Handling Frequent Updates of Moving Objects *

Bin Lin and Jianwen Su
Dept. of Computer Science, University of California, Santa Barbara
Santa Barbara, CA, USA
linbin@cs.ucsb.edu, su@cs.ucsb.edu

ABSTRACT

A critical issue in moving object databases is to develop appropriate indexing structures for continuously moving object locations so that queries can still be performed efficiently. However, such location changes typically cause a high volume of updates, which in turn poses serious problems on maintaining index structures. In this paper we propose a Lazy Group Update (LGU) algorithm for disk-based index structures of moving objects. LGU contains two key additional structures to group "similar" updates so that they can be performed together: a disk-based insertion buffer (I-Buffer) for each internal node, and a memory-based deletion table (D-Table) for the entire tree. Different strategies of "pushing down" an overflow I-Buffer to the next level are studied. Comprehensive empirical studies over uniform and skewed datasets, as well as simulated street traffic data show that LGU achieves a significant improvement on update throughput while allowing a reasonable performance for queries.

Categories and Subject Descriptors: H.3.3 [Database Management]: Information Search and Retrieval

General Terms: Algorithms, Performance.

Keywords: Moving objects database, lazy group update, TPR-Tree.

1. INTRODUCTION

With the rapid advances in wireless communications and ubiquitous computing technologies, applications involving moving objects are fast growing. To meet this application demand opens a door for many research problems. Major issues in moving objects databases include: modelling moving objects [8], query language and evaluation [27, 29, 17], and improving application performance by using efficient index structures [4, 12, 25, 31, 11, 19, 35]. In this paper we focus on indexing of moving objects.

Existing index structures for moving objects can be divided into two categories: *location-based* index structures focusing on current/anticipated future locations of moving objects [25, 31, 19], and *trajectory-based* ones for (historical) trajectories [4, 12, 35].

The locations of moving objects are frequently and continuously changing [34], which causes high volume of updates. The updates pose serious problems on index structures. Frequently changing data causes problems in information systems in general [21].

*Supported in part by NSF grant IIS-0101134.

CIKM '05 Bremen, Germany

One effective method to improve the performance of an disk-based index is to increase the memory size. This method can dramatically reduce the I/O cost of queries. But for updates, if one cannot put the entire index into memory, the same strategy is not equally effective. This is because updates are more likely random distributed and can display less locality.

In this paper we develop grouping updates techniques to achieve better memory management in order to effectively improve update performance of disk-based index structures. The key ideas are the use of disk-based "I-Buffers" to process group insertion and of a memory-based "D-Table" to perform group deletion.

Insertions are performed in groups top down through the I-Buffers while deletions are performed in groups bottom up via the D-Table. These techniques in conjunction with a hash based lookup table enable wise utilization of main memory. We also studied different push down strategies for group insertion and analyze the optimal I-Buffer size.

Our major contributions are:

- We develop a "lazy group update" (LGU) algorithm which can manage memory wisely via I-Buffer and D-Table and significantly improve update performance of a disk-based index.
- We formulate an analytical model for the LGU algorithm to estimate overall throughput and decide the optimal size of I-Buffers.
- Comprehensive empirical evaluation over uniform, skewed, and simulated street traffic data show that LGU can improve update throughput by about 100 times for TPR-tree [25], about 30 times for R-tree [9], and about 10 times for FUR-tree [15], when I-Buffer size is reasonably large.

The remainder of the paper is organized as follows. §2 gives a motivating example. §3 briefly reviews R-tree and TPR-tree. Detailed algorithm description is given in §4. §5 analyzes the LGU algorithm. §6 evaluates LGU by comparing it with other algorithms for both R-tree and TPR-tree. §7 concludes the paper.

2. FREQUENT UPDATES

An important consequence of continuous movement of moving objects is that objects' current locations are frequently updated. The combination of the update frequency and the large number of objects could result in failure of database systems to process the updates in time. In this section we first motivate the frequent update problem with an application example concerning the maintenance of moving object locations. We then discuss briefly possible approaches to the problem.

Suppose we are to track locations of 1 million cell phones in the Great Los Angeles area (of 4 million population). If the locations are used for ad hoc queries, index structures (likely external) need be maintained. If an application updates each cell phone's location once every hour on the average (not very frequent!), it translates to

around 280 updates per second. Such a high update rate is a challenge for external memory index structures. It should be noticed that in extreme cases when the update frequency is very high, it is easy to see that any disk based algorithms will fail. It will become necessary to seek in-memory solutions in these situations. Main-memory index structures such as cache conscious B$^+$-tree [22] and buffering access technique [23] are relevant. Discussion of this categories of methods is beyond the scope of this paper.

Fig. 1 illustrates a serious problem of TPR-tree in dealing with frequent updates. It shows the maximal number of location update operations TPR-tree can handle per second and the update frequencies of different size of data (the points labeled 100K and 400K). For the example mentioned above with 1 million moving points, TPR-tree can only process less than 15 out of the 280 requested updates per second (the detailed experimental setting is given in §6). Even if we reduce the number of moving points to 400 thousand, TPR-tree is still incapable of handling all incoming updates. The largest number of points a TPR-tree can handle in our experiments is around 100,000 points. Clearly, both the number of points and the update interval are at the lower spectrum of such applications.

A reason that TPR-tree cannot achieve higher update throughput is that it has to search the tree for the target entry upon a deletion. An immediate solution is to keep an in-memory look-up table for all entries' locations in the tree. The result is an improved algorithm TPRK [24]. Surprisingly TPRK does not improve TPR a lot (Fig. 1). This is because the look-up table occupies a significant portion of memory (which is used as system buffers to speed up performance in TPR).

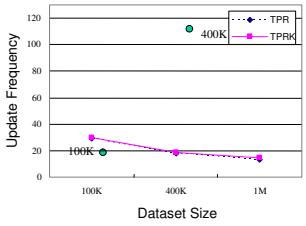

Figure 1: Update Throughput of TPR-tree

Different approaches have been taken to solve the frequent update problem. Group update [14, 16, 7, 3, 1] is a typical one. [16] discusses group update for in-memory tree structure (AVL and Red-Black trees). [14] focuses on Multi-way trees (similar to B$^+$tree). Neither algorithms can be extended to R-tree naturally. Another idea is to reduce the cost for each update. [7] introduces a group insertion algorithm for R-tree, which does not handle updates. [3] proposes a buffer-tree [1] based algorithm, which can perform group insertions, deletions and queries in a top-down way. This algorithm can perform interleaved updates and queries together in a lazy way. But queries can not be answered until they are pushed down to the leaf level, which means long response time [2] and is not favorable to moving objects applications.

Another approach is to improve the efficiency for individual updates. [13] provides direct access to leaf level of R-trees with the help of a main-memory resident index. [15] extends the main-memory index to a structured summary to help both updates and queries and presents more heuristics in update process to achieve better performance. But the improvement of individual update is limited because each update costs at least 2 I/Os.

[28] proposes a non-tree index structure aiming at reducing frequency of updates. This method uses hashing based buckets combining with filter layers. Moving objects are indexed in the buckets and updates are handled by filter layers. If the updated location is still inside the same bucket, the update is ignored by the index structure. Thus the number of actual updates to the index is reduced. But the efficiency of queries is significantly worse since all points in the buckets which intersect the query window have to be scanned by sequential search. In our algorithm we employ both buffer and group update technologies and achieve much higher update throughput for TPR-tree. [10] introduced an index structure based on dual transformations. Compared with TPR-tree, this method provides comparable query performance and better update performance. But the improvement of update performance is not very significant (e.g. 3-5 times in their experiments).

3. INDEX AND THROUGHPUT

R-tree [9] is widely used in different applications relating to spatial data and other low-dimensional data. All leaf nodes are at the same level in R-tree. Each internal node contains a minimum bounding rectangle (MBR) which bounds all objects in the sub-tree rooted at itself. There may be overlaps between internal nodes. So a range query on the R-tree may have multiple search paths from the root to the leaf level. R-tree has many variants, such as R^*-tree [5], R^+-tree [26], TPR-tree [25], TPR*-tree [30]. The techniques presented in this paper can be easily applied to most of them.

Three operations to modify an R-tree (TPR-tree, etc.) are:

- *Insertion* : adding a new object
- *Deletion* : removing an existing object
- *Update* : update an existing object's information

In moving objects applications, the frequency of updates is much higher than that of insertions and deletions, because moving objects keep changing their location information. In this paper we thus focus on the "update" operation.

TPR-tree [25] is an extension of R*-tree [5]. In the TPR-tree shown in Fig. 2, A is a bounding rectangle containing three 2-D moving objects a, b, c. The boundaries of A are also moving to ensure that a, b, c will always reside in the bounding rectangle, their velocities have to be determined by the maximum (or minimum for negative values) velocities of a, b, c. Initially ($t = 0$), the bounding rectangle is the minimum bounding rectangle (MBR) of a, b, c. As the time progresses, the bounding rectangle is no longer minimum (it will be tightened to MBR when next update touches it). TPR*-tree [30] is a variant of TPR-tree with query-parameterized cost model and new insertion/deletion algorithms.

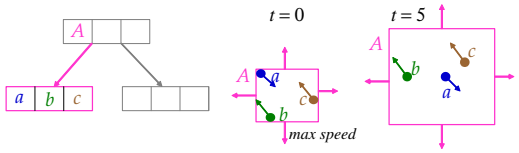

Figure 2: An example of TPR-tree

An update is issued when a moving object's velocity (direction or speed) changes, which is less frequent than its location changing. Suppose a two dimensional moving object o's location information is given by $(x_1 + v_{x,1}t, y_1 + v_{y,1}t)$. After the update at time t_0, the new location information is $(x_2 + v_{x,2}t, y_2 + v_{y,2}t)$, where $x_2 = x_1 + v_{x,1}t_0$ and $y_2 = y_1 + v_{y,1}t_0$.

Since TPR-tree performs much better than other R-tree variants in indexing moving objects' current and near future locations [25, 30], we choose TPR-tree to describe our algorithm. Our techniques can be easily applied to most R-tree variants.

Definition. The *update (query,* or *overall) throughput* of an index structure is the maximum number of updates (queries, query and update operations, resp.) it can process in a unit time.

In the cell phone example discussed in §2, the update throughput for TPR-tree is roughly 15 while the incoming update rate is 280 updates per second. Obviously this is a big gap. The aim of this

paper is to eliminate (or at least reduce) the gap between the high update request rate and the relative low update throughput.

In R-tree updates are executed as deletions followed by insertions. A top-down search is first issued to locate the entry of the object and delete it from the leaf node. Then another separate top-down search is used to find the best location for the new entry and insert it. This process is obviously costly. In addition, node splits and reinsertions of index entries may occur and cost even more.

Similar to other R-tree variants, TPR-tree performs updates using deletions and insertions. The insertion and deletion algorithms are based on the following four basic functions:

- *ChooseSubTree*(v, o): given a TPR-tree node v and a moving object o, choose a subtree (according to some heuristics) for o.
- *Split*(v): split a given TPR-tree node v into two new nodes v_1 and v_2, and update the parent of v accordingly.

  Since the number of children in v's parent is increased, this function may be propagated to upper levels.
- *Search*(v, o): given a TPR-tree node v and a moving object o, returns the leaf node containing the object o.
- *Rebalance*(v): delete the TPR-tree node v and re-insert all its entries and update the parent of v. Since the number of children of the parent is decreased, the change may be propagated.

To avoid the expensive search process, an immediate solution is to maintain an in-memory lookup table in the form of (oid, p), where oid is the object id and p indicates which page this object locates (TPRK) [24]. By using the lookup table the entry to be deleted can be directly located and removed. This method achieves better update performance but requires more memory occupation due to the existence of the lookup table.

4. LAZY GROUP UPDATE (LGU)

In this section we illustrate the ideas of the LGU algorithm with an example (§4.1), present insertion and the I-Buffer technique (§4.2) and deletion and the D-Table method (§4.3), discuss query evaluation (§4.4), and compare LGU with buffer-tree [3] (§4.5).

4.1 Algorithm Overview

LGU uses three key techniques to improve update throughput:

- It uses disk-based buffers (I-Buffers) to utilize memory wisely. One I-Buffer is associated with each internal node; insertions are "pushed" from root to leaf in group through the I-Buffers. A global memory-based buffer (D-Table) is utilized to perform group deletion in the bottom up manner.
- It performs insertions and deletions in different ways: insertions are performed lazily in a top-down way while deletions are performed lazily bottom-up.
- A hash lookup table similar to TPRK is used to enable direct access to the leaf level when performing deletions bottom-up.

Before describing the details of LGU, we first present a simple example. As shown in Fig. 3, I-Buffers are attached to internal nodes and a global D-Table is maintained for incoming deletions.

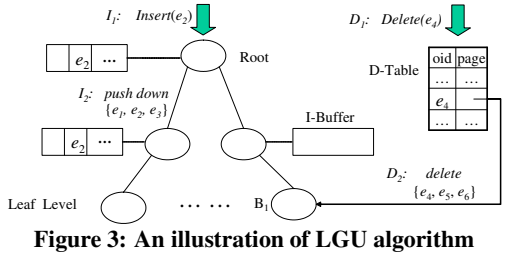

Figure 3: An illustration of LGU algorithm

I-Buffers are organized in the same way as leaf nodes, containing the inserted entries. D-Table records the object's ID and its located page. In Fig. 3, e_4's current located page is B_1.

If we insert an entry e_2 into the tree, it first goes to the I-Buffer of the root, which is the I_1 operation in the example. Then after some time, as shown in the I_2 operation, e_2 is pushed down with other entries to lower levels. Push down operations may repeat until e_2 reaches the leaf level.

The operations for a deletion are illustrated as D_1 and D_2. D_1 shows the first step: incoming deletions (e.g. *delete*(e_4)) are put in the D-Table. At the second step D_2, e_4 is performed together with other deletions who have the same target page as e_4.

The I-Buffer technique in LGU is a variant of Arge's "buffer tree technique" [1] and its R-tree version [3]. A comparison between LGU and Arge's algorithm is presented in §4.5. But the D-Table and the bottom-up group deletion are new techniques, and it is the first time for them to be applied on external tree updates.

4.2 Insertions

An I-Buffer (on disk) is attached to each internal node of a TPR-tree. With I-Buffers, all incoming insertions are performed in a lazy batch fashion: they are put in the I-Buffer of the root. When an I-Buffer becomes full, an algorithm *Pushdown* is invoked: it chooses some or all entries in the I-Buffer (according to some heuristics) and pushes them down to its child nodes (if its child nodes are leaves) or their I-Buffers.

An immediate question is how to choose the entries to push down. The naive solution is to push all entries in the I-Buffer, which is not good if the distribution of entries is not even.

Strategy	*Metric*	*Description*
Push-All	none	push all groups
Push-L	popularity	push the largest group
Push-Mp	age	push the youngest group

Figure 4: Pushdown Strategies

Fig. 4 shows three different strategies. In addition to "push all", LGU considers two other metrics. In both cases we divided the entries into groups according to their destinations. In Push-L strategy, we only push the largest group, which ensure us not to waste I/Os on small groups. In Push-Mp (Most Promising) strategy, we only push the group whose average age is the lowest (i.e. whose average update time is the latest). This strategy put high priority on "younger" entries because their life time are longer. The "old" entries may become obsolete very soon and will be removed from the I-Buffer directly, which is the cheapest operation for deletions.

Algorithm 1 *PushDownOne*(v)

```
1: Get v's buffer b
2: NodeClean(b)
3: Divide entries in b into groups according to their destinations
4: Compute the score for each group according to pushdown strategy
5: Choose the group g with the highest score, whose destination is d
6: Adjust d's bounding rectangle if necessary
7: Update the lookup table for all entries in g
8: if d is an internal node then
9:     Insert all entries in g into d's buffer
10:    if d's buffer is full then
11:        PushDownOne(d)
12:    end if
13: else
14:    Insert all entries in g into d
15:    if d is full then
16:        Split(d)
17:    end if
18: end if
```

Algorithm 1 shows the process of Push-L (and Push-Mp): divid-

ing entries into groups and only pushing down one group at a time. During the *Pushdown* process, LGU first calls a *Node Clean* process to clean all obsolete entries in the buffer (Line 2). The reason for this will be shown in Fig. 6. LGU then groups entries in the I-Buffer according to their destinations (the grouping processes are different for different strategies in Fig. 4) and chooses one group according to some metrics in Fig. 4 (Lines 3-5). After adjusting the destination node's bounding rectangles (Line 6) and updating the lookup table for all entries in the group (Line 7), LGU pushes these entries down. If the destination node is an internal node, the entries pushed down go to its buffer; if its buffer becomes full, *Pushdown* is invoked again (Lines 8-12). If the destination node is a leaf node, the entries go to itself; if it becomes full, *Split* is called to divide the full leaf into two leaves. The split process can be propagated to upper levels (Lines 13-17). *Split* behaves similar to TPR-tree's except that if the split node is an internal node, its I-Buffer is split as well. *Split* also has to update entries' page number information in the lookup table.

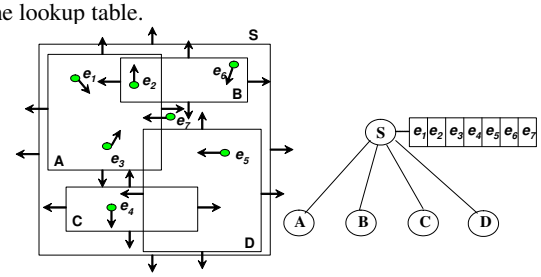

Figure 5: Illustrating push down strategies

Consider Fig. 5 where the buffer of S is full and needs to be pushed down. Three of the seven entries are going to child A (e_1, e_2, e_3), and one each to B (e_6), C (e_4), D (e_5), resp. Entry e_7 can go to A, B, or D. If we choose to push all entries, there are 1 I/O to read S's buffer, and 2 I/Os for reading and writing the buffers of A, B, C, D. The total cost is 9 I/Os for pushing down 7 entries. So it is averagely 1.3 I/Os per entry, not very efficient.

If we choose to push the largest group, we can push e_1, e_2, e_3, e_7 into A's buffer, which incurs 1 I/O to read S's buffer, and 2 I/Os for reading and writing the buffer of A. So the average I/O cost for each entry is 0.75, which is much better than the Push-All strategy.

If we choose the Push-Mp strategy and the update time for each entry is: $e_1(5), e_2(100), e_3(8), e_4(50), e_5(34), e_6(150), e_7(200)$. Then LGU can find that the group (e_2, e_6, e_7) has the largest (latest) update time (150). So all the three entries are pushed down to B's buffer, which incurs 1 I/O to read S's buffer, and 2 I/Os for reading and writing the buffer of B. And the average I/O cost for each entry is 1. If e_1 and e_3 are deleted directly from the buffer some time later (this is very possible since they are very "old" entries), that means we get "free" processes for 2 more entries due to the Push-Mp strategy. In this case, we pay 3 I/Os for 5 entries and retrieve 0.6 average I/O cost per entry.

Both Push-L and Push-Mp seem to perform better than Push-All.

4.3 Deletions

For deletions, LGU does not perform batch deletion in the same way as insertion, which is the strategy used in buffer-tree. This is because when you push down a deletion, you need to push it into all child nodes whose bounding rectangles contain the target entry. The case is worse when overlapping between bounding rectangles is high, which is a prominent phenomena in a TPR-tree since bounding rectangles in a TPR-tree are expanding over time.

LGU maintains a global in-memory *deletion table* (D-Table) for unprocessed deletions. The D-Table maintains each deletion's object ID (*oid*) and its location in the tree (*page number*). Incoming deletions are stored in the D-Table. A key-based direct access method is also employed to avoid search during deletions. LGU has to maintain an in-memory lookup table recording each existing object's located page in the TPR-tree. Note that in deletions of a TPR-tree, it may traverse the tree bottom up to tighten bounding rectangles. But in our algorithm, we choose not to traverse up to save process time and storage space of the look up table.

Deletions are performed based on the lookup table. LGU locates the destination page first and then remove the page number information of the entry from the lookup table. After that, an entry (e, p) is put into the D-Table. When the D-Table gets full, a *Buffer Clean* process is called to apply some deletions in the D-Table. First all deletions are grouped based on their target page, and then the group with the most deletions is applied. LGU chooses to perform the largest group because this makes the lowest average I/O cost for each deletion. If the target page is a buffer page, then the deletion process is over. Otherwise the target page must be a leaf node page. If it becomes underflow after the deletions, a $Rebalance$ process is called. This process is the same as the one used in TPR-tree. LGU does not adjust the target node's parent node's bounding rectangle because this will introduce the expensive $Search$ process. This strategy may result in larger intermediate bounding rectangles, but they will be tightened when some entries are inserted into them.

Due to the delay of deletions, there may be multiple entries of the same object in the tree. Among these entries, only the latest one is valid. The problem is if an obsolete (invalid) entry moves to other nodes, it can no longer be found. Fig. 6 displays this problem. Suppose there are 2 deletions in the D-Table for the same entry e and neither has been processed. So there are 2 copies of e resident in the tree at this time. We call the earlier one e_1 and the other e_2. In the D-Table the located pages of both entries are recorded, which belong to two buffers B_1 and B_2, respectively. And in the lookup table the corresponding located page is the same as e_2's. If at this time a push down process for B_1 is called and e_1 is pushed down to B_3, LGU faces a dilemma: if it does not update the entries of e in the lookup table, e_1 will be lost; if it does so e_2 will be lost.

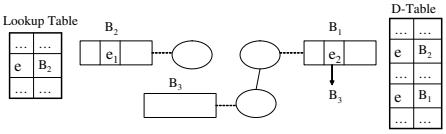

Figure 6: A problem of deletions

To avoid these obsolete entries moving around the tree, a *Node Clean* process should be called for both the source buffer and the target buffer whenever some entries' are moving from one page to another (e.g. push down, split). This process will eliminate obsolete entries from the buffer. Since during the entries' moving we need to access both the source and target buffers anyway, no extra disk I/Os are introduced.

4.4 Query Evaluation

In LGU queries are executed in a similar way to that in TPR-tree. Since we keep inserted entries into I-Buffers, it is necessary to check I-Buffers for complete answers. Due to the fact that D-Table may be contain obsolete entries, LGU also needs to check the D-Table to eliminate the obsolete answers.

For example, to answer a time stamp range query (other queries such as time interval queries, moving queries can be answered in the same way), LGU starts from the root and computes the intersections between the sub-trees' bounding rectangles and the query window. If a sub-tree's bounding rectangle is disjoint from the query window, the sub-tree is eliminated from further consideration. If a sub-tree's bounding rectangle is contained by the query window, all entries in the sub-tree are included into the candidate

answers. If a sub-tree's bounding rectangle intersects but is not contained in the query window, LGU recursively searches the sub-trees. For each intermediate node searched, its I-Buffer is searched for candidate answers as well. After collecting all candidate answers, LGU looks up the D-Table to eliminate obsoleted answers.

4.5 Comparisons with Buffer Tree

Our I-Buffer technique is extended from the buffer tree algorithm [3] with three important differences. **1.** LGU queries are allowed on I-Buffers and the D-Table, so queries can be processed immediately after their arrival. In buffer trees, the processing of queries cannot be completed before they are pushed to the bottom the tree. Moreover, to guarantee the correctness of the answers, queries cannot be finished before all the insertions and deletions issued earlier than them are pushed to the bottom. So that means long response time for queries [2]. This is not favorable because information in the tree is changing quickly and queries need to be answered as soon as possible. **2.** Buffer tree performs deletions top down lazily (not good for TPR-trees). LGU combines both key-based direct access and D-Table techniques to achieve group deletion in a bottom-up way (see §4.3). **3.** The buffer tree algorithm just chooses the naive push all strategy in the push down process. This is not good because some groups pushed down are of very small sizes. LGU allows two more push-down strategies Push-L and Push-Mp; the latter two provide better push down performance than Push-All.

5. ANALYTICAL MODEL

In this section we analyze the LGU algorithm. We focus on the overall throughput of the TPR-tree (Eq. 1 and 5). Since the D-Table is memory resident, it is obvious that bigger size will ensure better performance (in terms of disk accesses). We also focus on computing the optimal I-Buffer size (Eq. 6).

Since updates are performed as deletions and insertions, the overall I/O cost C of a TPR-tree over time interval $[t_s, t_e]$ is $C_{\text{total}} = C_{\text{I}}+C_{\text{D}}+C_{\text{Q}}$, where C_{I}, C_{D}, and C_{Q} are the I/O cost of insertions, deletions and queries, respectively.

Suppose during $[t_s, t_e]$, the numbers of incoming updates and queries are N_u and N_q, respectively, and each disk I/O takes t_{io} time, we can retrieve the overall throughput T by:

$$T = (N_u + N_q)/(C \times t_{io}) \tag{1}$$

In our analysis, we make three simplifying assumptions.

1. Data distributions are uniform, in spatial/velocity dimensions.
2. The tree is perfectly organized so that all moving objects are distributed evenly in the tree and there is no overflow and underflow during the running time.
3. All entries are deleted in the leaf level.

Assumptions (1-2) are standard for analysis of R-tree variants. Assumption (3) simplifies the analysis. Actually there are entries directly deleted from I-Buffers before they can reach the leaf level. The percentage of such entries is proportional to the size of I-Buffers. So our analysis provides an upper bound of the I/O cost.

Since there is no underflow and the deletions are performed in groups, and for each group we need to perform two I/Os (one for reading and one for writing), we obtain $C_{\text{D}} = 2N_u/g_d$, where g_d is the average group size for deletions.

For insertions, we focus on the Push-L strategy. (Other strategies can be done in a similar way.) Since all the entries are deleted in the leaf level, each inserted entry must be pushed down $h-1$ times when it moves from the root to the leaf level, where h is the height of the tree. We assume that the root's I-Buffer is always in memory. And the I/O cost for each push down between internal levels is $2B_i/f + 2$ (f is the fan-out factor and B_i is the I-Buffer size), where we pay $2B_i/f$ I/Os for reading and writing the target buffer (notice that the source buffer must be in memory already), and 2 I/Os for reading and writing the parent node. So the overall cost for one insertion C_{si} can be computed with Eq. 2, where the extra 2 I/Os are for pushing down to the leaf level.

$$C_{si} = (2B_i/f + 2)(h-2) + 2 \tag{2}$$

Then C_{I} can be computed with Eq. 3, where g_i is the average group size pushed down. For Push-L strategy, each time we push the largest group. Since how to determine the expected size of the largest group remains a partially open question [20], we approximate g_i by experiments. In the experiments we observes that if $2B_i/f \geq 1$, g_i converges to $2B_i/f + 1$ when f increases to unlimited. So we approximate g_i with $2B_i/f + 1$.

$$C_{\text{I}} = \frac{N_u C_{si}}{g_i} = \frac{N_u((2B_i + 2f)(h-2) + 2f)}{2B_i + f} \tag{3}$$

For queries our analysis is extended from [32]. We assume the entire (2 dimensional) workspace is a unit square. We focus on time instant queries in this analysis. Given a query $q = \{t, q_1, ..., q_n\}$, where n is the number of dimension, t the query time, and q_i ($i = 1, ..., n$) the size of the query window on i^{th} dimension. The probability of the query window intersecting a bounding rectangle in level j is $\Pi_{i=1}^{n}(s_{j,i}+q_i)$, where $s_{j,i}$ is the average extent in level j, dimension i. Since LGU algorithm searches I-Buffers, we can compute C_{Q} with Eq. 4 [32] where N is the dataset size. In Eq. 4 $(N/f)\Pi_{i=1}^{n}(s_{1,i} + q_i)$ I/Os are paid for searching the leaf level, and the rest part are for intermediate levels (the root and its buffer are assumed always in memory).

$$C_{\text{Q}}=N_q(\frac{N}{f}\prod_{i=1}^{n}(s_{1,i}+q_i)+\sum_{j=2}^{h-1}(\frac{B_i}{f}+1)\frac{N}{f^j}\prod_{i=1}^{n}(s_{j,i}+q_i)) \tag{4}$$

Since we assume that the tree is organized perfectly, there is no overlaps between adjacent bounding rectangles at the beginning and later overlaps are caused only by movement of objects. In the TPR-tree once an update touches a bounding rectangle, it will make it minimum bounded. After that, the bounding rectangle will expand with time until the next update comes in. Due to the uniformity and perfect tree assumptions, the size of the bounding rectangle right after an update touch is the same as its initial size.

Suppose ΔV_i represents the velocity difference on dimension i, and T_{ui} is the average update interval for each object. Then we can compute $s_{j,i}$ with $s_{j,i} = s_{j,i,0} + \Delta V_{j,i} T_{ui,j}/2$, where $s_{j,i,0} = (D_j f^j/N)^{1/n}$ [32] is the initial size of a bounding rectangle in level j dimension i, in which n is the number of dimensions and D_j is the data density in level j. And $T_{ui,j} = \frac{T_{ui}}{f^j}$ is the average update interval of bounding rectangle in level j.

We can now compute the overall I/O cost C_{total}=C_{I}+C_{Q}+C_{D}=

$$\frac{N_u((2B_i+2f)(h-2)+2f)}{2B_i+f} + N_q((\frac{B_i}{f}+1)A+E) + \frac{2N_u}{g_d} \tag{5}$$

where A=$\sum_{j=2}^{h-1}\frac{N}{f^j}\prod_{i=1}^{n}(s_{j,i}+q_i)$ and E=$\frac{N}{f}\prod_{i=1}^{n}(s_{1,i}+q_i)$.

To minimize $C(B_i)$, we let the derivative of $C(B_i)$ be 0. So

$$B_i = \sqrt{N_u/N_q}\sqrt{If/4A} - f/2 \tag{6}$$

where $I = 2f(h-2) + 4$.

The optimal I-Buffer size B_i is proportional to the square root of N_u/N_q, which confirms the intuition that I-Buffer improves update performance and slows down queries. In §6 we will compute the optimal I-Buffer size according to our experiment setup and compare the expected costs of updates and queries with our results.

6. EXPERIMENTAL RESULTS

In this section we evaluate the LGU algorithm by doing two sets of experiments. The first set compares LGU performance in a TPR-tree with the original update algorithm (TPR) [25] and a key-based

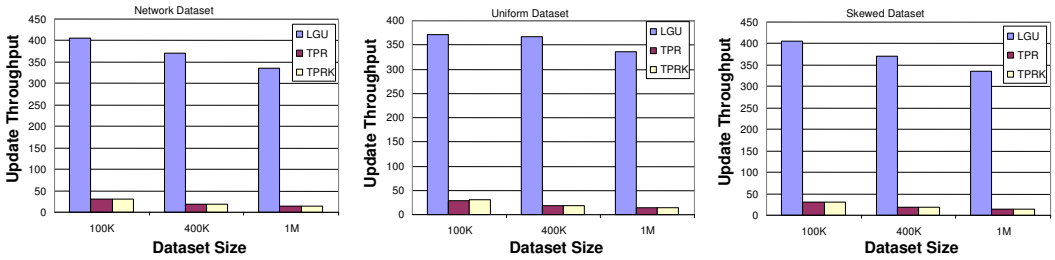

Figure 7: Update throughput with varying dataset size (TPR-trees)

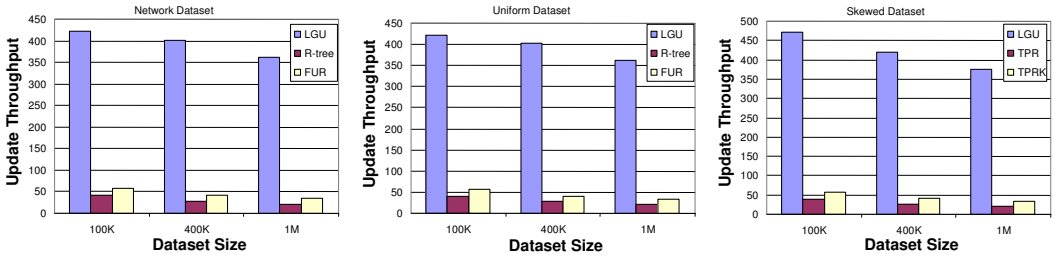

Figure 8: Update Throughput with varying dataset size (R-trees)

direct access version (TPRK) [24]. TPRK maintains a look-up table similar to LGU to enable key-based direct access to the leaf level. The second set compares a LGU-based R-tree, an original R-tree (R*-tree more precisely [5]), and a FUR-tree [15]. FUR-tree is the most recent technique for frequent updates in R-tree. All comparisons are measured with disk I/O cost, which is the major concern for disk-based index structures.

The main findings from the experimental study are:

- LGU-based TPR-tree significantly outperforms TPR and TPRK in update throughput (up to 100 times). And so does LGU over R-tree and FUR-tree.
- LGU's query throughput is comparable to TPR and TPRK.
- Push-L and Push-Mp strategies perform better than Push-All.
- Improvement of LGU over TPR, TPRK, R-tree, and FUR-tree increases when dataset size increases.

In all head-to-head comparisons, index structures all get the same amount of main memory, i.e., the comparisons are fair.

6.1 Experiments Setup and Parameter Settings

We use three different datasets in our evaluations:

- *Uniform*: All points' initial locations are uniformly distributed in the workspace, so are their speeds in the speed range.
- *Skewed*: All points' initial locations and speeds have higher probability to locate close to the center and to have slow speed using a Zipfian distribution generator in GSTD [33].
- *Network*: A dataset generated by Brinkhoff's generator [6]. With this generator, we simulate a cell phone tracking application in the city of San Francisco.

For the uniform and skewed datasets, moving points are generated in a workspace of 100×100 square kilometers. The speeds of moving points range from 75 meters per minute (pedestrians) to 300 meters per minute (vehicles). For the network dataset, moving points are moving along the streets in San Francisco.

The number of moving points is 1 million for most experiments. After the tree is built, a workload composed of both queries and updates is executed over the tree. In all experiments, we evaluate the tree's performance for 60 time units (minutes) after the tree building. The incoming query rate is 20 queries per time unit for most experiments.

For all experiments, the disk page size is set to 4K bytes. Unless explicitly mentioned, LGU uses *Push-L* strategy in dealing with push down. The size of I-Buffers is 16 pages and size of the D-Table is set to 4 times the number of leaf nodes (about 320 KBytes for 1 million objects in TPR-tree and 160 KBytes in R-tree). The look-up table in LGU and TPRK takes 8M bytes memory for 1 million objects. The system buffer size is 20 pages for LGU. To compensate the memory occupied by the D-Table in LGU, the system buffer size of TPRK is set to 30, 60, and 100 pages for the datasets of size 100K, 400K, and 1M, respectively. For TPR, R-tree, and FUR, the system buffer size is 200, 800, and 2000 pages for the datasets of size 100K, 400K, and 1M, respectively. The extra buffer pages for them are to compensate for the memory occupied by the D-Table and the look-up table in LGU.

For R-tree we only execute range queries. TPR-tree supports three types of queries: time instant range queries, time interval range queries, and moving range queries. A moving range query is a time interval query whose query window may change over time. A parameter *QWS* is used to described the size (area) of query windows in terms of a fraction of the total workspace. For most experiments we set $QWS = 0.25\%$. The compositions of the three types of queries are 60%, 20%, and 20%, the same as in [25]. For time interval queries (the latter two types), the query time interval is randomly picked from $[0, 20]$. For time instant/interval range queries, query windows are randomly scattered over the workspace; for moving range queries, the centers of query windows are always some moving points in the tree.

In computing update throughput, assume each I/O is a random disk access, which takes 10 milliseconds [18]. So the throughput $T = \frac{100}{N_{io}}$, where N_{io} is the average I/Os per operation.

6.2 Scalability

We observe that the update throughput is significantly improved by LGU in all three different datasets over TPR-tree and R-tree. The improvements are similar in different datasets.

Fig. 7 and 8 show the overall results of TPR-trees and, resp., R-trees. For the case of 1 million objects, LGU improves the update throughput by about 30 times over TPR and TPRK, about 18 times over R-tree, and about 10 times over FUR-tree. Also, the update throughput of LGU is less sensitive to dataset size, while TPR's update throughput drops a lot when the size increases. Since the performance comparisons in the uniform dataset and the skewed dataset are similar, we only present experimental results of the uniform dataset and the network dataset in the rest of this section.

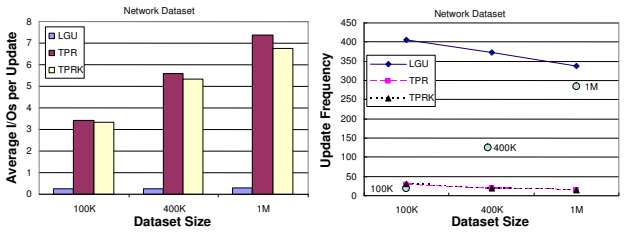

Figure 9: Update I/Os and frequency vs. dataset size (TPR)

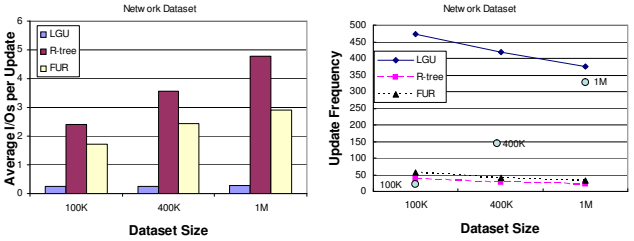

Figure 10: Update I/Os and frequency vs. dataset size (R-tree)

Fig. 9 (left) provides average I/Os per update for the network dataset. It is clear that the average I/Os per update of LGU is much less than 1. This is hard to achieve without grouping updates since usually an individual update takes at least 2 I/Os (one for reading and one for writing). Fig. 10 (left) shows a similar picture for LGU's performance in R-trees.

Fig. 9 (right) and 10 (right) represent the number of objects in the dataset and the largest update frequency an algorithm can handle. The three points labelled "100K", "400K", and "1M" are the application scenarios described in §2. It is obvious that LGU can easily satisfy the demands of the applications with all three sizes, while other algorithms can only meet the requirement of the "100K" case.

Finally, TPRK and TPR perform similarly and the improvement of FUR-tree over R-tree is not very significant. The reasons are (1) TPR is compensated with a large system buffer for the lookup table used in TPRK, and (2) both R-tree and FUR-tree have large system buffers, which weakens the effectiveness of the main memory summary structure used in FUR-tree.

6.3 Push Down Strategies

We find that Push-L always has the best performance among all the strategies, Push-Mp is slightly worse than Push-L, and Push-All is always the worst one.

Fig. 11 and 12 show the performance of different push down strategies on the network dataset (the results of two other datasets are very similar). Among the three strategies, Push-L achieves the best performance. Push-MP has similar performance (85%-90% of Push-L) as Push-L and the naive Push-All strategy is the worst (50%-60% of Push-L). Note that all three strategies achieve much better update performance than other algorithms.

6.4 Impact of I-Buffer Size

We observe that the update performance increases with I-Buffer size, while query performance decreases slightly.

We only present results for TPR-trees, results for R-trees are similar. We show both update and query throughput. The analysis result of §5 indicates that larger I-Buffers will increase update throughput and decrease query throughput. From Equation 6 and the experimental parameters provided in §6.1 we can compute the optimal I-Buffer size of the TPR-tree is 42.

The results in Fig. 13 and 14 confirm this prediction from the analytical model. Since I-Buffers are only attached to internal nodes, which are a small fraction of the tree nodes, query throughput does not decrease much when the I-Buffer size increases.

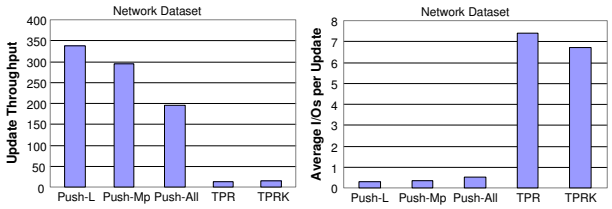

Figure 11: Update performance of pushdown strategies (TPR)

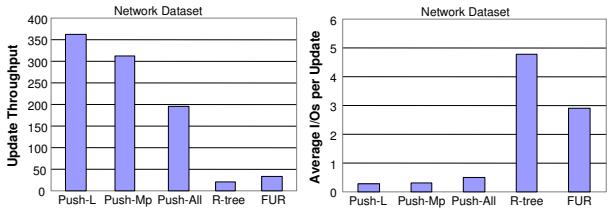

Figure 12: Update performance of pushdown strategies (R)

LGU with different I-Buffer size outperforms TPR/TPRK significantly in update throughput. The maximum improvement is observed for 100-page I-Buffers, at about 100 times faster than TPR/TPRK. We observe that the query performance in the network dataset is much better than that in the uniform dataset, due to restrictions on point locations.

We also compute the expected update (query) I/Os and compare them with the experiment results (Fig. 15 and 16). The expected query I/Os are close to the experiment results. But there is a gap between expected update I/Os and the experiment results. The reasons of this difference include: (1) The theoretical analysis assumes all entries go to the leaf level, which is an over-estimate since some entries are deleted directly from I-Buffers and some just stay in I-Buffers. (2) In the analysis, it is assumed that each time an I-Buffer is accessed, all pages in the I-Buffer are accessed. But this is not always true, especially for larger I-Buffers. (3) System buffers are not taken into consideration in the analysis, which reduces the update I/Os. (4) The average group size observed in experiments is greater than the approximation in the analysis.

6.5 Combined Update and Query Throughput

When the incoming query rate increases, we observe the overall throughput of LGU decreases mildly. But in all cases studied, LGU still out performs all other algorithms.

In this set of experiments, the incoming update rate is invariant while the incoming query rate is increased from 20 per time unit to 400 per time unit. Fig. 17 and 18 show that when the frequency of queries increases, the improvement of overall throughput achieved

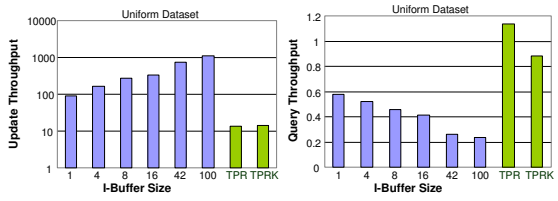

Figure 13: Update/Query Throughput vs I-Buffer Size (TPR)

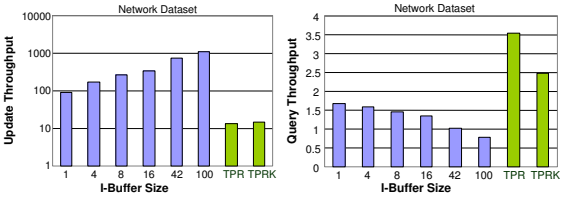

Figure 14: Update/Query Throughput vs. I-Buffer Size (TPR)

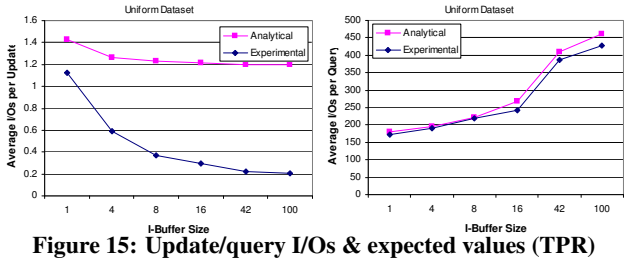

Figure 15: Update/query I/Os & expected values (TPR)

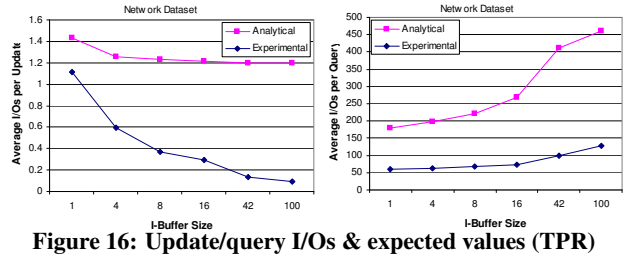

Figure 16: Update/query I/Os & expected values (TPR)

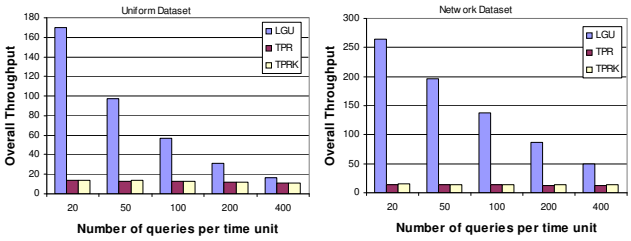

Figure 17: Overall throughput vs query frequencies (TPR)

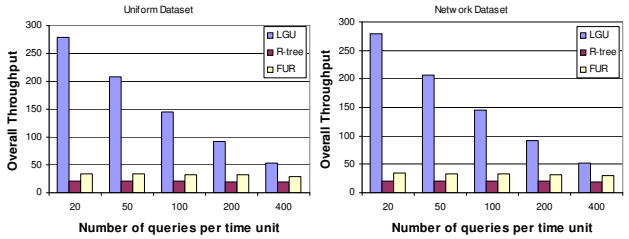

Figure 18: Overall throughput vs query frequencies (R-tree)

by LGU decreases. But in most cases studied, LGU still outperforms TPR and TPRK. And the overall improvement of LGU is higher in the network dataset.

7. CONCLUSIONS AND FUTURE WORK

In this paper we study the problem caused by frequent updates in disk-based index structures and introduce the LGU algorithm. LGU focuses on utilizing memory wisely by performing group insertions and deletions to reduce the I/O cost of a disk-based index. (Although LGU increase CPU cost, preliminary analysis indicates that the I/O cost remains dominant.) Future work may lead to better analytical model (e.g. with a more accurate assumption about entry deletion). It is clear that the group update technique will have its limitations; when the incoming update rate is extremely high, one has to consider other approaches such as main memory based algorithms.

8. REFERENCES

[1] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data structures*, 1995.

[2] L. Arge. Private communications, 2004.

[3] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *In Proc. of 1st Workshop on Algorithm Engineering and Experimentation*, 1999.

[4] B. Becker, S. Gschwind, T. Ohler, et al. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4), 1996.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, 1990.

[6] T. Brinkhoff. Generating traffic data. *Bulletin on Data Engineering, IEEE Computer Society*, 26(2):19–25, 2003.

[7] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A generalized R-Tree bulk-insertion strategy. In *Proc. SLSD*, 1999.

[8] R. Güting, M. Böhlen, M. Erwig, et al. A foundation for representing and querying moving objects. *ACM TODS*, 25(1), 2000.

[9] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, 1984.

[10] G. Kollios, D. Papadopoulos, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects using dual transformations. In *VLDB Journal*, 2005.

[11] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM PODS*, 1999.

[12] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1), 1998.

[13] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update R-tree. In *Proc. MDM*, 2002.

[14] K. S. Larsen. Relaxed multi-way trees with group updates. In *Proc. PODS*, 2001.

[15] M. L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting frequent updates in R-trees: a bottom-up approach. In *Proc. VLDB*, 2003.

[16] L. Malmi and E. S. Soininen. Group updates for relaxed height-balanced trees. In *Proc. PODS*, 1999.

[17] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *Proc. PODS*, 2002.

[18] S. W. Ng. Advances in disk technology : performance issues. *IEEE Computer*, May 1998.

[19] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects on the plane. In *DEXA*, 2002.

[20] M. Raab and A. Steger. Balls into bins – a simple and tight analysis. In *RANDOM*, 1998.

[21] R. Ramakrishnan et al. Science of design for information systems. *ACM SIGMOD Record*, 33(1), 2004.

[22] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. *ACM SIGMOD*, 2000.

[23] J. Zhou and Ke. A. Ross. Buffering Accesses to Memory-Resident Index Structures. *VLDB*, 2003.

[24] S. Saltenis. http://www.cs.auc.dk/ simas/tpr/tpr.tar.gz.

[25] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *Proc. ACM SIGMOD*, 2000.

[26] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, 1987.

[27] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. ICDE*, 1997.

[28] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proc. MDM*, 2001.

[29] Z. Song and N. Roussopoulos. k-nearest neighbor search for moving query point. In *Proc. SSTD*, 2001.

[30] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[31] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *The Computer Journal*, 41(3), 1998.

[32] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. PODS*, 1996.

[33] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. *In Proc. SSD*, 1999.

[34] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Proc. SSDBM*, 1998.

[35] H. Zhu, J. Su, and O. Ibarra. Trajectory queries and octagons in moving object databases. In *Proc. ACM CIKM*, 2002.