LR-tree: a Logarithmic Decomposable Spatial Index Method

PANAYIOTIS BOZANIS[1], ALEXANDROS NANOPOULOS[2] AND YANNIS MANOLOPOULOS[2]

[1]*Department of Computer and Communication Engineering, University of Thessaly, Volos 38221, Greece*

[2]*Department of Informatics, Aristotle University, Thessaloniki 54006, Greece*

Email: pbozanis@inf.uth.gr, alex@delab.csd.auth.gr, manolopo@csd.auth.gr

Since its introduction in 1984, R-tree has been proven to be one of the most practical and well-behaved data structures for accommodating dynamic massive sets of geometric objects and conducting a very diverse set of queries on such datasets in real-world applications. This success has led to a variety of versions, each one trying to tune the performance parameters of the original proposal. Among them, the most prominent one is R*-tree, which employs a number of carefully designed heuristics and is widely accepted as achieving the best performance in most cases. However, in the presence of actively changing datasets, R*-tree still does not avoid performance tuning with forced reinsertion, i.e. a process that performs a kind of local rebuilding. The latter fact has motivated the investigation of the adaptation of a known dynamization technique, based on carefully triggered local rebuildings, for converting static or semi-dynamic, main memory data structures to dynamic ones onto R*-trees. In this paper, we present *LR-trees*, a new efficient scheme for dynamic manipulation of large datasets, which combines the search performance of the bulk-loaded R-trees with the updated performance of R*-trees. Experimental results provide evidence on the latter statement and illustrate the superiority of the proposed method.

Received 28 March 2002; revised 23 October 2002

1. INTRODUCTION

Numerous applications have emerged during the last few years that demand the efficient manipulation of massive sets of geometric objects like points, lines, areas or volumes in one or more dimensions. To name just a few, this is particularly useful in geographical information systems, in CAD/VLSI design, and in mobile object movement detection and prediction. The databases that accommodate these specific kinds of objects are called *spatial* and they employ data structures that *must* be capable of answering a very diverse set of *geometric queries*, like *range queries* that ask for all objects lying within a given region, or *nearest-neighbor queries* that seek for the object closest to a given object, and *join queries* that search for all pairs of objects satisfying a given predicate on the set of objects that they accommodate.

The last requirement, combined with the 'massive' volume of the datasets under consideration, is the main reason why practical, general-purpose data structures are in fact used in virtually all real-world applications. This also explains why R-trees, since their introduction by Guttman in 1984 [1], became so popular from the research point of view as proved by the many variants [2]. An R-tree is a height-balanced tree similar to a B^+-tree; actually, it can be considered as an extension of the latter structure for multi-dimensional data. The minimum bounding rectangle (MBR) of each geometric object, along with a pointer to the address where the object actually resides, are stored into the leaves. Each internal node entry consists of a pair (pointer to a sub-tree $\mathcal{T}$, MBR of $\mathcal{T}$), with the MBR of a tree $\mathcal{T}$ defined as the MBR enclosing all the MBRs stored in $\mathcal{T}$. Like in B^+-trees, each node contains at least m and at most M entries, where $m \leq M/2$. On the other hand, unlike B^+-trees, a search query may activate several search paths from the root to the R-tree leaves, resulting, in the worst case, in a performance linear to the size of the dataset just to retrieve a few objects.

As mentioned above, various variants of the R-tree have been proposed, each one aiming at improving the performance by tuning some parameters. Among the members of the 'R-tree family', the most prominent one is the R*-tree of Beckmann *et al.* [3]. In the latter work, a number of heuristics were applied, like forced re-insertions during insertions (as in the case of deletions), buffering and optimization criteria for splitting/merging nodes and adjusting the involved MBRs, so that R*-trees are widely accepted as achieving the best performance. However, a hard fact remains: the construction of any R-tree version by using repeated insertions does not necessarily mean that a 'good' tree is produced in terms of query performance. In fact, the linear worst-case query time complexity cannot even be avoided.

On the other hand, bulk-loading techniques for R-trees do exist that, by exploiting *a priori* knowledge of static datasets,

build the structure from scratch and achieve better utilization and search performance in the average case. The last fact was the main motivation for the present work. Next, we briefly discuss bulk loading. Kamel and Faloutsos [4] used the Hilbert sorting technique to first sort data and then build the R-tree. Leutenegger *et al.* [5] extended this approach, employing a more elaborate technique, according to which successive sorting and division of data into slabs for each of the dimensions is applied. Bercken *et al.* [6] and Arge *et al.* [7] proposed the building of indices with repeated block-wise insertion; buffers are attached to index pages. During the bulk-load, each object is inserted into the buffer of the root. When the root buffer overflows, all objects are dispatched to the next level and so on, until the data level is reached. In [8] a recursive top-down algorithm is employed, which, operating in a manner similar to quick sort, determines the tree topology (height, fan-out, etc.) and uses a split strategy to bisect the data in secondary storage and construct the index directory in a depth-first, post order way.

We now introduce some terminology: a data structure for a searching problem can be *static*, *semi-dynamic* or *dynamic*. A static one is built for a fixed, static set S of objects and is employed for answering queries on S. A data structure is semi-dynamic when it allows either insertions in S, and is called *insertions-only*, or deletions from S, and is characterized as *deletions-only*. A dynamic data structure permits both insertions and deletions to the underlying set S. The dynamic data structures usually support updates employing some rebalancing operations that enable their adjustment to set changes without query performance degrade. We say that an object insertion or deletion is *weak*, whenever such an operation is performed in a 'lazy' manner (that is, without perfectly restoring balance), so that performance is not degraded drastically after executing cn of those operations (where n is the initial size of a perfectly balanced structure and c is a constant). Much research has been conducted in the area of *dynamizing* main memory data structures: since it is easier to design data structures for static sets with good query time, there have been a number of efforts aimed at devising general methods for transforming static or semi-dynamic data structures into dynamic ones for *decomposable* searching problems [9, 10, 11]. A searching problem is said to be decomposable if one can partition the input set into a set of disjoint subsets, perform the query on each subset independently and then easily compose the partial answers. The most useful geometric searching problems, like range searching, nearest-neighbor searching or spatial join, as we will see, are in fact decomposable.

In this paper we present LR-tree, a new index structure, which is based on the logarithmic dynamization method (to be explained later). LR-trees consist of a number of component sub-structures, called *blocks*. Each block is organized as a *weak R-tree*, termed wR-tree, i.e. a semi-dynamic, deletions-only R-tree version. Whenever an insertion operation must be served, a set of blocks is chosen for reconstruction. On the other hand, deletions are handled locally by the block structure that accommodates the involved object. Due to the algorithms for block construction, LR-trees achieve good tightness that provides improved performance during search queries. At the same time, LR-trees maintain the efficiency of dynamic indexes during update operations. We initially describe the applicability of the logarithmic method to the case of R-trees and formalize the problem. Due to the performance requirement to consider deletions as weak operations, we first introduce the wR-tree. LR-trees are defined next, along with the novel algorithms for querying and index updating. We examine performance by both describing theoretical bounds and presenting detailed experimental results, which illustrate the efficiency gains achieved by LR-trees, compared to existing counterparts.

The rest of the paper is organized as follows. Section 2 presents the logarithmic method. In Section 3 we introduce wR-trees, whereas Section 4 defines LR-trees and presents the algorithms for queries and updates. Section 5 gives the performance study. Finally, Section 6 concludes our work.

2. A DYNAMIZATION TECHNIQUE FOR DECOMPOSABLE SEARCHING PROBLEMS

In this section we will use the following notion of *decomposability* [9, 11].

DEFINITION 1. *A searching problem* $\mathbf{P}(q, S)$ *on a set* S *with query* q *is called* decomposable *if and only if for any partition* V

$$V = \{A, B\}, \quad S = A \cup B, \quad A \cap B = \emptyset$$

of S *and any query* q,

$$\mathbf{P}(q, S) = \square(\mathbf{P}(q, A), \mathbf{P}(q, B)),$$

for some operator $\square$ *computable in* $O(1)$ *time.*

It follows promptly that, in order to solve a decomposable problem, one can partition the input set into a set $V = \{V_0, V_1, \ldots\}$ of disjoint subsets, perform the query on each V_i independently and then compose the partial answers using the suitable operator.

Not all searching problems are decomposable. For example, asking whether a query point q lies in the convex hull of a set S of points is not. However, the most useful geometric searching ones, like range searching, nearest-neighbor searching or spatial join, *are* in fact decomposable. In the case of range searching, we can query each V_i independently and then simply merge the partial results using the trivial operator $\square = \cup$. In order to answer nearest-neighbor queries one can use as operator $\square =$ min distance to find the closest object. Finally, in the case of spatial join, we can perform join on every distinct V_i, V_j block combination and then employ $\square = \cup$ to get the result. Practically, the 'decomposability' property means that one need not store all points in one large data structure; instead, one can manipulate the input set as a dynamic set of disjoint subsets or 'blocks'. Each block is accommodated in an

independent data structure, and the queries are treated by merging the separate answers.

This block partition technique is widely applied in the area of main memory data structuring as a method for dynamizing static data structures. Two versions have been proposed [11]: (i) the *equal block method*, which maintains almost equal block sizes; and (ii) the *logarithmic block method*, that uses blocks of exponentially increasing sizes. In this paper, we will consider the second case since in [11] it was proved that the equal block method gives good trade-off bounds for query-deletion times *degrading* the performance of the insertion operation. The interested reader can consult [11] and the references of Section 1 for a more detailed treatment. In the following we will assume that the searching problem $\mathbf{P}(q, S)$ under consideration can be solved by a semi-dynamic, weak deletions-only data structure D.

The logarithmic method

Let us assume that the search problem on the set S in question can be solved by a static data structure D_S. According to the logarithmic method, block V_j accommodates zero or 2^j elements and is organized as a data structure D_{V_j}. Whether a block V_j is void or not is determined by the binary representation of the cardinality n of the original set S. In order to insert a new element, one must: firstly, locate the smallest i such that V_i is empty; secondly, destroy all V_j with $j < i$; and lastly, build a new data structure D_{V_i} corresponding to the block V_i, which, from now on, accommodates the new element and all the elements from the discarded blocks (data structures). The queries are answered by combining with the appropriate operator $\square$ the partial results obtained independently from each block V_j.

For example, assume that set S consists of $55 = 110111_2$ elements. Then S is organized in five blocks V_0, V_1, V_2, V_4, V_5 containing 1, 2, 4, 16 and 32 elements, respectively. In order to insert a new element, we destroy V_0, V_1 and V_2 and build V_3 since $56{=}111000_2$. The example shows the intuition behind the insertion algorithm: it is known that the amortized cost of the carry propagation is constant. When we insert a new point to the structure, the actions that follow are in one-to-one correspondence to adding one to the cardinality of the accommodated set of elements. And so, on the average, the number and the size of blocks getting involved in the insertion algorithm are small enough to compensate for the rebuilding cost. Additionally, one has the benefits of the well-organized block substructures when searching the structure.

One can actually serve deletions equally well, if the underlying, auxiliary data structure D is *weak semi-dynamic*, i.e. it can support deletions in a 'weak' only fashion. In other words, deletions operate in a quick-and-dirty way, that does not improve the query performance asymptotically, but it does not degrade it either. The following theorem summarizes the complexity bounds achieved with the logarithmic method in the case of a semi-dynamic data structure D with query time $\mathcal{Q}(n)$, bulk-loading[3] or preprocessing time $\mathcal{P}(n)$, weak deletion time $\mathcal{D}(n)$ and storage requirements $\mathcal{S}(n)$ [11].

THEOREM 1. *Given a semi-dynamic, weak deletions-only data structure D for a decomposable problem $\mathbf{P}$, one can build a fully dynamic data structure D' for $\mathbf{P}$, such that:*

$$\mathcal{Q}'(n) = \begin{cases} O(\mathcal{Q}(n)), & \text{when } \mathcal{Q}(n) = \Omega(n^\epsilon), \\ & \epsilon > 0, \\ O(\log n)\mathcal{Q}(n), & \text{otherwise} \end{cases}$$

$$\mathcal{I}'(n) = \begin{cases} O(\mathcal{P}(n)/n) & \text{when } \mathcal{P}(n) = \Omega(n^{1+\epsilon}), \\ & \epsilon > 0, \\ O(\log n)\mathcal{P}(n)/n & \text{otherwise} \end{cases}$$

$$\mathcal{D}'(n) = O(\log n + \mathcal{D}(n) + \mathcal{P}(n)/n)$$

$$\mathcal{S}'(n) = O(\mathcal{S}(n))$$

where $\mathcal{Q}'(n), \mathcal{I}'(n), \mathcal{D}'(n)$ and $\mathcal{S}'(n)$ denote the query time, the insertion time, the deletion time and the storage requirements of D', respectively.

This theorem actually states that if the underlying semi-dynamic data structure has superlinear building time and superlogarithmic query time, then one can construct a fully dynamic data structure without performance loss, asymptotically speaking, while the space requirements remain the same. Observe that if the construction or bulk-loading time is linear, then the insertion time is actually logarithmic. The update bounds are amortized, but they can also be converted into worst-case ones, in a rather complicated and impractical way, which is omitted for brevity. For all practical purposes, the amortized-case is adequate.

Similar formulae can be easily derived if the cardinalities of the subsets are powers of some constant $B \geq 2$ (cf. [10, 11]). In this case, we partition S into a logarithmic number of disjoint blocks V_j of size $a_j B^j$, where the a_j's are the coefficients of the representation of n in a numeric system with base B, and thus they take values less than or equal to $B - 1$. A block V_j of size $(B-1)B^j < B^{j+1}$ is called *completely filled*; otherwise, i.e. $0 \leq a_j \leq B - 2$, it is *partially filled*. Now, in such a configuration, a new element can be accommodated as follows. Let j be the minimum integer such that the block V_j is partially filled (observe that $1 + \sum_{k=0}^{j-1} |D_k| = B^j$.) We destroy the first $D_0, \ldots, D_j$ structures, and we build a single D_j structure, with the new point included. In the case where deletions are allowed, then the definition must be slightly changed: a block V_j is partially filled if it can accommodate the items of all blocks V_i, $i \leq j$, plus a new item without exceeding the B^{j+1} bound; otherwise, it is completely filled.

Static R-trees fulfil the requirements of the above theorem. Namely, their worst-case query time is linear to the number of stored elements and the preprocessing time is superlinear (i.e. $O(n \log n)$ to be precise [4]). If a

[3]This term is widely used by the database community for the procedure of constructing an index for a given dataset from scratch. In the context of main memory data structures, this procedure is known as *preprocessing*.

deletion algorithm, designed to be weak, is added in order to transform them into a semi-dynamic, weak deletions-only data structure, then one has at one's disposal a structure suitable for the organization of the blocks of Theorem 1. Thus, the application of the logarithmic method is possible, generating a data structure, let us call it an *LR-tree*, which has the same asymptotical performance characteristics in the worst case. However, since the whole construction is based on rebuilding that, by nature, gives trees with very good time-space complexity, one can expect that eventually the resulting index LR-tree is also very good as a whole. The fact that the insertion/deletion heuristics of the R*-trees essentially employ a kind of local rebuilding in the form of forced re-insertion simply strengthens our argument. In the remaining part of this paper, we will prove that our intuition is correct. Here, we must note that the logarithmic method was—independently to this work—employed in [12] in order to dynamize *wp-trees*, a secondary memory generalization of kd-trees, quad-trees, BBD-trees and the like, i.e. internal memory structures that are built by a recursive decomposition of the space into subspaces. The results presented in [12] involved: (i) a 'pagination' step, during which the nodes of the static, in-memory structures are transferred (bulk-loaded) to the disk using a top-down, breadth-first-like traversal; and (ii) the dynamization step, which transforms the static structure into a dynamic one; for this, they proposed the employment of either the partial rebuilding technique for kd-trees and quad-trees of [11] or the logarithmic method. The treatment was theoretical, without any experimental evaluation. We employ the logarithmic method to the context of R-trees, which means confronting different challenges like the re-insertion heuristic, while, additionally, we give carefully designed experimental results for the first time.

3. R-TREES AND WR-TREES

Among the several R-tree variations, R*-tree is considered as the most prominent one. Based on a number of carefully designed heuristics, it addresses the deficiencies of the original R-tree algorithms. It capitalizes on the insertion phase, since it is critical for query performance. R*-tree introduces the forced reinsertion technique, which avoids splits by reinserting a fraction of the entries from an overflowed node. Regarding node splitting, R*-tree takes several factors into account: overlap between nodes, node perimeters and storage utilization. Also, it uses the plane-sweep technique to separate the node entries. However, deletion and searching are identical to the respective R-tree algorithms.

THEOREM 2. *A set S of n geometric items can be accommodated in an R*-tree using $O(n/b)$ space so that a search for an item has linear worst-case time complexity, an insertion of an item can be served in $O(\log_b n \times b)$ worst-case I/O time, while, given the location of an item, it can be deleted in $O(\log_b n \times b)$ worst-case I/O time (b denotes the node capacity or block size and I/O time refers to the number of block retrievals).*

Proof. It follows from the description in [3]. □

Here we must note that, in the past, several analytical works for the query operations have appeared that are characterized by limited generality (see, for example, [13, 14, 15, 16].) More specifically, they simply derive approximate estimates based on a number of assumptions (e.g. uniformity of the underlying distribution, known aspect ratio of MBRs, etc.).

In order to apply Theorem 1, we must have at our disposal a semi-dynamic weak version of R-tree, which we will call *wR-tree* from now on. An R-tree that is built properly, i.e. which has no node capacity violations (neither overflows nor underflows) and all the MBRs of the leaves and the index nodes are correct, is called a *legitimate instance*.

DEFINITION 2. *A wR-tree on a set S of items is defined as an R-tree, which either is a legitimate created instance on only the members of S, or is constructed from a legitimate instance on members of a set $S' \supset S$ by deleting each $x \in S' - S$ from the leaves so that: firstly, all MBRs of the structure are correctly calculated; and, secondly, the tree nodes are allowed to be under-utilized (i.e. they may store less than m entries.)*

The above definition reflects a situation arising when deletions are allowed. During an insertion operation, a number of elementary data structures are discarded and replaced by a new, bulk-loaded data structure. Whenever an element has to be deleted, this must be done quickly but by preserving the upper levels MBRs at the same time. The simplest way to achieve this is by, once the element in question is located, just removing it from the lowest level, adjusting the MBRs and ignoring the violation of the utilization criterion that may result. This brute-force method works correctly without any performance loss, as we will see in Section 5. Typically, we have the following.

LEMMA 1. *A wR-tree can store a set S of n items so that: (i) it employs $O(n/b')$ space; (ii) it can search for items with the same to R-trees linear worst-case time complexity; and (iii) given the location of an item, it can delete it in $O(\log_{b'} n)$ time (with b' being the average minimum node capacity during its lifetime).*

Proof. Space and search complexity are immediate results of the R-tree nature of the structure. As for the deletion complexity bound, one has to observe that all that is needed is climbing the path leading from the involved leaf to the root and (probably) adjusting the respective MBRs. □

This approach is completely opposite to the time-consuming treatment adopted by R*-trees, according to which, in order to strictly maintain the utilization, all underflowed nodes on the search path are deleted and the resultant 'orphaned' entries are compulsorily re-inserted into the tree level to which they belong, resulting in $O(\log n \times b)$ performance in the worst case. It is important to note that B$^+$-trees follow a similar behavior in industrial applications, i.e. they do not perform node mergings but only free nodes when they are empty [17].

4. DEFINITION, UPDATE AND QUERY OPERATIONS OF LR-TREES

In this section we provide the formal definition of LR-trees, as well as the algorithms for the update and the query operations. Typically, an LR-tree is a collection of a varying, yet bounded from above, number of wR-trees.

DEFINITION 3. *An LR-tree on a set S of n items is defined as a collection $\mathcal{C} = \{T_0, T_1, \ldots\}$ of wR-trees on a partition $V = \{V_0, V_1, \ldots\}$ of S into disjoint subsets (blocks), such that:*

(a) *there is a 'one-to-one' correspondence $V_j \leftrightarrow T_j$ between the subsets (blocks) and the elements accommodated in the trees;*
(b) *$(B^{j-1} < |V_j| \leq B^{j+1}) \vee (|V_j| = 0)$, for some constant $B \geq 2$, which is called 'base'; and*
(c) $|\mathcal{C}| = O(\log_B n)$.

In the discussion that follows, we will assume that the root nodes of the wR-trees are stored into an array named *Root*, ordered by increasing sizes of their subtrees.

4.1. Insertion

As explained in Section 2, an insertion can be served by finding the first wR-tree T_j that can accommodate its own items, the items of all wR-trees to its left and the new item, then destroying all wR-trees T_k, where $k \leq j$, and finally bulk-loading a new index T_j, which stores the items in the discarded structures and the new element. The following algorithm demonstrates these steps.

Algorithm INSERT(**item** p, **array** Root)
Input: The item p to be inserted and the main memory array Root containing the root nodes of sub-trees of LR-tree.
Output: The new instance of LR-tree accommodating p.
1. Find min j such that $1 + \sum_{k=0}^{j} |\mathsf{Root}[k]| \leq B^{j+1}$
2. Destroy all $\mathsf{Root}[k]$, $k \leq j$
3. Create $\mathsf{Root}[j]$ with the contents of destroyed trees plus the new point p using bulk-loading
4. $N = N + 1$ /* N represents the total # points */
end of INSERT

Figure 1 illustrates a comprehensive example, for base $B = 2$ and node capacity $b = 4$: the LR-tree of Figure 1a accommodates 11 items. Since the binary representation of 11 is 1011, items are partitioned into three blocks, namely V_0, V_1 and V_3. Each block V_i is stored into wR-tree T_i. When the item L is inserted (Figure 1b), the cardinality of the collection becomes 12, which equals 1100 in the binary enumeration system. So, the first blocks V_0, V_1 must be destroyed and replaced by a single block V_3, consisting of the elements of the set $V_0 \cup V_1 \cup \{L\}$. That change is reflected in the LR-tree, by replacing the wR-trees T_0 and T_1 by a single one T_3.

4.2. Deletion

The deletion of an item p is a two-step operation: firstly, one should locate the wR-tree in which the item resides, if it actually does; and secondly, one must eventually delete p. More typically, the following algorithm is employed.

Algorithm DELETE(**item** p, **array** Root)
Input: The item p to be deleted and the main memory array Root containing the root nodes of sub-trees of LR-tree.
Output: The new instance of LR-tree with p deleted.
1. Locate the wR-tree $\mathsf{Root}[j_p]$ where p resides
2. **if** p does not exist
3. quit
4. **else**
5. delete p from $\mathsf{Root}[j_p]$
6. $N = N - 1$
end of DELETE

Because of the 'weak' nature of the underlying wR-trees, a deletion either can be promoted to upper levels, causing underflow(s), or may be restricted to the leaf level, causing no further alterations to the data structure. Figure 2 exemplifies these two potential cases. If, from the instance illustrated in Figure 1b, item H is deleted, then, firstly, the respective page underflows, while the MBR R must be recalculated to reflect the change. Figure 2b shows the simple case of deleting item F, without causing either structural or 'quality' changes to the LR-tree.

4.3. Range and nearest-neighbor search

A range query with query rectangle Q seeks for all items whose MBRs share with Q common point(s), while a nearest-neighbor query with point p asks for the item closest to p. These two operations are treated by querying the individual wR-trees and concatenating the partial results trivially in $O(1)$ time. Let RSEARCH(T, Q) and NNSEARCH(T, p) denote correspondingly the range searching and the nearest-neighbor searching procedures on a wR-tree T. Then we have the following algorithms.

Algorithm RANGESEARCH(**rect** Q, **array** Root)
Input: The query rectangle Q and the main memory array Root containing the root nodes of sub-trees of LR-tree.
Output: All items lying into Q.
1. $QS = \{\mathsf{Root}[i] \mid \mathsf{Root}[i] \cap Q \neq \emptyset\}$
2. $Answer = \bigcup_{T \in QS} \text{RSEARCH}(T, Q)$
end of RANGESEARCH

Algorithm
NEARESTNEIGHBORSEARCH(**point** p, **array** Root)
Input: The query point q and the main memory array Root containing the root nodes of sub-trees of LR-tree.

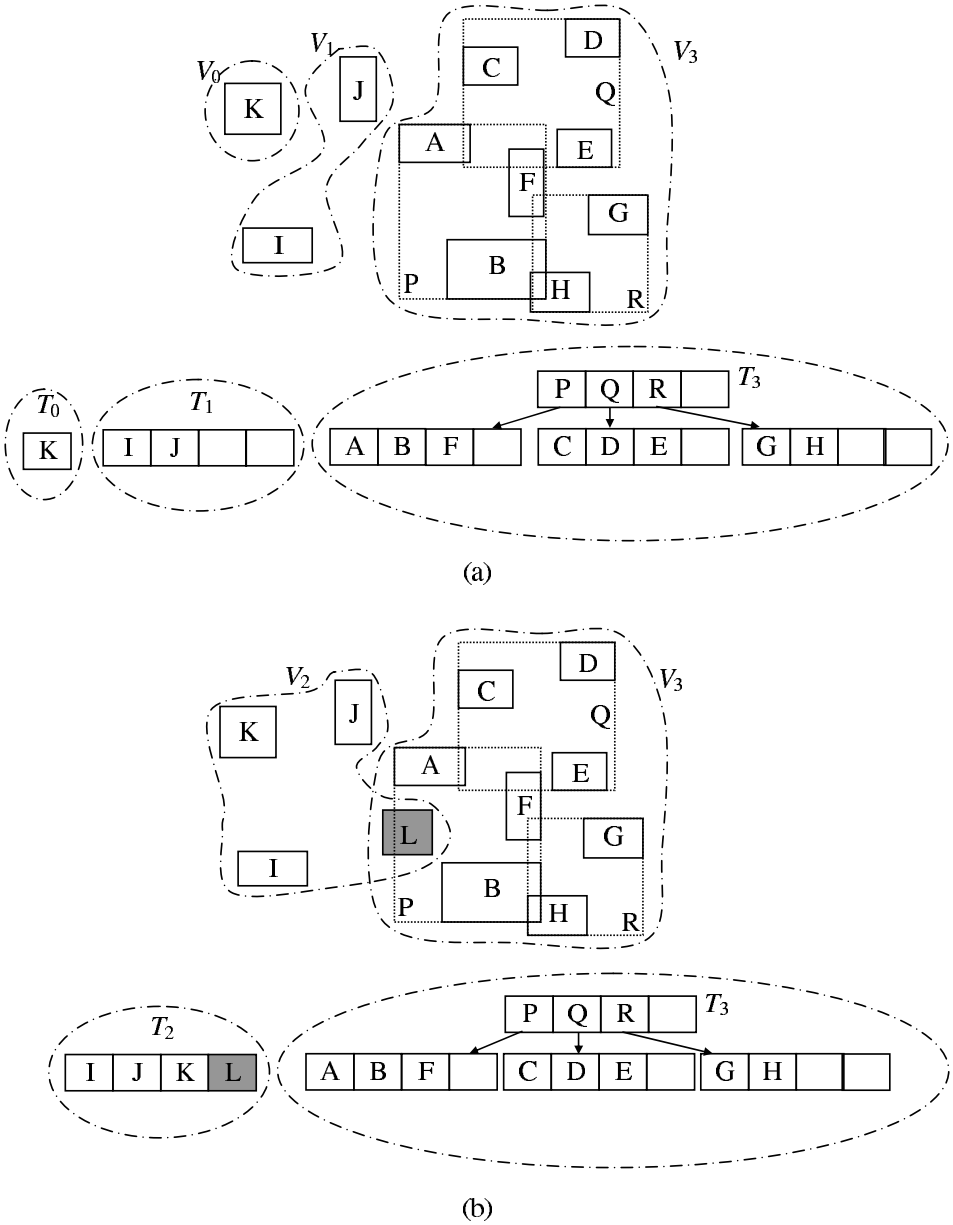


FIGURE 1. Insertion of item L in an LR-tree with $B = 2$. (a) The number of elements before insertion equals $11 = 1011_2$, so the blocks V_0, V_1, V_3 are implemented as wR-trees T_0, T_1, T_3, respectively. (b) After insertion we have $12 = 1100_2$ items and therefore destruction of trees T_0, T_1 and replacement by a single tree T_2.

Output: The nearest neighbor of p.

1. $QS = \{\text{Root}[i] \mid \text{Root}[i] \neq \emptyset\}$
2. $S = \bigcup_{T \in QS} \text{NNSEARCH}(T, p)$
3. $Answer = \min_{\{x \mid x \in S\}} \text{distance}(x, p)$

end of NEARESTNEIGHBORSEARCH

The above NEARESTNEIGHBORSEARCH algorithm can be easily extended so that the $k > 1$ nearest neighbors of a query point p can be detected: firstly, one queries the first $m \geq 1$ wR-trees for the k nearest neighbors, with m being the smallest number of wR substructures needed so that a 'seed' collection S of k points is formed. Then one gradually refines the answer set S by exploring, one by one, the other wR-trees for potentially closest points.

5. PERFORMANCE STUDY

This section presents the performance evaluation of the examined indexes, namely, LR-tree and R*-tree. Although we focus on dynamic indexing methods, for purposes of comparison, we include in our measurements the performance of the bulk-loaded R-tree, which represents the case of a static data structure and can be considered as a lower bound for the performance of the dynamic indexes.

We examine the query execution cost with respect to similarity search queries (i.e. range and k-nearest-neighbor

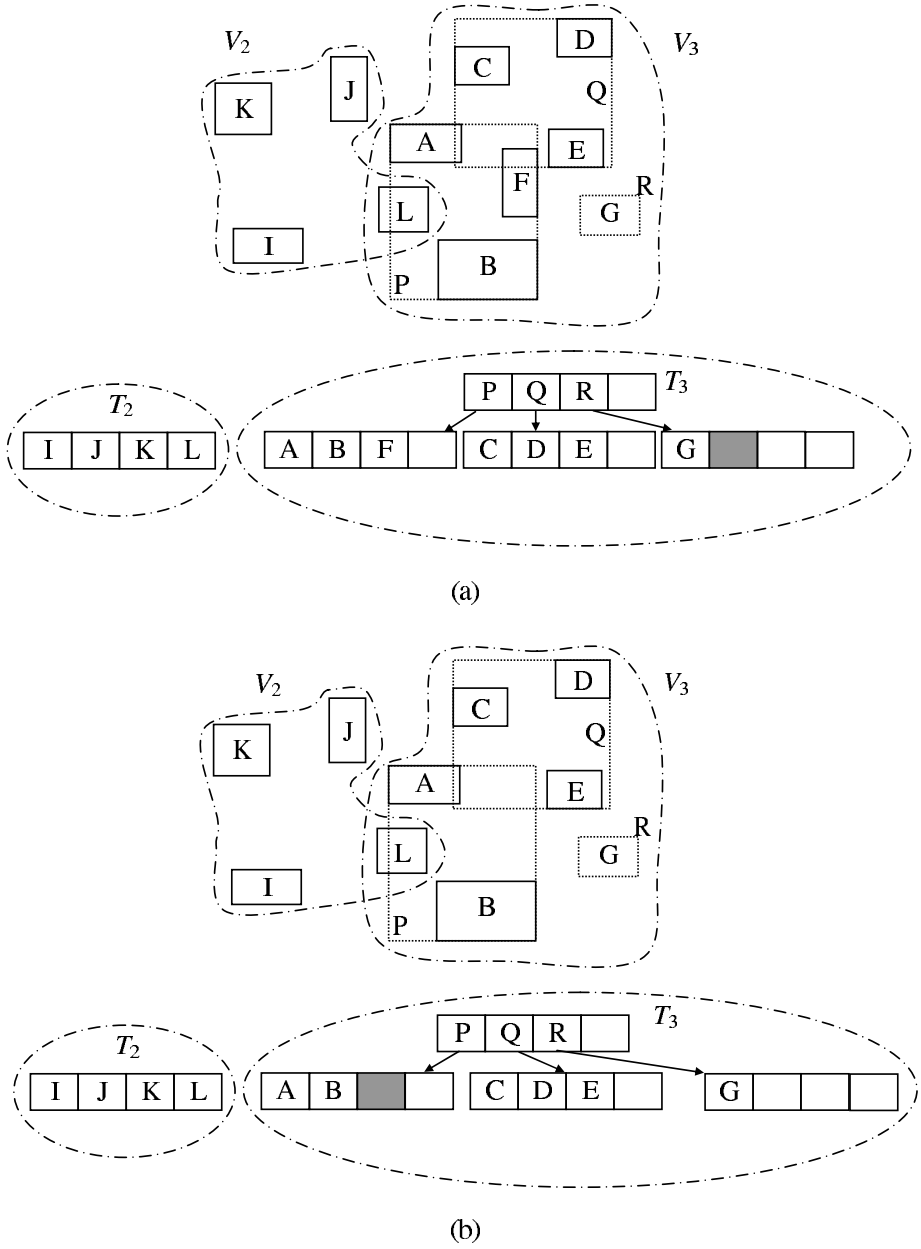


FIGURE 2. Two instances of item deletion. (a) Deletion of H causes (i) MBR shrinkage and (ii) node underflow. (b) Deletion of F is restrained to the leaf level.

queries). Update cost is examined for the insertion and deletion operations, whereas the impact of deletion on query performance (since deletion is a weak operation in LR-tree) is examined separately, for a mixture of operations. Finally we examine the scale-up with respect to the dataset size. We point out the examination of the performance of other operations (e.g. join query) as future work.

Our results illustrate that LR-tree clearly outperforms R*-tree and achieves a search query execution cost that is comparable to that of bulk-loaded R-tree. Regarding dynamic update operations, LR-tree presents comparable performance with R*-tree. Therefore, LR-tree is a dynamic index structure which combines the advantages of the bulk-loaded R-tree, with respect to search queries, and R*-tree, with respect to dynamic updates.

5.1. Theoretical bounds

As explained in Section 3, there are no general analytical formulae about the query performance of R-trees, except the trivial worst-case linear behavior, which can be exhibited with particular datasets. So, we find our first evidence about the applicability of the logarithmic method to R-trees assuming the worst-case behavior of R-trees; in the following sections, we will see that, in practice, one can expect much better performance characteristics.

THEOREM 3. *Let S be a set of n geometric items and $\mathcal{T}$ be an R*-tree on S with worst-case query time complexity $\mathcal{Q}_o(n)$, $o \in$ {nearestNeighbor,rangeSearch,spatialJoin}, worst-case insertion time $\mathcal{I}(n)$ and worst-case deletion time $\mathcal{D}(n)$. Then, S can be accommodated in an LR-tree achieving worst-case query time $\mathcal{Q}_o(n)$, $o \in$ {nearestNeighbor,rangeSearch,spatialJoin}, worst-case insertion time $\mathcal{I}(n)O(\log_B n/b)$ and average deletion time $O(\log_B n + \mathcal{D}(n)/b)$ (b being the node capacity or page size).*

Proof. It follows by applying Theorem 1 to the results of Theorem 2 and Lemma 1. □

5.2. Experimental setup

We have implemented all examined methods in C, using the same components for the common tasks among the methods. All experiments were performed on a machine with a Pentium III processor at 1.6 GHz, with 128 MB main memory and with a 30 GB hard-drive. We used both real and synthetic data sets. We present results for the LB data set, which contains 53,000 two-dimensional points representing postal addresses at Long Beach, and the NE data set, which contains 123,593 points representing postal addresses of three metropolitan areas (New York, Philadelphia and Boston). Both the aforementioned data sets have been used as benchmarks in prior work (e.g. [4]). For synthetic data sets we present results for two-dimensional 100,000 points following a uniform and zipfian distribution.[4] Based on the approach of Beckmann *et al.* [3], we used a default page size equal to 1 kB (other page sizes gave analogous results). We tried several values for the base B of the LR-tree. Henceforth we use B equal to 10, because this value led to the best performance results.

The implementation of the wR-trees was based on common components with the implementation of the R*-trees. We used the k-nearest-neighbor algorithm that is described in [18]. For the bulk-loading method we used the algorithm of [4]. It first scans the points that will be inserted and sorts them according to their Hilbert value. In the following, the algorithm groups points into leaf nodes (each leaf is entirely filled) and stores them on disk. When all leaf nodes are stored, their MBRs are formed and stored at the upper level. The procedure continues up to the root node and then terminates. Following [4], we perform the sorting of Hilbert values in main memory. However, it is easy to extent this procedure to consider external sorting, if necessary. We address the latter as an issue of future work.

For convenience, we follow the approach of prior work and we consider square-shaped range queries, characterized by the size of the square. We are interested in the relative performance of the examined methods; therefore, we use a path-buffer (containing the current path from root to leaf) but no other buffer space, to clearly examine the behavior of the methods regardless of the effect of buffering.

[4]For each dataset, all point coordinates were normalized in the range [0, 1024].

5.3. Experimental results

Our first experiment considers range search queries. We consider both real and synthetic data sets. As a performance measure we use the number of page accesses. Since the sizes (number of points) of the data sets are not equal, we present the normalized results, i.e. the page accesses of the LR-tree and R*-tree are normalized by the page accesses of the bulk-loading method. This way, the relative performance of the examined methods is more clearly presented for the different data sets. Figure 3 illustrates the results with respect to the range query size. The size of the query (square) for this measurement is given in terms of the percentage (%) of the work space.

Focusing on the real data sets, we see in Figures 3a and 3b that LR-tree clearly outperforms R*-tree in most cases. Only for small queries do the algorithms have similar performance, with the R*-tree presenting a slight improvement. For medium and large queries, the LR-tree performs similarly to bulk-loaded R-tree. However, the bulk-loaded R-tree is a static index, whereas the LR-tree offers the advantages of a dynamic index during updates. Analogous conclusions can be drawn from the synthetic data set with uniformly distributed points. For the synthetic data set with points following a Zipf distribution, the LR-tree compares favorably to the R*-tree. The LR-tree presents an improvement of a factor of about three with respect to the R*-tree and its performance is very close to the bulk-loaded tree. This can be explained because the zipfian data set is more demanding, since the distribution of the points (and the queries also) is very skewed. Thus, the better organization achieved by LR-tree clearly pays off in this case.

Next, we examined the k-nearest-neighbor queries. We present results for the two real data sets (the results for the synthetic data sets are analogous and are omitted for brevity). As in the case of range queries, the page accesses are normalized with respect to the bulk-loading method and are given in Figure 4 (due to the difference in the data sets' sizes, we examine different numbers of searched nearest neighbors in each case). Evidently, analogous conclusions can be stated as in the case of range queries. For queries requiring the finding of less nearest neighbors, the R*-tree performs slightly better than the LR-tree. However, for queries requiring the finding of more nearest neighbors the R*-tree clearly loses out. As previously, for such queries, the performance of LR-tree is comparable to that of the bulk-loaded R-tree.

Our next experiments examine update operations. As performance main measure we use the wall-clock time. Update operators involve sorting, plane-sweeping of node entries (during R*-tree split), finding the entries to be reinserted (during R*-tree reinsertion), sorting entries for bulk-loading (during LR-tree construction phase) and other operators which require non-negligible CPU cost, additional to the I/O cost (which is not the case for search queries, where I/O cost is the dominant one). Therefore, wall-clock time considers both these two cost factors.

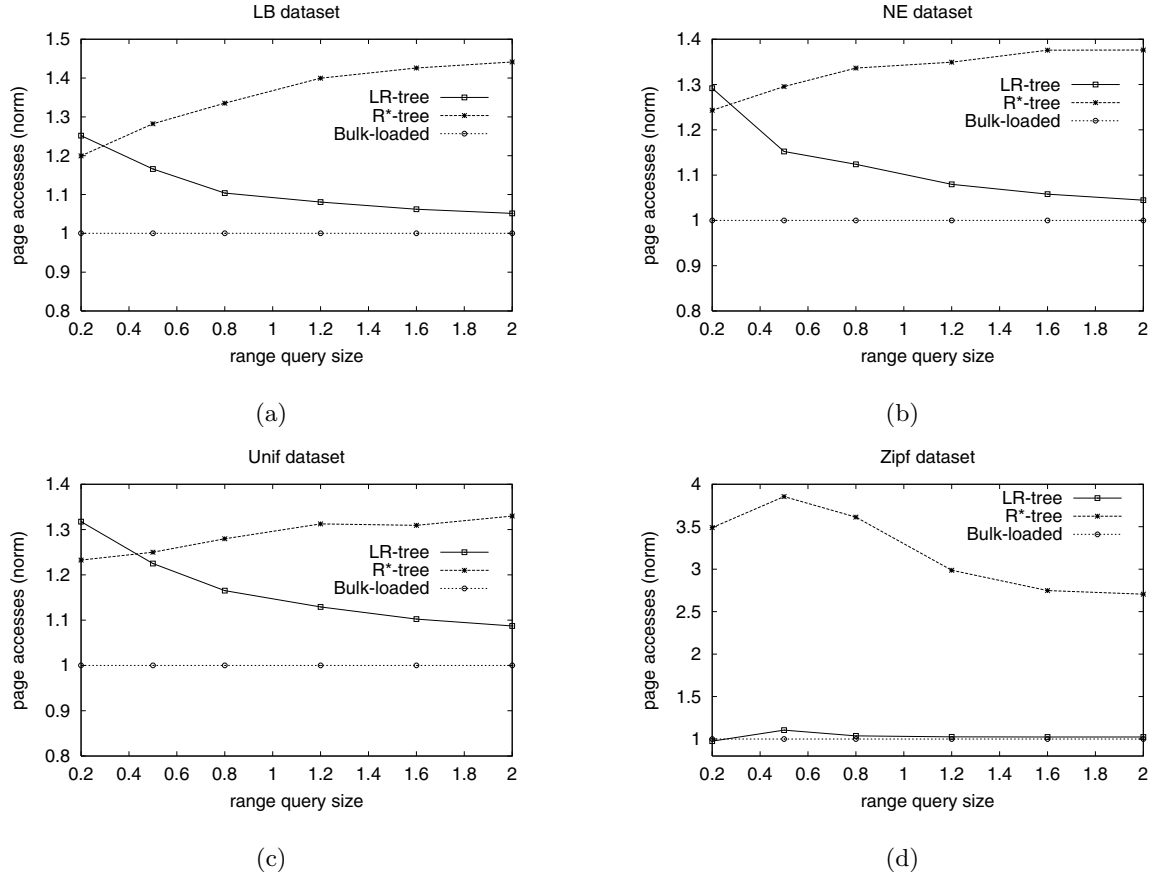


FIGURE 3. Range query. (a) Long Beach real data set. (b) North East real data set. (c) Synthetic uniform data set. (d) Synthetic zipfian dataset.

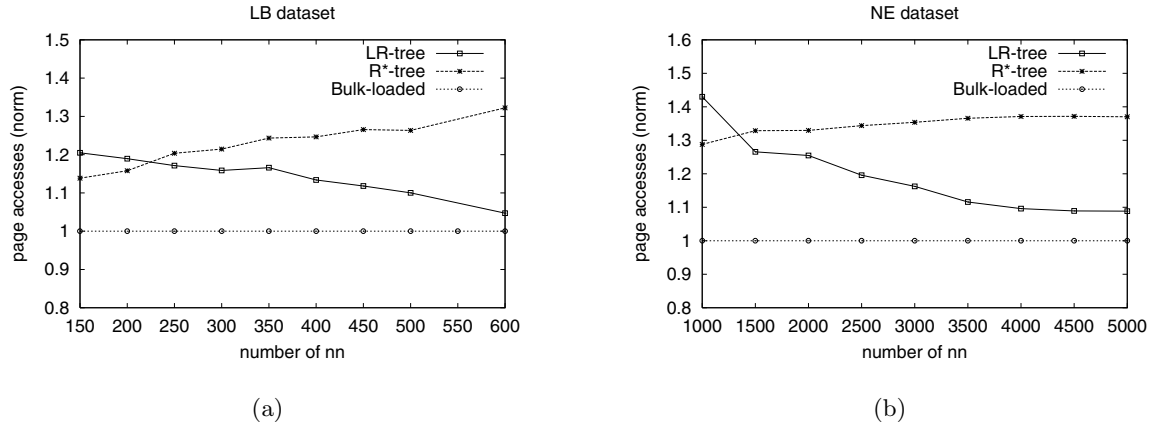
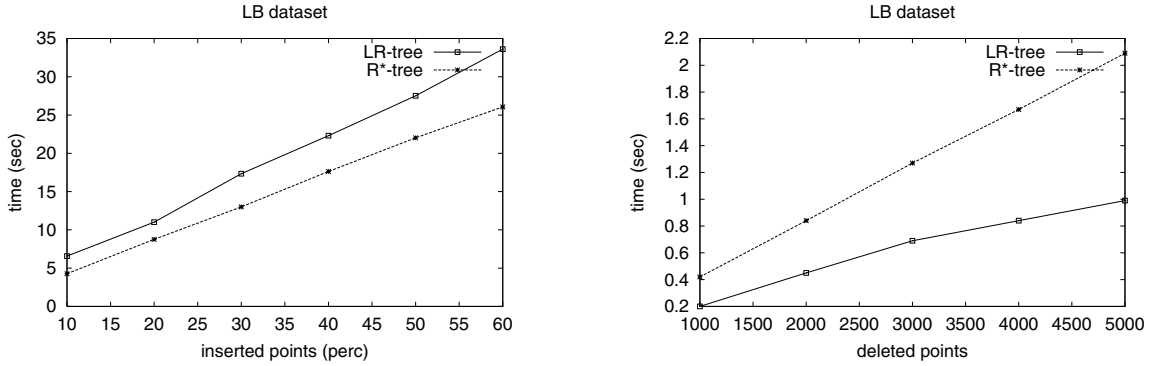


FIGURE 4. k-nearest-neighbor query. (a) Long Beach real data set. (b) North East real data set.

First, we focus on insertion. We present results for the Long Beach data set (the other data sets gave results that led to the same qualitative conclusions). We examined all methods by initially inserting a number of points from the data set and measuring the insertion time (in seconds) with respect to the number of remaining points. The number of inserted (i.e. remaining) points is given as a percentage of the total number of points in the data set. We omit the results for the case of the bulk-loaded R-tree, since it is a static index and cannot handle dynamic insertions.[5] Figure 5a illustrates the results. As depicted, the R*-tree has the best performance. However, LR-tree presents comparable performance. Therefore, LR-tree achieves the property of a dynamic index structure by efficiently handling dynamic insertions. For this experiment we also give, in

[5]By using bulk-loading for each insertion, we found that the bulk-loaded R-tree requires time that is two orders of magnitude larger than the other methods.



(a) (b)

FIGURE 5. Update operations. (a) Insertions. (b) Deletions.

TABLE 1. Page accesses (normalized) for the insertion operation.

Inserted points (%)	LR-tree	R*-tree
10	0.72	1
20	0.61	1
30	0.64	1
40	0.61	1
50	0.62	1
60	0.65	1

Table 1, separately the number of page accesses. The page accesses for the LR-tree are normalized with respect to those of the R*-tree. As shown, the LR-tree requires less page accesses during insertion, since it avoids the reinsertion procedure of the R*-tree. However, it involves the construction of subtrees, which uses a sorting procedure that requires a significant CPU cost. This is the reason why the measurement of the wall-clock time of Figure 5a reveals the actual cost for the dynamic insertion of points.

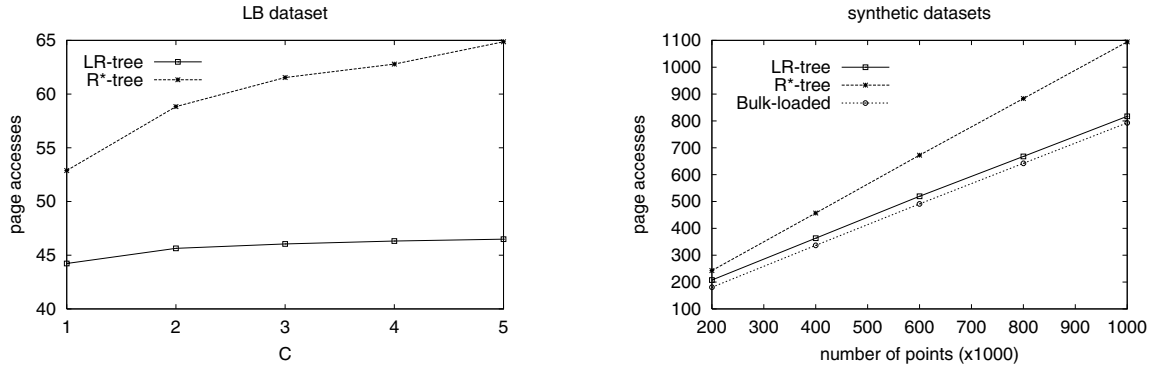
We also examined the execution time required by the dynamic deletion operation. Figure 5b illustrates the results with respect to the number of deleted points (for the reasons described earlier, we do not examine the bulk-loading method). Focusing on LR-tree, it performs better than R*-tree, since it does not use reinsertion (as explained in Section 3). Therefore, LR-tree can efficiently handle dynamic deletions. It has to be mentioned that execution times for the LR-tree deletion are expected to be much less, if the location of the object to be deleted is maintained in a B^+-tree (see Section 5.5).

The deletion is a weak operation in LR-tree. Therefore, we examined its impact on query execution. We used a mixture of deletion, insertion and range queries to simulate a typical workload. Initially, half of the dataset points were inserted and no deletion was performed. For the remaining points, the ratio, C, of the number of inserted points to the number of deleted points is a parameter[6] (deletions are performed by removing previously inserted points). Interleaved with insertions and deletions, range queries were performed and the average number of disk accesses required by the range queries was measured. Figure 6a illustrates the results with respect to C for the LB dataset (the results for the bulk-loaded R-tree are omitted due to very long execution times for the mixture of dynamic operations). As shown, for larger values of C (i.e. when the number of deletions is much smaller than that of insertions), LR-tree clearly outperforms R*-tree. This result is in accordance with the previously described ones, since the impact of deletion is small. Focusing on smaller values of C (i.e. when the impact of deletions is significant), LR-tree still outperforms R*-tree.[7] This result indicates that the performance of LR-tree is not affected by the weak deletion operation, due to the good properties achieved by the insertion operations, which organize the LR-tree contents and eliminate any possible effects of the weak deletion operations. For the above experiment, this is also verified by the node utilization (ratio of fanout to maximum node capacity) achieved by the LR-tree. Even for lower values of C (i.e. more deletion operations), the average utilization was constantly higher than 95%, whereas for larger C values, the average was close to 98%. This is due to the kind of rebuilding applied by LR-tree, which packs the node contents.

We tested the scale-up properties of each method with respect to the dataset size, using synthetic datasets (points following uniform distribution). Figure 6b depicts the results for the range search query for an increasing number of points in the dataset (given in thousands). It has to be noted that the size of the LR-index in this measurement ranged from 5 MB

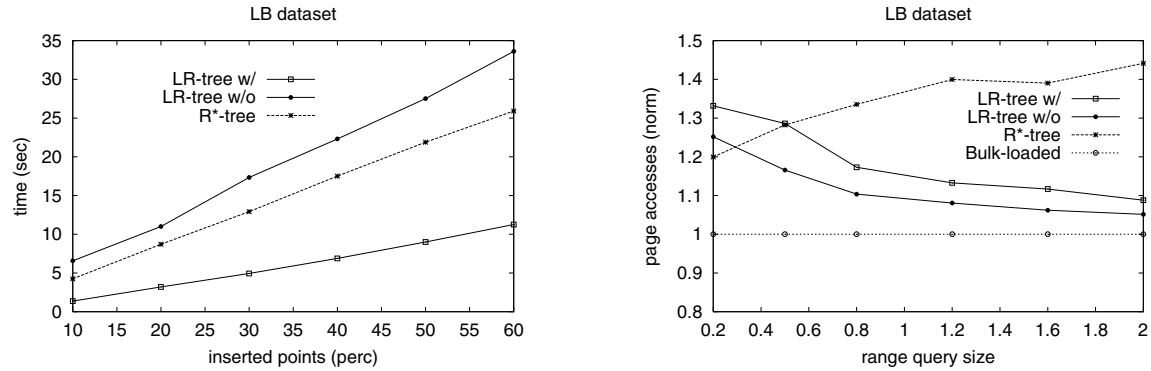
[6]The absolute number of insertions is constant, i.e. the 50% of the total number of points is inserted during the mix-operation phase. The absolute number of deletions can easily be derived, for example, when $C = 3$, it is one-third of the number of insertions.

[7]Disk accesses decrease with decreasing C because a smaller C value corresponds to more deletions and to fewer points in the tree, hence less disk accesses are required by the queries. Nevertheless, we are only interested in the relative performance of LR-tree and R*-tree for the different values of C.



(a) (b)

FIGURE 6. Performance with respect to (a) mixed operations and (b) scale-up.



(a) (b)

FIGURE 7. The heuristic adjustment. (a) Insertions. (b) Range queries.

(for 100,000 points) to 50 MB (for 1,000,000 points). As shown, LR-tree presents very good scalability, significantly outperforming R*-tree. In fact, LR-tree performance is comparable to that of bulk-loaded R-tree, which shows the best performance among the examined methods.

5.4. Heuristic adjustment

Finally, we examined a variation to the approach of wR-trees, called heuristic adjustment. The heuristic adjustment allows for a number of splits (user parameter) within a block during insertions. Assume that V_i is the block that can accommodate a new point during its insertion, as described in Section 2. Instead of destroying all previous data structures (i.e. all D_j wR-trees for $j \leq i$) and building a new one, the heuristic adjustment postpones the latter procedure. To handle the accommodation of the new point in the D_i structure, the heuristic adjustment may resort (when necessary) to the standard node splitting used by R-trees. Thus, blocks in this case are not strictly semi-dynamic data structures.

The aforementioned approach corresponds to a kind of *lazy rebuilding*, deferring the costly operation of destroying blocks and building a new one, without degrading performance. Evidently, following this approach, the time required for the insertion is expected to improve and, moreover, the performance of query processing will not be impacted significantly.

To verify the above statements, we performed measurements on the insertion time and query performance of the heuristic adjustment. We used a threshold value of 20 splits that are allowed before the destruction/rebuilding phase. For brevity we report results on the Long Beach real data set. Figure 7a depicts the insertion time with respect to the number of inserted points (given as a percentage of the total number of points in the data). In this figure, LR-tree w/ denotes the LR-tree with heuristic adjustment and LR-tree w/o the one without. As expected, the LR-tree with heuristic adjustment presents improved insertion times, since less rebuildings are required. Thus, it outperforms R*-tree and LR-tree without the heuristic adjustment.

Figure 7b depicts the results for the range query with respect to the query size (the query size is given in terms of percentage of the work space). Similar to the results in Figure 3, the results are the normalized page accesses with respect to the bulk-loading method. As shown, the performance of the LR-tree with the heuristic adjustment (LR-tree w/) is analogous to that of the LR-tree without this approach. Therefore, for medium and larger ones,

LR-trees (with and without the heuristic adjustment) clearly outperform R*-tree and their performance is close to that of the bulk-loading method.

5.5. Discussion

The presented results can be directly explained using Theorem 3. According to this, the insertion procedure is bounded by $O(\log_B n \log_b n)$ plus the search time for leaf detection; thus, it is a $\log_B n/b$ factor worse than that of R*-trees. However, with the heuristic adjustment, insertion times for LR-tree are improved significantly, outperforming R*-tree. Deletion is clearly logarithmic plus the time for item location and certainly better than that of R*-trees, whose time is a b factor larger due to forced re-insertions. In fact, if the correspondence between blocks and wR-trees is maintained using a B$^+$-tree $\mathcal{B}$, in which each element of S is kept along with a pointer to the specific wR-tree where it resides, deletion time can be strictly logarithmic, since the involved points can be located without any calls to the underlying wR-tree search procedure at the expense of increased storage requirements; this trade-off can be decided by considering the performance requirements that the corresponding application poses.

Finally, the various queries exhibit a performance that is not worse than R*-tree; actually, it is proven to be better, achieving percentage improvements of up to 27% for range searching, up to 23% for k-nearest-neighbor queries and up to 25% when it scales up. This performance is comparable to that of the static bulk-loaded indices, just as anticipated due to the 'tight' nature of the subtrees corresponding to the blocks. Here we must note that the static bulk-loaded R-trees are effectively the lower bound, since they exhibit a performance which is about 20% worse than the theoretical optimum on average [15].

6. CONCLUSION

We have presented a new scheme, called LR-tree, for the dynamic manipulation of large datasets, which offers the performance characteristics of bulk-loaded indices. The update operations as well as the basic geometric queries are theoretically investigated and thoroughly tested with carefully designed experiments, since there is truly a difficulty about expressing a non-trivial upper bound for R-tree-like spatial data structures. In order to achieve this, we employed the known logarithmic dynamization technique, from the area of main memory data structures, carefully adjusted to the demanding context of the R-trees, giving solutions to a number of issues like, for example, the definition of weak deletions or the multiplicity of activated search paths.

Independently to this work, the logarithmic method has also been used in [12] to accommodate—in conjunction to pagination—to the secondary memory the wp-trees, providing theoretical upper bounds without further experimental treatment. We, in addition to theoretical bounds, give for the first time experimental data for the logarithmic method, which is mandatory for the effective evaluation in real-world applications. The experimental results provide clear evidence of the superiority of LR-trees.

We believe that our work can be further improved in the following two ways. (a) The logarithmic block method actually bases its applicability on the constant amortized cost of the carry propagation. That is, when we insert a new point to the structure, the actions that follow are equivalent to adding one to the cardinality of the accommodated set of elements. There are redundant integer representation systems [19, 20] with the property that adding one involves constant *worst-case* carry propagation. In the context of the LR-tree this means that only $O(1)$ subtrees are involved in bulk-loading in the worst case. (b) Since the insertion procedure involves discarding a number of subtrees and bulk-loading from scratch the accommodated elements, one can hope to accelerate the whole process if it would be possible to exploit the 'order' the elements already possess. In other words, it would be quite interesting if one could merge R-trees in $O(n)$ or $o(n \log n)$ time, since bulk-loading has $O(n \log n)$ time complexity. We believe that this problem, having additional applications, is interesting on its own.

REFERENCES

[1] Guttman, A. (1984) R-trees: a dynamic index structure for spatial searching. In *Proc. 1984 ACM Int. Conf. on Management of Data (SIGMOD'84)*, Boston, MA, June 18–21, pp. 47–57. ACM Press, New York.

[2] Manolopoulos, Y., Theodoridis, Y. and Tsotras, V. (1999) *Advanced Database Indexing*. Kluwer Academic Publishers, Boston, MA.

[3] Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B. (1990) The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. 1990 ACM Int. Conf. on Management of Data (SIGMOD'90)*, Atlantic City, NJ, May 23–25, pp. 322–331. ACM Press, New York.

[4] Kamel, I. and Faloutsos, C. (1994) Hilbert R-tree: an improved R-tree using fractals. In *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB'94)*, Santiago de Chile, Chile, September 12–15, pp. 500–509. Morgan Kaufmann, San Francisco.

[5] Leutenegger, S., Lopez, M. and Edgigton, J. (1997) STR: a simple and efficient algorithm for R-tree packing. In *Proc. 13th Int. Conf. on Data Engineering (ICDE'97)*, Birmingham, UK, April 7–11, pp. 497–506. IEEE Computer Society, Piscataway, NJ.

[6] Bercken, J., Widmayer, P. and Seeger, B. (1997) A generic approach to bulk loading multidimensional index structures. In *Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB'97)*, Athens, Greece, August 25–29, pp. 406–415. Morgan Kaufmann, San Francisco.

[7] Arge, L., Hinrichs, K., Vahrenhold, J. and Vitter, J. (1999) Efficient bulk operations on dynamic R-trees. In *Proc. Int. Workshop on Algorithm Engineering and Experimentation, (ALENEX'99)*, Baltimore, MD, January 15–16, pp. 328–348. Springer, Berlin.

[8] Berchtold, S., Böhm, C. and Kriegel, H.-P. (1998) Improving the query performance of high dimensional index structures by bulk load operations. In *Proc. 6th Int. Conf. on Extending*

Database Technology (EDBT'98), Valencia, Spain, March 23–27, pp. 216–230. Springer, Berlin.

[9] Bentley, J. L. and Saxe, J. B. (1980) Decomposable searching problems I: static-to-dynamic transformations. *J. Algorithms*, **1**, 301–358.

[10] Mehlhorn, K. and Overmars, M. H. (1981) Optimal dynamization of decomposable searching problems. *Inform. Process. Lett.*, **12**, 93–98.

[11] Overmars, M. H. (1983) *The Design of Dynamic Data Structures*. Springer, Berlin.

[12] Agarwal, P. K., Arge, L., Procopiuc, O. and Vitter, J. S. (2001) A framework for index bulk loading and dynamization. In *Proc. 28th Int. Conf. on Automata, Languages and Programming (ICALP'01)*, Crete, Greece, July 8–12, pp. 115–127. Springer, Berlin.

[13] Huang, Y.-W., Jing, N. and Rundensteiner, E. A. (1997) Spatial joins using R-trees: breadth-first traversal with global optimizations. In *Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB'97)*, Athens, Greece, August 25–29, pp. 396–405. Morgan Kaufmann, San Francisco.

[14] Lang, C. A. and Singh, A. K. (2001) Modeling high-dimensional structures using sampling. In *Proc. 2001 ACM Int. Conf. on Management of Data (SIGMOD'2001)*, Santa Barbara, CA, May 21–24, pp. 389–400. ACM Press, New York.

[15] Pagel, B.-U., Six, H.-W. and Winter M. (1995) Window query-optimal clustering of spatial objects. In *Proc. 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS'95)*, San Jose, CA, May 22–25, pp. 86–94. ACM Press, New York.

[16] Papadopoulos, A. and Manolopoulos, Y. (1997) Performance of nearest-neighbor queries in R-trees. In *Proc. 6th Int. Conf. on Database Theory (ICDT'97)*, Delphi, Greece, January 8–10, pp. 394–408. Springer, Berlin.

[17] Johnson, T. and Shasha, D. (1993) B-trees with inserts and deletes: why free-at-empty is better than merge-at-half. *J. Comput. Syst. Sci.*, **47**, 45–76.

[18] Berchtold, S., Böhm, C., Keim, D. A. and Kriegel, H.-P. (1997) A cost model for nearest neighbor search in high-dimensional data space. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS'97)*, Tucson, AZ, May 12–14, pp. 78–86. ACM Press, New York.

[19] Clancy, M. J. and Knuth, D. E. (1997) *A Programming and Problem Solving Seminar*. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, Stanford, CA.

[20] Kaplan, H. and Tarjan, R. E. (1999) *New Heap Data Structures*. Technical Report TR-597-99, Department of Computer Science, Princeton University, Princeton, NJ.