2005

R-trees with Update Memos

Xiaopeng Xiong

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Report Number:
05-020

R-TREES WITH UPDATE MEMOS

Xiaopeng Xiong
Walid G. Aref

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #05-020
October 2005

R-trees with Update Memos

Xiaopeng Xiong Walid G. Aref

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{xxiong, aref}@cs.purdue.edu

Abstract

The problem of frequently updating multi-dimensional indexes arises in many location-dependent applications. While the R-tree and its variants are one of the dominant choices for indexing multi-dimensional objects, the R-tree exhibits inferior performance in the presence of frequent updates. In this paper, we present an R-tree variant, termed the RUM-tree *(stands for R-tree with Update Memo) that minimizes the cost of object updates. The RUM-tree processes updates in a* memo-based *approach that avoids disk accesses for purging old entries during an update process. Therefore, the cost of an update operation in the RUM-tree reduces to the cost of only an insert operation. The removal of old object entries is carried out by a* garbage cleaner *inside the RUM-tree. In this paper, we present the details of the RUM-tree and study its properties. Theoretical analysis and experimental evaluation demonstrate that the RUM-tree outperforms other R-tree variants by up to a factor of eight in scenarios with frequent updates.*

1. Introduction

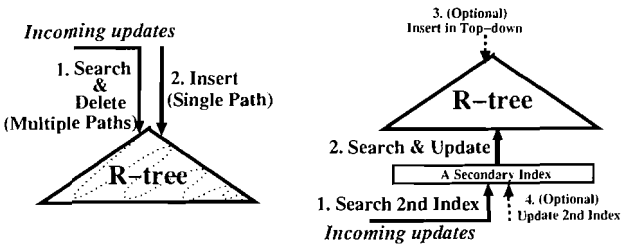
With the advances in positioning systems and wireless devices, spatial locations of moving objects can be sampled continuously to database servers. Many emerging applications require to maintain the latest positions of moving objects. In addition, a variety of potential applications rely on monitoring multidimensional items that are sampled continuously. Considering the fact that every sampled data value results in an update to the underlying database server, it is essential to develop spatial indexes that can handle frequent updates in efficient and scalable manners.

As one of the primary choices for indexing low-dimensional spatial data, the R-tree [6] and the R*-tree [1] exhibit satisfactory search performance in traditional databases when updates are infrequent. However, due to the costly update operation, R-trees are not practically applicable to situations with enormous amounts of updates. Improving the R-trees' update performance is an important yet challenging issue.

Two approaches exist to process updates in R-trees, namely, the top-down approach and the bottom-up approach. The top-down approach was originally proposed in [6] and has been adopted in many R-tree variants, e.g., [1, 8, 17, 21]. This approach treats an update as a combination of a separate deletion and a separate insertion. Firstly, the R-tree is searched from the root to the leaves to locate and delete the data item to be updated. Given the fact that R-tree nodes may overlap each other, such search process is expensive as it may follow multiple paths before it gets to the right data item. After the deletion of the old data item, a single-path insertion procedure is invoked to insert the new data item into the R-tree. Figure 1(a) illustrates the top-down update process. The top-down approach is rather costly due to the expensive search operation.

Recently, new approaches for updating R-trees in a bottom-up manner have been proposed [10, 11]. The bottom-up approach starts the update process from the leaf node of the data item to be updated. The bottom-up approach tries to insert the new entry into the original leaf node or to the sibling node of the original leaf node. For fast access of the leaf node of a data item, a secondary index such as a direct link [10] or a hash table [11] is maintained on the identifiers for all objects. Figure 1(b) illustrates the bottom-up update process. The bottom-up approach exhibits better update performance than the top-down approach when the change in an object between two consecutive updates is small. In this case, the new data item is likely to remain in the same leaf node. However, the performance of the bottom-up approach degrades quickly when the change between consecutive updates becomes large. Moreover, a secondary index may not fit in memory due to its large size, which may incur significant maintenance overhead to the update procedure. Note that the secondary index needs to be updated whenever an object moves from one leaf node to another.

In this paper, we propose the RUM-tree (stands for R-tree with Update Memo), an R-tree variant that handles



(a) Top-down Update Approach (b) Bottom-up Update Approach

Figure 1. Existing R-tree Update Approaches

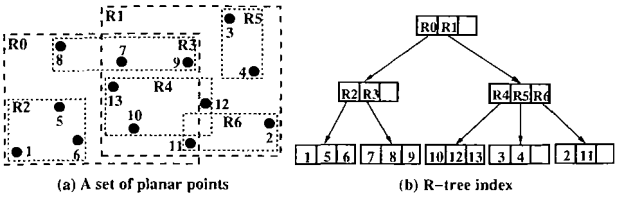


Figure 2. Example of R-tree

object updates efficiently. A *memo-based* update approach is utilized to minimize the update cost in the RUM-tree. The memo-based update approach enhances the R-tree by an *Update Memo* structure. The Update Memo eliminates the need to delete the old data item from the index during an update. Therefore, the total cost for update processing is reduced dramatically. Compared to R-trees with a top-down or a bottom-up update approach, the RUM-tree has the following distinguishing advantages: (1) The RUM-tree achieves significantly lower update cost while offering similar search performance; (2) The update memo is much smaller than the secondary index used by other approaches. The *garbage cleaner* guarantees an upper-bound on the size of the Update Memo making it practically suitable for main memory; (3) The update performance of the RUM-tree is stable with respect to the changes between consecutive updates, the extents of moving objects, and the number of moving objects.

The contributions of the paper can be summarized as follows:

- We propose the RUM-tree that minimizes the update cost while yielding similar search performance to other R-tree variants;
- We address the issues of crash recovery and concurrency control for the proposed RUM-tree, especially when the Update Memo is memory-based;
- We analyze the update costs for the RUM-tree and the other R-tree variants, and derive an upper-bound on the size of the Update Memo;
- A comprehensive set of experiments is presented. The performance results indicate that the RUM-tree outperforms other R-tree variants by up to a factor of 8 for frequent updates.

The remainder of the paper is organized as follows. Section 2 overviews the R-tree and summarizes related work. In Section 3, we present the RUM-tree. In Section 4, we give a cost analysis of the memo-based approach and compare it with the top-down and the bottom-up approaches. We derive an upper-bound for the size of the Update Memo in Section 4. Experiments are presented in Section 5. Finally, Section 6 concludes the paper.

2. R-tree-based Indexing and Related Work

The R-tree [6] is a height-balanced indexing structure. It is an extension to the B-tree in the multidimensional space. In an R-tree, spatial objects are clustered in nodes according to their Minimal Bounding Rectangles (MBRs). In contrast to the B-tree, the R-tree nodes are allowed to overlap each other. An entry in a leaf node is of the form: (MBR_o, p_o), where MBR_o is the MBR of the indexed spatial object, and p_o is a pointer to the actual object tuple in the database. An entry in an internal node is of the form: (MBR_c, p_c) where MBR_c is the MBR covering all MBRs in its child node, and p_c is the pointer to its child node c. The number of entries in each R-tree node, except for the root node, is between two specified parameters m and M ($m \leq \frac{M}{2}$). The parameter M is termed the *fanout* of the R-tree. Figure 2 gives an R-tree example with a fanout of 3 that indexes thirteen objects.

In the last two decades, several R-tree variants have been proposed, e.g., [1, 8, 17, 21]. With the recent attention on indexing moving objects, a number of R-tree-based methods for indexing moving objects have been proposed. They focus on one of the following approaches: (1) Indexing the historical trajectories of objects, e.g., [3, 7, 12, 13, 22, 23, 25]; (2) Indexing the current locations of objects, e.g., [5, 9, 14, 15, 18, 19]; and (3) Indexing the predicted trajectories of objects, e.g., [16, 20, 24]. Most of these works assume that the updates are processed in a top-down manner. The *memo-based* update technique presented in this paper is applicable to most of these works to improve their update performance.

To support frequent updates in R-trees, [10] and [11] propose a bottom-up update approach. This approach processes an update from the leaf node of the old entry, and tries to insert the new entry into the same leaf node or to its sibling node. The bottom-up approach works well when the consecutive changes of objects are small. However, in the case that consecutive changes are large, their performance degrades quickly. In Section 5, we show that the proposed memo-

based update approach of the RUM-tree outperforms the bottom-up approach and is more stable under various parameters.

3. The RUM-tree Index

In the existing update approaches, the deletion of old entries incurs overhead in update processing. In the top-down approach, the deletion involves searching in multiple paths. In the bottom-up approach, a secondary index is maintained to locate and delete an entry. In this section, we present the RUM-tree that minimizes additional disk accesses for such deletion and thus minimizes the update cost.

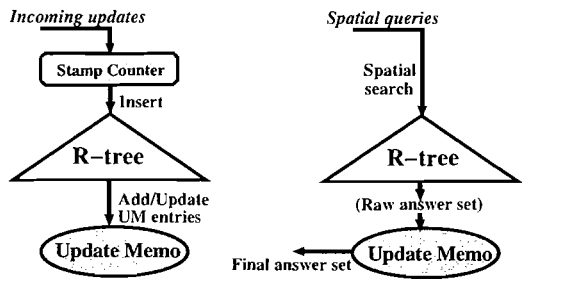
The primary feature behind the RUM-tree is as follows. As an update occurs, the old entry of the data item is not required to be removed. Instead, the old entry is allowed to co-exist with newer entries before it is removed later. Only one entry of an object is the most recent entry (referred to in the paper as the *latest* entry), and all other entries of the object are old entries (referred to in the paper as *obsolete* entries). The RUM-tree maintains an *Update Memo* to identify the latest entries from obsolete entries. These obsolete entries are identified and are removed from the RUM-tree by a *garbage cleaner* mechanism.

In Section 3.1, we describe the RUM-tree structure. In Section 3.2, we discuss the insert, update, delete, and query algorithms of the RUM-tree. The garbage cleaner is introduced in Section 3.3. Logging and crash recovery algorithms are presented in Section 3.4. Finally, we discuss concurrency control issues in Section 3.5.

3.1. The RUM-tree Structure

In the RUM-tree, each leaf entry is assigned a *stamp* when the entry is inserted into the tree. The stamp is assigned by a global *stamp counter* that increments monotonically. The stamp of one leaf entry is globally unique in the RUM-tree and remains unchanged once assigned. The stamp places a temporal relationship among leaf entries, i.e., an entry with a smaller stamp was inserted before an entry with a larger stamp. Accordingly, the leaf entry of the RUM-tree is extended to the form $(MBR_o, p_o, oid, stamp)$, where oid is the identifier of the stored object, $stamp$ is the assigned stamp number, and MBR_o and p_o are the same as in the standard R-tree.

The RUM-tree maintains an auxiliary structure, termed the *Update Memo* (UM, for short). The main purpose of UM is to distinguish the obsolete entries from the latest entries. UM contains entries of the form: $(oid, S_{latest}, N_{old})$, where oid is an object identifer, S_{latest} is the $stamp$ of the *latest* entry of the object oid, and N_{old} is the maximum number of *obsolete* entries for the object oid in the RUM-tree. As an example, a UM entry $(O_{99}, 1000, 2)$ entails that in the RUM-tree there exist at most two *obsolete* entries for the object O_{99}, and that the *latest* entry of O_{99} bears the *stamp* of 1000. Note that no UM entry has N_{old} equivalent to zero, namely, objects that are assured to have no obsolete entries in the RUM-tree do not own a UM entry. To accelerate searching, the update memo is hashed on the oid attribute. With the garbage cleaner provided in Section 3.3, the size of UM is kept rather small and can practically fit in main memory of nowadays machines. We derive the upper-bound for the size of UM in Section 4. The size of UM is further studied through experiments in Section 5.



(a) Insert/Update Process (b) Query Process

Figure 3. Operations in the RUM-tree

3.2. Insert, Update, Delete, and Search Algorithms

3.2.1. Insert and Update Inserting an entry and updating an entry in the RUM-tree follow the same procedure as illustrated in Figure 3(a). Pseudo-code for the insert/update algorithm is given in Figure 4. Firstly, an insert/update is assigned a stamp number when it reaches the RUM-tree. Then, along with the stamp and the object identifier, the new value is inserted into the RUM-tree using the standard R-tree insert algorithm [1]. After the insertion, the entry that has been the *latest* entry, if exists, for the inserted/updated object becomes an *obsolete* entry. To reflect such a change, the UM entry for the object is updated as follows. The UM entry of the object, if exists, changes S_{latest} to the *stamp* of the inserted/updated tuple and increments N_{old} by 1. In the case that no UM entry for the object exists, a new UM entry with the *stamp* of the inserted/updated tuple is inserted. N_{old} of the UM entry is set to 1 to indicate up to one obsolete entry in the RUM-tree. The old value of the object being updated is not required, which potentially reduces the maintenance cost of database applications.

3.2.2. Delete Deleting an object in the RUM-tree is equivalent to marking the latest entry of the object as obsolete. Figure 5 gives pseudo-code for the deletion algorithm. The object to be deleted is treated as an update to a special loca-

Algorithm MemoBasedInsert($oid, newLocation$)

1. $newTuple = (oid. newLocation)$;
2. $stamp \leftarrow StampCounter$; Increment $StampCounter$;
3. Insert $newTuple$ to the RUM-tree;
4. Let ne be the inserted leaf entry for $newTuple$, $ne.oid \leftarrow oid$, $ne.stamp \leftarrow stamp$;
5. Search oid in Update Memo UM.
   If no entry is found, insert $(oid, stamp, 1)$ to UM;
   Otherwise, let $umne$ be the found UM entry; $umne.S_{latest} \leftarrow stamp$; Increment $umne.N_{old}$;

Figure 4. Insert/Update in the RUM-tree

Algorithm MemoBasedDelete(oid)

- $stamp \leftarrow StampCounter$; Increment $StampCounter$;
- Search oid in Update Memo UM.
  If no entry is found, insert $(oid, stamp, 1)$ to UM;
  Otherwise, let $umne$ be the found UM entry; $umne.S_{latest} \leftarrow stamp$; Increment $umne.N_{old}$;

Figure 5. Delete in the RUM-tree

tion. The special update does not actually go through the R-tree. It only affects the UM entry for the object to be deleted, if exists, by changing S_{latest} to the next value assigned by the stamp counter, and incrementing N_{old} by 1. In the case when no UM entry for the given object exists, a new UM entry is inserted whose S_{latest} is set to the next stamp number and N_{old} is set to 1. In this way, all entries for the given object will be identified as obsolete and consequently will get removed by the garbage cleaner.

3.2.3. Search Figure 3(b) illustrates the processing of spatial queries in the RUM-tree. As the obsolete entries and the latest entry for one object may co-exist in the RUM-tree, the output satisfying the spatial predicates is a superset of the actual answers. In the RUM-tree, UM is utilized as a *filter* to purge false answers, i.e., UM filters obsolete entries out of the answer set. By adding such an additional *filter* step, any existing query processing algorithms in other R-tree variants can apply directly to the RUM-tree. The RUM-tree employs the algorithm given in Figure 6 to identify a leaf entry as *latest* or *obsolete*. The main idea is to compare the stamp of the leaf entry with the S_{latest} of the corresponding UM entry. Recall that S_{latest} of a UM entry is always the stamp of the latest entry of the corresponding object. If the stamp of the leaf entry is smaller than S_{latest} of the UM entry, the leaf entry is obsolete for the object; otherwise it is the latest entry for the object. In the case that no corresponding UM entry exists, the leaf entry is the latest entry.

Discussion. Sanity checking can be done at a higher level before invoking the index. The RUM-tree does not check the existence of an old entry when performing insert, update or delete. Thus, the RUM-tree may insert an object that already exists in the index or delete/update an object that never existed. However, based on the above algorithms, regardless of whether sanity checking is performed or not, the RUM-tree will always return only the correct latest insert/update values to queries. A related issue of *phantom* entries is addressed in Section 3.3.2.

Algorithm CheckStatus($leafEntry$)

1. Search $leafEntry.oid$ in UM. If no entry is found, return $LATEST$; Otherwise, let ume be the found UM entry;
2. If $(leafEntry.stamp == ume.S_{latest})$, return $LATEST$; Otherwise, return $OBSOLETE$;

Figure 6. Checking Entry Status

3.3. Garbage Cleaning

The RUM-tree employs a *Garbage Cleaner* to limit the number of obsolete entries in the tree and to limit the size of UM. The garbage cleaner deletes the obsolete entries *lazily* and in *batches*. Deleting *lazily* means that obsolete entries are not removed immediately; Deleting in *batches* means that multiple obsolete entries in the same leaf node are removed at the same time.

3.3.1. Cleaning Tokens A *cleaning token* is a logical token that traverses all leaf nodes of the RUM-tree horizontally. The token is passed from one leaf node to the next every time when the RUM-tree receives a certain number of updates. Such number is termed the *inspection interval* and is denoted by I. The node holding a cleaning token inspects all entries in the node and cleans its obsolete entries, and then passes the token to the next leaf node after I updates. To locate the next leaf node quickly, the leaf nodes of the RUM-tree are doubly-linked in cycle. In addition, each RUM-tree node maintains a pointer to its parent node. This is for the purpose of adjusting the RUM-tree in a bottom-up manner after the obsolete entries in a leaf node are removed.

Figure 8 gives the pseudo code of the cleaning procedure. Every entry in the inspected leaf node is checked by $CheckStatus()$ given in Figure 6, and is deleted from the node if the entry is identified as obsolete. When an entry is removed, N_{old} of the corresponding UM entry is decremented by one. When N_{old} reaches zero, indicating that no obsolete entries exist for this object, the UM entry is deleted. In occasional cases, the leaf node may underflow due to the deletion of obsolete entries. In this situation, the remaining entries of the leaf node are reinserted to the RUM-tree using the standard R-tree insert algorithm. If the leaf node does not underflow, the MBR of the inserted leaf node and the MBRs of its ancestor nodes are adjusted.

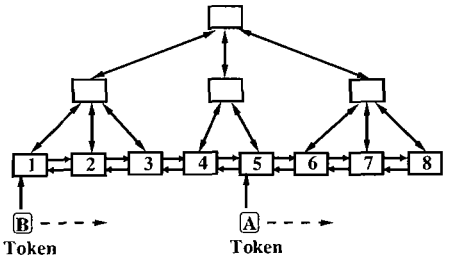


Figure 7. Garbage Cleaner: Cleaning Tokens

Algorithm Clean($leaf node N$)

1. For each entry e in N, if CheckStatus(e) returns $OBSOLETE$,
   (a) Delete e from N;
   (b) Let ume be the UM entry for $e.oid$, Decrement $ume.N_{old}$; If $ume.N_{old}$ equals 0, delete ume from UM;
2. If the number of entries in N is less than $MIN_{ENTRIES}$, reinsert the remaining entries of N into the RUM-tree; Otherwise, adjust the MBRs of N and N's ancestors in a bottom-up manner;

Figure 8. Cleaning A Leaf Node

To speed up the cleaning process, multiple cleaning tokens may work in parallel in the garbage cleaner. In this case, each token serves a subset of the leaf nodes. Figure 7 illustrates a RUM-tree with two cleaning tokens. Token A inspects Nodes 5 to 8 while Token B inspects Nodes 1 to 4. Tokens move either with the same inspection interval or with different inspection intervals. Note that each cleaning token incurs additional disk accesses to the cleaning procedure. Hence, there is a tradeoff between the cleaning effect and the overall cost.

We define the *garbage ratio* (gr) of the RUM-tree and the *inspection ratio* (ir) of the garbage cleaner as follows. The garbage ratio of the RUM-tree is the number of obsolete entries in the RUM-tree over the number of indexed moving objects. The garbage ratio reflects how clean the RUM-tree is. A RUM-tree with a small garbage ratio exhibits better search performance than a RUM-tree with a large garbage ratio.

The inspection ratio ir of the garbage cleaner is defined as the number of leaf nodes inspected by the cleaner over the total number of updates processed in the RUM-tree during a period of time. The inspection ratio represents the cleaning frequency of the cleaner. A larger inspection ratio results in a smaller garbage ratio for the RUM-tree. Assume that a RUM-tree has m cleaning tokens t_1 to t_m, and that t_k's inspection interval is I_k for $1 \leq k \leq m$, then ir of the cleaner is calculated as:

$$\begin{aligned} ir &= \frac{\frac{U}{I_1}+\frac{U}{I_2}+\cdots+\frac{U}{I_m}}{The\ total\ number\ of\ updates\ U} \\ &= \frac{1}{I_1}+\frac{1}{I_2}+\cdots+\frac{1}{I_m} \\ &= \frac{m}{I}\ (if\ I_1 = I_2 = \cdots = I_m = I) \end{aligned} \quad (1)$$

The cleaning token approach has the following important property.

Property 1: *Let O_t be the set of obsolete entries in the RUM-tree at time t. After every leaf node has been visited and cleaned once since t, all entries in O_t are removed out of the RUM-tree.*

Property 1 holds no matter whether there are new inserts/updates during the cleaning phase or not. Note that if some entries become obsolete due to new inserts/updates, these newly introduced obsolete entries are not contained in O_t. The proof of Property 1 is straightforward, because when a leaf node is visited by the garbage cleaner, all obsolete entries in the leaf node will be identified and cleaned.

3.3.2. Phantom Inspection In this section, we address the issue of cleaning *phantom* entries in the RUM-tree. A phantom entry is a UM entry whose N_{old} is larger than the exact number of obsolete entries for the corresponding object on the RUM-tree. Such an entry will never get removed from the UM because its N_{old} never returns to zero. Phantom entries are caused by performing operations on objects that do not exist in the RUM-tree, e.g., updating/deleting an object that does not exist in the RUM-tree. A special case is when inserting a new object to the RUM-tree[1].

The RUM-tree employs a *Phantom Inspection* procedure to detect and remove phantom entries. According to Property 1 in Section 3.3.1, we have the following lemma.

Lemma 1. *Let c be the value of the stamp counter at time t. After every leaf node has been visited and cleaned once since t, a UM entry whose S_{latest} is less than c is a phantom entry.*

Otherwise, if such a UM entry is not a phantom entry, by Property 1, it should have been removed out of UM after every leaf page has been visited and cleaned. Therefore, Lemma 1 holds.

Based on Lemma 1, the phantom inspection procedure works periodically. The current value of the stamp counter is stored as c. After the cleaning tokens traverse all leaf nodes once, the procedure inspects UM and removes all UM entries whose S_{latest} is less than c. Finally, c is updated for the next cycle's inspection. In this way, all phantom entries will be removed after one cycle of cleaning.

1 Recall that in the RUM-tree, an insert is handled in the same way as an update. Inserting an entry incurs a new UM entry anyway.

3.3.3. Clean Upon Touch Besides the cleaning tokens, garbage cleaning can be performed whenever a leaf node is accessed during an insert/update. The cleaning procedure is the same as in Figure 8. As a side effect of insert/update, such clean-upon-touch process does not incur extra disk accesses. When working with the cleaning tokens, the clean-upon-touch reduces the garbage ratio and the size of UM dramatically.

3.4. Crash Recovery

In this section, we address the recovery issue of the RUM-tree in the case of system failure. UM is in main-memory. When the system crashes, the data in UM is lost. The goal is to rebuild UM based on the tree on disk upon recovery from failure. We consider three approaches with different tradeoffs between the recovery cost and the logging cost.

Option I: Without log. In this approach, no log is maintained. When recovering, an empty UM is first created. Then, every leaf entry in the tree is scanned. If no UM entry exists for a leaf entry, a new UM entry is inserted. Otherwise, S_{latest} and N_{old} of the corresponding UM entry are updated continuously during the scan. The value of the stamp counter before the crash can also be recovered during the scan. The UM entries having N_{old} equal to zero are removed out of UM, and the resulting UM is the original UM before the crash. In this approach, the intermediate UM is possibly large in size depending on the number of moving objects.

Option II: With UM log at checkpoints. In this approach, UM and the current value of the stamp counter are written to log periodically at checkpoints. Since UM is small, the logging cost on average is low. When recovering, the UM from the most recent checkpoint is retrieved. Then, the UM is updated continuously in the same way as in Option I. However, only the leaf entries that are inserted/updated after the checkpoint will be processed. The resulting UM is a superset of the original UM due to having ignored the removed leaf entries since the checkpoint. This causes *phantom* entries as discussed in Section 3.3.2. Inspecting UM will lead to the original UM after one clean cycle.

Option III: With memo log at checkpoints and log of memo operations. This approach requires writing UM to log at each checkpoint and logging any changes to it after the checkpoint. At the point of recovery, UM at the latest checkpoint is retrieved and is updated according to the logged changes. Despite high logging cost, the recovery cost in this option is the cheapest as it avoids the need to scan the disk tree.

3.5. Concurrency Control

Dynamic Granular Locking (DGL) [4] has been proposed to provide concurrency in R-trees. DGL defines a set of lockable node-level granules that can adjust dynamically during insert, delete, and update operations. DGL can directly apply to the on-disk tree of the RUM-tree. Consider that the RUM-tree utilizes the standard R-tree insert algorithm in the insert and update operations. For deletion, garbage cleaning is analogous to deleting multiple entries from a leaf node.

Besides the on-disk tree, the hash-based UM and the stamp counter are also lockable resources. Each hash bucket of UM is associated with a *read* lock and a *write* lock. A bucket is set with the proper lock when accessed. Similarly, the stamp counter is associated with such read/write locks. The DGL and the read/write locks work together to guarantee concurrent accesses in the RUM-tree.

4. Cost Analysis

Let N be the number of leaf nodes in the RUM-tree, E be the size of the UM entry, ir be the inspection ratio of the garbage cleaner, P be the node size of the RUM-tree, C be the number of updates between two checkpoints, and M be the number of indexed moving objects.

4.1. Garbage Ratio and the Size of UM

We start by analyzing the garbage ratio and the size of UM. According to Property 1, after every leaf node is visited and is cleaned once, all obsolete entries that exist before the cleaning are removed. In the RUM-tree, every leaf node is cleaned once during $\frac{N}{ir}$ inserts/updates. In the worst case, $\frac{N}{ir}$ obsolete entries are newly introduced in the RUM-tree. Therefore, the upper-bound for the garbage ratio is $\frac{N}{ir*M}$. As each obsolete entry may own an independent UM entry, the upper-bound for the size of UM is $\frac{N*E}{ir}$.

It is straightforward to prove that the average garbage ratio is $\frac{N}{2ir*M}$, and that the average size of UM is $\frac{N*E}{2ir}$. This result implies that the garbage ratio and the size of UM are related to the number of leaf nodes that is far less than the number of indexed objects. Thus, the garbage ratio and the size of UM are kept small, and UM can reasonably fit in main memory. With the clean-upon-touch optimization, the garbage ratio and the size of UM can be further reduced, as we show in Section 5.

4.2. Update Cost

We analyze the update costs for the top-down, the bottom-up, and the memo-based update approaches. We in-

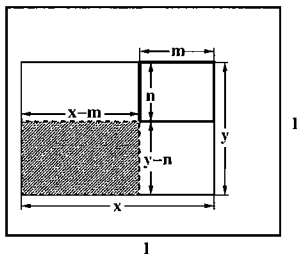


Figure 9. Probability of Window Containment

vestigate the number of disk accesses under realistic scenarios. Practically, the internal R-tree nodes are cached in the memory buffer. Therefore, our analysis focuses on the disk accesses for leaf nodes. In the following discussion, the data space is normalized to a unit square. Node underflow and overflow are ignored in all approaches as they happen quite rarely.

4.2.1. Cost of the Top-down Approach The cost of a top-down update consists of two parts, namely, (1) the cost of searching and deleting the old entry and (2) the cost of inserting the new entry. Unlike [11], we notice that an entry can be found only in nodes whose MBRs fully contain the MBR of this entry. To deduce the search cost, we present the following lemma:

Lemma 2. In a unit square, let W_{xy} be a window of size $x * y$, and let W_{mn} be a window of size $m * n$. When W_{xy} and W_{mn} are randomly placed, the probability that W_{xy} contains W_{mn} is given by:

$$max(x - m, 0) * max(y - n, 0)$$

Proof. Assume that the position of W_{xy} is fixed as shown in Figure 9. Then, W_{mn} is contained in W_{xy} if and only if W_{mn}'s bottom-left vertex lies in the shaded area. The size of the shaded area is given by $max(x - m, 0) * max(y - n, 0)$. Since W_{mn} is randomly placed, the probability of W_{xy} containing W_{mn} is also $max(x - m, 0) * max(y - n, 0)$. For arbitrary placement of W_{xy}, the above situation holds. Hence we reach Lemma 2.

Assume that the MBR of the entry to be deleted is given by $a * b$, where $0 \leq a, b \leq 1$. From Lemma 2, the expected number of leaf node accesses for searching the old entry is given by:

$$IO_{search} = \frac{1}{2} \sum_{i=1}^{N} (max(x_i - a, 0) * max(y_i - b, 0))$$

where x_i and y_i are the width and the height of the MBR of the ith leaf node. Once the entry is found, it is deleted and the corresponding leaf node is written back. In addition, inserting a new entry involves one leaf node read and one leaf node write. Therefore, the expected number of node accesses for the top-down approach is:

$$IO_{TD} = \frac{1}{2} \sum_{i=1}^{N} (max(x_i - a, 0) * max(y_i - b, 0)) + 3$$

4.2.2. Cost of the Bottom-up Approach The cost of the bottom-up approach, as we explain below, ranges from three to seven leaf node accesses depending on the placement of the new data.

If the new entry remains in the original node, the update cost consists of three disk accesses: reading the secondary index to locate the original leaf node, reading the original leaf node, and writing the original leaf node.

When the new entry is inserted into some sibling of the original node, the update cost consists of six disk accesses: reading the secondary index, reading and writing the original leaf node, reading and writing the sibling node, and writing the changed secondary index.

In the case that the new entry is inserted into any other node, the update cost consists of seven disk accesses: reading the secondary index, reading and writing the original leaf node, reading and writing the inserted node, writing the changed secondary index, and writing the adjusted parent node of the inserted node.

4.2.3. Cost of the Memo-based Approach For the memo-based approach, each update is directly inserted. Inserting an entry involves one leaf node read and one leaf node write. Given the inspection ratio ir, for a total number of U updates, the number of leaf nodes inspected by the cleaner is $U * ir$. Each inspected leaf node involves one node read and one node write. The clean-upon-touch optimization does not involve extra disk accesses. Therefore, the overall cost per update in the memo-based update approach is $2(1 + ir)$ disk accesses.

As discussed in Section 3.4, various recovery approaches involve different logging costs. Option I does not involve any logging cost. Based on the upper-bound of the size of UM derived in Section 4.1, the additional logging cost per update in Option II is $\frac{N*E}{ir*P*C}$. For Option III, the additional logging cost per update is $(\frac{N*E}{ir*P*C} + 1)$.

5. Experimental Evaluation

In this section, we study the performance of the RUM-tree through experiments and compare the performance with the R*-tree [1] and the Frequently Updated R-tree (FUR-tree) [11].

All the experiments are running on an Intel Pentium IV machine with CPU 3.2GHz and 1GB RAM. In the experiments, the number of moving objects ranges between 2 million and 20 million objects. The object set is generated by the *Network-based Generator of Moving Objects* [2]. We

use the road map of Los Angeles in the generator and normalize the road map to a unit square. The extent of the objects ranges between 0 (i.e., points) and 0.01 (i.e., squares with side 0.01). Each object issues an update periodically with a predefined moving distance between 0 and 0.01. For the search performance, we study the performance of range queries. The number of the queries is fixed at 100,000 queries. The queries are square regions of side length 0.03. The primary parameters used in the experiments are outlined in Table 1, where the default values are given in bold fonts.

As the primary metric, the number of disk accesses is investigated in most experiments. As discussed in Section 4, the internal R-tree nodes are cached in memory buffers for all the R-tree types. For the FUR-tree, the MBRs of the leaf nodes are allowed to extend 0.003 to accommodate object updates in their original nodes. For the RUM-tree, we implement both the original cleaning-token garbage cleaner (denoted by the RUM-tree$_{token}$ in this section) and the optimized clean-upon-touch cleaner (denoted by the RUM-tree$_{touch}$ in this section). Except the experiments in Section 5.5, Option II discussed in Section 3.4 for the RUM-tree is chosen as the default recovery option.

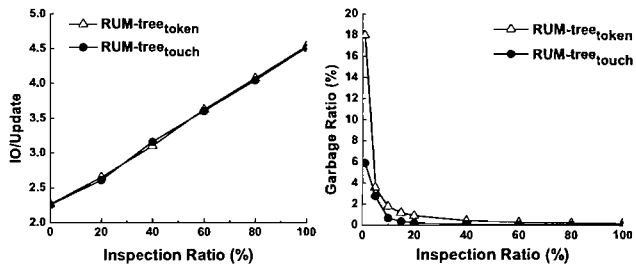
5.1. Properties of the RUM-tree

In this section, we study the properties of the RUM-tree under various inspection ratios and various node sizes.

5.1.1. Effect of Inspection Ratio Figure 10(a) gives the average I/O cost for updates in the RUM-tree when the inspection ratio increases from 0% to 100%. With the increase in the inspection ratio, both the RUM-tree$_{token}$ and the RUM-tree$_{touch}$ receive larger I/O costs due to more frequent cleaning. The costs of the RUM-tree$_{token}$ and of the RUM-tree$_{touch}$ are very similar. This is because the clean-upon-touch optimization of the RUM-tree$_{touch}$ does not involve additional cleaning cost besides the cost of cleaning tokens. Figure 10(b) gives the garbage ratios of the RUM-trees under the same parameters. The garbage ratio of either the RUM-tree$_{token}$ or the RUM-tree$_{touch}$ decreases rapidly when the inspection ratio increases to 20%. Observe that the inspection ratio of 20% achieves rather good update performance and a near-optimal garbage ratio for both the RUM-tree$_{token}$ and the RUM-tree$_{touch}$. If not otherwise stated, the RUM-tree$_{touch}$ with a 20% inspection ratio is studied for comparisons in the rest of the experiments.

PARAMETERS	VALUES USED
Number of objects	**2M**, 2M~20M
Moving distance between updates	**0.01**, 0~0.01
Extent of objects	**0**, 0~0.01
Node size (bytes)	1024, 2048, 4096, **8192**
Inspection Ratio of RUM-tree	**20%**, 0%~100%

Table 1. Experiment Parameters and Values



(a) Update I/O (b) Garbage Ratio

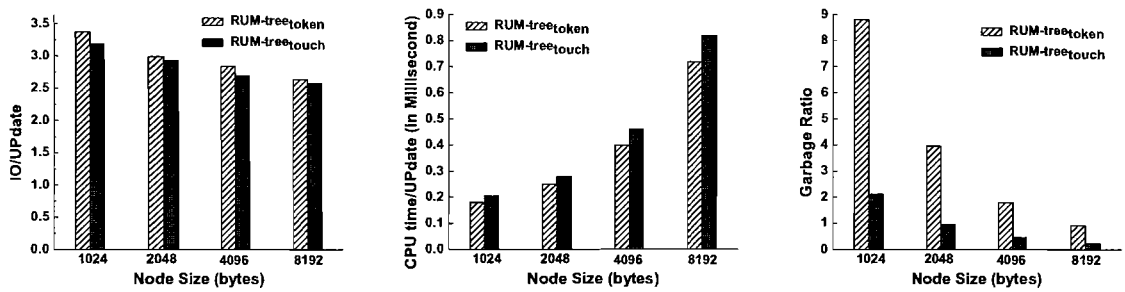
Figure 10. Effect of Inspection Ratio

5.1.2. Effect of Node Size In these experiments, we study the effect of various node sizes on the RUM-tree. Figure 11(a), 11(b), and 11(c) give the average I/O cost, the average CPU cost, and the garbage ratio of the RUM-trees under different node sizes, respectively. When a node has larger size, the average update I/O cost decreases slightly. This is due to fewer node splitting in a larger node. The average update CPU cost increases in a larger node because the garbage cleaner checks more entries in one node cleaning. For the same reason, the garbage ratio decreases quickly with the increase in the node size. Observe that the I/O cost dominates the CPU time as one I/O normally takes around 10 milliseconds. Therefore, the RUM-tree prefers a large node size over a small node size. In the rest of the experiments, we fix the node size at 8192 bytes.

5.2. Performance with Various Moving Distances

In this section, we study the performance of the R*-tree, the FUR-tree, and the RUM-tree when the changes between consecutive updates (referred to as *moving distance*) vary from 0 to 0.01.

5.2.1. Update Cost Figure 12(a) gives the update I/O costs for the three R-tree variants. The R*-tree exhibits the highest cost in all cases due to the costly top-down search. The update cost of the FUR-tree increases rapidly with the increase in objects' moving distance. In this case, more objects move far from their original nodes and require top-down insertions. The update cost of the RUM-tree is steady being only 22% of the cost of the R*-tree, and only 40% to 70% of the cost of the FUR-tree.



(a) Update I/O (b) Update CPU (c) Garbage Ratio

Figure 11. Effect of Node Size

5.2.2. Search Cost The search performance of the three indexing types along various moving distances is given in Figure 12(b). The R*-tree exhibits the best search performance, as its structure is adjusted continuously by the top-down updates. For the FUR-tree, the search cost exhibits a peak when the moving distance reaches 0.002. At that point, most of leaf nodes are able to enclose the object updates in the original nodes by expanding the node MBRs. Thus, the FUR-tree is not adjusted globally for optimal search performance. After that point, more updates are inserted in a top-down manner, thus the FUR-tree structure is more compact. The RUM-tree exhibits around 10% higher search cost than the R*-tree. This is mainly due to a smaller fanout of the RUM-tree leaf nodes to include more information in the leaf entries.

5.2.3. Overall Cost Figure 12(c) gives a comprehensive view of the performance comparison. In this experiment, we vary the ratio of the number of updates over the number of queries from 1:100 to 10000:1. When the ratio increases, the RUM-tree gains more performance achievement. At the point 10000:1, the average cost of the RUM-tree is only 43% of the FUR-tree and 23% of the R*-tree. This experiment demonstrates that the RUM-tree is more applicable than the R*-tree and the FUR-tree in dynamic environments.

5.2.4. Size of Auxiliary Structure Figure 12(d) compares the sizes of the auxiliary structures employed by the FUR-tree and the RUM-tree. For the FUR-tree, each object owns a corresponding entry in the secondary index, which results in a huge indexing structure. For the RUM-tree, UM is upper-bounded and can be kept small in size. For better visualization, we only show the size of UM in the rest of experiments.

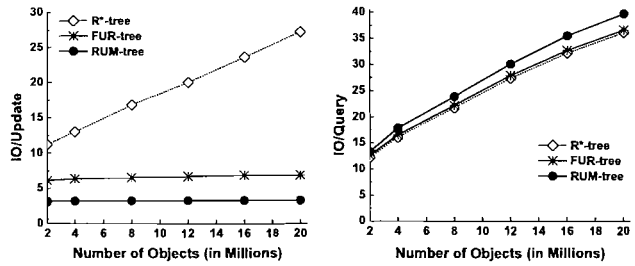
5.3. Performance with Object Extent

In previous experiments, the object set consists of point objects. In this section, we study the performance of the R-tree variants with different object sizes. In these experiments, the indexed objects are squares and their side length (referred to *object extent*) varies from 0 to 0.01.

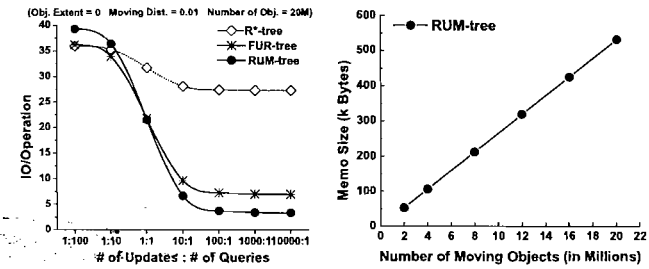
5.3.1. Update Cost Figure 13(a) gives the average update I/O cost of the three R-tree variants. The update cost of the R*-tree grows with the object extent. As a larger extent results in larger node MBRs, the R*-tree needs to search more nodes to locate the object to be updated. For the FUR-tree, the update cost decreases along with the increase in the object extent. This is because the update is more likely to be able to remain in the same leaf node when the MBRs of the nodes become larger. The update cost of the RUM-tree is around 14% to 25% that of the R*-tree, and is around 43% to 68% that of the FUR-tree. The RUM-tree exhibits stable update performance as the memo-based update approach is not affected by object extents.

5.3.2. Search Cost The search performance of the R-trees with various object extents is given in Figure 13(b). The R*-tree achieves the best performance followed by that of the FUR-tree. The search cost of the RUM-tree is around 12% higher than that of the R*-tree.

5.3.3. Overall Cost Figure 13(c) gives a comprehensive view of the performance comparison when the object extent is set as 0.01. Again, we study the performance under various ratios of updates over queries. Comparing with Figure 12(c), the performance of the FUR-tree and the R*-tree are both affected by the extents of the indexed objects. To the contrary, the performance of the RUM-tree is not affected by object extents. The RUM-tree outperforms both



(a) Update I/O (b) Search I/O



(c) Overall I/O (d) Size of Memo

Figure 14. Performance with Num. of Obj.

ure 14(c) when the number of objects is fixed at 20 million. The ratio of the number of updates to the number of queries varies from 1:100 to 10000:1. The RUM-tree outperforms the other two R-tree variants when the ratio is larger than 1:1. When the ratio reaches 10000:1, the average cost of the RUM-tree is only 50% of that of the FUR-tree, and is only 13% of that of the R*-tree.

5.4.4. Size of UM In Figure 14(d), we study the size of UM of the RUM-tree when the number of objects scales up to 20 million objects. The size of UM increases linearly with the number of indexed objects. This is because the garbage ratio of the RUM-tree is not affected by the number of objects. This property guarantees that the size of UM is scalable in terms of the size of the RUM-tree.

5.5. Log and Recovery

In this section, we study the logging costs and the recovery costs for the different options in Section 3.4. For Option II and III, one checkpoint is logged every 10,000 updates/inserts.

5.5.1. Update Cost under Logging Figure 15 gives the overall I/O cost per update when the RUM-tree works with

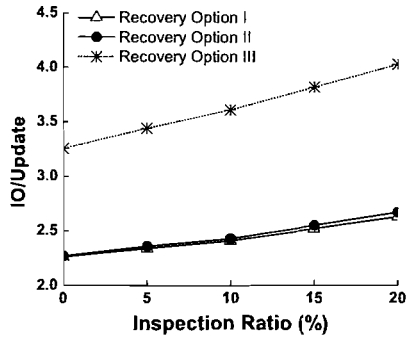


Figure 15. Update I/O with Log Options

different logging options. Option I has the lowest update cost as no log is maintained. The cost of Option II is only slightly higher than that of Option I where Option II occasionally writes UM to the log. Option III has the highest cost that is around 50% higher than the other two options, as it logs every memo change.

5.5.2. Recovery Cost Table 2 gives the number of disk accesses when recovering UM in the case of system failure. Option I incurs the largest cost. This is because the intermediate UM is too large to fit in memory, hence results in an excessive number of disk accesses. The recovery cost of Option II is significantly lower than Option I. Option II retrieves UM at the last checkpoint, and scans every disk node once. Option I achieves the best performance by only retrieving logged data. Considering the tradeoff between the logging cost and the recovery cost, we use Option II as the choice in our previous experiments.

5.6. Throughput under Concurrent Accesses

Figure 16 gives the throughput of the RUM-tree and the R*-tree. The throughput of the FUR-tree is not compared as there is insufficient knowledge about concurrency control in the FUR-tree. In these experiments, 100 threads update and query the R-tree variants concurrently. We vary the percentage of updates from 0% (i.e., queries only) to 100% (i.e., updates only). Our experiments indicate that the RUM-tree is more suitable for concurrent accessing than the R*-tree. The RUM-tree and the R*-tree have similar throughput when all transactions are queries. With the increase in the ratio of updates, the R*-tree suffers lower throughput

Option I	Option II	Option III
2008,000	7,218	11

Table 2. The Number of I/Os for Recovery

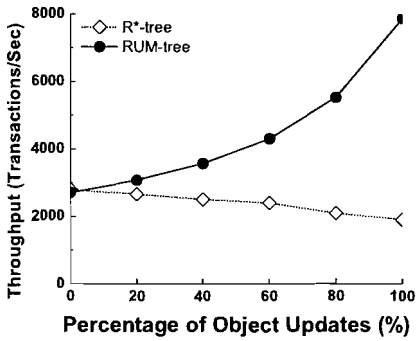


Figure 16. Throughput

while the RUM-tree exhibits higher throughput. The reason is that an update requires fewer locks than a query in the RUM-tree, while it is not the case for the R*-tree.

6. Conclusion

For R-tree updates, given the object id and the object's new value, the most costly part lies in searching the location in the R-tree of the objects to be updated. In contrast to the former update approaches, we presented a memo-based approach to avoid the deletion I/O costs. In the proposed RUM-tree, object updates are ordered temporally according to the processing time. By maintaining the update memo, more than one entry of an object may coexist in the RUM-tree. The obsolete entries are deleted lazily and in batch mode. Garbage cleaning is employed to limit the garbage ratio in the RUM-tree and confine the size of UM. The RUM-tree along with the garbage cleaner outperforms significantly other R-tree variants in the update performance, while yielding similar search performance. We believe that the memo-based update approach has potential to support frequent updates in many other indexing structures, for instances, B-trees, quadtrees and Grid Files.

References

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

[2] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2), 2002.

[3] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, 2003.

[4] K. Chakrabarti and S. Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. In *ICDE*, 1998.

[5] R. Cheng, Y. Xia, S. Prabhakar, and R. Shah. Change Tolerant Indexing for Constantly Evolving Data. In *ICDE*, 2005.

[6] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[7] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient Indexing of Spatiotemporal Objects. In *EDBT*, pages 251–268, Prague, Czech Republic, Mar. 2002.

[8] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB*, pages 500–509, 1994.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *PODS*, 1999.

[10] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.

[11] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[12] M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *Proc. of the ACM Symp. on Applied Computing, SAC*, pages 235–240, Feb. 1998.

[13] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*, pages 395–406, Sept. 2000.

[14] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *SSTD*, pages 59–78, Redondo Beach, CA, July 2001.

[15] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[16] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. In *Proc. of the Workshop on Algorithm Engineering and Experimentation, ALENEX*, pages 178–193, Jan. 2002.

[17] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. pages 17–31, 1985.

[18] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.

[19] S. Saltenis and C. S. Jensen. Indexing of now-relative spatio-bitemporal data. *The VLDB Journal*, 11(1):1–16, 2002.

[20] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[21] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[22] Y. Tao and D. Papadias. Efficient Historical R-trees. In *SSDBM*, pages 223–232, July 2001.

[23] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*, 2001.

[24] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.

[25] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of the IEEE Conference on Multimedia Computing and Systems, ICMCS*, June 1996.