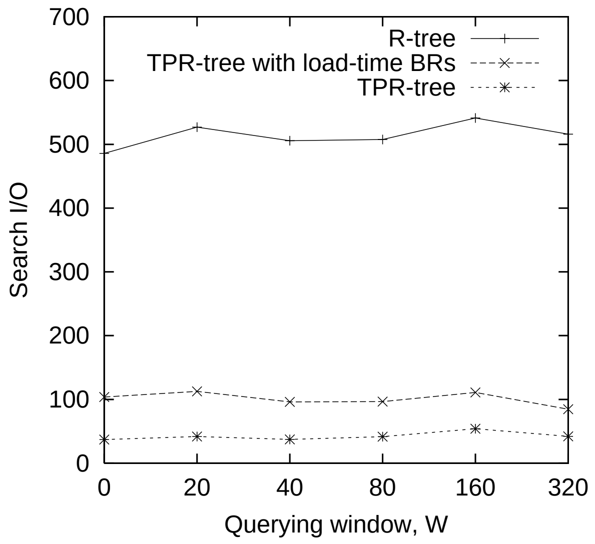

Figure 9: Search Performance for Varying W

Next, Figure 10 shows the average performance for queries with different-size spatial extents. The experiments were performed with three-dimensional data. The relatively high costs of the queries in this figure are indicative of how the increased dimensionality of the data adversely affects performance. An experiment with an R-tree using the shorter *H* of 120 is also included. Using this value for *H* is possible because uniform data is generated where no update interval is longer than $2UI$, and $UI = 60$ in our experiments. This significantly improves the performance of the R-tree, but it remains more than a factor of two worse than the TPR-tree.

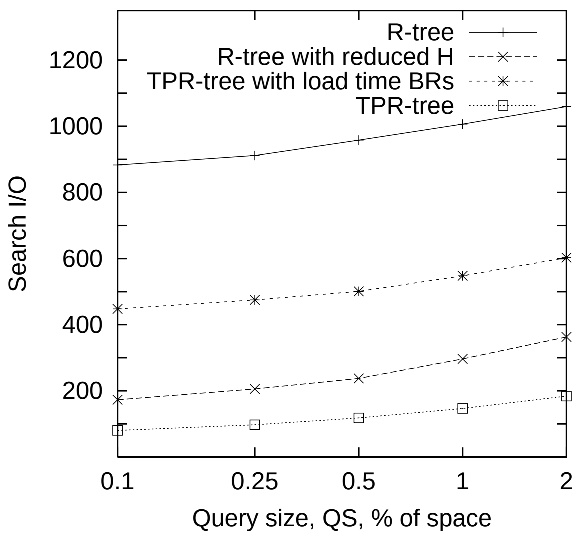

Figure 10: Search Performance For Varying Query Sizes and Three-Dimensional Data

To investigate the scalability of the TPR-tree, we performed experiments with varying numbers of indexed objects. When increasing the numbers of objects, we also scaled the spatial dimensions of the data space so that the density of objects remained approximately the same and so that the number of objects returned by a query was largely (although not completely) unaffected. This scenario corresponds to merging databases that are covering different areas into a single database. Uniform two-dimensional data was used in these experiments.

Figure 11 shows that, as expected, the number of I/O operations for the TPR-tree with update-time bounding rectangles remains almost constant (as long as the number of levels in the tree does not change). The results for the R-tree are not provided, because of excessively high numbers of I/O operations.

To explore how the search performances of the indices evolve with the passage of time, we compute, after each 60 time units, the average query performance for the previous 60 time units. Figure 12 shows the results. In this experiment (and in other similar experiments), the performance of the TPR-tree after 360 time units becomes more than two times worse than the performance at the beginning of the experiment, but from 360 to 600, no degradation occurs. This behavior is similar to the degradation of the performance of most multidimensional tree structures. When, after bulkloading, dynamic updates are performed, node splits occur, the average fan-out of the tree decreases, and the bounding rectangles created by the bulkloading algorithm change. After some time, the tree stabilizes.

As expected, the TPR-tree with load-time bounding rectangles shows an increasing degradation of performance. The bounding rectangles computed at bulkloading time become

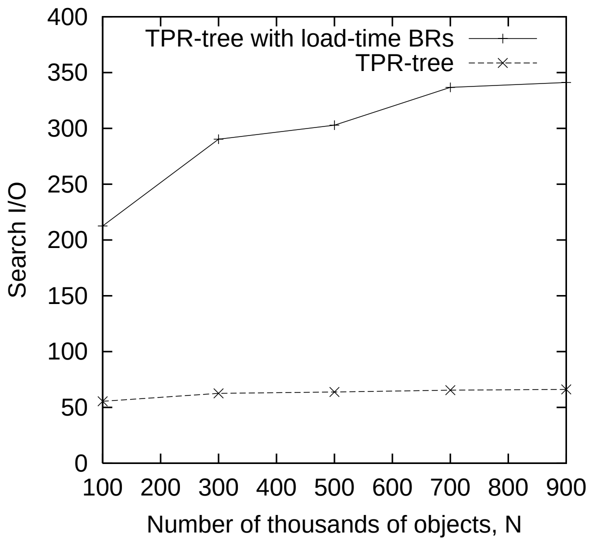

Figure 11: Search Performance for Varying Number of Objects

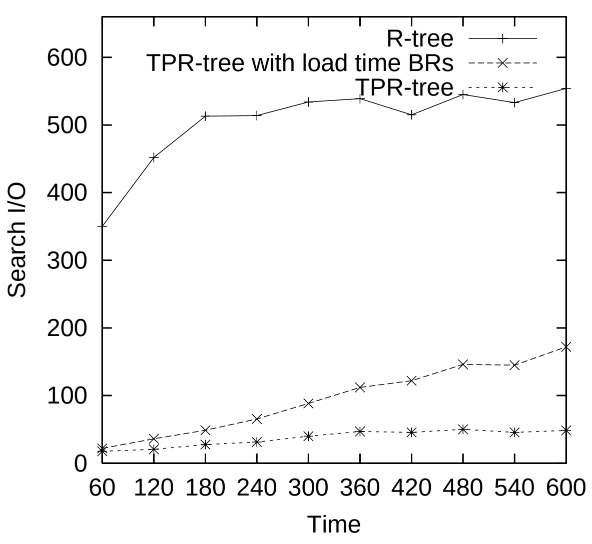

Figure 12: Degradation of Search Performance with Time

unavoidably larger as the more distant future is queried. The insertion algorithms try to counter this by making the velocity extents of bounding rectangles as small as possible. For example, in this experiment the average velocity extent of a rectangle (in one of the two velocity dimensions) is 1.32 after the bulkloading and becomes 0.35 after 600 time units (recall that the extent of the data space in each velocity dimension is 6 in our simulation).

5 Summary and Future Work

Motivated mainly by the rapid advances in positioning systems, wireless communication technologies, and electronics in general, which promise to render it increasingly feasible to track the positions of increasingly large collections of continuously moving objects, this paper proposes a versatile adaptation of the R*-tree that supports the efficient querying of the current and anticipated future locations of moving points in one-, two-, and three-dimensional space.

The new TPR-tree supports timeslice, window, and so-called moving queries. Capturing moving points as linear functions of time, the tree bounds these points using so-called conservative bounding rectangles, which are also time-parameterized and which in turn also bound other such rectangles. The tree is equipped with dynamic update algorithms as well as a bulkloading algorithm. Whereas the R*-tree's algorithms use functions that compute the areas, margins, and overlaps of bounding rectangles, the TPR-tree employs integrals of these functions, thus taking into consideration the values of these functions across the time when the tree is queried. The bounding rectangles of tree nodes that are read during updates are tightened, the objective being to improve query performance without affecting update performance much. When splitting nodes, not only the positions of the moving points are considered, but also their velocities.

Because no other proposals for indexing two- and three-dimensional moving points exist, the performance study compares the TPR-tree with the TRP-tree without the tightening of bounding rectangles during updates and with a relatively simple adaptation of the R*-tree. The study indicates quite clearly that the TPR-tree indeed is capable of supporting queries on moving objects quite efficiently and that it outperforms its competitors by far. The study also demonstrates that the tree does not degrade severely as time passes. Finally, the study indicates how the tree can be tuned to take advantage of a specific update rate.

This work points to several interesting research directions. Among these, it would be interesting to study the use of more advanced bounding regions as well as different tightening frequencies of these. While the tightening of bounding rectangles increases query performance, it negatively affects the update performance, which is also very important. Next, periodic, partial reloading of the tree appears worthy of further study. It may also be of interest to include support for transaction time, thus enabling the querying of the past positions of the moving objects as well. This may be achieved by making the tree partially persistent, and it will likely increase the data volume to be indexed by several orders of magnitude.

Acknowledgments

This research was supported in part by a grant from the Nykredit Corporation; by the Danish Technical Research Council, grant 9700780; by the US National Science Foundation, grant IRI-9610240; and by the CHOROCHRONOS project, funded by the European Commission, contract no. FMRX-CT96-0056.

References

[1] P. K. Agarwal et al. Efficient Searching with Linear Constraints. In *Proc. of the PODS Conf.*, pp. 169–178 (1998).

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. of the PODS Conf.*, to appear (2000).

[3] L. Arge, V. Samoladas, and J. S. Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing. In *Proc. of the PODS Conf.*, pp. 346–357 (1999).

[4] J. Basch, L. Guibas, and J. Hershberger. Data Structures for Mobile Data. In *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 747–756 (1997).

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

[6] B. Becker et al. An Asymptotically Optimal Multiversion B-Tree. *The VLDB Journal* 5(4): 264–275 (1996).

[7] R. Bliujūtė, C. S. Jensen, S. Šaltenis, and G. Slivinskas. R-tree Based Indexing of Now-Relative Bitemporal Data. In *the Proc. of the 24th VLDB Conf.*, pp. 345–356 (1998).

[8] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing Queries By Linear Constraints. In *Proc. of the PODS Conf.*, pp. 257–267 (1997).

[9] O. Günther and E. Wong. A Dual Approach to Detect Polyhedral Intersections in Arbitrary Dimensions. *BIT*, 31(1): 3–14 (1991).

[10] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. of the VLDB Conf.*, pp. 562–573 (1995).

[11] I. Kamel and C. Faloutsos. On Packing R-trees. In *Proc. of the CIKM*, pp. 490–499 (1993).

[12] J. Karppinen. Wireless Multimedia Communications: A Nokia View. In *Proc. of the Wireless Information Multimedia Communications Symposium*, Aalborg University, (November 1999).

[13] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *Proc. of the PODS Conf.*, pp. 261–272 (1999).

[14] W. Konháuser. Wireless Multimedia Communications: A Siemens View. In *Proc. of the Wireless Information Multimedia Communications Symposium*, Aalborg University, (November 1999).

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

[16] S. T. Leutenegger and M. A. Lopez. The Effect of Buffering on the Performance of R-Trees. In *Proc. of the ICDE Conf.*, pp. 164–171 (1998).

[17] J. Moreira, C. Ribeiro, and J. Saglio. Representation and Manipulation of Moving Points: An Extended Data Model for Location Estimation. *Cartography and Geographical Information Systems*, to appear.

[18] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an Analysis of Range Query Performance in Spatial Data Structures. In *Proc. of the PODS Conf.*, pp. 214–221 (1993).

[19] H. Pedersen. Alting bliver on-line. *Børsen Informatik*, p. 14, September 28, 1999. (In Danish)

[20] D. Pfoser and C. S. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. of the SSDBM Conf.*, pp. 111–132 (1999).

[21] D. Pfoser, Y. Theodoridis, and C. S. Jensen. Indexing Trajectories of Moving Point Objects. Chorochronos Tech. Rep. CH–99–3, June 1999.

[22] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[23] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. Technical Report R-99-5009, Department of Computer Science, Aalborg University (1999).

[24] S. Šaltenis and C. S. Jensen. R-Tree Based Indexing of General Spatio-Temporal Data. TimeCenter Tech. Rep. TR-45 (1999).

[25] A. Schieder. Wireless Multimedia Communications: An Ericsson View. In *Proc. of the Wireless Information Multimedia Communications Symposium*, Aalborg University, (November 1999).

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

[27] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. of the SSDBM Conf.*, pp. 111–122 (1998).

[28] 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).