Main-Memory Operation Buffering for Efficient R-Tree Update

Laurynas Biveinis Simonas Šaltenis Christian S. Jensen
Department of Computer Science, Aalborg University, Denmark

ABSTRACT

Emerging communication and sensor technologies enable new applications of database technology that require database systems to efficiently support very high rates of spatial-index updates. Previous works in this area require the availability of large amounts of main memory, do not exploit all the main memory that is indeed available, or do not support some of the standard index operations.

Assuming a setting where the index updates need not be written to disk immediately, we propose an R-tree-based indexing technique that does not exhibit any of these drawbacks. This technique exploits the buffering of update operations in main memory as well as the grouping of operations to reduce disk I/O. In particular, operations are performed in bulk so that multiple operations are able to share I/O. The paper presents an analytical cost model that is shown to be accurate by empirical studies. The studies also show that, in terms of update I/O performance, the proposed technique improves on state of the art in settings with frequent updates.

1. INTRODUCTION

As sensor, communication, and computing technologies continue to evolve, it is becoming increasingly feasible and attractive to monitor, record, and query data that capture the evolving states of continuously changing real-world phenomena.

As an example that is representative of this new class of update-intensive applications, consider advanced location-based services, e.g., in the context of intelligent transport systems, where the positions of a large population of GPS-equipped moving objects are tracked on a server. To maintain accurately the current positions of such objects, the positions must be updated frequently. Experiments with GPS data from vehicles traveling in semi-urban environments show that to know the positions of the vehicles being tracked with an accuracy of 200 meters requires an update from each vehicle on average every 15 seconds [8].

The efficiency of the processing of updates determines the maximum update throughput, which in turn determines the maximum number of objects that can be tracked with a given accuracy.

Spatial indexes were originally developed with applications in mind that were characterized by relatively static data, and focus was on the efficient support for queries. In particular, the predominant spatial index, the R-tree [3, 11], supports queries efficiently, but updates are far from being efficient enough to support applications such as the one that involves the tracking of a large population of moving objects [15].

To improve the efficiency of spatial-index updates, a number of approaches have recently been proposed. The so-called bottom-up approaches aim to reduce the need for the expensive top-down tree traversals involved in R-tree deletions. They do so by offering direct leaf-level access [14]. Another approach is to try to avoid performing deletions on disk, by accumulating them in main-memory [23]. Yet another type of approach is to buffer and group operations in main memory or on disk with the purpose of enabling operations to share I/O's [1, 15].

With few exceptions [24], previous approaches assume a *persistent* index, such that, upon finishing the processing of an update, the update is recorded in the index on disk. This means that an update costs at least one I/O. In the types of applications we consider, even this cost is too high to support a high throughput of updates. The high update rates offer an opportunity to soften the index-persistence requirement: even if some recent updates are lost due to a system crash, after only a short waiting period, fresh updates will arrive for the objects affected. We note that existing approaches can achieve sub-one I/O updates by simply using a write-back disk page buffer, where dirty pages are not committed to disk after each operation. However, this use of the available main-memory is inferior when the objective is to speed up updates—the paper offers experimental results that demonstrate this.

Assuming the softer persistency requirements, this paper explores the efficient use of all available main memory to reduce the I/O costs of R-tree update operations as much as possible. The main contribution is a data structure, named the $\mathrm{R^R}$-tree, that consists of a main-memory R-tree and a conventional disk-based R-tree. The former enables efficient search on a main-memory buffer of update operations, i.e., both insertions and deletions. In contrast to previous research that uses multiple, large disk-based buffers of update operations and supports only batched queries [2], our proposal supports instantaneous queries.

When the operation buffer gets full, all operations in the buffer are clustered into groups. The operations that belong to the largest group(s) are then performed in bulk on the disk R-tree in order to minimize the I/O. This is similar to buffer emptying in the LGU algorithm [15], but is different from the complete emptying of the large, disk-based buffers used in buffer-tree approaches [2]. The paper includes an analytical study that justifies the selective buffer emptying strategy.

An empirical study is also reported. A setting is assumed in which the positions of moving objects are known only with some

VLDB '07, September 23-28, 2007, Vienna, Austria.

predefined accuracy. Thus, a moving object issues an update when its current position deviates by the predefined accuracy from the position it reported most recently. This setting improves on those where accurate positions are assumed. The empirical study validates the analytical modeling, and it offers insights into the performance properties of the R^R-tree. In particular, the study shows that the R^R-tree is able to significantly outperform an R-tree that uses an LRU buffer, which is the current state of the art. The study also shows that the R^R-tree significantly outperforms a recent proposal for an update-efficient R-tree.

In contrast to previous research on I/O efficient update of R-trees, the R^R-tree does not *require* some minimum amount of main memory to work. Thus, if no main memory is available, the R^R-tree works (largely) as a regular R-tree. If all data fits in main memory, the R^R-tree works as a main-memory R-tree. This behavior makes the proposal quite flexible.

Finally, because the R-tree heuristics are treated as black-boxes, the proposed approach is not limited to R-trees, but is easily extendible to the wider class of so-called grow-post trees, which includes variants of the R-tree [16].

The paper starts with a formulation of the problem setting in Section 2. The proposed data structure, algorithms, and their properties are described in Section 3, and a cost model is presented in Section 4. The results of an experimental evaluation are reported in Section 5. Section 6 then covers related work. Finally, Section 7 concludes and identifies directions for future research.

2. PROBLEM SETTING

2.1 Data And Update Operations

Consider a location-based service or fleet management application where a server tracks the current positions of a population of moving objects. It is a fundamental observation that, due to the discrete sampling of the movements and the inherent inaccuracy of positioning technologies, the positions of the objects cannot be known accurately at the server. Instead, the position of an object o may be assumed to be known with an accuracy thr_o [8, 21].

The inaccuracy was only recently taken into account in moving-object indexing research [12, 13]. In particular, two approaches may be used that take into account the inaccuracy of positions while still achieving perfect recall of queries, meaning that queries are guaranteed to return all qualifying objects, possibly in addition to false positives. With *data enlargement*, the position of an object o is represented by a circle with radius thr_o, which can be replaced by a bounding square for indexing purposes. With the less flexible *query enlargement*, a global maximum accuracy threshold thr is used, and each query region is enlarged according to this threshold.

We assume data enlargement; thus, the database stores pairs consisting of an object id, $objId$, and the object's current position, p, represented by a square, which as a special case can degenerate to a point. Such an $\langle objId, p\rangle$ pair is termed an *identification tuple*.

We classify database update operations as object insertions, object deletions, and object updates. When an object reports its position for the first time, an insertion occurs: a new identification tuple $\langle objId, p_{new}\rangle$ is inserted into the index. When an object stops being tracked by the system, a deletion occurs: the identification tuple $\langle objId, p_{old}\rangle$ is deleted from the index. Updates are the most common operations in the workloads we consider. These consist of a pair of an index deletion and an index insertion that deletes the old identification tuple and inserts a new identification tuple: $\langle objId, p_{old}\rangle^-$, $\langle objId, p_{new}\rangle^+$, where "$-$" denotes a deletion and "$+$" denotes an insertion. In the following, we consider only index-level operations, so an (index) update operation denotes either an insertion or a deletion.

To simplify the presentation, we assume that the sequences of operations are valid. Thus, the operations concerning an object $objId$ come in the following order: $\langle objId, p_1\rangle^+$, $\langle objId, p_1\rangle^-$, $\langle objId, p_2\rangle^+, \ldots, \langle objId, p_{i-1}\rangle^-$, $\langle objId, p_i\rangle^+$.

As mentioned in the introduction, we do not require the index to be fully persistent. It is assumed that after a crash, the index is rebuilt by waiting until the objects report their positions, which will happen in a relatively brief time interval in a scenario with frequent updates. Alternatively, the proposed index can be used as a secondary index on otherwise persistent data. The index can then be rebuilt from scratch in the event of a crash.

Although the proposed data structure supports the same queries as an R-tree, for brevity, we focus on range queries that, given a query rectangle, return all objects such that their true positions intersect the rectangle.

2.2 The R-Tree

The R-tree is a height-balanced tree [11]. Each node consists of a set of entries. The identification tuples are stored as entries in the leaf nodes. The $objId$ values of these tuples may then be interpreted as pointers to additional data for the objects. Non-leaf node entries take the same form as the leaf-node entries, but an identifier now points to a tree node at the next level of the tree, and the position is a minimum bounding rectangle (MBR) that contains the (approximations of the) positions of all objects stored in the subtree pointed to.

The size of a node is that of a disk page. This size determines the maximum fan-out of the index, which is the maximum number of entries that fit in a node. The actual fan-out of any non-root node is between the maximum and some value that is no larger than the ceiling of the maximum divided by 2. The actual fan-out of the root is at least 2. The R-tree is equipped with update algorithms that ensure these properties. The algorithms eliminate underfull nodes and reinsert their entries, and they split overfull nodes.

Figure 1 shows an example of an R-tree that indexes the positions of 9 objects.

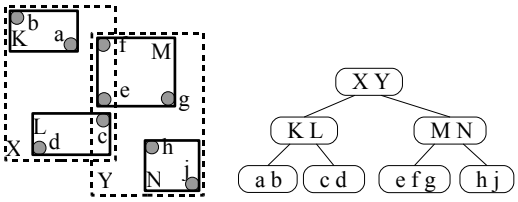

Figure 1: R-tree with maximum fanout 3, minimum fanout 2

The prototypical query supported by the R-tree is the range query: given a rectangle, this query retrieves all objects with positions that may overlap with this rectangle.

The R*-tree [3] is a variant of the R-tree that has the same data structure as the R-tree but is equipped with update algorithms that render it more efficient than the R-tree.

3. DATA STRUCTURE AND ALGORITHMS

Section 3.1 introduces the data structure and its properties. Sections 3.2–3.6 describe the update algorithms and their subroutines, simultaneously proving selected properties. Section 3.7 illustrates the workings of these algorithms by means of an example. Finally, Section 3.8 covers range search.

3.1 The R$^{\mathrm{R}}$-Tree Data Structure

The R$^{\mathrm{R}}$-tree data structure consists of two R-trees: a disk-based tree and an *operation-buffer* tree in main memory. The disk-based tree is a standard R-tree with the identification tuples introduced in Section 2 as leaf-node entries. In contrast, the operation-buffer tree stores operations yet to be performed on the disk-based tree. Its leaf-node entries thus consist of identification tuples augmented with *deletion flags* that indicate whether an entry represents an insertion or a deletion. In the following, if not specified otherwise, the terms *entry* and *node* refer to entries and nodes of the disk-based R-tree.

The operation buffer has room for C_{max} pending insertion and deletion operations, and the current number of operations in the buffer is denoted by C, $C \leq C_{max}$. Since flushing of the operation buffer to the disk-based tree is done only at certain points in time, the buffer might contain operations that invalidate some of the data in the disk tree. The property below characterizes the relation between the data on disk and the operations in the buffer. We use ω_d to denote a data entry in the disk-based R-tree, and ω_m^+ (ω_m^-) to denote a corresponding insertion (deletion) operation in the main-memory operation buffer.

Property 1. For any given identification tuple $\omega = \langle objId, p \rangle$ from the sequence of operations, exactly one of the following four cases is true at any given time. If the last operation concerning ω was a deletion, (1) there is no ω_d, no ω_m^+, and no ω_m^-, or (2) there is a ω_d and an ω_m^-. If the last operation concerning ω was an insertion, (3) there is no ω_d, but there is an ω_m^+ , or (4) there is an ω_d, and there is no ω_m^+ and no ω_m^-. □

Note that if no operations concerning ω have yet been performed, case 1 of the property holds trivially.

Figure 2 illustrates the four cases and the possible transitions between them. While the operation buffer is a main-memory R-tree, for simplicity, it is depicted as a list in figures. Also note that Property 1 is not formulated with respect to $objId$ or p alone. There may naturally exist several objects in the index that share the same location (equal p values). Further, several obsolete locations for the

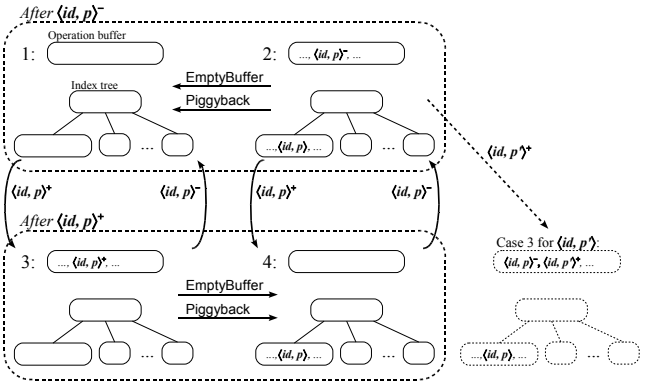

Figure 2: Possible states of the index for the $\langle id, p \rangle$ tuple

same object (equal $objId$ values) might be recorded in the index together with the object's current location. This case is illustrated in the bottom-right corner of the figure, where p' is the current position of the object id and p is an old position.

3.2 Insertion and Deletion Operations

Insertions and deletions are implemented by the same UPDATE algorithm, using the $delFlag$ parameter to distinguish the two types of operations (see Algorithm 1). This algorithm first checks whether the operation buffer (buf) has room for the current update operation and empties part of the buffer if necessary (EMPTYBUFFER). Then it checks if the opposite operation already exists in the buffer by querying it. An opposite operation is defined as the operation with the same p and $objId$, but with the opposite deletion flag value. If such an operation is found, the previous operation is removed from the buffer and the incoming operation is ignored. We call this an

Algorithm 1 The UPDATE algorithm

```
UPDATE(objId, p, delFlag)
1  if buf is full
2    then EMPTYBUFFER
3  oppositeEntry ← ⟨objId, p, ¬delFlag⟩
4  if buf.exists(oppositeEntry)
5    then buf.remove(oppositeEntry)
6    else buf.insert(⟨objId, p, delFlag⟩)
```

annihilation. Otherwise, if the opposite operation is not found, the incoming operation is inserted into the operation buffer.

The lemma that follows implies that the annihilation algorithm is correct with respect to Property 1.

LEMMA 1. *Property 1 holds after invocation of the* UPDATE *algorithm if $C < C_{max}$ holds as a precondition (the buffer is not full and* EMPTYBUFFER *is not called).*

PROOF. Proof by induction. In the case of a newly created, empty tree, Property 1 holds trivially. Assume that Property 1 holds before an UPDATE. Next assume that we are about to perform an insertion for $\omega = \langle objId, p \rangle$. This means that the most recent update for ω was a deletion or this is the first insertion for ω. As illustrated in Figure 2, the index is then either in state 1 or in state 2. If it is in state 1, processing an insertion is trivial: it is added to the buffer, transforming the index into state 3.

Processing ω^+ when the index is in state 2 yields a relatively rare case of annihilation. For example, this may occur when a stationary object repeatedly sends updates. In this case, an insertion arrives when there is already a deletion in the buffer for the same object at the same position. The annihilation removes both operations, transforming the tree into state 4: $\{\omega_d, \omega_m^-\} + \omega^+ = \omega_d$.

If the processed operation is a deletion then the most recent update was an insertion and the index is either in state 3 or in state 4.

In state 3, processing ω^- results in the most common case of annihilation, transforming the state of the structure into state 1: $\omega_m^+ + \omega^- = \emptyset$.

Finally, in state 4, ω^- is simply added to the buffer, transforming the index into state 2: $\omega_d + \omega^- = \{\omega_d, \omega_m^-\}$. □

While the annihilation in state 3 is intuitive, to better understand the intuition of annihilation in state 2, consider what would happen if the involved operations were allowed to complete on the disk tree one by one. The deletion from the buffer first removes the data from the tree; then, later, the insertion creates an entry that is identical to the one removed. The combined effect is that the tree transforms into state 4, the effect of which is identical to that of the annihilation.

3.3 EmptyBuffer Algorithm

The EMPTYBUFFER algorithm, presented in Algorithm 2[1], is responsible for committing some of the operations in the buffer to

[1]For brevity, we do not distinguish between nodes and entries pointing to these nodes in the pseudo code descriptions of the algorithms. Consequently, all MBR update and node reading and writing operations are implicit.

disk. Note that although all operations are removed from the buffer in line 2 of the pseudo code, some of them are usually put back into the buffer in the GROUPUPDATE algorithm, described in Section 3.4. In addition, with the appropriate bookkeeping, the implementation of the algorithm needs not materialize the *opList* as is done in the pseudo code.

Algorithm 2 The EMPTYBUFFER algorithm

```
EMPTYBUFFER()
1  opList ← buf.GetAllOperations
2  buf.remove(opList)
3  siblings ← GROUPUPDATE(rootNode, opList)
4  while |siblings| > 1
5  do rootNode ← newNodeWithEntries(siblings)
6     siblings ← GROUPSPLIT(rootNode)
```

The GROUPUPDATE algorithm performs the argument operations on the root node of the disk tree and returns a set of root-level sibling nodes, i.e., subtrees of the same height as the disk tree, including the original root. If this set contains other nodes than the original root, the tree is grown by creating a new root. It is possible that there were so many sibling entries that the new root overflows. In that case, the new root is split, which grows the tree further.

The GROUPSPLIT algorithm used in line 6 (and as a subroutine of GROUPUPDATE) checks whether the node is overflowing. If not, it returns the original node; otherwise, the node is repeatedly split by the regular R*-tree split algorithm [3] until none of the resulting nodes are overflowing, in which case all the nodes created are returned. Although more advanced implementations of GROUPSPLIT are possible [6], we do not consider them. This is because the node is usually split no more than once, as the number of buffer entries processed is typically a small fraction of the size of the tree disk.

3.4 GroupUpdate Algorithm

The GROUPUPDATE, presented in Algorithm 3, is the core of the proposed indexing technique. It executes the specified operation list (a sublist of the operations formerly present in the buffer) on the specified subtree.

Algorithm 3 The GROUPUPDATE algorithm

```
GROUPUPDATE(node, opList)
 1  if node is leaf
 2    then EXECUTEENTRIES(node, opList)
 3    else pushG ← GETPUSHDOWNOPS(opList, node)
 4         regroup ← false
 5         for each ⟨child, opSublist⟩ in pushG
 6         do if regroup
 7              then return GROUPUPDATE(node, pushG)
 8            pushG ← pushG \ opSublist
 9            node.remove(child)
10            newCh ← GROUPUPDATE(child, opSublist)
11            if newCh = {newChild}
12              then regroup
13                   ← INTEGRATECHILD(node, newChild)
14              else node.add(newChildren)
15  if |node| = 1 and node is non-leaf
16    then node ← node[1]
17  return GROUPSPLIT(node)
```

If the subtree is a leaf node then the operation list is executed on the node trivially by means of EXECUTEENTRIES, which just adds insertion entries to the node and removes existing entries matched by deletion entries. If the subtree is rooted at a non-leaf node, GETPUSHDOWNOPS returns a mapping from its child nodes to the individual operation sublists that should be executed on these nodes. As will be discussed later, some of the operations from *opList* can also be put back into the buffer and are thus not processed during this buffer emptying.

Each mapping in $pushG$ is then executed in turn by a recursive GROUPUPDATE call (line 10). This results in a list of new child nodes in $newCh$. The rest of the algorithm is concerned with integrating these nodes into the tree. In the simplest case, there are two or more such nodes. Their entries are trivially added to the current node. If only one child node is returned, further and more complicated treatment may be necessary in case the child is underfull or has shrunk in height. This is handled by INTEGRATECHILD and is discussed in the next section. This algorithm may change other sibling child entries, invalidating the remaining, non-processed mappings in $pushG$. This is signaled by the $regroup$ flag, which forces a restart of GROUPUPDATE. It is guaranteed that at least one mapping is completed before such a restart can occur; thus, the algorithm is guaranteed to terminate.

After performing all the operations, the current node might contain only one entry, in which case it is replaced by its child in the tree, thus shrinking the tree in height. The node may also end up overfull. It is then passed to GROUPSPLIT, which returns a list of non-overflowing nodes. This list is returned as the result of the algorithm.

The next lemma states that the UPDATE algorithm complies with Property 1 when EMPTYBUFFER is called. The proof assumes that those operations that reach the leaves of the disk tree are correctly routed there by the GETPUSHDOWNOPS algorithm.

LEMMA 2. *Property 1 holds after invocation of the* UPDATE *algorithm if $C = C_{max}$ holds as a precondition (the buffer is full).*

PROOF. According to the Lemma 1, Property 1 holds as a precondition. For every identification pair ω that is contained in the buffer, the tree might be in either state 2 or state 3. EMPTYBUFFER will either leave the ω in the buffer, or the operation will be performed correctly on the disk tree and removed from the buffer. In the latter case, the index will move from state 2 to state 1, or from state 3 to state 4 (see Figure 2). After buffer emptying, the incoming operation will be processed as in Lemma 1. □

A consequence of Lemmas 1 and 2 is the following theorem. The proof is trivial and thus omitted.

THEOREM 1. *After invocation of the* UPDATE *algorithm, Property 1 holds.*

3.5 IntegrateChild Algorithm

The INTEGRATECHILD algorithm (Algorithm 4) is responsible for adding an arbitrarily-looking R-tree to the current node. Several cases must be handled. One case is where the root of the new tree is the only child of the current node. Then the current node is removed and the child node takes its place. More commonly, the root of the new tree has siblings in the current node and is either underfull or normal. The contents of an underfull node are merged into the sibling nodes by the MERGESUBTREES call, whereas a normal node is inserted into tree by INSERTSUBTREE, as discussed shortly.

Subroutines INSERTSUBTREE and MERGESUBTREES serve similar purpose—putting a subtree into the main tree—and have similar implementations. The principal difference is that INSERTSUBTREE handles a subtree whose root is a normal (not underfull) node, whereas MERGESUBTREES handles the case of an underfull node.

Algorithm 4 The INTEGRATECHILD algorithm

```
INTEGRATECHILD(node, newCh)
1  if |newCh| ≠ 0
2    then if |node| = 0
3           then node ← newCh
4           else if newCh is underfull
5                  then
6                    return MERGESUBTREES(node, newCh)
7                  else
8                    return INSERTSUBTREE(node, newCh)
9  return mustNotRegroup
```

The INSERTSUBTREE (Algorithm 5) checks whether the specified node is at the right level for insertion of the subtree, i.e., the distance to the leaf level from the node in the main tree must be equal to the subtree height.

Algorithm 5 INSERTSUBTREE algorithm

```
INSERTSUBTREE(node, subtree)
1  if subtree is one level below node
2    then node.add(subtree)
3    else childNode ← CHOOSESUBTREE(node, subtree)
4         INSERTSUBTREE(childNode, subtree)
5         childNodes ← GROUPSPLIT(childNode)
6         node.add(childNodes)
7         if |childNodes| > 1
8           then return mustRegroup
9  return mustNotRegroup
```

If so, a new entry for the subtree is created and inserted into the node. This case is the most common, and it does not incur any additional I/O. However, if the specified node is not at the right level to accommodate the subtree, then, using the minimum bounding rectangle of the subtree, the standard R*-tree CHOOSESUBTREE algorithm is called to identify an appropriate node one level below, and INSERTSUBTREE is called recursively. After returning from the recursive call, GROUPSPLIT is called on the potentially overflowing child node, and the resulting entries are inserted into the parent node.

Since GROUPSPLIT may replace one child node with two new child nodes, it invalidates any parent GROUPUPDATE mappings for this node. This situation is signaled to the caller by the return value.

The MERGESUBTREES algorithm is very similar to INSERTSUBTREE. The only difference is that when the node at the right level is found, it is merged with the underfull node passed as an argument to the algorithm.

To complete the definition of the update algorithms, the GETPUSHDOWNOPS algorithm is discussed next. It is used in algorithm GROUPUPDATE, to route groups of operations down the disk tree.

3.6 GetPushDownOps Algorithm

Given an operation list and a node of the disk tree, the GETPUSHDOWNOPS algorithm (Algorithm 6) determines which of the operations should be pushed down and to which children of the node. First, the GROUPOPERATIONS subroutine is called to establish the mapping between the operations and the child nodes according to the usual R*-tree rules (see Algorithm 7). The mapping returned is a list of $\langle childNode, opSublist \rangle$ tuples . Each insertion is mapped to a single child node according the R*-tree's CHOOSESUBTREE algorithm [3]. Each deletion is mapped to all

Algorithm 6 The GETPUSHDOWNOPS algorithm

```
GETPUSHDOWNOPS(node, opList)
1  mapping ← GROUPOPERATIONS(node, opList)
2  if node is the root
3    then for each ⟨chNode, sublist⟩ in mapping
4         do if |sublist| < k
5              then
6                mapping ← mapping \ {⟨chNode, sublist⟩}
7                buf.insert(sublist)
8  return mapping
```

Algorithm 7 The GROUPOPERATIONS algorithm

```
GROUPOPERATIONS(node, opList)
1  opMapping ← ∅
2  for each op in opList
3  do if op is a deletion
4       then matchingChildren ← Covering(node, op)
5            for each chNode in matchingChildren
6            do opMapping ← opMapping ∪ {op → chNode}
7       else childNode ← CHOOSESUBTREE(node, op)
8            opMapping ← opMapping ∪ {op → childNode}
9  return opMapping
```

child nodes whose bounding rectangle includes it (as returned by the *Covering* method in line 4 of the algorithm).

As shown in Algorithm 6, if the node is not the root, the algorithm just returns the mapping obtained from the call to GROUPOPERATIONS. If the node is the root, the algorithm removes from the mapping all operation sublists that have fewer than k operations. Operations in these sublists are put back into the buffer. The idea using an operation threshold for the buffer emptying is to avoid spending expensive I/O on the small groups, so that as many operations as possible share the same I/O.

Choosing a threshold k is not trivial. If a large value for k is chosen, this will cause only very few groups to be emptied, leading in turn to the need for very frequent buffer emptying. Choosing a small k value also has obvious drawbacks. The analytical cost modeling in Section 4 provides a way to choose an optimal value of k.

Note that because some sublists of operations are put back into the buffer, one copy of a deletion may be put back into the buffer while another is sent down the tree. For this reason, if the deletion succeeds to delete an entry in the index tree, the EXECUTEENTRIES algorithm (called from GROUPUPDATE) removes the corresponding copy of the deletion operation from the buffer if there is one.

Two special cases are not shown in the pseudo code. If all sublists of operations have fewer than k operations, one largest sublist is sent down the tree. Finally, a special and rare case occurs when only groups, consisting solely of copies of deletions, are sent down the tree. If none of these deletions are successful, no operations are removed from the buffer. If this occurs, we empty the whole buffer.

Having described all the algorithms needed for maintaining the R^R-tree, we proceed to illustrate the workings of the algorithms.

3.7 Example

Figure 3 shows a small example were a number of update operations are performed, causing a buffer emptying. The leftmost part of the figure shows the positions of eight objects stored in an R-tree. The four empty circles (a_1, a_2, c_2, j) represent insertions that are in the buffer or future updates that will be discussed in the

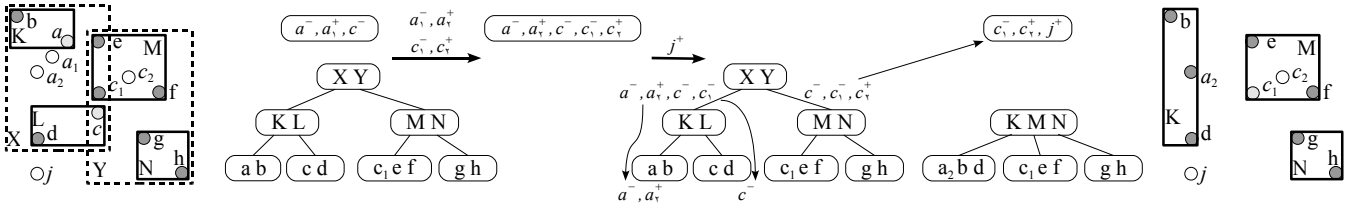

Figure 3: Performing a set of operations on the R^R-tree

following. The light gray circles (a, c) represent the positions that are in the disk tree, but that have corresponding deletion entries in the buffer. In the example, object a moves from a to a_1 to a_2, and object c moves from c to c_1 to c_2. We assume a disk R-tree with a maximum fan-out of 3 and a minimum fan-out of 2. The buffer size is $C = 5$ and the operation threshold is $k = 4$.

Note that the initial state of the tree holds two positions, c and c_1, for object c. This happened after the part of the buffer containing the insertion of c_1 was emptied, while the deletion of c was left in the buffer. Next, a and c update their positions, resulting in the following sequence of updates: a_1^-, a_2^+, c_1^-, c_2^+. The deletion of a_1 and the corresponding insertion already in the buffer annihilate. The remaining three operations are inserted into the buffer.

The insertion of a new object j triggers a buffer emptying. The figure shows how the GROUPOPERATIONS algorithm divides the operations into two groups: one for node X and one for node Y. Note that, due to the overlap between the MBRs of X and Y, c^- and c_1^- are copied into both groups. Because the group for node Y has fewer than k elements, its entries are put back into the buffer. The operations in the other group proceed down the tree to nodes K and L, as shown in the figure. Note that, after reading node X, the algorithm discovers that c_1 can not be found in this subtree and c_1^- is not sent further, but c^- is sent to node L, it succeeds, and its copy is removed from the buffer.

The performed operations result in a number of structural changes to the disk tree. Node L is rendered underfull and is merged with node K. As a result, node X contains only one child; thus, X is replaced by its child, reducing the height of the subtree to one. The shorter subtree is then inserted into node Y using the INSERTCHILD algorithm. Finally, the single-entry root is removed, finishing the buffer emptying. At last, the operation that caused the buffer emptying, j^+, is inserted into the buffer. The resulting disk tree and buffer are shown at the rightmost end of the figure.

Note that the demonstrated shrinking (and growing) of subtrees are very rare for realistic node sizes and workloads that mostly contain deletion-insertion pairs.

3.8 Search Algorithm

The SEARCH algorithm in Algorithm 8 is an adjusted standard R-tree search algorithm. It operates by first querying the disk tree. The resulting answer set is modified to take the buffer data into account. Thus, objects that have corresponding deletions in the buffer are removed from the set, and any objects from the buffer that overlap with the query rectangle are included into the set. As the following theorem shows, the resulting set is the correct answer.

THEOREM 2. *The* SEARCH *algorithm, performed after a series of calls to the* UPDATE *algorithm, returns correct results.*

PROOF. Assume that the current position of object $objId$ is pc. In a series of update operations concerning object $objId$, the last operation for any identification tuple $\langle objId, p\rangle$ $(p \neq pc)$ is a deletion. According to Theorem 1, the index is in state 1 or in state 2 with respect to these identification tuples (see Figure 2). Trivially, the query will not return any such identification tuples corresponding to state 1. Identification tuples corresponding to state 2, even if chosen in line 4 of the algorithm, are subsequently eliminated by matching entries from the buffer in line 7.

Now let the last operation for $\langle objId, pc\rangle$ be an insertion. According to Theorem 1, the index is in state 3 or in state 4 with respect to this identification tuple (the tuple is either in the buffer or in the disk tree). Thus, if pc satisfies the query, it will be returned either from the disk tree (line 4) or the buffer (line 9). □

Algorithm 8 SEARCH algorithm

```
SEARCH(rectangle)
1  leafNodes ← GETOVERLAPPINGLEAFNODES(rectangle)
2  answer ← ∅
3  for each leaf in leafNodes
4  do answer ← answer ∪ SEARCHNODE(leaf, rectangle)
5     PIGGYBACK(node)
6     for each entry in answer
7     do if there is a matching deletion in the operation buffer
8           then answer ← answer \ {entry}
9     return answer ∪ buf.SearchInsertions(rectangle)
```

In addition to querying, the SEARCH algorithm takes the opportunity to perform some of the operations in the buffer on the leaf nodes of the disk tree that are read during querying. This is done by calling PIGGYBACK presented in Algorithm 9.

Potentially all the insertion entries that are contained in the node MBR and all the deletion entries that can remove corresponding node entries might be performed and thus removed from the buffer. However, the number of operations actually performed is limited by the requirement to keep the node from neither overflowing nor underflowing, to avoid the cost and complexity of having to adjust the tree.

Algorithm 9 PIGGYBACK algorithm

```
PIGGYBACK(node)
1  matchingBufEntries ← SEARCHBUFFER(MBR(node))
2  i ← getNumOfInsertions(matchingBufEntries)
3  d ← getNumOfDeletions(matchingBufEntries)
4  newSize ← |node| + i − d
5  if newSize > S_max
6     then i ← i − (newSize − S_max)
7     else if newSize < S_min
8             then d ← d − (S_min − newSize)
9  EXECUTEFROMBUFFER(matchingBufEntries, node, i, d)
```

The operations are executed by means of an algorithm called EXECUTEFROMBUFFER, which is similar to EXECUTEENTRIES used in GROUPUPDATE, except that it takes two additional parameters that specify thresholds for the number of insertions and deletions that can be executed. It also removes the completed operations from the buffer.

In Algorithm 9, S_{min} and S_{max} is the minimum and the maximum number of allowed entries in an R-tree node, respectively.

Because such *piggybacking* corresponds to the emptying of part of the buffer, the reasoning in the proof of Lemma 2 also applies to show that PIGGYBACK does not violate Property 1. Thus, query-result correctness is not compromised. Also, note that piggybacking is designed to be a performance optimization that can be disabled without any effects in the correctness of SEARCH or any other algorithm.

4. ANALYTICAL COST MODELING

As mentioned in Section 3.6, choosing the right value for the operation threshold (k) is non-trivial. The cost of buffer emptying has to be balanced against the frequency of the buffer emptying. Nevertheless, the emptying of larger groups at the root level should intuitively lead to more sharing of I/O operations. This section describes a cost model that confirms this intuition. More specifically we show that if the buffer size is not very small compared to the total number of objects, the best value for k is the size of the largest group of operations, i.e., only the largest group of operations has to be emptied from the buffer on each buffer emptying.

Because the objective of the cost model is not to model the performance accurately in absolute terms, we are able to make a number of simplifying assumptions. We explain these assumptions at the beginning of Sections 4.1 and 4.2.

We first model how often the buffer is emptied and then estimate the I/O cost of the buffer emptying. The notation used is summarized in Table 1.

C	Current number of operations in the buffer
C_s	Starting number of operations in the buffer
C_{max}	Capacity of the buffer
O	Number of objects
k	Operation threshold
S	Average fan-out of disk tree nodes
R	Fan-out of the root
X	Sequence of update operations
W	Expected number of buffer entries
K	I/O costs associated with buffer emptying

Table 1: Cost Model Notation

4.1 Frequency of EmptyBuffer Invocations

To enable the estimation of the frequency of EMPTYBUFFER invocations, we assume that the number of objects being tracked is O and that $C_{max} < O$. We also assume that the index is in *steady state*, such that the set of objects being tracked neither grows nor shrinks. A workload then consists of pairs of deletions and insertions. As we assume a general setting where updates are much more frequent than queries, we assume that the effects of piggybacking on the buffer can be ignored.

Assume that the an invocation of EMPTYBUFFER has emptied x operations from the buffer, so $C_s = C_{max} - x$ operations remain in the buffer. We then proceed to determine how many operations that can be entered into the buffer before the buffer gets full and the next invocation of EMPTYBUFFER occurs. To do that, we need to consider the effects of annihilation because these reduce the size of the buffer.

The second case of annihilation, where an incoming insertion annihilates with a deletion in the buffer (the transition from state 2 to state 4 in Figure 2), is rare, as it occurs only when an object sends duplicate updates in rapid succession.

The first case of annihilation, where an incoming deletion annihilates with an insertion in the buffer, is more frequent. Its probability depends on the number of insertions currently residing in the buffer. Assuming that operations come in deletion-insertion pairs and buffer emptying empties equal amounts of deletions and insertions, half of the operations in the buffer are deletions and half of the operations are insertions, i.e., at any time, there are $C/2$ insertions in the buffer and the probability of annihilation is $C/(2O)$.

To see why this is true, consider what happens when a deletion-insertion pair is added to the buffer. If no annihilation occurs, the number of insertions and the number of deletions in the buffer are each increased by one. If the deletion annihilates with an insertion already in the buffer, the addition of the new insertion results in unchanged numbers of deletions and insertions in the buffer. Summarizing, a deletion-insertion pair leaves C unchanged with probability $C/(2O)$ and increases C by 2 with complementary probability $1 - C/(2O)$.

Between buffer emptyings, the probability of C increasing (P_{inc}) varies from $\max(P_{inc}(C)) = P_{inc}(C_s) = 1 - C_s/(2O)$ to $\min(P_{inc}(C)) = P_{inc}(C_{max}) = 1 - C_{max}/(2O)$. Such a process can be modeled precisely as a Markov chain, but, to simplify the analysis, P_{inc} is assumed to be constant and equal to the average of the minimum and the maximum values:

$$P_{inc} = \frac{\min(P_{inc}(C)) + \max(P_{inc}(C))}{2} = 1 - \frac{C_s + C_{max}}{4O}$$

We now know the probability with which an incoming deletion-insertion pair will increase the buffer size by two. Assume a workload X of operations consisting of $|X|/2$ deletion-insertion pairs. Starting with C_s operations in the buffer, and assuming that the buffer does not become full in the process, the expected final number of entries in the buffer after processing the workload is:

$$\begin{aligned} W(X, C_s, C_{max}) &= C_s + 2P_{inc}\frac{|X|}{2} \\ &= C_s + \frac{|X|(4O - (C_s + C_{max}))}{4O} \end{aligned} \quad (1)$$

Then the expected buffer-filling workload size $|X|$ can be trivially derived from Equation 1 as follows:

$$\begin{aligned} C_{max} &= W(X, C_s, C_{max}) \\ \Rightarrow\ & |X|(C_s, C_{max}) = \frac{4O(C_{max} - C_s)}{4O - (C_s + C_{max})} \end{aligned}$$

Expressing $|X|$ as a function of C_{max} and x, where $x = C_{max} - C_s$, we obtain:

$$|X|(C_{max}, x) = \frac{4Ox}{4O - 2C_{max} + x} \quad (2)$$

The next section models the I/O cost of buffer emptying.

4.2 EmptyBuffer Cost Model

All update I/O is due to EMPTYBUFFER invocations. In order to estimate the cost of EMPTYBUFFER, we assume that incoming operations are uniformly distributed in the data space; thus, an operation from the buffer is equally likely to be routed by the R-tree algorithms to any of the root's subtrees.

Next, we assume that a deletion is only sent to one subtree at the root level. This is a reasonable assumption for R-trees storing point or small-region data (which is the case in our setting) because this results in MBRs with little overlap. We also assume that each object has only one position recorded in the disk tree. Thus, cases such as the one shown for object c in Figure 3 are very rare. Finally, we assume that those sublists of operations that are sent to the subtrees of the root read and write all the nodes in these subtrees.

Under the above assumptions, we show that emptying only the largest group leads to the smallest amortized cost of update operations. We end by discussing when the last assumption holds.

The first step is to compare the amortized I/O cost per update operation resulting from using two different operation threshold values. Thus, we compute the cost when $k = k_{max}$, the size of the largest group of operations at the root. Then, we compare this cost with the cost when $k = k_{max} - \epsilon$ ($\epsilon \geq 0$), the size of the next largest group.

If K_1 is the cost of reading and writing one subtree of the root then the total cost of emptying the largest group from the buffer is $2 + K_1$ (including the reading and writing of the root node). Dividing this cost by the number of update operations that are necessary to fill the buffer again, we get the amortized update cost when only the largest group is emptied:

$$K_{U,1} = \frac{2 + K_1}{|X|(C_{max}, k_{max})}$$

Combining this with Equation 2, we get:

$$K_{U,1} = \frac{(2 + K_1)(4O - 2C_{max} + k_{max})}{4Ok_{max}} \quad (3)$$

If two largest groups are emptied ($2k_{max} - \epsilon$ operations in total), two subtrees of the root are accessed, and the corresponding amortized update cost becomes:

$$\begin{aligned} K_{U,2} &= \frac{2 + 2K_1}{|X|(C_{max}, 2k_{max} - \epsilon)} \\ &= \frac{(2 + 2K_1)(4O - 2C_{max} + 2k_{max} - \epsilon)}{4O(2k_{max} - \epsilon)} \\ &= \frac{(1 + K_1)(4O - 2C_{max} + 2k_{max} - \epsilon)}{4Ok_{max} - 2O\epsilon} \end{aligned} \quad (4)$$

Comparison of Equations 3 and 4 sheds light on when $K_{U,1} \leq K_{U,2}$. First, note that if the root node was kept in main memory, the first term in the numerators of both formulas would become simply K_1, as reading and writing the root would not be necessary. In this case, $K_{U,1}$ is always smaller than $K_{U,2}$.

We proceed to show that even if the root node is not pinned in memory, $K_{U,1} \leq K_{U,2}$ for sufficiently large C_{max}. In particular, the inequality $K_{U,1} \leq K_{U,2}$ can be simplified to:

$$(4O - 2C_{max})(2k_{max} - \epsilon(2 + K_1)) \leq K_1 k_{max}(2k_{max} - \epsilon) \quad (5)$$

To see when the inequality holds, we estimate k_{max} and K_1. The cost K_1 is equal to twice the average number of nodes in the subtree of the root. Due to the large fan-out of an index tree, the size of the subtree can be accurately approximated by the number of leaf nodes in the subtree, which is, on average, equal to $O/(SR)$, where S is the average fan-out of the disk R-tree nodes (except the root) and R is the current fan-out of the root. Thus, $K_1 \approx 2O/(SR)$.

The size of k_{max} depends on C_{max} and R. We assume that the size of the largest group is f times larger than the average group size, i.e., $k_{max} = fC_{max}/R$, where $f > 1$. With these estimates of K_1 and k_{max}, it is easy to prove that Equation 5 is satisfied when:

$$C_{max} \geq \frac{3SR^2}{f}.$$

This means that for large enough buffers, emptying only the largest group of operations is the most efficient strategy even if the root node of the disk R-tree is not kept in main memory.

We proceed to compute the smallest size of the group of operations at the root level, such that emptying this group from the operation buffer results in touching the whole subtree of the root with high probability. This is a key assumption of the analytical model and it provides, indirectly, another lower bound for C_{max}.

For an entire subtree to be touched when performing a group of operations, all leaves of the subtree must receive at least one operation. In general, distributing n objects among g groups with uniform probability results in groups of objects whose sizes follow the binomial distribution $B(n, 1/g)$ [18]. When k_{th} operations are distributed among $O/(SR)$ leaves, the number of operations reaching a leaf follows the distribution $B(k_{th}, SR/O)$. The probability that a leaf of the subtree will not get any operations then is:

$$f(0; k_{th}, SR/O) = \left(1 - \frac{SR}{O}\right)^{k_{th}}$$

We require that this probability is low, specifically lower than 0.05:

$$f(0; k_{th}, SR/O) \leq 0.05 \quad \Rightarrow \quad k_{th} \geq \frac{\ln 0.05}{\ln(1 - SR/O)} \quad (6)$$

Using the Taylor series centered at 1 to approximate the denominator, we get $k_{th} \geq 3O/SR$. For the cost model to apply, we require that $k_{max} - \epsilon \geq k_{th}$. Assuming that ϵ is small and that $k_{max} = fC_{max}/R$, this condition can be expressed as follows:

$$C_{max} \geq \frac{3O}{Sf}. \quad (7)$$

Factor f does not have a simple analytical form, but it is a small constant larger than one. In Section 5, we explore experimentally for which settings of C_{max} the average size of the largest group of operations satisfies Equation 6. How to choose the optimal value of k for small buffers, when Equation 6 is not satisfied, is an interesting topic for future research.

5. EXPERIMENTAL EVALUATION

In this section we describe the results of performance experiments with the R$^{\mathrm{R}}$-tree and two competing proposals. The settings for the experiments are described first. This is followed by the presentation of results.

5.1 Experimental Setup

Three indexing techniques are covered in the experiments: the R$^{\mathrm{R}}$-tree, the RUM-tree [23], and the R*-tree. The XXL library [4] implementation of the R*-tree was used. The R$^{\mathrm{R}}$-tree was also implemented on top of the XXL library [4]. The RUM-tree implementation was kindly provided to us by its inventors [23]. We modified the implementation to add an LRU buffer.

Unless stated otherwise, a 4 kilobyte page size (and thus tree node size) is used both for disk trees and the buffer tree of the R$^{\mathrm{R}}$-tree. Also when not stated otherwise, the size of the operation buffer in the R$^{\mathrm{R}}$-tree is chosen so that, on average, it is able to store the number of operations that is equal to the 5% of the number of tracked objects. For most of the experiments with the R$^{\mathrm{R}}$-tree, no disk-page buffer is used. When a page buffer is used, the LRU write-back page replacement policy is applied so that dirty pages are written to disk only when they are evicted from the buffer.

To generate a workload, the generator from the COST benchmark [13] is used. This generator simulates a number of moving objects that update their positions using the shared-prediction based update approach. More specifically, point-based tracking is used, meaning that an object updates its position when its current position reaches a distance to its most recently reported position that is equal to an agree-upon accuracy threshold thr. The objects' positions are inserted into the index as bounding squares of the circles with radius thr.

We use two kinds of workloads generated by the COST generator. In the uniform workloads, an object starts from a random position and moves in a random direction and at a random speed between 0 and 180 km/h in-between updates. In the non-uniform workloads, simulated objects move in a road network that is a complete graph connecting 20 randomly placed intersections. Three classes of objects are generated with maximum speeds of 45, 90, and 180 km/h.

Each of the generated workloads is a mix of insertions, deletions, and queries. First, all objects are inserted. Then, they send updates (deletion-insertion pairs) according to the update approach described above. The I/O performance is measured also only after the index reaches steady state, i.e., all objects are inserted.

Range queries are distributed uniformly in the workload and in the data space. Each query range is a rectangle covering 0.02% of the data space. Table 2 shows the default values of the workload parameters used for all of the experiments (unless the value of the parameter is varied in the experiments). Note that in this table

Parameter	Default Value
Number of moving objects	$100,000$
Size of the dataspace	100km $\times$ 100km
Accuracy threshold	200m
Number of updates	$400,000$
Queries per update	$1:20,000$

Table 2: Workload Parameters

(and later in the graphs) an update denotes an individual deletion or insertion, not counting the initial insertions. Thus, when the ratio of the number of queries to the number of updates is equal to $1/20,000$, this means that one query is performed for every $10,000$ deletion-insertion pairs.

We have chosen to use $400,000$ operations in the workloads because the use of workloads with larger numbers of operations does not significantly change the performance numbers. A performance graph later in this section demonstrates this.

5.2 Cost Model Validation

The first set of experiments were run with the goal of validating the conclusions of the cost model presented in Section 4, namely, that emptying only the largest group of operations is the most efficient strategy. For this purpose, we compared this strategy with the strategy where threshold k is set to some static value (static thresholding). Uniform and non-uniform workloads were run with different static values of the operation threshold k and two settings for the operation buffer size: 5% and 20% of the number of moving objects. Note that when $k = 1$, the buffer is emptied completely. Also when, for a specific buffer emptying, $k > k_{max}$ the algorithm empties just the largest group of operations. Figure 4 plots the update performance of the $\mathrm{R^R}$-tree under different buffer-emptying strategies. The position of the point for the largest-group emptying on the x-axis shows the average size of the largest group of operations. The graph demonstrates that emptying only the largest group is more efficient than using any static threshold value.

Figure 5 plots the average size of the largest group of operations when the size of the buffer is varied. It also shows the smallest value of this parameter when the cost model assumptions still hold (see Equation 6). The graph demonstrates that the cost model applies when the buffer is larger than 2% of the number of objects. This applies for both uniform and non-uniform workloads. Note, of course, that this percentage depends on the average fan-out of the R-tree used in the experiments (S), as can be seen from Equation 7.

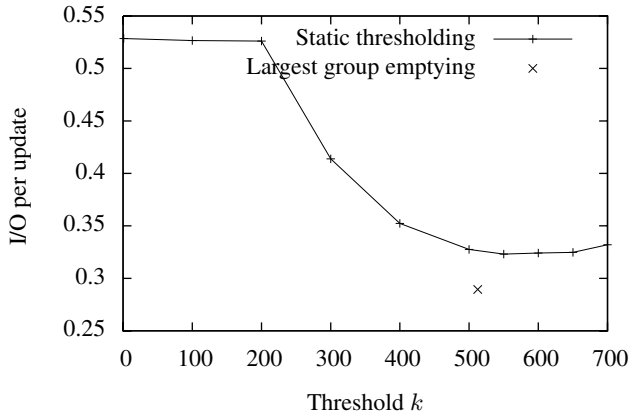

Figure 4: Largest group emptying vs. static thresholding, 5% buffer, non-uniform workload

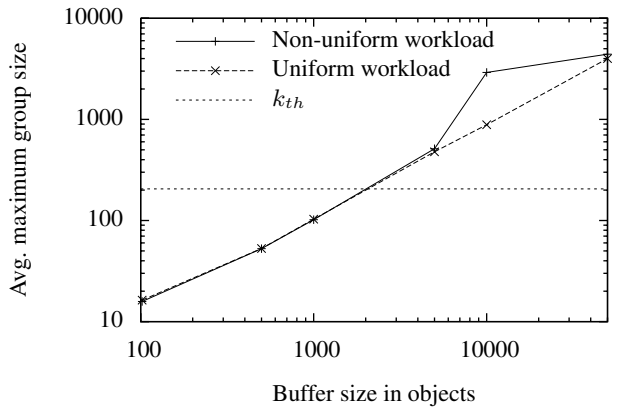

Figure 5: Average largest buffer group size relation to the buffer size, non-uniform dataset

5.3 Exploring the $\mathrm{R^R}$-Tree Algorithms

For the rest of the experiments, we use non-uniform workloads. First, we explore selected properties of the $\mathrm{R^R}$-tree. The goal of the first experiment is to determine how effective the operation buffer is at increasing the update performance in comparison to the use of an LRU write-back page buffer. In this experiment, part of the available memory is allocated to the operation buffer, and the rest is allocated to the LRU page buffer. The total amount of main memory is kept constant, but the fraction allocated to the operation buffer is changed. The experiment is run with two workloads, one with few queries and one with an equal amount of queries and object updates. The I/O performance is averaged over all three types of operations: insertions, deletions, and queries.

Figure 6 shows very clearly that when most of the operations are updates, all of the available main memory should be allocated to the operation buffer. On the other hand, reducing the size of the page buffer increases the cost of queries. Thus, when the workload contains a lot of queries, part of the memory should be allocated to the operation buffer and part to the page buffer.

The effect of query piggybacking (as described in Section 3.8) was studied for operation buffer sizes of 5% and 20% of the number of moving objects. The experiments show that piggybacking starts to have a noticeable effect when there is a substantial fraction of queries in the workload. For example, for the 5% buffer

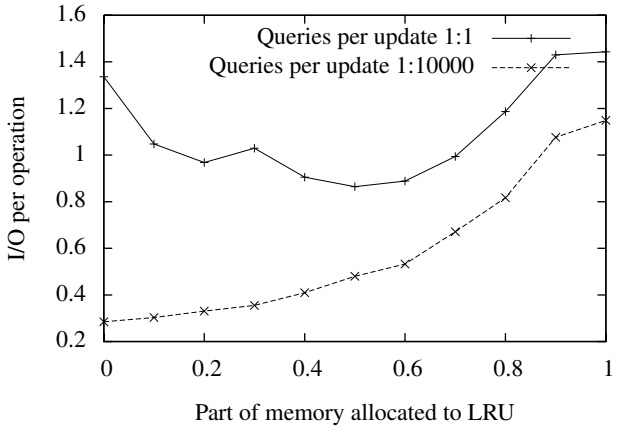

Figure 6: I/O performance for varying main-memory divisions

size and 200 updates per query in the workload, only about 4.9% of updates are performed by the PIGGYBACK algorithm during the processing of queries. When the number of updates per query drops to 20, about half of the updates are performed by the PIGGYBACK algorithm.

5.4 Comparing the Indexes

To ensure that the comparison of the three indexes fair and thus obtain meaningful results, in each of the experiments, we allocate exactly the same amount of main memory to each index. All the available main memory is used for the page buffer for the R*-tree.

The $\mathrm{R^R}$-tree uses all of the available memory for the operation buffer in most experiments. The exception is that in the experiments with high ratios of queries, the $\mathrm{R^R}$-tree divides memory between the operation buffer and the page buffer. The division used in each case was chosen through a number of performance experiments. The RUM-tree uses a part of the available main memory

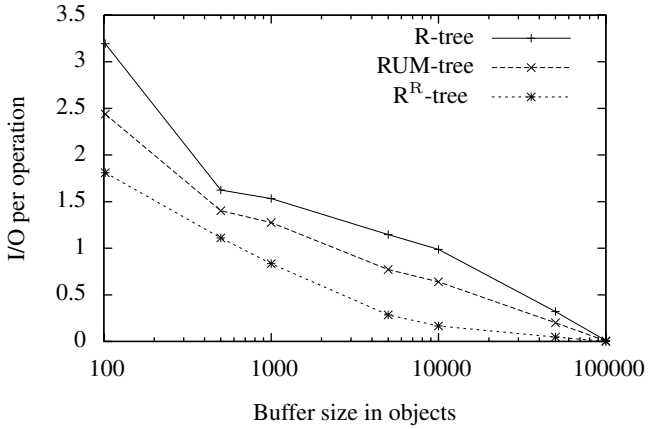

Figure 7: Scalability with respect to available main memory

for its update memo, and the rest is used for the page buffer. Two parameters, the inspection ratio and the clean-on-touch flag [23], are used to control how eagerly the obsolete entries are removed from the RUM-tree. Experiments with three different configurations of the RUM-tree were run to determine which one performs the best: 20% inspection ratio with clean-on-touch, 0% inspection ratio with clean-on-touch, and 0% inspection ratio without clean-on-touch. The configuration with 20% inspection ratio and clean-on-touch was chosen. This result is in correspondence with the findings of the inventors of the RUM-tree.

Figure 7 shows how the indexes compare for different sizes of available main memory. Note that the amount of main memory is given in terms of the average number of objects that can fit in the memory organized into an R-tree. The amount of main memory is varied from 0.1% to 100% of the number of moving objects. The $\mathrm{R^R}$-tree clearly outperforms the other two approaches. For example, when the amount of main-memory is equal to 10% of the number of moving objects, the update performance of the $\mathrm{R^R}$-tree is almost four times better than that of the RUM-tree and more than seven times better than that of the R$*$-tree.

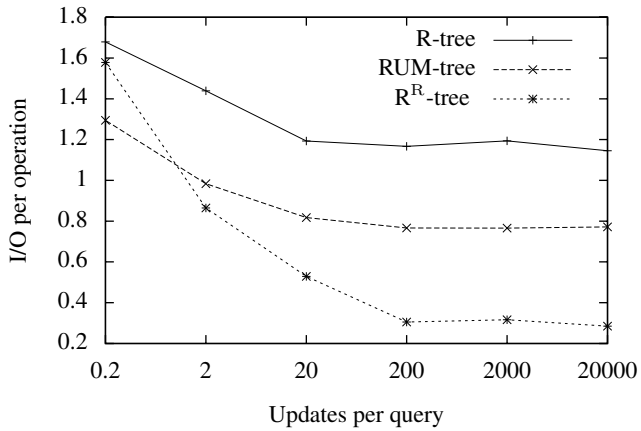

Figure 8: I/O performance versus the updates/query ratio

Figure 8 shows how the different approaches perform when the ratio of updates and queries is changed. Again the $\mathrm{R^R}$-tree is better

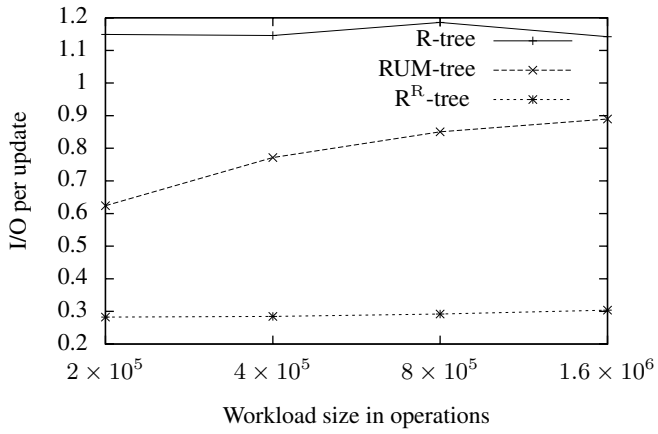

Figure 9: Scalability with respect to workload duration

under all settings except when there are more queries than updates. In this setting, where the query performance is the dominant factor, the RUM-tree performs better than the $\mathrm{R^R}$-tree, because the $\mathrm{R^R}$-tree uses a smaller page buffer than the RUM-tree.

The next two sets of experiments explore the performance of the indexes under different workload sizes and different disk page sizes. The graphs in Figure 9 show that the performance of the $\mathrm{R^R}$-tree tree does not degrade with an increasing length of the workloads. The graphs in Figure 10 demonstrate that, as expected, the I/O performance is improved when the disk page size is increased.

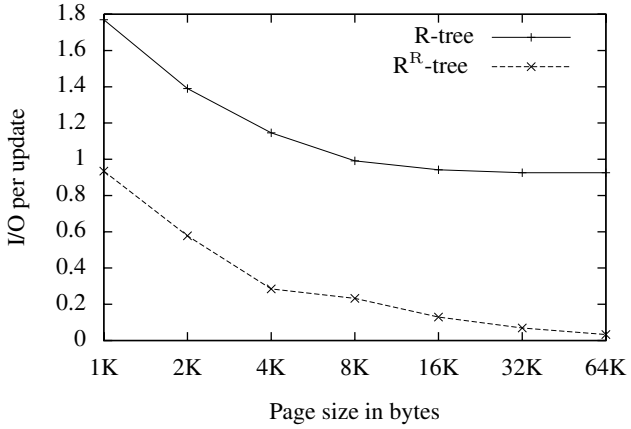

Figure 10: Scalability with respect to disk page size

6. RELATED WORK

We proceed to consider relevant previous work on the efficient update of general disk-based data structures and, particularly, on efficient updates in R-trees.

Techniques for the bulk-loading of data structures are relevant for the problem of efficient updates. Notably, Choubey et al. [7] have proposed algorithms that aim to perform many insertions on an existing index structure in one go. However, these algorithms are not directly applicable in our setting because they focus on the search performance of the resulting structure, whereas the performance of the bulk-loading itself is of secondary importance. In addition, they do not consider deletions.

An approach to dealing with frequent updates of continuous variables such as positions is to model the positions as linear functions of time instead of the standard constant functions of time. When representing the position of an object by a linear function as, e.g., proposed by Wolfson et al. [22], the modeled position stays close to the actual for longer periods of time. It has been shown that, for a range of reasonable accuracy thresholds, the number of updates needed to maintain the positions of vehicles is reduced by almost a factor of 3 [8]. The use of such linear functions on the top of R-tree was first proposed by Šaltenis et al. [19] and subsequently explored by others. These proposals are orthogonal to our proposal, and it is possible to combine them to obtain the combined benefits.

In a survey, Graefe [9] presents a general overview of techniques for speeding up B-tree updates. One of the techniques covered—the buffering of insertions in a separate data structure—is used in our proposal. The survey also describes a differential file design that enables the buffering of deletions and updates in addition to insertions. Differential indexes and on-line index construction were researched in the context of B-trees and relaxed index persistence [17, 20]. These approaches are similar to the buffer of the R^R-tree that buffers all incoming updates (i.e., deletions and insertions). To the best of our knowledge, partial and selective emptying of the operation buffer was not explored in these works. Graefe [9] also mentions a non-logged B-tree, which directly corresponds to our notion of relaxed persistence or use as a secondary index. Finally, in recent work, Graefe [10] briefly discusses how so-called merged B-trees can be used to conveniently store a main index and to-do lists that contain unprocessed keys.

The problem of frequent updates in R-tree was recently tackled by Lee et al. [14], who present the most competitive proposal for bottom-up updates of R-trees. The bottom-up approach avoids potentially expensive top-down traversals and exploits the update locality of continuous variables. However, an auxiliary data structure, required in order to access the bottom level, uses a significant amount of space and is disk based. Because of this, updates still take at least 3 I/Os. Additionally, the proposal includes a main-memory data structure that compactly stores a summary of the tree. This structure helps save I/O in the case of non-local updates, but uses a fixed amount of main memory. In contrast, our proposed technique is top-down and mitigates the drawbacks of top-down traversals by performing operations in batches. Thus, there is no need for direct leaf-level access, and our proposal uses all available main memory.

Arge has proposed a general tree buffering technique that associates a buffer with every non-leaf node [1]. Operations are then not performed immediately, but are placed in the buffers. Once a buffer gets full, its contents are moved to the buffers at the next level in its subtree. Main memory is used for the buffer emptying. Arge et al. have also applied the technique to R-trees [2], and Van den Bercken and Seeger [5] have explored similar techniques. A drawback of all these techniques is that they cannot answer queries immediately. Either queries are buffered together with the insertions and deletions or all the buffers have to be emptied before processing a query. In both cases, the resulting query latencies may not be acceptable for on-line applications. Our proposal shares the idea of buffering and performing updates lazily and in batches. A key difference is that, in order to support efficient, instantaneous queries, the R^R-tree does not use disk-based buffers for the intermediate nodes—it uses only a single main-memory buffer.

Lin and Su combine the buffer-tree and bottom-up techniques [15]. They attach disk-based insertion buffers to intermediate nodes and perform insertions top-down in batches. Similarly to the R^R-tree, partial buffer emptying is employed to increase sharing of I/O. To avoid multiple partial traversals for a single deletion, deletions are gathered in a main-memory deletion buffer that is applied lazily and directly to the leaf-level nodes. A main-memory based leaf-level access table facilitates this. The major drawback of this approach is that the size of the main-memory based leaf-level access table is of the same order of magnitude as the size of the database. For example, for R-trees, the size of this access table is one third of the size of the index.

A recent approach by Xiong and Aref [23] significantly lowers the average update cost by performing deletions in main memory. However, insertions are performed using the ordinary R-tree algorithm, so there are no further performance gains for these. A main-memory data structure, called update memo, is responsible for keeping information about deletions as well as about the latest and obsolete data entries. This approach is similar to ours, in that deletions are performed in main memory.

The LUGrid approach [24] is the only related approach that indexes spatial data and makes the same persistence assumptions we do. The approach uses a disk-based grid file, a main-memory memo structure for keeping track of deletions, and a main-memory grid for storing unprocessed insertions. Similar to our proposal, the LUGrid flushes the insertions from a main-memory grid cell when it becomes full. In contrast to our proposal, the LUGrid cannot store objects with extents, the circular object-positions of moving objects cannot be indexed.

While the related approaches described above strive to solve problems similar to ours, with the exception of the B-tree approaches and the LUGrid, they cannot be directly compared to our proposal because of the different persistence assumptions. Those approaches assume a persistent index; thus, an update requires always at least one I/O. In contrast to this, the R^R-tree does not have to write ev-

ery incoming operation to disk. As a result, better performance than one I/O per update can be achieved.

7. CONCLUSION AND RESEARCH DIRECTIONS

Motivated by emerging database applications that involve the monitoring of large collections of continuous variables and thus are characterized by high rates of updates, this paper presents a novel data structure, called the R^R-tree, that supports updates more efficiently than existing proposals while also supporting queries. In contrast to related work, this data structure is capable of efficiently using any amount of main memory, and it supports the same settings and operations as an ordinary R-tree. Another core difference to the related work is a relaxed persistence assumption: we argue that relaxed persistence of an index is appropriate in a setting with hyper-dynamic data.

The R^R-tree builds on two main ideas. First, *operation buffering* in main memory enables execution of the majority of update operations quickly, and it allows some updates to annihilate one another, thus avoiding any I/O at all. Second, by *grouping* operations on buffer emptying, the index enables all operations that travel to the same node of the tree to share I/O.

A strong point of this approach is its orthogonality to the tree-like data structure being buffered. This makes it possible to adapt the approach to other types of trees. This yields a quite general contribution. Furthermore, this approach is orthogonal to other means of exploiting the available main memory. This enables adaptation of the R^R-tree to different workloads, by combining the operation buffer with some page-cache buffer, e.g., an LRU cache.

The empirical performance study includes favorable comparisons with the conventional LRU-cached R-tree and with the state-of-the-art RUM-tree. The study also offers general insight into the benefits of LRU caching in the paper's setting.

Several interesting directions for future research exist. We expect it to be very interesting to apply the proposed techniques to other indexes in the class of grow-post trees, e.g., the TPR-tree. Next, we note that this paper's focus has been on I/O efficiency. However, the R^R-tree does use an in-memory R-tree to efficiently support querying. Thus, the proposal should be competitive with respect to CPU performance. Still, studies of CPU performance and the investigation of techniques that reduce the CPU cost are in order. Finally, while we mention that a simple approach to recover from a main-memory loss is to wait for all the objects to report their positions, other, more advanced, log-based recovery techniques may be invented.

Acknowledgments

The authors would like to thank Bernhard Seeger for helpful discussions on this paper's subject. This work was supported by a grant from the Nykredit Corporation.

8. REFERENCES

[1] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proc. WADS*, Volume 955 of *Lecture Notes in Computer Science*, pp. 334–345. Springer Verlag, 1995.

[2] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1): 104–128, 2002.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD*, pp. 322–331, 1990.

[4] J. Van den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL—a library approach to supporting efficient implementations of advanced database queries. In *Proc. SSD*, pp. 91–108, 1999.

[5] J. Van den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. VLDB*, pp. 461–470, 2001.

[6] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis. Revisiting R-tree construction principles. In *Proc. ADBIS*, pp. 149–162, 2002.

[7] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A generalized R-tree bulk-insertion strategy. In *Proc. SSD*, pp. 91–108, 1999.

[8] A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE TKDE*, 17(5): 698–712, 2005.

[9] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1): 39–44, 2006.

[10] G. Graefe. Master-detail clustering using merged B-tree indexes. Manuscript, 19 pages, 2007.

[11] A. Guttman. *R*-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[12] C. S. Jensen, D. Tiešytė, and N. Tradišauskas. Robust B^+-tree-based indexing of moving objects. In *Proc. MDM*, p. 12, 2006 (e-proceedings).

[13] C. S. Jensen, D. Tiešytė, and N. Tradišauskas. The COST benchmark-comparison and evaluation of spatio-temporal indexes. In *Proc. DASFAA* pp. 125–140, 2006.

[14] M.-L. Lee, W.Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: a bottom-up approach. In *Proc. VLDB*, pp. 608–619, 2003.

[15] B. Lin and J. Su. Handling frequent updates of moving objects. In *Proc. CIKM*, pp. 493–500, 2005.

[16] D. B. Lomet. Grow and post index trees: roles, techniques and future potential. In *Proc. SSD*, pp. 183–206, 1991.

[17] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Concurrency control in B-trees with batch updates. *IEEE TKDE*, 8(6):975-984, 1996.

[18] M. Raab and A. Steger. "Balls into bins"—a simple and tight analysis. In *Proc. RANDOM*, pp. 159–170, 1998.

[19] S. Šaltenis, C. S. Jensen, S T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc ACM SIGMOD*, pp. 331–342, 2000.

[20] V. Srinivasan and M. J. Carey. Performance of on-line index construction algorithms. In *Proc. EDBT*, pp. 293–309, 1992.

[21] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases* 7(3): 257–387, 1999.

[22] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Proc. SSDBM*, pp. 111–122, 1998.

[23] X. Xiong and W. G. Aref. R-trees with update memos. In *Proc. ICDE*, p. 22, 2006 (e-proceedings).

[24] X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: update-tolerant grid-based indexing for moving objects. In *Proc. MDM*, p. 13, 2006 (e-proceedings).