The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries

Yufei Tao[†]
[†]Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cityu.edu.hk

Dimitris Papadias[§]

Jimeng Sun[§]
[§]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{*dimitris, jimeng*}*@cs.ust.hk*

Abstract

A predictive spatio-temporal query retrieves the set of moving objects that will intersect a query window during a future time interval. Currently, the only access method for processing such queries in practice is the TPR-tree. In this paper we first perform an analysis to determine the factors that affect the performance of predictive queries and show that several of these factors are not considered by the TPR-tree, which uses the insertion/deletion algorithms of the R*-tree designed for static data. Motivated by this, we propose a new index structure called the TPR*-tree, which takes into account the unique features of dynamic objects through a set of improved construction algorithms. In addition, we provide cost models that determine the optimal performance achievable by any data-partition spatio-temporal access method. Using experimental comparison, we illustrate that the TPR*-tree is nearly-optimal and significantly outperforms the TPR-tree under all conditions.

1. Introduction

Spatio-temporal databases that manage large volumes of dynamic objects are becoming increasingly important due to numerous emerging applications (e.g., traffic control, meteorology monitoring, mobile computing, etc.). Such systems can be classified in two major categories depending on whether they deal with past information retrieval, or future prediction. In this work we focus on the second category, i.e., we assume that the database stores current data of moving objects and we wish to answer queries about the future. In particular, a *predictive window query* (window query, for short) specifies a query region q_R and a future time interval q_T, and retrieves the set of objects that will intersect q_R at any timestamp $t \in q_T$ (e.g., "find all the airplanes that will be over Hong Kong in the next 10 minutes"). If q_T contains a single (future) timestamp, then the query is called a *timestamp query*; otherwise, it is an *interval query*.

Instead of recording objects' locations at individual timestamps, spatio-temporal databases usually represent objects' movements as *motion functions*, so that updates are triggered by the changes of function parameters. The most common function corresponds to linear motion because it requires the minimum number of parameters and can be used to describe more complex movements (using interpolation). Accordingly, the record of an object o contains (i) its extent o_R at some reference time t_{ref} (a system parameter), and (ii) its current velocity o_V. Given this information, the object's location and extent at any future time t can be obtained as $o_R+(t-t_{ref})\cdot o_V$. In this case, an update is necessary only when the velocity changes.

- **Motivation**

With the exception of few structures (reviewed in Section 2) that are either purely theoretical, or applicable only to one-dimensional spaces, the Time Parameterized R-tree (TPR-tree) [SJLL00] is the sole practical spatio-temporal index for predictive queries. The TPR-tree uses the insertion/deletion algorithms of the R*-tree [BKSS90], which minimize certain *penalty metrics* to improve the quality of the resulting structure. Since the original metrics were designed for static objects, the TPR-tree replaces them with the corresponding *integral metrics*. Several problems, however, remain open. Since there is no analytical model for cost estimation, query optimization using the TPR-tree is currently impossible. Furthermore, there is no way to quantify the performance of the TPR-tree and any possible improvement.

- **Contribution**

This paper settles the above problems. Specifically:

- We derive *the first probabilistic model* that accurately estimates the number of disk accesses in answering a window query with a spatio-temporal index (including, but not limited to, the TPR-tree).
- We analyze, using this model, *the optimal*

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

performance of any data-partition index.

- We show that the current TPR-tree is *far from being optimal*, which implies that it may be significantly improved.
- We propose *the TPR*-tree, which integrates novel insertion and deletion algorithms* to enhance performance.
- We prove, through extensive experiments, that *the TPR*-tree is nearly optimal, and consistently outperforms the TPR-tree*.

The rest of the paper is organized as follows. Section 2 surveys previous spatial and spatio-temporal indexes, focusing on R*- and TPR-trees. Section 3 presents the analysis on the cost of predictive window queries, while Section 4 describes the concrete algorithms of the TPR*-tree. Section 5 contains an extensive experimental evaluation, and Section 6 concludes the paper with directions for future work.

2. Related Work

A number of structures [CR00, PJT00, KGT+01, TP01, HKTG02] have been proposed for historical spatio-temporal databases. These structures, however, are not suitable for future prediction as they are based on different principles (e.g., storage of discrete locations instead of motion functions) and have different goals (i.e., retrieval of information about the past instead of the future). Kollios et al. [KGT99] establish lower bounds for the cost of answering predictive window queries (using linear, or non-linear space) and design several nearly-optimal indexes for 1D objects. Agarwal et al. [AAE00] extend the solutions to two dimensions with the kinetic approach [BGH97]. Although the resulting methods have good asymptotical performance, they are not applicable in practice due to the large hidden constants. From the practical perspective, Tayeb et al. [TUW98] adapt the Quadtree [S90] for indexing the movements of 1D objects. Finally, Saltenis et al. [SJLL00] propose the TPR-tree, which adapts the R*-tree construction algorithms to moving objects. Section 2.1 reviews the R*-tree due to its influence in the development of the TPR-tree, and Section 2.2 describes the TPR-tree.

2.1 The R*-Tree

The R*-tree [BKSS90] can be regarded as an extension of the B-tree for multi-dimensional static objects. Figure 2.1 shows a two-dimensional example where 10 rectangles (a,b,…,j) are clustered according to their spatial proximity into 4 leaf nodes N_1,…,N_4, which are then recursively grouped into nodes N_5, N_6 that become the entries of the root. Each entry is represented as a minimum bounding rectangle (MBR). Specifically, the MBR of a leaf entry denotes the extent of an object, while the MBR of a non-leaf entry (e.g., N_1) tightly bounds all the MBRs (i.e., a,b,c) in its child node. The R*-tree is optimized for the window query, which retrieves all the objects that intersect a query region. In Figure 2.1, for example, the query visits the root of the R-tree, N_6, N_4, and returns object i.

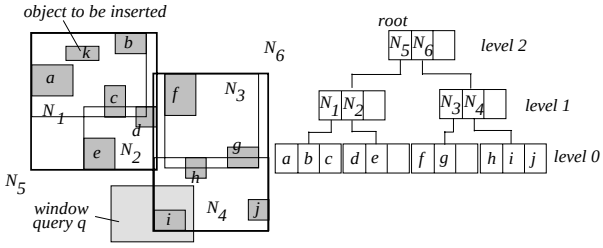


Figure 2.1: An R*-tree

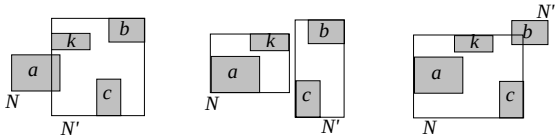
The R*-tree construction algorithm aims at minimizing the following *penalty metrics*: (i) the area, (ii) the perimeter of each MBR, (iii) the overlap between two MBRs (e.g., N_1,N_2) in the same node, and (iv) the distance between the centroid of an MBR (e.g., a in Figure 2.1) and that of the node (e.g., N_1) containing it. As discussed in [PSTW93], minimization of these metrics decreases the probability that an MBR intersects a query region.

Given a new entry, the insertion algorithm decides, at each level of the tree, the branch to follow in a greedy manner. Assume that we insert an object k into the tree in Figure 2.1. At the root level, the algorithm chooses the entry whose MBR needs the least area enlargement to cover k; N_5 is selected because its MBR does not need to be enlarged, while that of N_6 must be expanded considerably. Then, at the next level (i.e., child node of N_5), the algorithm chooses the entry whose MBR enlargement leads to the smallest overlap increase among the sibling entries in the node. Note that different metrics are considered at level 1 (leaf nodes are at level 0) and higher levels.

An *overflow* occurs if the leaf node reached (i.e., N_1 in the example) is full (i.e., it already contains the maximum number of entries). In this case the algorithm attempts to remove and re-insert a fraction of the entries in the node, trying to avoid a split if any entry could be assigned to other nodes. The set of entries to be re-inserted are those whose centroid distances are among the largest 30%. In Figure 2.1, b is selected since its centroid is the farthest from that of N_1 (compared to a,k,c).

Node splitting is performed if the overflow persists after the re-insertion (e.g., b is re-inserted back to N_1 in Figure 2.1, causing N_1 to overflow again). The R* split algorithm consists of two steps. The first step decides a split axis (from the x-, y-dimensions) as the one with the smallest *overall perimeter* computed as follows. On, for example, the x-axis, the algorithm sorts all the entries by the coordinates of their left boundaries (in Figure 2.1, the sorted order is a,k,c,b). Then, it considers every division of the sorted list that ensures that each node is at least 40% full. Figure 2.2 continues the example, which, for simplicity, omits this minimum node utilization constraint (we assume that a node can have a single entry, which corresponds to 33% utilization). The 1-3 division (Figure

2.2a), for instance, allocates the first entry (of the sorted list) into N, the other 3 entries into N'. The algorithm computes the perimeters of N and N', and performs the same computation for the other (2-2, 3-1) divisions. A second pass repeats this process with respect to the MBRs' right boundaries. Finally, the overall perimeter on the x-axis equals the sum of all the perimeters obtained from the two passes.



(a) 1-3 division (b) 2-2 division (c) 3-1 division

Figure 2.2: Possible divisions in splitting N_1 on the x-axis

After deciding the split axis (i.e., the one with the minimal overall perimeter), the split algorithm sorts the entries (according to their lower or upper boundaries) on the selected dimension, and again, examines all possible divisions. The final division is the one that has the minimum overlap between the MBRs of the resulting nodes. Continuing the previous example, assume that the split axis is x; then, among the possible divisions in Figure 2.2, the 2-2 incurs zero overlap (between N and N') and thus becomes the final splitting. Figure 2.3 demonstrates the R-tree after the insertion of k (observe the MBR changes and the new entry N_7 added to N_5).

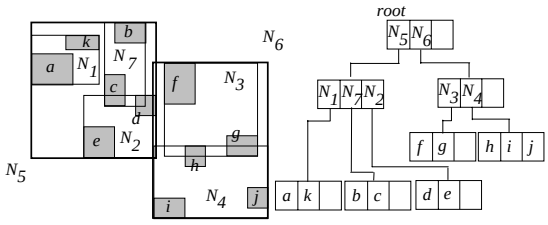
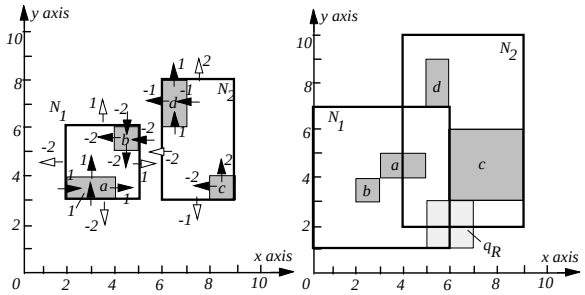


Figure 2.3: The R*-tree after inserting k

The deletion algorithm of the R*-tree is relatively simple. First, the leaf node that contains the entry to be removed is identified. If the node does not generate an underflow (i.e., it does not violate the minimum node utilization), the deletion terminates. Otherwise, the underflow is handled by simply re-inserting all the entries of the node, using the regular insertion algorithm. Both overflows and underflows may propagate to upper levels, which are handled in the same way.

2.2 The TPR-Tree

A moving object o is represented with (i) an MBR o_R that denotes its extent at reference time 0, and (ii) a velocity bounding rectangle (VBR) $o_V=\{o_{V1-},o_{V1+},o_{V2-},o_{V2+}\}$ where o_{Vi-} (o_{Vi+}) describes the velocity of the lower (upper) boundary of o_R along the i-th dimension ($1\leq i\leq 2$). Figure 2.4a shows the MBRs and VBRs of 4 objects a,b,c,d. The arrows (numbers) denote the directions (values) of their velocities, where a negative value implies that the velocity is towards the negative direction of an axis. The VBR of a is $a_V=\{1,1,1,1\}$ (the first two numbers are for the x-dimension), while those of b,c,d are $b_V=\{-2,-2,-2,-2\}$, $c_V=\{-2,0,0,2\}$, and $d_V=\{-1,-1,1,1\}$ respectively. A non-leaf entry is also represented with an MBR and a VBR. Specifically, the MBR (VBR) tightly bounds the MBRs (VBRs) of the entries in its child node. In Figure 2.4a, the objects are clustered into two leaf nodes N_1, N_2, whose VBRs are $N_{1V}=\{-2,1,-2,1\}$ and $N_{2V}=\{-2,0,-1,2\}$ (their directions are indicated using white arrows).



(a) MBRs & VBRs at time 0 (b) MBRs at time 1

Figure 2.4: Entry representations in a TPR-tree

Figure 2.4b shows the MBRs at timestamp 1 (notice that each edge moves according to its velocity). The MBR of a non-leaf entry always encloses those of the objects in its subtree, but it is not necessarily tight. For example, N_1 (N_2) at timestamp 1 is much larger than the tightest bounding rectangle for a,b (c,d). A predictive window query is answered in the same way as in the R*-tree, except that it is compared with the (dynamically computed) MBRs at the query time. For example, the query q_R at timestamp 1 in Figure 2.4b visits both N_1 and N_2 (although it does not intersect them at time 0).

The TPR-tree is optimized for timestamp queries in interval $[T_C, T_C+H]$, where T_C is the current update time, and H is a tree parameter called the *horizon* (i.e., how far the tree should "see" in the future). The update algorithms are exactly the same as those of the R*-tree, by simply replacing the four penalty metrics of the previous section with their *integral* counterparts. Specifically, the area (or perimeter) of an entry N equals $\int_{T_C}^{T_C+H} A(N,t)dt$ (or $\int_{T_C}^{T_C+H} P(N,t)dt$), where $A(N,t)$ (or $P(N,t)$) returns the area (perimeter) of N at time t. Similarly, the overlap (or the centroid distance) between two MBRs N_1 and N_2 is computed as $\int_{T_C}^{T_C+H} OVR(N_1,N_2,t)dt$ (or $\int_{T_C}^{T_C+H} CDist(N_1,N_2,t)dt$), where $OVR(N_1,N_2,t)$ (or $CDist(N_1,N_2,t)$) returns the overlapping area (centroid distance) between N_1 and N_2 at time t. These integrals are solved into closed formulae [SJLL00].

When an object is inserted or removed, the TPR-tree tightens the MBR of its parent node. Figure 2.5 shows the MBRs after inserting a new object e (into N_1) at time 1. N_1 is adjusted to the tightest MBR bounding a,b,e, by computing their respective extents at time 1. Note that this does not compromise the update cost because N_1 must be loaded (written back) from (to) the disk anyway to complete the insertion. On the other hand, the MBR of N_2 is not tightened because it is not affected by the insertion

(attempting to adjust N_2 will increase the update cost).

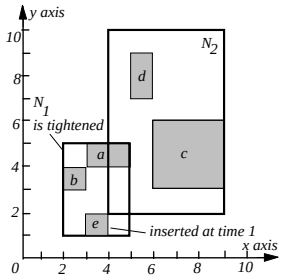


Figure 2.5: N_1 is tightened during an insertion at time 1

Saltenis et al. [SJLL00] analyze the optimal node extents that minimize the integral penalty, assuming, however, only bulk-loaded uniform data. Further, they do not discuss query performance. Recently, Saltenis and Jensen [SJ02] describe a method to decrease the query cost for a different problem, where the database is aware of the (future) time when each object will issue the next update. Their technique can also be applied to the TPR*-tree.

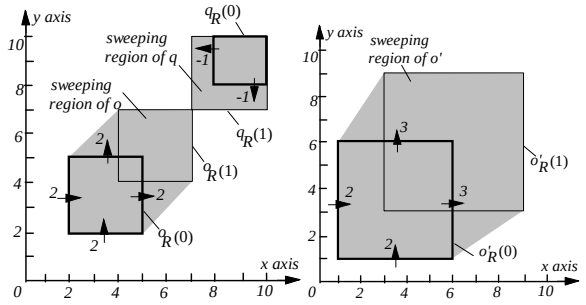
3. Performance Analysis

In Section 3.1 we derive a cost model that predicts the performance of the TPR-tree and reveals the factors that determine the query cost. The resulting equations are applicable to incrementally constructed or bulkloaded trees on uniform/non-uniform data. Based on the model we analyze the optimal query cost in Section 3.2. Without loss of generality, we consider the two-dimensional space, where the data universe has unit length on each axis. The discussion extends to arbitrary dimensionality and universe extent.

Given a moving rectangle o, we denote its MBR (VBR) as o_R=$\{o_{R1-},o_{R1+},o_{R2-},o_{R2+}\}$ (o_V=$\{o_{V1-},o_{V1+},o_{V2-}$, $o_{V2+}\}$), where o_{Ri-} (o_{Vi-}) is the coordinate (velocity) of the lower boundary of o on the i-th ($1\le i\le 2$) dimension. Similarly, o_{Ri+} (o_{Vi+}) refers to the upper boundary. Let o_{Ri}=$[o_{Ri-},o_{Ri+}]$ (o_{Vi}=$[o_{Vi-},o_{Vi+}]$) be the extent of o on the i-th spatial (velocity) axis, and $|o_{Ri}|$ ($|o_{Vi}|$) be the extent length, i.e., $|o_{Ri}|=o_{Ri+}-o_{Ri-}$ ($|o_{Vi}|=o_{Vi+}-o_{Vi-}$). The MBR of o at any future time t is represented as $o_R(t)=o_R+o_{V'}t$ (i.e., assume, without loss of generality, that the reference time for o_R is 0). The extent of $o_R(t)$ on the i-th spatial dimension is $[o_{Ri-}(t), o_{Ri+}(t)]$ and its length is $|o_{Ri}(t)|$.

3.1 A cost model for the TPR-Tree

Let o and q be two moving objects. The *transformed rectangle o'* of *o with respect to q*, has (i) MBR o'_R whose extent on the i-th axis is $\{o_{Ri-}-|q_{Ri}|/2, o_{Ri+}+|q_{Ri}|/2\}$, and (ii) VBR o'_V whose extent on the i-th axis is $\{o_{Vi-}-q_{Vi+}, o_{Vi+}-q_{Vi-}\}$. As an example, Figure 3.1a shows the MBRs (at time 0 and 1, respectively) and VBRs of two objects o, q. Figure 3.1b shows the transformed object o' of o with respect to q. Note that the MBR (VBR) of o' is enlarged from that of o, by the MBR (VBR) extent of q.



(a) Moving objects o,q (b) Transformed rectangle o'

Figure 3.1: Sweeping regions of moving rectangles

Given a moving object o and a time interval T, the *sweeping region* $SR(o,T)$ is the region swept by o during T. Denote $A_{SR}(o,T)$ as the area of $SR(o,T)$. The shaded areas in Figures 3.1a and 3.1b illustrate $SR(o,[0,1])$, $SR(q,[0,1])$, $SR(o',[0,1])$, with areas $A_{SR}(o,[0,1])=21$, $A_{SR}(q,[0,1])=9$, $A_{SR}(o',[0,1])=58$. Next we present a general cost model for spatio-temporal indexes adopting the MBR/VBR node representation used in TPR-trees.

Theorem 3.1 (Query cost model): Let q be a window query whose (i) MBR uniformly distributes in the data space, and has extent $|q_{Ri}|$ on the i-th dimension, (ii) velocity vector is q_V, and (iii) query interval q_T is $[q_{T-},q_{T+}]$. Then the average number of node accesses for answering q is:

$$Cost(q)=\sum_{\text{every node } o} A_{SR}(o',q_T) \qquad (3\text{-}1)$$

where o is the moving rectangle representation of a node, and o' is the transformed rectangle of o with respect to q.

Proof (sketch): A node o is visited if its MBR intersects the query MBR during q_T, which as shown in [TSP03], occurs if and only if $SR(o',q_T)$ covers the centroid of the query MBR (i.e., a static point). Given that, the query MBR distributes uniformly in the data space, the access probability equals the area $A_{SR}(o',q_T)$ of the sweeping region of o' during q_T. The summation of probabilities for all nodes gives the expected number of accesses. ■

It is important to note that, for all queries with the same parameters $|q_{Ri}|$, q_V and q_T, $A_{SR}(o',q_T)$ is the same for a specific node o. Furthermore, $A_{SR}(o',q_T)$ is different from the integral metric $\int_{T_C}^{T_C+H} A(o',t)dt$ (where $A(o',t)$ is the MBR area of o' at time t). Consider Figure 3.2, where the MBR $o_1'_R$ of transformed object o_1' has constant size during its movement in time [0,1], while o_2' is static and its MBR $o_2'_R$ has the same area as $o_1'_R$. Clearly, $A_{SR}(o_1',[0,1]) > A_{SR}(o_2',[0,1])$, meaning that o_1 has higher probability to be accessed. This cannot be captured by the integral metric since $\int_0^1 A(o_1'_R,t)dt = \int_0^1 A(o_2'_R,t)dt$.

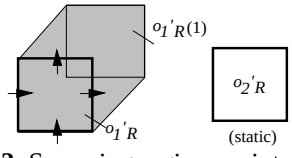


Figure 3.2: Sweeping region vs. integral metric

3.2 A hypothetical optimal tree

According to Theorem 3.1, a TPR-tree optimized for queries with *specific parameters* $|q_{Ri}|$, q_V, and q_T, should aim at minimizing equation 3-1. Let the *extended work space* (EWS) be the 4D space whose dimensions include those of the original 2D data and velocity spaces. Each (moving) rectangle o with MBR o_R and VBR o_V can be mapped to a 4D box in EWS, whose projection onto the spatial (velocity) dimensions corresponds to o_R (o_V). The *volume* of o is defined as the volume of its 4D box, which equals the product of the areas of o_R and o_V, or formally $Vol(o)=(\prod_{i=1\sim2}|o_{Ri}|)\cdot(\prod_{i=1\sim2}|o_{Vi}|)$. If o is a point object, then its EWS representation is a 4D point (i.e., a degenerated box) with zero volume. Note that the TPR-tree can be regarded as a 4D R-tree in EWS where the 4D box of a non-leaf entry encloses those of all the entries in its child node.

To simplify the discussion, we first consider a dataset that contains N moving points, whose MBRs and VBRs uniformly distribute in the data and velocity spaces, respectively. We consider a well-organized TPR-tree where the 4D boxes of the leaf nodes are (i) mutually disjoint, and (ii) their union covers the entire EWS. As a result, a new (moving) data point p is inserted to the leaf node whose 4D box covers that of p (conditions (i) and (ii) guarantee that such node is unique). This is the best choice because it incurs no increase in any sweeping region, whereas inserting p in any other node requires expanding its MBR or VBR leading to larger sweeping region. Since the data distribution is uniform, the probability that an object is inserted to a particular node o equals its volume $Vol(o)$ divided by the volume of EWS. Therefore, the leaf node with the largest volume receives the highest number of objects, and will generate the next overflow.

A node o that overflows is split into o_1 and o_2 in an overlap-free manner on a particular dimension of the EWS. Specifically, let SA be the split dimension and sp be the split position (on SA). Then, the extents of o_1 and o_2 can be decided from those of o as follows: (i) for any spatial (or velocity) dimension $i\neq SA$, the extents of o_1, o_2 equal those of o; (ii) on the split dimension $i=SA$, the extent of o is divided into o_1, o_2 at sp.

The value of sp must guarantee that both o_1 and o_2 satisfy the minimum node utilization. Specifically, let n be the number of entries in the original node o; then, after the split o_1 contains $n_1=n\cdot Vol(o_1)/Vol(o)$ entries, where $Vol(o)$ and $Vol(o_1)$ denote the volumes of o and o_1, respectively. Similarly, the number of entries in o_2 equals $n_2=n\cdot Vol(o_2)/Vol(o)$. Assuming ξ to be the minimum utilization threshold (40% in TPR-trees), n_1 and n_2 must satisfy $n_1\geq\xi\cdot n$, and $n_2\geq\xi\cdot n$. By solving these inequalities on dimension i, we obtain that a valid sp (on dimension i) must be in the following *valid range*: if i is a spatial dimension, then $sp\in[o_{Ri^-}+\xi\cdot|o_{Ri}|,\ o_{Ri^+}-\xi\cdot|o_{Ri}|]$; otherwise, $sp\in[o_{Vi^-}+\xi\cdot|o_{Vi}|,\ o_{Vi^+}-\xi\cdot|o_{Vi}|]$.

We consider that each split in the hypothetical tree is performed in an optimal manner, namely, it minimizes the increase ΔA_{SR} of equation 3-1, or specifically:

$$\Delta A_{SR}=A_{SR}(o_1',q_T)+A_{SR}(o_2',q_T)-A_{SR}(o',q_T) \quad (3\text{-}2)$$

where o', o_1', o_2' are the transformed rectangles of o, o_1, o_2 with respect to the target optimization query parameters. Figure 3.3 presents an algorithm that computes the extents of the leaf nodes in the hypothetical tree described above. The algorithm maintains a priority queue QN, which contains the extents of all the leaf nodes created so far. Initially, QN contains a single node, which corresponds to the root of an empty tree whose MBR and VBR cover the entire data and velocity spaces, respectively. The sorting keys of nodes in QN are the volumes of their 4D boxes in EWS. At each iteration, the node o_{head} with the largest volume is removed from QN and split into o_1 and o_2, at the best split axis SA_{best} and position sp_{best} that minimize ΔA_{SR} as in equation 3-2 (see Lemma A.1 in the appendix for deciding SA_{best} and sp_{best}). The extents of o_1 and o_2 are then computed from those of o according to SA_{best} and sp_{best}, after which they are inserted into QN. Note that after each iteration, the total number of nodes in QN increases by 1. The algorithm terminates after N_0-1 iterations, where N_0 is the total number of leaf nodes, computed as $\lceil N/f \rceil$, where N is the dataset cardinality and f the fanout (i.e., average number of entries in a node). The extents that remain in QN are the extents of the final leaf nodes. The algorithm in Figure 3.3 can also be used to estimate the extents of non-leaf nodes, by passing the total number N_i of nodes at level i. Given the cardinality N and fanout f, N_i equals $\lceil N/f^{i+1} \rceil$, for all $0 \leq i \leq h-1$, where $h = \lceil \log_f N \rceil$ is the height of the tree.

Algorithm Estimate Node Extents (N_i, q)
/* **Input**: N_i is the number of nodes desired; q specifies the target query optimization parameters. **Output**: QN contains the node extents when the algorithm terminates. */
1. initialize a priority queue QN;
2. insert o(o_R=data space, o_V=velocity space) into QN; cnt=1
3. while ($cnt<N_i$)
4. o_{head}=de-queue(QN) //o_{head} has the largest volume
5. $min\Delta A_{SR}=\infty$ //next, decide best split axis and position
6. for i=each spatial/velocity dimension
7. let sp_i be the best split position on dimension i //decided using Lemma A.1 (see appendix)
8. compute MBRs, VBRs of nodes o_1,o_2 by splitting o along i at position sp_i
9. $\Delta A_{SRi}= A_{SR}(o_1',q_T)+A_{SR}(o_2',q_T)-A_{SR}(o',q_T)$
10. if $\Delta A_{SRi}<min\Delta A_{SR}$
11. $min\Delta A_{SR}=\Delta A_{SR}$; $SA_{opt}=i$; $sp_{opt}=sp_i$
12 compute MBRs, VBRs nodes o_1,o_2 by splitting o along SA_{opt} at position sp_{opt}
13. en-queue(QN, o_1)
14. en-queue(QN, o_2)
15. $cnt=cnt+1$
End Estimate Node Extents

Figure 3.3: Algorithm for predicting node extents

Our algorithm can estimate the node extents of a tree for non-uniform data with the following modifications. First, the volume of a node is now defined as the number of objects whose 4D boxes (in EWS) are covered by that of the node. Second, the valid range $[v_-, v_+]$ for the split position is defined as follows. If node o is split into o_1, o_2 at v_-, then the volumes of o_1, o_2 equal $Vol(o)\cdot\xi$, and $Vol(o)\cdot(1-\xi)$, respectively, where ξ is the utilization threshold. Similarly, if the split position is at v_+, the volumes of o_1, o_2 equal $Vol(o)\cdot(1-\xi)$ and $Vol(o)\cdot\xi$. The volume of a node can be computed with spatio-temporal histograms [CC02, TSP03]. Having the node extents of all levels, the query cost of the tree is calculated using equation 3-1.

Notice that the hypothetical tree is constructed by making "locally optimal" decisions (which may not be "globally optimal"). Since, however, all dynamic indexes make only local decisions, *the cost of the hypothetical tree provides a practical lower bound for the performance of an actual index*. As shown in the experimental section, *the cost of the TPR-tree is significantly higher than this lower bound*, which motivates the development of the TPR*-tree.

4. The TPR*-Tree

The TPR*-tree improves the TPR-tree by employing a new set of insertion and deletion algorithms that aim at minimizing equation 3-1. However, this equation refers to a specific set of query parameters, while in practice different queries may have significant diversity. This raises the question about the choice of appropriate parameter values used for optimization. We optimize the TPR*-tree for the *static point interval query q*, whose (i) MBR has length $|q_{Ri}|=0$ on each axis, (ii) VBR=$\{0,0,0,0\}$, and (iii) query interval $q_T=\{0,H\}$, where H is the horizon parameter (also used in the original TPR-tree). As shown in the experiments, this choice leads to nearly-optimal performance independently of the query parameters. Section 4.1 first describes the insertion algorithm, and Section 4.2 discusses deletion.

4.1 Insertion

Figure 4.1 shows the high level description of the TPR* insertion. Specifically, given a new entry e at insertion time T_l, the TPR*-tree first identifies the leaf N that will accommodate e with the *choose path* algorithm. If N is full, a set of entries, selected by *pick worst*, are removed from N and re-inserted. Any leaf node that overflows during the re-insertion will be split using *node split*, after which a new entry will be added to the parent node. This may cause the parent to overflow, and is handled in a similar way. Next we elaborate *choose path*, *pick worst*, and *node split*, and explain why the corresponding algorithms in the TPR-tree are not efficient.

Algorithm Insert (e)
/* **Input**: e is the entry to be inserted. */
1. $re\text{-}inserted_i$=false for all levels $1 \le i \le h-1$ (h is the tree height)
2. initialize an empty re-insertion list $L_{reinsert}$
3. invoke **Choose Path** to find the leaf node N to insert e
4. Invoke **Node Insert**(N, e)
5. for each entry e' in the $L_{reinsert}$
6. invoke **Choose Path** to find the leaf node N to insert e'
7. Invoke **Node Insert**(N, e)

End Insert
Algorithm Node Insert (N, e)
/* **Input**: N is the node where entry e is inserted */
1. if N is a leaf node
2. enter the information of e
3. if N overflows
4. if $re\text{-}inserted_0$=false //no re-insertion at leaf level yet
5. invoke **Pick Worst** to select a set S_{worst} of entries
6. remove entries in S_{worst} from N; add them to $L_{reinsert}$
7. $re\text{-}inserted_0$=true
8. else
9. invoke **Node Split** to split N into itself and N'
10. let P be the parent of N
11. **Node Insert**(P,$\varnothing$) or **Node Insert**(P,N') if N has been split
12. else //N is a non-leaf node
13. similar to lines 2-9 except that (i) the MBR/VBR of the affected child node is adjusted, and (ii) in lines 4, 7 replace $re\text{-}inserted_0$ with $re\text{-}inserted_i$ where i is the level of N

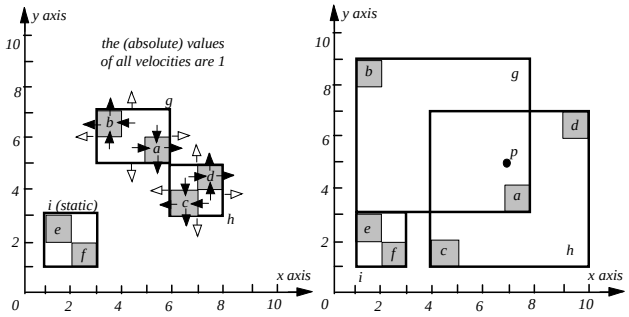
End Node Insert

Figure 4.1: Overview of the TPR* insertion algorithm

- **Choose Path**

Given a new object, the traditional TPR-tree selects, at each non-leaf level, the branch with the smallest deterioration (in terms of certain penalty metrics) to continue the insertion. The efficiency of this "greedy" approach drops considerably, if multiple branches have the same (zero) deterioration. To illustrate this, we use Figure 4.2a that shows 6 leaf nodes $a,b,\ldots,f$ with their parent nodes g,h,i that are the entries of the root (the absolute values of all velocities are 1). Note that although the MBRs of g,h are disjoint at time 0, they overlap significantly at timestamp 2 (Figure 4.2b).

Consider the insertion of (static) point p at time 2. At the root level, g and h have no deterioration because inserting p into either one does not expand the corresponding MBR/VBR. In this case the algorithm must rely on the "tie-breaking" conditions which, however, are much less effective. In the example, h is preferred because it has smaller MBR, inside which the best leaf node to include p is d. The best choice, however, is to insert p to node a, as it requires significantly smaller MBR expansion than d. Note that this problem becomes even more serious as time progresses and the overlaps between MBRs become increasingly larger. Eventually, the greedy algorithm becomes almost random, i.e., it just picks one of the numerous candidate branches with zero penalty. The problem is less serious in R-trees (i.e., static data) where the MBRs do not grow with time.



(a) MBRs & VBRs at time 0 (b) MBRs at time 2

Figure 4.2: Inserting *p* at time 2

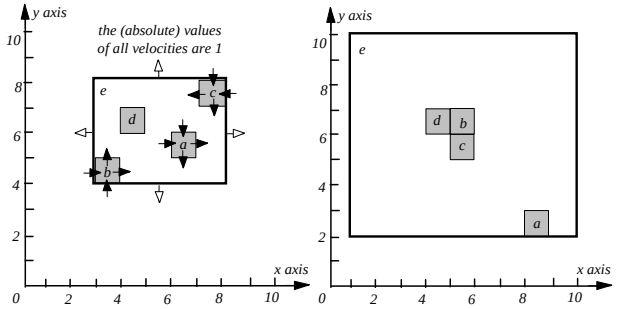
Motivated by this, we propose a *choose path* algorithm which, given a new object, returns the insertion path with the minimal increase in equation 3-1 (called *cost degradation* in the sequel) *among all the paths*. Towards this, *choose path* maintains a priority queue *QP* that records the candidate paths inspected so far. In Figure 4.2b, at the root *QP* is initiated with {[(*g*),0], [(*h*),0], [(*i*),20]}, where each number indicates the cost degradation (for the static point interval query with q_T=[0,1]), if *p* is inserted into the corresponding path. The degradation is 0 for *g* and *h* because, as mentioned earlier, their MBRs/VBRs do not need to be expanded. Note that at this point we have not accessed any of nodes *g*,*h*,*i*, i.e., the cost degradation is computed from their extents stored in the root. At each step, *choose path* explores the path with the smallest cost degradation. In this example, it visits node *g* and inserts two paths (*a*,*g*) and (*b*,*g*) in *QP*, after which *QP* = {[(*h*),0], [(*a*,*g*),3], [(*i*),20], [(*b*,*g*),32]}. Notice that (*a*,*g*) and (*b*,*g*) are *complete*, meaning that they include the leaf level (although leaves *a* and *g* are not visited). Similarly, the next path expanded is (*h*), and *QP* becomes *QP*={[(*a*,*g*),3], [(*d*,*h*),9], [(*c*,*h*),17], [(*i*),20], [(*b*,*g*),32]}. Now the algorithm terminates with (*a*,*g*) as the overall best path, because its (accumulated) cost degradation is smaller than that of all the other entries in *QP*. Note that [(*i*),20] is not explored at all, as it already incurs larger degradation at the highest level.

C*hoose path* finds the best insertion path at the cost of some extra node accesses. This, however, pays off due to the following reasons. First, it leads to a better tree structure, which improves the query performance. Second, our experiments show that in most cases it only needs to explore on average 2-3 complete paths because most paths will terminate at very high levels (e.g., (*i*) in Figure 4.2b). Third, *choose path* only visits non-leaf nodes that usually reside in the buffer. Fourth, each update in spatio-temporal databases usually involves one deletion (followed by an insertion), which as explained in the next section, is usually the dominating factor in the total update cost. The deletion requires a query to locate the object to be removed. Due to its improved query performance with respect to the TPR-tree, the update overhead of the TPR*-tree is much lower. A similar situation exists for the relative performance of updates in R*- and R-trees; although the R*-tree involves more complex insertion operations, it results in faster updates due to its better structure.

- **Pick Worst**

Insertion to a full node generates an overflow, in which case both TPR- and TPR*-trees re-insert a fraction of the entries from the node. The TPR-tree, following the strategy of R*-trees, selects the entries by evaluating the distances between the centroids of their MBRs and that of the node. We observe that this usually does not lead to decrease in the MBR/VBR of the overflowing node, and hence limits the effectiveness of re-insertion. Figure 4.3a shows an example where leaf node *e* contains objects *a*,*b*,*c*,*d* (all velocities have absolute values 1), and Figure 4.3b illustrates their MBRs at time 2. Assume that node *e* generates an overflow at time 0 and one entry must be re-inserted. Notice that *b*, *c*, *d* move towards the centroid of *e* (whose coordinates are (5.5,6) during [0,2]), while *a* moves away and is selected for re-insertion (its centroid has the farthest integrated distance, introduced in Section 2.2, from that of *e* during [0, 2]).

The removal of *a*, however, does not affect the extents of *e*, whose MBR and VBR are actually decided by *b*, *c* (see Figure 4.3a). This means that *a* has a high chance to be re-inserted back into *e* again (especially if our *choose path* is applied), because this does not lead to extent expansion (and performance degradation). So the re-insertion becomes useless and a node split must occur. In general, entries selected in this manner are usually those that move away most quickly from the centroid of the bounding MBR, instead of those that decide the extents. Again, this problem is not important for conventional R-trees. For instance, in Figure 4.3a, if the MBRs were in an R-tree, object *c* would be removed, resulting in smaller MBR for *e*.



(a) MBRs & VBRs at time 0 (b) MBRs at time 2

Figure 4.3: Selecting an entry for re-insertion at time 0

To overcome this problem, *pick worst* returns a set of entries whose removal reduces the MBR or VBR of the parent node. Towards this, it targets directly entries that determine the MBR/VBR of their parent. Specifically, on a selected spatial/velocity dimension *i*, the algorithm sorts the entries by the starting points of their extents on dimension *i*. In Figure 4.3a, for instance, the sorting list is {*b*,*d*,*a*,*c*} if *i* is the *x*-spatial dimension, or {*c*,*d*,*a*,*b*} if *i* is

the x-velocity dimension. Obviously, removing the first λ (=30% in our implementation) entries in the list will guarantee smaller parent node extent on dimension i. The same holds for sorting with respect to the ending points of entries' extents.

It remains to decide (i) the sorting dimension, and (ii) whether sorting should be performed on the starting or ending points (of the extents). One easy way to achieve this is to actually sort on every dimension/direction, which, however, requires sorting all the entries $4 \cdot d$ times (while the original TPR-tree requires only one sorting), where d is the dimensionality of the data space. Instead, we make decisions (i) and (ii) by *estimating* the decrease of the sweeping region area of the parent node for each possible combination. Consider, for instance, the sorting according to the starting point on the spatial dimension i, on which the parent node o has extent (before removing any entry) $[o_{Ri\text{-}}(T_I), o_{Ri+}(T_I)]$ at the insertion time T_I. Then, after removal its extent would become $[o_{Ri\text{-}}(T_I)+\lambda \cdot |o_{Ri}(T_I)|, o_{Ri+}(T_I)]$ (where $|o_{Ri}(T_I)|=o_{Ri+}(T_I)-o_{Ri\text{-}}(T_I)$), assuming the starting points of the entries' extents uniformly distribute in $[o_{Ri\text{-}}(T_I), o_{Ri+}(T_I)]$. On the other dimensions, the extents of o are approximately the same as those of o' (the node after removal). The difference between the sweeping region areas of o and o' is the "benefit" of this sorting. Then, the final decisions (of (i) and (ii)) correspond to the combination with the largest benefit, after which the entries for re-insertion are obtained with only one sorting (according to the selected combination).

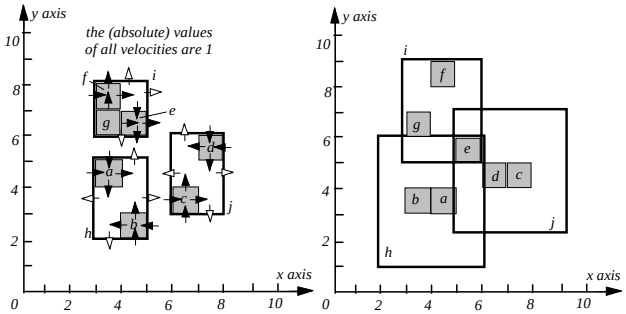
It is worth mentioning that, even though the overall data distribution may be non-uniform, the distribution inside leaf nodes can be regarded as uniform. This is because the MBR (VBR) of a leaf covers a small area of the data (velocity) space, inside which the data distribution may not change significantly. On the other hand, since the MBRs/VBRs of nodes at higher levels (≥2 in our implementation) cover larger areas, their contents are less uniform. For these nodes the set of re-inserted entries are decided by performing the actual $(4 \cdot d)$ sorting as described earlier. Note that this does not increase the total update cost significantly, because the number of non-leaf overflows accounts for a negligible fraction of the total number of overflows at the leaf level.

- **Node Split**

Similar to TPR-trees, the split algorithm of the TPR*-tree computes the overall perimeter for each dimension i, by considering all possible divisions of the entry list sorted according to the starting/ending points of their extents on this dimension. Then, the split axis is selected as the one with the smallest overall perimeter. The difference is that, in our case the "perimeter" of a node should be defined as the perimeter of the sweeping region of the corresponding transformed rectangle (with respect to the optimization query). The reasoning is that, (i) a polygon (i.e., the sweeping region) with small perimeter usually has small area (but not the opposite), and (ii) the sweeping regions of nodes created this way are more "square" (i.e., we avoid sweeping regions that are especially elongated on one particular axis). Note that the perimeter computation is very efficient because the number of vertices of a sweeping region is small (at most 6 in 2D space). The sorting on a spatial dimension is based on the entry extents at insertion time T_I. After deciding the split axis SA, the algorithm sorts the entries according to the starting/ending points of their extents on SA, and considers all possible divisions. The one that minimizes equation 3-2 becomes the final splitting.

4.2 Deletion

To remove an object e whose (i) MBR at the deletion time T_D is $e_R(T_D)$, and (ii) VBR is e_V, the deletion algorithm first identifies the leaf node that contains e, by searching the tree using $e_R(T_D)$ as the query window. Specifically, a node o is visited if and only if (i) its MBR $o(T_D)$ at time T_D contains $e_R(T_D)$, and (ii) its VBR o_V contains e_V. A difference from normal window queries is that the search terminates as soon as e is found. We note that the deletion overhead dominates the cost of an update (which involves one deletion and insertion) in TPR- (also TPR*-) trees because of the ever-increasing MBR overlap. Consider, for example, Figure 4.4a where objects $a,b,\ldots,f$ are stored in 3 leaf nodes h,i,j whose MBRs are mutually disjoint at time 0. Assume at time 1 object e changes its velocity, and thus its previous record must be removed from the index (before a new record can be inserted). To find the node that contains e, the algorithm starts from the root, and considers those nodes whose MBRs at time 1 contain $e_R(1)=\{5,6,5,6\}$ and VBRs contain $e_V=\{1,1,-1,-1\}$. In Figure 4.4b, all the leaf nodes satisfy these conditions and hence in the worst case all of them must be searched. Let the access order be alphabetic; then, node h is first visited, followed by i, where record e is found and removed.



(a) MBRs & VBRs at time 0 (b) MBRs at time 1

Figure 4.4: Deleting record e

The traditional TPR-tree will tighten the MBR of node i (enclosing f and g) after the deletion. We propose an improved *active tightening* technique that allows adjusting multiple MBRs in a single deletion, when the object to be removed falls in the overlapping region of these MBRs. For instance, although node h does not incur

any change, we can still tighten its MBR without any additional cost (observe that the MBR will decrease considerably), because the root node must be written to disk anyway (to reflect the extent change of *i*). The MBR of *j*, however, cannot be adjusted because it was not visited during the search for *c* (hence its tightest MBR at time 1 cannot be computed).

Figure 4.5 shows a more general case where the tree consists of 3 levels and the entry *e* to be removed is found in node N_4, after visiting (leaves) N_1, N_3, (level-1) nodes N_5, N_6 and the root (accessed nodes are shaded). The tree path that must be written to disk after deletion includes N_4, N_6, and the *root*; as a result, the set of entries that can be tightened includes n_3, n_4, n_5, n_6. Particularly, notice that n_1 cannot be modified even though N_1 is accessed (this will force to write N_5). On the other hand, the MBR of n_5 is adjusted to tightly bound n_1 and n_2 (which, however, may not be the tightest for their respective child nodes).

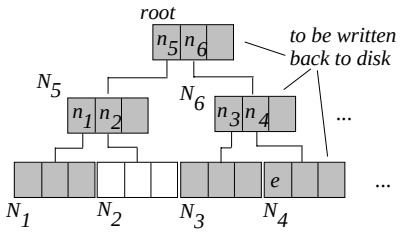


Figure 4.5: MBR tightening after deletion of *e*

This technique is especially useful for datasets where object updates are infrequent. In such cases, many nodes may not receive any update for a long time and their MBRs will become increasingly "loose". Active tightening can also be applied in *choose path*, however with small effect because, as described earlier, *choose path* does not access any leaf node. The rest of the deletion algorithm is similar to that of the TPR-tree. Specifically, if (after removing the entry) the leaf node generates an underflow, all its entries are re-inserted and the node is erased. This may cause a higher level node to underflow, which is handled in a similar way.

5. Experiments

This section (i) assesses the accuracy of the probabilistic model, (ii) evaluates the number of node accesses (NA) of the TPR- and TPR*-tree against the lower bound provided by the hypothetical structure, and (iii) compares the query/update performance of TPR- and TPR*-trees in practice. For all experiments, the disk page size is set to 1k bytes, and the maximum number of entries in a node is 27 for all indexes. We use a relatively small page size so that the number of nodes in an index simulates realistic situations, where the dataset cardinality is higher. Similar methodology was also used to evaluate R*-trees [BKSS90]. The section is divided into two parts, focusing on the evaluation of the analytical model and practical performance, respectively.

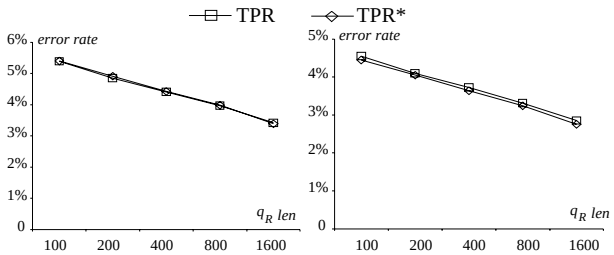
5.1 Evaluation of the cost model

Our first goal is to show the accuracy of the cost model and assess the efficiency of the structures with respect to the lower bound. Towards this direction, we deploy spatio-temporal data that contain insertions at a single timestamp 0. Specifically, objects' MBRs are taken from a real spatial dataset LA or LB [Tiger] (containing 128k and 109k 2D rectangles, respectively) where each axis of the space is normalized to [0,10000]. Then, each object is associated with a VBR such that on each dimension (i) the velocity extent is zero (i.e., the object does not change spatial extents during its movement), (ii) the velocity value distribution is skewed (Zipf, biased coefficient 0.8) towards 0 in range [0,50], and (iii) the velocity can be positive or negative with equal probability. For each dataset, all indexes have similar sizes. Specifically, for LA, each tree has 4 levels and around 6700 leaves, while the numbers are 4 and 5700 for LB.

Each query *q* has three parameters: $q_R len$, $q_V len$, and $q_T len$, such that (i) its MBR q_R is a square, with length $q_R len$, uniformly generated in the data space, (ii) its VBR is $q_V=\{-q_V len/2, q_V len/2, -q_V len/2, q_V len/2\}$, and (iii) its query interval is $q_T=[0, q_T len]$. The query cost is measured as the average number of node accesses in executing a workload of 200 queries with the same parameters. For each workload, the horizon parameters of the TPR- and TPR*-trees are set to the corresponding $q_T len$.

- **Cost model accuracy**

We evaluate the accuracy of equation 3-1 by using it to predict the query costs of TPR- and TPR*-trees. The error is measured as $\sum_i |act_i - est_i| / \sum_i act_i$, where act_i (est_i) denotes the actual (estimated) number of node accesses for the *i*-th query in the workload. Figure 5.1 plots the error rate, for LA and LB, as a function of $q_R len$ (which ranges from 100 to 1600, i.e., q_R covers up to 2.56% of the data space), fixing $q_V len$ and $q_T len$ to 5 and 50, respectively. Our model is accurate in all cases (maximum error below 6%), confirming our probabilistic derivation. The error rates with respect to other parameters (i.e., $q_V len$ and $q_T len$) are similar and omitted due to space constraints.



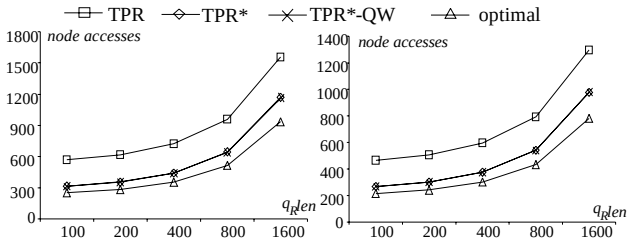
(a) Error rate vs. $q_R len$ (LA) (b) Error rate vs. $q_R len$ (LB)

Figure 5.1: Accuracy of equation 3-1 ($q_V len$=5, $q_T len$=50)

- **Comparison with the optimal performance**

This set of experiments compares the costs of TPR- and

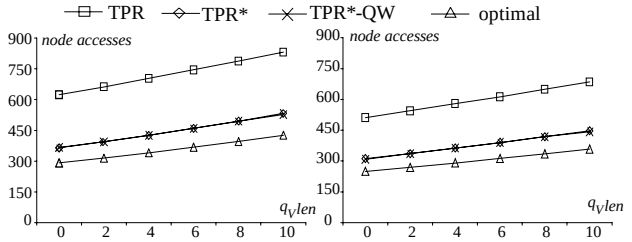
TPR*-trees to the optimal performance computed by the algorithm in Figure 3.3. In order to verify the effectiveness of optimizing the TPR*-tree with respect to static point queries, for each query workload, we create a special TPR*-tree (referred to as TPR*-QW), which minimizes equation 3-1 for the parameters of the workload. Figure 5.2 shows the number of node accesses as a function of $q_R len$, fixing $q_V len$=5 and $q_T len$=50. The TPR-tree incurs almost twice the cost of the lower bound, while the TPR* and TPR*-QW are nearly optimal. It is important to note that, *although the TPR* is not specifically optimized for the query workload parameters, it has almost the same performance as the TPR*-QW* (difference below 2%), confirming its general applicability (also verified by subsequent experiments).



(a) NA vs. $q_R len$ (LA) (b) NA vs. $q_R len$ (LB)

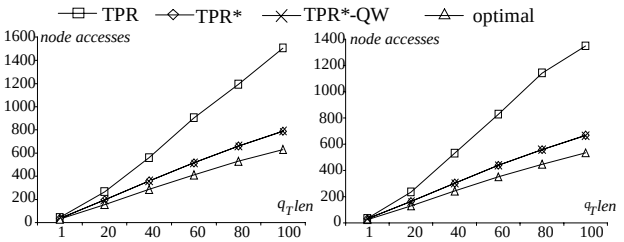
Figure 5.2: Query performance ($q_V len$=5, $q_T len$=50)

Next we fix $q_R len$, $q_T len$ to 400, 50 respectively, and vary $q_V len$ from 0 (static query) to 10. As shown in Figure 5.3, the query cost increases with $q_V len$, because higher $q_V len$ leads to faster enlargement of the query MBR and hence more node accesses. The TPR*-tree and TPR*-QW again have similar performance, and outperform the TPR-tree significantly. Figure 5.4 repeats the same experiments for different $q_T len$ (1 to 100), fixing $q_R len$, $q_V len$ to their median values 400 and 5, respectively. The TPR-tree is competitive only for very small $q_T len$ because in this case the indexes are optimized for the very near future, and thus behave like conventional R-trees (i.e., the data clustering is mainly according to objects' MBRs). The TPR* and TPR*-QW, however, achieve considerable speedup for longer $q_T len$, and the difference becomes larger as $q_T len$ increases. In all cases, the TPR*- has identical performance to TPR*-QW, and their costs are close to the optimal values (10%-20% higher).



(a) NA vs. $q_V len$ (LA) (b) NA vs. $q_V len$ (LB)

Figure 5.3: Query performance ($q_R len$=400, $q_T len$=50)



(a) NA vs. $q_T len$ (LA) (b) NA vs. $q_T len$ (LB)

Figure 5.4: Query performance ($q_R len$=400, $q_V len$=5)

5.2 Evaluation of practical performance

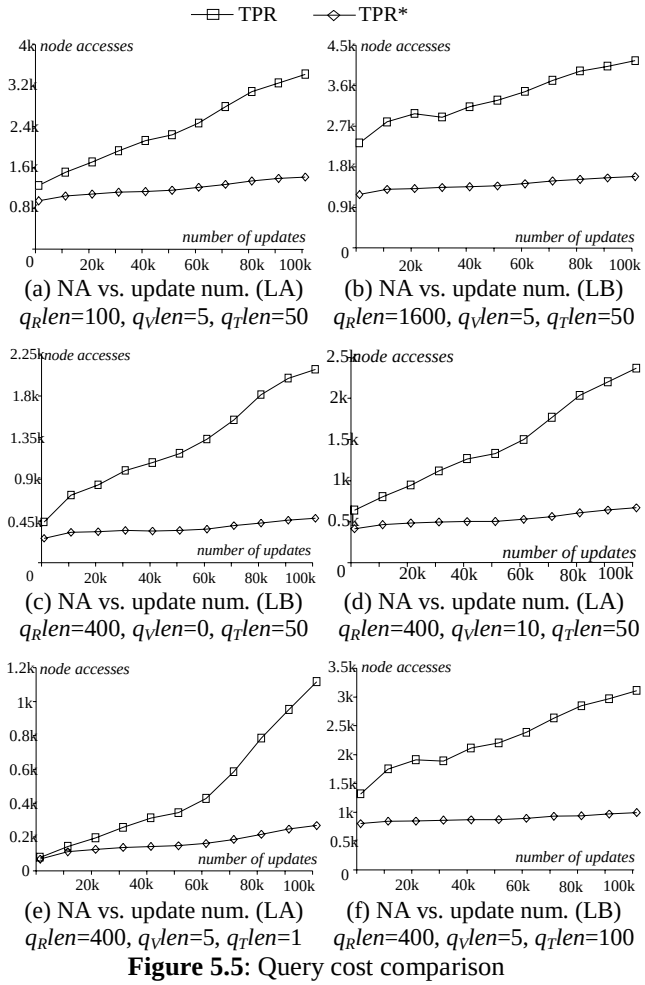
We now proceed to compare TPR- and TPR*-trees in practical scenarios where (i) object updates are chronological, and (ii) the queries are "versatile", i.e., a workload involves queries with distinct parameters. Due to the lack of real datasets, we use synthetic data simulating moving aircrafts. First 5000 rectangles are sampled from a real spatial dataset (LA/LB) and their centroids serve as the "airports". At timestamp 0, 100k aircrafts (each represented as a point) are generated such that for each aircraft *o*, (i) its location is at one (random) airport, (ii) it (randomly) chooses a destination airport, and (iii) its velocity value *o.Vel* uniformly distributes in [20,50], and (iv) the velocity direction is decided by the source and destination airports. If *o.Dist* is the Euclidean distance (between the starting and ending airports), an aircraft generates the next update at time *o.Dist*/*o.Vel* (when it reaches its destination), by randomly selecting the next destination. The average values for *o.Dist* and *o.Vel* are around 3000 and 35 respectively, so that every object issues an update approximately every 90 time units.

For each dataset, we construct a TPR- and TPR*-tree, whose horizons are fixed to 50, by first inserting 100k aircrafts at timestamp 0, and then, for every object update, performing one deletion and insertion. As a result, the size of each tree remains the same at all times. Specifically, each tree contains 4 levels and around 5400 leaf nodes. The cost is measured again as the average number of node accesses in executing a workload consisting of 200 queries with the same parameters $q_R len$, $q_V len$, $q_T len$. A query is generated as follows: (i) on each spatial (velocity) dimension i ($1 \le i \le 2$), the starting point of its extent uniformly distributes in [0,10000–$q_R len$] ([-10, 10–$q_V len$]), and (ii) the starting query timestamp $q_{T.}$ is uniform in [T_C, T_C+120–$q_T len$] (T_C is the time when the query workload is performed). Note that, unlike the experiments in the previous section, queries in a workload have different VBRs and starting timestamps.

- **Query cost comparison**

In order to study the deterioration of the indexes with time, we measure the performance of TPR and TPR*, using the same query workload, after every 10k updates. Figure 5.5 shows the query cost (for datasets generated from LA and LB as described above) as a function of the

number of updates, using workloads with different parameters. Specifically, in each row (e.g., Figures 5.5a, 5.5b) we fix two workload parameters (e.g., $q_V len$ and $q_T len$), and use the smallest and largest values for the third (e.g., $q_R len$). It is clear that the deterioration of the TPR-tree is considerably faster in all cases. In particular, after 100k updates, the query cost of the TPR-tree is up to 5 times higher than the TPR*-tree. This indicates that the TPR* update algorithms, which take the special characteristics of moving objects into account, are more effective than the algorithms of the TPR, which follow directly those of the R*-tree.

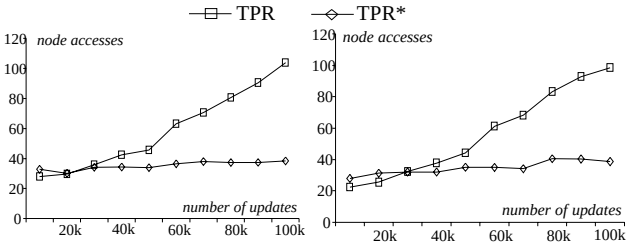


(a) NA vs. update num. (LA) $q_R len$=100, $q_V len$=5, $q_T len$=50
(b) NA vs. update num. (LB) $q_R len$=1600, $q_V len$=5, $q_T len$=50
(c) NA vs. update num. (LB) $q_R len$=400, $q_V len$=0, $q_T len$=50
(d) NA vs. update num. (LA) $q_R len$=400, $q_V len$=10, $q_T len$=50
(e) NA vs. update num. (LA) $q_R len$=400, $q_V len$=5, $q_T len$=1
(f) NA vs. update num. (LB) $q_R len$=400, $q_V len$=5, $q_T len$=100

Figure 5.5: Query cost comparison

- **Update cost comparison**

Figure 5.6 compares the average cost (amortized over each insertion and deletion) as a function of the number of updates. The TPR*-tree has nearly constant update cost, while the cost of the TPR-tree increases significantly (e.g., nearly 100 node accesses after 100k updates!). This is because, as mentioned earlier, each deletion must perform a query to retrieve the object to be removed, and the cost of this query increases with the number of updates (see Figure 5.5). The TPR*-tree is slightly more expensive before 20k updates because, up to this time the TPR-tree has not degraded considerably and the insertion algorithm (*choose path*) of TPR* accesses more nodes.



(a) NA vs. update num.(LA) (b) NA vs. update num.(LB)

Figure 5.6: Update cost comparison

In summary, the experimental evaluation confirms that:

- The proposed cost model is highly accurate, yielding error below 6%.
- The TPR*-tree has nearly optimal performance and consistently outperforms the TPR-tree by a wide margin.
- The TPR*-tree remains efficient as time evolves, while the TPR-tree degrades significantly.

6. Conclusion

Although spatio-temporal indexes are crucial for the efficient processing of conventional and novel query types ([TP02, BJKS02]), the existing structures are either purely theoretical, or based on traditional spatial access methods with limited provision for moving objects. This paper proposes the first analytical model that (i) accurately estimates the costs of predictive window queries, and (ii) quantifies the performance of spatio-temporal access methods. Then it presents the TPR*-tree, a new spatio-temporal access method highly optimized for moving data. Extensive experiments prove that the TPR*-tree significantly outperforms the conventional TPR-tree under all conditions.

This work initiates several interesting directions for future work. First, we would like to investigate alternative predictive queries using the TPR*-tree, in particular, nearest neighbors and joins. A *predictive nearest neighbor* query specifies a (moving) query point q and retrieves the database objects that will come closest to q during the query interval. A *predictive spatio-temporal join* will return all pairs of objects from two datasets (each indexed by a TPR*-tree) that will come within distance d from each other during the query interval.

Currently, all existing spatio-temporal access methods either aim at the past or the future, but not both. It would be interesting to develop a "persistent" version of the TPR*-tree, suitable for historical and future information retrieval. In such a tree, outdated versions of objects are not deleted, but kept separately. Queries can now involve both past and future intervals (e.g., find all airplanes that appeared in the last five minutes, or will appear in the next five minutes in the airspace of Hong Kong).

Acknowledgements

This work was supported by grants HKUST 6197/02E and HKUST 6081/01E from Hong Kong RGC.

References

[AAE00] Agarwal, P., Arge, L., Erickson, J. Indexing Moving Points. PODS, 2000.

[BGH97] Basch, J., Guibas, L., Hershberger, J. Data Structures for Mobile Data. SODA, 1997.

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reserve Nearest Neighbor Queries for Moving Objects. IDEAS, 2002.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD, 1990.

[CC02] Choi, Y., Chung, C. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. SIGMOD, 2002.

[CR00] Cai, M., Revesz, P. Parametric R-Tree: An Index Structure for Moving Objects. COMAD, 2000.

[HKTG02] Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D. Efficient Indexing of Spatiotemporal Objects. EDBT, 2002.

[KF93] Kamel, I., Faloutsos, C. On Packing R-Trees. CIKM, 1993.

[KGT+01] Kollios, G., Gunopulos, D., Tsotras, V., Delis, A., Hadjieleftheriou, M. Indexing Animated Objects Using Spatiotemporal Access Methods. TKDE, 13(5): 758-777, 2001.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. PODS, 1999.

[PJT00] Pfoser, D., Jensen, C, Theodoridis, Y. Novel Approaches to the Indexing of Moving Object Trajectories. VLDB, 2000.

[PSTW93] Pagel, B., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance in Spatial Data Structures. PODS, 1993.

[S90] Samet, H. The Design and Analysis of Spatial Data Structures. Addison-Wesley Publishing Company, 1990.

[SJ02] Saltenis, S., Jensen, C. Indexing of Moving Objects for Location-Based Services. ICDE, 2002.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. SIGMOD, 2000.

[Tiger] Http://www.census.gov/geo/www/tiger/

[TP01] Tao, Y., Papadias, M. The MV3R-Tree: A Spatial-Temporal Access Method for Timestamp and Interval Queries. VLDB, 2001.

[TP02] Tao, Y., Papadias, D. Time-Parameterized Queries in Spatio-Temporal Databases. SIGMOD, 2002.

[TSP03] Tao, Y., Sun, J. Papadias, D. Selectivity Estimation for Predictive Spatio-Temporal Queries. ICDE, 2003.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. TKDE, 12(1): 19-32, 2000.

[TUW98] Tayeb, J., Ulusoy, O., Wolfson, O. A Quadtree-Based Dynamic Attribute Indexing Method. The Computer Journal, 41(3): 185-200, 1998.

Appendix

Lemma A.1 (Best split position): Let o be the node to be split, i be the split dimension being considered, and q the target query for optimization.

- If i is a spatial dimension, let v_C be the middle point of the extent of o on dimension i, namely, $v_C=(o_{Ri-}+o_{Ri+})/2$; then, $sp_i=v_C$.
- If i is a velocity dimension, let $[v_-, v_+]$ be the valid range for sp_i (i.e., $v_-=o_{Vi-}+\xi\cdot|o_{Vi}|$, $v_+=o_{Vi+}-\xi\cdot|o_{Vi}|$), and v_C be the middle velocity of the valid range (i.e., $v_C=(v_-+v_+)/2$). Then, the best value for sp_i is obtained by comparing $[v_-, v_+]$ with $[q_{Vi-}, q_{Vi+}]$: (i) If $v_+<q_{Vi-}$, $sp_i=v_+$; (ii) If $q_{Vi+}<v_-$, $sp_i=v_-$; (iii) If $[v_-, v_+]$ intersects $[q_{Vi-}, q_{Vi+}]$, let $[I_-, I_+]$ be the intersection range (i.e., $I_-=\max(v_-,q_{vi-})$, $I_+=\min(v_+,q_{vi+})$. Then, sp_i is decided according to the relation between $[I_-, I_+]$ and v_C (i.e., middle of the validity range): (a) if $I_+<v_C$, $sp_i=I_+$; (b) if $v_C<I_-$, $sp_i=I_-$; (c) if $I_+\le v_C\le I_+$, $sp_i=v_C$. ■

The first part of the lemma is simple, i.e., the best split position on a spatial axis is at the middle of the node's extent on this dimension. We explain the second half (split on velocity) using Figure A.1 where rectangle *ABCD* denotes the MBR of o (to be split), whose VBR is {-1,4,1,1}. Let the minimum node usage ξ be 40% and hence the valid range for the split position sp on the x-velocity axis is [1,2]. Assume, for simplicity, that the target query q has parameters $|q_{Ri}|=0$, $q_V=\{0,0,0,0\}$, and $q_T=[0,1]$. Using Lemma A.1, we obtain that the best sp on this dimension is sp=1. As a result, the MBRs of the new nodes o_1, o_2 are the same as that of o; the VBR of o_1 is {-1,sp=1,1,1} while that of o_2 is {sp=1,4,1,1}. As shown in Figure A.1 the MBRs $o_1'_R(1)$, $o_2'_R(1)$ of o_1, o_2 at time 1 are rectangles *IJKL*, *EFGH*, respectively. Hence the sweeping region $SR(o_1',[0,1])$ is hexagon *ABJKLI*, while $SR(o_2',[0,1])$ is *ABFGHD*. On the other hand, $SR(o',[0,1])$, the sweeping region of the original node, is hexagon *ABFGLI*. Thus, ΔA_{SRi} (see equation 3-2) is the area of hexagon *ABJKHD* that includes (i) rectangle *ABCD*, (ii) trapezoids *DCKH*, and (iii) *BJKC*. Note that, the areas of (i) and (ii) are invariant to sp (the split position), or specifically $area(ABCD) = |AB|\cdot|BC| = |o'_{Rx}|\cdot|o'_{Ry}|$, and $area(DCKH) = |DC|\cdot 1\cdot|q_T| = |o'_{Rx}|\cdot 1\cdot|q_T|$, where "1" corresponds to the velocity of edge *DC*. Hence, to minimize ΔA_{SRi}, we should minimize $area(DCKH) = |BC|\cdot sp\cdot|q_T|$ (the shaded region in Figure A.1). Thus, the best sp is the smallest value (i.e., 1) in the valid range, confirming the value returned by the lemma.

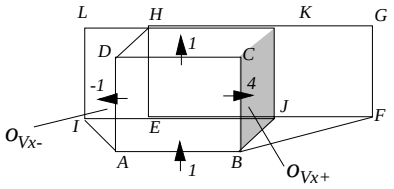


Figure A.1: Illustration of Lemma A.2