STRIPES: An Efficient Index for Predicted Trajectories

Jignesh M. Patel Yun Chen V. Prasad Chakka
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109, USA
{jignesh, yunc, vchakkab}@eecs.umich.edu

ABSTRACT

Moving object databases are required to support queries on a large number of continuously moving objects. A key requirement for indexing methods in this domain is to efficiently support both update and query operations. Previous work on indexing such databases can be broadly divided into two categories: indexing the past positions and indexing the future predicted positions. In this paper we focus on an efficient indexing method for indexing the future positions of moving objects.

In this paper we propose an indexing method, called STRIPES, which indexes *predicted* trajectories in a dual transformed space. Trajectories for objects in *d*-dimensional space become points in a higher-dimensional *2d*-space. This dual transformed space is then indexed using a regular hierarchical grid decomposition indexing structure. STRIPES can evaluate a range of queries including time-slice, window, and moving queries. We have carried out extensive experimental evaluation comparing the performance of STRIPES with the best known existing predicted trajectory index (the TPR*-tree), and show that our approach is significantly faster than TPR*-tree for both updates and search queries.

1. INTRODUCTION

Over the last decade, we have witnessed an increasing interest in techniques for managing databases consisting of a large number of continuously moving objects. These research interests have been fuelled by rapid advances in hardware technologies that allow for cheap location-aware devices, which are often packaged in small physical devices. These devices have found applications in a variety of civilian and military monitoring applications. Many of these applications demand extremely efficient techniques for dealing with a large update rate triggered by objects continuously updating their location information. In addition, these applications also require efficient techniques for querying on the location information. Queries in these applications can be divided into two broad classes: queries on the past positions of moving objects, and queries on *predicted* positions of the moving objects. In this paper, we focus on this latter class of queries, which are often referred to as *predictive* queries. While there are a number of proposals for modeling the predicted positions of moving objects, the most widely-used model specifies the predicted position as a function of the current position and a velocity vector indicating the direction of the future motion [11, 15, 22-24, 29]. Our paper also uses this commonly-used model.

SIGMOD 2004, June 13–18, 2004, Paris, France.

To efficiently answer queries on predicted position, it is intuitive to ask the question if effective indexing methods can be built on these moving object data sets. Naturally, a substantial amount of research has been undertaken in the recent past to answer this question. Some of the early work in this area employs dual transformation techniques [1, 15, 37]. These techniques typically represent the predicted position of an object moving in a *d*-dimensional space as a point in a *2d*-dimensional space. Most of this body of work is largely theoretical in nature, and for most parts focuses on objects moving in a one-dimensional space [15]. More recent work in this area has focused primarily on *practical* implementations of indexing structures for predictive queries [6, 25-29, 34]. Of these indexing methods, perhaps the most influential indexing method is the TPR-tree [29]. This indexing structure uses the basic R*-tree indexing structure [2], and expands the traditional definition of bounding boxes to include *time-parameterized* bounding boxes. Essentially each bounding box now has an associated velocity vector that captures the growth of the box as time progresses. The TPR-tree has inspired a flurry of research aimed at improving the basic TPR-tree algorithms. A recently proposed indexing structure, called the TPR*-tree [34] has been shown to vastly outperform the basic TPR-tree index.

Surprisingly, in the more empirical research on predictive trajectory indexing, the early dual transformed methods have been largely dismissed. (The TPR-tree [29] employs techniques that are inspired by the dual transformed methods, but doesn't actually experimentally compare the TPR-tree index with any dual transform-based methods.) Perhaps the reasons for this dismissal are: a) the research in dual transform indexing methods has largely focused on deriving asymptotic performance bounds, and b) these researchers have suggested that the dual transformed space be indexed using methods such as partition trees [15], which are not as widely used as R-trees in practice. The body of empirical research has largely dismissed these theoretical results arguing that the asymptotic performance bounds, though interesting, don't lead to practical structures since these bounds have large constant factors [29, 34].

In this paper, we reexamine this issue and consider if a practical indexing structure can be built using a dual transformation technique. The motivation for our interest in this approach stems from the observation that R-tree based indexing techniques are known to rapidly degrade in performance as the dimensionality of

the underlying space increases [4]. All the TPR-tree based indexing structures use time-parameterized boxes, which require representing a velocity value for each dimension. In effect, the underlying indexing space can be viewed as a *2d*-dimensionsal space for objects moving in a *d*-dimensional space. The dual transform techniques also need to index in a *2d*-dimensional space. However, rather than indexing boxes the dual transform techniques only have to index points, which is potentially easier to index efficiently. This key insight is at the heart of the STRIPES indexing structure, which we propose in this paper.

STRIPES is a Scalable Trajectory Index for Predicted Positions in Moving Object Databases. The STRIPES index maps predicted positions to points in a dual transformed space and indexes this space using a disjoint regular partitioning of space. This style of partition is called quadtrees, and can be extremely efficient, especially for indexing multi-dimensional point data [30]. Even though the traditional quadtree leads to an unbalanced tree, it has proven to be an effective disk-based indexing structure in some cases [10, 13]. STRIPES essentially employs a disk-based bucket PR quadtree structure [30]. STRIPES can evaluate the entire range of predictive queries, which include time-slice, window, and moving queries [29].

In this paper, we compare the performance of STRIPES with the currently best known indexing structure, namely the TPR*-tree[34]. Using actual implementations of these two indices, we demonstrate that STRIPES outperforms the TPR*-tree for both updates and query operations. In most cases, updates in STRIPES are more than an order of magnitude faster that the TPR*-tree, and queries are about 4x faster with STRIPES.

This paper makes an important contribution which includes proposing a new indexing structure that is extremely efficient for predictive queries. In addition, we have essentially come to a full circle with this work, where we now show that the intuition behind the earlier theoretical work on dual transform-based techniques can indeed be leveraged to produce a practical and efficient predicted trajectory indexing method.

The remainder of this paper is organized as follows: In Section 2, we cover the model used for representing predicted positions. Section 3 describes the TPR-tree and the TPR*-tree indices. The STRIPES index is described in Section 4, with experimental results in Section 5. Related work is reviewed in Section 6, and we present our conclusions and plans for future work in Section 7.

2. BACKGROUND AND MODEL

Location data for moving objects is continuously changing between any two successive updates of the location of the mobile object. This poses a problem in representing the location of the object at all times because most conventional models for data representations are static in nature. A commonly used model for representing trajectory data approximates the motion of an object as a straight line segment between two consecutive updates [11, 15, 22-24, 29]. The same linear model is used for predicting future trajectories as well [29, 34]. The object is assumed to move with some specified current velocity from the current position until a new update is explicitly issued. If the current position and velocity of an object is represented as $(\overline{p}(t), \overline{v}(t))$ at time *t*, then the position at time *t' (t'>t)* can be calculated using $\overline{p}(t) = \overline{p}(t') + \overline{v}(t) \times (t - t')$. When the actual update arrives, which can be different from the predicted position, the velocity and the current positions are updated in the index to reflect the new predicted trajectory.

As has been noted in previous studies, when indexing predicted trajectories an optimal packing of a group of objects into a node in the index at time *t* is unlikely to be optimal at a later time *t'*. Consequently, an index that is optimal for queries at *t* will not be optimal at time *t'*, and the index performance gradually deteriorates as *t'* increases. To reduce the dramatic performance degradation of an index built at a long time in the past, the trajectory indexing mechanisms often employ the notion of an index *lifetime* [29]. The lifetime is the time interval for which the index is designed to give good performance. After this time interval the performance of the index is likely to deteriorate. The index is rebuilt periodically to avoid such rapid deterioration.

Another practical observation is that for an object moving in a *d*-dimensional space, the predicted trajectory includes a *d*-dimensional current spatial coordinate, and a *d*-dimensional velocity vector. Consequently indexing predicted trajectories requires indexing these two *d*-dimensional entities, which essentially requires indexing entities in *2d*-dimensional space. The so called curse of dimensionality [3], and the related challenges with query evaluation methods in high-dimensional space [4, 5] quickly start becoming performance issues in this domain. In addition, since the optimal node for a new update can be different from the node containing the old representation for the object, updating the predicted trajectory of an index will often result in traversing multiple paths down an index.

2.1 Query Types

There are three classes of future queries that have been extensively used in the previous research for querying on predicted trajectories [29]. These three classes are time-slice query, window query, and moving query. For a one-dimensional space, Figure 1 shows one example for each of these query types. In this figure the x-axis represents the time dimension, and the y-axis represents the single spatial dimension.

In Figure 1, Q1 is a time-slice query, which finds all objects at some specified future time *t* in some spatial region *R*. Q2 is a window query for finding all objects in time window [*t, t'*] in region *R*. Q3 is a moving query to find all objects in time window [*t, t'*] in region *R* that is moving with velocity *v*.

3. TPR AND TPR*-TREE INDICES

In this section we review the two popular predicted indices, namely the TPR-tree index [29] and the TPR*-tree index [34].

3.1 TPR-Tree

The TPR-tree [29] is essentially a time parameterized R*-tree. The index stores velocities of the elements along with their positions in nodes. Since the elements are not static, the corresponding MBRs are dynamic (see Figure 2).

The index structure as well as the algorithms for search, insert and delete used are very similar to that of R*-tree[2]. The R*-tree uses a number of static parameters such as the area, perimeter, distance from the centroid, and the intersection between the two MBRs. The TPR-tree uses time parameterized metrics for these parameters. The time parameterized metric is computed using the

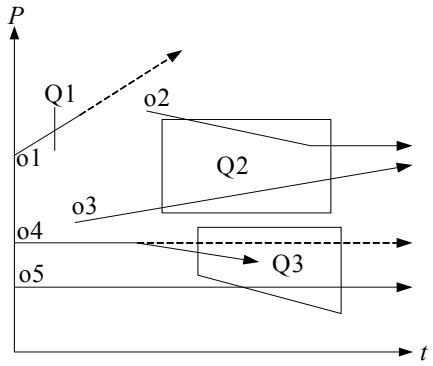

Figure 1: Query examples for objects moving in a one-dimensional space.

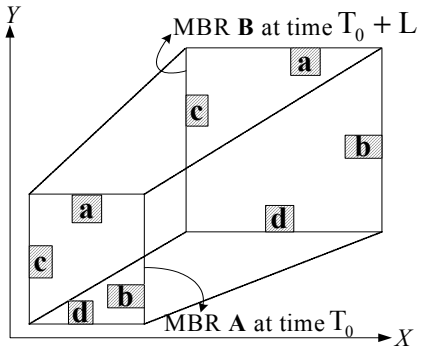

Figure 2: Area computation in the TPR-tree.

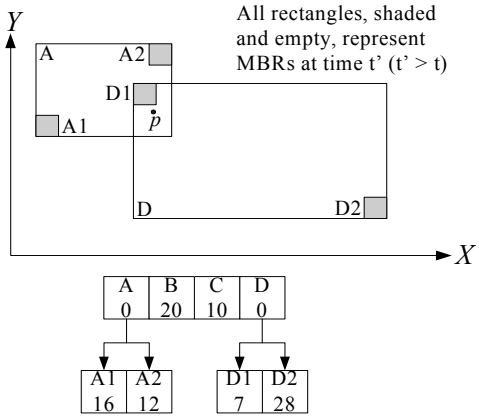

Figure 3: Motivation for the insert algorithm in TPR*-tree.

formula $\int_{T_0}^{T_0+L} M(t)dt$, where $M(t)$ is some metric that is used in the original R*-tree (for example the area), and L is the *lifetime* (see Section 2) of the index. The lifetime of an index is the time for which the index is used andqueried.

Figure 2 shows an example of the time-parameterized area metric for four objects a, b, c, and d, moving in a two-dimensional space. In this figure, the actual data objects are shown as shaded boxes. The MBR of the index node at time T_0 is labeled as **A**, and the MBR of the node at time T_0+L is labeled as **B**. The size and the position of B are calculated by extrapolating (position, velocity) of the entries with in the node. Then the area metric used is the volume of the trapezoid that is formed by the moving MBR of the node from time T_0 to T_0+L. All the other metrics are computed in a similar fashion.

The insert algorithm chooses a node such that the expansion in volume is the smallest at non-leaf nodes and the expansion in integrated perimeter is the smallest at the leaf node level. When such a node is full, it is split similar to R*-tree. Now instead of just sorting boundaries of elements, the velocity vectors are also sorted to choose the best distribution of the elements.

The TPR-tree inherits all the problems related to the R*-tree, such as overlap and dead space. Since the positions and the velocities are estimated and can change, the optimal combination of elements can not be maintained at all times in the future.

3.2 TPR*-Tree

The recently proposed TPR*-tree [34] provides a number of optimization over the basic TPR-tree algorithms. The key observation made by the authors is that during an insertion operation making a choice based on a local optimization may lead to a poor performing predicted trajectory index. To illustrate this key insight, consider the example shown in Figure 3. This figure shows a number of MBRs in the TPR-tree at a given time for objects moving in a two-dimensional space. In this example, the point p is being inserted into the index.

In an R-tree based insert algorithm, a least deterioration cost node is chosen for inserting the point p. In Figure 3, at the top level, the least deterioration cost is for nodes A and D. The greedy algorithm in TPR-tree will pick node A because it requires the least area and perimeter expansion. And so at the level 2, node A2 will be picked for inserting the point p. However, the overall optimal node for this insertion is node D1, which is the descendant of node D. The insert algorithm of TPR*-tree recognizes that a local optimal solution at a level (node A in the example shown in Figure 3) can be from a broken-tie resulting from two elements having the same deterioration, and that the sub-elements (node A2 in this example) of that element may not be optimal. It proposes a novel *ChoosePath* algorithm that determines the node at any level that has the least cost of deterioration. It maintains a priority queue that is ordered by the cost of deterioration for each node, and the node with the least cost is picked for traversal at each step of the algorithm. The traversal continues until the bottom-most non-leaf node that has the least cost is found. This node is then chosen as the candidate for insertion. The authors argue that the extra cost incurred in traversing can be offset by the benefits of finding an optimal node for insertion. This algorithm leads to a tighter packing of elements in nodes and thus better query and insert performance.

The algorithm to deal with overflow nodes in the TPR*-tree is to first force reinsert and then split the node, if necessary. For objects moving in a two-dimensional space, the nodes are first sorted along all the eight (4×d) possible dimensions and the first λ (=30%) entries from the best possible sort are chosen for reinsertion. If during the reinsertion, a node overflows then the node is split. The authors propose a heuristic to reduce the number of sorts to just one, by recognizing that the elements at leaf nodes can be assumed to be uniformly distributed, and the largest extent of all the dimensions (positions and velocities) would give the best benefit.

The TPR*-tree authors also propose a cost model, and a hypothetical optimal tree for predictive indices using a TPR-tree style of indexing. They show that the performance of the TPR*-tree is very close to the optimal index. Consequently, one can conclude that the TPR*-tree is currently the best known practical indexing technique for predicted trajectories.

4. STRIPES

In this section, we introduce the STRIPES index. To facilitate our discussion, we will use the notations described in Table 1.

Table 1. Notations

Notation	Description
d	Number of dimensions in real space.
D	Number of dimensions in dual transformed space ($D = 2d$).
L	Index lifetime.
t^{ref}	Reference time, a.k.a. initialization time of an index.
$v_i^{\max}$	Maximum absolute velocity value in i^{th} dimension.
$\overline{v^{\max}}$	Vector of maximum velocities.
$p_i^{\max}$	Maximum position value of a moving object in the i^{th} dimension. In the i^{th} dimension the position of an object ranges between 0 and $p_i^{\max}$.
$\overline{p^{\max}}$	Vector defining the dimensions of the physical space.
p_i	Position of an object in the i^{th} dimension in original space.
$\overline{p}$	Position vector of an object in original space.
v_i	Velocity of an object in the i^{th} dimension of original space.
$\overline{v}$	Velocity vector of an object in original space.
p_i^{ref}	Reference position of an object at time t^{ref} in i^{th} dimension of original space.
$\overline{p^{ref}}$	Reference position vector of an object in original space at time t^{ref}.
P_i^{ref}	Reference position of an object in i^{th} plane of transformed dual space.
$\overline{P^{ref}}$	Reference position vector of an object in dual transformed space.
$\overline{P}$	Position vector of an object in dual transformed space.
$\overline{V}$	Velocity vector of an object in dual transformed space.
f	Non-leaf node fanout, $f = 2^D$.

4.1 Dual Transform for Moving Objects

The STRIPES index represents the moving object in a dual transformed space. The basic idea of a dual transform technique for predictive queries [1, 15] is to transform a linear trajectory defined by equation $\overline{p} = \overline{p^{ref}} + \overline{v}(t - t^{ref})$ in d+1-dimensional space (t being the additional dimension) into a point $(\overline{V}, \overline{P^{ref}})$ in $2d$-dimensional dual space. Here, $\overline{V} = (V_1, V_2, \cdots, V_d)$, and $\overline{P^{ref}} = (P_1^{ref}, P_2^{ref}, \cdots, P_d^{ref})$ are the transformed velocity and reference position vectors. We incorporate both negative and positive values for velocity by applying the following transform: Given $(\overline{v}, \overline{p})$, the velocity and position vectors of an object, the corresponding transformed velocity and reference position vectors $(\overline{V}, \overline{P^{ref}})$ are calculated as follows:

$$\overline{V} = \overline{v} + \overline{v^{\max}}$$
$$\overline{P^{ref}} = \overline{p} - (\overline{V} - \overline{v^{\max}})(t - t^{ref})$$

Thus the range for $\overline{V}$ is $[\overline{0}, 2\overline{v^{\max}}]$, and the range for $\overline{P^{ref}}$ is $[-\overline{v^{\max}} \times (t - t^{ref}), \overline{p^{\max}} + \overline{v^{\max}} \times (t - t^{ref})]$.

Since time is monotonically increasing, the value of P_i^{ref} is not bounded, which makes building an index that extends into infinity impossible. To solve this problem, we use the same technique that has been used in previous works [15, 29, 34], namely requiring that objects periodically issue an update to maintain a valid entry in the index. This time period is essentially the lifetime L of the index. As in previous works [15, 29, 34], we also employ a two-index strategy where we keep two distinct index structures in the system. The first index covers the time range from 0 to L, and the second index covers the time range from L to $2L$ The reference time of the first index is $t_1^{ref} = 0$ and the reference time for the second index is $t_2^{ref} = L$. Since an update consists of the deletion of the old entry and the insertion of the new entry, when an update with timestamp $> 2L$ arrives, we can simply delete the entries in the first index (either it is empty or the entries in that index have *expired* their lifetime [15, 29, 34]). At this point, we clear the first index structure and update its t^{ref} to $2L$. New updates with timestamps in the range $[2L, 3L]$ are now inserted into this index. Using this strategy, we can observe that the range for $\overline{P^{ref}}$ in each of the indexes is $[-\overline{v^{\max}} \times L, \overline{p^{\max}} + \overline{v^{\max}} \times L]$. To simplify the computation of index entry coordinates, we add $\overline{v^{\max}} \times L$ to $\overline{P^{ref}}$ at transform time, and convert the range to $[\overline{0}, \overline{p^{\max}} + 2 \times \overline{v^{\max}} \times L]$. Thus, the transform equation becomes:

$$\overline{P^{ref}} = \overline{p} - (\overline{V} - \overline{v^{\max}})(t - \overline{t^{ref}}) + \overline{v^{\max}} \times L$$

And, the linear motion equation becomes:

$$\overline{p} = \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t - t^{ref}) - \overline{v^{\max}} \times L$$

4.2 Index Structure

The STRIPES index is essentially a disk-based multi-dimensional PR bucket quadtree. Each of the d dual planes, $\{(V_1, P_1), (V_2, P_2), \ldots, (V_d, P_d)\}$, are equally partitioned into four *quads*. This partitioning results in a total of $4^d = 2^{2d}$ partitions, which we call *grids*. The fanout of non-leaf nodes f is thus 2^{2d}.

A non-leaf node stores the following information:

1. *level*: indicating the level of the non-leaf node.
2. *grid*: which encodes information about the quadrant corresponding to this node, in each of the d dual planes. In our implementation we simply indicate the quadrant by the lower vertex of the quadrant (this increases storage cost, but reduces computation time).
3. *children pointer array*: an array of 2^{2d} children pointers
4. *isLeaf array*: a vector of length 2^{2d} indicating whether each of the 2^{2d} children pointers points to a leaf or a non-leaf node.
5. *size*: indicating total number of actual data entries stored in all the leaf nodes in the subtree below this non-leaf node.

Leaf nodes store the *level*, *grid*, and *size* information, and the set of points that are being stored in the leaf node.

We note that the grids consist of a series of d quads from the d two-dimensional planes (i.e. the planes $(V_1, P_1), (V_2, P_2), \ldots, (V_d, P_d)$). Thus each grid is uniquely defined by the tuple $\left(\overline{V}, \overline{P^{ref}}, \overline{SL^V}, \overline{SL^P}\right)$, where $\overline{V}$ ($\overline{P^{ref}}$) is the vector of velocity (reference position) coordinates of the leftmost (lowest) vertex of the d quads, and $\overline{SL^V}$ ($\overline{SL^P}$) is the vector of side lengths along the velocity (reference position) axis of the d quads.

4.3 Insertion

Being a dynamic index structure, STRIPES allows the insertion of objects on the fly.

We first discuss the algorithm used to find the target leaf node given an object to insert into the index.

Given the tuple $(\overline{v}, \overline{p})$ of a moving object, we first obtain transformed $(\overline{V}, \overline{P^{ref}})$ tuple using the transform algorithm discussed in section 4.1. Then starting from the root node, we recursively identify the next level target node by calculating its array index in the *children pointer array* using the following formula:

$$array\ index = \sum_{D=1}^{d}\left(2^{2D-1}\left(\left\lceil \frac{P_D^{ref} - P'^{ref}_D}{SL_D^P/2} \right\rceil - 1\right) + 2^{2D-2}\left(\left\lceil \frac{V_D - V'_D}{SL_D^V/2} \right\rceil - 1\right)\right) \quad \textbf{Eq. 1}$$

where V'_D, P'^{ref}_D, SL_D^V, and SL_D^P are the velocity, reference position, and side length parameters along the velocity and reference position axes of the *grid* of the current node in the D^{th} dual plane. Equation 1 is essentially an ordering mechanism for the children of a non-leaf node that assigns array indices based on the children's coordinates.

The recursion terminates when either of the following two cases occurs: i) the target leaf node is non-existent; ii) the target leaf node is found. Since for case ii) there are two sub-cases considering whether the leaf node is full or not, we end up having to consider the following three cases during an insert operation:

Case 1: the target leaf node is non-existent.

Case 2: the target leaf node is found and not full.

Case 3: the target leaf node is found and is full.

Next, we discuss each of these cases in turn.

In case 1, a new leaf node is created, and the new entry is inserted into this node. The grid parameters for the new leaf node are determined as follows:

$$\overline{SL'^V} = \overline{SL^V}/2$$

$$\overline{SL'^P} = \overline{SL^P}/2$$

$$\overline{P'^{ref}} = \left(\left\lceil \frac{\overline{P^{ref}}}{\overline{SL'^P}} \right\rceil - \overline{1}\right) \times \overline{SL'^P}$$

$$\overline{V'} = \left(\left\lceil \frac{\overline{V}}{\overline{SL'^V}} \right\rceil - \overline{1}\right) \times \overline{SL'^V}$$

(Note: Multiplications and divisions between vectors in the above equation imply element-wise operations.)

In the above equations, $\left(\overline{V'}, \overline{P'^{ref}}, \overline{SL'^V}, \overline{SL'^P}\right)$ are the parameters of the newly created leaf node; the vectors $(\overline{V}, \overline{P^{ref}})$ correspond to the new entry being inserted; $\overline{SL^V}$ and $\overline{SL^P}$ are existing parameters of the current node.

In case 2, the object is directly inserted into the leaf node.

In case 3, a split operation is performed, where the target leaf node is promoted to a non-leaf node, and new leaf nodes are created. For creating the new leaf nodes, we follow the same process as defined in case 1, and reinsert data entries from the old leaf node in the sub-tree below this new non-leaf node.

An important aspect of this indexing structure is that new nodes are created only when necessary, which results in an efficient insert operation. The drawback of this approach is that it results in an unbalanced tree. However the actual disk space used for the non-leaf nodes is small and the non-leaf nodes often stay resident in the buffer pool.

4.4 Deletion

When the motion parameters of an object are updated, the delete method is invoked to remove the previous entry for the object. Objects send in updated motion parameters together with the old parameters which are used to locate their old entries in the index. The method for locating the old entry recursively applies Equation 1 (see Section 4.3) to locate the leaf node that contains this object. (Recall from the discussion in section 4.1 that it is possible that this object may have *expired*. In this case the update is simply treated as an insert for a new object.)

At deletion time, non-leaf nodes are checked for under-fill, which is defined as whether the number of objects contained in the subtree below this node (indicated by the *size* information stored in the non-leaf node) is less than or equal to the capacity of a leaf node. The following two cases apply:

Case 1: The non-leaf node is not under-filled, in which case the target entry is directly deleted.

Case 2: The non-leaf node is under-filled, in which case all the entries within this node are first collected. Then, this node is converted to a leaf node, and the collected entries are re-inserted into the new leaf node. Finally, the target entry is deleted.

4.5 Update

Updates issued by objects contain the tuple $\left(\left(t_{old}, \overline{v_{old}}, \overline{p_{old}}\right), \left(t_{new}, \overline{v_{new}}, \overline{p_{new}}\right)\right)$ and are evaluated as a delete followed by an insert. The t_{old} and t_{new} reference times are used to determine which of the two indexes the old and new entries belong to.

4.6 Queries

We consider three types of queries: time-slice query, window query, and moving query, which were originally defined in [29]. For ease of reference, we modify the definition in [29] to better fit within our context.

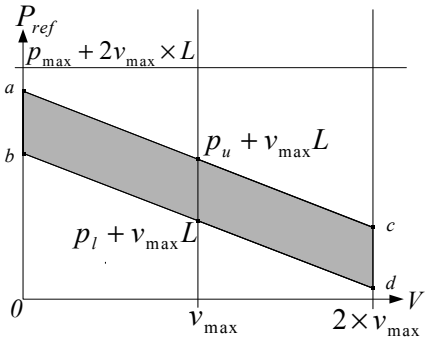

Figure 4: Transformed one-dimensional time-slice query: This example uses query Q1 from Figure 1.

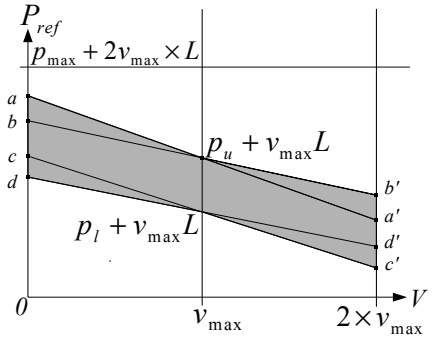

Figure 5: Transformed one-dimensional window query: This example uses query Q2 from Figure 1.

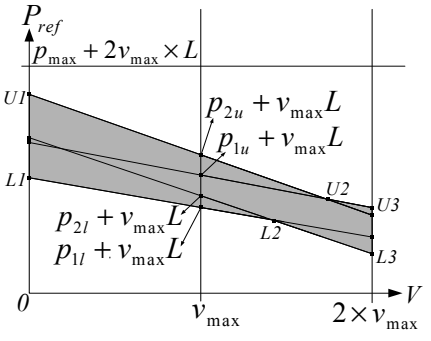

Figure 6: Transformed one-dimensional moving query: This example uses query Q3 from Figure 1.

Let $\overline{p^l} < \overline{p^u}$, $\overline{p_1^l} < \overline{p_1^u}$, and $\overline{p_2^l} < \overline{p_2^u}$ be the vectors of lower bounds and upper bounds in position, and t, t^l, t^u be three time instants not earlier than current time, such that $t < t^l < t^u$. The three types of queries can now be defined as:

Time-slice query: $Q = (\overline{p^l}, \overline{p^u}, t)$ specifies a hyper-rectangle bounded by $[\overline{p^l}, \overline{p^u}]$ at time t.

Window query: $Q = (\overline{p^l}, \overline{p^u}, t^l, t^u)$ specifies a hyper-rectangle bounded by $[\overline{p^l}, \overline{p^u}]$ that covers the time interval $[t^l, t^u]$, i.e., this query retrieves points with trajectories in $\overline{p} - t$ space crossing the (d+1)-dimensional hyper-rectangle $([p_1^l, p_1^u], [p_2^l, p_2^u], \cdots, [p_d^l, p_d^u], [t^l, t^u])$.

Moving query: $Q = ([\overline{p_1^l}, \overline{p_1^u}], [\overline{p_2^l}, \overline{p_2^u}], t^l, t^u)$ specifies the (d+1)-dimensional trapezoid obtained by connecting the hyper-rectangle bounded by $[\overline{p_1^l}, \overline{p_1^u}]$ at time t^l and the hyper-rectangle bounded by $[\overline{p_2^l}, \overline{p_2^u}]$ at time t^u.

Figure 1 illustrates the three query types on objects moving in a native one-dimensional space.

In Figure 1, Q1 is a time-slice query that returns object o1, Q2 is a window query that returns objects o2 and o3, and Q3 is a moving query that returns objects o4 and o5.

The most general query type is the moving query, which is of the form $Q = ([\overline{p_1^l}, \overline{p_1^u}], [\overline{p_2^l}, \overline{p_2^u}], t^l, t^u)$. Window queries are essentially moving queries with $\overline{p_1^l} = \overline{p_2^l}$, and $\overline{p_1^u} = \overline{p_2^u}$, whereas time-slice queries are just window queries with $t^l = t^u$. In essence, the general query Q translates into the following set of inequalities:

$$\begin{cases} \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L \geq \overline{p_1^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L \leq \overline{p_1^u} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L \geq \overline{p_2^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L \leq \overline{p_2^u} \end{cases} \quad \textbf{Eq. 2}$$

4.6.1 Time-slice Queries

For time-slice queries, $\overline{p_1^l} = \overline{p_2^l}$, $\overline{p_1^u} = \overline{p_2^u}$, and $t^l = t^u$, Eq. 2 effectively becomes:

$$\begin{cases} \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t - t^{ref}) - \overline{v^{\max}} \times L \geq \overline{p^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t - t^{ref}) - \overline{v^{\max}} \times L \leq \overline{p^u} \end{cases} \quad \textbf{Eq. 3}$$

The query region for the one-dimensional time-slice query Q1 shown in Figure 1 is illustrated in Figure 4.

Points **a** and **b** are obtained by plugging V = 0 into Eq. 3, and points **c** and **d** are obtained by plugging $V = 2v^{\max}$ into Eq. 3.

4.6.2 Window Queries

For window queries, $\overline{p_1^l} = \overline{p_2^l}$, $\overline{p_1^u} = \overline{p_2^u}$, Eq. 2 effectively becomes:

$$\begin{cases} \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L \geq \overline{p^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L \leq \overline{p^u} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L \geq \overline{p^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L \leq \overline{p^u} \end{cases} \quad \textbf{Eq. 4}$$

The query region for the one-dimensional window query Q2 from Figure 1 is illustrated in Figure 4. The values for the points **a**, **b**, **c**, and **d** are obtained by plugging V = 0 into Eq. 4, and the values for the points **a'**, **b'**, **c'**, and **d'** are obtained by plugging $V = 2v^{\max}$ into Eq. 4.

4.6.3 Moving Query

Figure 6 illustrates the query region for a one-dimensional moving query, using query Q3 from Figure 1 as an example.

In all cases, the query region for one-dimensional queries is a bounded polygon that is confined within an upper bound and a lower bound. Note that the upper bound and the lower bound are not necessarily straight lines (refer to Figures 5 and 6), since we take into consideration the case where objects move in opposite directions. We thus define the query region with six points, $U1$, $U2$, $U3$, $L1$, $L2$, and $L3$ in Figure 6, among which the four marginal points $U1$, $U3$, $L1$, and $L3$ are obtained by calculating intersections of the four query region boundary lines (which are

Algorithm RelativePosition(*R*, U1, U2, U3, L1, L2, L3)
if (lowerLeftVertex(*R*) is on or above LineSegment(L1, L2) **and**
lowerLeftVertex(*R*) is on or above LineSegment(L2, L3) **and**
upperRightVertex(*R*) is on or below LineSegment(U1, U2) **and**
upperRightVertex(*R*) is on or below LineSegment(U2, U3)
then return INSIDE
else if (lowerLeftVertex(*R*) is above LineSegment(U1, U2) **and**
lowerLeftVertex(*R*) is above LineSegment(U2, U3)) **or**
(upperRightVertex(*R*) is below LineSegment(L1, L2) **and**
upperRightVertex(*R*) is below LineSegment(L2, L3))
then return DISJUNCT
else return OVERLAP

Figure 7: Algorithm to test the relative positions of data and query regions.

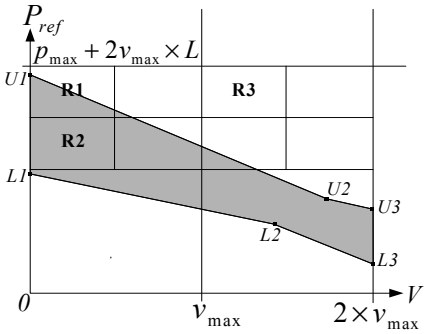

Figure 8: Relative positions of data and query regions for one-dimensional points.

produced by setting the comparison in Eq. 4 to equals), with the boundaries of the underlying dual transformed space.

$L2$ is obtained by calculating the intersection of the following set of lines:

$$\begin{cases} \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L = \overline{p_1^l} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L = \overline{p_2^l} \end{cases}$$

$U2$ is obtained by calculating the intersection of the following set of lines:

$$\begin{cases} \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^l - t^{ref}) - \overline{v^{\max}} \times L = \overline{p_1^u} \\ \overline{P^{ref}} + (\overline{V} - \overline{v^{\max}})(t^u - t^{ref}) - \overline{v^{\max}} \times L = \overline{p_2^u} \end{cases}$$

In the case where either of $L2$ and $U2$ is outside the boundaries, the end points are used.

Effectively, a *d*-dimensional query body consists of d such distinctive query regions corresponding to the d dual transformed planes.

4.6.4 STRIPES Search Algorithm

Queries are processed in STRIPES as follows: At level *l*, each of the *f* grids are tested for relative position to the query body. This test is performed as a conjunction of *d* two-dimensional relative position tests between data regions and the corresponding query region. Relative positions include INSIDE, OVERLAP, and DISJUNCT. A grid is INSIDE a query body if and only if all the position tests return INSIDE; it is DISJUNCT as soon as one of the position tests returns DISJUNCT; otherwise OVERLAP is returned. For all the grids that return an INSIDE result, we immediately retrieve the entries within. DISJUNCT results are discarded and OVERLAP results are further probed recursively. Figure 7 shows the algorithm for relative position test between a data region and a query region. Figure 8 shows the relative positions between data regions and the query region. In Figure 8, the shaded area is the query region, which is a representation of the most general case query: a moving query (note that for a time-slice query the query region can degenerate to a parallelogram). As shown in Figure 8, R3 is DISJUNCT to the query region, while R2 is INSIDE the query region and R1 OVERLAPs the query region.

An additional optimization technique that we use is based on the following observation: At each non-leaf node, each of the *d* 2-dimensional planes partitions the data space into quads in each of the *d* planes. For any node, if any of these quads is DISJUNCT from the query region, then we can safely discard all nodes that are mapped to this quad. Effectively, whenever such quads are determined, the number of search nodes that must be examined is reduced by 25%. This optimization technique quickly prunes away unnecessary node accesses, making the search very efficient.

5. EXPERIMENTAL EVALUATION

In this section, we present results comparing the performance of the STRIPES and the TPR*-tree index.

5.1 Implementation Details and Experimental Platform

We implement both STRIPES and TPR*-tree [34] in the SHORE storage manager [7]. We compiled the storage manager with a 4KB page size. In all our experiments, we set the buffer pool size to 2048 pages; in making this choice for a small buffer pool size, we are essentially following the same philosophy as in previous studies [18, 29, 34] to keep the experiments manageable. SHORE pointers are 16 bytes in size, and we use 4-byte floating points for all the coordinate representation.

The TPR*-tree is implemented using the algorithms described in [34]. The insert algorithm used in the TPR*-tree employs a priority queue that is implemented using heapsort (used in the *ChoosePath* algorithm in [34]). The priority queue stores the cost degradation for each node to insert the update. This queue is then used to determine the best node for inserting an update. The *PickWorst* algorithm of [34] is used to deal with overflow nodes. The best possible entries are removed and then reinserted. Any possible overflow nodes are then split. The index is optimized for static point interval query as in [34].

For the STRIPES index, we simply create non-leaf nodes as (small) SHORE records. Since all sibling non-leaf nodes for a given parent are created concurrently, these nodes are usually stored sequentially on disk. This clustering property results in efficient disk access for the non-leaf nodes.

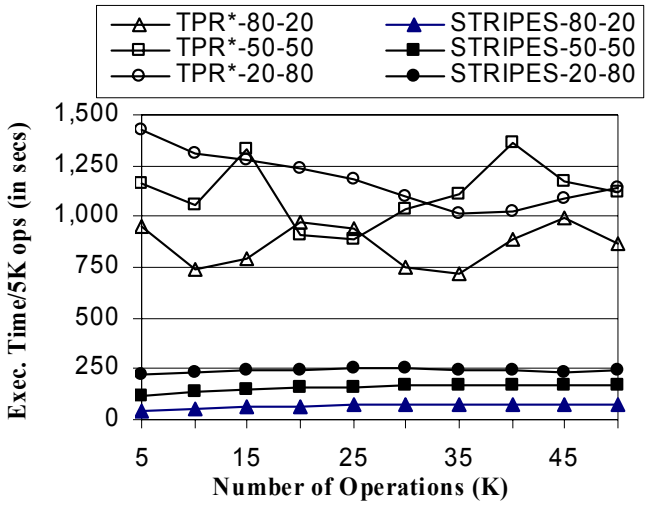

Figure 9: 500K-Uniform: Continuous performance for every batch of 5K operations, for the first 50K operations.

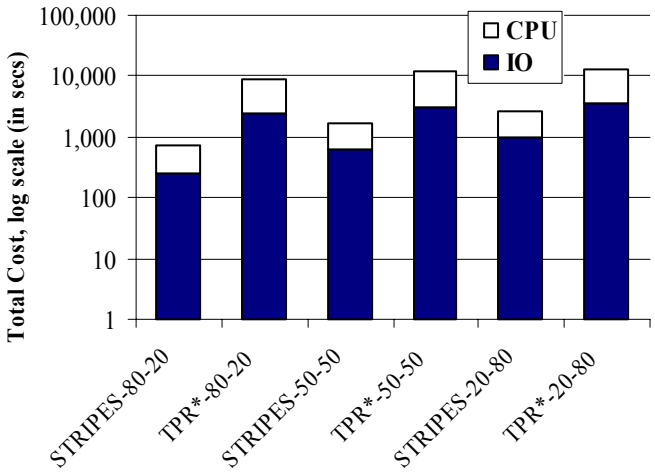

Figure 10: 500K-Uniform: IO and CPU costs for 50K operations.

To implement the leaf node in STRIPES, we use two leaf node sizes, which in the following discussion are referred to as *small* and *large*. When a leaf node is first created, its size is set to *small*, which is approximately half a disk page size. When a small leaf node overflows, it is promoted to a large node. Large nodes occupy exactly one disk page. We adopt this strategy since a split of a leaf node results in the creation of 16 new leaf nodes (for objects moving in two-dimensions). In practice we have found that many of these leaf nodes are empty, and we don't create disk pages for these nodes during the split. Nevertheless the leaf page occupancy is still low at around 12%. Using the two leaf node size allows us to nearly double this page occupancy. With this implementation we find that the STRIPES index is about 2.4 times larger than the TPR*-tree index. In the future, we plan on extending our current implementation to use more than two leaf node sizes, which will increase the occupancy of the leaf-nodes further. However, based on current experimental evaluation, we expect that this may have limited additional benefit on the actual performance of the index as the index size is an issue only in very limited cases. The key to the performance of STRIPES comes from having a relatively small disk footprint for the non-leaf nodes, which results in significant performance advantages over the TPR*-tree index. As an example, for a data set with 500K users, the TPR*-tree index has a height of four and the index occupies around 4,600 disk pages; whereas, the STRIPES index has a maximum height of seven and occupies around 11,200 pages. For this data set the STRIPES index has only 1,486 non-leaf nodes. Each non-leaf node uses 352 bytes for its disk representation, which allows for around 11 non-leaf nodes to fit on a single disk page. Even as the index is updated over time, the non-leaf nodes are contained within a few hundred pages.

The experimental platform used in these experiments is a 2 GHz Intel Xeon machine with a 512KB L2 cache, a 40GB Western Digital 7,200 RMP IDE Hard Drive, running Red Hat Linux 9.

5.2 Data Sets and Workload

We generated a number of workloads using the workload generator which is provided by the inventors of the original TPR-tree [29]. This workload generator simulates objects moving in a two-dimensional space, and has a number of different parameters which can be varied. Although we experimented with a wide range of workloads with different combination of parameter values, in the interest of space, in this section we only present results from using a few representative workloads. These workloads closely correspond to the default values used in the generator, which essentially generates the key data sets used in [29]. In the following paragraphs, we describe the key parameters of this workload generator, and also specify the values for these parameter that we used for generating our workloads.

The workload generator of Šaltenis et al. [29] allows generation of both uniform data workloads, and skewed workloads. In skewed workloads, two-dimensional objects move in a network of routes connecting a number of destinations, *ND*. As the value of *ND* decreases, the skew in the data increases. In our experiments, we generate skewed data sets with *ND* = 20, 40 and 60.

In our workloads, we vary the number of moving objects, *N*, from 100K to 900K. For the 100K data set, the objects move around in a space with dimensions of 1000 x 1000 kilometers. For larger data sets, we scale the dimension size to maintain the same density across all data sets (this strategy for generating scaled data sets is also recommended by [29]). For the uniform workloads, the initial positions of objects are uniformly distributed in space. The workload generator assigns initial positions for each moving object in the system, and then generates a workload which is a mix of update and query operations. The ratio of the number of update and query operations can be varied, and we present results using a mix of 80-20, 50-50, 20-80. For the 80-20 case, 80% of the operations are updates and 20% are queries.

For updates, the directions of the velocity vectors are assigned randomly. The default values for speeds are uniformly distributed between 0 and 3 km/min. The rate of updates is controlled by a parameter, called the update interval, *UI*. The time interval between successive updates is uniformly distributed between 0 and 2*UI*. In the experiments presented in this section, we set *UI* to the default value of 60. The workloads are generated for the default 600 time units.

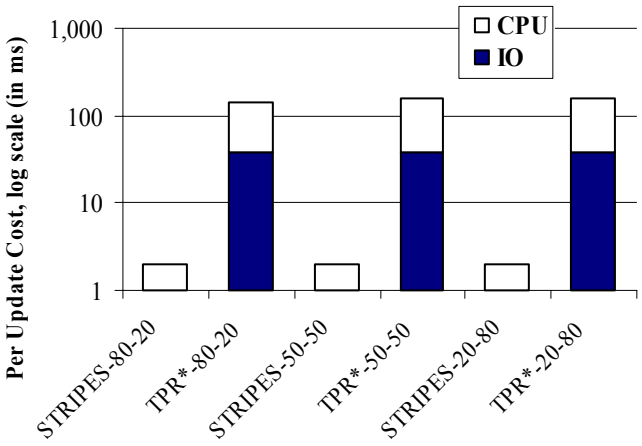

Figure 11: 500K-Uniform: Average per update costs.

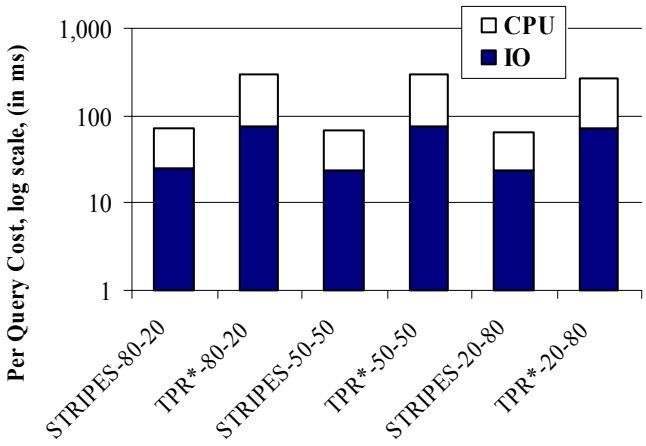

Figure 12: 500K-Uniform: Average per query costs.

For queries, any arbitrary mix of time-slice, window, and moving queries can be generated. The default values for the query mix are 60%, 20% and 20%; all workloads used in this paper are generated using this default setting. For the queries, the temporal range is set to the default value of 40, and the spatial range is set to the default value of 0.25% of the entire spatial extent.

5.3 Effect of Workload Mix

In this first experiment, we use a uniform data set with 500K moving objects. The experimental result for this data set is shown in Figure 9. In this figure, we plot the total execution time for the two index structures shown in batches of 5K operations for the first 50K operations. This experiment lets us determine if the performance of the indices deteriorates as the update operations change the underlying partition boundaries used by the indices.

As can be seen from Figure 9, the TPR*-tree index has a fairly good steady state behavior. This result is consistent with the results presented in the [29]. (In [29] the researchers also show that in contrast to the TPR*-tree, the performance of the original TPR-tree rapidly degrades for a similar experiment.) The TPR*-tree has a good steady state behavior since it uses a much more sophisticated update algorithm (the *ChoosePath* component), which prevents the R*-tree from getting into situations when increasing amounts of dead space and overlap amongst the bounding boxes lead to a rapid drop in performance.

From Figure 9 we observe that STRIPES also demonstrates good steady state behavior. Furthermore, STRIPES is at least **4x faster** than the TPR*-tree index! The reason for this efficiency is that the non-leaf nodes of the STRIPES index occupy only a few hundred pages even as the indexing structure changes with new updates. These nodes are typically resident in the buffer pool and IOs are usually only needed for accessing the leaf-pages. In contrast, during an insert operation in the TPR*-tree, multiple paths are traversed down the tree in the *ChoosePath* algorithm, which results in a large number of IOs.

We see these results more clearly in Figure 10, which breaks down the total costs for the first 50K operations into the CPU and the IO components. To produce this cost breakup, we tracked the time spent in IO operations, and used this measure to divide the total execution time into the IO and the CPU components. Note that in this figure the y-axis uses a log-scale.

As shown in Figure 10, both the IO and CPU costs for TPR*-tree are very high relative to the costs for STRIPES. For updates, every operation in STRIPES requires at most two traversals down the tree (one for removing the old entry if necessary, and one for inserting the new entry). This operation is very CPU efficient, and only incurs a handful of IOs. For this data set (with 500K objects), the STRIPES non-leaf nodes are spread across a few hundred disk pages. These are usually resident in the buffer pool, and IOs are only needed for the leaf-level pages. In contrast, the TPR*-tree index incurs a large number of IOs. In this case, the index size is 4,600 pages and the height of the index is four. During the insert operation, the *ChoosePath* algorithm has to find a good leaf node for the insertion. To accomplish this task, it uses a priority queue based technique to traverse multiple paths to the leaf nodes (see Section 3.2). This technique results in large number of IOs, and also leads to poor referential locality as successive updates are likely to traverse different parts of the index. In addition, when an overflow occurs during an insertion, forced reinsert is performed where a certain percentage of the entries in the overflow node (30% in our implementation) must be reinserted into the tree. Consequently, every time there is an overflow the cost of the insert operation goes up dramatically.

For this workload, we also plot the average cost per update in Figure 11, and the average cost per query in Figure 12. As can be seen from these figures, the IO cost for the TPR*-tree index is significantly higher than the IO cost incurred by STRIPES for *both* the update and query operations. The difference is much more dramatic for the update operation, which is extremely efficient in STRIPES (see Figure 11). Update operation in both index structures requires an insert operation. An insertion in STRIPES only requires inserting a point object, which can be accomplished by a *single* path traversal from the root (see Section 4.3). This operation is extremely fast in quadtree-based structures because of the non-overlapping regular decomposition strategy used by the index structure. In contrast, multiple paths are traversed by the TPR*-tree, which results in a much higher IO cost. For queries, the TPR*-tree often requires traversing multiple paths down the tree, which results in higher query IO costs.

Figures 11 and 12 also show that the CPU costs incurred by STRIPES is much lower than the CPU costs for the TPR*-tree index. For updates, the reason for this is once again the efficiency

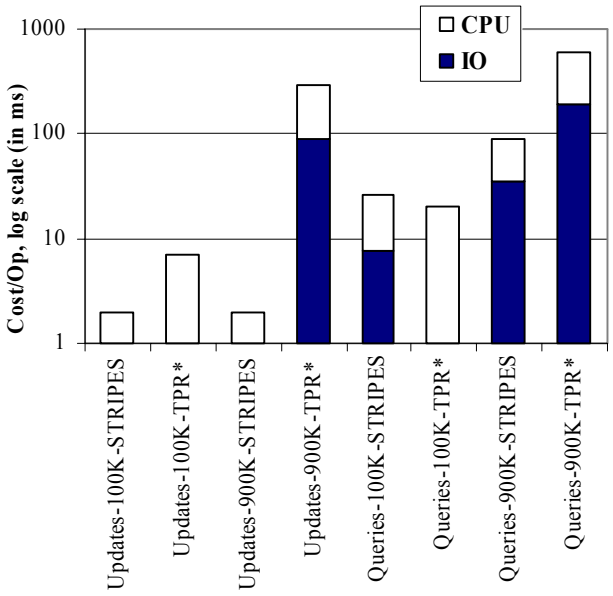

Figure 13: Effect of changing the number of moving objects.

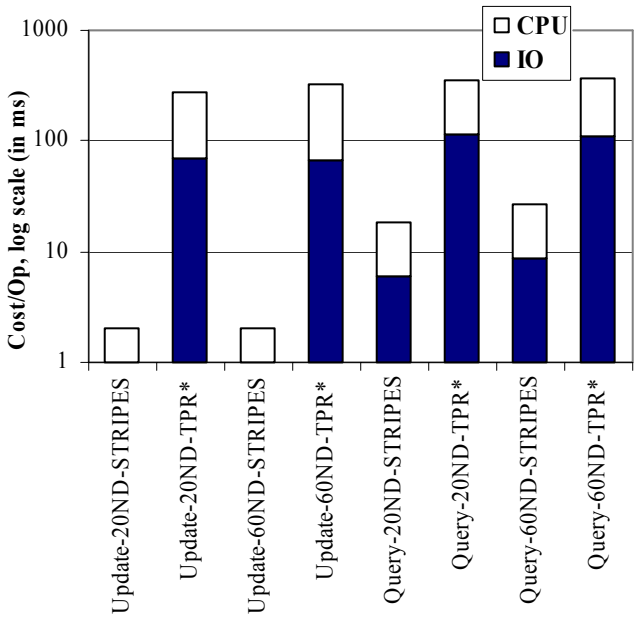

Figure 14: 500K-Skew: Performance for skewed data set.

of the update operation in STRIPES, as compared to the much more expensive technique of multiple path traversals used by the TPR*-tree, which require expensive overlap comparisons at each node. In addition, the CPU costs for the TPR*-tree insert also includes the costs of reinsertions when an overflow occurs (which is very expensive), and the cost for sorting entries. Calculation of the integrals needed for the TPR*-tree are also expensive and contribute to the high CPU cost.

For queries, the techniques employed by STRIPES (described in Section 4.6.4) are much more CPU efficient as compared to the overlap comparisons that are need in the TPR*-tree.

5.4 Scaling with Increasing Number of Moving Objects

In this experiment we explore the effect of increasing the number of moving objects from 100K to 900K users for the three workloads (80-20, 50-50, and 20-80). In the interest of space we only present results for the 50-50 case for 100K and 900K data set cardinalities. These results are shown in Figure 13 as per query and update costs, broken down by the CPU and IO costs.

For the 100K data set the TPR*-tree index fits in the buffer pool and incurs no IO cost, whereas STRIPES incurs IOs, especially for the queries. For this case the smaller index size of TPR*-tree works to its advantage, and the query performance of STRIPES is about 35% worse than the TPR*-tree index. Aggressive disk space optimization outlined in section 5.1, may improve the performance of STRIPES in this case, and we plan on undertaking this effort as part of our future work.

Note that even in the case with the 100K data set, the update operation in STRIPES is about 5x faster as compared to the TPR*-tree. Again the reason for this performance gap is the difference between the expensive insert operation in TPR*-tree and the highly efficient insert operation in STRIPES.

For the 900K data set the performance gap between the two indices widens even further from what we observed with the 500K data set in Section 5.4. The reasons for this performance gap are the same as outlined in Section 5.4. Even for this larger data set, the STRIPES index can keep most of its non-leaf nodes resident in memory. The TPR*-tree's insert algorithm degrades even further because of the larger data set.

5.5 Effect of Data Skew

In this final experiment, we evaluate the performance of the two indices for skewed data sets. We experimented with the *ND* parameter in the workload generator (see Section 5.2) and generated data sets with *ND* = 20, 40, and 60. In the interest of space we only present the results for *ND* = 20 (highly skewed) and *ND* = 60 (small skew) data set, for only the 50-50 workload. Figure 14 plots the per update and per query costs for this experiment. Comparing the update and query costs in this figure with the costs for the 50-50 workload in Figures 11 and 12, we can observe that both index structures handle skewed data sets well, and STRIPES continues to outperform the TPR*-tree by over an order of magnitude for updates, and by 4x for queries.

5.6 Summary

In summary, we have shown through extensive experimental evaluation that STRIPES is significantly faster than the TPR*-tree index. The update operation in STRIPES is often more than an order of magnitude faster, and the query performance is around 4x faster as compared to the TPR*-tree. The regular disjoint decomposition of space that is used by STRIPES results in extremely efficient inserts. In addition, even with very large data sets (relative to the available buffer pool size), the amount of space needed to hold the non-leaf nodes of STRIPES is very small. Consequently, IOs are rarely incurred for the non-leaf nodes. In contrast the TPR*-tree suffers from having to traverse multiple paths down the index, which is IO intensive and results in a reference pattern that has poor cache locality.

These differences in the indexing approaches also manifest in the CPU costs as the TPR*-tree has to carry out many expensive box overlap computations as it traverses down the index. In contrast STRIPES employs a number of optimizations (refer to Section 4.6.4) to keep CPU costs low.

6. RELATED WORK

Within the broader context of indexing trajectories for moving objects, there are two classes of related work: a) methods for indexing the historical and the current positions, and b) methods for indexing the predicted locations of moving objects. The methods for indexing the past and the current locations are typically concerned with queries on exact trajectory points, whereas methods for indexing on the future locations are concerned primarily with indexing the parameters of the predicted trajectory representations (which typically include a velocity vector and a start position vector). In the next paragraph we review the methods for indexing the past trajectory locations, and then turn our attention to the more closely related work in indexing predicted trajectories.

Most of the work on indexing the *past* locations of trajectories is based on variations of the R-tree [12] and the R*-tree [2]. These methods include the 3-D R-trees [36] which simply treats time as a third dimension. The MR-tree [38] and the HR-tree [20] are also 3-D R-tree structures and maintain a separate R-tree for each time stamp. The MV3R-tree[33] is a hybrid structure that uses a multi-version R-tree (MVR) for time-stamp queries, and a small 3D R-tree for time-interval queries. This indexing structure has been shown to outperform other historical trajectory indexing structures, such as the popular TB-tree [23]. SEB[32] and SETI [8] are historical trajectory indexing techniques that partition the spatial extents, and build indices on the temporal dimension. A number of indexing methods have also focused on efficient methods for indexing the *current* location of moving objects [16, 17, 21, 31]. All the methods described in this paragraph are not concerned with indexing the predicted location, and index the native space of the trajectories. In contrast, STRIPES indexes the predicted locations in dual transformed space.

Two main approaches have been used for indexing the *predicted* locations of trajectories. These two approaches are a) methods that index the predicted trajectories in the original spatial and temporal dimensions, and b) methods that transform the predicted trajectories into a dual transform space and index the dual transformed space.

One of the early works on indexing predicted trajectories is by Tayeb et al. [35]. In this work, trajectories in a *d*-dimensional space are treated as lines in a *d+1*-dimensional space, with time as the additional dimension. The line is then indexed using a PMR quadtree [30]. The drawbacks of this approach are that the index may have excessive dead space and replication since it is indexing high dimensional lines.

The TPR-tree [29] is a popular method for indexing predicted trajectories. This index structure uses the basic R-tree indexing structure and extends the notion of bounding boxes to time-parameterized bounding boxes as described in Section 3.1. The notion of time-parameterized bounding box has also been used by other related indexing structures [6, 25]. One of the problems with the time-parameterized boxes is that estimating it requires reasoning about the positions of the objects enclosed by the box over some period of time. The original TPR-tree paper [29] used a conservative bounding box, but this has been improved in a number of different ways [26-28], often by exploiting various additional parameters such as expiration times or the maximum speed. The TPR*-tree is an index structure which improved the methods proposed in the original TPR-tree, and has been shown to be significantly faster than the TPR-tree. In this paper we compare STRIPES with the TPR*-tree, and show that STRIPES outperforms the TPR*-tree by very significant margins.

Dual transformation techniques have been successfully employed for querying static spatial data [14]. Drawing inspiration from this success, dual transformation techniques have also been proposed for indexing predicted trajectories [37]. These indexing methods include the Kinetic data structure [1], the R-tree based parameterized space indexing method [25], and the SV-model [9]. Perhaps the most popular dual transformation approach for predicted trajectories is the work by Kollios et al [15]. In this work, the authors derive nice lower bounds on the cost of answering predictive queries using dual transformation. Most of the paper is concerned with objects moving in one-dimensional space, and the paper sketches extensions to higher-dimensional space. In addition, the paper only considers window queries. The largely theoretical approach has served as the basis for some of the choices made in the TPR-tree [29], but has largely been dismissed by recent work that use a more systems-approach [29, 34]. The dual transformation method used in STRIPES is based on the Hough-X transform used in [15]. STRIPES can handle the entire range of predictive queries, including moving window queries, and we show that STRIPES vastly outperforms the current best know method for indexing predicted trajectories.

A more extensive coverage of related work can be found in a recent review of spatio-temporal access methods [19].

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new indexing structure called STRIPES for indexing and answering queries on predicted positions in moving object databases. This new indexing structure draws inspiration from earlier largely theoretical work in this area, advocating the use of dual transformation for indexing such data sets. The STRIPES index leverages these dual transformation techniques and uses a disjoint regular partitioning technique to efficiently index the points in a dual transformed space. The STRIPES index can support all the types of commonly used predictive queries [29], which include time-slice, window, and moving queries. We have compared the performance of STRIPES with the most efficient predictive indexing structure – the TPR*-tree [34]. Our comprehensive experimental evaluations demonstrate that STRIPES outperforms the TPR*-tree index for both updates and queries; updates are often more than an order of magnitude faster using STRIPES, and queries are often faster by a factor of 4x. These differences can be seen in both the IO and the CPU costs. Consequently, STRIPES is an extremely efficient and practical indexing structure for evaluating predictive queries.

As part of our future work, we plan on investigating index-based algorithms for supporting more complex predictive queries, such those involving nearest-neighbor and join operations. The STRIPES indexing method was developed as part of a larger project called COMET, in which we are investigating a number of research issues related to querying moving object databases. As

part of this larger project, in the future, we also plan on investigating methods for detecting recurring patterns in both historical and predicted trajectories.

8. REFERENCES

1. Agarwal, P.K., Arge, L. and Erickson, J., Indexing Moving Points. In *PODS*, 2000, 175-186.
2. Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B., The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990, 322-331.
3. Bellman, R. *Adaptive Control Processes*. Princeton University Press., Princeton, NJ, 1961.
4. Berchtold, S., Keim, D.A. and Kriegel, H.-P., The X-tree : An Index Structure for High-Dimensional Data. In *VLDB*, 1996, 28-39.
5. Beyer, K., Goldstein, J., Ramakrishnan, R. and Shaft, U., When is "Nearest Neighbor" Meaningful? In *Intl. Conf. on Database Theory (ICDT)*, 1999, 217-235.
6. Cai, M.C., Keshwani, D. and Revesz, P.Z., Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Satabases. In *EDBT*, 2000, 430-444.
7. Carey, M.J., DeWitt, D.J., Franklin, M.J., et al., Shoring Up Persistent Applications. In *SIGMOD*, 1994, 383-394.
8. Chakka, V.P., Everspaugh, A.C. and Patel, J.M., Indexing Large Trajectory Data Sets with SETI. In *First Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2003, 164-175.
9. Chon, H.D., Agrawal, D. and Abbadi:, A.E., Storage and Retrieval of Moving Objects. In *IEEE Intl. Conf. on Mobile Data Management (MDM)*, 2001, 173-184.
10. Gargantini, I. An Effective Way to Represent Quadtrees. *CACM*, *25* (12) 1982, 905-910.
11. Guting, R.H., Bohlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M. and Vazirgiannis, M. A Foundation for Representing and Quering Moving Objects. *ACM TODS*, *25* (1) 2000, 1-42.
12. Guttman, A., R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984, 47-57.
13. Hjaltason, G.R. and Samet, H. Speeding up Construction of PMR Quadtree-based Spatial Indexes. *VLDB Journal*, *11* (2) 2002, 109-137.
14. Jagadish, H.V., On Indexing Line Segments. In *VLDB*, 1990, 614-625.
15. Kollios, G., Gunopulos, D. and Tsotras, V.J., On Indexing Mobile Objects. In *PODS*, 1999, 261-272.
16. Kwon, D., Lee, S. and Lee, S., Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *IEEE Intl. Conf. on Mobile Data Management (MDM)*, 2002, 113-120.
17. Lee, M.-L., Hsu, W., Jensen, C.S., Cui, B. and Teo, K.L., Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003, 608-619.
18. Leutenegger, S.T. and Lopez, M.A., The Effect of Buffering on the Performance of R-trees. In *IEEE TKDE*, 2000, 33-44.
19. Mokbel, M.F., Ghanem, T.M. and Aref, W.G. Spatio-Temporal Access Methods. *IEEE Data Engineering Bulletin*, *26* (2) 2003, 40-49.
20. Nascimento, M.A. and Silva, J.R.O., Towards Historical R-trees. In *ACM Symposium on Applied Computing*, 1998, 235-240.
21. Nascimento, M.A., Silva, J.R.O. and Theodoridis, Y., Evaluation of Access Structures for Discretely Moving Points. In *Intl. Workshop on Spatio-Temporal Database Management (STDBM)*, 1999, 171-188.
22. Papadias, D., Tao, Y., Kalnis, P. and Zhang, J., Indexing Spatio-temporal Data Warehouses. In *ICDE*, 2002, 166-175.
23. Pfoser, D., Jensen, C. and Theodoidis, Y., Novel Approaches to the Indexing of Moving Objects Trajectories. In *VLDB*, 2000, 395-406.
24. Pitoura, E. and Samaras, G. Locating Objects in Mobile Computing. *IEEE TKDE*, *13* (4) 2001, 571-592.
25. Porkaew, K., Lazaridis, I. and Mehrotra, S., Querying Mobile Objects in Spatio-temporal Databases. In *Symposium on Spatial and Temporal Databases (SSTD)*, 2001, 59-78.
26. Prabhakar, S., Xia, Y.N., Kalashnikov, D.V., Aref, W.G. and Hambrusch, S.E., Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. In *IEEE Transactions on Computers*, 2002, 1124-1140.
27. Procopiuc, C.M., Agarwal, P.K. and Har-Peled, S., STAR-Tree: An Efficient Self-adjusting Index for Moving Objects. In *4th Intl. Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002, 178-193.
28. Saltenis, S. and Jensen, C.S., Indexing of Moving Objects for Location-Based Service. In *ICDE*, 2002.
29. Saltenis, S., Jensen, C.S., Leutenegger, S.T. and Lopez, M.A., Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000, 331-342.
30. Samet, H. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, *16* (2) 1984, 187-260.
31. Song, Z. and Roussopoulos, N., Hashing Moving Objects. In *IEEE Intl. Conf. on Mobile Data Management (MDM)*, 2001, 161-172.
32. Song, Z.X. and Roussopoulos, N., SEB-tree: An Approach to Index Continuously Moving Objects. In *IEEE Intl. Conf. on Mobile Data Management (MDM)*, 2003, 340-344.
33. Tao, Y. and Papadias, D., MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB Journal*, 2001, 431-440.
34. Tao, Y., Papadias, D. and Sun, J., The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, 2003, 790-801.
35. Tayeb, J., Ulusoy, O. and Wolfson, O. A Quadtree-based Dynamic Attribute Indexing Method. *Computer Journal*, *41* (3) 1998, 185-200.
36. Theodoridis, Y., Vazirgiannis, M. and Sellis, T.K., Spatio-Temporal Indexing for Large Multimedia Application. In *IEEE Intl. Conf. on Multimedia Computing and Systems*, 1996, 441-448.
37. Wolfson, O., Xu, B., Chamberlain, S. and Jiang, L., Moving Objects Databases: Issues and Solutions. In *Statistical and Scientific Database Management (SSDBM)*, 1998, 111-122.
38. Xu, X., Han, J. and Lu, W., RT-tree: An Improved R-Tree Index Structure for Spatiotemporal Databases. In *Intl. Sym. on Spatial Data Handling*, 1990, 1040-1049.