Recently, new approaches for updating R-trees in a bottom-up manner have been proposed [12,13]. 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 to the leaf node of a data item, a secondary index such as a direct link [12] or a hash table [13] is maintained on the identifiers of all objects. Figure 1b illustrates the bottom-up update process. The bottom-up approach exhibits better update performance than the top-down approach when the change of 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 changes between consecutive updates become large. Moreover, a secondary index may not fit in memory due to its large size, which may add 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 (R-tree with update memo), an R-tree variant that handles object updates efficiently. In the RUM-tree, a *memo-based* update approach is utilized to reduce the update cost. 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 operation. Specially designed *garbage cleaners* are employed to remove old data entries lazily. Therefore, the cost of an update operation is reduced approximately to the cost of an insert operation and the total cost of 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 in scenarios with frequent updates: (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 various factors, i.e., the changes between consecutive updates, the extents of moving objects, the number of moving objects, and the distribution of the moving objects in the space.

The contributions of the paper can be summarized as follows:

- We propose an R-tree variant, named the RUM-tree, that reduces the update cost while yielding similar search performance to other R-tree variants in scenarios with frequent updates;
- We address the issues of crash recovery and concurrency control for the proposed RUM-tree;

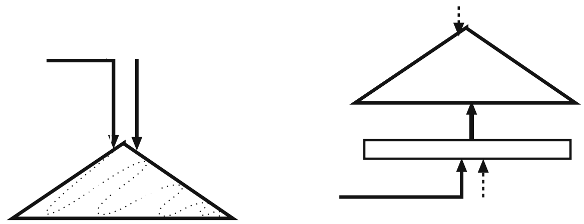


Fig. 1 Existing R-tree update approaches

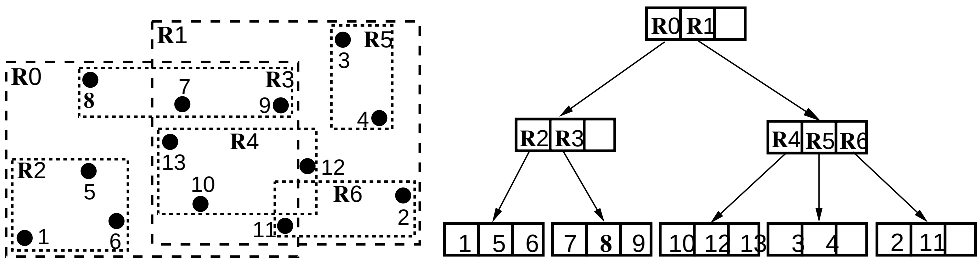
- We analyze the update costs for the RUM-tree and the other R-tree variants, and derive an upper-bound on the size of the update memo;
- We present a comprehensive set of experiments indicating that the RUM-tree outperforms other R-tree variants by up to one order of magnitude.

The remainder of the paper is organized as follows. Section 2 gives an overview of the R-tree and summarizes related work in the literature. Section 3 presents the details of the RUM-tree, including the issues of crash recovery and concurrency control. Section 4 gives a cost analysis of the memo-based update approach and compares it with the top-down and the bottom-up update approaches. This section presents also a derivation of an upper-bound for the size of the update memo. Experimental results are presented in Sect. 5. Finally, Sect. 6 concludes the paper.

2 R-tree-based indexing and related work

The R-tree [1] 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. An entry in a leaf node is of the form (MBR_o, oid), where MBR_o is the MBR of the indexed spatial object, and oid is a unique identifier of the corresponding 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.[1] 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 three that indexes thirteen objects.

The R-tree [1] and its variants [2,9,20,24] were designed mainly for static data. Update processing is cumbersome because the update is treated as a delete followed by an insert. The claim is that updates are not frequent in traditional



(a) A Set of Planar Points (b) Corresponding R-tree Index

Fig. 2 An example of R-tree

applications. However, in spatio-temporal databases, objects continuously change and update the underlying indexing structures.

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 the following aspects: (1) Indexing the historical trajectories of objects, e.g., [4,7,15,16,25,26,28]; (2) Indexing the current locations of objects, e.g., [6,11,17,18,21,22]; or (3) Indexing the predicted trajectories of objects, e.g., [19,23,27]. For more detail about R-tree variants, interested readers are referred to [14] for a comprehensive survey. Most of these works assume that the updates are processed in a top-down manner. Although the memo-based update technique presented in this paper can be applied to improve the update performance of most of these works, this paper focuses on trees that index current locations of objects.

To support frequent updates in R-trees, [12] and [13] propose a bottom-up update approach. The Lazy-update R-tree (LUR-tree) [12] modifies the original R-tree structure to support frequent updates. The main idea is that if an update to a certain object p would result in a deletion followed by an insertion in a new R-tree node, it would be better to increase slightly the size of the minimum boundary rectangle of the R-tree node that contains p to accommodate the new location of the object. The Frequently Updated R-tree (FUR-tree) [13] extends the LUR-tree by performing a bottom-up approach in which a certain moving object can move to one of its siblings. The bottom-up approach works well when the consecutive changes to objects are small. However, in the case that consecutive changes are large, the performance of the bottom-up approach degrades quickly. Besides, both the LUR-tree and the FUR-tree rely on an auxiliary index to locate the old object entries. In Sect. 5, we show that the proposed memo-based update approach of the RUM-tree outperforms the bottom-up approach significantly and is more stable under various conditions.

Although some approaches have been proposed to handle both data and indexes stored completely in main memory (e.g., see [8,10]), in this paper we focus on index strategies that handle data stored on disk. As in practical implementations of the R-tree and its variants, the non-leaf nodes of the RUM-tree are stored in memory to take advantage of the faster access speeds of main memory devices. The data and RUM-tree leaf nodes are stored on disk allowing the support of very large datasets and the integration of recovery mechanisms to protect the integrity of the data.

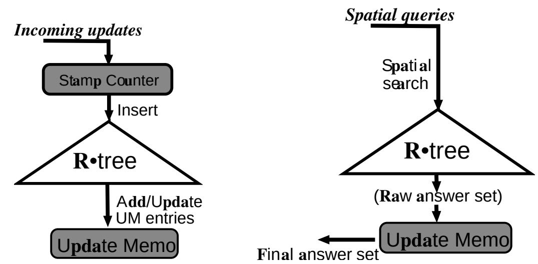
An earlier version of this paper appeared in [29]. This paper extends on [29] by: (1) an improved structure for the RUM tree that uses pointers to parents only in nodes that are stored in main memory and consequently reduces the number of required I/O accesses when the tree structure needs to be updated due to node splitting; (2) an extended experimental section that includes: (a) an evaluation of the RUM-tree under datasets with various object distributions in space, (b) a study of query performance for various query sizes, (c) a comparison of the storage requirements (in memory and disk) of the different trees, (d) an analysis of the dynamic structural properties of the studied trees such as fanout and number of nodes, and (e) a study of the CPU cost of querying the different trees for varying values of garbage collection ratios; (3) a new garbage cleaning mechanism, termed the *least-recently cleaned mechanism*, for the RUM-tree to control the memory size of the update memos (Sect. 3.3); (4) a crash recovery mechanism for the RUM-tree (Sect. 3.4); (5) a concurrency control mechanism for the RUM-tree (Sect. 3.5); (6) experimental evaluation of the algorithms in items (3)–(5) (Sects. 5.1, 5.8, 5.9).

3 The RUM-tree index

In the existing update approaches, the deletion of old entries generates extra overhead during the update process. 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, an R-tree variant that reduces additional disk accesses for such deletions and thus reduces the update cost.

The primary feature behind the RUM-tree is that the old entry of the data item is not required to be removed when it gets updated. 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 as the *latest* entry), and all other entries of the object are old entries (referred to as *obsolete* entries). The RUM-tree maintains an *update memo* structure to detect if an entry is obsolete or not. The obsolete entries are identified and removed from the RUM-tree by a *garbage cleaner* mechanism.

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



(a) Insert/Update **(b)** Search

Fig. 3 Operations in the RUM-tree

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 $newTuple$; $ne.stamp \leftarrow stamp$;
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 $umne.N_{old}$;

Fig. 4 Insert/update in the RUM-tree

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 every time it is used. 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, oid, stamp), where stamp is the assigned stamp number, and MBR and oid 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. For example, a UM entry (O_{99}, 900, 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 900. 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, UM is hashed on the oid attribute. With the garbage cleaner provided in Sect. 3.3, the size of UM is kept rather small and can practically fit in main memory of nowadays machines. UM only stores entries for the updated objects instead of entries for all the objects. As will be shown in Sect. 4, the size of the UM constructed this way can be upper-bounded. The size of UM is further studied through experiments in Sect. 5.

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 Fig. 3a. Pseudo-code for the insert/update algorithm is given in Fig. 4. First, 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 [2]. 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

Algorithm MemoBasedDelete(oid)

1. $stamp \leftarrow StampCounter$; Increment $StampCounter$;
2. 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 $umne.N_{old}$;

Fig. 5 Delete in the RUM-tree

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;

Fig. 6 Checking entry status in the RUM-tree

deleted is treated as an update to a special location. 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 3b 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 query predicates is a superset of the actual answer. 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. The type of queries considered here is range queries. Other query processing algorithms, e.g., k-NN queries, will require the integration of this filtering step with the specific query algorithms. The RUM-tree employs the algorithm given in Fig. 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; otherwise it is the latest entry. In the case that no corresponding UM entry exists, the leaf entry is the latest entry.

Discussion. Sanity checking can be performed at a higher level before invoking the index. Since the RUM-tree does not check the existence of an old entry when performing insert, update or delete operations, it would be possible to delete/update an object that never existed. An instance of this case could happen when an update is reported for an object O1 that does not exist in the tree. An insert will be performed without checking if a previous instance of O1 exists and a UM entry will be created for this object. This UM entry will have the value 1 assigned to its N_{old} field although the exact number of obsolete entries is 0. We call this UM entry a *phantom* entry given that it will never be removed by the cleaning mechanisms presented in Sects. 3.3.1, 3.3.2, and 3.3.3. In Sect. 3.3.5 we present a mechanism to detect and remove *phantom* entries. Based on the presented 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. Sanity checking can be implemented to detect that O1 is a new object and insert it using the regular R-tree insert procedure without modifying the UM structure. However, given that phantom entries can be detected and eliminated easily, the implementation of sanity checking is not necessary. In the previous example, we assumed that the expected behavior is to consider an update operation on an object that does not exist in the database simply as an insertion of a new object. If this is not the case and the update should be detected as error, we could easily implement this test in a higher layer and avoid further processing.

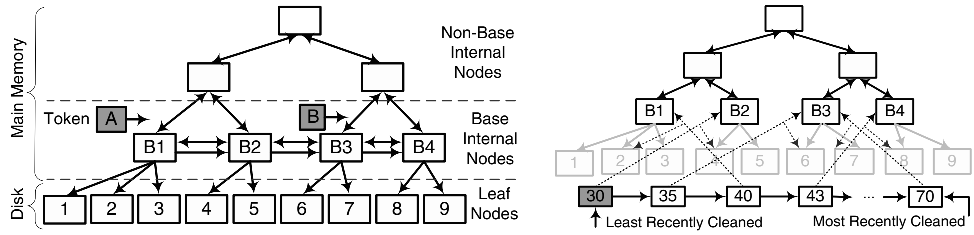
3.3 Garbage cleaning

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

Figure 7a gives an example of the RUM tree structure. The non-leaf nodes are divided in two groups: base internal nodes (nodes in the lowest level of the tree formed by the non-leaf nodes) and non-base internal nodes. In practical implementations, it is expected that the non-leaf nodes of the RUM tree are stored in memory while the leaf nodes remain on disk.

A *cleaning token* is a logical object that is used to traverse all leaf nodes of the RUM-tree horizontally. When a specific leaf node N is cleaned it may be necessary to update the MBR of N and its ancestors in a bottom-up manner. (see the algorithm in Fig. 8). Adding pointers to parents in the tree nodes is a good approach to allow quick access to parents. The



(a) Mechanism I: Cleaning Tokens **(b)** Mechanism II: LRC List

Fig. 7 Garbage cleaners inside the RUM-tree

Algorithm Clean($leafnodeN$)

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 $MIN_{ENTRIES}$, re-insert the remaining entries of N into the RUM-tree; Otherwise, adjust the MBRs of N and its ancestors in a bottom-up manner;

Fig. 8 Cleaning a leaf node

only exception is at the leaf level. In this case, if a split occurs in the level immediate above it, it would require updating the parent pointer of all the leaf nodes that have a new parent. Each of these updates would cost one I/O because leaf nodes are stored on disk. Consequently, only non-leaf nodes are extended with pointers to parents.

The cleaning tokens cannot traverse the leaf nodes directly because if a cleaning token processes a leaf node that requires changing the node's ancestors, there is no pointer from that leaf node to its parent. The actual tokens traverse the base internal nodes. Each token has the following structure (BINPtr, curEntryIndex). BINPtr is a pointer to a base internal node BIN and curEntryIndex is the index of an entry in BIN. Each time the cleaning process is called, it updates BINPtr and curEntryIndex so that BINPtr.entries[curEntryIndex].p_c points to the next leaf node to be processed and then performs the cleaning tasks on this leaf node. To locate the next base internal node quickly, the base internal nodes of the RUMtree are doubly-linked in cycle. The cleaning process is called every time the RUMtree receives new I updates. I is known as the *inspection interval*. When a leaf node is cleaned, all its entries are inspected and the obsolete entries deleted.

Figure 8 gives the pseudo code of the cleaning procedure of a leaf node. Every entry in the inspected leaf node is checked by CheckStatus() given in Fig. 6, and is deleted from the node if the entry is identi ed 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. Occasionally, the leaf node may under ow 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 under ow, the MBR of the updated leaf node and the MBRs of its ancestor nodes are adjusted. The parent P of a leaf node being processed is always pointed at via the token's BINPtr. The ancestors of P can be obtained using the pointers to parents.

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 7a illustrates a RUM-tree with two cleaning tokens. Token A inspects nodes B1 to B2 while Token B inspects nodes B3 to B4. Tokens move either with the same inspection interval or with different inspection intervals. If all the subsets of nodes share the same value for the inspection interval and they are approximately of the same size, all the nodes of the tree are cleaned with approximately the same frequency. If it is known in advance that certain subsets will receive most of the updates, these segments should receive smaller inspection intervals to be cleaned more frequently. In the following sections, we propose two extensions to the Cleaning Tokens approach that prioritize the cleaning process of nodes that receive more updates. Notice that each cleaning token generates additional disk accesses during the cleaning procedure. Hence, there is a tradeoff between the cleaning effect and the overall cost.

We de ne 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 re ects 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 de ned 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. Assuming that a RUM-tree has m cleaning tokens t_1 to t_m, and that the inspection interval of t_k is I_k for $1 \le k \le m$, then ir of the cleaner is calculated as:

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

The cleaning token approach has the following straightforward but 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 given that 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 Clean upon touch

Besides the cleaning tokens, garbage cleaning can be performed whenever a leaf node is accessed during an insert/update operation. The cleaning procedure is the same as in Fig. 8. As a side effect of insert/update, such clean-upon-touch process does not incur extra disk accesses other than the ones required to reorganize the tree when the cleaned node underflows. When working with the cleaning tokens, the clean-upon-touch reduces the garbage ratio and the size of UM dramatically.

3.3.3 Least recently cleaned list

Note that the Cleaning Tokens mechanism cleans the RUM-tree nodes in a round-robin order, and the clean-upon-touch mechanism cleans the RUM-tree nodes in a random order. When both mechanisms are concurrently running, a leaf node that just got cleaned by one mechanism may be inspected by the other mechanism in a short period of time, which compromises the cleaning effect. This situation happens because the cleaners have no knowledge about the cleaning history of the nodes. To maximize the effect of cleaning, the garbage cleaners should inspect only the nodes that have not been cleaned recently. These nodes have the potential of containing a large number of obsolete entries. Based on the above observation, we enhance the RUM-tree by maintaining the least recently cleaned (LRC) list according to the cleaning history of the RUM-tree nodes.

Basically, the (LRC) list is a linked list in memory. Logically each element of the list points to a leaf node of the tree. In the actual implementation of the list, each of its elements points to a leaf element n through a pointer to its parent node and the index in this node that points to n. Each list element contains (pNode, leafIndex, lastClean). pNode is the pointer to the parent node, leafIndex is the index in the parent node that points to the leaf node, and lastClean stores the historical value of the stamp counter when the node n was cleaned for the last time. Figure 7b illustrates a RUM-tree with the LRC list. Here, the value of lastClean is stored within each LRC element.

When a leaf node is cleaned either by the Cleaning Tokens mechanism or by the clean-upon-touch mechanism, the corresponding LRC element is updated and is moved to the end of the LRC list. In addition, node split results in inserting a new element to the end of the list, and node deletion results in removing the corresponding element from the LRC list. Consequently, the elements in the LRC list are ordered based on the cleaning history of their corresponding RUM-tree leaf nodes. To maximize the cleaning effect, the Cleaning Tokens mechanism only inspects the leaf node pointed at by the head element of the LRC list. To avoid repeated cleaning of the same RUM-tree nodes in a short period of time, an inspection threshold T is specified. Then, when a RUM-tree leaf node is touched by an insert or update operation, the node is cleaned up only if the RUM-tree has received T updates since the last time that the node was cleaned. Note that the number of updates to the RUM-tree since the last time a node was cleaned can be calculated from the difference between the current value of the stamp counter and the value of lastClean of the corresponding node. In Sect. 5, we demonstrate that utilizing the LRC list yields a smaller garbage ratio for the RUM-tree.

3.3.4 Properties of the proposed cleaning strategies

Cleaning tokens are used in all the cleaning strategies presented in previous sections. The fundamental property of the Cleaning Tokens mechanism is that, as shown in Sect. 4, it ensures an upper-bound on the size of the UM structure. It also ensures that there are no obsolete entries that never get removed. The clean-upon-touch approach used with the cleaning tokens ensures additionally that the nodes that receive more updates are also more frequently cleaned. This approach will be effective whether the updates are uniformly distributed on all the nodes or are concentrated on a small subset of nodes. Finally, the LRC list approach used in conjunction with the previous two strategies changes the order in which nodes are cleaned by the tokens such that nodes that were not cleaned recently are cleaned first. When the distribution of updates on the leaf nodes is close to uniform or is not very skewed this strategy cleans first the nodes that have more obsolete entries. In cases where the distribution of updates is very skewed, the order itself may not be better than a round-robin order but the use of the clean-upon-touch mechanism still ensures that the nodes that receive most of the updates are cleaned more frequently. The experimental section reveals that the clean-upon-touch and the LRC list approaches yield very small garbage ratios while bearing a small overhead.

3.3.5 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 reaches the value 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] Assume that object O1 is inserted at time T1. The entry (O1, T1, 1) will be added to the UM structure. The value 1 means that there is at most one obsolete entry. Assuming that O1 is not updated, when the cleaning process calls CheckStatus (see Fig. 6) for this entry, O1 will always receive the value LATEST. Consequently, the entry in UM will never be deleted. We should observe that when there are not obsolete instances of an object, we do not need to have an entry in UM for this object.

The RUM-tree employs a *Phantom Inspection* procedure to detect and remove phantom entries. According to Property 1 in Sect. 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 inspection cycle. 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. The insert operation always generates a new UM entry.

3.4 Crash recovery

In this section, we address the recovery issue of the RUM-tree in the case of system failure. Given that UM is stored 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. If the non-leaf nodes of the tree were also stored in memory, then the tree can be rebuilt inserting all the leaf level entries in a new tree. This insertion should be performed using a standard R-tree insertion or bulk loading processes and assigning also the *stamp* attribute value on the leaf level entries from the data stored on disk. We consider three approaches to recover UM, each 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, 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 Sect. 3.3.5. 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) [5] 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 $\frac{N}{ir \times M}$. As each obsolete entry may own an independent UM entry, the upper-bound for the size of UM is $\frac{N \times E}{ir}$. In real spatiotemporal applications the number of objects N can change over time. In this case we should use an estimate of the maximum value of the number of objects as N. This value will generate a safe estimate of the upper bound for the size of UM.

It is straightforward to prove that the average garbage ratio is $\frac{N}{2ir \times M}$, and that the average size of UM is $\frac{N \times E}{2ir}$. 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 Sect. 5.

4.2 Update cost

We analyze the update costs for the top-down, the bottom-up, and the memo-based update approaches. We investigate the number of disk accesses.

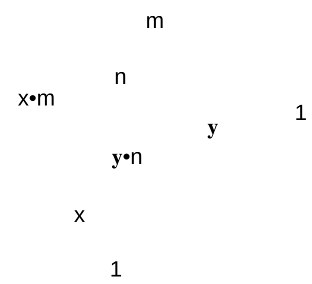


Fig. 9 Probability of containment

Practically, the internal R-tree nodes are cached in the memory buffer while the leaf nodes remain in disk. Otherwise, a Direct Access Table as used in [13] can be utilized to avoid excessive accesses to internal R-tree nodes. 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 [13], 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 $x \times y$, and let W_{mn} be a window of size $m \times n$, where $0 < x, y, m, n < 1$. When W_{xy} and W_{mn} are randomly placed, the probability that W_{xy} contains W_{mn} is given by:

$$\max(x - m, 0) \times \max(y - n, 0)$$

Proof Assume that the position of W_{xy} is fixed as shown in Fig. 9. Then, W_{mn} is contained in W_{xy} if and only if the bottom-left vertex of W_{mn} lies in the shaded area. The size of the shaded area is given by $\max(x - m, 0) \times \max(y - n, 0)$. Since W_{mn} is randomly placed, the probability of W_{xy} containing W_{mn} is also $\max(x - m, 0) \times \max(y - n, 0)$. 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 $a \times b$, where $0 < a, b < 1$. From Lemma 2, the expected number of leaf node accesses for searching the old entry is given by:

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

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:

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

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 $U \times ir$. 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 Sect. 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 Sect. 4.1, the additional logging cost per update in Option II is $\frac{N \times E}{ir \times P \times C}$. For Option III, the additional logging cost per update is $\left(\frac{N \times E}{ir \times P \times C} + 1\right)$.

5 Experimental evaluation

In this section, we study the performance of the RUM-tree through experiments and compare this performance with that of the R*-tree [2] and the Frequently Updated R-tree (FUR-tree) [13].

All the experiments are performed on an Intel Pentium IV machine with CPU Dual Core 1.83GHz and 2GB RAM. In the experiments, the number of moving objects ranges between 1 and 10 million objects.

Three datasets are used in the experiments: ROADS-SKW, ROADS-UNI, and UNIFORM. In ROADS-SKW and ROADS-UNI, the moving objects are restricted to move on the roads of a city while in UNIFORM objects are uniformly distributed in the space and assigned a random direction. All datasets are scaled to a unit square. ROADS-SKW and ROADS-UNI are generated using the Network-based Generator of Moving Objects [3] with the road map of Oldenburg, Germany. The objects in ROADS-UNI are always distributed in a close to uniform fashion over the roads of the city. The objects in ROADS-SKW, the dataset used by default, aim to simulate the moving objects in a real life scenario where the number of moving objects in the downtown of the city is higher than the one in the outskirts of the city. The extent of the objects ranges between 0 (i.e., points) and 0.01 (i.e., squares with side 0.01).

The moving distance ranges from between 0.001 and 0.1. 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 ranging from 0.02 to 0.1. The primary parameters used in the experiments are outlined in Table 1, where the default values are given in bold fonts.

Both disk accesses (I/O) and CPU time are investigated in the experiments. However, in most cases we only report the I/O cost since it is the dominant cost. As discussed previously, 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. The value 0.003 is used since it was found that it yields the best performance results under similar evaluation conditions [13]. Except the experiments in Sect. 5.8, Option II discussed in Sect. 3.4 for the RUM-tree is chosen as the default recovery option.

Table 1 Experiment parameters and values

Parameters	Values used
Number of objects	1M, 1 10M
Moving distance between updates	0.04, 0.001 0.1
Extent of objects	0, 0 0.01
Query square side lenght	0.06, 0.02 0.1
Node size (KB)	1, 2, 4, 8
Inspection ratio of the RUM-tree	10%, 1 100%

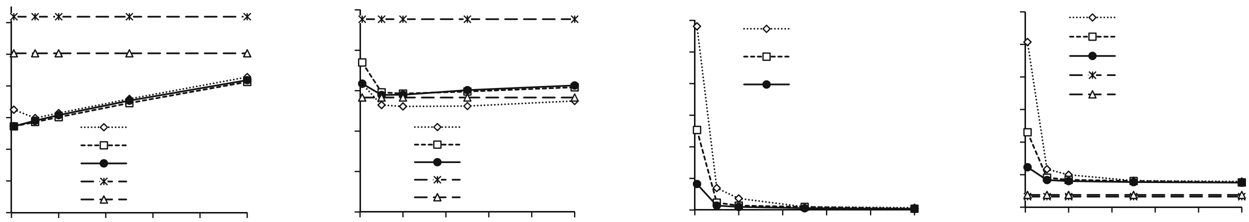


Fig. 10 Effect of the inspection ratio

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. We implement three types of RUM-trees, each RUM-tree employs one kind of garbage cleaning mechanism as discussed in Sect. 3.3, namely, the cleaning-token mechanism (denoted by the RUM-tree Token in this section), the clean-upon-touch mechanism combined with cleaning tokens (denoted by the RUM-tree Touch in this section), and the LRC mechanism (denoted by the RUM-tree LRC in this section).

5.1.1 Effect of inspection ratio

Figure 10a gives the average update I/O costs for the RUM-trees when the inspection ratio increases from 1 to 100%. With the increase in the inspection ratio, the RUM-tree Token, the RUM-tree Touch and the RUM-tree LRC all have larger update I/O costs due to the more frequent cleaning. The update costs of the three RUM-trees 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. Meanwhile, the LRC mechanism is mainly designed to maximize the cleaning effect rather than reduce the number of disk accesses. The update I/O costs for the FUR and R* trees, included only as reference in this figure, are constant since they do not depend on the inspection ratio. The update I/O cost of the RUM-tree approaches is only 44–68% of the I/O cost of the R*-tree and only 54–83% of the I/O cost of the FUR-tree. In general, lower values of inspection ratio increase the advantage of the RUM-tree approaches over the RUM and R* trees.

Figure 10b presents the update CPU time for the RUM, FUR and R* trees. The average CPU time required to process an update operation in the RUM and FUR trees is similar and approximately 60% of the time required in the R*-tree approach. The approach with the lowest update CPU time for most values of inspection ratio is the RUM-tree Token approach. The other two RUM-tree approaches consume slightly higher CPU time because they make use of the clean-upon-touch mechanism besides using cleaning tokens.

Figure 10c presents the garbage ratios of the RUM-trees under various values of inspection ratio. The garbage ratios of all the RUM-trees decrease along with the increase in the inspection ratio. Specifically, the garbage ratios decrease rapidly when the inspection ratio increases from 1 to 20%. Observe that the inspection ratio of 10% achieves quite good update performance (around 2.9 I/Os per update) and a near-optimal garbage ratio for all the RUM-trees (smaller than 3.5%). The RUM-tree Touch has smaller garbage ratio than that of the RUM-tree Token because the former approach maintains clean the nodes that are more frequently updated. The RUM-tree LRC has a smaller garbage ratio than that of the RUM-tree touch because, in addition to using the clean-upon-touch mechanism, its cleaning tokens clean first the nodes that have not been cleaned recently. In many cases, these nodes are the ones that have more obsolete entries.

Greater values of the inspection ratio produce less obsolete entries and consequently reduce the number of nodes that need to be read from disk and the CPU time spent filtering obsolete entries to answer a query. This relationship between the CPU time per query and the inspection ratio of the RUM-trees is presented in Fig. 10d. This figure also presents for reference the query CPU time for the FUR and R* trees. The CPU costs to process a query in the RUM-trees is bigger than the ones in the FUR and R* trees because of the extra time required to filter the obsolete entries and the reduced fanout of the RUM-trees due to its extra fields and pointers. Similarly to the case of the garbage ratio, the CPU time for processing a query decreases quickly when the inspection ratio increases from 1 to 20%. For values greater than 10% the query CPU cost associated to the RUM trees is just slightly higher than the ones associated to the FUR and R* trees. If not otherwise stated, RUM-trees use an inspection ratio value of 10% in the rest of the experiments.

5.1.2 Effect of node size

In these experiments, we study the effect of various node sizes of the RUM-tree. Figure 11a–d give the average update I/O cost, the average update CPU cost, the garbage ratio, and the average query I/O cost of the RUM-trees under 1, 2, 4

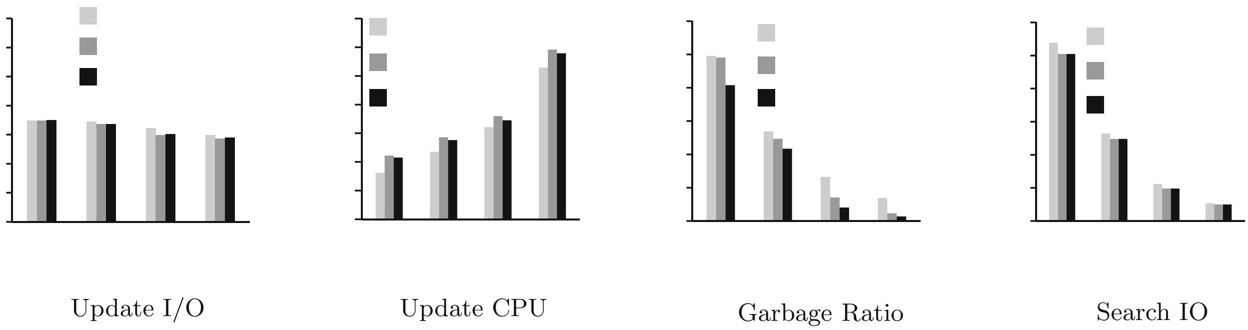


Fig. 11 Effect of node size

and 8 K node sizes, respectively. When the RUM-tree node has larger size, the update I/O cost for any of the three RUM-trees decreases slightly. This is mainly due to fewer node splitting in a larger node. The update CPU cost increases for all the RUM-trees when the RUM-tree node becomes larger. This is because the garbage cleaners need to check more entries every time a node is cleaned. The RUM-tree Touch has a higher CPU cost than those of the other two RUM-trees because it cleans a node whenever the node is accessed. The CPU cost of the RUM-tree LRC is smaller than that of the RUM-tree Touch because the RUM-tree LRC avoids cleaning a node if the node has been cleaned recently. For the garbage ratio, the RUM-tree LRC outperforms both the RUM-tree Token and the RUM-tree Touch, while the RUM-tree Touch outperforms the RUM-tree Tokens. The garbage ratios of all the RUM-trees decrease quickly with the increase in node size. As we observed previously, a direct effect of a smaller value of garbage ratio is a smaller number of entries that satisfy a query and need to be retrieved from disk. The last figure of this section shows how the I/O query cost of all the RUM-trees decreases when the node size increases. Notice that the I/O cost dominates the CPU time, and the experiments demonstrate that the RUM-trees favor a large node size over a small node size. In the rest of the experiments, we fix the node size to 8,192 bytes and use the RUM-tree LRC approach as representative of the RUM-trees.

5.2 Performance while varying moving distance

In this section, we study the performance of the R*-tree, the FUR-tree, and the RUM-tree when the changes in object location between consecutive updates (referred to as *moving distance*) vary from 0.001 to 0.1.

5.2.1 Update cost

Figure 12a 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 with the increase in 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 47% of the cost of the R*-tree, and only 55–65% of the cost of the FUR-tree for most values of moving distance. Notice that the only case in which the FUR-tree slightly outperforms the RUM-tree is when the moving distance is extremely small. In this case, under the FUR-tree approach, most of the updates are performed in the original node of the entry being updated.

5.2.2 Search cost

The search performance of the three indexing types along various moving distances is given in Fig. 12b. The R*-tree exhibits the best search performance as its structure is adjusted continuously by the top-down updates. The RUM-tree exhibits around 30–60% higher search cost than that of the R*-tree and around 25% higher search cost than that of the FUR-tree. The query I/O cost of the FUR-tree is greater than that of the R*-tree because the FUR-tree approach increases the size of the MBRs to accommodate the new object locations and consequently more nodes need to be read and processed for answering a query. On the other hand, the query I/O cost of the RUM-tree is greater than that of the R*-tree because of the presence of obsolete entries that satisfy the query and the reduced fanout of the RUM-tree due to extra attributes in the tree nodes.

5.2.3 Overall cost

Figure 12c gives a comprehensive view of the I/O performance comparison when the moving distance is set to 0.1. Given that our focus is on scenarios with frequent updates, we vary the ratio of the number of updates over the number of queries from 1:1 to 10,000:1. When the ratio increases, the RUM-tree gains more performance achievement. At the point 10,000:1, the average cost of the RUM-tree is only 55% of the FUR-tree and 48% of the R*-tree. This experiment

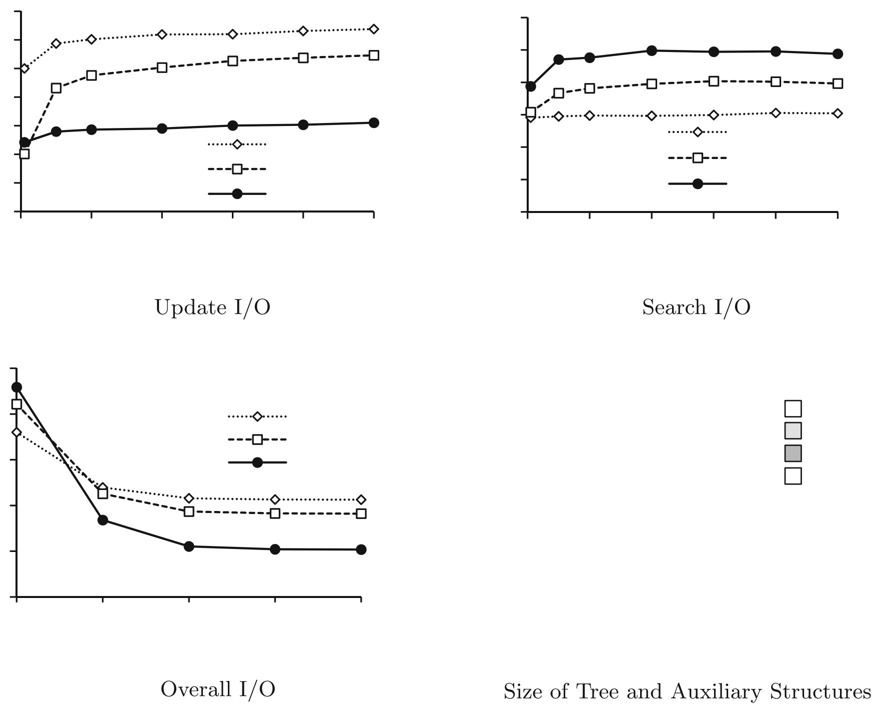


Fig. 12 Performance while varying moving distance

demonstrates that the RUM-tree is more applicable than the R*-tree and the FUR-tree in environments with frequent updates.

5.2.4 Size of tree and auxiliary structures

Figure 12d compares the sizes of the trees and auxiliary structures employed by the FUR, RUM and R* trees. The tree used in the R*-tree approach is smaller than the ones used in the other two approaches while the tree used in the RUM-tree is slightly greater than the one used in the FUR-tree approach. The size of the tree used in the RUM-tree approach is around 50% greater than the one used in the R*-tree approach and around 10% greater than the one used in the FUR-tree. However, the total size (tree size plus auxiliary structures sizes) of the RUM-tree approach is smaller than the total size of the FUR-tree approach. Observe that in the FUR-tree, each object owns a corresponding entry in the secondary index, which results in a huge indexing structure. In the RUM-tree, UM is upper-bounded and can be kept small in size. Also, notice that the size of the tree used in the FUR-tree approach increases signi cantly when the moving distance increases. The tree size of the other two approaches remains more stable when the moving distance changes. The increase in the FUR-tree size when the moving distance increases is due to the poor structure that the tree gets when there are many updates that move the updated entries to sibling nodes and the subsequent node splits. There is not effort made by the FUR-tree approach to nd the best sibling to store the updated entry. This technique is outperformed by the approach used by the R*-tree that is also implemented in the RUM-tree, which uses a combined optimization of the area, margin and overlap of the MBRs during the selection of a node to store the updated object. Generally, the R*-tree strategy generates a tree with less overlap among neighboring nodes, with less splits, and with better storage utilization than those of the FUR-tree. The greater tree size of the RUM-tree is mainly due to the reduced fanout of the tree and the presence of obsolete entries. The rather stable tree size of the RUM-tree for varying moving distance values is due to the use of the same node selection optimization strategy used by the R*-tree.

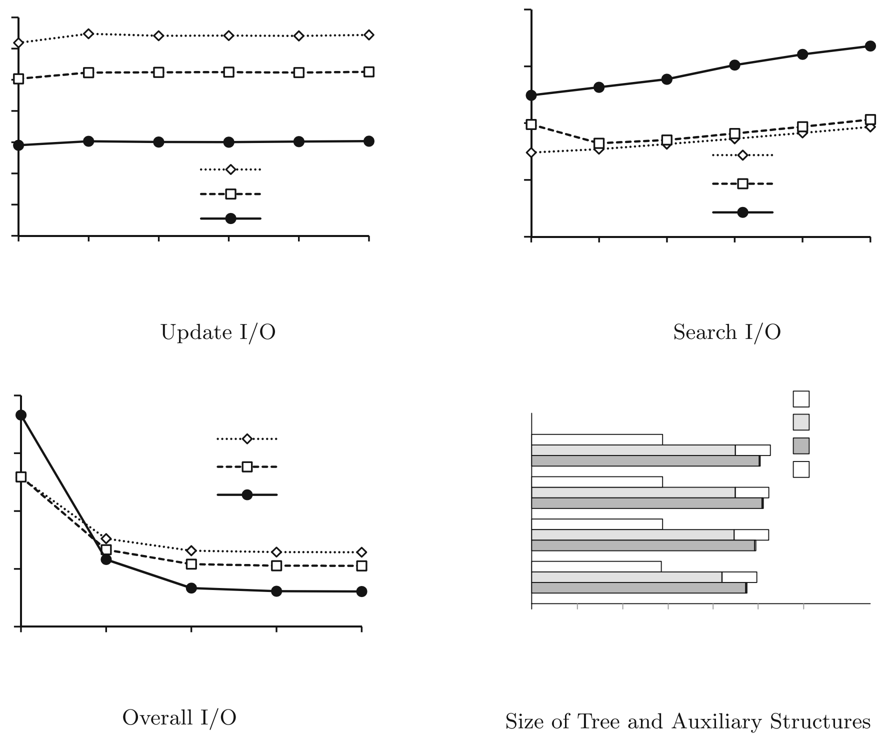
5.3 Performance while varying object extent

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 as *object extent*) varies from 0 to 0.01.

5.3.1 Update cost

Figure 13a gives the average update I/O cost of the three R-tree variants for different values of object extent. For all the evaluated trees the I/O cost is in general invariant to the objects extent. The update cost of the RUM-tree is around 46% of the cost in the R*-tree, and is around 57% of the cost in the FUR-tree.

Fig. 13 Performance while varying object extent



5.3.2 Search cost

While the object extent does not affect signi cantly the update I/O cost of the studied tree structures, it does affect their query I/O cost as shown in Fig. 13b. In general, for all the approaches, the query I/O cost increases when the object extent increases. The R*-tree achieves the best performance followed by that of the FUR-tree. The search I/O cost of the RUM-tree is around 25–65% higher than that of the RUM-tree and around 70% higher than that of the R*-tree.

5.3.3 Overall cost

Figure 13c gives a comprehensive view of the I/O performance comparison when the object extent is set as 0.01. Again, we study the performance under various ratios of updates over queries. The RUM-tree outperforms both the R*-tree and the FUR-tree when the ratio is larger than or equal to 10:1. At the point 10,000:1, the average cost of the RUM-tree is only 57% of the FUR-tree and 47% of the R*-tree.

5.3.4 Size of tree and auxiliary structures

Figure 13d gives the size of the trees and auxiliary structures for different values of object extent. For all the approaches, the size of the tree and auxiliary structures are not affected signi cantly by the objects extent. The tree size of the R*-tree is around 34% smaller than that of the FUR-tree and 43% smaller than that of the RUM trees. Furthermore, although the tree size of the RUM-tree is 10% larger than that of the FUR-tree, the total size (tree and auxiliary structures) of the RUM-tree approach is 5% smaller than that of the FUR-tree approach.

5.4 Performance while varying query size

In this section, we study the scalability of the three R-tree variants when increasing the query size. In these experiments the queries are squares and their side length (referred to as query size) varies from 0.02 to 0.1.

5.4.1 Update cost

The analysis of the update I/O cost is included in this section only to facilitate the comparison of this cost with the update and overall I/O costs. As expected, the update I/O cost shown in Fig. 14a remains unaffected for all the studied trees when the size of the query increases. The update I/O cost of the RUM-tree is only 58% of that of the FUR-tree and 46% of that of the R*-tree.

5.4.2 Search cost

On the other hand, the query I/O cost of all the tree approaches, shown in Fig. 14b, increases when the query size increases. The reason being that when the query size increases, more objects qualify to be part of the answer sets;

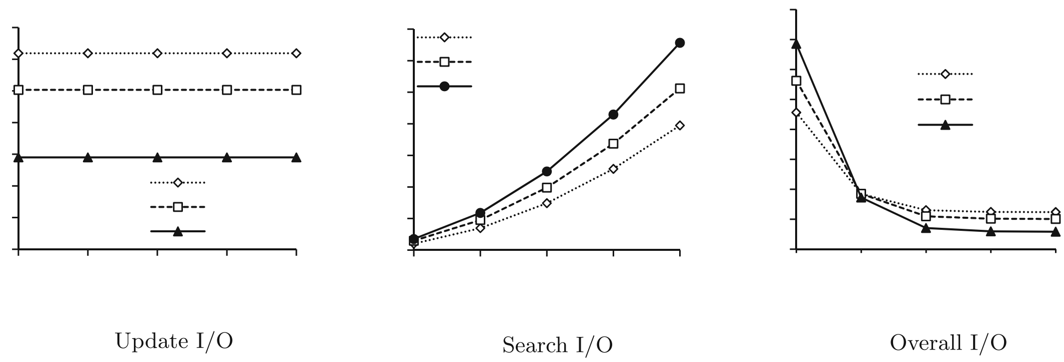


Fig. 14 Performance while varying query size

consequently more leaf nodes need to be read from disk. Additionally, for the RUM-tree approach, the number of obsolete entries increases when the query size grows.

The query I/O cost of the FUR-tree is higher than the query cost of the R*-tree because this approach extends the MBRs of the tree nodes. When the MBRs are extended, the overlap among the MBRs and the number of nodes that need to be analyzed to answer a given query increase too. The query I/O cost of the RUM-tree is higher than the query cost of the R*-tree because of the reduced tree fanout and the presence of obsolete entries. The query I/O cost of the R*-tree is 57–60% of the query cost of the RUM-tree and 68–77% of the query cost of the FUR*-tree. The query I/O cost of the FUR-tree is 77–84% of the query cost of the RUM-tree.

5.4.3 Overall cost

The comprehensive I/O costs of the R*-tree, the FUR-tree and the RUM-tree when the query size is set to 0.1 are given in Fig. 14c. The ratio of the number of updates to the number of queries varies from 1:1 to 10,000:1. The RUM-tree outperforms the other two R-tree variants when the ratio is larger than 10:1. When the ratio reaches 10,000:1, the average cost of the RUM-tree is only 58% of that of the FUR-tree, and only 46% of that of the R*-tree.

5.5 Scalability while varying the number of objects

In this section, we study the scalability of the three R-tree variants when increasing the data set up to 10 million point objects.

5.5.1 Update cost

Figure 15a gives the update I/O performance of the three R-tree variants for datasets of different sizes. When increasing the number of objects, the R*-tree exhibits a growing update cost. The reason is that more R-tree nodes are searched to locate the objects to be updated. In general, the update I/O cost of the FUR-tree increases slightly when the number of objects increases. This is mainly due to the higher number of updates that are performed in a top-down manner when the number of objects grows. For the RUM-tree, the update cost is basically unaffected by the number of objects. The reason is that the update cost of the RUM-tree is a combination of the cost of the insertion and the cost of the cleaning processes. Both factors, as analyzed in Sect. 4.2.3, are basically invariant to the size of the RUM-tree or the number of objects. The update cost of the RUM-tree, which also includes the cost of cleaning, is around 25–45% of the update cost of the R*-tree, and is around 55–65% of the update cost of the FUR-tree.

5.5.2 Search cost

Figure 15b gives the search I/O performance of the R-tree variants while varying the number of objects. The search I/O cost of all the approaches increases when the number of objects increases. The reason is that although the size of the queries remains the same, the number of the entries that satisfy the queries increases when the dataset size grows. The performance of the R*-tree and the FUR-tree are very similar and in general get closer when the number of objects increases. The reason is that for large datasets most updates on the FUR-tree are performed using the top-down approach and consequently the FUR-tree structure gets closer to the R*-tree structure. Due to the smaller fanout and the presence of obsolete entries, the search costs of the RUM-tree is higher than the search cost in the other two approaches. For instance, when the number of objects is 10 million, the search cost in the RUM-tree is 19% higher than that of the FUR-tree, and 21% higher than that of the R*-tree.

5.5.3 Overall cost

The comprehensive I/O costs of the R*-tree, the FUR-tree and the RUM-tree for a large dataset (10 million objects) are given in Fig. 15c. The RUM-tree outperforms the other two

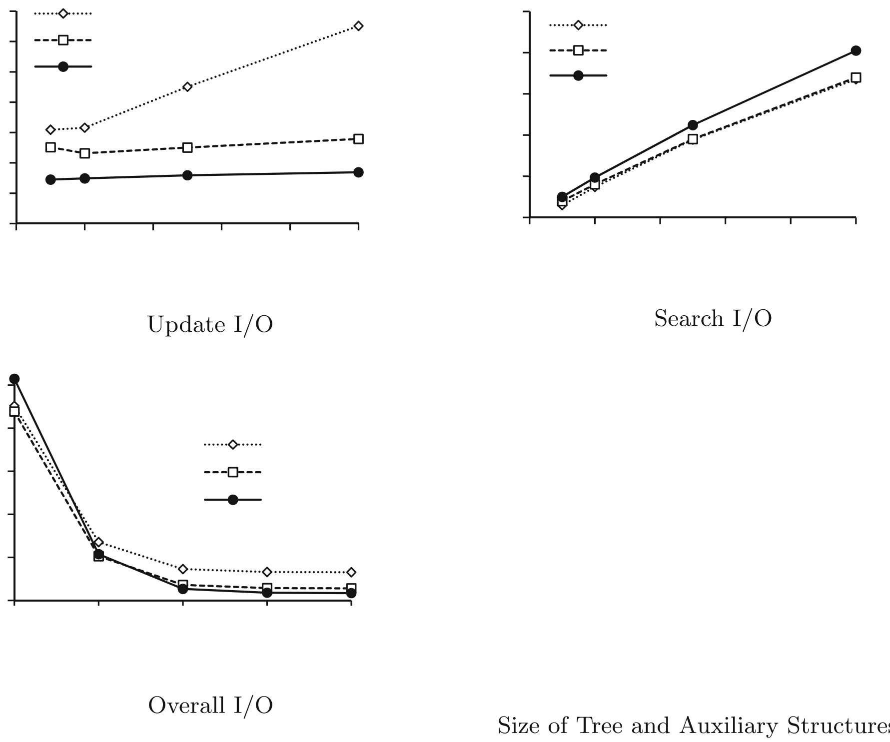


Fig. 15 Performance while varying number of objects

R-tree variants when the ratio is larger than 10:1. When the ratio reaches 10,000:1, the average cost of the RUM-tree is only 60% of that of the FUR-tree, and only 25% of that of the R*-tree.

5.5.4 Size of tree and auxiliary structures

Figure 15d gives the size of the trees and auxiliary structures for all the studied trees and different dataset sizes. For all the approaches, the size of the tree increases when the dataset size grows. This is naturally the case since more objects need to be stored in the trees. Furthermore the tree size of the FUR-tree gets closer to that of the R*-tree when the number of objects increases. As we stated before, the reason is that for large datasets most updates on the FUR-tree are performed using the top-down approach and the FUR-tree structure gets closer to the R*-tree structure. Although the tree size of the RUM-tree is around 10–20% greater than that of the FUR-tree under the different dataset sizes, the total size (tree and auxiliary structures) of the RUM-tree approach is in general slightly smaller than that of the FUR-tree approach.

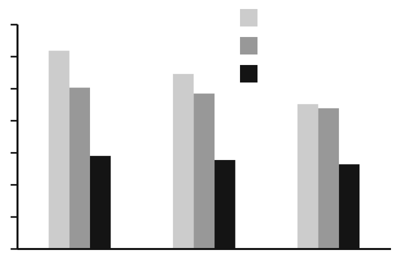
5.6 Performance using various datasets

In this section, we study the performance of the different trees under three different datasets: ROADS-SKW, ROADS-UNI, and UNIFORM. As explained in the introduction of the experimental section, ROADS-SKW, the dataset used in the previous experiments, tries to reproduce as close as possible the movement of cars (moving objects) on the roads a real city, i.e., the objects are more concentrated in the roads that belong to the downtown of the city. ROADS-UNI is similar to ROADS-SKW in that the objects are restricted to move only on the roads of the city but in this dataset the objects are distributed on the roads in a close to uniform fashion, i.e., objects are not more concentrated in the downtown roads. UNIFORM is not a road based dataset. In this dataset, the objects are uniformly distributed on the space and are assigned a random direction. Given the distribution properties of each dataset, we can consider ROADS-UNI as an intermediate state between ROADS-SKW and UNIFORM.

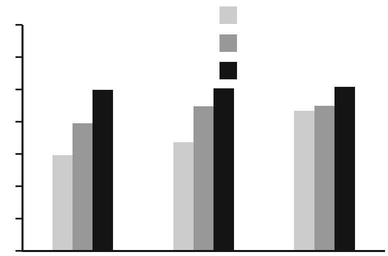
5.6.1 Update cost

Figure 16a gives the update I/O costs for the three R-tree variants and the three datasets. The update I/O cost decreases in all approaches when the data becomes more uniform. The reason is mainly a better tree structure and node utilization when the data gets more uniform. The update I/O cost of the RUM-tree is signi cantly smaller than the update cost of the other trees in all the datasets. However, the RUM-tree becomes slightly less advantageous when the data becomes more uniform, especially in comparison to the R*-tree. The update cost of the RUM-tree is 57% of that of the FUR-tree under the ROADS-SKW dataset. This update cost of the RUM-tree increases to be 60% of that of the FUR-tree under the UNIFORM dataset. On the other hand, the update cost of the RUM-tree is 46% of that of the FUR-tree under

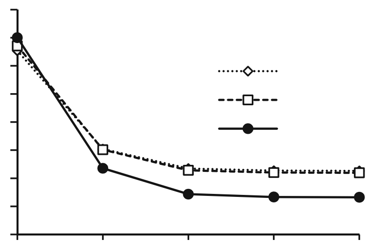
Fig. 16 Performance using various datasets



Update I/O



Search I/O



Overall I/O

Size of Tree and Auxiliary Structures

the ROADS-SKW dataset. This update cost of the RUM-tree increases to be 58% of that of the R*-tree under the UNIFORM dataset.

5.6.2 Search cost

The search performance of the three indexing types under the different studied datasets is given in Fig. 16b. The search I/O cost of the three approaches increases when the data becomes more uniform. The reason is that when data is uniform all the queries return approximately the same number of elements as their result set while when the data is more skewed, there might be a signi cant number of queries that return fewer or no objects. The search cost of the RUM-tree is always higher than the ones of the FUR and R* trees. However, this difference gets smaller when the data becomes more uniform. The search cost of the RUM-tree is 25% higher than that of the FUR-tree under the ROADS-SKW dataset. Under the UNIFORM dataset, the search cost of the RUM-tree is only 13% higher than that of the FUR-tree. On the other hand, the search cost of the RUM-tree is 67% higher than that of the FUR-tree under the ROADS-SKW dataset. Under the UNIFORM dataset, the search cost of the RUM-tree is only 17% higher than that of the R*-tree. An important conclusion is that when data gets more uniform, the advantage of the update performance of the RUM-tree in comparison the other two approaches decreases slightly, while the query performance of the RUM-tree becomes signi cantly closer to the ones of the other two approaches.

5.6.3 Overall cost

Figure 16c gives a comprehensive view of the I/O performance comparison under the UNIFORM dataset. Similar to the case of previous experiments that use the ROADS-SKW dataset, the RUM-tree performs better when the ratio increases. At the point 10,000:1, the average cost of the RUM-tree is only 60% of the FUR-tree and 58% of the R*-tree. This experiment demonstrates that the RUM-tree is more applicable than the R*-tree and the FUR-tree in environments with frequent updates and dynamic distribution of objects.

5.6.4 Size of tree and auxiliary structures

Figure 16d compares the sizes of the trees and auxiliary structures employed by the FUR, RUM and R* trees under the different datasets. The sizes of all the trees decrease when the data gets more uniform. This is due to the better tree structures and node utilization that can be achieved when the data is uniformly distributed on the space. Under all the datasets the tree size of the R*-tree is smaller than those of the FUR and RUM trees. However, the RUM-tree and FUR-tree sizes get closer to the R*-tree size when the data gets more uniform. In all the cases, the tree size of the RUM-tree is around 12–16% bigger than that of the FUR-tree. However, the total size of the RUM-tree (tree and auxiliary structures) is around 5–8% smaller than the total size of the FUR-tree (tree and auxiliary structures). The reason is that, under all the datasets, the auxiliary structures used by the RUM-tree approach are very small and upper bounded (less than 1% of the tree size) while the auxiliary structures used by the FUR-tree are very large (between 18 and 26% of the tree size).

5.7 Comparison of structure and dynamic properties of trees

This section presents the analysis of the dynamic properties of the R*, FUR, and RUM trees.

Given that the RUM-tree makes use of additional attributes in the nodes of the tree, the number of entries that can be stored in its nodes, i.e., fanout of the tree, is smaller than in the R* and FUR trees (assuming a xed node size). A reduced fanout has in general a negative effect on a tree since it increases the size of the tree and the number of nodes that need to be accessed to answer a query or to process an update operation. The previous experiments show that, although using a smaller fanout, the RUM-tree has better performance than the RUM and R* trees for multiple scenarios with frequent updates. This advantage is logically not free of cost and the cost is materialized in extra disk and main memory space required by the RUM-tree approach.

Figure 17 compares several important properties of the studied trees after the execution of 1 million updates and 100,000 queries.

The fanout of the RUM-tree (340) is approximately 83% of the fanout of the other two trees (409). In all the studied trees, we assume that all the nodes of a tree have the same fanout. The height of the tree in this experiment is the same for all the trees. The height remains the same even in other experiments with 10 million objects. The number of non-base internal nodes is also the same and is equal to 1 (the root node). The number of the base internal nodes (the level on top of the leaf level) in the RUM-tree is 33% higher than that of the FUR-tree and 85% higher than that of the R*-tree. Furthermore, the number of leaf nodes of the RUM tree is 12% higher than that of the FUR-tree and 64% higher than that of the R*-tree.

# of non-base int. nodes	1	1	1
# of base int. nodes	13	18	24
# of leaf nodes	3675	5378	6042
Tree space in RAM (MB)	0.10	0.14	0.16
Tree space on Disk (MB)	28.7	42.0	47.2
Tree space RAM+Disk (MB)	28.8	42.2	47.4
Total space (Tree+Aux. Str.) (MB)	28.8	49.8	47.6

Fig. 17 Comparison of structure and dynamic properties of trees

The size of a leaf node is the same in all the studied trees and is equal to 8 KB. Having the same leaf node size is important for the experimental section because it ensures a correct comparison of the I/O costs of the different trees. Given that non-leaf nodes are stored in main memory (referred to as RAM), these nodes can have a smaller size than the leaf nodes. This happens when the space required to store F entries (where F is the fanout of the tree) in a non-leaf node is smaller than the space required to store F entries in a leaf node. In our experiment, the non-leaf nodes sizes are 8 KB for the FUR and R* trees. In this case, leaf and non-leaf nodes have the same size since the size of a node entry is similar in both types of nodes. In the case of the RUM tree the non-leaf nodes have a smaller size than the leaf nodes. The reason is that the additional attributes used by the RUM-tree approach affect the leaf nodes more than the non-leaf nodes. In the case of a non-leaf node, only 1 or 3 pointers are added per node. One pointer (to parent) is added in the case of the non-base internal nodes and three pointers (to parent and siblings) in the case of base internal nodes. In the case of a leaf node, a timestamp eld is added for each entry of the node. Given that the size of the leaf node is xed (8 KB) the number of (extended) entries that t in a leaf-node of a RUM-tree is smaller than those of the FUR and R* trees. Speci cally, this number is 340, which is used as the fanout of the RUM-tree. The space required to store 340 entries in a non-leaf node is 6.7 KB. This size is signi cantly smaller than the sizes of the non-leaf nodes in the other two approaches (8 KB). The space used in RAM by the RUM-tree is 0.16 MB while the FUR-tree and R* tree require 0.14 and 0.1 MB, respectively. For all the indexes, the space used in RAM is a very small fraction of the space used on disk. The space used on disk by the RUM-tree is 47.2 MB, 12% more than the disk space used by the FUR-tree that uses 42 MB and 64% more of the disk space used by the R*-tree that uses 28.7 MB.

The previous discussion focuses on the analysis of the tree structures. It is also important to observe that, the size of auxiliary structures used by the RUM-tree index is always very small and upper-bounded while the auxiliary structures of the FUR-tree index are usually very large and make the total size (tree and auxiliary structures) of the FUR-tree index be greater than the total size of the RUM-tree index. In this experiment for instance, the size of the auxiliary structures used in the RUM-tree is only 0.2 MB while that of the FUR-tree is 7.7 MB. The total size (tree and auxiliary structures) used by the RUM-tree index is 47.6 MB while the total size (tree and auxiliary structures) used by the FUR-tree index is 49.8 MB.

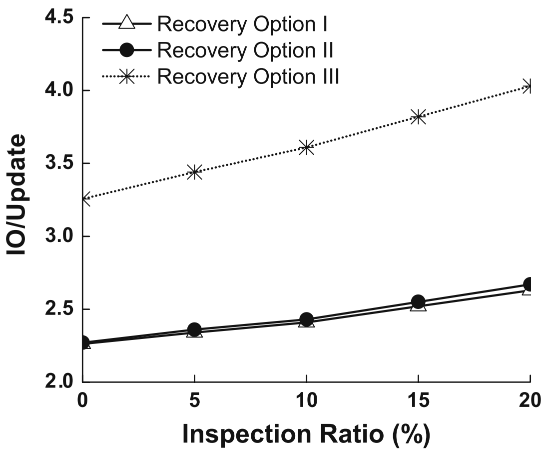


Fig. 18 Update I/O with log options

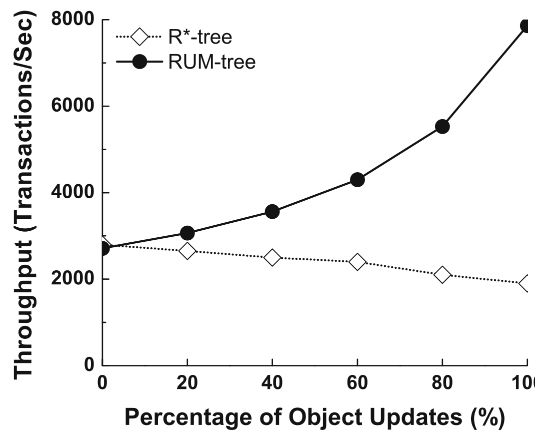


Fig. 19 Throughput comparison

5.8 Log and recovery

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

5.8.1 Update cost under logging

Figure 18 gives the overall I/O cost per update when the RUM-tree works with 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.

5.8.2 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.

Table 2 The number of I/Os for recovery

Option I	Option II	Option III
2,008,000	7,218	11

5.9 Throughput under concurrent accesses

Figure 19 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 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 an object id and its new value, the most costly part lies in searching the location in the R-tree of the objects to be updated. In contrast to 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. In frequent update scenarios, the RUM-tree 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.

Acknowledgement This work was partially supported by NSF Grant Number IIS-0811954.

References

1. Antonin Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD (1984)
2. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: an ef cient and robust access method for points and rectangles. In: SIGMOD (1990)
3. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica 6(2), (2002)
4. Chakka, P.V., Everspaugh, A., Patel, J.M.: Indexing large trajectory data sets with SETI. In: Proceeding of the Conference on Innovative Data Systems Research, CIDR (2003)
5. Chakrabarti, K., Mehrotra S.: Dynamic granular locking approach to phantom protection in r-trees. In: ICDE (1998)
6. Cheng, R., Xia, Y., Prabhakar, S., Shah, R.: Change tolerant indexing for constantly evolving data. In: ICDE (2005)
7. Hadjieleftheriou, M., Kollios G., Tsotras, V.J., Gunopulos, D.: Ef cient indexing of spatiotemporal objects. In: EDBT, pp. 251–268, Prague, March (2002)
8. Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E.: Main memory evaluation of monitoring queries over moving objects. Distrib. Parallel Databases 15(2), 117–135 (2004)
9. Kamel, I., Faloutsos, C.: Hilbert R-tree: an improved R-tree using fractals. In: VLDB, pp. 500–509 (1994)
10. Kim, K., Cha, S.K., Kwon, K.: Optimizing multidimensional index trees for main memory access. In: SIGMOD (2001)
11. Kollios, G., Gunopulos, D., Tsotras, V.J.: On indexing mobile objects. In: PODS (1999)
12. Kwon, D., Lee, S., Lee, S.: Indexing the current positions of moving objects using the lazy update R-tree. In: Mobile Data Management, MDM (2002)
13. Lee, M.-L., Hsu, W., Jensen, C.S., Teo, K.L.: Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In VLDB, (2003)
14. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-trees have grown everywhere. In: Technical Report, Available at http://citeseer.ist.psu.edu/706599.html (2003)
15. Nascimento, M.A., Silva, J.R.O.: Towards historical R-trees. In: Proceeding of the ACM Symposium on Applied Computing, SAC, pp. 235–240, February (1998)
16. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches in query processing for moving object trajectories. In: VLDB, pp. 395–406, September (2000)
17. Porkaew, K., Lazaridis, I., Mehrotra, S.: Querying mobile objects in spatio-temporal databases. In: SSTD, pp. 59–78, Redondo Beach, July (2001)
18. Prabhakar, S., Xia, Y., Kalashnikov, D.V., Aref, W.G., Hambrusch, S.E.: Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. IEEE Trans. Comput. 51(10), 1124–1140 (2002)
19. Procopiuc, C.M., Agarwal, P.K., Har-Peled, S.: STAR-tree: an ef cient self-adjusting index for moving objects. In: Proceeding of the Workshop on Algorithm Engineering and Experimentation, ALENEX, pp. 178–193, January (2002)
20. Roussopoulos, N., Leifker, D.: Direct spatial search on pictorial databases using packed r-trees. In: SIGMOD, pp. 17–31 (1985)
21. Saltenis, S., Jensen, C.S.: Indexing of moving objects for location-based services. In: ICDE (2002)
22. Saltenis, S., Jensen, C.S.: Indexing of now-relative spatio-bitemporal data. VLDB J. 11(1), 1–16 (2002)
23. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the positions of continuously moving objects. In: SIGMOD (2000)
24. Sellis, T.K.: Nick Roussopoulos, and Christos Faloutsos. The r+-tree: a dynamic index for multi-dimensional objects. In: VLDB, pp. 507–518 (1987)
25. Tao, Y., Papadias, D.: Ef cient historical R-trees. In: SSDBM, pp. 223–232, July (2001)
26. Tao, Y., Papadias, D.: MV3R-tree: a spatio-temporal access method for timestamp and interval queries. In: VLDB (2001)
27. Tao, Y., Papadias, D., Sun, J.: The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: VLDB (2003)
28. Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-temporal indexing for large multimedia applications. In: Proceedings of the IEEE Conference on Multimedia Computing and Systems, ICMCS, June (1996)
29. Xiong, X., Aref, W.G.: R-trees with update memos. In: ICDE (2006)