GeoInformatica 7:4, 337–354, 2003

Performance Evaluation of Lazy Deletion Methods in R-trees

ALEXANDROS NANOPOULOS,[1] MICHAEL VASSILAKOPOULOS,[2] AND YANNIS MANOLOPOULOS[1]
[1]Department of Informatics, Aristotle University of Thessaloniki, 541 24 Thessaloniki, Greece
E-mail: {alex,manolopo}@delab.csd.auth.gr
[2]Department of Informatics, Technological Educational Institute of Thessaloniki, P.O. Box 14561, 541 01 Thessaloniki, Greece
E-mail: vasilako@it.teithe.gr

Received September 20, 2002; Revised May 15, 2003; Accepted June 12, 2003

Abstract

Motivated by the way R-trees are implemented in commercial databases systems, in this paper we examine several deletion techniques for R-trees. In particular, in commercial systems R-tree entries are mapped onto relational tables, which implement their own concurrency protocols on top of existing table-level concurrency mechanisms. In analogy, the actual industrial implementations of B-trees do not apply the well-known merging procedure from textbooks in case of node underflows, but rather they apply the free-at-empty technique. This way, space is sacrificed for the benefit of faster deletions and less locking operations, whereas the search performance practically remains unaffected. In this context, we examine the efficiency of modifications to the original R-tree deletion algorithm, which relax certain constraints of this algorithm and perform a controlled reorganization procedure according to a specified criterion. We present the modified algorithms and experimental results about the impact of these modifications on the tree quality, the execution time for the deletion operation and the processing time of search queries, considering several parameters. The experimental results indicate that the modified algorithms improve the efficiency of the deletion operation, while they do not affect the quality of the R-tree and its performance with respect to search operations.

Keywords: spatial databases, R-tree, lazy deletion, performance evaluation

1. Introduction

Spatial databases are used for the representation, storage and processing of spatial objects like points, lines or polygons in one, two or more dimensions. Spatial database systems (SDBSs) find applications in geographical information systems (GIS), CAD or multimedia databases, and VLSI design. Common tasks for which SDBSs are useful include urban planning, resource management and geomarketing. SDBSs contain several types of queries. Among the most common ones are the range query (find all spatial objects within a given region), join query (find pairs of spatial objects that satisfy a given predicate) and nearest-neighbor query (find the point that is closest to a given point).

Nowadays, the R-tree [4] is one of the most common spatial access methods (SAMs) and has been implemented in several commercial databases (e.g., Oracle, Informix). It is a height-balanced tree and can be considered as an extension of the B^+-tree for

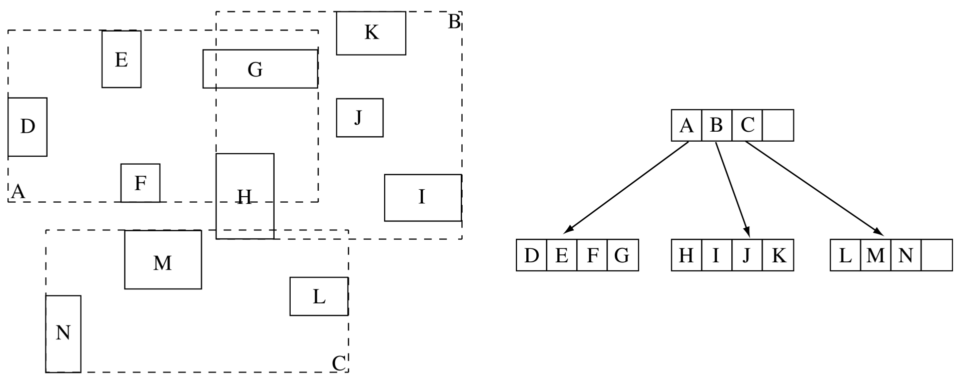


Figure 1. An example of an R-tree.

multidimensional data. Each entry of an internal node contains a pointer to a subtree and the minimum bounding rectangle (MBR) of this subtree (the MBR that encloses all the MBRs appearing in the subtree). The leaf entries contain pointers to the actual spatial objects (examined at the refinement step). Each R-tree node contains at least *m* and at most *M* entries. Hence, minimum utilization can be specified. Unlike a B^+-tree, several paths from the root to the leaves may be examined during a search operation in an R-tree. Figure 1 depicts an example of an R-tree with *M* equal to four.

Several R-tree variations have been proposed during the previous years. A complete survey can be found in Gaede and Guenther [5]. The R^+-tree [18] guarantees that exactly one path from the root to a leaf will be followed during a search, under the cost of redundancy. The R^*-tree [2] uses an enhanced split algorithm and the technique of forced re-insertion. The Hilbert R-tree [8] stores the Hilbert values at the leaf level and ranges of those values at the upper levels, to facilitate search operations. In a recent effort, the structure of LR-trees [3] proposes an efficient decomposable R-tree variation, which comprises of a number of smaller R-trees in a manner similar to binomial queues.

1.1. Motivation

The R-tree and several of its variations (especially the R^*-tree) have been implemented in prototype database systems. However, in implementations of commercial systems [6], [9], [16], the R-tree is built on top of existing infrastructure suited for relational data, i.e., relational tables. Therefore, certain modifications have been carried out, dictated by the specific context of these implementations. This approach can impact index-update operations. One reason is that it implements its own concurrency protocols on top of existing, table-level concurrency mechanisms [9].

What is required, to overcome the aforementioned problem, is the development of alternative update methods, which will lead to improved execution times. For instance, in the paradigm of B-tree indexes, their actual industrial implementations do not apply the well-known merging procedure from textbooks in case of node underflows, but rather they apply the free-at-empty technique. According to the latter technique, underflowed nodes are left without performing any merging until they eventually contain no entries, in which case they are returned to the garbage collector (except for the root). This way, space is sacrificed for the benefit of faster operations. The most important reason for coming up with faster operations is that more efficient locking schemes can be used. Thus, both deletions and searches are performed more efficiently.

As mentioned and induced from Ravada and Sharma [16], in this context the reinsertion procedure may not comprise a wishful solution, due to the concurrency control operations that are required in commercial database systems. Thus, here we present in detail some variations of the original R-tree deletion algorithm [4] having in mind this specific context, i.e., the actual implementation in commercial database systems. In this context, we examine the efficiency of modifications to the original R-tree deletion algorithm, which relax certain constraints of this algorithm and perform a controlled reorganization procedure according to a specified criterion.

The importance of the deletion operation in spatial databases evidently depends on the particular application. For instance, if the indexed objects correspond to landscape entities (e.g., lakes), then deletions are uncommon. However, there exist several application domains where spatial deletions occur more frequently. Object-relational DBMSs (ORDBMSs) comprise one such emerging domain. ORDBMSs provide an extensible architecture, which may involve tables with both relational and spatial attributes, where the latter ones can be indexed with R-trees [6], [9]. In this case, deletions from a spatial index can be the result of deleting records from a table according to its relational (i.e., non spatial) attributes. Apparently, such deletions can occur quite frequently. Assume, for instance, a large corporation that hires persons to work as agents with limited-time contracts. Also, assume that the agents are geographically distributed around the country. The corporate's database can contain a table with the IDs of the agents, whereas their spatial location (so as to be able to identify those that are, e.g., nearest to a given location) could be indexed with an R-tree. When an agent stops working for the corporate, s/he has to be deleted from the table. To keep the database consistent, the corresponding entry has to be removed from the R-tree as well. As mentioned, this is an example where deletion is not performed according to the spatial attribute (the deletion is done with the agent's ID), but it leads to a deletion from the spatial index. Nevertheless, one can also find several applications where deletions yield directly from spatial attributes. For instance, in the previous setting, the corporate may want to close a specific branches, thus a deletion can be done for all agents within a region of, say, 100 miles from the location of the branch.

Moreover, deletions can result from update operations to existing entries. In general, the update of the primary-key for an existing entry is handled with a combination of a deletion and an insertion (i.e., delete the entry with the old key and insert an entry with the new key). For spatial indexes, this corresponds to an update of the spatial coordinates of an existing entry, a case that may occur frequently in several applications. Therefore, the performance of the deletion operation can also affect the performance of the update operation. For the case of updating existing entries in B-tree structures through a combination of a deletion and an insertion, it has been found that the free-at-empty deletion method leads to low space utilization (39%) [7]. Therefore, in the case of spatial indexes, it worths examining alternatives to the free-at-empty deletion method.

1.2. Contribution and paper organization

The main contributions of this paper are the detailed examination of alternative deletion algorithms and related reorganization procedures plus a thorough experimental study, which assesses the impact of these approaches on the tree quality, the execution time for the deletion operation and the processing time of search queries. The experimental results indicate that the presented alternatives (i) improve the efficiency of the deletion operation, (ii) do not affect the quality of the R-tree and its performance with respect to search operations, for a mixed workload that contains a significant portion of deletion operations, whereas (iii) the total required time is reduced significantly.

The rest of this paper is organized as follows. Section 2 presents related work and Section 3 describes the alternative deletion algorithms and their complete algorithmic description. The experimental results are reported in Section 4. Finally, Section 5 draws the conclusions and future work.

2. Background and related work

2.1. Lazy and deferred methods

Several papers in the literature describe dynamic data structures (such as variations of B-trees and dynamic hashing schemes), which are maintained by ''lazy'' algorithms. Similarly, another used term is ''deferred''. Both these expressions mean that during the execution of an operation, a specific constraint is relaxed and a certain action is postponed for the future.

For example, imagine that after an insertion and an overflow, we do not split the leaf and recursively all the nodes along the path to the root. This way, we hope that a subsequent deletion of an entry residing in the same leaf will lead the structure to the previous situation and absorb locally the side-effects without affecting the nodes along the path to the root. The opposite strategy could be followed in case of a deletion, i.e., no merge after an underflow, in anticipation of a future insertion.

The question is what should be done algorithmically instead of a node split or a merge of two nodes. Under a lazy approach, in the case of insertion-overflow, it has been proposed to apply other techniques, such as the use of elastic buckets, chaining or lazy parent split [1], [11]–[13], [17], [19], [21]. In the context of the present paper where we examine the case of deletions-underflows, we notice that, in general, three methods may be followed to handle underflows [10]: merge-at-half, reinsertion and free-at-empty. The avoidance of dynamic underflow handling excludes the use of re-insertion. Therefore, during a deletion operation, only one path, i.e., from the root to the leaf that contains the deleted entry, will be active. In the sequel, we examine such lazy approaches for deletions in R-trees.

```
Algorithm Delete (R-tree node $Root$, entry $e$)
Find the leaf that includes $e$;
Remove $e$ from the leaf;
IF the leaf underflows THEN
    Remove from the parent node the pointer to the leaf;
    Adjust the upper levels up to the root;
    Reinsert the orphaned entries;
```

Figure 2. Original R-tree deletion algorithm.

2.2. R-tree deletion

According to the original deletion algorithm [4], to delete an entry from an R-tree, the tree is traversed, the appropriate leaf is located, and the specific entry is removed. Then, in case the leaf remains with no less than m entries (the minimum allowed number of entries), the leaf MBR is calculated and if it should be modified, then all the necessary modifications are propagated up to the root. Alternatively, if the leaf underflows after the entry deletion (i.e., contains less than m entries), then all its entries, called orphaned, are removed and the leaf is deleted. In the sequel, the deletion procedure is applied recursively in the parent node, in order to delete the corresponding entry of the leaf. Thus, there is a case where the parent node (lying one level up) may also become underflowed. In this case, the previous procedure is applied, too. When the recursion reaches the root, then all orphaned entries, from all levels, are re-inserted. The R-tree deletion algorithm is illustrated in figure 2.

Therefore, a node underflow is handled as soon as it occurs and is treated dynamically by deleting the node and reinserting its remaining entries. This guarantees the minimum utilization (m) in each node but may require significant total execution time, since the reinsertion is a high cost procedure. Hence, if a dataset is rather dynamic and deletions occur quite frequently, then a significant portion of the total time for the R-tree maintenance will be required for reinsertion. Moreover, the overall performance depends on the type of workload. If the number of deletion queries is significant with respect to the overall number of queries, then the total completion time may be impacted significantly.

2.3. Commercial implementations

The R-tree has been implemented in commercial database systems (Oracle, Informix, Illustra, Post gress). In several cases the implementation is carried out on the top of relational tables [6], [9]. This is done by adopting an extensible indexing framework, which is illustrated in figure 3.[1] Data are stored in base tables, e.g., in an object-relational table as shown in the figure. The spatial column of this table contains the spatial attributes, from which MBRs are extracted and stored in the R-tree. The nodes of the R-tree correspond to tuples of an index table (i.e., a relation dedicated for the R-tree), and their entries are organized within these tuples. In addition, metadata for the R-tree (e.g., root address, number of nodes, etc.) are stored as a row in a separate metadata table.

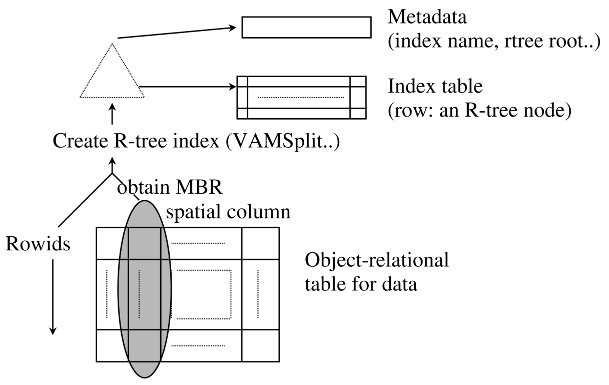


Figure 3. R-tree storage within relational tables.

In prototype research systems, efficient R-tree variations are usually implemented (the R*-tree in particular). However, the re-insertion technique is not used neither during insertion, nor during deletion [16]. Although re-insertion [2] is an enhancement over the original insertion algorithm and can be considered as optional, the same does not apply for the deletion algorithm because the underflow is handled only with re-insertion [4]. Thus, the need again arises for alternative deletion algorithms.

It has been proposed that tree nodes should only be deleted in the course of a reorganize operation.[2] Following such a policy, underflow is not handled dynamically. Instead, when a node underflows, it continues to remain underflowed. Periodically, a criterion is tested and if it is fulfilled, a global reorganization is applied that eliminates all underflowed nodes and re-inserts all the orphan entries.

Besides the issues regarding the implementation over existing infrastructure, the representation of R-tree nodes as tuples within relations and the avoidance of dynamic reorganization has the advantage of simplest concurrency control, which is an important aspect in all commercial database systems. Effective concurrency control techniques have been developed for traditional indexes, like the B$^{\text{link}}$-tree [15]. Similar approaches have also been proposed for the R-tree, like the R$^{\text{link}}$-tree [14], but have not been implemented yet in commercial or prototype systems. The difference from traditional indexes is that during deletion (and insertion) several paths may be concurrently active, due to reinsertion. For more information on concurrency issues in access methods, see Chapter 12 of Manolopoulos et al. [10].

Also, avoiding dynamic reorganization can significantly simplify the procedure of recovery. For instance, in the Informix database, the R-tree secondary access method creates its own logical records (logging) so as to recover from deletion operations from the leaf nodes [6] (for deletions from internal nodes an extensible log manager is used). Evidently, the design of a roll-back operation is much more complex when taking into account the reinsertion of entries for almost every deletion, since this can yield to roll-backs from numerous insertions.

Finally, consider the case of deletions that affect many rows. For example, assume that we delete a large number of records from a table that contains a spatial attribute indexed with an R-tree, whereas the deletion is performed according to a non-spatial attribute (i.e., the where-clause of the deletion query does not involve the indexed spatial attribute). Therefore, the corresponding entries have to be removed from the R-tree as well. In commercial R-tree implementations (e.g., the R-tree secondary access method in Informix DBMS), this case is handled by first searching each entry that will be deleted (the searching values are the ones of the spatial attributes of the deleted rows), since there is no other way to find them [6]. Hence, according to Informix Corp. [6], ‘‘a delete affecting many rows may execute slowly due to the presence of an R-tree’’. If to this cost we add the cost of dynamic reinsertions, then in some cases this operation may be rendered infeasible. In contrast, by avoiding dynamic reinsertions, the cost of deleting multiple rows can be kept at a logical level.

3. Lazy R-tree deletion methods

When a node underflows, the original R-tree deletion algorithm [4] forces the reinsertion of its remaining entries. Underflow is determined by the minimum allowed number of entries, *m*, which for the R*-tree is set to 40% of the maximum node capacity [2]. As described, the reinsertion operation is costly and does not permit simple locking schemes to be developed. To overcome the aforementioned problems, lazy deletion methods can be applied. Their objective is to defer the costly operation of reinsertion, without degrading performance.

Two general categories can be considered for lazy deletion methods:

Local. The methods in this category are restricted in each single node, trying to postpone the reinsertion of its contents, when it underflows.

Global. The methods in this category have a global perspective, maintaining statistics for the tree nodes, upon which they postpone the reinsertion of the contents of all underflowed nodes.

The above categorization is general. According to the particular approaches that can be followed for each category, different methods can be developed. In the following, we focus on developing such specific methods for each category.

3.1. Local methods

The most direct way of deferring the reinsertion for a node with less than *m* entries, is to allow it to have a number of entries, which is less than the prespecified value *m*. Therefore, for the case of R*-tree for which *m* is 40% of the maximum capacity, the lower limit could be set, e.g., to 30%, 20%, or even less.

For the special case where the lower limit is 0%, the operation of reinsertion is never applied. Instead, a node is allowed to become empty, and after this, it is removed by deleting the respective entry from the parent node. It has to be noticed that the latter procedure may need to be applied recursively, up to the root. This scheme has been also applied to other structures, like the B-tree, and is called free-at-empty technique [7] (henceforth, this technique is denoted as FE).

Evidently, by lowering the minimum allowed number of node contents, the average node utilization is expected to decrease. This corresponds to an increase of space cost. Also, the query performance can be influenced for the same reason. However, such local methods opt to a small degradation of query performance and a significant improvement of the deletion cost, due to the deferring of the costly reinsertion operation. The aforementioned issues are examined experimentally in the following.

Besides the minimum allowed number of node entries, other criteria may be followed as well. For instance, the reinsertion of an underflowed node's entries can be postponed until the dead space within the node is larger than a given threshold. We have developed several such criteria, but their experimental evaluation did not show significant difference compared to the criterion of lowering the minimum number of entries. However, the main drawback of these criteria is that they cannot be tuned easily. Therefore, in the following, we do not consider any further instantiations of lazy deletion methods besides that of lowering the minimum allowed number of node entries.

3.2. Global methods

With global lazy deletion methods, unlike the original and local methods, underflowed nodes are not treated independently. Instead, underflowed nodes (with less than m entries, or possibly empty) remain intact in the tree. The contents of these nodes are collected and reinserted altogether, during a global reorganization operation (the empty nodes are simply removed from the tree).

The reorganization procedure is required to maintain the quality of the tree, since the existence of underflowed nodes affects space overhead and query performance. Clearly, there is a tradeoff between the frequency with which the reorganization procedure is invoked (due to its significant time overhead) and the tree quality, which may decrease substantially if reorganization is not performed. Therefore, what is required is criteria to determine if (and when) reorganization should be applied. Such criteria are discussed in the following.

Regarding the locking of nodes during the deletion of an entry, first the leaf that contains the entry to be deleted, is located. At this step no locking is required. The entry is removed from the leaf and the deletion propagates up the tree. Each time, the nodes are locked for the update. To simplify further the locking scheme, a simple policy can be followed: when possible, the corresponding MBR of ancestor nodes are not updated. Therefore, fewer node updates are required, thus fewer node lockings are applied. Additionally, this policy is expected to further reduce the cost of the deletion operation, without significantly impacting query performance. The aforementioned policy is examined experimentally in the sequel.

3.2.1. Criteria for reorganization. A specified criterion should point out when reorganization has to be applied. This decision has to be based on measures of tree quality. Since we want the reorganization to be bound with the deletion operation, we examine the reorganization criterion only after each deletion. However, the statistics that the criterion will be based on are maintained and updated from other operations also (e.g., insertion).

In general, the tree quality can be measured according to space and time overhead that the underflowed (or empty) nodes produce. For instance, reorganization can be applied when:

- A substantial fraction of tree nodes are underflowed.
- A substantial fraction of tree nodes are ineffective. This is due to the fact that large MBRs may be stored for nodes that contain few, scattered rectangles. Thus, the selectivity of the tree is reduced significantly.

In the following, we focus on the first option, which we call global-reorganization deletion method (henceforth, this method is denoted as GL). Therefore, a counter of the underflowed nodes should be maintained. During a deletion, several nodes may become underflowed, thus the counter should be increased by their number. Also, an insertion may turn several nodes to stop being underflowed.

We have also examined several criteria for the second option. Example by measuring the average node utilization, the dead space within nodes, or the spatial distribution of node contents (i.e., their deviation from the center of their MBR). Therefore, we could apply a reorganization when the aforementioned values exceed a specified threshold. However, the experimental evaluation of these criteria did not show significant improvement compared to the criterion that is based on the fraction of underflowed nodes (first option), neither their tuning is easy.

3.2.2. GL method. In this section we describe the GL method in more detail, and give the corresponding algorithm. Let U denote the number of underflowed nodes. After deletion of an entry, if the fraction of U over the total number of nodes N, is equal or larger than a maximum value max U, then a tree reorganization takes place. Table 1 summarizes the parameters that play a role in the experimentation part.

Table 1. Parameters and explanation.

m	Minimum number of entries in a node
M	Maximum number of entries in a node
N	Number of nodes
U	Number of underflowed nodes
max$U$	Threshold ratio U/N
mm	Ratio m/M
r	Ratio of the number of deletes over the number of queries
w	Size of the square window query

For a deletion of an entry, first the leaf containing the entry is identified and then the entry is removed from the leaf. If the node underflows, then U is increased by one. Reinsertion of node entries is not applied, but, instead, the node continues to be underflowed. Depending on the policy to (or not to) update the upper levels, when the node's MBR is shrunk, the respective adjustment is done (not done) in the parent node. Regardless of this policy, no other node at the upper levels will become underflowed since no other deletion takes place (as it would be the case if the underflowed node was removed). Therefore, only leaves may underflow after a deletion. Since nodes remain underflowed, there is a chance that a leaf becomes completely empty after a deletion; however, even in this case, the node is not removed during the deletion operation.

U is increased only at the time that a node becomes underflowed, i.e., when a node with exactly m entries remains with $m - 1$ entries after a deletion. Further deletions of entries from an underflowed node do not increase U. In an analogous manner, U is decreased when an insertion of a new entry takes place in a node that contains exactly $m - 1$ entries. At this time the node stops being underflowed. This is the only required modification to the insertion operation.

If after a deletion operation the fraction U/N is equal or larger than $\max U$, then reorganization is applied. A post order traversal of the tree is performed, i.e., the leaves are examined first. Each underflowed leaf is removed from the tree and its orphaned entries are collected in a list. Then, for each removed entry the corresponding entry in its parent node is also removed, reducing U (and N as well) by one since the underflowed node does not exist any more. If the parent becomes underflowed after this removal (i.e., it contained exactly m entries before the removal), in its turn it is considered as underflowed and the same procedure is applied for it. At the end of the reorganization, all the collected orphaned entries are reinserted. This procedure is done with the same reinsertion procedure that is used by the original R-tree deletion algorithm.

An example of the reorganization procedure is depicted in figure 4. The left part of the figure illustrates an R-tree with M equal to 2 and m equal to 2. Also, it is assumed that $\max U$ is equal to 0.5. If entry G is removed, then the middle leaf will become underflowed (i.e., it will contain less than m entries). Currently, U is equal to 0.25, i.e., only one out of four nodes in total is underflowed. Thus, G is removed but no reorganization takes place. Then, if the J entry is deleted, the right-most leaf becomes underflowed. In this case U becomes equal to 0.5 and, thus, reorganization is performed. The result of the reorganization is depicted at the right part of figure 4. As illustrated, the orphaned entries H and I are inserted in the middle node, whereas the right-most leaf has been deleted along with the corresponding entry, C, in the parent node.

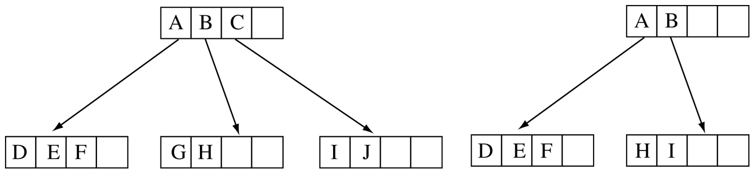


Figure 4. Left: Before deletions of G and J; Right: After reorganization.

```
Algorithm GL (Node Root, entry e)
BEGIN
   Find the leaf that includes e;
   Remove e from the leaf;
   IF the leaf underflows THEN
        $U = U + 1$;
        IF the leaf MBR changes THEN
             Depending on the policy,
             adjust the upper levels;
  IF $\frac{U}{N} \leq \max U$ THEN Reorganize;
END
```

```
Procedure Reorganize
BEGIN
   Find all nodes with less than m entries;
   Remove these nodes from the tree;
   Collect orphaned entries;
   Delete from parent nodes the pointers to the removed nodes;
   Reinsert all the orphaned entries;
   Update U and N values;
END
```

Figure 5. Left: GL algorithm. Right: The Reorganization procedure.

Now, we present the algorithmic description of the GL method, which is based on the ideas discussed above. The left part of figure 5 illustrates the algorithm. As depicted, the main difference in comparison to the original R-tree deletion algorithm (figure 2) is that when a node underflows, only the U counter is updated and, depending on the policy about updating upper-level nodes, adjustments of MBRs are propagated up to the root. The other difference is in the last step of GL where, depending on U/N, a reorganization is performed. Thus, this test implements the criterion for reorganization. The procedure for reorganization is depicted in the right part of figure 5 and it is based on the description made above.

4. Performance results

This section contains the experimental results on the performance of the described deletion techniques. To evaluate their impact on the cost of deletion operation and query performance, we examine workloads that consist of a mixture of such operations (i.e., range search queries and deletions). In the following, we first describe in more detail the experimental setup, and next we give the experimental results.

4.1. Experimental setup

We have implemented all the examined deletion methods and the R-tree structure in C++, under the Windows 2000 operating system.[3] Following the approach of Ravada and Sharma [16], for fair comparison, we used the R*-tree variation but we used forced reinsertion (for the insertion) only for the case of the original algorithm and local methods. For GL, insertions are done without forced reinsertion.

For searching we use the range search query, since it is one of the most basic search queries in spatial databases. The mixture of deletion and search queries is formed as follows. First, all entries are inserted in the tree. Next, we perform a series of deletions and range queries. We keep track of the ratio of the number of points that have been removed divided by the number of points that have been searched. Both deletion and searching are done for square regions of size *w*. More precisely, a range search finds all points that reside within a square of length *w* and a deletion removes all points that reside within a square of length *w*. These squares are generated randomly within the data space. According to the current value of the ratio, if it is smaller than a specified value, *r*, deletion is performed. Otherwise, a range search is performed. Thus, *r* determines the formation of the workload (the lower *r* is, the more the workload is dominated by search queries and less by deletions). The whole procedure is repeated until the R-tree contains less than half of the points that were initially inserted.

We have examined several real and synthetic data sets. Henceforth, for brevity, we present results for the North-East data set,[4] which contains 123,593 points representing postal addresses of three metropolitan areas (New York, Philadelphia and Boston), since the other examined data sets gave no qualitative different results. The page size was set to 4 K and the buffer size to 20% of the data set size. The main performance measure is the total number of disk accesses, i.e., the number of disk accesses required for the completion of the given workload. The reason that we examined this measure instead from the average response time is that the deletion and searching queries in R-trees are I/O bounded, thus the number of disk accesses safely characterizes query performance.[5] Also, we wanted to more clearly examine the impact of different deletion methods on the R-tree independently from the particular implementation.

4.2. *Results*

4.2.1. Local method. First we examined the local deletion method. We varied the lower threshold *mm* for the number of node contents, beyond which the node is considered underflowed. We used a workload for which *w* was set to 0.05 and *r* was set to 0.1. The results with respect to the aforementioned threshold *mm* (e.g., *mm* equal to 0.2 means 20% of the node capacity) are depicted in figure 6(a). The total number of disk accesses required for deletions and range queries are denoted separately. In this figure, the threshold value 0.4 corresponds to the standard R*-tree [2], whereas the value 0 corresponds to the FE technique.

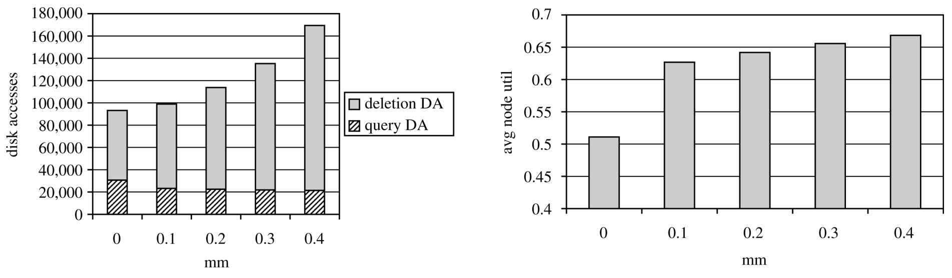


Figure 6. Local deletion method w.r.t. minimum underflow threshold. Left: number of disk accesses. Right: average node utilization.

As shown, the number of disk accesses (for the entire workload) for deletions is reduced significantly with reduced threshold values. This is expected (see Section 3.1), since reinsertion is applied less frequently. On the other hand, the number of disk accesses for range queries (again for the entire workload) is increased slightly when reducing the threshold value. This can be explained by the decrease in the average node utilization with respect to reduced threshold values, which is illustrated in figure 6(b). Clearly, with a reduction of node utilization, more nodes are probable to be fetched during a range query. Nevertheless, considering the entire workload, the small increase in the number of disk accesses due to range queries is by far compensated by the reduction in the number of disk accesses due to deletions. Clearly, the total number of disk accesses, shown in Figure 6(a), illustrate that the FE method (i.e., threshold value equal to 0) attains the best performance for the given workload. For this reason, FE is henceforth the only local deletion method that is examined in the following.

4.2.2. Tuning of the GL method. As described, for the GL (global) method we can follow the policy of, when possible, not updating the MBRs of the upper-level nodes. This leads to a reduced number of required locking operations, and also to a reduction in the cost of deletions. Herein, we first examine this policy, compared to the case where update is performed normally. We used a workload for which r was set to 0.1. The $\max U$ parameter was set to 0.3. The total number of disk accesses (i.e., both for deletions and range queries during the entire workload) are given in figure 7 with respect to query window size w. In this figure, ''w/o'' (''w/'') denotes the policy that performs deletions without (with) updating the upper-level nodes.

Evidently, both cases lead to similar performance. This is explained as follows. Without the updating of the upper-level nodes the query cost increases (due to the not updated MBRs). However, this increase is compensated by the reduction in the cost of deletions (due to less I/O for the node updating).[6] Therefore, in the following we use the policy of not updating the upper-level nodes, since it does impact the overall performance and leads to simpler locking schemes.

The value of $\max U$ (the threshold for the number of underfilled nodes) is significant for the tuning of GL, since it determines how frequently reorganization is applied. We

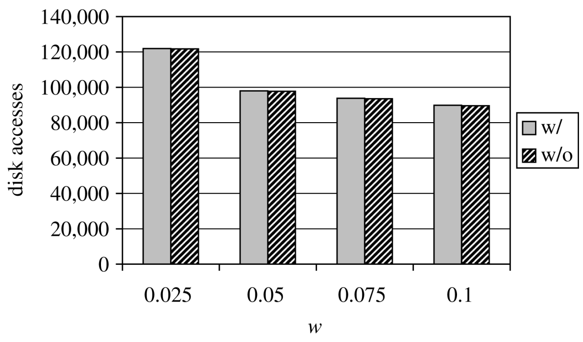


Figure 7. Examination of the policy for GL method.

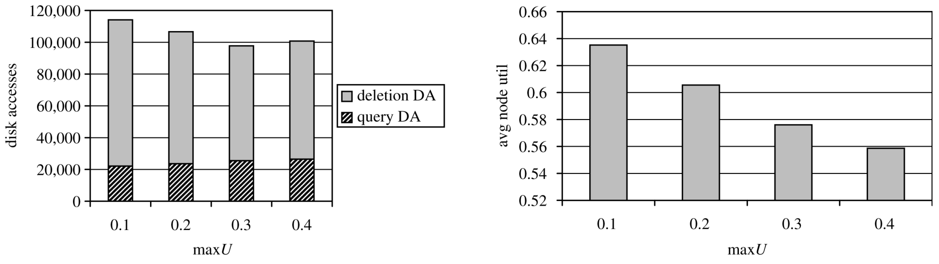


Figure 8. GL method w.r.t. ratio of underflowed nodes threshold (max$U$). Left: Number of disk accesses. Right: Average node utilization.

measured its impact by using a workload for which w was set to 0.05 and r to 0.1. The results are depicted in figure 8(a), where the total number of disk accesses for deletions and range queries are denoted separately.

As shown, max$U$ equal to 0.1 leads to the maximum number of disk accesses for deletion, since the reorganization is applied more often. However, it requires the minimum number of disk accesses for range queries, due to better tree quality that is achieved. With increasing max$U$, the total number disk accesses for deletion is reducing, until when max$U$ is equal to 0.3. On the other hand, the number of disk accesses for the range queries increases slightly with increasing max$U$. This is explained by figure 8(b), which shows the average node utilization during the execution of the mixture of operations. As expected, with increasing max$U$, node utilization decreases, since more underflowed nodes occur (i.e., reorganizations are applied less often). From the total number of disk accesses (for the entire workload) shown in figure 8(a), it is illustrated that the best performance is achieved when max$U$ is equal to 0.3, since this presents a medium ground between the cost of more often reorganizations and retaining tree quality. For this reason, this value is used henceforth in the following experiments.

4.2.3. Comparison of all methods. Finally, we compared FE (best local), GL (global) and the original deletion algorithm that is used by the R*-tree. We used w equal to 0.05, and we varied r to get different types of workloads. The results are depicted in figure 9. In

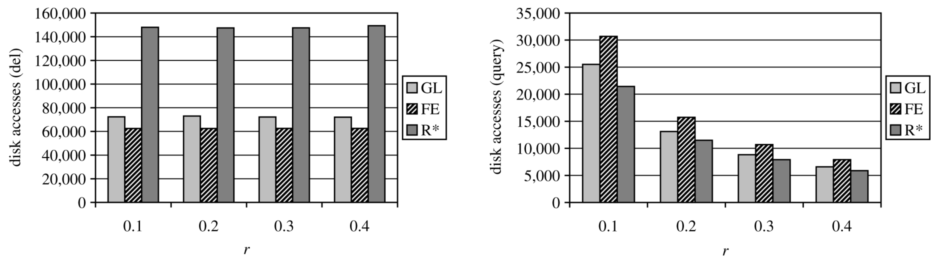


Figure 9. Comparison of all methods. Left: Number of disk accesses for deletion. Right: Number of disk accesses for range queries.

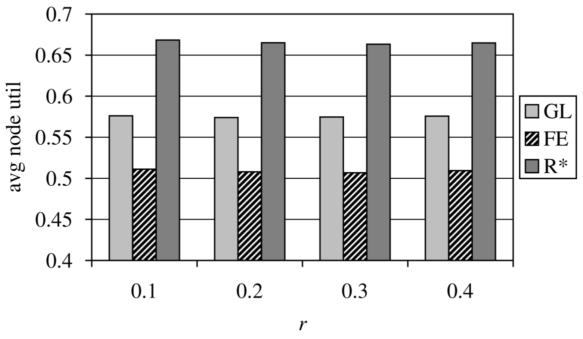


Figure 10. Average node utilization for all compared methods.

particular, figure 9(a) shows the total number of disk accesses (for the entire workload) required by deletions, whereas figure 9(b) those by range queries.

Evidently, the original deletion algorithm (denoted as R^*) requires by far the most disk accesses for deletions in all cases (with a slight increase with increasing *r*). FE requires the least, but GL presents comparable performance to FE. On the other hand, as expected, the original deletion algorithm presents the best query performance, whereas FE the worst, being clearly outperformed by GL. FE results to more underflowed nodes that impact query performance. This can also be explained by figure 10, which depicts the average node utilization during the execution of the mixture of operations. As expected, FE has the worst utilization and is outperformed by GL, whereas the original deletion algorithm attains the best node utilization.

From the above, it can be induced that FE represents the one extreme point that reduces the cost of deletions, by deferring as much as possible the reinsertions, leading however to the worst query performance. The original deletion algorithm represents the other extreme point that achieves the best query performance, but it requires the largest cost for deletions due to the large number of reinsertions. Therefore, GL represents a medium ground between these two points, trying to reduce the cost of deletions without severely impacting query performance.

5. Conclusions

We have presented a performance evaluation of lazy R-tree deletion methods. Their objective is the reduction of the cost required for the deletion operation by the original R-tree deletion algorithm. They defer the costly operation of reinsertion for underflowed nodes.

We have considered two categories of lazy deletion algorithms: (i) local, (ii) global. The local methods are restricted in each single node, treating it independently from the others. In contrast, global methods do not treat underflowed nodes independently and are based on a reorganization procedure. Reorganization is applied according to specified criteria, that are also discussed.

The lazy methods and the original deletion algorithm are examined experimentally. We used workloads consisting of range search queries and deletions. The results indicate that:

- The best local method is the free-at-empty, which allows for a node to have no entries before it is removed.
- The performance of the policy of not propagating the updates to the upper-level nodes for the global method, for mixed workloads, is as good as of the case of propagating updates. Moreover, the former leads to simpler locking schemes.
- The global method represents a medium ground between the original deletion algorithm and the FE method, trying to reduce the cost of deletions without severely impacting query performance.

Future work may include the examination of other reorganization criteria and the development of bulk-loading methods for the reinsertion of orphaned entries after the reorganization procedure.

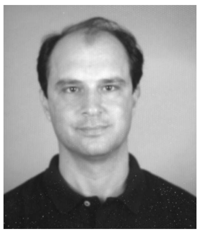
Notes

1. A similar figure was presented during the talk related to Ravada and Sharma [16] (it does not appear in the published paper, however).
2. During the talk related to Ravada and Sharma [16].
3. Since the evaluation was not performed via a real-system, the locking behavior could not be analyzed thoroughly.
4. Available at www.unipi.gr/ ~ ytheod
5. This is the reason why query optimization in R-trees is done according to the estimated number of disk accesses [20].
6. It has to be noticed that the total number of disk accesses are reduced with increasing *w* because with larger *w* more deletions take place at the same time, so the workload terminates more early. However, we are interested in a comparative examination between the two methods, which is not affected by the aforementioned issue.

References

1. R.A. Baeza-Yates and P.A. Larson. ''Performance of B^+-Trees with partial expansions,'' *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1(2):248–257, 1989.
2. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. ''The R^*-tree: an efficient and Robust access method for points and rectangles,'' *Proceedings ACM SIGMOD Conference*, 322–331, 1990.
3. P. Bozanis, A. Nanopoulos, and Y. Manolopoulos. ''LR-tree: a logarithmic decomposable spatial index method,''<http://delab.csd.auth.gr/papers/BNMtcj03.pdf>, The Computer Journal, Vol. 46(3): 319–331, 2003.
4. A. Guttman. ''R-trees: A dynamic index structure for spatial searching,'' *Proceedings ACM SIGMOD Conference*, 47–57, 1984.
5. V. Gaede and O. Guenther. ''Multidimensional access methods,'' *ACM Computing Surveys*, Vol. 30(2):170–231, 1998.
6. Informix Corp.: Informix R-tree Index User's Guide, Version 9.3, 2001.
7. T. Johnson and D. Shasha. '' B-Trees with inserts and deletes: Why free-at-empty is better than merge-at-half,'' *Journal of Computer and System Sciences*, Vol. 47(11):45–76, 1993.

8. I. Kamel and C. Faloutsos. ''Hilbert R-tree: An improved R-tree using fractals,'' *Proceedings 20th Conference on Very Large Databases (VLDB)*, 500–509, 1994.
9. R. Kothuri, S. Ravada, and D. Abugov. ''Quadtree and R-tree indexes in Oracle spatial: A comparison using GIS data,'' *Proceedings ACM SIGMOD Conference*, 546–557, 2002.
10. Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. Advanced Database Indexing. Kluwer, 2000.
11. Y. Manolopoulos. ''B-trees with lazy parent split,'' *Information Sciences*, Vol. 79(1–2):77–88, 1994.
12. A. Manousaka and Y. Manolopoulos. ''Fringe analysis of 2–3 trees with lazy parent split,'' *The Computer Journal*, Vol. 43(5):420–429, 2000.
13. G. Matsliach. ''Using multi-bucket data leaves with overflow chains—performance analysis,'' *Information Systems*, Vol. 16:497–508, 1991.
14. V. Ng and T. Kameda. ''Concurrent access to R-trees,'' *Proceedings 4th Symposium on Spatial Databases (SSD)*, 163–172, 1994.
15. Y. Sagiv. ''Concurrent operations on B^*-trees with overtaking,'' *Journal of Computer and System Sciences*, Vol. 33(2):275–296, 1986.
16. S. Ravada and G. Sharma. ''Oracle8i: Experiences with extensible databases,'' *Proceedings 6th Symposium on Spatial Databases (SSD)*, 355–359, 1999.
17. M. Scholl. ''New file organizations based on dynamic hashing,'' *ACM Transactions on Database Systems*, Vol. 6(1):194–211, 1981.
18. T. Sellis, N. Roussopoulos, and C. Faloutsos. ''The R^+-tree: A dynamic index for multidimensional objects,'' *Proceedings 13th Conference on Very Large Databases (VLDB)*, 89–104, 1987.
19. B. Srinivasan. ''An adaptive overflow technique to defer splitting in B-trees,'' *The Computer Journal*, Vol. 34:416–425, 1991.
20. Y. Theodoridis and T. Sellis. ''A model for the prediction of R-tree performance,'' *Proceedings ACM Symposium on Principles of Database Systems (PODS'96)*, 161–171, 1996.
21. E. Veclerov. ''Analysis of dynamic hashing with deferred splitting,'' *ACM Transactions on Database Systems*, Vol. 10(1):90–96, 1985.



Yannis Manolopoulos was born in Thessaloniki, Greece in 1957. He received his B.Eng (1981) in Electrical Engineering and a Ph.D. (1986) in Computer Engineering, both from the Aristotle University of Thessaloniki. Currently, he is Professor at the Department of Informatics of the latter university. He has been with the Department of Computer Science of the University of Toronto, the Department of Computer Science of the University of Maryland at College Park and the University of Cyprus. He has published over 130 papers in refereed scientific journals and conference proceedings. He is co-author of a book on *Advanced Database Indexing* and *Advanced Signature Indexing for Multimedia and Web Applications* published by Kluwer. He is also author of two textbooks on data structures and file structures, which are recommended in the vast majority of the computer science/engineering departments in Greece. He served/serves as PC Co-chair of the 8th Panhellenic Conference in Informatics (2001), the 6th ADBIS Conference (2002) the 5th WDAS Workshop (2003), the 8th SSTD Symposium (2003) and the 1st Balkan Conference in Informatics (2003). Also, currently he is vice-chairman of the Greek Computer Society. His research interest includes access methods and query processing for databases, data mining, and performance evaluation of storage subsystems. For more information, please visit http://delab.csd.auth.gr/ ~ manolopo/yannis.html

Alexandros Nanopoulos received his B.Sc. and a Ph.D. degree in Informatics from the Aristotle University of Thessaloniki, Greece, in 1996 and 2003 respectively. Currently, he is serving in the Greek army. His research interest includes data mining, databases for web and spatial databases. For more information, please visit http://delab.csd.auth.gr/ ∼ alex/

Michael Vassilakopoulos was born in Thessaloniki, Greece in 1967. He received his B.Eng. (1990) in Computer Engineering and Informatics from the University of Patras and a Ph.D. (1995) in Electrical and Computer Engineering from the Aristotle University of Thessaloniki. His B.Eng. thesis deals with Algorithms of Computational Geometry and his doctoral thesis deals with Spatial Databases. Apart from the above, his research interest includes access methods, spatio-temporal databases, WWW databases, multimedia and GIS. He has published over 35 papers/chapters in refereed scientific journals, conference proceedings and books. He has been with the Department of Applied Informatics of the University of Macedonia and the Department of Informatics of the Aristotle University of Thessaloniki. For three years he also served the Greek Public Administration as an Informatics Engineer where he participated in the evaluation and supervision of several information technology projects. Currently, he is an Associate Professor at the Department of Informatics of the Technological Educational Institute of Thessaloniki.