# Indexing of Moving Objects for Location-Based Services

Simonas Šaltenis Christian S. Jensen
Department of Computer Science, Aalborg University
{simas,csj}@cs.auc.dk

## Abstract

*Visionaries predict that the Internet will soon extend to billions of wireless devices, or objects, a substantial fraction of which will offer their changing positions to location-based services. This paper assumes an 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. Due to the possibility of many "expiring" objects, a highly dynamic database results. The paper presents an R-tree based technique for the indexing of the current positions of such objects. Different types of bounding regions are studied, and new algorithms are provided for maintaining the tree structure. Performance experiments indicate that, when compared to the approach where the objects are not assumed to expire, the new indexing technique can improve search performance by a factor of two or more without sacrificing update performance.*

## 1 Introduction

We are currently experiencing rapid technological developments that promise widespread use of on-line mobile personal information appliances. Industry analysts uniformly predict that mobile Internet terminals will significantly outnumber the desktop computers on 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. It is essential for many services, termed location-based services, that they be sensitive to the users' changing locations. Location awareness is made possible by a combination of political developments, e.g., the de-scrambling of the GPS signals and the US E911 mandate, and the continued advances in positioning technologies. Examples of location-based services 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 applications such as these that warrant the study of the indexing of objects that move.

The continuous movement of such objects 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 [15]. 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. 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 the system can automatically remove "expired" information.

This paper proposes the $\mathrm{R}^{\mathrm{EXP}}$-tree, an $\mathrm{R}^*$-tree [4] based access method that 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 equip the $\mathrm{R}^{\mathrm{EXP}}$-tree with insertion and deletion algorithms that lazily remove expired information from the index during the regular index update operations. Finally, the paper summarizes the results of an experimental comparison with the closest competitor, 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 summarizes results of performance experiments, and Section 6 concludes 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 to have the reference position for all objects be associated with the 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 [9, 16]. For example, a mobile yellow pages service is likely to be much less sensitive to imprecise positions than a traffic monitoring system.

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 a 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 it an expiring object. If unknown, the expiration time can be set to infinity, although in most cases, it 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. Finally, trivial upper bounds on the expiration times can be derived from the finite extents of the space in which 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.

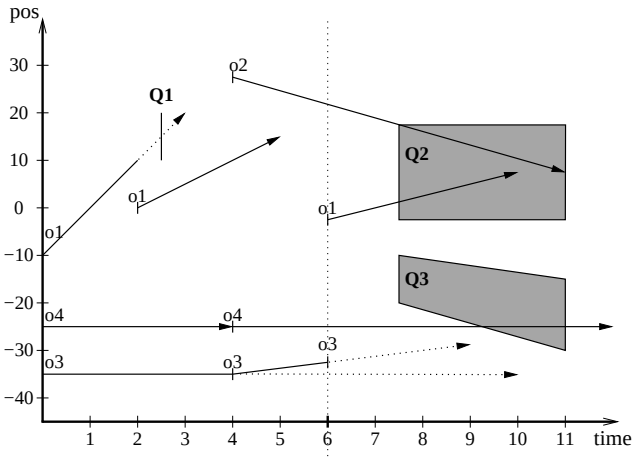


**Figure 1. One-Dimensional Data and Queries**

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.

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 among 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 $R_1$, and $R_2$ be two $d$-dimensional rectangles and $t^{\vdash}$ and $t^{\dashv}$ ($t^{\vdash}<t^{\dashv}$) be times that do not precede the current time. A *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 $t^{\dashv}$ (see $Q3$ in Figure 1). A *window* query ($Q2$, in Figure 1) is a special case of the moving query when $R_1 = R_2$. Finally, a *timeslice* query ($Q1$) is a special case of a window query when $t^{\vdash} = t^{\dashv}$.

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 or they expire. For example, if issued before time 2, query $Q1$ would return $o1$; and no object would have qualified if it were issued later because $o1$ was updated at time 2.

For these reasons, we expect queries to be 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, for most queries, $iss(Q) \leq t \leq iss(Q) + W$ and $iss(Q) \leq t^{\vdash} \leq t^{\dashv} \leq iss(Q) + W$, where $iss(Q)$ is the query issue time.

## 2.2 Previous Work

A number of approaches for indexing of the current and predicted future positions of moving points involve partitioning of the space into which the objects are embedded. Tayeb et al. [14] use PMR-Quadtrees [11], Kollios et al. [8] employ the so-called dual data transformation, and Agarwal et al. [1] use the ideas of so-called kinetic data structures [3]. While addressing similar problems, the approaches explored in these papers are not closely related to our work.

Since the technique proposed in this paper builds on the basic ideas of the TPR-tree, we review briefly the TPR-tree and 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 shows three one-dimensional moving points 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.

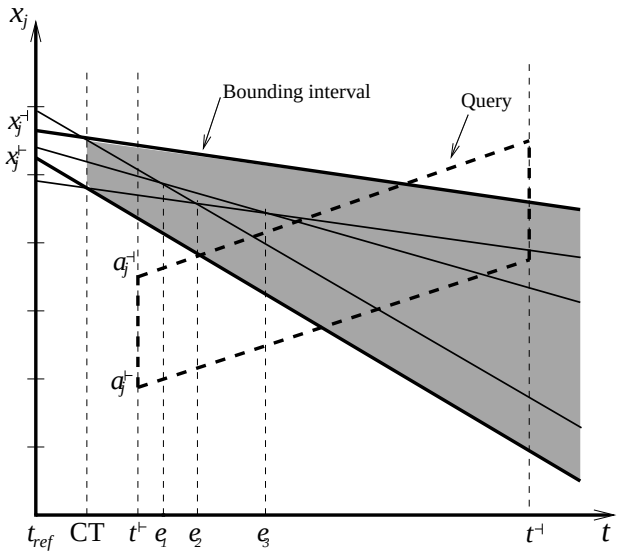


**Figure 2. A Bounding Interval and a Query in the TPR-tree**

In addition to the use of time parameterized bounding rectangles, the TPR-tree differs from the R*-tree in how its insertion algorithms group points into nodes. The R*-tree 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 illustrates a time integral of the length of the bounding interval. This use of integrals in the algorithms allows the index to systematically take the objects' velocities as well as their current positions into account when grouping them.

The bounding 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 achieve a trade-off between the number of events and the accuracy of the bounding interval. Based on these ideas, Pocopiuc et al. [10] propose the STAR-tree index for moving objects. This index seems to be most suited for workloads with quite infrequent updates.

Cai and Revesz have recently proposed a Parametric R-tree [5] 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.

# 3 Using the TPR-Tree for Expiring Objects

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 a 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, and the scheduled deletion event has to be inserted into the B-tree. Next, when processing the deletion event, the event has to be removed from the B-tree, and the deletion has to be performed in the TPR-tree. Performance experiments [13] indicate that this approach can be competitive with the 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 R$^{\mathrm{EXP}}$-tree. We first explore possibilities for computing time-parameterized bounding rectangles by maximally exploiting expiration times. Then 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 interval length with 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 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, as demonstrated in Figure 3.

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.

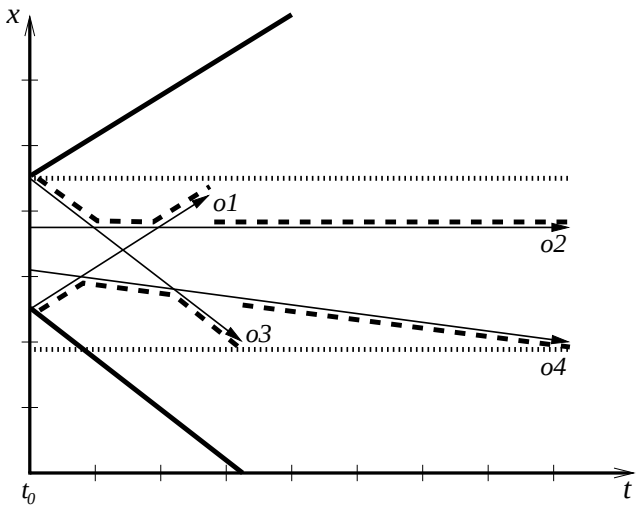


**Figure 3. Conservative (Bold), Always Minimum (Dashed), and Static (Dotted) Bounding Intervals**

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., $v_i^{\vdash} > v_i^{\dashv}$, 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 (see 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 the *conservative* bounding rectangles that are used in the TPR-tree. They 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 (upper) bound of the interval is set to move with the minimum (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 illustrates a conservative bounding interval, as used in the TPR-tree. 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 also 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.

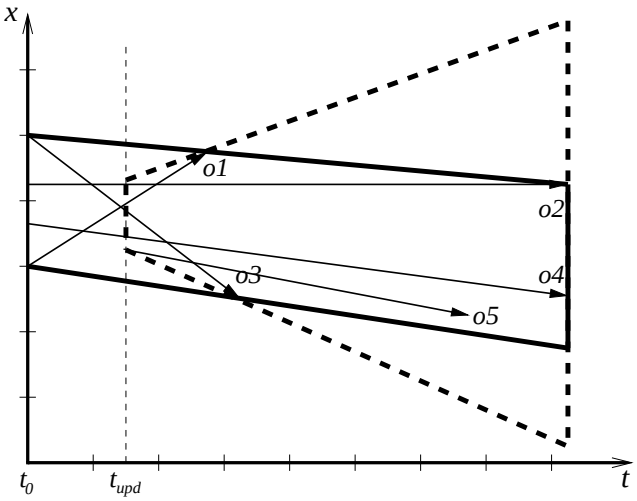


**Figure 4. Update-Minimum Interval (Bold) and the Same Interval Recomputed after Insertion of $o5$ (Dashed)**

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 affect 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 find a trapezoid with minimum area that is bounding and extends from $t = t_{upd}$ to $t = t_{upd} + h$. To find such a 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 $x^{\dashv}(t_{exp})$. Let the set $S$ include all trajectory endpoints. To capture the positions of points or intervals at $t_{upd}$, the minimum and maximum of these positions at $t_{upd}$ are also 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 [5], 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.

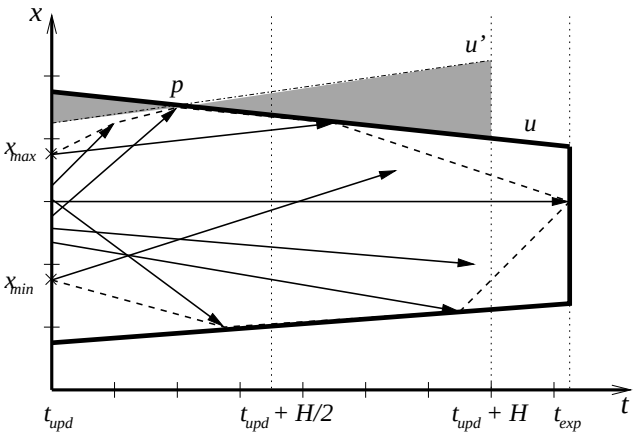


**Figure 5. A Convex Hull and an Optimal Bounding Interval**

It should be noted that if the median line contains one of the vertices of the convex hull, any line that crosses the convex hull only at this vertex can serve as the bound of the minimum trapezoid.

Any of a number of convex-hull computation algorithms (e.g., a Graham scan [6]) 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 [7] provide a linear algorithm to find such edges, which they call "bridges." Compared to the Graham scan, the algorithm is quite complex, and its implementation uses finite precision floating point arithmetics and 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 Rectangle (TPBR) in multiple dimensions is much harder. We desire a simple algorithm that produces "satisfactory" results. One approach is to compute the parameters of the bounding rectangle independently in each dimension [5]. For the $i$-th dimension, the bridge-finding algorithm could be applied to the projections of the trajectories into the $(x_i, t)$-plane.

It is 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

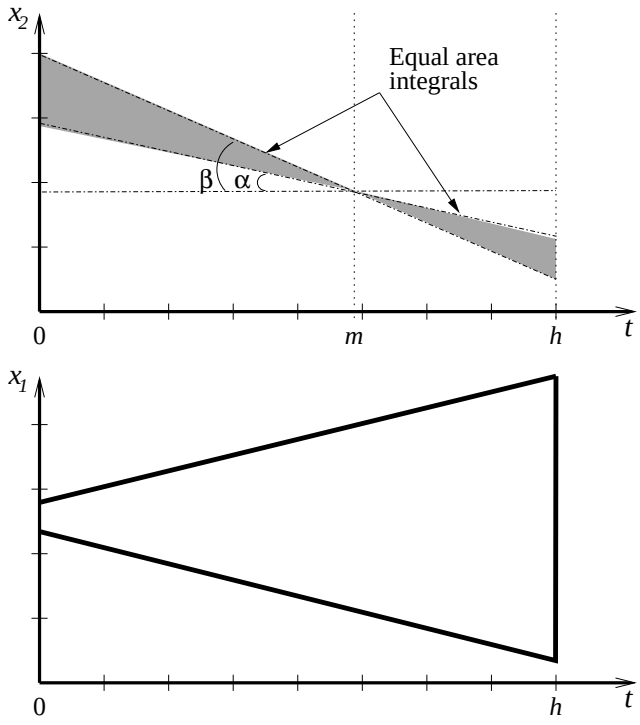


**Figure 6. Finding a Median Line for the Second Dimension (Top), When the First Dimension is Computed (Bottom)**

median line in the bridge-finding algorithm. Note that, to prove Lemma 4.1, we rely on the fact that the shaded triangle to the left of any point $p$ that is to the left of the median has a smaller area than the triangle to the right of $p$ (cf. Figure 5). If $p$ lies on the median line, both triangles have the same area. 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, we assume without loss of generality that $t_{upd} = 0$. Suppose $k$ dimensions are already computed and we want to find the median line for the computation of the $(k+1)$-st dimension. Let $a_i = x_i^{\dashv} - x_i^{\vdash}$ and $w_i = v_i^{\dashv} - v_i^{\vdash}$, $1 \leq i \leq k$, 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 of $t$ to the $i$-th power 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 the 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 $k = 1$, $m = h(3a_1 + 2w_1 h)/(6a_1 + 3w_1 h)$.

Using this lemma, our algorithm for computing a multi-dimensional TPBR visits dimensions one by one until the 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 a worst-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*.

For the sake of comparison, we also implemented an algorithm that computes true optimal multi-dimensional TPBRs. The worst-case running time of this algorithm is $O(|S|^{d-1}\log|S|)$, which is described elsewhere [13].

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

Any of these types of TPBRs can be used for answering queries in the same way that conservative TPBR's are used for query processing in the TPR-tree, only with expiration times taken into account.

## 4.2 Heuristics for Tree Organization

The heuristics that determine how to group moving objects and their TPBRs into nodes in the R$^{\mathrm{EXP}}$-tree are adopted from the TPR-tree.

The idea is to tune the index so that it efficiently supports queries when assuming a querying window length, *W*. In addition, *UI*—the average duration between the two successive updates of an object is an important problem parameter. Intuitively, the total duration of time when queries will

"see" the current insertion is $H = UI + W$. We term this the *time horizon*.

As in the TPR-tree, the R$^{\mathrm{EXP}}$-tree insertion algorithms have the structure of the R*-tree algorithms, but the objective functions $A_r(t)$ of area, margin, and overlap of bounding rectangles are replaced with their integrals over the time periods where queries are most likely to access the index.

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

The computation of such integrals is described in more detail elsewhere [12].

To avoid the user of the index having to set manually the parameters *UI* and *W*, we provide an automatic means of maintaining the values of these parameters, which is described elsewhere [13].

### 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 address two issues.

First, expired entries should be discarded from the tree at one time or 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 the deletion algorithm and the insertion algorithm.

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 have a very similar structure. 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 its child the 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.

$\mathbf{PropogateUp}(node, orphans)$:

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$^{\mathrm{EXP}}$-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 performance experiments show this to be 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 7 illustrates the workings of the algorithm (the expiration times of the entries are shown after the slashes). Here, an insertion of a new entry purges expired entries in part of the tree, shrinking the tree in the process. The example assumes a maximum of 5 and a minimum of 3 entries

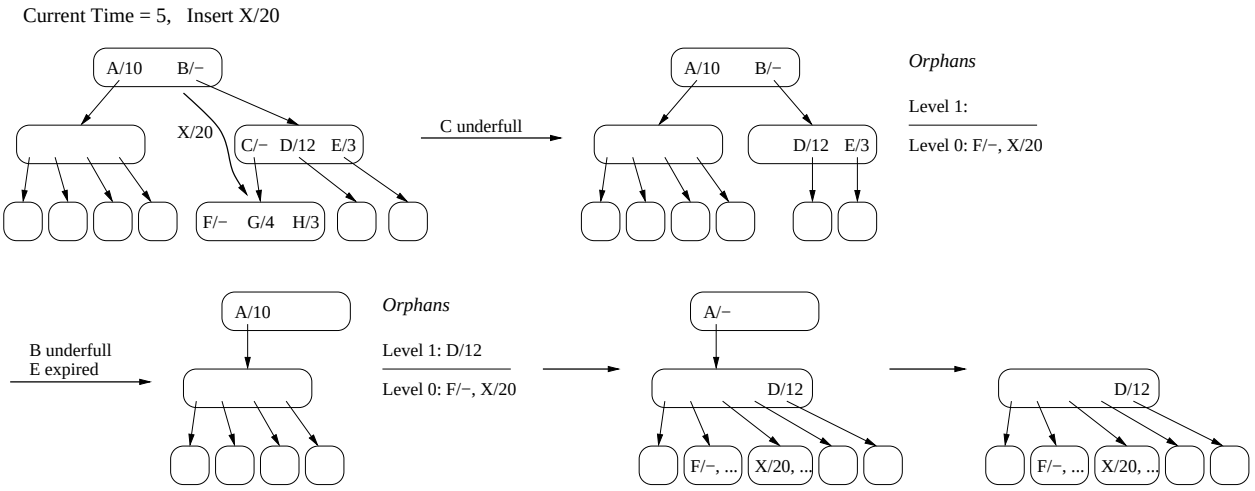


**Figure 7. Purging of Expired Entries Triggered by an Insertion**

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, and its live entries are posted to *orphans*, now in 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) where 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 worst-case cost of a single update operation.

## 5 Summary of Performance Experiments

This section summarizes the findings of the performance experiments that were performed with the $R^{EXP}$-tree. The detailed description of the experimental setup and the results of the experiments can be found in the longer version of this paper [13].

The performance studies are based on artificially generated index workloads that mix updates and queries. In most of the experiments, workloads are generated based on the simulation of objects moving on a two dimensional network of "roads." In some of the workloads, positions of objects expire after a predefined period of time after the last update—the *expiration duration*. In other workloads, an object's position expires when the object has traveled a predefined distance from its last update—the *expiration distance*.

We also experimented with workloads where simulated objects do not delete themselves, but stop reporting their positions. New objects are introduced to replace the "dead" ones.

Performance experiments with various types of time parameterized bounding rectangles show that, in most cases, near-optimal bounding rectangles without expiration times (expiration times are stored only with data entries) perform the best. Use of optimal bounding rectangles does not substantially change query performance. In most cases, optimal bounding rectangles do not differ, or differ only slightly, from their near-optimal counterparts. Most interestingly, update-minimum bounding rectangles are almost as good as near-optimal ones, although their influence on performance is more dependent on the characteristics of a workload. Static bounding rectangles, having the virtue of almost doubling the fan-out of non-leave nodes, perform satisfactorily for workloads with relatively short expiration durations.

Despite of only moderate gains achieved from introducing new kinds of bounding rectangles, the $R^{EXP}$-tree outperforms the TPR-tree by almost a factor of two, if expiration durations of objects are not exceedingly large. The

gains are increased when some objects become "dead" and new objects are introduced. In such situations, the size of the TPR-tree keeps increasing, degrading its performance.

The performance experiments show that most of the performance gain of the $R^{EXP}$-tree when compared to the TPR-tree is achieved by the regrouping of entries that occurs due to the lazy removal of expired entries. The performance studies also show that, in the simulated workloads that we used, the frequency of updates is high enough for the algorithms described in Section 4.3 to remove most of the expired entries.

Finally, the experiments also show that the lazy removal of expired entries does not degrade update performance. On contrary, due to an improved grouping of entries, the update performance appears to be better than the update performance of the TPR-tree.

## 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 not long 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, without sacrificing update performance.

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 long-term effect that different types of bounding shapes have on the grouping of finite line segments deserves a more detailed study [13]. The studies of bounding shapes may be also useful in connection with the indexing of the histories of moving points, represented as polylines.

## Acknowledgments

This research was supported in part by a grant from the Nykredit Corporation and by the Danish Technical Research Council, grant 9700780.

## References

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

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

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

[4] 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.

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

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

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

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

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

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

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

[12] 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.

[13] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. TIMECENTER Tech. Rep. TR-63, 2001.

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

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

[16] 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.