Query and Update Efficient B$^+$-Tree Based Indexing of Moving Objects

Christian S. Jensen[1] Dan Lin[2] Beng Chin Ooi[2]

[1]Department of Computer Science
Aalborg University, Denmark
csj@cs.auc.dk

[2]School of Computing
National University of Singapore, Singapore
{*lindan, ooibc*}*@comp.nus.edu.sg*

Abstract

A number of emerging applications of data management technology involve the monitoring and querying of large quantities of continuous variables, e.g., the positions of mobile service users, termed moving objects. In such applications, large quantities of state samples obtained via sensors are streamed to a database. Indexes for moving objects must support queries efficiently, but must also support frequent updates. Indexes based on minimum bounding regions (MBRs) such as the R-tree exhibit high concurrency overheads during node splitting, and each individual update is known to be quite costly. This motivates the design of a solution that enables the B$^+$-tree to manage moving objects. We represent moving-object locations as vectors that are timestamped based on their update time. By applying a novel linearization technique to these values, it is possible to index the resulting values using a single B$^+$-tree that partitions values according to their timestamp and otherwise preserves spatial proximity. We develop algorithms for range and k nearest neighbor queries, as well as continuous queries. The proposal can be grafted into existing database systems cost effectively. An extensive experimental study explores the performance characteristics of the proposal and also shows that it is capable of substantially outperforming the R-tree based TPR-tree for both single and concurrent access scenarios.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

1 Introduction

An infrastructure is emerging that enables data management applications that rely on the tracking of the locations of moving objects such as vehicles, users of wireless devices, and goods. Further, a wide range of other applications, beyond those to do with moving objects, rely on the sampling of continuous, multidimensional variables. The provisioning of high performance and scalable data management support for such applications presents new challenges. One key challenge derives from the need to accommodate frequent updates while simultaneously allowing for efficient query processing [6, 13].

This combination of desired functionality is particularly troublesome in the context of indexing of multidimensional data. The dominant indexing technique for multidimensional data with low dimensionality, the R-tree [5] (and its descendants such as the R*-tree [1]), was conceived for largely static data sets and exhibits poor update performance. The Time-Parameterized R-tree (TPR-tree) [19] (as well as several of its recent descendants [11]) models object locations as linear functions of time and supports queries on the current and anticipated near-future positions of moving objects. While the use of linear rather than constant functions may reduce the need for updates by a factor of three [3], update performance remains a problem.

Individual updates tend to be costly, and the problem is exacerbated by the concurrency control algorithms of the R-trees, such as the Rlink-tree [8], not being able to adequately handling a high degree of concurrent accesses that involve updates. Notably, frequent tree ascents caused by node splitting and propagation of MBR updates lead to costly lock conflicts. This problem is inherent in many multi-dimensional indexes. Another problem with existing solutions to moving-object indexing is that they are not easily integrated into existing database systems.

This paper proposes a novel way of indexing moving objects using the classical B$^+$-tree without compromising on query and storage efficiency. The motivation for using the B$^+$-tree is threefold. First, the B$^+$-tree is used widely in commercial database systems and has proven to be very efficient with respect to queries as well as updates, ro-

bust with respect to varying workloads, and scalable. Second, being a one-dimensional index, it does not exhibit the update performance problems associated with MBR-based multi-dimensional indexes. Third, it is typically appropriate to model moving-object extents as points. This enables linearization and subsequent B^+-tree indexing.

To use the B^+-tree, we must be able to linearize the representation of the locations of the moving objects. This is done by means of a space-filling curve, which enumerates every point in a discrete, multi-dimensional space. Attractive space-filling curves such as the Peano curve (or Z-curve) and the Hilbert curve, which we use in this paper, preserve proximity, meaning that points close in multidimensional space tend to be close in the one-dimensional space obtained by the curve [12].

A B^+-tree with the above space-filling curves works very well for static databases. A naive way to accommodate moving points is to update each object in the database at each time interval. To avoid an excessive update overhead, we propose a novel indexing method, termed the B^x-tree, where "x" indicates the flexibility of the proposed method in employing a specific ("x") space-filling curve as part of the linearization function.

First, we model moving objects as linear functions of time. Thus, the data to be indexed in the B^x-tree are not points (constant functions), but linear functions coupled with the times they were updated. Intuitively, an update occurs when the position predicted by an existing function is deemed inaccurate [3]. Second, we effectively "partition" the index, placing entries in partitions based on their update time. More specifically, we first partition the time axis into intervals where the duration of an interval is an approximation of the maximum duration in-between two updates of any object location. We then partition each such interval into n equal-length sub-intervals, termed *phases*, where n is determined based on minimum time duration within which each object issues an update of its position. Each phase is assigned the time point it ends as a *label timestamp*, and a label timestamp is mapped to a partition. An update is placed in the partition given by the label timestamp of the phase during which it occurs. For an object, the value indexed by the B^x-tree is the concatenation of its partition number and the result of applying the underlying space-filling method to the position of the object as of the label timestamp of its phase.

This mapping scheme overcomes the limitation of the B^+-tree, which is able to only keep the snapshot of all the objects at the same time point. This scheme reduces the update frequency, it preserves spatial proximity within each partition, and it facilitates queries on anticipated near-future positions.

Based on the above, we propose efficient algorithms for range and k nearest neighbor queries, as well as for continuous queries. The algorithms are general and can be applied to indexes that use sampling techniques to model moving objects. Like any new indexing method built on top of the B^+-tree, the paper's proposal can be grafted into existing database systems cost effectively.

The paper reports on an extensive experimental study, which includes a comparison with the TPR-tree. The results show that the B^x-tree is efficient with respect to storage space and range and k nearest neighbor queries. Indeed, the B^x-tree is capable of outperforming the TPR-tree by a wide margin in single and concurrent access environments.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3, describes the structure of the proposed B^x-tree, and it presents the associated query and update operations. Section 4 covers comprehensive performance experiments. Finally, Section 5 concludes.

2 Related Work

Traditional indexes for multi-dimensional databases, such as the R-tree [5] and its variants (e.g., [1]) were, implicitly or explicitly, designed with the main objective of supporting efficient query processing as opposed to enabling efficient update. This works well in applications where queries are relatively much more frequent than updates. However, applications involving the indexing of moving objects exhibit workloads characterized by heavy loads of updates, in addition to frequent queries.

Several new index structures have been proposed for moving-object indexing, and recent surveys exist that cover different aspects of these [11, 13]. One may distinguish between indexing of the past positions versus indexing of the current and near-future positions of spatial objects. Our approach belongs to the latter category.

Past positions of moving objects are typically approximated by polylines composed of line segments. It is possible to index line segments by R-trees, but the trajectory memberships of segments are not taken into account. In contrast to this, the Spatio-Temporal R-tree [15] attempts to also group segments according to their trajectory memberships, while also taking spatial locations into account. The Trajectory-Bundle tree [15] aims only for trajectory preservation, leaving other spatial properties aside.

The representations of the current and near-future positions of moving objects are quite different, as are the indexing challenges and solutions. Positions are represented as points (constant functions) or functions of time, typically linear functions. The Lazy Update R-tree [9] aims to reduce update cost by handling updates of objects that do not move outside their leaf-level MBRs specially, and a generalized approach to bottom-up update in R-trees has recently been examined [10].

Tayeb et al. [24] use PMR-Quadtrees [20] for indexing the future linear trajectories of one-dimensional moving points as line segments in (x, t)-space. The segments span the time interval that starts at the current time and extends some time into the future, after which time, a new tree must be built. Kollis et al. [7] employ dual transformation techniques which represent the position of an object moving in a d-dimensional space as a point in a $2d$-dimensional space. Their work is largely theoretical in nature. Based on a similar technique, Patel et al. [14] have most recently de-

veloped a practical indexing method, termed STRIPES, that supports efficient updates and queries at the cost of higher space requirements.

Finally, we cover the Time-Parameterized R-tree (TPR-tree) [19] in some detail, as we use this tree for comparison in our performance study. An extension to the R*-tree, the TPR-tree indexes linear functions of time. The current location of a moving point is found simply by applying the function representing its location to the current time. MBRs are also functions of time. Specifically, in each dimension, the lower bound of an MBR is set to move with the maximum downward speed of all enclosed objects, while the upper bound is set to move with the maximum upward speed of all enclosed objects. As enclosed objects may be both moving points and moving rectangles, this ensures that the bounding rectangles are indeed bounding at all times considered. Frequent updates are needed to ensure that moving objects that are currently close are assigned to the same bounding rectangles. Further, bounding rectangles never shrink and are generally larger than strictly needed. To counter this phenomenon, the so-called "tightening" is applied to bounding rectangles when they are accessed.

Algorithms for nearest neighbor and reverse nearest neighbor queries on moving objects have been proposed based on the TPR-tree [2]. Next, two notable proposals exist that build on the ideas of the TPR-tree. Procopiuc et al. [16] propose the STAR-tree. This index seems to be best suited for workloads with infrequent updates. Tao et al. [22] adopt assumptions about the query workload that differ slightly from those underlying the TPR-tree. This leads to a different grouping of objects into index tree nodes.

To the best of the authors' knowledge, no proposals for the indexing of moving objects exist that use a combination of temporal partitioning and space-filling curves. However, our work adopts a design philosophy similar to that of iDistance [25], where application of a mapping function that uses reference points and metric distances with respect to the reference points enables B$^+$-tree indexing of high-dimensional points for the purpose of nearest neighbor search.

3 Structure and Algorithms

We first describe the structure of the B^x-tree. We then cover algorithms for range and kNN queries and continuous queries. Finally, update, insertion, and deletion are covered.

3.1 Index Structure

The base structure of the B^x-tree is that of the B$^+$-tree. Thus, the internal nodes serve as a directory. In order to support B-link concurrency control [21], each internal node contains a pointer to its right sibling (the pointer is non-null if one exists). The leaf nodes contain the moving-object locations being indexed and corresponding index time. We proceed to describe how object locations are mapped to single-dimensional values.

Specifically, we use a space-filling curve for this purpose. Such a curve is a continuous path which visits every point in a discrete, multi-dimensional space exactly once and never crosses itself.

We consider versions of the B^x-tree that use the Peano curve (or Z-curve) and the Hilbert curve (see Figure 1). Although other curves may be used, these two are expected to be particularly good. Analytical and empirical studies [4, 12] show that for the two-dimensional space we consider, these curves are effective in preserving proximity, meaning that points close in multidimensional space tend to be close in the one-dimensional space obtained by the curve. The Hilbert curve is expected to be (slightly) better than the Peano curve [4].

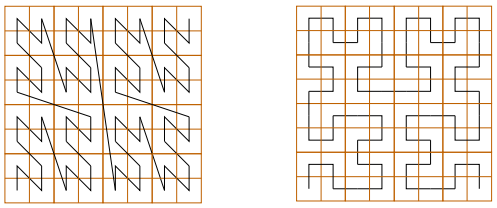


Figure 1: Space-Filling Curves

In what follows, we term the value obtained from the space-filling curve the x_value; and for brevity, we use the Peano curve in most discussions.

To reduce this load, we model point values as linear functions of time, rather than simply as static points, i.e., constant functions. A recent study of GPS logs obtained from two dozen cars traveling in a semi-urban environment measures the number of updates needed to ensure that the values recorded in the database do not differ by more than some threshold from the real values. For realistic thresholds, the use of linear functions reduces the amount of updates to one third in comparison to constant functions [3].

An object location is thus given by $O = (\overrightarrow{x}, \overrightarrow{v})$, a position and a velocity, and an update time, or timestamp, t_u, where these values are valid.

In a leaf-node entry, an object O updated at t_u is represented by a value $B^x\,value(O, t_u)$:

$$B^x\,value(O, t_u) = [index\_partition]_2 \oplus [x\_rep]_2 \quad (1)$$

where $index\_partition$ is an index partition determined by the update time, x_rep is obtained using a space-filling curve, $[x]_2$ denotes the binary value of x, and $\oplus$ denotes concatenation. We proceed to detail this definition.

If we index the timestamped object locations without differentiating them based on their timestamps, we not only lose the proximity preserving property of the space-filling curve; the index will also be ineffective in locating an object based on its x_value. To overcome such problems, we effectively "partition" the index, placing entries in partitions based on their update time. More specifically, we denote by Δt_{mu} the time duration that is the maximum duration in-between two updates of any object location. We then

partition the time axis into intervals of duration Δt_{mu}, and we sub-partition each such interval into n equal-length sub-intervals, termed *phases*.

By mapping the update times in the same phase to the same so-called *label timestamp* and by using the label timestamps as prefixes of the representations of the object locations, we obtain index partitions, and the update times of updates determine the partitions they go to. In particular, an update with timestamp t_u is assigned a label timestamp $t_{lab} = \lceil t_u + \Delta t_{mu}/n \rceil_l$, where operation $\lceil x \rceil_l$ returns the nearest future label timestamp of x.

For example, Figure 2 shows a B^x-tree with $n = 2$. Objects with timestamp $t_u = 0$ obtain label timestamp $t_{lab} = \frac{1}{2}\Delta t_{mu}$; objects with $0 < t_u \leq \frac{1}{2}\Delta t_{mu}$ obtain label timestamp $t_{lab} = \Delta t_{mu}$; and so on.

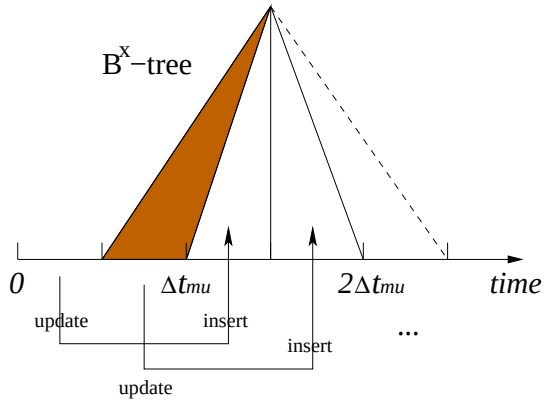


Figure 2: The B^x-Tree

Next, for an object with label timestamp t_{lab}, we compute its position at t_{lab} according to its position and velocity at t_u. We then apply the space-filling curve to this (future) position to obtain the second component of Equation 1.

This mapping has two main advantages. First, it enables the tree to index object positions valid at different times, overcoming the limitation of the B$^+$-tree, which is only able to index a snapshot of all positions at the same time. Second, it reduces the update frequency compared to having to update the positions of all objects at each timestamp when only some of them need to be updated. The two components of the mapping function in Equation 1 are consequently defined as follows:

$$index\_partition = (t_{lab}/(\Delta t_{mu}/n) - 1) \; mod \; (n+1)$$
$$x\_rep = x\_value(\overrightarrow{x} + \overrightarrow{v} \cdot (t_{lab} - t_u))$$

With the transformation, the B^x-tree will contain data belonging to $n+1$ phases, each given by a *label timestamp* and corresponding to a time interval. Within each of these, we apply a space-filling curve to an object position.

The choice of the value of n affects query performance and storage space. A large n results in smaller enlargements of query windows (covered in Section 3.2), but also results in more partitions and therefore a looser relationship among object locations. In addition, a large n yields a higher space overhead due to more internal nodes. When n is larger than 2, the storage space is a little more than that of the TPR-tree. When n is 1, query windows must be enlarged more than the enlargements of MBRs in the TPR-tree (enlargement details are covered in Section 3.2.1). Therefore, we choose $n = 2$.

To exemplify, let $n = 2$, $\Delta t_{mu} = 120$, and assume a Peano curve of order 3 (i.e., the space domain is 8×8).

Object positions $O_1 = ((7,2),(-0.1,0.05))$, $O_2 = ((0,6),(0.2,-0.3))$, and $O_3 = ((1,2),(0.1,0.1))$ are inserted at times 0, 10, and 100, respectively. We calculate the $B^x\,value$ for each as follows.

Step 1: Calculate label timestamps and index partitions.
$t^1_{lab} = \lceil (0 + 120/2) \rceil_l = 60$, $index\_partition^1 = 0 = (00)_2$
$t^2_{lab} = \lceil (10 + 120/2) \rceil_l = 120$, $index\_partition^2 = 1 = (01)_2$
$t^3_{lab} = \lceil (100 + 120/2) \rceil_l = 180$, $index\_partition^3 = 2 = (10)_2$

Step 2: Calculate positions x_1, x_2 and x_3 at t^1_{lab}, t^2_{lab}, and t^3_{lab}, respectively.
$x'_1 = (1,5)$, $x'_2 = (2,3)$, $x'_3 = (4,1)$.

Step 3: Calculate Z-values.
$[Z\_value(x'_1)]_2 = (010011)_2$
$[Z\_value(x'_2)]_2 = (001101)_2$
$[Z\_value(x'_3)]_2 = (100001)_2$

Step 4: Calculate $B^x\,value$.
$B^x value(O_1, 0) = (00010011)_2 = 19$
$B^x value(O_2, 10) = (01001101)_2 = 77$
$B^x value(O_3, 100) = (10100001)_2 = 161$

It is worth noting that *at most three* ranges exist at a single point in time. As time passes, repeatedly the first range expires (shaded area), and a new range is appended (dashed line). This use of rolling ranges enables the B^x-tree to handle time effectively.

3.2 Querying

In the following, we outline the search strategies for the B^x-tree.

3.2.1 Range Query

A range query retrieves all objects whose location falls within the rectangular range $q = ([qx^l_1, qx^u_1], [qx^l_2, qx^u_2])$ at time t_q not prior to the current time ("l" denotes lower bound, and "u" denotes upper bound).

A key challenge is to support predictive queries, i.e., queries that concern future times. Traditionally, indexes that use linear functions handle predictive queries by means of MBR enlargement (e.g., the TPR-tree); to the best of our knowledge, no algorithm for the predictive queries has been proposed for indexes that use snapshots of moving objects (e.g., the LUR-tree). We present a generic approach to processing such queries that is not constrained by the base structure. Figure 6 outlines the range query algorithm, which we proceed to explain.

To handle queries on the anticipated near future positions of objects, the B^x-tree uses query-window enlargement instead of MBR enlargement. This is done through the TimeParameterizedRegion function call in the algorithm. Because the B^x-tree stores an object's location as of some time after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time.

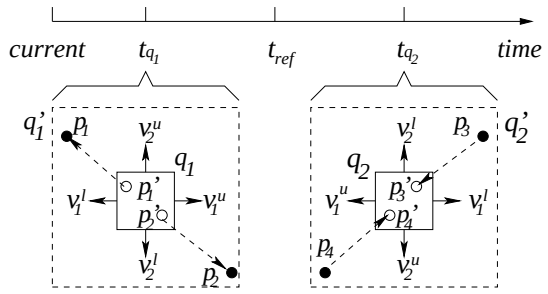


Figure 3: Query Window Enlargement

Consider the example in Figure 3, where t_{ref} denotes the time when the locations of four moving objects are updated to their current value index, and where predictive queries q_1 and q_2 (solid rectangles) have time parameters t_{q_1} and t_{q_2}, respectively. The figure shows the stored positions as solid dots and positions of the two first objects at t_{q_1} and the positions of the two last at t_{q_2} as circles. The two positions for each object are connected by an arrow.

The relationship between the two positions for each object is $p_i' = p_i + \overrightarrow{v} \cdot (t_q - t_{ref})$. The first two of the four objects thus are in the result of the first query, and the last two objects are in the result of the second query. To obtain this result, query rectangle q_1 needs to be enlarged to q_1' (dashed). This is achieved by attaching maximum speeds to the sides of q_1: v_1^l, v_2^l, v_1^u, and v_2^u. For example, v_1^u is obtained as the largest projection onto the x-axis of a velocity of an object in q_1'. (As we do not yet know q_1', a conservative approximation is used; more on this shortly.)

For q_2, the enlargement speeds are computed similarly. For example, v_2^u is obtained by projecting all velocities of objects in q_2' onto the y-axis; v_2^u is then set to the largest speed multiplied by -1.

The enlargement of query $q = ([qx_1^l, qx_1^u], [qx_2^l, qx_2^u])$ is given by query $q' = ([eqx_1^l, eqx_1^u], [eqx_2^l, eqx_2^u])$:

$$eqx_i^l = \begin{cases} qx_i^l + v_i^l \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx_i^l + v_i^u \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \quad (2)$$

$$eqx_i^u = \begin{cases} qx_i^u + v_i^u \cdot (t_{ref} - t_q) & \text{if } t_q < t_{ref} \\ qx_i^u + v_i^l \cdot (t_q - t_{ref}) & \text{otherwise} \end{cases} \quad (3)$$

The implementation of the computation of enlargement speeds proceeds in two steps. We first set them according to the maximum speeds of all objects, thus obtaining a preliminary q'. Then, with the aid of a two-dimensional histogram (e.g., a grid) that captures the maximum and minimum projections of velocities onto the axes of objects in each cell, we obtain the final enlargement speed in the area where the query window resides. Such a histogram can easily be maintained in main memory.

The time argument of a query exceeds the reference time of any object by at most Δt_{mu}. This is reasonable, as it is of little use to query so far into the future that all the values on which the result is based will have been updated before that time is reached. Considering the example in Figure 4, suppose a query is issued between $\frac{1}{2}\Delta t_{mu}$ and Δt_{mu} and that T_0, T_1, and T_2 are the partitions corresponding to the label timestamps $\frac{1}{2}\Delta t_{mu}$, Δt_{mu}, and $\frac{3}{2}\Delta t_{mu}$, respectively. Partition (or subtree) T_0 may need to be extended to Δt_{mu}

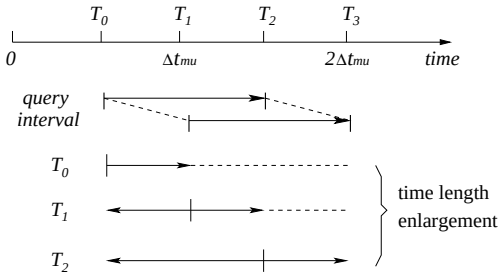


Figure 4: Time Length Enlargement

at most, and after that, T_0 expires; the T_1 may be extended backward to $\frac{1}{2}\Delta t_{mu}$ and forward to Δt_{mu}; T_2 may be extended backward to $\frac{1}{2}\Delta t_{mu}$ and forward to $\frac{3}{2}\Delta t_{mu}$. For either subtree, we can see that the maximum enlargement length is Δt_{mu}.

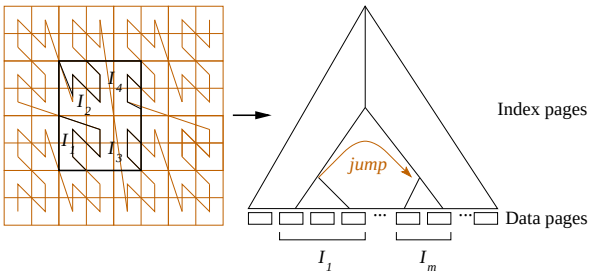


Figure 5: "Jump" in the Index

Next, we traverse the partitions of the B^x-tree with label timestamp no less than $\lceil t_q - \Delta t_{mu} \cdot (n-1)/n \rceil$ (i.e., they should be valid at t_q) to find objects falling in the enlarged query window q'. For example, if $\Delta t_{mu} < t_q \leq \frac{3}{2}\Delta t_{mu}$, Partition T_0 needs not be searched. In each partition, the use of a space-filling curve means that a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space—see Figure 5. Hence multiple traversals of the index result. We optimize these traversals by calculating the start and end points of the one-dimensional ranges and traverse the intervals by "jumping" in the index (as in [17]).

Let us step through the entire algorithm in detail. For each partition of the B^x-tree, we check whether it is valid at the query time t_q according to its label timestamp (lines 1–2). If it is valid, we enlarge query window q to q' by

function TimeParameterizedRegion (line 3) and calculate all start and end points $i_1, i_2, \ldots, i_{2m-1}, i_{2m}$ in ascending order of their x_values, where m is the number of intervals (line 4). The pair of points (i_{2j-1}, i_{2j}) start and end interval I_j $(1 \leq j \leq m)$.

We locate the leaf node containing the first point i_1, then traverse its right siblings using the B-link sibling pointers until we reach the next point i_2, where interval I_1 ends (lines 5–10).

To find i_3, the start point of interval I_2, we backtrack to a higher level where one "jumping" occurs, upon which we proceed to retrieve the objects with positions in interval I_2. This takes place in line 6 of the algorithm, where traversal from the root is avoided as much as possible. When all the intervals have been checked in this manner, we have obtained the set of all objects that may possibly belong to the result of range query q. For each object, we compute its position at t and return only those objects whose positions are actually in the query window q (lines 11–14).

Algorithm Range_query(q, t_q)
Input: q is the query range and t_q is the query time

1. **for** $j \leftarrow 0$ **to** n
2. **if** partition T_j of the B^x-tree is valid at t_q **then**
3. $q' \leftarrow \text{TimeParameterizedRegion}(q, t_q)$
4. calculate start and end points $i_1, \ldots, i_{2m}$ for q'
5. **for** $k \leftarrow 1$ **to** m **do**
6. locate leaf node containing point i_{2k-1}
7. **repeat**
8. store candidate objects in L
9. follow the right pointer to the sibling node
10. **until** node with point i_{2k} is reached
11. **for** each object in L **do**
12. **if** the object's position at t_q is inside q **then**
13. add the object to the result_set
14. **return** result_set

Figure 6: Range Query Algorithm

3.2.2 k Nearest Neighbor Query

Assuming a set of $N > k$ objects and given a query object with position $q = (qx_1, qx_2)$, the k nearest neighbor query (kNN query) retrieves k objects for which no other objects are nearer to the query object at time t_q not prior to the current time.

We compute this query by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. The algorithm is outlined in Figure 7. We first construct a range R_{q1} centered at q and with extension $r_q = D_k/k$, where D_k is the estimated distance between the query object and its k'th nearest neighbor; D_k can be estimated by the equation [23]:

$$D_k = \frac{2}{\sqrt{\pi}} \left[1 - \sqrt{1 - \left(\frac{k}{N} \right)^{\frac{1}{2}}} \right]$$

We compute the range query with range R_{q1} at time t_q, by enlarging it to a range R'_{q1} and proceeding as described in the previous section. If at least k objects are currently covered by R'_{q1} and are enclosed in the inscribed circle of R_{q1} at time t_q, the kNN algorithm returns the k nearest objects and then stops. It is safe to stop because we have considered all the objects that can possibly be in the result.

Otherwise, we extend R_{q1} by r_q to obtain R_{q2} and an enlarged window R'_{q2}. This time, we search the region $R'_{q2} - R'_{q1}$ and adjust the neighbor list accordingly. This process is repeated until we obtain an R_{qi} so that there are k objects within its inscribed circle.

Algorithm kNN_query$(q(qx_1, qx_2), k, t_q)$
Input: a query point $q(qx_1, qx_2)$, a number k of neighbors, and a query time t_q

1. construct range R_{q1} with q as center and extension r_q
2. $R'_{q1} \leftarrow \text{TimeParameterizedRegion}(R_{q1}, t_q)$
3. *flag* $\leftarrow$ true // *not enough objects*
4. $i \leftarrow 1$ // *first query region is being searched*
5. **while** *flag*
6. **if** $i = 1$ **then**
7. find all objects in region R'_{q1}
8. **else**
9. find all objects in region $R'_{qi} - R'_{qi-1}$
10. **if** k objects exist in inscribed circle of R_{qi} **then**
11. *flag* $\leftarrow$ false
12. **else**
13. $i \leftarrow i + 1$
14. $R_{qi} \leftarrow \text{Enlarge}(R_{qi-1}, r_q)$
15. $R'_{qi} \leftarrow \text{TimeParameterizedRegion}(R_{qi}, t_q)$
16. **return** k NNs with respect to q

Figure 7: kNN Query Algorithm

In some B$^+$-tree implementations, leaf nodes are not only chained left to right, but also right to left. The kNN search algorithm can exploit right to left sibling pointers to avoid always having to traverse the tree from the root when an interval is extended for a next iterative range search. This reduces the search cost but increases the update cost.

3.2.3 Continuous Queries

The queries considered so far in this section may be considered as one-time queries: they run once and complete when a result has been returned. Intuitively, a continuous query is a one-time query that is run at each point in time during a time interval. Further, a continuous query takes a now-relative time $now + \Delta t_q$ as a parameter instead of the fixed time t_q we have used so far. The query then maintains the result of the corresponding one-time query at time $now + \Delta t_q$ from when the query is issued at time t_{issue} and until it is deactivated.

Such a query can be supported by a query q_l with time interval $[t_{issue} + \Delta t_q, t_{issue} + \Delta t_q + l]$ ("l" is a time interval) [2]. Query q_l can be computed by the algorithms we have presented previously, with relatively minor modifications: (i) we use the end time of the time interval to perform forward enlargements, and we use the start time of the time

interval for backward enlargements; (ii) we store the answer sets during the time interval. Then, from time t_{issue} to $t_{issue} + l$, the answer to q_l is maintained during update operations. At $t_{issue} + l$, a new query with time interval $[t_{issue} + \Delta t_q + l, t_{issue} + \Delta t_q + 2l]$ is computed.

To maintain a continuous range query during updates, we simply add or remove the object from the answer set if the inserted or deleted object resides in the query window. Such operations only introduce CPU cost.

The maintenance of continuous kNN queries is somewhat more complex. Insertions also only introduce CPU cost: an inserted object is compared with the current answer set. Deletions of objects not in the answer set does not affect the query. However, if a deleted object is in the current answer set, the answer set is no longer valid. In this case, we issue a new query with a time interval of length l at the time of the deletion. If the deletion time is t_{del}, a query with time interval $[t_{del} + \Delta t_q, t_{del} + \Delta t_q + l]$ is triggered at t_{del}, and the answer set is maintained from t_{del} to $t_{del} + l$.

The choice of the "optimal" l value involves a trade-off between the cost of the computation of the query with the time interval and the cost of maintaining its result. On the one hand, we want to avoid a small l as this entails frequent recomputations of queries, which involves a substantial I/O cost. On the other hand, a large l introduces a substantial cost: Although computing one or a few queries is cost effective in itself, we must also take into account the cost of maintaining the larger answer set, which may generate additional I/Os on each update.

We note that maintenance of continuous range queries incur only CPU cost. Thus, we compute a range query with a relatively large l such that l is bounded by $\Delta t_{mu} - \Delta t_q$ since the answer set obtained at t_{issue} is no longer valid at $t_{issue} + \Delta t_{mu}$. For the continuous kNN queries, we examine the effect of l further in the experiments.

3.3 Update, Insertion, and Deletion

The insertion algorithm is straightforward. Given a new object, we calculate its index key according to Equation 1, and then insert it into the B^x-tree as in the B$^+$-tree. To delete an object, we assume that the positional information for the object used at its last insertion and the last insertion time are known. Then we calculate its index key and employ the same deletion algorithm as in the B$^+$-tree. Therefore, the B^x-tree directly inherits the good properties of the B$^+$-tree, and we expect efficient update performance.

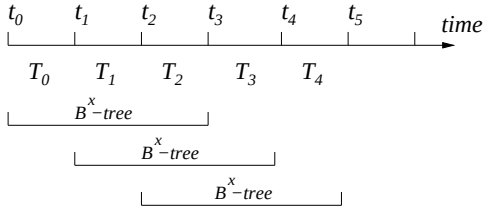


Figure 8: B^x-Tree Evolution

However, one should note that update in the B^x-tree does differ with respect to update in the B$^+$-tree. The B^x-tree only updates objects when their moving functions have been changed. This is realized by clustering updates during a certain period to one time point and maintaining several corresponding sub-trees. For example (see Figure 8), objects updated between t_0 and t_1 are stored in partition T_0; objects updated between t_1 and t_2 are stored in T_2; etc. T_0, T_1, and T_2 co-exist before t_3. From t_3 to t_4, T_1, T_2, and T_3 co-exist, and T_0 has expired. The total size of the three sub-trees is equal to that of one tree indexing all the objects.

In some applications, there may be some object positions that are updated relatively rarely. For example, most objects may be updated at least each 10 minutes, but a few objects are updated once a day. Instead of letting outliers force a large maximum update interval, we use a "maximum update interval" within which a high percentage of objects have been updated.

Object positions that are not updated within this interval are "flushed" to a new partition using their positions at the label timestamp of the new partition. In the example shown in Figure 8, suppose that some object positions in T_0 are not updated at the time when T_0 expires. At this time, we move these objects to T_2. Although this introduces additional update cost, the (controllable) amortized cost is expected to be very small since outliers are rare.

The forced movement of an object's position to a new partition does not cause any problem with respect to locating the object, since the new partition can be calculated based on the original update time. Likewise, the query efficiency is not affected.

4 Performance Studies

4.1 Experimental Settings

Two versions of the B^x-tree were implemented: B^x(Z-curve) and B^x(H-curve), denoting the B^x-tree using the Peano and the Hilbert curve, respectively. Both B^x-trees and the TPR-tree were implemented in C, and all the experiments were conducted on a 2.6G PentiumIV Personal Computer with 1 Gbyte of memory.

We use synthetic datasets of moving objects with positions in the space domain of 1000×1000. In most experiments, we use uniform data, where object positions are chosen randomly, where the objects move in a randomly chosen direction, and where a speed ranging from 0 to 3 is chosen at random. One may think of the unit of space being kilometer and the unit of speed being kilometer per minute.

Other datasets were generated using an existing data generator, where objects move in a network of two-way routes that connect a given number of uniformly distributed destinations [19]. Objects start at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Whenever an object reaches one of the destinations, it chooses the next target destination at random. Objects accelerate as they leave a destination, and they decelerate as they approach a destination.

For each dataset, we constructed the index at time 0, and measured the average query cost after the index ran for 10 time units. The parameters used are summarized in Table 1, where values in bold denote default values used.

Parameter	Setting
Page size	4K
Node capacity	200
Max update interval	120
Max predictive interval	60, **120**
Query window size	**10**, ..., 50, ..., 100
k (kNN query)	10, **20**, 30, 40, 50
Number of queries	200
Dataset size	100K, ..., **500K**, ..., 1M
Space-filling curve	Z, H
Dataset	Uniform, Network-based

Table 1: Parameters and Their Settings

4.2 Storage Requirement

Storage requirement is an important issue in moving object databases since some applications may choose to cache the whole index in main memory to improve performance. Figure 9 shows the storage requirement of both indexes, in which the B^x-trees require less storage space than the TPR-tree. The TPR-tree requires slightly more storage space as its fanout is slightly less than that of the B^x-tree.

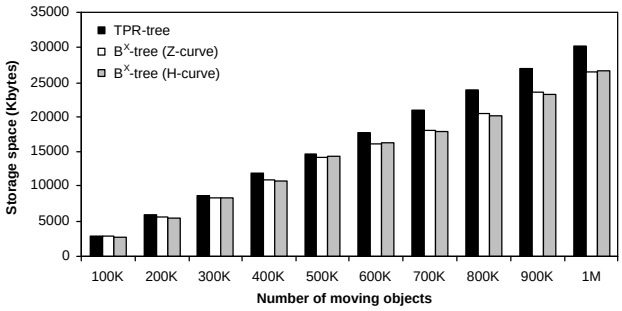


Figure 9: Storage Requirement

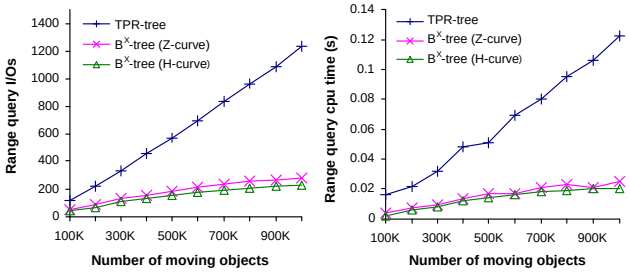
4.3 Range Query

4.3.1 Effect of Data Sizes

In the first set of experiments, we study the range query performance of the TPR-tree and the B^x-trees while varying the number of uniformly distributed moving objects from 100K to 1M. Figure 10 shows the average number of I/O operations and the CPU time per range query for each index.

We observe that both B^x-tree variants scale very well and maintain consistent performance, while the TPR-tree degrades linearly with the increase of the dataset size.

When the dataset reaches 1M objects, the B^x-trees are nearly 5 times better than the TPR-tree. This behavior may be explained as follows. In the B^x-trees, every object has a linear order, which is determined by the space domain and



(a) I/O (b) CPU time

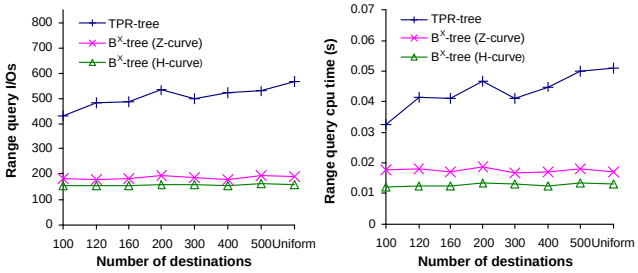
Figure 10: Effect of Data Sizes on Range Query Performance

is relatively independent of the number of moving objects. As the dataset grows, the range query cost of the B^x-trees increases mainly due to the increase of the number of objects inside the range. However, the structure of the TPR-tree is affected more by the dataset size. When the number of objects increases, the MBRs in the TPR-tree have higher probabilities of overlapping; this is consistent with earlier findings for the R-tree [18].

The B^x-tree(H-curve) achieves better performance than the B^x-tree(Z-curve) because the Hilbert curve generates a better distance-preserving mapping than the Peano curve, and hence yields fewer search intervals on the B^x-tree, i.e., less disk access.

4.3.2 Effect of Data Distribution

This experiment uses the road network dataset to study the effect of data distribution on the indexes. The dataset contains 500K data points. Figure 11 shows the range query cost when the number of destinations in the simulated network of routes is varied. The term "uniform" in the figure indicates the case where the objects can choose their moving directions freely.



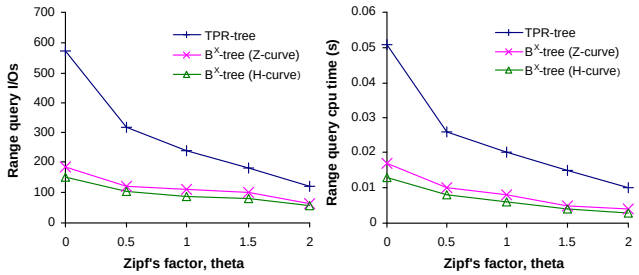
(a) I/O (b) CPU time

Figure 11: Effect of Data Distribution on Range Query Performance

Observe that the query cost in the TPR-tree increases slightly with the number of destinations, i.e., as the datasets becomes increasingly "uniform." This is consistent with previous results [19]. In contrast, the performance of the B^x-trees is not affected by the data skew because objects are stored using space-filling curves, meaning that the density has less of an effect on the index.

4.3.3 Effect of Speed Distribution

Figure 12 shows the effect of speed of moving objects on the TPR-tree and the B^x-trees, by varying the θ value of the Zipf distribution from 0 (uniform distribution) to 2 (skewed, 80% objects have speed lower than 20% of the maximum speed). All the indexes yield better performance when the number of fast moving objects decreases because MBRs in the TPR-tree obtain smaller expanding speeds and because the enlargements made to query windows for the B^x-trees also become smaller. The results for kNN queries exhibit similar performance trends, so we omit the results due to space constraints.

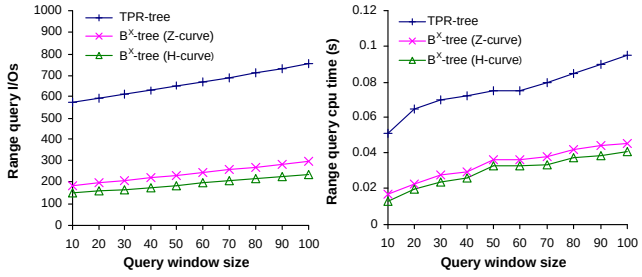


(a) I/O (b) CPU time

Figure 12: Effect of Speed on Range Query Performance

4.3.4 Effect of Query Window Sizes

We next study the effect of the query window size, varying the window length from 10 to 100 for a dataset of size 500K. As expected, the result in Figure 13 shows that the query cost increases with the query window size. Larger windows contain more objects and therefore lead to more node accesses, and the effect is slightly more obvious on the TPR-tree.

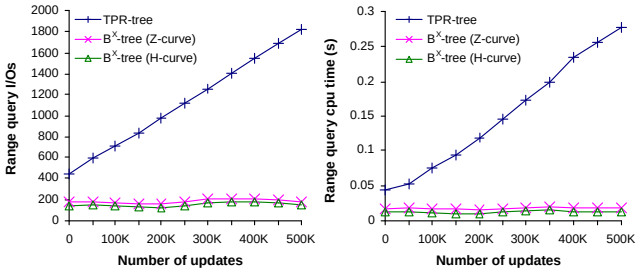


(a) I/O (b) CPU time

Figure 13: Effect of Query Window Sizes on Range Query Performance

4.3.5 Effect of Time

To study the search performance of the indexes with the passage of time and updates, we compute the query cost using the same 200 range queries with query window size 50, but after every 50K updates in a 500K dataset. Figure 14 summarizes the results, showing that the TPR-tree degrades considerably faster than the B^x-trees due to continuous enlargements of the MBRs which are not updated



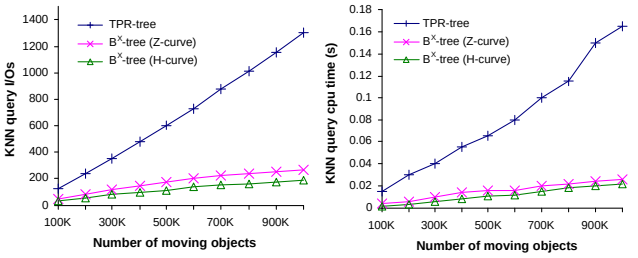
(a) I/O (b) CPU time

Figure 14: Effect of Time Elapsed on Range Query Performance

as time passes. In contrast, the B^x-tree structure is not affected as much by the updates. In fact, the B^x-trees are almost time independent.

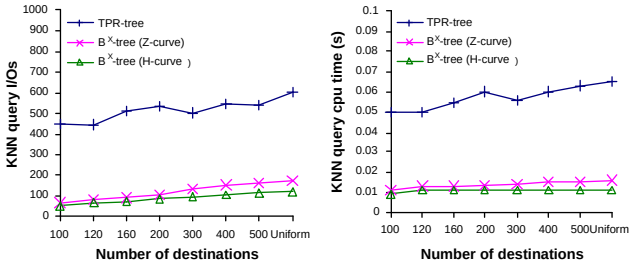
4.4 kNN Query

We proceed to evaluate the efficiency of kNN queries using the same settings as for range queries. Figures 15–17 show in turn the effect of dataset size, data distribution, and time passed on kNN query performance. The performance difference between the TPR-tree and the B^x-tree of the kNN queries exhibits a behavior similar to that of range queries. The B^x-tree's kNN search algorithm is essentially an incremental range query algorithm; hence, the results exhibit similar patterns as the results for range queries.



(a) I/O (b) CPU time

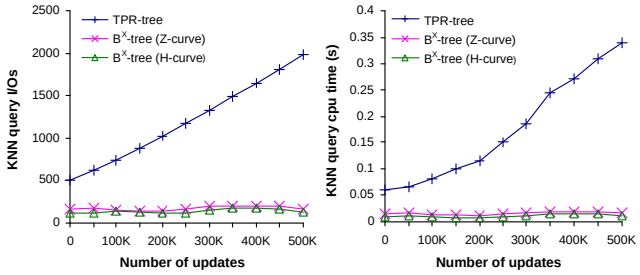
Figure 15: Effect of Data Sizes on kNN Query Performance



(a) I/O (b) CPU time

Figure 16: Effect of Data Distribution on kNN Query Performance

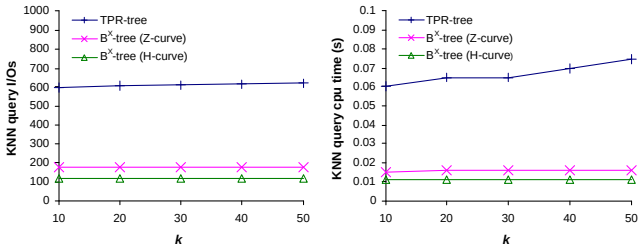
Figure 18 shows the effect on performance of the number k of required nearest neighbors. As k increases, the search and CPU costs increase slightly for both indexes.



(a) I/O (b) CPU time

Figure 17: Effect of Time Elapsed on kNN Query Performance

Due to the data size and side effect of the query and MBR enlargement, the effect of k is not significant.

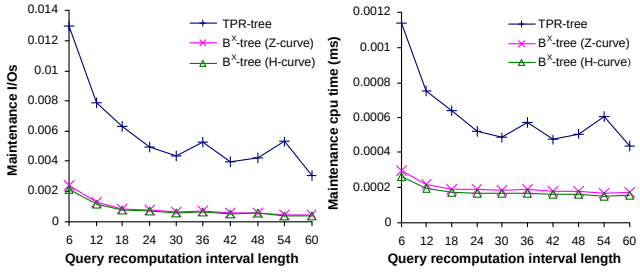


(a) I/O (b) CPU time

Figure 18: Effect of k on kNN Query Performance

4.5 Continuous Range and kNN Queries

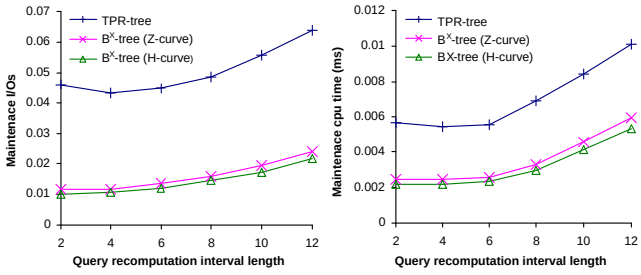
In moving object database applications, continuous queries are expected to be common, and hence efficient support for such queries is important. To investigate the maintenance cost of continuous queries, we perform a series of experiments where we vary the length of the query recomputation interval l. We evaluate the amortized cost per single update operation (insertion or deletion) in maintaining one continuous query. Indexes were created at time 0, and after running 10 time units, 200 queries (with maximum predictive interval 60) were issued. Then the workload was run for another 120 time units while maintaining the result set of the queries.



(a) I/O (b) CPU time

Figure 19: Maintenance Cost of Continuous Range Query

Figure 19 shows the continuous range query performance. Since the maximum predictive length is 60 and the maximum update interval is 120, the maximum recomputation interval tested is $120 - 60 = 60$. As can be observed, the maintenance cost decreases with the increase of the recomputation length in all indexes. This is because the cost to maintain the answer set under continuous updates is very small, and the recomputation cost constitutes the major I/O cost. The smaller the recomputation interval l, the more number of recomputations.



(a) I/O (b) CPU time

Figure 20: Maintenance Cost of Continuous kNN Query

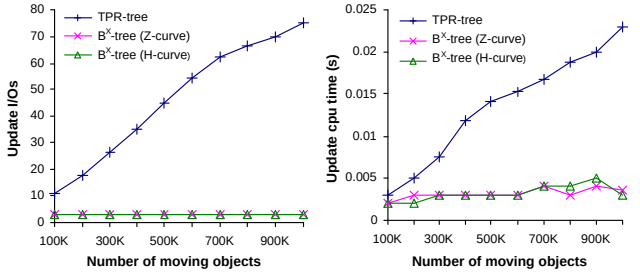
Figure 20 shows the performance of the continuous kNN query. We observe that, for all the indexes, the maintenance cost first decreases until a point before it increases again. The best l is approximately 4 for the TPR-tree and approximately 3 for the B^x-trees. As the l becomes larger, the number of recomputation decreases, however, the possibility to remove objects from the results increases, and consequently, additional recomputations result.

4.6 Update

We compare the average update cost (amortized over insertion and deletion) of the B^x-trees against the TPR-tree. Note that for each update, one deletion and one insertion are issued, leaving the size of the tree unchanged.

4.6.1 Effect of Data Sizes

First we examine the update performance with respect to dataset size. We compute the average update cost after the maximum update interval of 120 time units. From Figure 21, we can see that the B^x-trees achieve significant improvement over the TPR-tree. In most cases, one update



(a) I/O (b) CPU time

Figure 21: Effect of Data Sizes on Update Cost

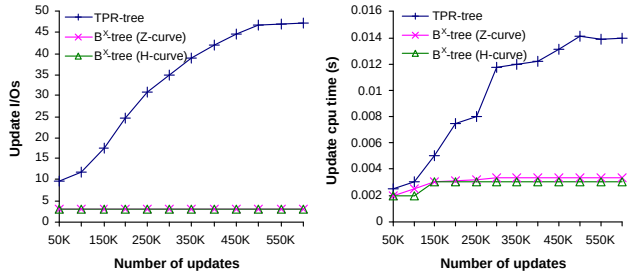
in the B^x-tree only incurs several I/O operations. However, the update cost in the TPR-tree increases significantly as

the dataset grows in size. This is because in the B^x-trees, given the key, an insertion and a deletion needs to traverse only one path, no matter how large the dataset is. Thus, the cost of update in the B^x-tree is only related to the height of the tree.

In this experiment, the performance of the B^x-tree(Z-curve) and the B^x-tree(H-curve) are comparable, since the update efficiency is independent of the spatial proximity preservation. However, in the TPR-tree, traversal of multiple paths is inevitable due to the overlaps among MBRs. As the density of values increases due to the increase in data size, more overlap and hence higher update cost results.

4.6.2 Effect of Time

Next, we investigate performance degradation across time. We measure the performance of the TPR-tree and the B^x-trees after every 50K updates. Figure 22 shows the update cost as a function of the number of updates. As before, the



(a) I/O (b) CPU time

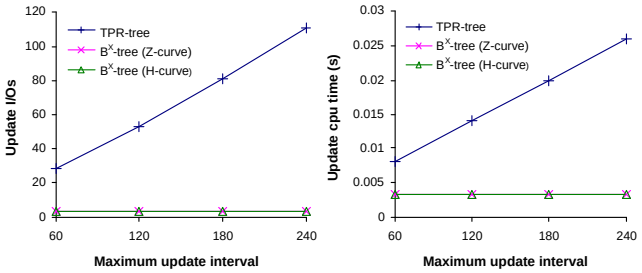
Figure 22: Effect of the Number of Updates on Update Cost

B^x-trees are not affected by time, which again illustrates the efficiency and feasibility of the B^+-tree. We observe that the gap between the TPR-tree and the B^x-trees widens as time passes. At the point when the TPR-tree stabilizes after 500K updates, the cost of the TPR-tree is nearly 10 times that of the B^x-trees. The reason for the degeneration of the TPR-tree is that each deletion entails a search to retrieve the object to be removed, and the cost of this search increases with the number of updates.

We note that this cost can be reduced by maintaining a hash-table for quickly locating the object and then performing a bottom-up update [10]. However, such an auxiliary structure incurs additional storage overhead and increases complexity.

4.6.3 Effect of Update Interval Length

In this experiment, we investigate the effect of maximum update interval length on the indexes, by varying the maximum update interval from 60 to 240. Figure 23 shows the average update cost after the indexes run for one maximum update interval. We observe that the performance of the TPR-tree degrades fairly quickly as the maximum update interval increases, whereas the B^x-trees are not affected. The main reason is that, as the update interval increases, the overlap among MBRs becomes more severe and thus



(a) I/O (b) CPU time

Figure 23: Effect of Maximum Update Interval

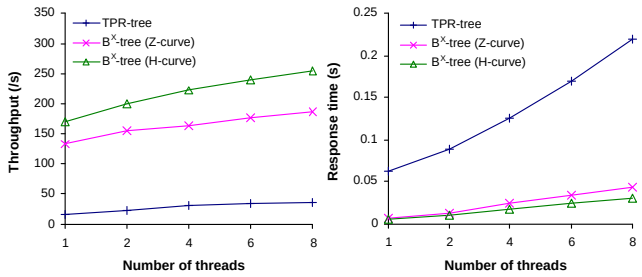
affects the performance of the TPR-tree significantly. In contrast, the update operation in the B^x-tree depends only on the key value which does not change over time.

4.7 Effect of Concurrent Accesses and Buffer Space

In this section, we compare the concurrent performance of the TPR-tree and the B^x-tree. We implemented the R-link technique [8] for the TPR-tree and the B-link technique [21] for the B^x-tree.

We used multi-thread programs to simulate multi-user environments. The number of threads varies from 1 to 8. Workloads contain an equal number of queries and updates. We investigated the throughput and response time of search and update operations. Throughput is the rate at which operations could be served by the system. Response time is the time interval between issuing an operation and getting the response from the system when the task was successfully completed.

Figure 24 shows throughput and response time for the three indexes. The throughputs of the B^x-trees are much



(a) Throughput (b) Response time

Figure 24: Effect of Concurrent Operations

higher than that of the TPR-tree, and the response times of the B^x-trees are always less than those of the TPR-tree. The main reason is that the B^x-trees seldom lock internal nodes. Recall that, in the query processing, we will first travel down to the leaf level, then retrieve the leaf nodes for the answers by following the left-to-right sibling links. We may occasionally ascend to an internal node for a "jump," but this often happens at the lower levels of the index. For the TPR-tree, a query triggers searching of multiple paths, which introduces locks on the internal nodes that reduce the parallelism of concurrent operations.

We also included an LRU buffer and studied the effects of different buffer sizes. As with other indexes, both indexes experience reduced I/O's as the buffer size increases. Since the B^x-tree incurs less page reads originally, the effect of an increasing buffer size on the index is consequently less pronounced than for the TPR-tree. We also examined the effects of the density of objects over the data space. The results are as may be expected; due to space constraints, we do not include the graphs here.

5 Conclusion and Research Directions

Database applications that entail the storage of samples of continuous, multi-dimensional variables pose new challenges to database technology. This paper addresses the challenge of providing support for indexing that is efficient for querying as well as update.

We proposed a new indexing scheme, the B^x-tree, which is based on the B^+-tree. This scheme uses a new linearization technique that exploits the volatility of the data values, i.e., moving-object locations, being indexed. Algorithms are provided for range and kNN queries on the current or near-future positions of the indexed objects, as well as for so-called continuous counterparts of these types of queries. Queries that reach into the future are handled via query region enlargement, as opposed to the MBR enlargement used in TPR-trees.

Extensive performance studies were conducted that indicate that the B^x-tree is both efficient and robust. In fact, it is capable of outperforming the TPR-tree by factors of as much as 10. Further, being a B^+-tree index, the B^x-tree may be grafted into existing database systems cost effectively.

Several promising directions for future work exist, one being to consider the use of the B^x-tree for the processing of new kinds of queries. Another is the use of the B^x-tree for other continuous variables than the positions of mobile service users. Yet another direction is to apply the linearization technique to other index structures.

Acknowledgements

The work of D. Lin and B. C. Ooi was in part funded by an A*STAR project on spatial-temporal databases. In addition to his primary affiliation with Aalborg University, C. S. Jensen is an adjunct professor in Department of Technology, Agder University College, Norway. His work was funded in part by grants 216 and 333 from the Danish National Center for IT Research.

References

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

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proc. IDEAS*, pp. 44–53, 2002.

[3] A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. In *Proc. MobiQuitous*, 2004, 10 pages, to appear.

[4] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *Proc. PODS*, pp. 247–252, 1989.

[5] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[6] C. S. Jensen and S. Saltenis. Towards Increasingly Update Efficient Moving-Object Indexing. *IEEE Data Eng. Bull.*, 25(2): 35–40, 2002.

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

[8] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. In *Proc. VLDB*, pp. 134–145, 1995.

[9] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree. In *Proc. MDM*, pp. 113–120, 2002.

[10] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *Proc. VLDB*, pp. 608–619, 2003.

[11] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull.*, 26(2): 40–49, 2003.

[12] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TKDE*, 13(1): 124–141, 2001.

[13] B. C. Ooi, K. L. Tan, and C. Yu. Fast Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes. In *Proc. of Int. Web GIS Workshop, Keynote*, 2002.

[14] J. M. Patel, Y. Chen and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *Proc. ACM SIGMOD*, 2004, to appear.

[15] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proc. VLDB*, pp. 395–406, 2000.

[16] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. Star-Tree: An Efficient Self-Adjusting Index for Moving Objects. In *Proc. ALENEX*, pp. 178–193, 2002.

[17] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-Tree Into a Database System Kernel. In *Proc. VLDB*, pp. 263–272, 2000.

[18] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In *Proc. ACM SIGMOD*, pp. 17–31, 1985.

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

[20] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comp. Surv.*, 16(2): 187–260, 1984.

[21] V. Srinivasan and M. J. Carey. Performance of B-Tree Concurrency Control Algorithms. In *the Proc. ACM SIGMOD*, pp. 416–425, 1991.

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

[23] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces. *IEEE TKDE*, to appear.

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

[25] C. Yu, B. C. Ooi, K. L. Tan and H. V. Jagadish. Indexing the Distance: An Efficient Method to KNN Processing. In *Proc. VLDB*, pp. 421–430, 2001.