Merging R-trees

Vasilis Vasaitis
Dept. of Informatics
Aristotle University
Thessaloniki, 54006, Greece
vvas@hal.csd.auth.gr

Alexandros Nanopoulos
Dept. of Informatics
Aristotle University
Thessaloniki, 54006, Greece
ananopou@egnatia.ee.auth.gr

Panayiotis Bozanis
Computer & Commun. Eng. Dept.
University of Thessaly
Volos, 38221, Greece
pbozanis@inf.uth.gr

Abstract

R-trees, since their introduction in 1984, have been proven to be one of the most well-behaved practical data structures for accommodating dynamic massive sets of geometric objects and conducting a diverse set of queries on such data-sets in real-world applications. In this paper we introduce a new technique for merging two R-trees into a new one of very good quality. Our method avoids both the employment of bulk insertions and the solution of bulk-loading, from scratch, the new tree using the data of the original trees. Additionally, unlike previous approaches, it does not make any assumptions about data-set distributions. Experimental results provide evidence on the runtime efficiency of our method and illustrate the good query performance of the produced indices.

1. Introduction

During the last years, several applications appeared that call for the efficient manipulation of massive sets of geometric objects like points, lines, areas or volumes in one or more dimensions; this is particularly evident in Geographical Information Systems, in CAD/VLSI design, and in mobile object movement detection and prediction. Databases that accommodate this kind of objects are called *spatial* and they employ data structures capable of answering various *geometric queries*, like *range queries* that ask for all objects lying within a given region, or *nearest-neighbour queries* that seek the object closest to a given object, or *join queries* that question for all pairs of objects satisfying a given predicate on the set of objects that they accommodate.

The "diverse query types" requirement, combined with the massive volume of the involved data-sets, was the main reason why practical, general-purpose data structures are in fact used in almost all real-world applications. This also justifies why R-trees [13] became so popular from the research point of view as proved by the so many variants [20]. In short, an R-tree is a height-balanced tree similar to a B^+-tree; actually, it can be considered 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 subtree T, MBR of T), with the MBR of a tree T defined as the MBR enclosing all the MBRs stored in 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 linear to the size of data-set performance just to retrieve a few objects.

Each one of the proposed variants of the R-tree aims at improving the performance by adjusting some parameters. For example, in R*-trees of Beckmann et al. [3], which are widely accepted as achieving the best performance, a number of heuristics were proposed, 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. Still, in spite of the intensive research, 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, there are numerous applications that generate large amounts of data. That data must either be accommodated in an existing index or be used to build a new index from scratch. The first problem is known as *bulk insertion*, while the second one as *bulk-loading*. Finally, there exists a third case of dealing with large amounts of data, in which one needs to unify data-sets indexed by separate (auxiliary) data structures into a single indexed data-set. This operation is usually called *merging*. In this paper, we deal with merging.

While the problem of merging is a common one in main memory data structures, like, for example, priority queues

or red-black trees [11, 21], there exists limited research in the database area. Actually, to our knowledge, the only works dealing with merging database indices —R-trees, to be specific— in a rather restricted context are those of Rundensteiner et al. [7, 8, 9], despite the practical nature of the operation. For instance, consider the situation of the consolidation of two companies; it is reasonable for one to expect that the existing (separate) spatial indices must be merged. Or the case where a research institute department maintains geological data-sets concerning adjoining areas which must be combined into larger ones for research purposes.

One can argue that the problem of merging can be treated with either bulk-inserting the data of one tree into the other or just bulk-loading, from scratch, a new tree using simply the data of the two old ones. Both actions ignore the useful information that the existing trees already carry. Furthermore, superlinear time complexity is needed to accomplish the task in either case. These observations were the starting point of the present work, which aimed at devising an efficient merging algorithm for R-trees, not only from the time complexity perspective though; additionally, the produced indices are well-behaved.

The rest of the paper is organized as follows: Section 2 presents the (closely) related problems of bulk- insertion/loading while discussing the previous solution of Rundensteiner et al. In Section 3 we present the algorithms for merging two R-trees. Section 4 gives the performance study, while Section 5 concludes our work.

2. Related work

Various techniques for R-trees have been introduced which, by exploiting an a priori knowledge of the static data-sets, build the structure from scratch and achieve better utilization and search performance in the average case than construction by repeated insertions would achieve. The majority of them apply a pre-sorting on data, according to some total order, and then bulk-load the index: [24] was one of the first attempts on bulk-loading R-trees. In [16] a Hilbert sorting technique was used to first sort data before building the R-tree. [18] 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. In [4, 6] 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. Here we must note that bulk-loading algorithms for other categories of indices than R-trees do exist; the interested reader could, for instance, consult [10, 12, 14, 15].

As for bulk insertions in R-trees, [5, 2] proposed the building of indices with repeated block-wise insertion, where buffers are attached to index pages. During the insertion, 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 leaf level is reached. Alternative approaches, based on sorted-wise insertions, were presented in [17, 22, 23]. Finally, [7, 8] presented algorithms for merging two R-trees, in the context of bulk insertions, under rather strict conditions: When one wants to bulk-insert a moderate number of new data, one constructs a small R-tree and then tries to identify a suitable index entry in the existing large tree, into which the root of the small one is inserted, and then handles the overflow that may occur. As the authors state, their approach, termed Small-Tree-Large-Tree (STLT), "*... is a solution for bulk insertion of skewed data sets but not for non-skewed data insertions*". The STLT approach, which relates remotely to the Seeded R-tree spatial joins [19], was also employed to cope with bulk insertions in [9], where the GBI method is introduced, which, firstly, finds the clusters of the input set with a k-means clustering algorithm, and then repeatedly applies STLT to accommodate the identified clusters.

In the following section we will present our solution, which, given two R-trees, constructs a new one, their union, by a linear top-down traversal of the smallest one. Our approach does not make any assumptions on data distribution; it just capitalizes on the information that the input R-trees carry: whole subtrees of one of the input trees are accommodated whenever possible, while the entirety of the tree can be inserted as a single entry in the ideal case. This process is driven by applying certain criteria for deciding whether each handled subtree is going to be left intact or decomposed to its individual entries.

3. Description of our method

3.1. Prerequisites

In this section we will describe a series of individual algorithms that, when combined together, accomplish merging two R-trees into one. However, before we can do that, it is necessary to lay the groundwork, to explain the concepts that will aid the comprehension of all that follows.

The merging is performed in an asymmetric manner. That is, the two trees are not treated equally during the process; on the contrary it is always assumed that one of them is to be inserted into the other. Still, the algorithms are defined in such a way that it is perfectly possible for any R-tree to be inserted in any other, so that the role of the two trees is chosen based solely on performance considerations. The tree on which the insertion is performed and the tree which is being inserted will henceforth be called the *receiving tree* and the *giving tree*, respectively.

Now, in order to insert one of the trees into the other, the availability of certain auxiliary structures in external memory is deemed necessary. More specifically, two buffers need to be attached to every node of the receiving tree that is accessed during the merging process. Note that they are not required to have been allocated beforehand, but instead they can be created and reused as needed. In addition, it should be noted that these are variable-size buffers, expanded and contracted according to the number of elements that are stored within. These two buffers will henceforth be called the *insertion queue* and the *local insertion queue*. In particular, the insertion queue of a node may store entries that are destined to the node itself or to its children, while the local insertion queue may only store entries that are destined to the node itself. It should be obvious from their definition that the two queues are one and the same on the leaves.

During the merging process, there are certain queues that have to be accessed at any time. In particular, the insertion queue of the current node is read from and written into, the insertion queues of the child nodes are written into, and the local insertion queue of the current node is written into as well. Queues are read from sequentially, from the beginning to the end, and are always written into beyond their end, where new entries are appended. Thus, if M is the maximum number of entries inside a node, at least $M+3$ pages of main memory need to be available for the queues to be accessed in an optimal fashion: 2 pages for the node's insertion queue, M for the child nodes' insertion queues, and 1 for the node's local insertion queue. In other words, one page is needed for reading, storing the page that contains the current entry, and one page for writing, storing the tail of each queue.

All this means that merging the trees requires more memory compared to the elementary R-tree algorithms, which only need one page at a time that stores the current node. In addition, in accordance with those algorithms, it is preferred to keep the entire current path of the receiving tree in main memory, thus increasing its usage somewhat more. If the total available memory is more than the minimally required amount (and it usually is), the rest can be used as a cache for the nodes of the giving tree as well as for the various queues. On the other hand, it is not necessary to cache the nodes of the receiving tree, since each of them is accessed at most once during the merging process.

After covering these prerequisites, we can now proceed with the description of the algorithms that result in the merging. We will work our way bottom-up, gradually building the whole out of the parts. Therefore each algorithm presented will only depend on the ones that will have already been presented, so as to assist its comprehension.

3.2. Generalised split

The splitting algorithm used for the merging of the trees is the split of the R*-tree [3]. As with all splitting algorithms that have been proposed for the R-tree and its variants, the algorithm separates the $M+1$ entries of an overfull node into two parts. No other quantity of entries needs to be taken into account, since the elementary insertion algorithm always inserts one element at a time.

However, during merging there is a significant quantity of elements that are inserted to the receiving tree, and the way the algorithms operate makes it perfectly possible, and likely, that multiple elements are inserted into one node at the same time. Therefore, the split of a node needs to be generalised, in order to be able to handle nodes with an arbitrary number of entries.

Fortunately, such a generalization is not very hard to conceive; all we have to do is redefine the minimum and maximum number of entries that each of the two resulting nodes may receive. It is reminded here that, when dealing with $M+1$ entries, each of the resulting nodes may receive from m to $M+1-m$ out of those, where m is the minimum number of entries that a node can contain. So, in the generalised case of L entries contained inside the overfull node, we need to find the minimum l and maximum $L-l$ number of entries that each of the new nodes may contain.

One of the simplest solutions to this problem, which is also the one that was chosen as the preferred one, is to keep the minimum number of entries in the new nodes and the total number of entries of the overfull node under a constant ratio, and make this ratio be the same as in the original, non-generalised case. Or, in other words, $l = \left\lfloor \frac{Lm}{M+1} \right\rfloor$. This way, the original case remains as it is, and after that there is a gradual increase of the minimum value.

All that remains to be explained is how this generalised split behaves with regard to the amount of available main memory. That is, in order to function properly, it first has to be able to load into main memory all the entries which are destined to the node that is being operated upon. Fortunately, thanks to the minimum requirements of the whole process that have already been presented, it will almost always be possible to do exactly that. Nevertheless, when this is not the case, the following method is used: The maximum possible number of entries is loaded into main memory, and the split is performed based on those. Then the rest of them are distributed among the two new nodes, according to the well-known criterion of minimising the area of the minimum bounding rectangle (MBR) of the nodes.

3.3. Multiple split

Having defined the generalised split, we quickly notice the following: because the node that is to be split may con-

tain an arbitrary number of entries, likewise the resulting nodes may, too, contain an arbitrary number of entries; thus they can be overfull themselves, just like the original node. So the splitting process has to work in such a way, that in the end all we are left with are valid nodes, nodes with an acceptable number of entries.

Same as before, an extremely simple approach turns out to be quite adequate: execute recursively the generalised split for the nodes that it itself produces, until all the nodes that remain contain no more than M entries. Thus works the multiple split algorithm, as can be seen in Figure 1, so that an overfull node can be split into multiple new nodes, to the degree that something like that is necessary, of course.

If the current node contains more than M entries:
Execute the generalised split for the node.
For each of the two resulting nodes:
Execute the multiple split.

Figure 1. Multiple split

Naturally, the exact interaction of the algorithm with external memory has to be explained as well. First of all, if the entries belonging to the current node are already in main memory, or it is possible to load all of them into it (meaning, those contained in the node itself, as well as those contained in the node's local insertion queue), then the whole splitting process can continue in main memory, with all the resulting nodes being written out to external memory at the end. In practice, this is exactly what will happen in most cases. However, in the rare case when there is not enough main memory available, the generalised split is performed for as many entries as possible, with the rest of them being distributed among the resulting nodes afterwards, as explained in the previous subsection. Note that, in this case, only the local insertion queues of the resulting nodes need to be created and not the nodes themselves, because it is certain that the nodes will be overfull as well, thus requiring further splits.

3.4. Tree insertion

We are now ready to describe the main algorithm of the merging, the tree insertion. This algorithm is presented here in a recursive form, where each instance always operates on one particular node of the receiving tree. The insertion begins at the root of the tree, and recursively descends it until it reaches the leaf level. With this algorithm however, and unlike the R-tree single element insertion, multiple paths can be followed inside the tree, since a massive quantity of elements is inserted, organised as an already existing tree.

The general idea behind the algorithm is the following: The algorithm attempts to insert whole subtrees of the giving tree whenever possible, while the entirety of the giving tree can be inserted as a single entry in the ideal case. Nevertheless, in each node of the receiving tree for which the algorithm is executed, certain criteria are used to decide whether each handled subtree of the giving tree is going to be left intact or decomposed to its individual entries, which can point to smaller subtrees or single elements. The criteria used are the following:

Area criterion. If the subtree is destined to a lower level of the receiving tree than the current one, the following procedure is followed: The subtree is routed to the child node whose MBR admits the least area enlargement to include it, and that enlargement is recorded. Then, each of the individual entries of the subtree is likewise routed to the child where it causes the least area enlargement, and the sum of area enlargements caused to each child (*not* the sum of area enlargements caused by each entry) is recorded as well. If the first recorded value is less or equal to the second, then the subtree is propagated as a whole to the suitable child; otherwise, it is decomposed to its entries.

Overlap criterion. If, on the other hand, the subtree is destined to the current level, thus to the current node, a different approach is followed: First, the overlap enlargement that would be caused to the current node, if the subtree were inserted into it, is recorded. Then, just like in the previous case, the entries of the subtree are distributed to the children where they would cause the least area enlargement, and the overlap enlargement that is caused to *the current node* by such a distribution is recorded. If the first recorded value is less or equal to the second, and additionally less or equal to the area of the subtree's MBR, then the subtree is inserted as a whole to the current node; otherwise, it is decomposed to its entries.

It needs to be mentioned that, when a subtree is decomposed to smaller subtrees, the latter are then examined according to the above criteria *inside the same node*; that is, they are not automatically forwarded to the next level.

Figure 2 presents the tree insertion algorithm. It should be noted that, wherever inside the algorithm's description there is a reference to the suitable child to receive an entry, the child whose MBR needs the least area enlargement to include the entry's MBR is actually referred to. This definition is kept out of the figure in order to save space and avoid undue repetition.

3.5. Tree merging

All that remains to describe is the tree merging algorithm itself, which, directly or indirectly, uses all the other algo-

Beginning with the root of an R-tree as the current node C, insert all the entries that are contained in the insertion queue of C.

If C is not a leaf:
- For each entry E in the insertion queue of C:
  - If E refers to a single element:
    - Insert E into the insertion queue of the suitable child of C.
  - If E refers to a subtree:
    - If *E/subtree* has a smaller height than the level of C, and the area criterion is satisfied:
      - Insert E into the insertion queue of the suitable child of C.
    - If *E/subtree* has an equal height to the level of C, and the overlap criterion is satisfied:
      - Insert E into the local insertion queue of C.
    - If *E/subtree* has a greater height than the level of C, or contains less than m entries, or none of the two above conditions were satisfied:
      - Insert all the entries of *E/subtree* into the insertion queue of C.
- For each entry E in T:
  - If the insertion queue of *E/child* is not empty:
    - Recursively execute the algorithm for $C \leftarrow$ *E/child*.

Execute the multiple split for C.

If new sibling nodes have been created because of splits:
- Insert suitable entries for them into the local insertion queue of the parent of C.

Figure 2. Tree insertion

rithms that have been described so far. This algorithm accepts two R-trees as its input, and produces a new one as its output, which contains the union of the elements of these two trees. To be exact, after the algorithm has finished its execution, one of the two input trees (the giving tree) has become empty, while the other (the receiving tree) has been converted to the output tree. The algorithm is presented in Figure 3.

Choose the tree with the greatest height as the receiving tree. If both tree have the same height, choose the one with the largest number of elements.

Insert the root of the giving tree into the insertion queue of the root of the receiving tree.

Execute the tree insertion for the root of the receiving tree.

While there are new nodes that have been created because of splitting the root of the receiving tree:
- Create a new root for the receiving tree.
- Insert the old root as well as the new nodes into the local insertion queue of the new root.
- Execute the tree insertion for the new root.

Figure 3. Tree merging

It is worth mentioning that, at the final stage of the algorithm's execution, the height of the receiving tree might need to be increased more than once. Therefore this stage is treated in an iterative fashion, until there is no longer a need for further expansion of the tree. Otherwise, the operation of the algorithm does not require any particular clarifications.

Thus ends the description of the algorithms that accomplish merging two R-trees. In the following section, an experimental evaluation of their performance will be presented. Before that though, this section will be concluded with a brief note about the R*-tree, a popular variant of the R-tree, and its relation to these algorithms.

3.6. Merging R*-trees

Since R*-trees have the same internal structure as R-trees, the algorithms that have been presented in this section can be applied on any of these two variations, without the slightest change in their behaviour. In fact, all the improvements that the R*-tree introduces are contained in the element insertion process, and in particular, these improvements are: (a) a different strategy for choosing the best leaf an element should go to, (b) forced reinsertion of elements before attempting to split, to achieve global reorganization of the data, and (c) a wholly new node splitting algorithm. The reader is referred to [3] for the details.

It has already been stated that the tree merging process utilises the R*-tree split, since it behaves a lot better, producing nodes with much less overlap. The other two methods, on the other hand, are deliberately ignored and excluded from the process, for the following reasons:

Reinsertion, to begin with, is very closely tied to the no-

tion of a single element being inserted at a time. Thus, it would be quite difficult to generalize it in order to fit in our framework, where a massive quantity of elements is inserted at a time, and multiple node splits can occur on each level of the tree.

Now, when it comes to choosing a suitable leaf for an element, the situation is somewhat different. This time, it would be perfectly possible to adopt the R*-tree approach, and try to minimize the overlap enlargement when choosing the right leaf. However, because of the significantly increased CPU cost associated with this calculation, and the massive quantity of elements that can be subjected to it while merging, using this strategy turns out to be practically infeasible, since it would completely monopolize the running time of the merging process.

4. Experimental evaluation

4.1. Test setup

In this section we present a series of tests that demonstrate the behaviour of the proposed method. Since there is no direct alternative to our method of merging two R-trees, a comparison is made against discarding the existing structure and using the union of the data-sets of the two trees to bulk-load a new tree from scratch. For that purpose, the bulk-loading algorithm presented in [4, 6] is used. The same algorithm is also used for constructing the two trees that are merged out of their respective data-sets.

Both construction time and query performance of the resulting trees are measured. More specifically, the methodology employed is the following: For each of the tests, two data-sets are to be unified. Initially, two R-trees are constructed out of them using bulk-loading. Then the time needed to merge the two trees is measured, as well as the time needed to construct a new tree from scratch, using bulk-loading, out of the union of the data-sets. To measure query performance, 500 window queries are performed on each of the two resulting trees, and the average number of external memory accesses needed is measured. Six different window sizes are utilised: 0.01%, 0.02%, 0.05%, 0.1%, 0.2% and 0.5%.

All tests were conducted on a PC with the AMD-K6-2/500MHz processor and 768MB of main memory, under the Debian GNU/Linux operating system. The R-tree structure and all the relevant algorithms are implemented in C++ and compiled with g++ (GCC's C++ compiler).

As far as the tree parameters are concerned, the maximum number of entries per node M is page size specific, which in turn is data-set specific, chosen so that the resulting trees are neither too deep nor two shallow. Nevertheless, the minimum number m is always set to 40% of the maximum, while for bulk-loading the initial space utilization of

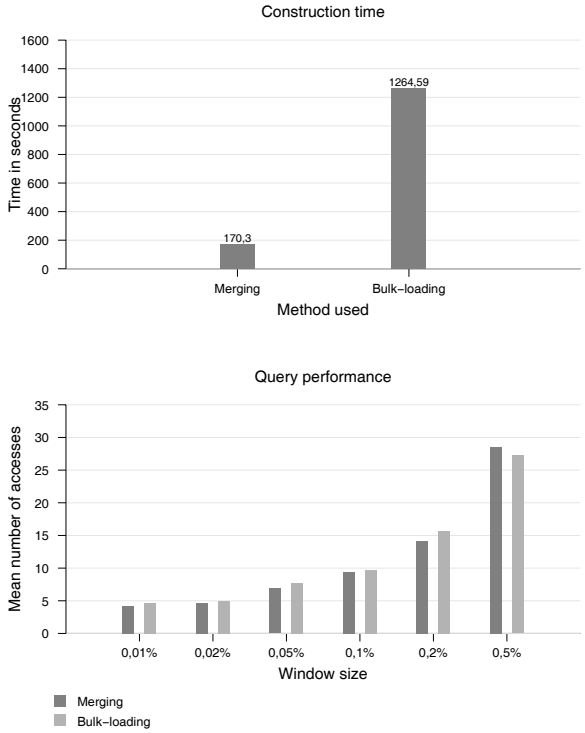

Figure 4. rivers + roads

the tree is set to 70%. The amount of main memory available is always enough to hold about 5–10% of the available data, depending of their total amount but regardless of the way they are actually stored on disk.

4.2. Test results for real data

The real data we used are geographical data of Greece, taken from the R-tree portal [25]. In particular, the data-sets of the roads, rivers and lakes of Greece were used, all of which contain two-dimensional, non-point data. The data contained therein are far from uniform, and pose quite a challenge to the algorithms involved.

For the first test presented here, the data-sets of rivers and roads were used, which have more or less the same number of elements (24650 rectangles of rivers and 23268 rectangles of roads). Since the data-sets are also situated in the same area, such a test turns out to be a worst case scenario for the merging algorithms, since it results to the complete decomposition of the giving tree during the merging process. Thus its also ideal for comparing the performance of merging compared to other alternatives. The results of the test are shown in Figure 4.

It is quite obvious that merging is significantly more efficient compared to bulk-loading, even for a worst case like

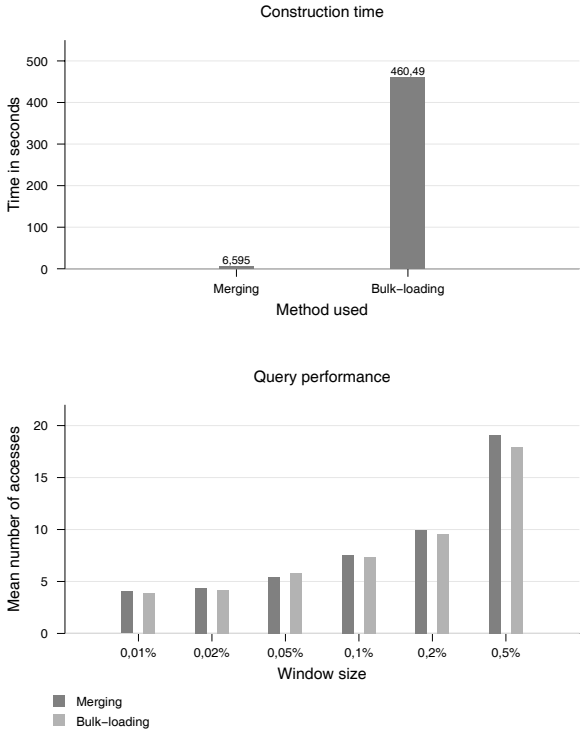

Figure 5. rivers + lakes

this one, as the construction time for merging is almost an order of magnitude smaller than for bulk-loading. On the other hand, the query performance of the resulting trees does not seem to differ much, if at all.

The other test that was performed with real data involves merging the rivers and lakes data-sets. This time, one of the two data-sets is extremely small compared to the other, since there are only 77 lake rectangles. This test resulted in the diagrams that are shown in Figure 5.

This time, the difference in performance is spectacular. While bulk-loading requires building the whole tree structure from scratch, merging simply routes the lake rectangles to the right position, inside the existing tree of rivers. Moreover, just like before, there is not much difference in the quality of the resulting trees, since queries once again need to access the same number of pages, more or less.

4.3. Test results for synthetic data

All the synthetic data-sets that have been used for testing are composed of two-dimensional rectangles (that is, non-point data) that are uniformly distributed inside a square area. The size of the rectangles is uniformly distributed as well. Even though uniform data are not much of a challenge for the various algorithms that operate on the R-tree, especially compared to real data, they allow us to measure the behaviour of the algorithms relative to various parameters. To be exact, four such parameters were measured during our tests and are presented here. Each of them is examined for five different values, all of which are presented together, in order to aid drawing the relevant conclusions.

All of these four different cases share the worst test they end up to, the one that is most difficult for the merging process. Thus, for all of them, the last test (rightmost in all diagrams) consists of two data-sets that occupy the same area, contain 1000000 rectangles in total (500000 each), and have a rectangle density of 100%, meaning that, on average, all the available space is occupied by one rectangle. For each case then, the chosen parameter is what defines how the rest of the tests deviate from that one.

It should be noted that, for the synthetic tests, the results of the query performance measurements are shown only for two of the six window sizes, namely for 0.01% and 0.1%. This is done in order to save space in the diagrams, but also because all these measurements lead us to the same conclusion: that there is no noteworthy difference in quality between the trees produced by merging and bulk-loading.

As a first case, the performance of the algorithms was measured for a variable number of elements for both data-sets, with the two of them being equal to each other in size. Thus the total number of elements varies from 20% to 100% of the maximum 1000000 rectangles, with a corresponding variance to the size of the containing area. The results of the measurements are shown in Figure 6.

As can be seen in the results, bulk-loading performs more or less as expected from its $O(n \log n)$ complexity. On the other hand, merging exhibits an essentially linear behaviour, which means that its already obvious performance advantage only gets bigger as the trees grow in size. At the same time, there is no perceivable difference in quality among the trees produced by the two methods.

Moving on to another case, this time the size of both the data-sets and the containing area remain constant. However, the two data-sets are no longer contained in the exact same area, but one of the areas is moving along both axes, so that their overlap varies from 20% to 100% of their total area. The results of tests for the various settings of overlap are shown in Figure 7.

As was expected, bulk-loading is not affected from the overlap of the data-sets, since it unifies them before constructing the tree. On the contrary, merging takes advantage of the presence of non-overlapping areas, since they allow it to insert a lot of subtrees of the giving tree without decomposing them. Once again, query performance is not affected; the deviation at 40% seems like an isolated case, which is not enough to cause any worries.

The next case is the only one where the number of elements for the two data-sets is not equal, but one of them

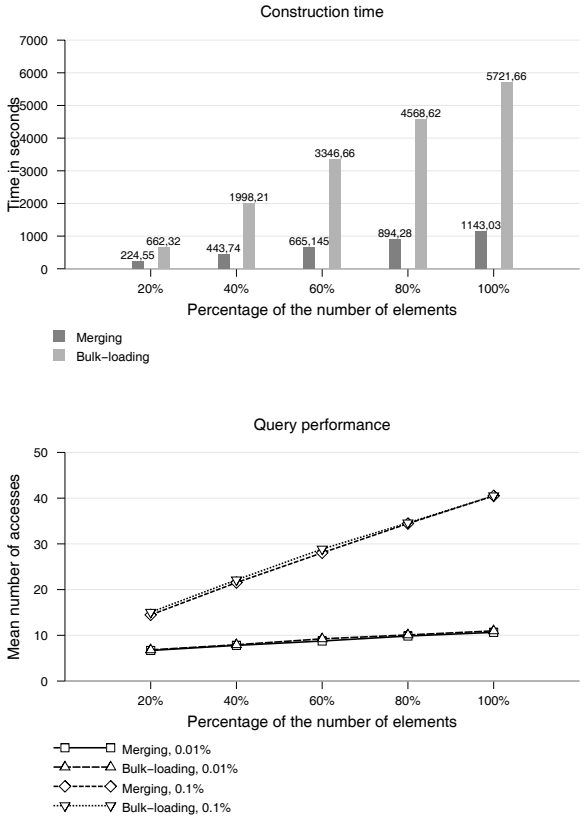

Figure 6. Number of elements

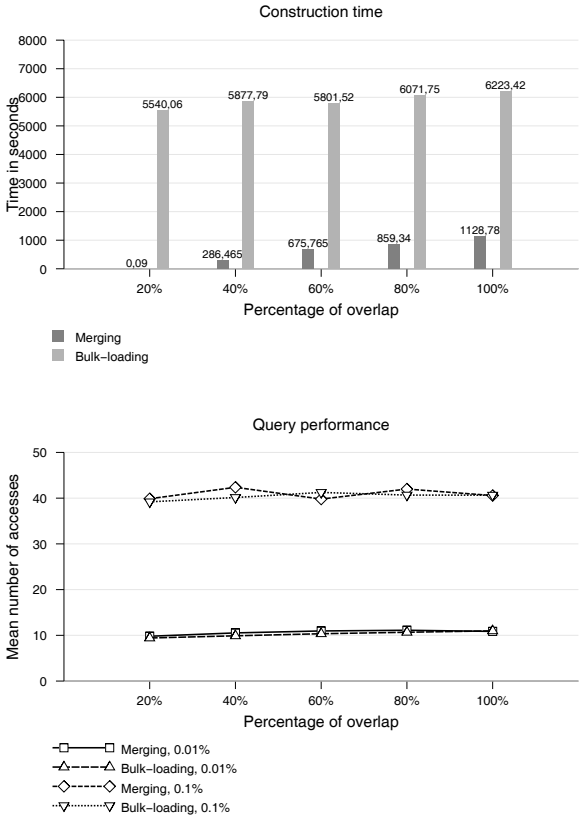

Figure 7. Overlap of the areas

contains from 20% to 100% of the number of elements of the other one. There is a similar variance for the containing area of this data-set, which is always situated inside the containing area of the larger data-set though, so that, for each of the two axes, its centre is situated at one third of the distance between the limits of the larger area. The measurements that result from such a set-up are shown in Figure 8.

Here we observe that the construction time for merging is slightly more than linear to the size of the smaller data-set, and thus only slightly worse to varying the size of both data-sets. On the other hand, the construction time of bulk-loading depends on the sum of the sizes of both data-sets, as was expected. Furthermore, yet again query performance is not affected much by the method used.

Finally, a batch of tests was performed in order to examine whether the algorithms are affected by the density of the rectangles inside the available space. By varying the size of the containing area, tests were performed with densities from 20% to 100%. However, as can be seen in Figure 9, this particular parameter does not seem to affect either method.

Afterwards, all tests with synthetic data were repeated for three, four and five dimensions as well. As an indication of the results, diagrams are included for the third case of our tests, in three and four dimensions, as shown in Figure 10 and Figure 11, respectively. In general, the findings were similar, so they are omitted due to limited space.

5. Conclusions

In this paper we present a novel method for merging two R-trees into a new one of very good quality. Unlike generic alternatives like bulk-loading and bulk insertion, our technique takes advantage of the existing structure of both of the trees involved in the operation. In addition, unlike other solutions such as the Small-Tree-Large-Tree (STLT) approach, no assumptions are being made about data-set distributions. Instead, our method attempts to accomodate whole subtrees of one the two trees into the other, inserting them in a way analogous to single-element insertion. Simple, well-known criteria of area and overlap enlargement are used to decide whether each subtree is left intact or decomposed to its individual entries. In order to handle the massive quantity of data that might occur out of this de-

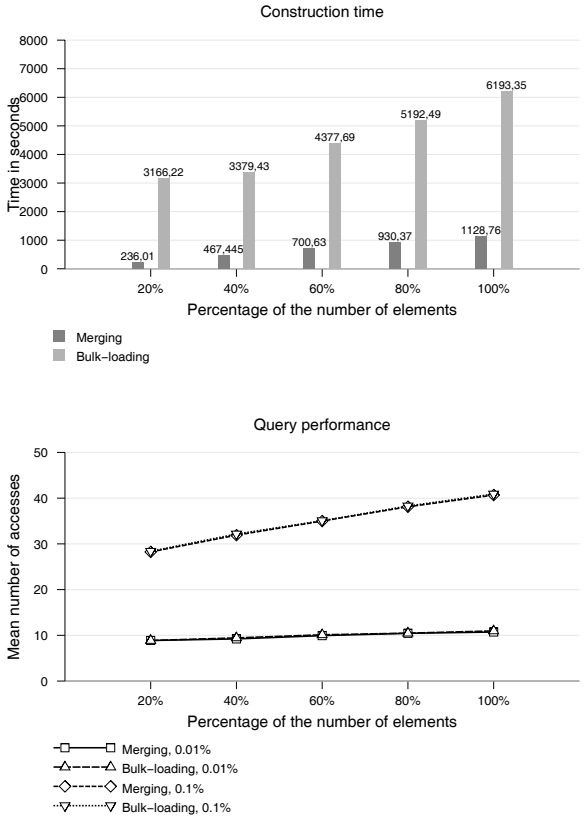

Figure 8. Ratio of the data size

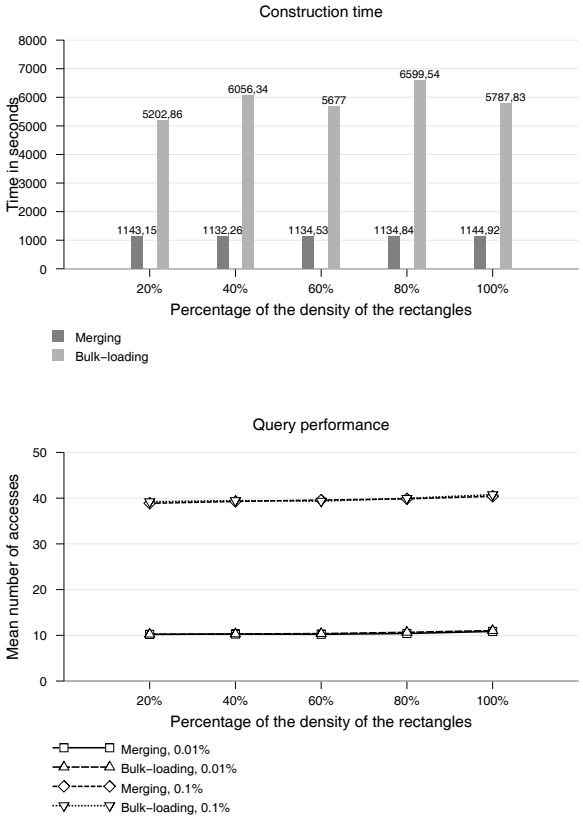

Figure 9. Density of the rectangles

composition, dynamic, variable-size buffers are employed, while the node-splitting algorithm is extended to handle an arbitrary number of elements. The experimental evaluation of our method indicates that it is very efficient speed-wise, and also that the resulting trees maintain an excellent query performance.

Future work on the subject will concentrate, among others, on the following matters: First of all, finding ways to better take advantage of the available main memory during the merging process. In addition, we would like to research the correlation between our method and buffer-based bulk insertion techniques, compare their performance, and examine whether it is possible to combine the best of both worlds.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc and J. S. Vitter, "A Framework for Index Bulk Loading and Dynamization," in *Proc. ICALP*, 2001, pp. 115–127.

[2] L. Arge, K. Hinrichs, J. Vahrenhold and J. Vitter, "Efficient Bulk Operations on Dynamic R-tress," in *Proc. ALENEX*, 1999, pp. 328–348.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles," in *Proc. SIGMOD*, 1990, pp. 322–331.

[4] S. Berchtold, C. Böhm and H.-P. Kriegel, "Improving the Query performance of High Dimensional Index Structures by Bulk Load Operations," in *Proc. EDBT*, 1998, pp. 216–230.

[5] J. Bercken, P. Widmayer and B. Seeger, "A Generic Approach to Bulk Loading Multidimensional Index Structures," in *Proc. VLDB*, 1997, pp. 406–415.

[6] C. Böhm and H.-P. Kriegel, "Efficient Bulk Loading of Large High Dimensional Indexes," in *Proc. DaWaK*, 1999, pp. 251–260.

[7] L. Chen, R. Choubey and E. A. Rundensteiner, "Bulk-Insertions into R-trees Using the Small-Tree-Large-Tree Approach," in *Proc. ACM GIS*, 1998, pp. 161–162.

[8] L. Chen, R. Choubey and E. A. Rundensteiner, "Merging R-trees: Efficient Strategies for Local Bulk Insertion," *GeoInformatica*, vol. 6, pp. 7–34, March 2002.

[9] R. Choubey, L. Chen and E.A. Rundensteiner, "GBI: A Generalized R-Tree Bulk-Insertion Strategy," in *Proc. SSD*, 1999, pp. 91–108.

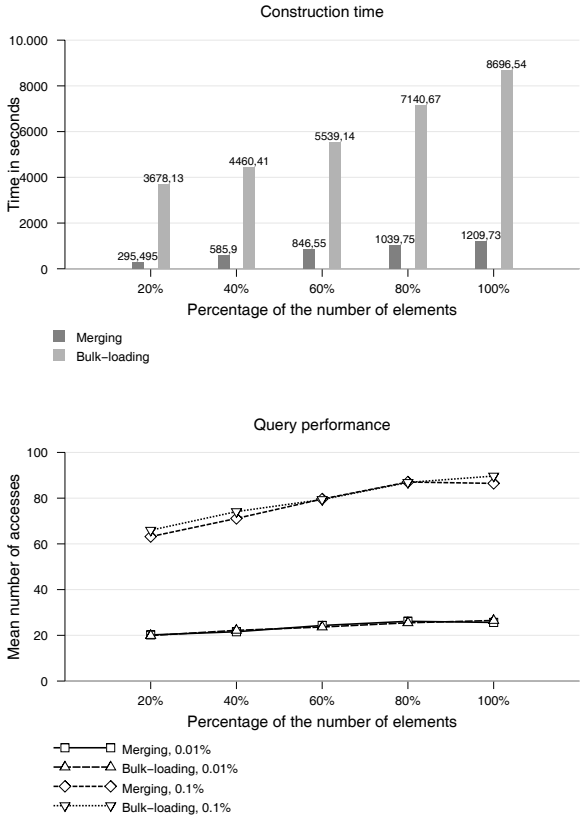

Figure 10. Ratio of the data size, 3D

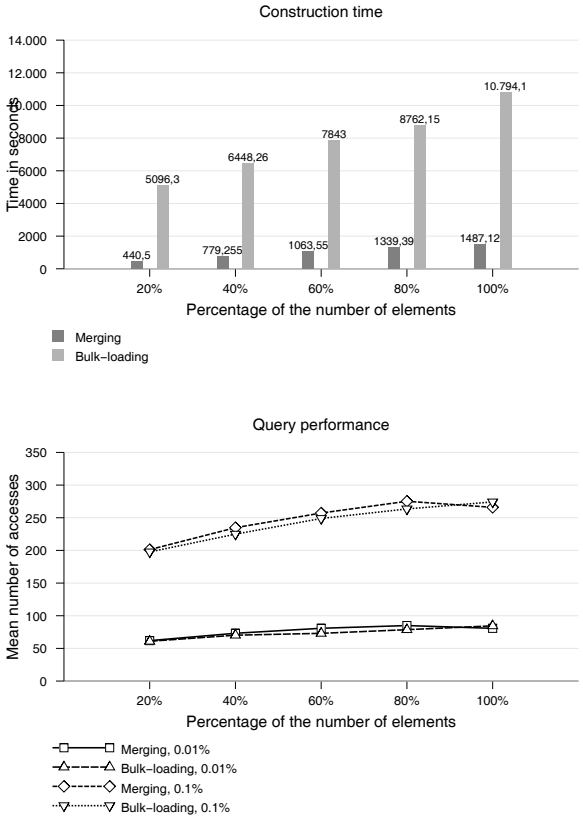

Figure 11. Ratio of the data size, 4D

[10] P. Ciaccia and M. Patella, "Bulk Loading the M-tree," in *Proc. ADC*, 1998, pp. 15–26.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press and McGraw-Hill Book Company, NY 2001.

[12] T. M. Ghanem, R. Shah, M. F. Mokbel, W. F. Aref and J. S. Vitter, "Bulk Operations for Space-Partitioning Trees," accepted for presentation at *20th Intl. Conf. on Data Engineering (ICDE)*, March 30 - April 2, Boston, 2004.

[13] A. Guttman, "R-trees: a Dynamic Index Structure for Spatial Searching," in *Proc. SIGMOD*, 1984, pp. 47–57.

[14] G. R. Hjaltason and H. Samet, "Improved Bulk-Loading Algorithms for Quadtrees," in *Proc. ACM GIS*, 1999, pp. 110–115.

[15] G. R. Hjaltason, H. Samet and Y. J. Sussmann, "Speeding up Bulk-Loading of Quadtrees," in *Proc. ACM-GIS*, 1997, pp. 50–53.

[16] I. Kamel and C. Faloutsos, "Hilbert R-tree: an Improved R-tree using Fractals," in *Proc. VLDB*, 1994, pp. 500–509.

[17] I. Kamel, M. Khalil and V. Kouramajian, "Bulk Insertions in Dynamic R-Trees," in *Proc. SDH*, 1996, pp. 31–42.

[18] S. Leutenegger, M. Lopez and J. Edgigton, "STR: a Simple and Efficient Algorithm for R-tree Packing," in *Proc. ICDE*, 1997, pp. 497–506.

[19] M.-L. Lo and C.-V. Ravishankar, "Spatial Joins Using Seeded Trees," in *Proc. SIGMOD*, 1994, pp. 209–220.

[20] Y. Manolopoulos, Y. Theodoridis and V. Tsotras, *Advanced Database Indexing*, Kluwer Academic Publishers, Boston, 1999.

[21] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer, Berlin, 1984.

[22] S.R. Ning An, K. V. R. Kanth and S. Ravada, "Improving Performance with Bulk-Inserts in Oracle R-Trees," in *Proc. VLDB*, 2003, pp. 948–951.

[23] N. Roussopoulos, Y. Kotidis and M. Roussopoulos, "Cubetree: Organization of and Bulk Updates on the Data Cube," in *Proc. SIGMOD*, 1997, pp. 89–99.

[24] N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-trees", in *Proc. SIGMOD*, 1985, pp. 17–31.

[25] Y. Theodoridis, *The R-Tree Portal*, World Wide Web, http://www.rtreeportal.org/, 2003.