Bulk Insertion for R-tree by Seeded Clustering [*]

Taewon Lee[1], Bongki Moon[2] and Sukho Lee[1]

[1] School of Electrical Engineering and Computer Science
Seoul National University, Seoul, Korea
{warrior@db,shlee@cse}.snu.ac.kr
[2] Department of Computer Science, University of Arizona, Tucson, AZ 85721
bkmoon@cs.arizona.edu

Abstract. In many scientific and commercial applications such as Earth Observation System (EOSDIS) and mobile phone services tracking a large number of clients, it is a daunting task to archive and index ever increasing volume of complex data that are continuously added to databases. To efficiently manage multidimensional data in scientific and data warehousing environments, R-tree based index structures have been widely used. In this paper, we propose a scalable technique called *Seeded Clustering* that allows us to maintain R-tree indexes by bulk insertion while keeping pace with high data arrival rates. Our approach uses a *seed tree*, which is copied from the top k levels of a target R-tree, to classify input data objects into clusters. We then build an R-tree for each of the clusters and insert the input R-trees into the target R-tree in bulk one at a time. We present detailed algorithms for the seeded clustering and bulk insertion as well as the results from our extensive experimental study. The experimental results show that the *bulk insertion by seeded clustering* outperforms the previously known methods in terms of insertion cost and the quality of target R-trees measured by their query performance.

1 Introduction

In many data-intensive applications, there has been an upsurge of interest in dealing with the problem of *bulk insertions* of new data into an existing database. It is important to add newly collected data into an existing database quickly, because data are continuously generated and added to databases. Construction of a new index structure each time from scratch for both the existing data as well as the new data is not likely to scale well.

In this paper, we present a bulk insertion technique using *Seeded Clustering* for an R-tree based index structure and show that it outperforms the previous bulk insertion methods in terms of insertion cost and query processing cost. Most of the previous work followed a common approach. They first group input

---

data items into clusters and then insert each cluster one at a time in bulk [3, 4, 6]. Under the approach, input data items that are spatially close are grouped into clusters. Although each cluster of data will cover a small extent of area, it is unlikely to guarantee the least of the dead space of existing R-tree nodes. This is because input data items clustered by themselves without considering the structure of a target R-tree. The query performance of the resulting target R-tree can be degraded.

Our *Seeded Clustering* utilizes the structure of a target R-tree in clustering input data items. We use the top k levels of a target R-tree as a guide to classify the input data items into clusters in a linear time. Then we build an input R-tree from each of the clusters and insert them into the target R-tree one at a time in bulk. However, the insertion of an input R-tree into the target R-tree is not such a simple process. The target R-tree should remain as a legitimate R-tree even after the insertion is done. This property is not guaranteed by some of the previous work.

There are two important aspects of bulk insertions. First, the bulk insertion itself should be fast enough to catch up with the rate of the data generation. Second, the query performance should not be compromised by bulk insertion. Since our approach attempts to reduce the overlap during the bulk insertion, the quality of the target tree is preserved or even better restructured so that the query performance can be improved.

For the performance evaluation, extensive experiments were conducted both with a real data set and various synthetic data sets. We used a TIGER/Line data set which is a popular data set for geographic information systems. One of the synthetic data sets was generated by a TPC/H data generator. We also used three more synthetic data sets, which are uniform distributed, skewed and clustered data set.

The paper is organized as follows. In section 2, we briefly overview the related work on bulk insertion. In section 3, the problem we are going to solve is described and the algorithm is presented roughly. In section 4, the structure of a seed tree is described. Classifying the input data items using a seed tree is also given. In section 5, the detailed algorithm of the bulk insertion is provided. In section 6, we show the result of performance evaluation and section 7 concludes this paper.

2   Previous Work

In an early piece of work on the bulk insertion for an R-tree, the data items to be inserted are first sorted by their spatial proximity (e.g., the Hilbert value of the center) and then packed into blocks of B rectangles [6]. These blocks are then inserted one at a time using standard insertion algorithm. Intuitively, the algorithm should give an insertion speed-up of B (as a block of B data items is inserted at a time), but it is likely to increase overlap and thus produces a worse index in terms of query performance. This can be explained as follows. Although the block may contain data items that are spatially close, in the tar-

get tree's point of view, this block is randomly constructed against its nodes. Therefore a block and the nodes in the target tree are not guaranteed to be non-overlapping which might increase overlap between them. This can be supported by the empirical results [6].

There was another work on the bulk insertion which used a STLT (*small-tree-large-tree*) approach [3]. The STLT constructs an R-tree (*small tree*) from the data set and inserts it into the target R-tree (*large tree*). To insert a *small tree* into a *large tree*, it chooses an appropriate location to maintain the balance of the resulting *large tree*. However, this approach has the following shortcomings. In the STLT approach, a *small tree* is built from the input data items and inserted into the *large tree*. Therefore, if a *small tree* covers large area, the node into which a *small tree* is inserted needs to be enlarged for the *large tree* to enclose it. This means the STLT only works well for highly skewed data set [4]. And there is another restriction that the depth of a *small tree* must be smaller than the *large tree*.

A variant of STLT is the GBI (generalized bulk insertion) technique [4]. In this work, the input data set is partitioned into a number of clusters by grouping spatially close data items into the same cluster. After clustering, from each of these clusters, R-trees are built. Finally, these R-trees are inserted into the target tree one at a time. The data items not included in any cluster are classified as outliers and inserted one by one using normal R-tree insertion. This work alleviated the limitation of the STLT which depends on the data distribution. However, this has also the same problem that the R-trees being inserted may increase the overall overlap of the target R-tree for the same reason mentioned in the first paragraph of this subsection. In addition to this, GBI and STLT have a serious problem that the resulting tree may not be a legitimate R-tree by definition. From the properties of an R-tree, the root node of a *small tree* can have less than m entries(m is the minimum number of entries a node can have). However, after a *small tree* is inserted, the root of the *small tree* becomes an internal node of the *large tree*. The root node of the *small tree* having less than m entries breaks the property of an R-tree. This is not an easy issue to deal with but must be addressed properly. In section 5, we present how this problem is solved by our bulk insertion method.

The other class of algorithm presented a new buffer strategy for performing bulk operations on dynamic R-trees [1]. Their method uses the buffer tree idea, which takes advantage of the available main memory and the page size of the operating system. Although their bulk insertion strategy shows improved results over the normal insertion algorithm in terms of the insertion cost, it is conceptually identical to the repeated insertion algorithm that the query performance of the resulting tree shows no better than using repeated insertion.

3   Overview of Our Approach

We suppose there is a target R-tree indexing a large number of data objects. A number of new data items arrive and these items need to be inserted into

the target R-tree. There is no pre-built index for these input data items. This should be done fast and the quality of the resulting target R-tree should be good enough in terms of query performance.
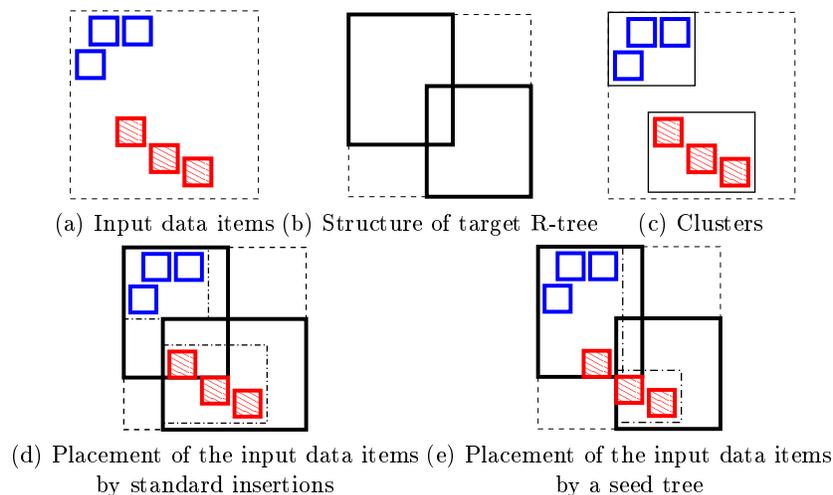
Our proposed work for bulk insertion algorithm is performed in two stages: *seeded clustering* and *bulk insertion*. First, we build a *seed tree* by taking a few top levels of nodes from a target R-tree. A seed tree guides the way the input data items are clustered. Then an R-tree is generated for each individual cluster. We call them *input* R-trees.

Next is the insertion step. During the insertion of an input R-tree, we attempt to reduce the overlapping area between the nodes of a target R-tree and an input R-tree to improve the query performance. Inserting an R-tree into another R-tree is described in Section 5 in detail.

4   Seeded Clustering

In the previous work, input data sets are partitioned into clusters using various clustering methods. Then an R-tree is built for each cluster. These R-trees are then inserted one at a time using a slightly modified standard insertion algorithm. This is the common way to perform bulk insertion [3, 4, 6].

Let us suppose that the input data items are located as in Fig. 1 (a) and the structure of the target R-tree is given as (b). Most of the known clustering methods may classify the data items as in (c). After building input R-trees from clusters and inserting them into the target R-tree as in (d), it is likely to increase the MBR of the nodes in the target R-tree since input R-trees are built independently from the target R-tree. However, we can utilize the structure of the target tree as shown in (e). If the structural information of the target R-tree

(a) Input data items (b) Structure of target R-tree   (c) Clusters

(d) Placement of the input data items (e) Placement of the input data items
by standard insertions   by a seed tree

Fig. 1. Motivation of Seeded Clustering

can be exploited, we know in advance that the lower three rectangles should not be placed in the same cluster.

A *seed tree* is constructed by copying the top k levels of the target R-tree. We use this seed tree for clustering the input data set. In each level of a seed tree, an entry whose MBR fully encloses an input data item is chosen. If a data item can reach a leaf node of a seed tree, it is classified into the cluster that corresponds to the leaf node. By doing this, a seed tree guides an input data item to a node that would be chosen if an input data item were inserted in a normal way.

During the classification of input data items, there may be more than one entries that fully include an input data item. We can make a naïve selection by choosing the first entry that fully encloses an input data item. This is the fastest way possible. There may be other ways to choose an entry which might lead to the better index structure but we use this naïve method for simplicity.

If it fails to find an entry that encloses a data item in some internal node of a seed tree, a data item is classified as an *outlier* of the node. Outliers are inserted one by one using normal R-tree insertion method. For all the data sets we used in the performance evaluation, the proportion of outliers in the input data items was less than 0.1%.

In the following section, we will describe the insertion step of the algorithm in detail. Input R-trees are generated from the clusters and they are inserted into a target R-tree in the insertion step. Inserting an R-tree into a target R-tree is presented in detail.


5   Bulk Insertion

In this section, we describe the bulk insertion step of the *bulk insertion by seeded clustering*. Before we begin, let us define the leaf level of the R-tree as level 0. Therefore, the height of an R-tree is 1 + the level of the root node.

5.1   Inserting an Input R-tree to the Target R-tree

As described in section 4, we get a number of clusters and outliers after clustering. To perform bulk insertion, we first create an input R-tree from each of the clusters and insert them into the target R-tree one at a time in bulk. We use a bulk loading method (e.g., [7, 8]) to build an input R-tree from each cluster.

Inserting an R-tree to another R-tree can be done in a similar way to inserting a data item into an R-tree except some post-processing. To insert an input R-tree into the target R-tree, we treat an input R-tree as if it were a data item whose MBR is that of the root of the input R-tree. However, to maintain the property of an R-tree that all the leaf nodes must be in the same level, the root node should not be inserted into the leaf level of the target R-tree.

Suppose that the height of an input R-tree is h_i. For the target R-tree after insertion to be a legitimate R-tree, the root N_i of the input R-tree needs to be inserted into a node of level h_i of the target R-tree. This is depicted in Figure
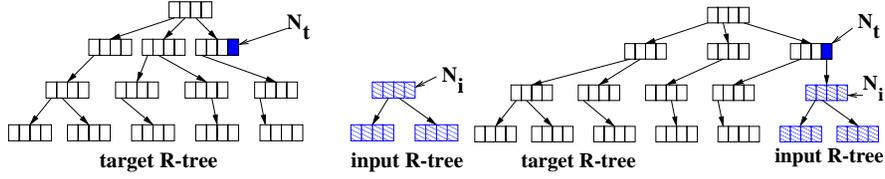
(a) the target R-tree and an input R-tree. (b) the target R-tree after insertion. The input R-tree is going to be inserted into the node N_t

Fig. 2. inserting input R-tree into the target R-tree

2. By the definition of a seed tree, we already know where a particular input R-tree will be inserted.

There is another important property for the target R-tree to be a legitimate R-tree after the insertion. Every node except the root must have at least m entries [5]. *Node underflow* is the term used for the node with less than m entries [2]. Suppose the number of entries in the root node N_i of an input R-tree is less than m. By definition, this is a legal R-tree as the root can have less than m entries. However, the target R-tree after inserting this input R-tree becomes illegal because the root of the input R-tree is now an internal node of the target R-tree and the node has less than m entries.

We solve this problem by distributing all the entries of N_i among overlapping entries of the target node N_t. In other words, we reinsert the entries of the node N_i into the node N_t.

For the case where the node N_i has enough entries, we insert the input R-tree into the node N_t of the target R-tree. However, there can be some complicated problem in doing this. Suppose the level of N_t is h_t. For an input R-tree to be fit for the node N_t, it should be of height h_t. It is possible that the data items are skewed and as a result, the size of a cluster can be large such that the input R-tree generated from this cluster becomes of height greater than h_t. In this case, it is unfit for the node N_t. On the contrary, if the data items are so sparse that the cluster has a small number of data items, an input R-tree built from this cluster may be of height lower than h_t. In this case, it cannot be inserted as a direct entry of the node N_t. We deal with these three cases appropriately. However, we omit the discussion here for lack of space.

5.2 Repacking : Locally reducing the overlapped area

Since there can be large overlap between the node N_i and the entries of the node N_t into which N_i is inserted, it is required to reduce overlap between them to make the target R-tree efficient for query processing. The *repacking* we describe in this section is the post-processing step of our algorithm that locally minimizes overlap between nodes.

Algorithm 1: Post-process : reducing the overlap during the bulk insertion

Function `post-process` (N_t, N_i)

N_t : node into which N_i is inserted

N_i : the root of an input R-tree

begin

     $NoOvlp \leftarrow$ entries of N_t that do not overlap with N_i

     $Ovlp \leftarrow$ entries of N_t that overlap with N_i

     $RepackedNodes \leftarrow$ `REPACK` $(Ovlp, N_i)$

     return `pack` $(NoOvlp \cup RepackedNodes)$

end

Algorithm 2: Repack the node N_{input} and the nodes in the set N_{set}

Function `REPACK` (N_{set}, N_{input})

N_{set} : a set of nodes whose parent nodes overlap with the node N_{input}

N_{input} : a node from an input R-tree

begin

1      $ent \leftarrow$ entries of each elements of N_{set}

2      $NoOvlp \leftarrow$ subset of ent that do not overlap with any entry of N_{input}

3      $Repacked \leftarrow \emptyset$

4   **if** N_{input} *is a leaf node* **then**

5   **return** `pack` $(ent \cup$ entries of $N_{input})$

     else

6   **foreach** $e \in$ *entries of* N_{input} **do**

7             $entOvlp \leftarrow$ find elements of ent that overlap with e

8   **if** $entOvlp \neq \emptyset$ **then**

9                 $Repacked \leftarrow Repacked \cup$
                       `REPACK` $(entOvlp, e)$

            else

10                 $Repacked \leftarrow Repacked \cup e$

            end

        end

     end

11   **return** `pack` $(NoOvlp \cup Repacked)$

end

Basic idea of the repacking is as follows. We take the entries out of the largely overlapped nodes and rebuild MBRs that enclose those entries. Detailed algorithms are given through Algorithm 1 to 3.

For lack of space, we briefly describe the algorithm. The entries of the target node N_t, into which N_i is inserted, can be divided into two groups. The entries that do not overlap with the node N_i, and the entries that overlap with the node N_i. We repack the latter group with N_i. Before repacking them, we first repack their child entries. This can be recursively defined and is given in Algorithm 2.

Algorithm 3: Pack the entries to create node(s) that contain all of them
---
Function pack (N_{set})

N_{set} : set of nodes to pack

begin

    | **return** create nodes that enclose the elements of N_{set} using bulk loading
    | method

end
---

6 Experiment

In this section, we present the result of an extensive experimental study to show the validity and the effectiveness of our approach. We have implemented a disk based R-tree in C++ on the linux machine. A node corresponds to a 4KB-disk block and can hold approximately 100 entries per node. We compared our experimental results with the repeated insertion method of an R-tree (OBO; *one by one*) and GBI. We presented the insertion cost and the query cost in average number of disk I/O. From now on, we will use SCB to represent our bulk insertion by seeded clustering algorithm.

We used a number of synthetic data sets having different characteristics as well as real data set. For the real data set, the standard benchmark data used in spatial databases, namely rectangles obtained from the TIGER/Line data set was used [9]. We extracted about 2,600,934 line objects from the states of Arizona, New Mexico, Utah and Colorado. For synthetic data sets, we used a data set from TPC-H, an ad-hoc, decision support benchmark [10] and three other data sets that have different distribution characteristics. They are uniform, zipfian(for skewed data set) and clustered data set, respectively. Each data set has 1,000,000 data items.

6.1 Insertion Cost and Query Cost for the Different Data Sets

We evaluated the performance of SCB as compared to OBO and GBI, for all the data sets described formerly. The size of an input data set was 5% of the size of the target R-tree. A seed tree for each data set was generated initially by copying the top 2 levels of the target R-tree which was of height 4. The target

	SCB	GBI	OBO
TIGER/Line data	36263	80757	151689
TPC-H data	18752	53021	108018
Uniform distributed data	16443	49855	103809
Zipfian distributed data	13247	43724	98096
Clustered data	11607	25351	56945

Table 1. Insertion Costs for various data set(# of I/Os)

	Initially	SCB	OBO	GBI
TIGER/Line	2.665	2.496	3.034	3.215
TPC-H	2.111	1.547	2.234	2.561
Uniform	5.446	4.313	5.678	5.941
Zipfian	4.442	3.733	4.645	5.018
Clustered	4.234	3.471	4.523	4.833

Table 2. Query Costs for various data set (# of I/Os). Point queries were used.

R-tree was initially built using the repeated insertion method. We measured the query cost for 5,000 random point queries. The insertion cost for each data set is given in Table 1 and the query cost is given in Table 2.

The result shows that the SCB we proposed in this paper outperforms the OBO and GBI in both insertion and query performance. Although it does not show extremely high gain in insertion cost, we can still see high gain from the result. This is mainly because our approach is designed to perform insertion in bulk.

An important result is that our method decreases not only the insertion cost, but also the query cost. We are targetting application areas where it is almost impossible to rebuild an index from scratch. Until now, OBO showed the best query performance. Previous work on bulk insertion showed no improvements in query performance compared to OBO [1, 4]. However, SCB showed better query performance than OBO after the insertion. This is because SCB tries to reorganize the target tree to reduce overlap between the nodes.

6.2   Repeated Insertion with Time Interval

In this experiment, we performed an experiment to see if the target tree maintains acceptable query performance after successive insertion of data with time
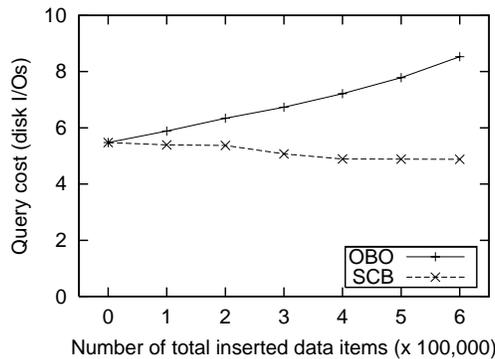
Fig. 3. Repeated insertion with time interval

gap. For a target R-tree to handle the continuously added data, the insertion method should maintain the quality of a target R-tree. The result is given in Figure 3.

7 Conclusion

In this paper, we have proposed an effective bulk insertion method in the environment where data are continuously added to databases. We have presented the *Seeded Clustering* technique which utilizes the structure of the target R-tree to quickly and effectively cluster the input data objects. By clustering, input data objects are grouped with their locality and from each of the clusters, input R-trees are built and inserted one at a time in bulk. We also have presented a local overlap minimizing algorithm called *repacking* which reorganizes nodes of the target R-tree and an input R-tree during the insertion to minimize the overlapped area. This makes our algorithm scalable. As the data items are continuously added, our method yields a good quality tree to a certain point that it outperforms previous bulk insertion methods in terms of query performance.

References

1. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient Bulk Operations on Dynamic R-Trees. *Algorithmica*, 33(1):104–128, 2002.
2. Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.
3. Li Chen, Rupesh Choubey, and Elke A. Rundensteiner. Bulk-insertions into R-trees using the small-tree-large-tree approach. In *Proceedings of the sixth ACM international symposium on Advances in geographic information systems*, pages 161–162, 1998.
4. Rupesh Choubey, Li Chen, and Elke A. Rundersteiner. GBI: A Generalized R-tree Bulk-Insertion Strategy. In *Advances in Spatial Databases*, pages 91–108, 1997.
5. Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, June 1984.
6. I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH'96)*, pages 3B.31–3B.42, 1996.
7. Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management*, pages 490–499, 1993.
8. S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the IEEE Data Engineering*, pages 497–506, 1997.
9. TIGER/Line Files, 2000 Technical Documentation, U.S. Bureau of Census, Washington DC, accessible via URL `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`.
10. TPC-H, Transaction Processing Performance Council, accessible via URL `http://www.tpc.org/tpch/`.