Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Information Processing Letters 101 (2007) 6–12
Information Processing Letters
www.elsevier.com/locate/ipl

Execution time analysis of a top-down R-tree construction algorithm ☆

Houman Alborzi, Hanan Samet *

Department of Computer Science and Center for Automation Research, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

Received 9 May 2006; received in revised form 27 July 2006; accepted 27 July 2006

Available online 8 September 2006

Communicated by J. Chomicki

Abstract

A detailed CPU execution-time analysis and implementation are given for a bulk loading algorithm to construct R-trees due to García et al. [Y.J. García, M.A. López, S.T. Leutenegger, A greedy algorithm for bulk loading R-trees, in: GIS'98: Proc. of the 6th ACM Intl. Symp. on Advances in Geographic Information Systems, Washington, DC, 1998, pp. 163–164] which is known as the top-down greedy split (TGS) bulk loading algorithm. The TGS algorithm makes use of a classical bottom-up packing approach. In addition, an alternative packing approach termed top-down packing is introduced which may lead to improved query performance, and it is shown how to incorporate it into the TGS algorithm. A discussion is also presented of the tradeoffs of using the bottom-up and top-down packing approaches.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Spatial databases; R-trees; Bulk loading; Packing; Data structures

1. Introduction

The R-tree [7,10] was developed as an index structure for the efficient management of multi-dimensional and spatial data such as points and regions. Common operations performed on an R-tree include point location queries, range queries and nearest neighbor queries. Given a set of input data objects, an R-tree could be constructed by the repeated insertion of each data item. This approach does not take advantage of the fact that all the data items are known beforehand, as in this case it is preferable to insert all of the data items using a single operation. Such an operation is called *bulk loading*. An additional motivation for bulk loading is to enable the construction of an R-tree which can perform queries faster.

There have been a number of bulk loading techniques developed for R-trees (e.g., [1,3,8,9,11]). In this paper we present a formal analysis of the cost of building an R-tree using the *Top-down Greedy Split* (TGS) bulk loading technique that was originally proposed by García et al. [6]. Our approach differs from theirs by providing a detailed implementation which enables a more precise analysis of the algorithm. In particular, the

☆ The support of the National Science Foundation under Grants EIA-00-91474 and CCF-0515241, Microsoft Research, and the University of Maryland General Research Board is gratefully acknowledged.
* Corresponding author.
E-mail addresses: houman@umiacs.umd.edu (H. Alborzi), hjs@umiacs.umd.edu (H. Samet).

0020-0190/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2006.07.010

analysis given in [6] only considers the number of disk pages accessed for bulk loading of the data, while a formal analysis of the needed CPU time is missing. Given that memory is getting cheaper, many spatial databases fit into memory (e.g., in-car applications) and thus an analysis of the number of disk page accesses is not sufficient. This is especially true in the case of a bulk loading algorithm such as TGS which performs many sorting operations in order to obtain an R-tree that minimizes a particular cost function. The algorithm of García et al. in [6] uses a classical bottom-up packing approach. We also introduce a top-down packing approach, show how to incorporate it into the TGS algorithm, and discuss the tradeoffs in choosing one versus the other.

Our motivation for presenting the analysis and implementation of the TGS algorithm is to try to provide analytical support to the experimental results reported in [5] which showed that the R-tree built using the TGS bulk loading technique performs much better compared to those built using other bulk loading techniques, even though the bulk loading process is slightly slower for TGS. It is important to note that in this paper we do not analyze the performance of queries executed on an R-tree constructed by the TGS bulk loading operation; instead, we repeat, our contribution is to formally analyze the time required for performing the bulk loading operation.

The rest of this paper is organized as follows. Section 2 reviews R-trees and the bulk-loading process. Section 3 provides a description of the TGS bulk loading algorithm as well as a sample implementation. Section 4 describes the two approaches to packing that are used in bulk loading algorithms. Section 5 contains the formal analysis of the TGS algorithm, while Section 6 contains some concluding remarks.

2. Background

The most basic object that is stored in an R-tree is an *axis-aligned rectangle*, also called a *bounding box*. An R-tree data structure is a height balanced data structure similar to a B-tree [4] which facilitates storage of spatial data in secondary storage. Each leaf node of an R-tree holds two items for each data record. One is the bounding box of the record, and one is a pointer (or an identifier) to the data record itself. Similarly, each nonleaf node of an R-tree holds two items for each of its children: a bounding box of the child, and a pointer to the child. Furthermore, to ensure that an R-tree is height balanced, each node has between b and $M \geqslant 2b$ children, where M is called the *page capacity* of an R-tree node. In general, the page capacity of a leaf node is different from the page capacity of a nonleaf node. A node that has less than b children is termed *underpacked*. The root node of an R-tree is allowed to be underpacked.

A common query on an R-tree is a *window query* which reports all the data records in the R-tree whose boxes intersect a query rectangle w. When w is a point, the query is called a *point query*. A window query is performed by examining the root of an R-tree and recursively searching all its children that intersect w.

The efficiency of operations on an R-tree depends on the geometric relation of the nodes with respect to each other as well as on the height of the R-tree. For example, during a point query, all the nodes of the R-tree that cover the query point are visited. The query's performance is thus proportional to the number of nodes visited. If the query point is inside the bounding box of two or more sibling R-tree nodes, then all such nodes must be visited. Queries can be performed faster if the sibling nodes of an R-tree have little or no overlap. Intuitively, reducing the overlap of sibling R-tree nodes also results in better performance. A *cost function* quantifies this notion by assigning a cost to the geometric relation of the sibling nodes of an R-tree, which is usually [2,7,8] a function of their areas, perimeters, and their overlap area.

Roussopoulos and Leifker [9] introduced the concept of a *packed* R-tree, where all nodes of the R-tree are as full as possible. This results in an R-tree with the lowest possible height, thereby possibly improving the performance of search queries. However, search performance is still dependent on the amount of overlap between nodes. Their approach for building a packed R-tree is a *bottom-up* approach.

In general, a bottom-up approach for building packed R-trees is a two step process. It the first step, the n data rectangles are sorted according to a predetermined sort order. In the second step, groups of M data rectangles are placed into $\lceil n/M \rceil$ leaf nodes. After building the leaf nodes, the same process is applied to the bounding boxes of the leaf nodes to build another level of the R-tree. This process is applied iteratively until the root node of the R-tree is obtained.

On the other hand, a *top-down* approach, builds the higher levels of the R-tree first. The data rectangles are sorted according to a predetermined sort order and then the groups of n/M data rectangles are associated with the M children of the root. The process will be repeated for each of the M children of the root. In such an approach only one sort is needed for the first iteration, as the order of the boxes does not change during the subsequent iterations. The time complexity of a top-down approach is also $O(n \log n)$. Kamel and Faloutsos [8]

use a Hilbert curve sort order to build packed R-trees by sorting the collection of data rectangles only once. Hence, their method while described as bottom-up approach is essentially a top-down approach.

The bulk loading approaches described so far do not take into account any notion of a cost function. Depending on the sort order chosen, these approaches may or may not produce a desirable R-tree. The TGS (Top-down Greedy Split) algorithm of García et al. [6] proposes to overcome this issue by taking into account a cost function and tries to find a partition with a low cost. The n data rectangles are first sorted using an appropriate sort key, and then inserted in order into M bins each holding n/M rectangles. Moreover, the minimum bounding box of the rectangles in a bin is computed and kept as the bounding box of the bin. The bins are also numbered from 1 to M using each of the possible sort orders. At this point, we try to find an optimal partition of the M bins into two sets containing the first i bins and the next $M - i$ bins so that the value of the cost function on the minimum bounding rectangles of the bins that make up each of the two sets is minimized (e.g., their overlap). The key to this step is that we try to find the optimal partition using all of the possible sort orders. It should be clear that in this initial step there are $M - 1$ possible partitions and the TGS algorithm takes all of them and all of the possible sort orders into account when determining the optimal one at this step. This is a *greedy binary* split of the bins and the rectangles that they contain into two partitions. Each partition of the data rectangles is split again until all of the data rectangles are partitioned into M partitions of sizes less than or equal to n/M, at which time we have obtained the first level of the R-tree. The same algorithm is then applied recursively to the individual nodes of the R-tree until all nodes at a given level contain at most M data rectangles. Given S different sort orders, the TGS algorithm sorts the n rectangles in S different orders. While the S sort orders used in [6] are based on the $2d$ coordinates of the d-dimensional rectangles, any sort order defined using a sort key, such as the Hilbert order, could be used in the TGS algorithm. Section 3 contains a more detailed description of the algorithm.

3. TGS bulk loading algorithm

In this section, we present a detailed implementation-level description of the TGS algorithm given in [5,6]. The input to the TGS bulk loading algorithm is a list D of d-dimensional data rectangles. The algorithm builds an R-tree for these data rectangles. Each d-dimensional rectangle r is defined by d pairs of scalars, where each pair $r_i = (r_i^-, r_i^+)$ denotes the range that r spans in the ith dimension. We use the notation $p \boxplus q$ to denote the minimum bounding box of two rectangles p and q. For $r = p \boxplus q$, we have $r_i^- = \min(p_i^-, q_i^-)$ and $r_i^+ = \max(p_i^+, q_i^+)$ for $i = 1 \ldots d$.

We assume that there are S different sort keys associated with each rectangle r in D, where $\textsc{SortKey}(r, s)$ denotes the sth sort key on r. For example, the sort keys of a two-dimensional rectangle r could be chosen as its extents: $\textsc{SortKey}(r, 1) = r_1^-$, $\textsc{SortKey}(r, 2) = r_1^+$, $\textsc{SortKey}(r, 3) = r_2^-$, and $\textsc{SortKey}(r, 4) = r_2^+$. We assume further that each sort key associated with a rectangle is uniquely defined, and a mechanism for breaking the ties is in place. That is, for the sth sort key and the distinct data rectangles r and p, $\textsc{SortKey}(r, s) \neq \textsc{SortKey}(p, s)$.

The algorithm is invoked by $\textsc{BulkLoad}(D)$ (Algorithm 1), where D is the list of input rectangles. BULKLOAD proceeds by sorting the data in ascending order using S different sort keys, and storing the results in lists $D^{(1)}, \ldots, D^{(S)}$. It then determines the height of the R-tree and invokes BULKLOADCHUNK, which generates an R-tree with the specified height using the sorted data. Note that the boldface symbol **D** denotes the sorted lists $D^{(1)}, \ldots, D^{(S)}$.

BULKLOADCHUNK (Algorithm 2) simply returns an R-tree leaf if the desired height of the R-tree is zero. Otherwise, it determines m, the desired number of data items that need to be placed under each node (line 6). m is chosen so that all the nodes will have the max-

```
Input: D = {r_1, ..., r_n} is a list of n rectangles.
  (* S is the number of sort keys defined on each rectangle. *)
  (* N is the capacity of leaf nodes, and M is the capacity of nonleaf nodes. *)
  (* Top-Down-Greedy-Split bulk loading algorithm *)
  for i = 1 to S do
    D^(i) ← SORT(D, i) (* Sort D on the ith sort key *)
  h ← max(0, ⌈log_M |D|/N⌉) (* Desired height of the R-tree. *)
  return BULKLOADCHUNK(D, h)
```

Algorithm 1. BULKLOAD(D).

```
   (* Bulk load data in D into an R-tree of height h. *)
   (* M is the capacity of nonleaf nodes. *)
   if h = 0 then
     return BuildLeafNode(D^(1))
       (* Note that any of the sorted lists could have been used. *)
 5: else
     m = N · M^(h−1)
       (* Desired number of data items under each child of this node. *)
     {D_1, ..., D_k} ← Partition(D, m)
       (* Partition of D into k ⩽ M parts. *)
     for i = 1 to k do
     n_i = BulkLoadChunk(D_i, h − 1)
       (* Recursively bulk load lower levels of the R-tree. *)
10: return BuildNonleafNode(n_1, ..., n_k)
```

Algorithm 2. BulkLoadChunk($\mathbf{D}, h$).

```
   (* Partition data into k = ⌈|D^(1)|/m⌉ parts of size m ≠ 0. *)
   if |D^(1)| ⩽ m then
     return D (* one partition *)
   L, H ← BestBinarySplit(D, m)
5: return Concatenation of Partition(L, m) and Partition(H, m)
```

Algorithm 3. Partition($\mathbf{D}, m$).

```
   (* Find the best binary split of D. *)
   (* m ≠ 0 is the size of each partition. *)
   M ← ⌈|D^(1)|/m⌉ (* Number of partitions *)
   c* ← ∞ (* Best cost found so far *)
 5: for s = 1 to S do
     F, B = ComputeBoundingBoxes(D^(s), m)
     for i = 1 to M − 1 do
       c ← cost(F_i, B_i)
       if c < c* then
10:       c* ← c (* Best cost *)
         s* ← s (* Best sort order *)
         key = SortKey(D^(s)_(i·m), s) (* Sort key of split position *)
   for s = 1 to S do
     L^(s), H^(s) = SplitOnKey(D^(s), s*, key)
15: return L, H
```

Algorithm 4. BestBinarySplit($\mathbf{D}, m$).

imum number of data rectangles under them. Next, it uses the Partition algorithm (Algorithm 3) to partition the data into sets of size m, and recursively builds an R-tree node for each partition, returning a nonleaf R-tree node as their parent.

The Partition algorithm partitions the input set **D** into partitions of size m using a greedy paradigm. It uses the BestBinarySplit algorithm (Algorithm 4) to find a desirable binary split of the input set **D** into two partitions **L** and **H**. Note that the boldface symbols **L** and **H** denote the sorted lists $L^{(1)}, \ldots, L^{(S)}$ and $H^{(1)}, \ldots, H^{(S)}$. It then recursively partitions **L** and **H** and builds a bigger partition by joining them.

The BestBinarySplit algorithm considers the S different orderings of the input set **D**. It uses each ordering to group the data rectangles into groups of size m. That is, if there are $M \cdot m$ rectangles, then the first m rectangles are grouped together, then the second m rec-

Output: $L_i = D_1 \boxplus \cdots \boxplus D_{i \cdot m}$ for $1 \leqslant i < M$.
Output: $H_i = D_{i \cdot m+1} \boxplus \cdots \boxplus D_n$ for $1 \leqslant i < M, n = |D|$.
(* Compute the lower and higher bounding boxes of possible binary splits of D list of n rectangles into groups of size m *)
(* $m \neq 0$ is the size of each group. *)
$M \leftarrow \lceil \frac{|D|}{m} \rceil$ (* Number of groups *)
B is a list of M rectangles.
5: L, H are each a list of $M - 1$ rectangles.
for $i = 1$ **to** M **do**
$B_i \leftarrow D_{(i-1) \cdot m+1} \boxplus D_{(i-1) \cdot m+2} \boxplus \cdots \boxplus D_{\min(|D|, i \cdot m)}$
$L_1 \leftarrow B_1$
$H_{M-1} \leftarrow B_M$
10: **for** $i = 2$ **to** $M - 1$ **do**
$L_i \leftarrow L_{i-1} \boxplus B_i$
$H_{M-i} \leftarrow B_{M-i+1} \boxplus H_{M-i+1}$
return L, H

Algorithm 5. COMPUTEBOUNDINGBOXES(D, m).

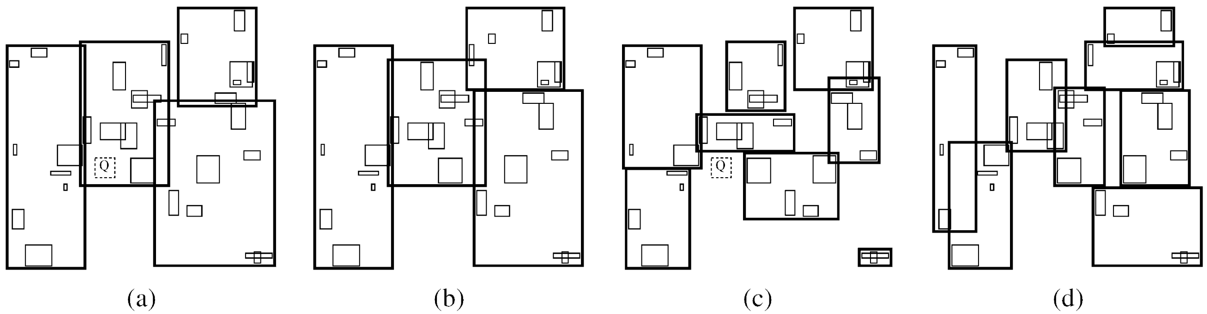

Figure 1. Result of applying a TGS bulk loading algorithm to an R-tree that minimizes (a) overlap area with bottom-up packing, (b) total area with bottom-up packing, (c) overlap area with top-down packing, and (d) total area with top-down packing.

tangles are grouped together, and so forth. It then considers all possible splits of the groups into two parts. In particular, if there are M groups, it considers $S \cdot (M - 1)$ possible ways of splitting the groups into two parts. The BESTBINARYSPLIT algorithm chooses the split with the lowest cost, and accordingly splits the input set **D** (i.e., the data and its S orderings) into two parts using the SPLITONKEY algorithm.

The COMPUTEBOUNDINGBOXES algorithm (Algorithm 5) determines the bounding boxes that are needed for determining the cost of each binary split considered in BESTBINARYSPLIT. It first computes B, the bounding boxes for each group of m rectangles. It then computes lower bounding boxes (L) and the higher bounding boxes (H).

The SPLITONKEY algorithm (not given here) will split a sorted list D, into two sorted lists L and H based on a threshold t, and the sth sort key among the S sort keys defined on rectangles. At the end of SPLITONKEY, L will hold all elements of I such that their sth key is less than t, and H will hold the rest.

4. Bottom-up packing versus top-down packing algorithms

Fig. 1 shows a set of 30 randomly generated rectangles that are bulk loaded using the TGS algorithm into an R-tree with page capacity of 8 (i.e., $N = M = 8$). Each R-tree in the figure consists of a root and four leaf nodes under the root. The inner rectangles correspond to the data rectangles, and the outer rectangles correspond to the bounding boxes of each leaf node. Four sort keys are used in the generation of the figure. In particular, the four sort keys of a rectangle are its two extents in each of the two dimensions. The cost functions used to generate Fig. 1 involved minimizing the overlap area of two rectangles (Fig. 1(a)) and minimizing the total area of two rectangles (Fig. 1(b)).

Traditionally, packing methods work by filling the leaf nodes as much as possible and then proceed to apply the same filling criteria to the nonleaf nodes. We characterize such an approach as *bottom-up packing* and is the one used in the implementation of the TGS algo-

rithm described in Section 3 and illustrated in Fig. 1. We could also proceed by starting at the root and packing the nonleaf nodes as much as possible. Such an approach can be characterized as *top-down packing*. The TGS algorithm whose implementation we described could also be converted to use top-down packing by modifying line 6 of BULKLOADCHUNK (Algorithm 2) to be:

$m = \left\lceil \frac{|D^{(1)}|}{M} \right\rceil$ (* Desired number of data items under each child of this node. *)

Figs. 1 (c) and (d) were obtained using this modification, with the same dataset as in Figs. 1 (a) and (b). Given N and M as the capacities of the leaf and nonleaf nodes, respectively, the bottom-up packing and top-down packing yield identical results when n, the total number of data objects, equals $N \cdot M^h$ for some integer value $h > 0$. However, when n is not equal to $N \cdot M^h$, the top-down packing yields a different result as can be seen by comparing Figs. 1 (a) and (b) with Figs. 1 (c) and (d). We note that top-down packing can potentially allow the queries to be performed faster as there are more children under each nonleaf node thereby permitting more effective pruning when answering queries. However, an R-tree constructed with the bottom-up packing TGS algorithm has fewer nodes than an R-tree constructed with the top-down TGS algorithm. Therefore, we can identify a tradeoff between the two packing approaches. In particular, the top-down packing TGS algorithm builds R-trees that can potentially be used to answer queries faster than R-trees built by the bottom-up packing TGS algorithm at the expense of requiring more storage space. We point out that the relative merit of the two packing strategies depends on the query model. For example, to answer the window query Q shown in Fig. 1(a), one leaf node needs to be read from disk. However, to answer the same query shown in Fig. 1(c), only the root node needs to be examined. On the other hand, answering a window query that intersects all the leaf nodes with the bottom-down packing is obviously faster than with the top-down packing. Finally, we observe that one of the consequences of using top-down packing is that some of the leaf nodes of the R-tree may be underpacked (e.g., recall that one leaf node in Fig. 1(c) has just two data rectangles). Of course, when using bottom-up packing at most one node at each level is underpacked. However, this does not affect the correctness of results.

5. Analysis

In this section we analyze the running time of the TGS bulk loading algorithm. As we pointed out in Section 1, the analysis provided in [6] was only in terms of the number of disk page accesses, whereas here we focus on the CPU time in light of the repeated invocation of the sorting steps by the algorithm in the process of minimizing the particular cost function. To simplify the analysis, we assume that the number of input data rectangles results in a fully packed R-tree, i.e., there are $n = NM^h$ data rectangles, where h denotes the height of the resulting R-tree.

Let $T(n)$ denote the time complexity of the BULKLOAD algorithm. The BULKLOAD algorithm performs S sorts. We have,

$$T(n) = \mathrm{O}(Sn \log n) + B(n, h), \tag{1}$$

where $B(n, h)$ denotes the time complexity of the BULKLOADCHUNK algorithm.

Notice that as the initial number of the data rectangles results in a fully packed R-tree, it suffices to derive $B(N \cdot M^h, h)$. Letting $C(h)$ denote $B(N \cdot M^h, h)$. We have,

$$C(h) = \begin{cases} \mathrm{O}(N), & h = 0, \\ P(N \cdot M^h, N \cdot M^{h-1}) + M \cdot C(h-1) \\ \quad + \mathrm{O}(M), & h > 0, \end{cases} \tag{2}$$

where $P(n, m)$ denotes the time complexity of the PARTITION algorithm, and $\mathrm{O}(M)$ corresponds to the cost of invoking BUILDNONLEAFNODE.

We now proceed to derive $P(n, m)$. We first notice that the PARTITION algorithm consists of a call to the BESTBINARYSPLIT algorithm and two recursive calls to itself. The worst-case scenario arises when each call to BESTBINARYSPLIT results in a minimum partition. That is, BESTBINARYSPLIT(D, m) yields two sets, such that one of them is of size m. We have the following recurrence relation:

$$P(n, m) = \begin{cases} \mathrm{O}(1), & n \leqslant m, \\ E(n, m) + P(m, m) \\ \quad + P(n - m, m), & n > m, \end{cases} \tag{3}$$

where $E(n, m)$ denotes the time complexity of the BESTBINARYSPLIT algorithm.

Observe that the execution times of the COMPUTEBOUNDINGBOXES algorithm and the SPLITONKEY algorithm are linear in their input size. Moreover, as the BESTBINARYSPLIT algorithm invokes the COMPUTEBOUNDINGBOXES algorithm S times, its execution time, $E(n, m)$, is $\mathrm{O}(S \cdot n)$, where n is the number of input rectangles.

Therefore, we can rewrite Eq. (3) as:

$$P(n, m) = \begin{cases} \mathrm{O}(1), & n \leqslant m, \\ \mathrm{O}(S \cdot n) + P(m, m) \\ \quad + P(n - m, m), & n > m. \end{cases} \tag{4}$$

To further simplify the analysis, we assume that $n = L \cdot m$, where L is the number of groups that the PARTITION algorithm considers. Notice that for each initial call of PARTITION from BULKLOADCHUNK, we have $L = M$. Therefore, we can rewrite Eq. (4) in closed form:

$$P(L \cdot m, m) = \sum_{i=2}^{L} \mathrm{O}(S \cdot i \cdot m) = \mathrm{O}(S \cdot L^2 \cdot m). \tag{5}$$

By substituting $N \cdot M^{h-1}$ for m and M for L in Eq. (5) we get $P(N \cdot M^h, N \cdot M^{h-1}) = \mathrm{O}(S \cdot (M^2)M^{h-1}) = \mathrm{O}(S \cdot M^{h+1})$, and we can rewrite Eq. (2) as:

$$C(h) = \begin{cases} \mathrm{O}(N), & h = 0, \\ \mathrm{O}(S \cdot M^{h+1}) + M \cdot C(h-1) & \\ \quad + \mathrm{O}(M), & h > 0. \end{cases}$$

The recurrence relation for $C(h)$ can be solved to yield

$$\begin{aligned} C(h) &= \mathrm{O}\big(M^h \cdot (S \cdot h \cdot M + N)\big) \\ &= \mathrm{O}\left(n \cdot \left(S \cdot h \cdot \frac{M}{N} + 1\right)\right), \end{aligned}$$

where we have used $n = N \cdot M^h$.

Recalling that $C(h) = B(NM^h, h)$ and that $h = \log_M \frac{n}{N}$, we obtain from Eq. (1) that

$$T(n) = \mathrm{O}\left(Sn \log n + n \cdot \left(S \frac{M}{N} \log_M \frac{n}{N}\right)\right).$$

In particular, for $M = N = \mathrm{O}(1)$, we have $T(n) = \mathrm{O}(Sn \log n)$, which demonstrates that the observed improved performance of the TGS algorithm by García et al. [6] comes at a cost of a factor of S over that resulting from the use of a bottom up bulk loading approach. Given that S is relatively low for low dimensional data, the improvement seems worth the extra effort. However, in the case of high dimensional data, this may not be the case.

6. Concluding remarks

A formal analysis of the TGS R-tree bulk loading algorithm was provided. Our approach differs from theirs by providing a detailed implementation which enabled a more precise analysis of the algorithm. In particular, we focused on the CPU time requirements rather than the number of disk page accesses, which is what was done in [6]. We also discussed the tradeoffs of using classical bottom-up packing and top-down packing, and showed how to incorporate top-down packing in the TGS algorithm.

References

[1] L. Arge, K.H. Hinrichs, J. Vahrenhold, J.S. Vitter, Efficient bulk operations on dynamic R-trees, in: ALENEX'99: Proc. of the 1st Workshop on Algorithm Engineering and Experimentation Baltimore, MD, Jan. 1999, in: Lecture Notes in Computer Science, vol. 1619, Springer, Berlin, 1999, pp. 328–348.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, in: SIGMOD'90: Proc. of the Internat. Conf. on Management of Data, Atlantic City, NJ, May 1990, pp. 322–331.

[3] L. Chen, R. Choubey, E.A. Rundensteiner, Bulk-insertions into R-trees using the Small-Tree-Large-Tree approach, in: GIS'98: Proc. of the 6th ACM Intl. Symp. on Advances in Geographic Information Systems, Washington, DC, 1998, pp. 161–162.

[4] D. Comer, The ubiquitous B-tree, ACM Comput. Surv. 11 (2) (1979) 121–137.

[5] Y.J. García, M.A. López, S.T. Leutenegger, A greedy algorithm for bulk loading R-trees, Computer Science Technical Report 97-02, University of Denver, Denver, CO, 1997.

[6] Y.J. García, M.A. López, S.T. Leutenegger, A greedy algorithm for bulk loading R-trees, in: GIS'98: Proc. of the 6th ACM Intl. Symp. on Advances in Geographic Information Systems, Washington, DC, 1998, pp. 163–164.

[7] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD'84: Proc. of the Intl. Conf. on Management of Data, Boston, MA, June 1984, pp. 47–57.

[8] I. Kamel, C. Faloutsos, On packing R-trees, in: CIKM'93: Proc. of the 2nd Intl. Conf. on Information and Knowledge Management, Washington, DC, Nov. 1993, pp. 490–499.

[9] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed R-trees, in: SIGMOD'85: Proc. of the Intl. Conf. on Management of Data, Austin, TX, May 1985, pp. 17–31.

[10] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, San Francisco, CA, 2006.

[11] J. van den Bercken, B. Seeger, P. Widmayer, A generic approach to bulk loading multidimensional index structures, in: VLDB'97: Proc. of the 23rd Intl. Conf. on Very Large Databases, Athens, Greece, Aug. 1997, pp. 406–415.