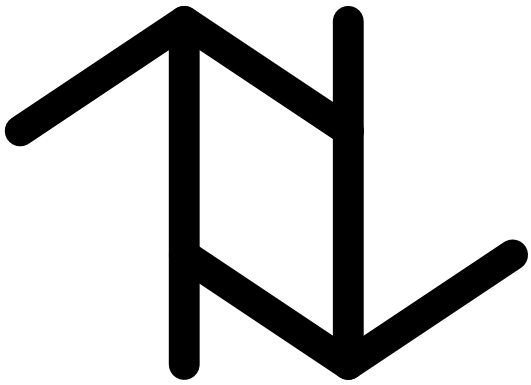

Indexing of Moving Objects for Location-Based Services

Simonas Šaltenis and Christian S.Jensen

TR-63

A TIMECENTER Technical Report

Title	Indexing of Moving Objects for Location-Based Services
	Copyright © 2001 Simonas Šaltenis and Christian S.Jensen. All rights reserved.
Author(s)	Simonas Šaltenis and Christian S.Jensen
Publication History	July 2001. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Dengfeng Gao, Vijay Khatri, Bongki Moon, Sudha Ram

Individual participants
Curtis E. Dyreson, Washington State University, USA
Fabio Grandi, University of Bologna, Italy
Nick Kline, Microsoft, USA
Gerhard Knolmayer, Universty of Bern, Switzerland
Thomas Myrach, Universty of Bern, Switzerland
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, University of Alberta, Canada
John F. Roddick, University of South Australia, Australia
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, amazon.com, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, University of Mons-Hainaut, Belgium
Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/TimeCenter>

Any software made available via TIMECENTER *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

With the continued proliferation of wireless networks, e.g., based on such evolving standards as WAP and Bluetooth, visionaries predict that the Internet will soon extend to billions of wireless devices, or objects. A substantial fraction of these will offer their changing positions to the (location-based) services, they either use or support. As a result, software technologies that enable the management of the positions of objects capable of continuous movement are in increasingly high demand. This paper assumes what we consider a realistic Internet-service scenario where objects that have not reported their position within a specified duration of time are expected to no longer be interested in, or of interest to, the service. In this scenario, the possibility of substantial quantities of "expiring" objects introduces a new kind of implicit update, which contributes to rendering the database highly dynamic. The paper presents an R-tree based technique for the indexing of the current positions of such objects. Extensive performance experiments explore the properties of the types of bounding regions that are candidates for being used in the internal entries of the index, and they show that, when compared to the approach where the objects are not assumed to expire, the new indexing technique can improve the search performance by as much as a factor of two or more without sacrificing update performance.

1 Introduction

We are currently experiencing rapid developments in technology areas, such as wireless technology, miniaturization of electronics, and ergonomics. This development promises widespread use of mobile personal information appliances, most of which will be on-line, i.e., on the Internet. Industry analysts uniformly predict that wireless, mobile Internet terminals will outnumber the desktop computers in the Internet.

This proliferation of devices offers companies the opportunity to provide a diverse range of e-services. Successful services are expected to be relevant, unobtrusive, personalized, and context aware; and it is essential for many services, termed location-based services, that they be sensitive to the user's changing location. Location awareness is made possible by a combination of political developments, e.g., the descrambling of the GPS signals and the US E911 mandate, and the continued advances in both infrastructure-based and handset-based positioning technologies.

The area of location-based games offers good examples of services where there is a need to track the positions of the mobile users. In the recent released BotFighters game, by Swedish company It's Alive, players get points for finding and "shooting"other players via their mobile phones (using SMS messages or using WAP). Only players close by can be shot. To enable the game, players can request the positions of other nearby players. In such mixed-reality games, the real physical world becomes the backdrop of the game, instead of the world created on the limited displays of wireless devices [8]. These games are expected to generate very large revenues in the years to come. Datamonitor, a market research company, estimates that 200 million mobile phone users in western Europe and the United States will play Web games via handsets, generating $6 billion in revenues [19]. To track and coordinate large numbers of continuously moving objects, their positions are stored in databases.

Continuous movement poses new challenges to database technology. The conventional assumption is that data remains constant unless it is explicitly modified. Capturing continuous movement accurately with this assumption requires very frequent updates. To reduce the number of updates required, functions of time that express the objects' positions may be stored instead of simply the static positions [23]. Then updates are necessary only when the parameters of the functions change "significantly." 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.

Independently of how object positions are represented, the accuracy of the positions and, thus, their utility for providing a location-based service decreases as time passes, so when an object has not reported its position for a certain period of time, the recorded position is likely to be of little use. Consequently, it

is natural to associate expiration times with positions so that these can be disregarded. The system, then, should automatically remove such "expired" information.

To provide fast answers to queries that locate the mobile objects in a certain area, an index on object positions must be maintained. No previous work on the indexing of the positions of continuously moving objects (discussed in Section 2.2) has addressed the issue of "expiring" information. This paper proposes the $\mathrm{R}^{\mathrm{EXP}}$-tree, an R*-tree [5] based access method that builds on the ideas of the TPR-tree [21]. The $\mathrm{R}^{\mathrm{EXP}}$-tree indexes the current and anticipated future positions of moving point objects, assuming that their positions expire after specified time periods.

To take advantage of information being valid only for a limited time, the proposed index uses a new type of bounding region. We show that the choice of bounding regions is non-trivial, and we experimentally compare a number of possible alternatives. In addition, we propose a modification of the R*-tree insertion and deletion algorithms that, during the regular index update operations, lazily removes expired information from the index. We provide also a mechanism by which the $\mathrm{R}^{\mathrm{EXP}}$-tree algorithms automatically tune themselves to match an important characteristic of the workloads they are subjected to—the average update rate. Finally we provide experimental comparison of the new index with the existing index, the TPR-tree, which assumes non-expiring information.

The next section presents the problem addressed by the paper and covers related research. As a precursor to presenting the new index, Section 3 explores issues related to the use of existing moving-object indexes, e.g., the TPR-tree, for the indexing of data with expiration times. In Section 4, this is followed by a description of the bounding regions and algorithms employed by the new index. It is assumed that the reader has some familiarity with the R*-tree. Section 5 reports on performance experiments, and Section 6 summarizes and offers research directions.

2 Problem Statement and Related Work

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

2.1 Problem Statement

An object's position at 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. We model this position 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}, $\bar{x}(t_{ref})$, termed the reference position. The second parameter is a velocity vector for the object, $\bar{v} = (v_1, v_2, \ldots, v_d)$. Thus, $\bar{x}(t) = \bar{x}(t_{ref}) + \bar{v}(t - t_{ref})$. Although the times ($t_{obs}$) when different objects were most recently sampled may differ, it is convenient for the indexing purposes to have the reference position for all objects be associated with a single reference time, t_{ref}. Such a reference position can always be computed knowing the velocity vector $\bar{v}$ observed at t_{obs} and the position $\bar{x}(t_{obs})$ observed at t_{obs}.

Modeling object positions as functions of time not only enables us to make tentative near-future predictions, but, more importantly, alleviates the problem of the frequent updates that would otherwise be required to approximate continuous movement in a traditional setting where only positions are stored. In our setting, objects may report their parameter values when their actual positions deviate from what they have previously reported by some threshold. The choice of update frequency then depends on the type of movement, the desired accuracy, and the technical limitations [24, 16]. For example, a mobile yellow pages service is likely to be much less sensitive than a traffic monitoring system to imprecise positions.

An object's reference position and velocity vector describe its predicted movement from now and indefinitely far into the future. In the applications we consider, such far-reaching predictions are not possible. An object does not usually move for a long period of time within a useful threshold of its predicted movement.

Rather, if such an object does not report the new, up-to-date position and velocity, after some time, its old positional information becomes too imprecise to be useful—we say that it expires.

To avoid reporting such expired objects in response to queries, we associate an expiration time, t_{exp}, with each object and call them expiring objects. If unknown, the expiration time can be set to infinity, although, it in most cases should be easy to find a finite upper bound. Upper bounds can be dictated by a number of application specific factors. For example, moving objects may be forced to make changes in their movement due to an underlying infrastructure such as a road network, or objects may move according to some predetermined routes and schedules, as in a public transportation system [6]. Finally, trivial upper bounds on the expiration times can be derived from the finite extents of the space where the objects move.

Figure 1 exemplifies how predicted trajectories of moving objects are recorded and updated. For simplicity, one-dimensional moving objects, such as cars on a road, are shown. The positions of objects are plotted on the y-axis, and time is on the x-axis. The current time is assumed to be 6, and the part of the picture to the left of the current-time line shows the past evolution of the data set, where insertions, deletions, and updates are represented by vertical bars and expiration times are represented by arrows. For indexing purposes, the data set at any time point consists of a set of finite line segments in (x, t)-space.

The figure illustrates that many objects are updated before they expire, while some expire before being updated. For example, object $o1$ was updated at time 2—before its expiration time (3). But then no update occurred prior to its new expiration time (5). The latter may be more common in applications with unreliable or intermittent connectivity. For example, mobile telephones that are turned off may not be guaranteed to report to the system. In these cases, only expiration times guarantee that objects are removed from the data set.

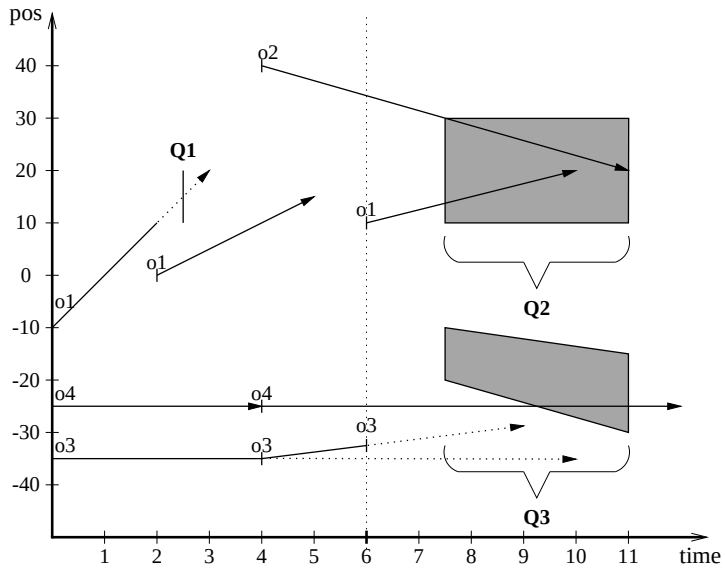

Figure 1: Example One-Dimensional Data Set and Queries

The figure also exemplifies the types of queries that we aim to support. These queries retrieve all objects with predicted positions within specified regions at specified times. We distinguish between three kinds, based on the space-time regions they specify. In the sequel, a d-dimensional rectangle R is specified by its d projections $[a_1^\vdash, a_1^\dashv], \ldots [a_d^\vdash, a_d^\dashv]$, $a_j^\vdash \leq a_j^\dashv$, onto the coordinate axes. Let R, R_1, and R_2 be three d-dimensional rectangles and t, $t^\vdash$, and $t^\dashv$ ($t^\vdash < t^\dashv$) be three times that do not precede 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 $([a_1^\vdash, a_1^\dashv], [a_2^\vdash, a_2^\dashv], \ldots, [a_d^\vdash, a_d^\dashv], [t^\vdash, t^\dashv])$.

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. The types of

queries are exemplified in Figure 1, where $Q1$ is a timeslice query, $Q2$ is a window query, and $Q3$ is a moving query.

Notice that queries are positioned on the time axis according to the times t, $t^{\vdash}$, and $t^{\dashv}$ specified in the queries, not according to the time they were issued. The greater the distance between these times and the query issue time, the more tentative the answer of the query is because objects update their parameters as time goes. For example, the answer to query $Q1$ would have been $o1$, if it were issued before time 2, and no object would have qualified for this query if it were issued later because $o1$ was updated at time 2.

This example illustrates that queries far in the future are likely to be of little value, because the positions as predicted at the query time become less and less accurate as queries move into the future and because updates may occur. (The usage of expiration times will eliminate many "wrong" objects from the answers to such queries.) We expect applications to issue queries that are concentrated in some limited time window extending from the current time. The more frequently the parameters of the objects are updated, the shorter this window is likely to be.

We introduce a problem parameter, *querying window length* (*W*), which represents an expected upper bound on how far queries "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, where $iss(Q)$ is the query issue time. The querying window is one of the important problem parameters used by our proposed index algorithms.

2.2 Previous Work

We consider first approaches that involve the partitioning the space into which the objects are embedded, then consider R-tree based approaches.

2.2.1 Approaches Based on Data Space Partitioning

Early work on the considered indexing problem has concentrated mostly on points moving in one-dimensional space.

Tayeb et al. [22] use PMR-Quadtrees [20] for indexing the future trajectories of one-dimensional moving points as line segments in (x, t)-space (cf. Figure 1). The segments span the time interval that starts at the index construction time and extends some fixed number of time units into the future, which leads to a tree that has to be rebuilt periodically. This approach introduces substantial data replication in the index—a line segment is usually stored in several nodes.

Kollios et al. [13] employ the so-called 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.

Basch et al. [4] propose so-called kinetic main-memory data structures for mobile objects. The idea is that even though the objects move continuously, the relevant combinatorial structure changes only at certain discrete times, e.g., when two points pass each other. Thus, future events are scheduled that update a data structure at these times so that necessary invariants of the structure hold. Agarwal et al. [1] apply these ideas to external range trees [3], obtaining a data structure can answer a Type 2 query in optimal $O(\log_B n + k/B)$ I/Os using only slightly more than a linear number of disk blocks (here B is the disk block size, n is the number of objects, and k is the size of the query result). This result holds only when queries arrive in chronological order—once a kinetic event has changed the data structure, no queries can refer to time points before the event. 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. Although achieving good asymptotic bounds that are very important from a theoretical point of view, the practical utility of the structures remain in question.

2.2.2 Approaches Based on Time-Parameterized Bounding Rectangles

As the technique proposed in this paper builds on the basic ideas of the TPR-tree, we review briefly the main ideas of the TPR-tree and other related access methods.

The TPR-tree is based on the R*-tree and indexes points that move in one, two, or three dimensions. It employs the basic structure and algorithms of the R*-tree, but the indexed points 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 velocities of the edges of bounding rectangles are chosen so that the enclosed moving objects remain inside the rectangles at all times in the future. Figure 2 demonstrates this. Here, three one-dimensional moving points are shown together with their one-dimensional bounding rectangle (i.e., a bounding interval). The figure shows that answering window or moving queries in the TPR-tree involves the checking for intersection between two $(d+1)$-dimensional trapezoids—a query and a bounding rectangle.

In addition to the usage of the time parameterized bounding rectangles, the TPR-tree differs from the R*-tree in how its insertion algorithms group points into nodes. The R*-tree uses the heuristics of minimized area, overlap, and margin of bounding rectangles to assign points to the nodes of a tree. To take into account the temporal evolution of these heuristics, they are replaced by their integrals over time in the TPR-tree. The area of the shaded region in Figure 2 shows the 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.

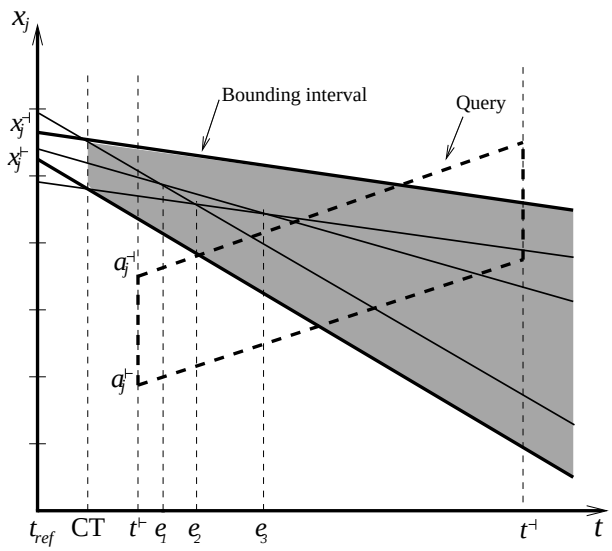

Figure 2: A Bounding Interval and a Query in the TPR-tree

The bounding interval in Figure 2 is minimum only at the current time (CT). At later times, it is larger than the true minimum bounding interval. It is possible to record a true minimum bounding interval by storing all future events when the true minimum bounding interval changes (e_1, e_2, and e_3 in the figure), but this is impractical because the number of such events in the worst case is equal to the number of objects enclosed by the bounding interval. Agarwal et al. [2] show how to reduce the number of events to a constant depending on ϵ by allowing the bounding interval at all times to have a length no larger than $(1+\epsilon)$ of the length of the minimum bounding interval.

Based on these ideas, Pocopiuc et al. [18] propose the STAR-tree index for moving objects. In contrast to the TPR-tree, the STAR-tree groups points according to their current locations. This may result in points moving with very different speeds being included in the same rectangle. To avoid such bounding rectangles growing too much, special events are scheduled to regroup the points. The STAR-tree is most suited for workloads with very infrequent updates.

Revesz et al. [6] present a data model for spatiotemporal data based on parameteric rectangles that

closely resemble the bounding rectangles inside the TPR-tree. The spatiotemporal evolution of an object with extent is described by a number of parametric rectangles, each associated with a specific validity time interval.

Cai and Ravesz have recently proposed a Parametric R-tree [7] that is quite similar to the TPR-tree. The main difference is that they index the *past* evolution of objects with extent, meaning that they know at index construction time the entire evolution of the objects. This is a different and, in some ways, simpler problem than the one addressed here. Indexing of the past trajectories of moving points represented as polylines (connected line segments) in ($\bar{x}$, t)-space is also the theme of papers by Pfoser et al. [17] and Kollios et al. [14].

3 Indexing Expiring Objects With the TPR-Tree

The TPR-tree presented in the previous section indexes the future trajectories of moving objects as infinite lines. The future trajectories of expiring objects may also be indexed with the TPR-tree, by replacing the finite line segments of the expiring objects with corresponding infinite lines.

This setup introduces two issues that must be addressed. First, objects that have expired by the times specified in a query may introduce false drops in query answers, leading to overly large intermediate results from the index that must be filtered to produce the correct answer. Second, it may be desirable to have automatic means of deleting expired objects, which clutter the index.

One way of eliminating expired entries is to schedule deletions. To accomplish this, a secondary data structure is required that maintains the resulting queue of scheduled deletions. This structure must support operations not only for checking and removing the top element of the queue and inserting the new element, but also for efficiently deleting or updating any of the scheduled deletion events in the queue. This latter functionality is necessary because objects may be deleted or updated before they expire.

Such a structure does not generally fit in main memory, as its size is on the order of the size of the primary index structure. A B-tree on the composite key of the expiration time and the object id could be used. The topmost element of the queue can be found easily in the leftmost leaf page of the tree, and the insertion, deletion, and update operations can be performed efficiently.

In such a setting, the amortized cost of introducing one expiring object consists of four terms. First, the object has to be inserted into the TPR-tree. Next, the scheduled deletion event has to be inserted into the B-tree. Finally, when processing the scheduled deletion event, the event has to be removed from the B-tree, and the scheduled deletion has to be performed in the TPR-tree. In Section 5, where we describe our performance experiments, we show that this approach can be competitive with the $\mathrm{R}^{\mathrm{EXP}}$-tree only if the B-tree costs are ignored.

It should be also mentioned that unless queries arrive in chronological order, the scheduling of deletions does not allow to avoid the filtering step in answering future queries. Objects that expire after the current time, but before the query time, are reported as false drops and must be filtered.

4 Structure and Algorithms

This section presents the structure and algorithms of the $\mathrm{R}^{\mathrm{EXP}}$-tree. First, we explore possibilities for computing time-parameterized bounding rectangles by maximally exploiting expiration times. Next, we investigate how the different types of bounding rectangles can be used to guide the grouping of entries in the tree, and how this grouping is automatically adjusted depending on a specific update rate and querying window length. Finally, we describe the modified insertion and deletion algorithms that ensure the efficient disposal of expired entries.

In the following, when only one-dimensional moving points or bounding intervals are mentioned, it is assumed that the extension to higher dimensions is trivially done by applying the same procedure or definition to each of the dimensions, or by exchanging the length of the interval with the area, volume, or hyper-volume. Also, we use the term rectangle for any d-dimensional hyper-rectangle.

4.1 Index Structure and Time Parameterized Bounding Rectangles

The $\mathrm{R}^{\mathrm{EXP}}$-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 (time-parameterized) region that bounds the positions of all moving points or other bounding regions in that subtree.

4.1.1 Representation of Points and Bounding Rectangles in the Index

As suggested in Section 2, the position of a moving point is represented by a reference position, a corresponding velocity vector, and an expiration time—(x, v, t_{exp}) in the one-dimensional case, where $x = x(t_{ref})$. We let t_{ref} be equal to the index creation time, t_0.

To bound a group of d-dimensional moving points, d-dimensional rectangles are used that are also time-parameterized and that enclose 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, as noted in Section 2.2, doing so deteriorates in the general case to enumerating all the enclosed moving points or rectangles. Consider Figure 3. Here, four one-dimensional moving points are bounded with an always minimum bounding interval, which bounds the positions of points tightly at all future time points. Each of the four points defines the upper or lower bound at some time in the evolution of the bounding interval.

To achieve a compact description of the enclosed entries, we use a single linear function as the bound (upper or lower) of the bounding interval. Following the representation of moving points, we let $t_{ref} = t_0$ and capture a one-dimensional time-parameterized bounding interval $[x^{\vdash}(t), x^{\dashv}(t)] = [x^{\vdash}(t_0) + v^{\vdash}(t - t_0), x^{\dashv}(t_0) + v^{\dashv}(t - t_0)]$ for $t < t^{\dashv}_{exp}$ as $(x^{\vdash}, x^{\dashv}, v^{\vdash}, v^{\dashv}, t^{\dashv}_{exp})$. Here $t_{exp} = \max_i\{o_i.t_{exp}\}$, where i ranges over the moving points or bounding intervals to be enclosed.

Note that we could as well choose not to record t_{exp} for bounding rectangles, reducing the size of internal index entries. Even in this case, a "natural," finite t_{exp} can be derived for bounding rectangles that shrink in some dimension, i.e., $v^{\vdash}_i > v^{\dashv}_i$, for some i. For such a rectangle, t_{exp} should be set to the time when its area becomes zero. In performance experiments, we investigate whether it pays off to record expiration times in internal index entries. In the following, we assume that bounding rectangles have expiration times even though some of them may be infinite.

There are a number of possible ways to compute $x^{\vdash}$, $x^{\dashv}$, $v^{\vdash}$, and $v^{\dashv}$. One goal is to choose these parameters so as to minimize the integral of the interval's length from the time of bounding interval computation, t_{upd}, to $t_{upd} + h$, where $h = \min\{H, t_{exp} - t_{upd}\}$ and H approximates how far into the future queries are most likely to access the computed bounding rectangle. We discuss how the value of H is estimated in Section 4.2.

Minimizing this integral is equivalent to minimizing the area (or the part of it between t_{upd} and $t_{upd} + h$) of a trapezoid that bounds the trajectories of the enclosed points or intervals and that has bases orthogonal to the time axis. The shaded region in Figure 2 exemplifies such a trapezoid. In the following, the term "bounding trapezoid" is used to refer to this kind of a trapezoid.

4.1.2 Simple Time-Parameterized Bounding Rectangles

If all entries are infinite, the only reasonable choice—if the interval is described by the above four parameters—is what we term *conservative* bounding rectangles, which are minimum at the point of their computation, but possibly (and most likely!) not at later times. To ensure that a conservative bounding interval is bounding for all future times, the lower bound of the 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 is a very simple construction that is independent of H.

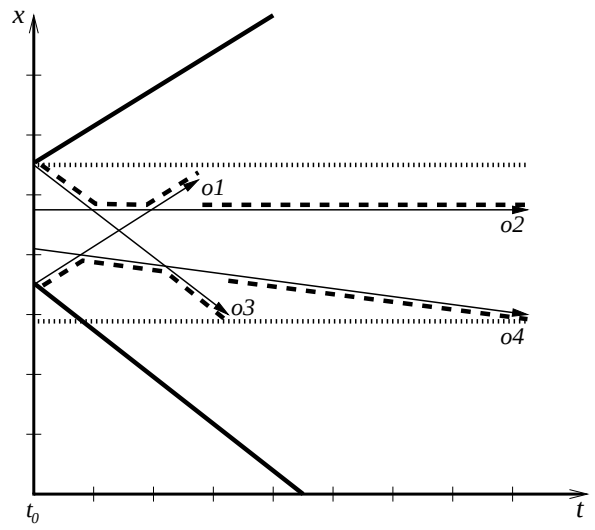

Figure 3: Conservative (Bold), Always Minimum (Dashed), and Static (Dotted) Bounding Intervals

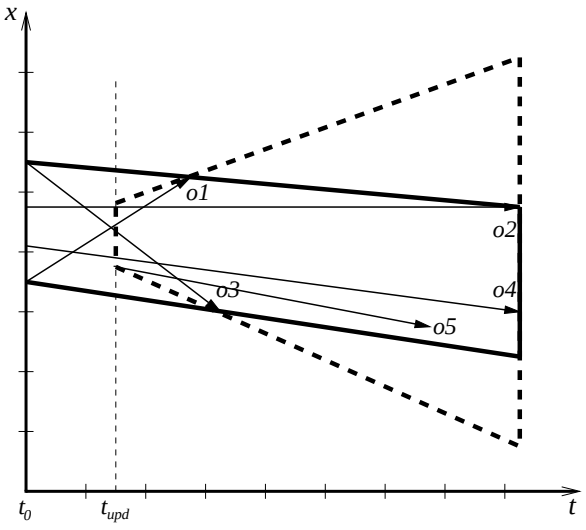

Figure 4: Update-Minimum Interval (Bold) and the Same Interval Recomputed after Insertion of $o5$ (Dashed)

Figure 3 illustrates a conservative bounding interval. This interval bounds the four points tightly at t_0, but to keep all points enclosed at all future times (assuming that objects have infinite trajectories), the upper bound of the interval moves at the speed of object $o1$, while the lower bound of the interval moves at the speed of object $o3$. Although the figure illustrates the concept, it should be noted that the TPR-tree algorithms most likely would not place $o1$ and $o3$ in the same node as $o2$ and $o4$.

The straightforward bounding interval for finite entries has both $v^{\vdash}$ and $v^{\dashv}$ equal to zero and is termed a *static* bounding interval. Figure 3 illustrates such a bounding interval. In addition to being simple, the main advantage of this type of interval is that by not storing $v^{\vdash}$ and $v^{\dashv}$ in the internal index entries, we increase the fan-out of internal tree nodes by almost a factor of two.

The last obvious and simple way of taking advantage of the expiration times is to use improved conservative bounding intervals, where the speed of the upper bound is reduced as much as possible and, analogously, the speed of the lower bound is increased as much as possible. We term such bounding intervals *update-minimum* intervals because, like conservative intervals, they are minimum at the time of the last update. Figure 4 shows how the speeds of the bounds are reduced or increased. Here, the speed of the upper bound of the bounding interval is not set to the speed of the fastest object ($o1$), but to some smaller speed that is enough to contain $o1$, knowing its expiration time. Notice that because the resulting bounding interval is relatively "nice" (it barely grows), the tree algorithms are very likely to group the four given objects in a single node. However, if the bounding interval is recomputed at some later time (t_{upd}), e.g.,

because of the insertion of a new object ($o5$), the interval-length integral is increased unnecessarily. How often this will happen and how it will effect the performance of the index is investigated in performance experiments.

4.1.3 One-Dimensional Optimal Time-Parameterized Bounding Rectangles

As mentioned earlier, the goal is to minimize the area of a trapezoid that is bounding and extends from $t = t_{upd}$ to $t = t_{upd} + h$. To find such a minimum bounding trapezoid, it suffices to consider only the endpoints of trajectories. When we are to bound moving points, each trajectory has one endpoint, and when we are to bound time-parameterized intervals, each trajectory has two endpoints—$x^{\vdash}(t_{exp})$ and $x^{\dashv}(t_{exp})$. In the following, we denote this set of points by S. To capture the positions of points or intervals at t_{upd}, the minimum and maximum of these positions at t_{upd} are included in S. Figure 5 shows these points—$x_{min} = \min_i\{o_i.x^{\vdash}(t_{upd})\}$ and $x_{max} = \max_i\{o_i.x^{\dashv}(t_{upd})\}$, where $o_i.x^{\vdash}(t_{upd}) = o_i.x^{\dashv}(t_{upd}) = o_i.x(t_{upd})$ when points, not intervals, are being bounded. As noted by Cai and Revesz [7], the following lemma holds.

Lemma 4.1 *The lower and the upper bounds of a bounding trapezoid of S with minimum area between times t_{upd} and $t_{upd} + h$ are the lines containing the edges of the convex hull of S that intersect the median line $t = t_{upd} + h/2$.*

Here, the lower and upper bounds of the trapezoid are the lines described by the trajectories of the lower and the upper bound of the corresponding time-parameterized interval.

To understand why this lemma holds, consider the upper bound of the trapezoid. It is trivial that this bound should contain at least one vertex of the upper chain of the convex hull of S. Suppose it contains only vertices of the hull to the left of the median line, and let p be the rightmost of these (cf. Figure 5). Then we can reduce the area of the trapezoid by replacing this upper bound (u') with a line u that has a smaller slope and contains the edge of the convex hull with p as its left point. Figure 5 illustrates why the area is reduced. The shaded triangle to the right of p shows the area that was eliminated, which is larger than the area of the shaded triangle to the left of p that shows the area that was gained. This is true for any p to the left of the median line. We can continue this process until the upper bound contains vertices both to the left of the median line and to the right of it. Similar argument can be made when we start with an upper bound that contains only points to the right of the median line and when the lower bound is considered.

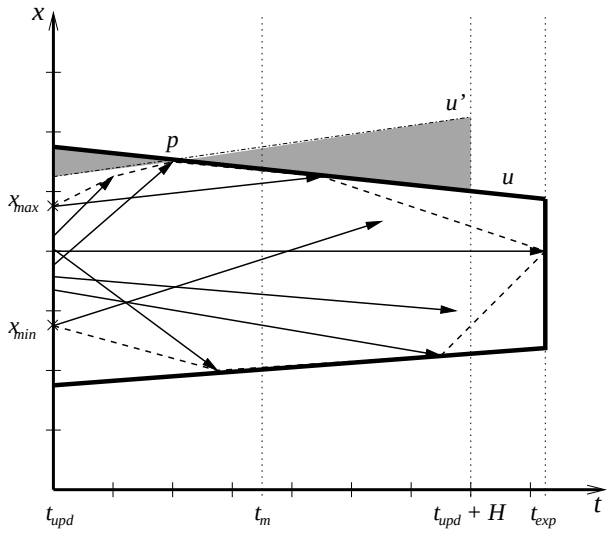

Figure 5: A Convex Hull and an Optimal Bounding Interval

It should be noted that if the median line contains one of the vertices of the convex hull, the median line can be said to cross either the hull's edge to the right of the vertex or to the left of it; either interpretation produces a minimum trapezoid with the same area.

Any of a number of convex-hull computation algorithms (e.g., a Graham scan [9]) can be used to find the convex hull of S in $O(|S| \log |S|)$ time. However, observe that we need to find only the edges of the convex hull that intersect the median line. This can be formulated as a linear programming problem. Inspired by linear programming algorithms, Kirkpatrick and Seidel [12] provide a linear algorithm to find

such edges, which they call "bridges." Nevertheless, compared to the Graham scan, the algorithm is quite complex, and its implementation using finite precision floating point arithmetics is complicated. Therefore our implementation uses a bridge-finding algorithm based on the Graham scan.

4.1.4 Multi-Dimensional Time-Parameterized Bounding Rectangles

The more general problem of finding a minimum Time-Parameterized Bounding Rectangles (TPBR) in multiple dimensions is much harder. It is a non-convex, non-linear mathematics programming problem. We desire simple algorithm that produces "satisfactory" results. One approach is to compute the parameters of the bounding rectangle independently in each dimension [7]. For the i-th dimension, the bridge-finding algorithm could be applied to the projections of the trajectories into the (x_i, t)-plane.

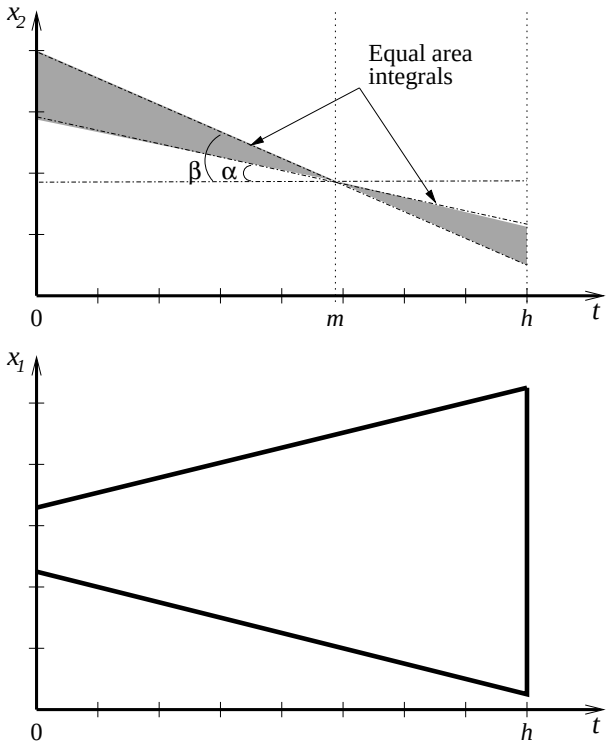

Figure 6: Finding a Median Line for the Second Dimension (Top), When the First Dimension is Computed (Bottom)

It is very easy to improve such a straightforward algorithm without adding complexity. The idea is to introduce dependencies among the dimensions. Specifically, when considering the i-th dimension, the already computed dimensions can be taken into account by adjusting the position of the median line in the bridge-finding algorithm. Note that in the proof of Lemma 4.1, we relied on the fact that a shaded triangle to the left of any point p that is to the left of the median has a smaller area that the triangle to the right of p (cf. Figure 5). If p lies on the median line, both triangles have the same are. In multiple dimensions, not simple areas, but time integrals of hyper-volumes have to be compared.

To understand the issue, consider an example (illustrated in Figure 6). In the first dimension x_1, the computed bounding interval grows from left to right, i.e., with increasing time. Then a unit of bounding interval length in the second dimension x_2 has less weight at smaller times than at later times. Thus, the median line should be shifted to the right when computing its bounding interval.

In the following assume, without loss of generality, that $t_{upd} = 0$. Suppose k dimensions are already computed and we want to find a median line for the computation of the $(k+1)$-st dimension. Let $a_i = x_i^{\dashv} - x_i^{\vdash}$ and $w_i = v_i^{\dashv} - v_i^{\vdash}$, $1 \le i \le k$, be the spatial and velocity extents of the bounding rectangle. Then, considering only the computed dimensions, the hyper-volume at time t is $\prod_{i=1}^{k}(a_i + w_i t) = \sum_{i=0}^{k} C(i)t^i$, where $C(i)$ is the sum of the coefficients at the i-th power of t in the above polynomial.

Lemma 4.2 *If the parameters of a TPBR in the first k dimensions are computed and fixed, the optimal parameters of a TPBR in the $(k+1)$-st dimension can be computed using the median line $t = m$, where*

$$m = \frac{\sum_{i=0}^{k} \frac{h^{i+1}}{i+2} C(i)}{\sum_{i=0}^{k} \frac{h^i}{i+1} C(i)} .$$

PROOF: As mentioned earlier, the median line has the property that, for any point contained in it, the hyper-volume integral corresponding to its left "shaded triangle" is equal to the integral corresponding to its right "shaded triangle" (cf. Figure 6). The left integral is $I_l = \int_0^m (\tan\beta - \tan\alpha)(m-t)\sum_{i=0}^{k} C(i)t^i dt$. The right integral is $I_r = \int_m^h (\tan\beta - \tan\alpha)(t-m)\sum_{i=0}^{k} C(i)t^i dt$. It is not difficult to see that $I_r = I_l + (\tan\beta - \tan\alpha)(\sum_{i=0}^{k} \frac{h^{i+1}}{i+2}C(i) - m\sum_{i=0}^{k} \frac{h^i}{i+1}C(i))$. Solving the equation $I_l = I_r$ for m proves the lemma. □

As an example, if $k = 2$, $m = h(3a_1 + 2w_1 h)/(6a_1 + 3w_1 h)$.

Using this lemma, our algorithm for computing a multi-dimensional TPBR visits dimensions one by one until TPBR parameters in all the dimensions have been computed. The order in which dimensions are visited may influence the resulting TPBR. We choose a random order, so that no dimension is given preference. This algorithm, combined with a linear bridge-finding algorithm, has an expected-case running time of $O(d|S|)$, where d is the number of dimensions. We term the bounding rectangles produced by this algorithm *near-optimal*.

To measure the loss in performance caused by using near-optimal TPBRs, we implemented an algorithm that computes optimal multi-dimensional TPBRs. The idea is to compute convex hulls in each of the (x_i, t)-planes. Then, by using sweeping median lines in each of the first $d-1$ dimensions, we consider all possible combinations of choices of bridge-edges in these dimensions. Finally, for each of the combinations, using Lemma 4.2, we compute m for the d-th dimension and find the bridge-edges in this dimension using binary search on the edges of the convex hull in this dimension. The worst-case running time of this algorithm is $O(|S|^{d-1}\log|S|)$.

Although we do not discuss this in detail, the presented algorithms can be easily generalized to handling the case when some of the bounded points or rectangles have infinite expiration times.

Section 5 investigates the differences in performance resulting from using static, update-minimum, near-optimal, or optimal TPBRs.

4.1.5 Using TPBRs when Querying

When answering a query in the $\mathrm{R}^{\mathrm{EXP}}$-tree, we need to be able to check whether the trapezoid formed by the query intersects the trapezoid formed by a TPBR. The same algorithm as in TPR-tree can be used [21], the only modification being that intersection should be checked between $t^{\vdash} = t_q^{\vdash}$ and $t^{\dashv} = \min(t_q^{\dashv}, t_{exp})$, where $[t_q^{\vdash}, t_q^{\dashv}]$ is the time interval specified in the query.

4.2 Heuristics for Tree Organization

We proceed to describe the heuristics that determine how to group moving objects and their TPBRs into nodes so that the tree most efficiently supports queries when assuming a querying window length, *W*.

4.2.1 Integrated R*-Tree Heuristics

Intuitively, when the index is under continuous change due to time parameterization, the decision of where to place an object's updated, time-varying position should take into account the evolution of the relevant parts of the index from the current time to the next update of the same object or to the time the object expires. Thus, the average duration (UI) between the two successive updates of an object is an important problem parameter, which is directly related to the frequency of index updates. Similar considerations apply to node splits. On average, a split-generated distribution of entries will not persist longer than UI time units, upon which most of the entries of both nodes have been updated.

Although the decisions about an entry's placement will affect queries only for UI time units, some queries may be future queries that look as far as *W* time units into the future. Thus, the total length of the time duration when queries will "see" the current insertion is $H = UI + W$. We term this the *time horizon*.

It should be noted that the above considerations about a single object cannot be stated strictly because, in addition to the continuous change due to time-parameterization, the index is constantly changing due to the updates of other objects. Nevertheless, performance experiments with the TPR-tree demonstrate that insertion algorithms using the time horizon $H = UI + W$ consistently show good query performance.

The insertion algorithms of the R*-tree, which we extend 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 the interval $[t_{upd}, t_{upd} + H]$. Specifically, given an objective function $A_r(t)$ of a bounding rectangle r, we replace it with the following integral in the insertion algorithms.

$$\int_{t_{upd}}^{t_{upd}+\min\{H, r.t_{exp}\}} A_r(t)dt \tag{1}$$

If $A_r(t)$ 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 2). Note, that for objective functions depending on the two bounding rectangles, say r_1 and r_2, the upper integration bound becomes $t_{upd} + \min\{H, r_1.t_{exp}, r_2.t_{exp}\}$. The computation of such integrals is described in more detail in [21].

4.2.2 Bounding Rectangles and Grouping of Entries

While the replacement of the objective functions of area, distance, margin, and overlap of bounding rectangles with the corresponding integrals represents a natural and simple adoption of the R*-tree algorithms, additional issues may need to be addressed.

One of them is what we term the *non-associativity* of TPBR computation. In the R-trees, with the Hilbert R-tree [11] as a notable exception, the bounding rectangles are used for two purposes—pruning search (in queries) and guiding insertion decisions. Suppose that the insertion algorithm has reached a node that is a parent to leaf nodes and it must be decided which of the child nodes should receive a new entry. The R*-tree chooses a child node using the ChooseSubtree algorithm, which asks a "what if" question for each child node. Specifically, for each child node, the algorithm computes the area of the overlap between the sibling bounding rectangles and the existing "old" bounding rectangle, and then compares it with the area of overlap between the "new" bounding rectangle, obtained by extending the existing bounding rectangle to include the new entry, and the sibling rectangles. In the R*-tree, the "new" bounding rectangle is exactly equal to the bounding rectangle that the node would have if the new entry were inserted into it and the bounding rectangle were computed by looking at all its entries, including the new one. We use the term *associativity* for this property of bounding rectangle computation.

The TPBRs of the same entries computed at different times may be different. Thus, the TPBR computation is non-associative, if computations of the "old" and "new" TPBRs are performed at different times. In addition, for optimal or near-optimal TPBRs, the computation is non-associative even if the computations are performed simultaneously. This is illustrated in Figure 7. Here, four objects $o_1, \ldots, o_4$ are bounded with *old TPBR*. The ChooseSubtree algorithm has to determine how the TPBR would look if a new object were added. The figure shows that the TPBR computed by ChooseSubtree based on *old TBPR* and *new object* is different from the TPBR that would be obtained if the new entry were actually added, i.e.,
$TPBR(TPBR(o_1, o_2, o_3, o_4), new\ object) \neq TPBR(o_1, o_2, o_3, o_4, new\ object)$.

This non-associativity means that insertion heuristics work with inaccurate information. Thus, even if an optimal TPBR means the best TPBR for a given set of entries, the same TPBR, used for insertion

decisions may in the long term lead to worse groupings of entries than when using other types of bounding rectangles. That is why our performance experiments consider the range of possible types of bounding rectangles described in Section 4.1.

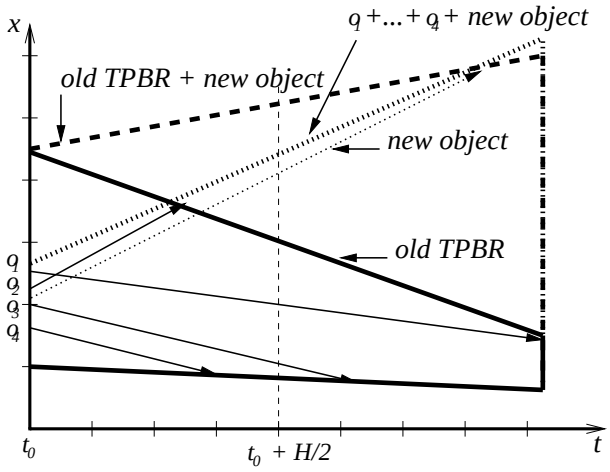

Figure 7: Non-Associativity of the TPBR Computation

Intuitively the inaccuracy introduced by non-associativity of the TPBR computation is greater when objects with significantly different velocities are grouped together. For example, in Figure 7, object o_2 is the main cause of the difference between the differently computed TPBRs. The grouping of entries with different velocities also leads to update-minimum bounding rectangles that deteriorate, as shown in Figure 4. In addition, a node of entries with significantly different velocities is more difficult to split.

A simple way to avoid grouping entries with very different velocities together is to consider all entries (representing both objects and bounding rectangles) as being infinite when making insertion decisions. Performance experiments in Section 5 investigates the effect of this on query performance.

Not considering the differences of the types of TPBRs used, the ChooseSubtree, Split, and RemoveTop[1] algorithms of the R$^{\mathrm{EXP}}$-tree are the same as those of the TPR-tree. The only difference is that the ChooseSubtree does not use overlap enlargement as a heuristic [5]. This simplifies the algorithm, making it linear instead of quadratic. Our performance experiments with different types of TPBRs show that using overlap enlargement as heuristics in the ChooseSubtree of the R$^{\mathrm{EXP}}$-tree does not improve query performance.

4.2.3 Dynamic Maintenance of the Time Horizon

As discussed in Section 4.2.1, the decisions made in insertion algorithms depend on the time horizon *H*, which, in turn, depends on the average update interval length (UI) and the average querying window length (*W*). To obtain a versatile and robust index, the values of these parameters should be maintained automatically by tracking the operations on the index.

To maintain an approximate value of UI, the R$^{\mathrm{EXP}}$-tree tracks the current number of entries (N) in the leaf level of the tree. It is increased every time a new leaf entry is inserted and decreased when a leaf-entry is deleted or discarded due to expiration. In addition, every n insertions, a special timer is reset to measure the time duration (Δt) it took to receive the last n insertions. Here, n is the number of entries in a node. Parameter UI is also updated every n insertions. It is set to be $(\Delta t/n)N$. One could also choose other periodic policies for updating UI, but, at least initially, it should not be done less frequently than every n insertions, so that, when *H* is used for the first time (in the first split of the root), the initial estimate of UI is available.

The above UI is used for making decisions in the insertion algorithm. When computing optimal and near-optimal TPBRs, a value that is smaller than the computed UI should be used because the TPBR can be recomputed any time one of the enclosed objects is updated. Thus, in addition to $N = N_0$, the number of leaf-entries, the R$^{\mathrm{EXP}}$-tree maintains the number of entries at each level of the tree (N_i). Then, UI_i, the average time between two recomputations of a bounding rectangle that bounds a node of level $i-1$, is equal

[1] RemoveTop is used in connection with forced reinsert [5].

to $UI_{i-1}/(N_{i-1}/N_i)$, where N_{i-1}/N_i is the average number of entries in a node at level $i-1$.

To compute the *H* parameter, the query window length *W* is needed. One could track queries to estimate *W*, but this approach has the disadvantage that an index may not be queried for long periods of time, leaving *W* outdated. We employ a different approach. In realistic scenarios, *W* is intimately related to UI. It makes little sense to ask queries that reach much further into the future than UI because the results of such queries will most probably be grossly invalidated by future updates. Thus, we choose $W = \alpha \cdot UI$, where $0 < \alpha < 1$ in most realistic scenarios.

4.3 Removal of Expired Entries

To contend with expiring entries in both the leaf level and in internal nodes of a tree, the insertion and deletion algorithms must be adjusted further.

First, the expired entries should be discarded from the tree at one time or the another. Second, a node may be noticed to be underfull, counting only non-expired entries, termed *live*, not only after removing an entry from a node during a deletion operation, but at any stage in both deletion and insertion algorithms.

To address this, a range of strategies can be adopted, ranging from very eager strategies, where expired entries are deleted by scheduled deletions as soon as they expire, to lazy strategies, where expired entries are allowed to stay in the index. We adopt a lazy strategy for the removal of expired index entries. Only live entries are considered during search, insertion, and deletion operations, but expired entries are physically removed from a node only when the contents of the node is modified and the node is written to disk. In addition, when an expired entry in an internal node is discarded, either when writing the node to the disk or deallocating it, the whole subtree rooted at this entry has to be deallocated.

To handle consistently the events of nodes becoming underfull (and overfull), the algorithms for insertion or deletion are modified to become very similar. First, as in the regular R*-tree, the leaf node is found where a new entry has to be inserted or the existing one deleted. From here, both algorithms proceed in the same way by calling the function $\mathbf{CorrectTree}(leaf)$, below, for the leaf node that was changed.

$\mathbf{CorrectTree}(leaf)$:

CT1 Initialize a list of orphaned entries, *orphans*, to be empty. The level of the tree from which the entry was removed is recorded with each entry in *orphans*.

CT2 Call $orphans = \mathbf{PropogateUp}(leaf, orphans)$.

CT3 While *orphans* is not empty

- CT3.1 Remove an entry with the highest level from *orphans* and insert it into a node at the appropriate tree level (in the same way as a data entry is inserted at leaf level), or if the root node of the tree is empty, insert it into the root node. Let *node* be the node where the entry was inserted.
- CT3.2 Call $orphans = \mathbf{PropogateUp}(node, orphans)$.

CT4 If the root node was modified and has only one entry, reduce the number of tree levels by declaring the child of it a new root.

The exotic case of the root becoming empty in CT3.1 may occur if all but one entry in the root expire and the single live entry is removed from the root by function $\mathbf{PropogateUp}$. This function checks for both the node being underfull or overfull (counting only live entries) and propagates the necessary changes up the tree.

PropogateUp$(node, orphans)$:

PU1 If *node* is overfull, then, as in R*-tree, either move a number of its live entries to *orphans* for later reinsertion (if that was not yet performed at this level), or split *node*.

PU2 If *node* is underfull, then move all its live entries to *orphans* and deallocate the node.

PU3 Remove the entry from *node*'s parent, *parent*, if the node was deallocated; install a new entry in *parent*, if the node was split (a new root is created if the root was split). Otherwise, update the bounding rectangle in the parent's entry that points to the node, if necessary.

PU4 If *node* is not the root, call $orphans = \mathbf{PropogateUp}(parent, orphans)$. Return *orphans*.

The presented algorithm ensures that all nodes modified by the algorithm have the right number of live entries. All other nodes, even if read by the algorithm, may be underfull.

It should be noted that the deletion algorithm in the R^{EXP}-tree uses a regular search procedure to find a leaf entry to be deleted. This procedure does not "see" expired entries. Consequently, if a delete operation is performed on an expired entry, the operation fails. This could be changed to allow the deletion algorithm to see the expired entries, but, as performance experiments show, this is unnecessary. The lazy strategy of purging expired entries as described above is able to maintain a very low percentage of expired entries in the index.

Figure 8 illustrates the workings of the algorithm. Here, an insertion of a new entry purges expired entries in part of the tree, and shrinks the tree in the process. The example assumes a maximum of 5 and a minimum of 3 entries in a node. In the first step, the entry X/20 is directed to leaf node C. As G and H in C have expired (the current time is 5), C is underfull and is discarded, while its live entries are temporarily stored in $orphans$. After removing C's entry from B, we notice that B is underfull. Again, B is discarded,

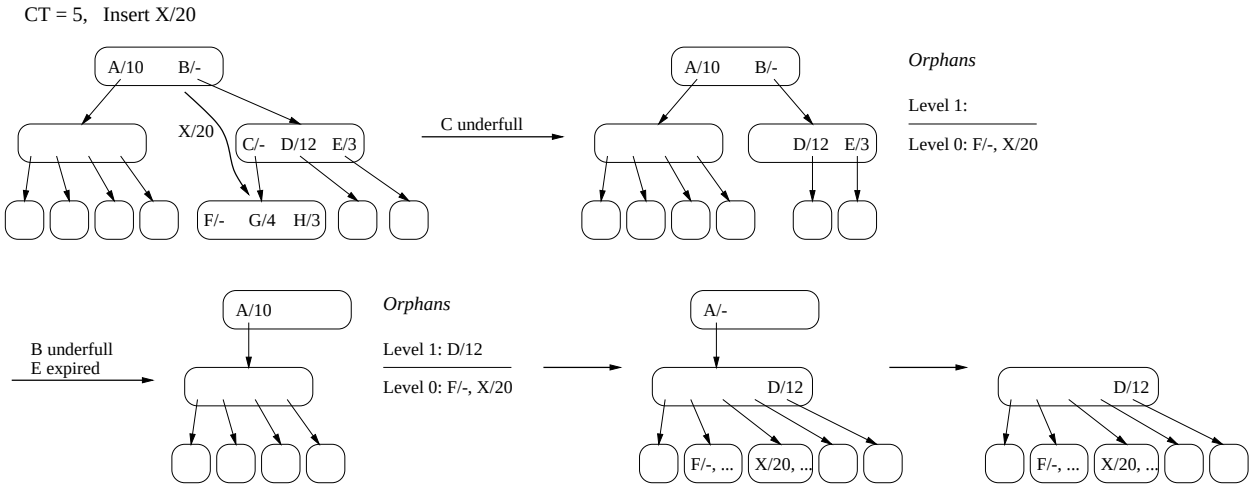

Figure 8: Purging of Expired Entries Triggered by an Insertion (Expiration Times Are Shown After Slashes)

and its live entries are posted to $orphans$, now at the list of level 1. In addition, when discarding B's expired entry E, we take care to deallocate the whole subtree rooted at E, which in this case happens to be a single leaf node. It is also worth noting that after this step, if A had expired, the algorithm would run into the situation of an empty root (CT3.1) when a new root is created from entries in $orphans$. In the last two steps, entries from $orphans$ are inserted one by one, starting with the higher-level entries. Finally, the tree is shrunk by discarding the single entry root.

Except for always checking for underfull or overfull nodes, the presented algorithm does not differ substantially from the R*-tree insertion and deletion algorithms. It should be noted, though, that in the new algorithm, the number of entries in the list *orphans* in the worst case is bounded only by the number of

entries in the whole tree, meaning that the list may not fit in main memory. For example, this could happen after a long period during which the system, for some reason, did not receive any updates. A natural solution to this problem is to fix the maximum size of *orphans* and stop handling underfull nodes in step PU2 when *orphans* is almost full. Limiting the size of *orphans* also limits the cost of any single update operation.

5 Performance Experiments

This section reports on performance experiments with the R^{EXP}-tree. The generation of two-dimensional moving point data and the settings for the experiments are described first, followed by the presentation of the results of the experiments.

5.1 Experimental Setup and Workload Generation

The R^{EXP}-tree was implemented in C++ using, as the basis, an adapted GiST [10] class library implementation. The page size (and tree node size) is set to 4k bytes, which results in 170 entries in a full leaf node and 102 entries in a full non-leaf node. A page buffer of 200k bytes, i.e., 50 pages, is used [15], where the root of a tree is pinned and the least-recently-used page replacement policy is employed. Nodes 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 artificially generated workloads that intermix queries and update operations, thus simulating index usage across a period of time. Initially, the index is empty and is populated gradually, with entries being added when simulated objects send their first positions. After the initial insertion of an object into the index, each subsequent update consists of the deletion of the old positional information, followed immediately by the insertion of the new. We proceed to describe how the updates and the queries are generated.

Because it is unrealistic to expect uniformly distributed positions and velocities for moving objects, 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 or intersections. Twenty destinations are distributed uniformly in the space with dimensions 1000×1000 kilometers. The destinations serve as the vertices in a fully connected graph of 380 one-way routes. When a new object is introduced, it is placed at a random position on a random route. The object is assigned with equal probability to one of three groups of objects 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 subsequent updates is approximately equal to a given parameter UI, which is fixed at 60 in most experiments.

In addition to using the above-described data, some experiments also use workloads with uniform data. In these workloads, the initial coordinates of a newly introduced object are chosen randomly. The directions of the velocity vectors are assigned randomly as well, 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 $2\,UI$.

Two different approaches are used for generating expiration times. In one set of the experiments, the expiration time of an object is set to be equal to the time of the update plus the *expiration period ExpT* that is common to all objects. Most experiments use $2\,UI$ for *ExpT*. In the other approach, expiration times are

dependent on the speeds of the objects. The positional information of fast objects becomes imprecise sooner than the positional information of slow objects. Consequently, fast objects should expire sooner. To achieve this, we introduce the concept of *expiration distance*. On an update, an object with speed v is assigned the expiration time $t_{upd} + ExpD/v$, where *ExpD* is varied in the experiments.

Depending on how expiration times are generated, a large portion of objects may expire and be removed from the index before they are updated. This means that the number of objects in the index is smaller than the number of objects participating in the simulated scenario. Based on the assumption that the time interval between successive updates is uniformly distributed between 0 and $2UI$, the workload generation algorithm, if necessary, increases the number of objects participating in the scenario so that the average number of leaf entries is around 100,000.

As mentioned in Section 2.1, there may be application scenarios where objects that stop reporting their positions are not guaranteed to notify the system and be deleted from the index. Our workload generation algorithm simulates this scenario by randomly "turning off" objects. To maintain a constant number of objects, a new object is introduced for each turned off object. A workload generation parameter *NewOb* specifies the fraction of objects initially participating in the simulated scenario that are replaced by new objects during the course of the workload.

All workloads contain one million insertion operations. In addition to insertions and deletions, workloads include one query for each 100 insertions (10,000 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 $W = UI/2$ and starting at the current time. Thus, the α parameter mentioned at the end of Section 4.2.3 is set to 0.5. Only for workloads with $ExpT = 30$, the setting of $W = 15$ was used. The spatial part of each query is a square occupying 0.25% of the space. 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 bold-face.

Parameter	Description	Values Used
ExpT	Expiration duration (time interval until expiration)	30, 60, **120**, 180, 240
ExpD	Expiration distance (distance traveled until expiration)	45, 90, 180, **270**, 360
NewOb	Fraction of new objects	0, **0.5**, 1, 1.5, 2
UI	Update interval length	30, **60**, 90, 120

Table 1: Workload Parameters

5.2 Experiments with Near-Optimal Time Parameterized Bounding Rectangles

As mentioned in Section 4.2.2, the ChooseSubtree algorithm may need slight modifications when combined with near-optimal TPBRs. In addition, not recording expiration times in TPBRs (as suggested in Section 4.1.1) may be beneficial. To investigate these issues, we compared four flavors of the $\mathrm{R^{EXP}}$-tree algorithms. Two use the regular ChooseSubtree algorithm, and the other two use the modified ChooseSubtree algorithm that assumes (only for decision-making purposes) the entry being inserted and the entries of the tree to have infinite expiration times. The latter should result in a more pronounced grouping of objects according to their velocities. For either choice of the ChooseSubtree algorithm, both options—recording TPBR expiration times and not recording them—are explored (expiration times for data entries are always recorded).

Figures 9 and 10 show the average numbers of I/O operations per query for workloads with varying *ExpT* and UI parameters. The graphs, as well as the results of a number of other experiments, demonstrate

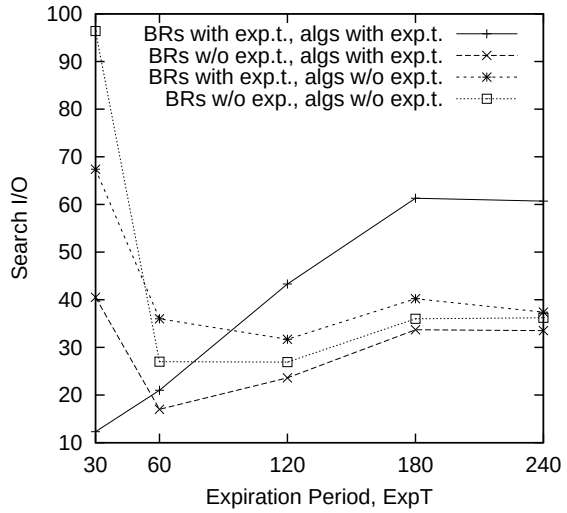

Figure 9: Search Performance For Varying *ExpT*

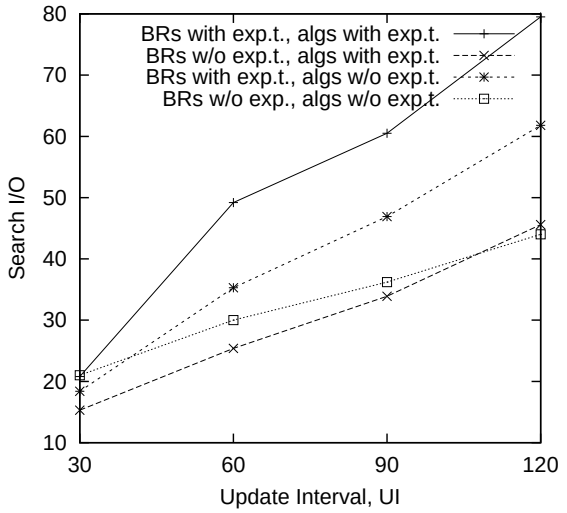

Figure 10: Search Performance For Varying *UI*

that, if TPBR expiration times are recorded, ChooseSubtree has to be modified to consider all entries as infinite (cf. the top two lines in Figure 10). Nevertheless, in most cases, the best results are produced when the normal ChooseSubtree algorithm is combined with TPBRs without expiration times. Other types of bounding rectangles also lead to better search performance when their expiration times are not recorded.

5.3 Comparing Different Time Parameterized Bounding Rectangles

A set of experiments with varying workloads was performed in order to compare the relative performance of near-optimal, optimal, static, and update-minimum bounding rectangles. For update-minimum bounding rectangles, both versions of the ChooseSubtree algorithm, discussed in the previous subsection, were explored. The version of the ChooseSubtree algorithm that ignores expiration times should avoid grouping regions in a way that causes update-minimum bounding rectangles to degrade later (illustrated in Figure 4).

Figure 11 shows the average number of I/O operations for uniform workloads, when *ExpT* is varied, and Figure 12 shows the result of experiments with workloads with speed-dependent expiration times.

The graphs demonstrate that, in most cases, near-optimal bounding rectangles perform best, and we use them in the experiments reported in the next subsection. Usage of the optimal bounding rectangles does not improve query performance. Most interestingly, update-minimum bounding rectangles are almost as good as near-optimal ones. Observe though that, in Figure 11, update-minimum TPBRs give better results when combined with the normal ChooseSubtree algorithm; and in Figure 12, update-minimum bounding rectangles are better when combined with the ChooseSubtree algorithm that ignores expiration times. This can be explained by observing that the situations similar to the one shown in Figure 4 are much more common in workloads where expiration times are dependent on the speeds. In these workloads, fast objects, such as $o1$ and $o3$ in Figure 4, expire sooner than slow objects.

For the same reasons, static TPBRs perform quite well for workloads with speed-dependent expiration times. Static TPBRs can effectively bound the trajectories of objects only if the trajectories with long expiration times have speeds close to zero, i.e., such trajectories are parallel to time axis (cf. Figure 3). This is exactly the case for workloads with speed-dependent expiration times.

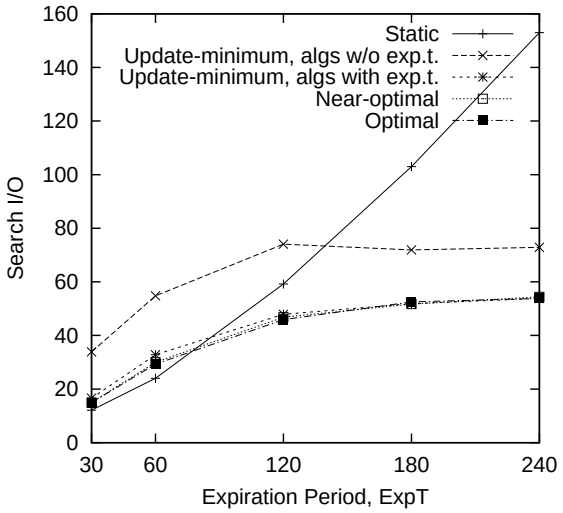

Figure 11: Search Performance for Uniform Data and Varying *ExpT*

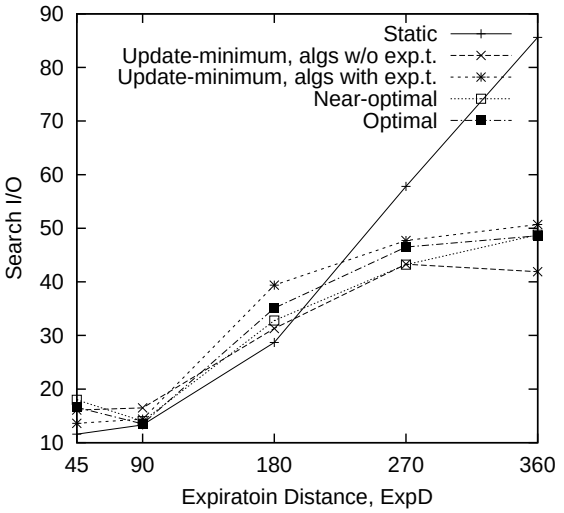

Figure 12: Search Performance for Varying *ExpD*

5.4 Comparing the R$^{\mathrm{EXP}}$-Tree with Alternative Approaches

To evaluate the R$^{\mathrm{EXP}}$-tree, we compared it with the TPR-tree and the TPR-tree with an associated B-tree for storing scheduled deletions as described in Section 3. To evaluate the difference between the lazy approach to purging expired entries and the approach where deletions are scheduled, we consider also the R$^{\mathrm{EXP}}$-tree with scheduled deletions.

Figures 13 and 14 show the search performance for workloads with varying *ExpD* and *NewOb*. The graphs demonstrate that if expiration durations are not too large, the R$^{\mathrm{EXP}}$-tree outperforms the TPR-tree by almost a factor of two even for workloads with no new objects being introduced. Naturally, when the number of "turned off" objects increases, the performance of the TPR-tree drops significantly because the size of the index increases (see Figure 15).

The performance of the R$^{\mathrm{EXP}}$-tree is only slightly worse than that of the approaches that employ scheduled deletions (which, as we shall see shortly, have excessive update costs). Most of the difference can be attributed to the better organized index that results from more updates when scheduled deletions are added to regular update operations. As Figure 15 shows, the difference in sizes between the R$^{\mathrm{EXP}}$-tree and the R$^{\mathrm{EXP}}$-tree with scheduled deletions is negligible. This means that at any moment, the frequency of updates is high enough for the algorithms described in Section 4.3 to remove most of the expired entries.

Figure 14 also shows that the difference between search performances of the TPR-tree with scheduled deletions and the R$^{\mathrm{EXP}}$-tree with scheduled deletions is very small. Note that the R$^{\mathrm{EXP}}$-tree is penalized in this setting by unnecessarily recording expiration times. This is illustrated by the size differences of the two indices (see Figure 15). Nevertheless, the above-mentioned small difference demonstrates that most of the performance gain of the R$^{\mathrm{EXP}}$-tree when compared to the TPR-tree is achieved by the regrouping of entries that occurs due to the lazy removal of expired entries.

Figure 16 shows that automatic removal of expired entries does not result in bad update performance. Here the average number of I/O operations per single insertion or deletion operation is shown. Note that the provided graphs do not include the costs associated with B-trees for the approaches with scheduled deletions. Adding these costs to the performance numbers of the TPR-tree with scheduled deletions would almost double them and, thus, make the update performance of this approach much worse than the update performance of the R$^{\mathrm{EXP}}$-tree.

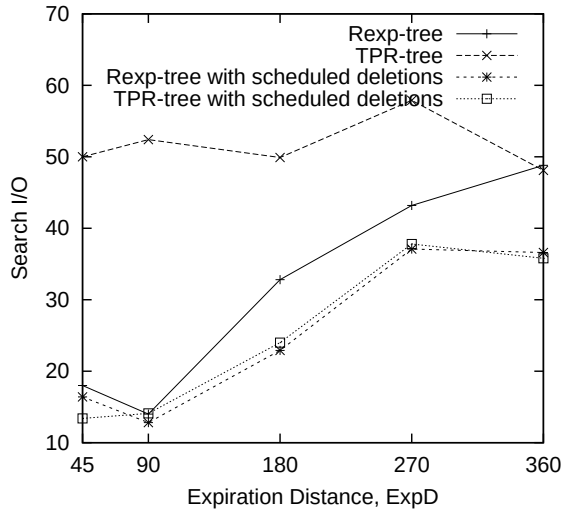

Figure 13: Search Performance For Varying *ExpD*

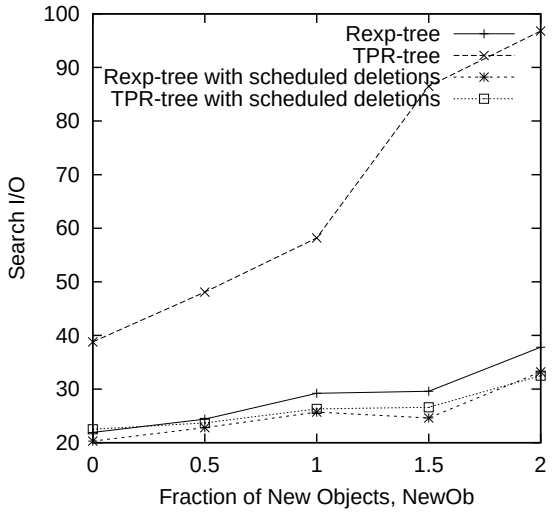

Figure 14: Search Performance for Varying Fraction of New Objects, *NewOb*

6 Conclusions and Future Work

Motivated by the emerging mobile Internet and location-based services, which may benefit from the ability to track large numbers of on-line mobile objects, this paper proposes an R*-tree-based index for the current and anticipated future positions of moving point objects.

The proposed R^{EXP}-tree captures the future trajectories of moving points as linear functions of time. To address the issue that, in many applications, the positional information is expected to be irrelevant and outdated soon after it is recorded, the R^{EXP}-tree stores expiration times in leaf entries of the index.

We provide insertion and deletion algorithms for the index that support expiration times. The algorithms implement a lazy technique for removing expired entries from the index. Performance experiments show that, for realistically dynamic index workloads, the algorithms are able to eliminate all but a very small fraction of the expired entries. By removing expired entries and, in the process, recomputing bounding rectangles and handling the resulting underfull nodes, the R^{EXP}-tree algorithms reorganize the index to improve query performance. In addition, the removal of expired entries does not result in high update costs.

The R^{EXP}-tree borrows the idea of time-parameterized bounding rectangles from the TPR-tree, but to take advantage of expiration times, we have investigated a number of different ways of computing such rectangles. Performance experiments show that choosing the right bounding rectangles and corresponding algorithms for grouping entries is not trivial and is dependent on the characteristics of the workloads. The so-called near-optimal time-parameterized bounding rectangles exhibited overall good query performance.

The long-term effect that different types of bounding shapes have on the grouping of finite line segments deserves a more detailed study (see Section 4.2.2). A possible approach would involve separating the information that guides the grouping decisions from the information that guides search. Such studies may be also useful in connection with the indexing of the histories of moving points.

References

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

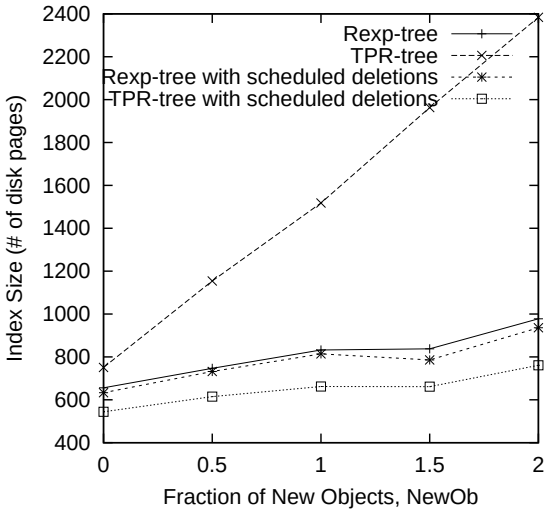

Figure 15: Index Size for Varying Fraction of New Objects, *NewOb*

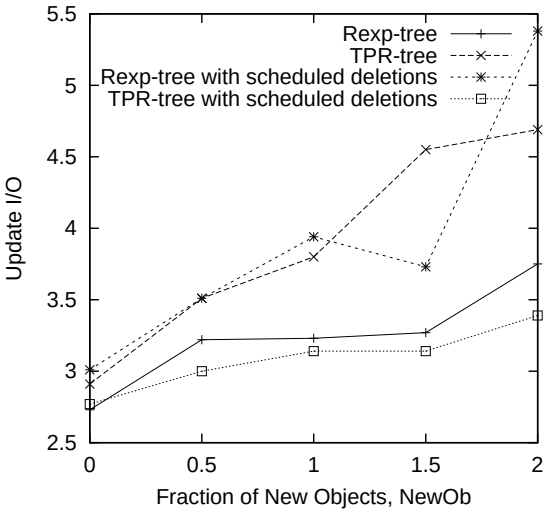

Figure 16: Update Performance for Varying Fraction of New Objects, *NewOb*

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

[4] J. Basch, L. Guibas, and J. Hershberger. Data Structures for Mobile Data. *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. *Proc. of the ACM SIGMOD Conf.*, pp. 322–331 (1990).

[6] M. Cai, D. Keshwani, and P. Z. Revesz. Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Databases. *Proc. of the EDBT Conf.*, pp. 430–444 (2000).

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

[8] J. Elliott. Text Messages Turn Towns into Giant Computer Game. *Sunday Times*, April 29, 2001.

[9] R. L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133 (1972).

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

[11] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. *Proc. of the VLDB Conf.*, pp. 500–509 (1994).

[12] P. G. Kirkpatrick and R. Seidel. The Ultimate Planar Convex Hull Algorithm? *SIAM Journal on Computing* 15(1): 287–299 (1986).

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

[14] G. Kollios et al. Indexing Animated Objects Using Spatiotemporal Access Methods. TimeCenter Tech. Rep. TR-54 (2001).

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

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

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

[18] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. Manuscript (2001).

[19] Ch. E. Ramirez. Games Find Home in Mobile Phones. *The Detroit News*, March 26, 2001.

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

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

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

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

[24] 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).