Available online at www.sciencedirect.com

SCIENCE DIRECT®

Data & Knowledge Engineering 59 (2006) 86–106

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Bulk insertion for R-trees by seeded clustering

Taewon Lee [a,*], Bongki Moon [b], Sukho Lee [a]

[a] *Seoul National University, School of Computer Science and Engineering, Kwanakgu, Shillimdong, Seoul ENG4190, Republic of Korea*

[b] *Department of Computer Science, University of Arizona, Tucson, AZ 85721, United States*

Received 20 May 2005; received in revised form 20 May 2005; accepted 27 July 2005
Available online 2 September 2005

Abstract

We propose a scalable technique called *Seeded Clustering* that allows us to maintain R-tree indices 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. The experimental results show that the *bulk insertion by seeded clustering* outperforms the previously known methods.
© 2005 Elsevier B.V. All rights reserved.

Keywords: R-tree; Bulk insertion; Repacking; Seeded clustering; Seed tree

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. It is not likely to scale well if we insert every new data into an existing database one at a time.

[*] Corresponding author.
E-mail address: warrior@db.snu.ac.kr (T. Lee).

doi:10.1016/j.datak.2005.07.011

In this paper, we present a bulk insertion technique using *Seeded Clustering* for an R-tree based index structure [1] and show that the resulting R-tree using our bulk insertion method outperforms the R-tree generated by 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 [2–4]. 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 not to overlap with existing R-tree nodes. This is because input data items are clustered by themselves without considering the structure of a target R-tree. The query performance of the resulting target R-tree is likely to be degraded because of the increased overlap area between the existing R-tree nodes and the newly inserted nodes.

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 as fast as possible and faster than individual insertions. Second, the query performance should not be compromised by bulk insertion. Since our approach attempts to reduce the overlap between the existing nodes and the newly inserted nodes during the bulk insertion, the quality of the target tree is preserved or even better restructured so that the query performance can be improved over that of the target R-tree before the bulk insertion.

For the performance evaluation, extensive experiments were conducted both with a real data set and two synthetic data sets. We used a TIGER/Line data set which is a popular data set for geographic information systems. We used uniform distributed and skewed data set as synthetic data sets.

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 Sections 5 and 6, the detailed algorithm of the bulk insertion is provided. In Section 7, we describe some issues that must be handled during the bulk insertion. In Section 8, we show the results of performance evaluation and Section 9 concludes this paper.

2. Previous work

In this section, we provide a brief overview of previous work on bulk insertion for R-tree environment.

In an early attempt on 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 [4]. 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 target tree's point of view, this block is constructed independently of the tree nodes. Therefore a block and the nodes in the target tree are not guaranteed to be non-overlapping which might increase overlap between them. The empirical results confirms that randomly created block results in the degraded query performance because of the increased overlap between nodes [4].

There is another work on the bulk insertion which uses a STLT (*small-tree-large-tree*) approach [2]. 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 shortcoming: if a *small tree* covers a large area, the node of a *large tree* into which a *small tree* is inserted needs to be enlarged to enclose it. This means the STLT only works well for highly skewed data sets [3].

A variant of STLT is the GBI (Generalized Bulk Insertion) technique [3]. 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. 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 is highly dependent 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 7, we present how this problem is solved by our bulk insertion method.

Another class of bulk operations rely on a buffer strategy for dynamic R-trees [5]. This approach adopts the lazy buffering technique of the buffer tree [6]. Basic idea of their work is to attach buffers to the internal nodes of an R-tree in precalculated levels and keep the total size of the buffers to fit in the memory. Then, when an object is inserted, it is stored in the buffer until it gets full. When the buffer is full, data objects in the buffer are pushed down to the buffer at the lower level. Since data objects are only inserted into the leaf level, it is only when the data objects arrive at the leaf level that disk accesses occur. By using buffers, they insert the data object as soon as it arrives without having to gather to perform bulk operation and are possible to reduce disk accesses by delaying insertions using the buffers. 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 is no better than that of repeated insertion.

3. Overview of our approach

In this section, we provide a brief overview of the seeded clustering algorithm we propose. Then we will give a detailed look on the algorithm from Section 4 through Section 6. To clearly define the problem we are going to solve, we suppose there is a *target* R-tree indexing a large number of data objects. A number of newly generated data items arrive and these items need to be inserted into the target R-tree. There is no pre-built index structure for these input data items. This should be done fast and the quality of the resulting target R-tree in terms of query performance should be good enough.

We use two-dimensional data in this paper. Although we do not provide experiments with higher dimensional data, our approach can be extended to handle multidimensional data higher than 2. Because R-trees suffer from dimensionality curse problem on approximately eight-dimensional data, our approach will not work effectively also in that high dimension.

Our proposed work for bulk insertion algorithm is performed in two stages: *seeded clustering* and *insertion*. In the seeded clustering step, we first 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. In the insertion step, we propose two different methods. In one method, we take each data item from a cluster and insert it into a target R-tree one at a time using the standard R-tree insertion method. Although it inserts data items one by one, it reduces the construction cost dramatically because of localized insertions.

In the other method, we build an R-tree from each of the clusters and insert them into the target R-tree one at a time in bulk. We call these R-trees as *input* R-trees. For the target R-tree to be a legitimate R-tree after the insertion of an input R-tree, the node underflow and height balance issues need to be addressed properly. We describe how to insert an R-tree into another R-tree in Sections 6 and 7 in detail.

4. Seeded clustering

In the previous approaches, input data items are partitioned into groups using various clustering methods. Each group forms an R-tree or a block(a node). Then they are inserted into the target tree one at a time. This is the common way to perform bulk insertion [2–4].

Suppose that the input data items are located as in Fig. 1(a) and the structure of the target R-tree is given as Fig. 1(b). Most of the previous clustering methods will classify the data items as in Fig. 1(c). After building input R-trees from clusters and inserting them into the target R-tree as in Fig. 1(d), it is likely to enlarge the MBR of the nodes in the target R-tree since input R-trees are built independently of the target R-tree.

On the other hand, with the structural information of the target R-tree, four leftmost and two rightmost rectangles would be grouped together, rather than the upper three and lower three. As a result, it is possible to maintain the MBRs of the target tree as was shown in Fig. 1(e). We call this approach *Seeded Clustering* since it uses the structural information of the target R-tree as a seed. We introduce a seed tree which helps the seeded clustering of the input data set.

4.1. Structure of a seed tree

The idea of the *seed tree* was partially adopted from the “seeded R-tree” which is used for joining two spatial data sets one of which has an R-tree for it and the other of which does not [7]. The seeded tree has seed levels and growth levels. Seed levels are built by copying the structural

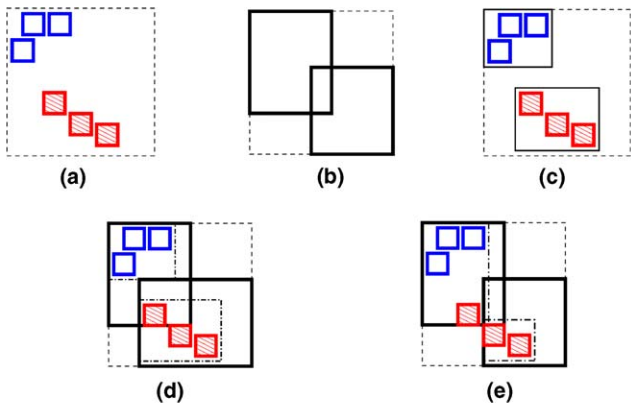

Fig. 1. Motivation of seeded clustering: (a) input data items, (b) structure of target R-tree, (c) clusters, (d) placement of the input data items by standard insertions and (e) placement of the input data items by a seed tree.

information from the seeding tree (that corresponds to the target tree of ours) such as MBRs or the center points of the MBRs. The seeded tree aids in reducing the spatial join cost by joining only the related part of a data set with others. This is achieved by the growth levels. From this part, the "seeded tree" is totally different from the seed tree.

Before describing how to create a seed tree, let's first look into the way this seed tree helps clustering. As described above, we utilize the structural information of the target R-tree in clustering the input data items. We use only a part of the target tree to reduce clustering overhead and speed up clustering. By using a seed tree, we attempt to group input data items that might be placed in the same node of the target R-tree if they were inserted using normal insertion method. For each of the input data items, we perform a process similar to an insertion operation as follows.

From the root node of a seed tree, an entry whose MBR fully encloses a data item is chosen. For simplicity, let's say we choose the first entry encountered. Then, we proceed to the child node pointed to by this entry and repeat the same steps until we reach the leaf level of a seed tree. If we can reach a leaf node of the seed tree, it means that this data item is spatially enclosed in this leaf node. If it fails to find an entry that satisfies the criteria in some non-leaf node of a seed tree, we stop clustering for the data item and classify it as an *outlier*.

To create a seed tree, we first need to decide the number of levels k to copy from the target R-tree. We choose k such that the insertions of data items in one cluster do not cost any additional I/Os. We can calculate the height of an R-tree that fits in the memory as follows. We estimate the height of the tree by assuming that each node is full. This is because when we build an input R-tree, we use the bulk loading method. Since bulk loading methods always try to fill the node by full capacity, we can roughly estimate the height of the R-tree before really building it from the number of items.

Suppose the maximum fanout of the input R-tree is M and the height of the target R-tree is h_t. If there are N items in the input data set, we can decide the value k as follows. There are n_c clusters which we can get from the number of nodes in the level $h_t - k$ of the target R-tree. This is because the number of clusters is same as the number of leaf nodes of the seed tree in most cases.

$$h_i = \lceil \log_M (N/n_c) \rceil \tag{1}$$

$$k = h_t - h_i + 1 \tag{2}$$

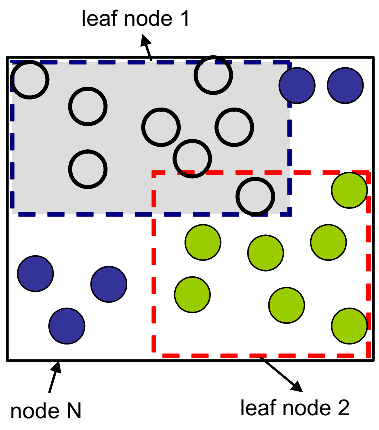

Fig. 2. Clusters and outliers.

Of course, in the worst case, every input items are clustered into one cluster. In that case, $h_i = \lceil \log_M N \rceil$.

During the classification of input data items, there can be more than one entry 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 easiest and fastest way. However, there can be alternative ways to select an entry which might lead to a better index structure. One of the possible ways is to choose an entry whose area is the smallest among them. Clusters generated in this way are likely to cover smaller area than the clusters obtained by selecting in the naïve way. Alternatively, we can choose an entry whose center is the closest to that of an input data item. This concentrates on the relationship between a data item and an entry. However, our preliminary results showed no significant differences, so we used the first entry to simplify the seeded clustering step.

4.2. Outliers

An outlier is an input data item that is enclosed in some internal node but in no leaf node of a seed tree. In Fig. 2, data items that reside inside the leaf node 1 are placed in cluster 1. Another group of items that are inside of leaf node 2 form another cluster 2. Remaining data items are classified as outliers. We insert outliers into the target R-tree using the standard insertion method individually.

One thing to notice here is that outliers exist only in the part where leaf nodes of a seed tree do not cover. In most cases the ratio of the outliers to non-outliers is so small that the overhead of inserting outliers one by one is negligible.

5. Naïve approach

In this and next sections, we describe two different algorithms for insertion. We first propose a naïve algorithm called *Seeded Clustering Individual insertion* (*SCI*).

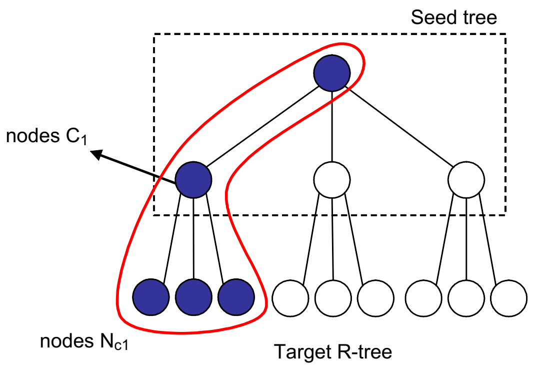

Fig. 3. Localized node access in Seeded Clustering Individual insertion (SCI).

This is basically same as the standard insertion method of an R-tree except that we insert items after seeded clustering step in such a way that the next item to insert is spatially close to the one inserted in the previous insertion operation. Seeded clustering has the effect of grouping together spatially close data items. If we insert input items in random order, we have to access some disk pages more than once because of the limited size of a buffer pool. However, if we process input items in the order of clusters, we can localize the disk access and take advantage of higher hit rate in the buffer. In Fig. 3, to insert all the input items in a cluster under node C_1, we only need to access nodes N_{C_1}. We can make the subtree that is rooted at node C_1 fit in the memory by choosing the height of the seed tree calculated using the memory size and the subtree size. We can process a cluster by loading related nodes into memory and perform insertions in memory and then write back each node to the disk. This means that the number of disk I/Os will be bounded by at most two times the number of nodes of the resulting R-tree. We will show you the experimental results in Section 8 that SCI dramatically improves the insertion cost. However, since it is fundamentally identical to the standard insertion method of an R-tree, it does not improve query performance. This is also given in Section 8.

6. Bulk insertion (SCB)

Although SCI improves the insertion cost dramatically, it does not help to improve query performance. In this section, we propose an algorithm that can improve both insertion and query cost. Before we begin, let's define the leaf level of the R-tree as level 0. And the level of the parent is greater than the level of the child by 1. Therefore, the height of an R-tree is 1 + the level of the root node.

As described in Section 4, we get a number of clusters and outliers after clustering using a seed tree. To perform bulk insertion, we first create a packed input R-tree from each of the clusters. We use a bulk loading method (e.g., [8–12]) to build an input R-tree from each cluster.

After creating an input R-tree, we insert input data items in a cluster in bulk into the target R-tree by inserting the input R-tree at once. During the insertion, there are a few things to consider for the resulting R-tree to be a legitimate R-tree. In addition to this, we need some post processing to improve the query performance. We discuss the post-processing in Section 7.

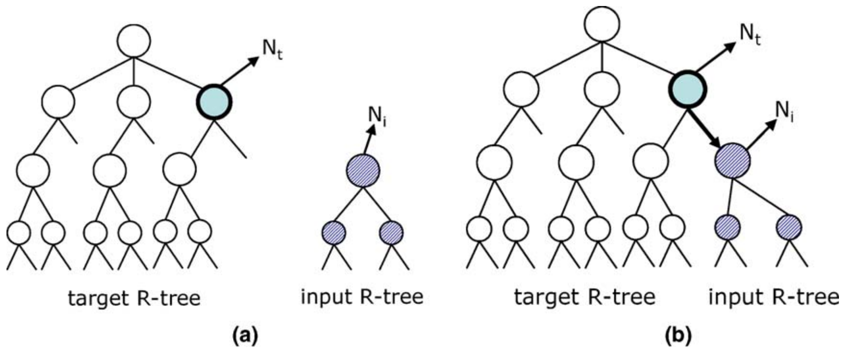

Fig. 4. Inserting an input R-tree into the target R-tree: (a) the target R-tree and an input R-tree. The input R-tree is going to be inserted into the node N_t and (b) the target R-tree after insertion.

To insert an input R-tree into the target R-tree, we treat the root node N_i of the input R-tree as a data object. We insert N_i using the standard insertion method. The only difference from a real data object is that this object should not be inserted into the leaf level of the target R-tree.

In an R-tree, all the leaf nodes should be in the same level of the tree. For the target R-tree to be a legal R-tree after inserting an input R-tree, we have to find a suitable level of a node to insert N_i that represents an input R-tree. Let the height of the input R-tree be h_i. Then we have to insert N_i into the level $h_i + 1$ of the target R-tree in order to ensure that the resultant target tree remains balanced. This is depicted in Fig. 4.

An input R-tree is built from one of the clusters, and this cluster is composed of the input data items that are fully included in a leaf node of the seed tree, say N_s. Therefore, we can say that the corresponding node, say N_t, of the target tree is the target node into which an input R-tree should be inserted according to the definition of a seed tree. However, depending on the data distribution of the input data set, the number of input data items in a cluster can vary and the height of an input R-tree created from this cluster may also vary. Therefore, simply inserting an input R-tree into the target node will not always work. However, the target node can be a good start point of finding a proper place to insert an input R-tree.

Ideally, every input R-tree fits in the target node if there are a proper number of data items in the corresponding cluster. However, since a seed tree is designed to help clustering of input data items but cannot control the number of data items classified into each of the clusters, an input R-tree may or may not fit in the target node. This input R-tree placement can be divided into three cases by the number of data items in a cluster. In the following sections, we will present a detailed look into each of these three cases.

For description, let R_i denote an input R-tree constructed from the data items in a cluster corresponding to the target node N_t. Let N_t be a node in level l_t and the height of R_i be h_i.

6.1. Case 1: input R-tree fits in the target node

Suppose the number of data items clustered in N_s is in the range such that the height of an input R-tree for the data items is equal to l_t. Then we just insert an entry, which points to the root node of R_i, into the node N_t. Although it may be a valid target R-tree without further processing (except for the case where the root node of R_i is underflow), it is likely to have large overlap for node

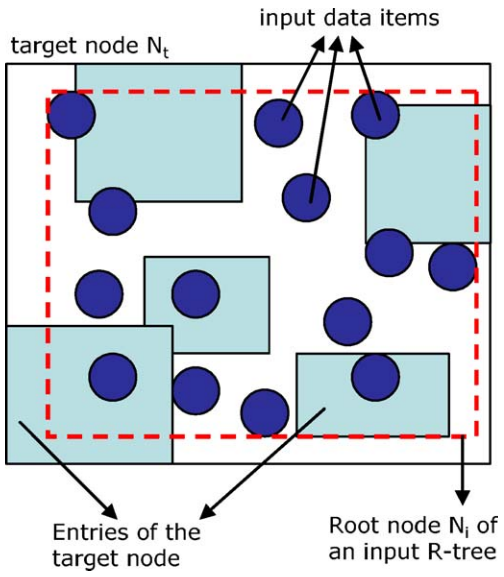
input data items
target node N$_t$
Entries of the target node
Root node N$_i$ of an input R-tree

Fig. 5. Input R-tree R_i and entries of the target node N_t are likely to have large overlap area.

N_i and entries of N_t as shown in Fig. 5 since the input data items in a cluster are likely to spread all over the area of the target node N_t. We need to perform post-processing called "repacking" to improve the query performance. This process restructures the target R-tree to improve its query performance. Repacking is described in Section 7.1 in detail.

6.2. Case 2: input R-tree is not shorter than the level of the target node

Suppose input data items are skewed so that a large number of data items are clustered into the node N_s and the height of an input R-tree R_i built from this cluster becomes greater than l_t. In this case, R_i does not fit in N_t. Since R_i is spatially enclosed by MBR of the target node, as shown in Fig. 6(b), its entries are fully included in MBR of N_t. Therefore we insert each subtree of the root node of R_i into the target node N_t. For each insertion of the subtrees, repacking step is required.

6.3. Case 3: input R-tree is shorter than the level of the child level of the target node

If there are too few data items clustered in the node N_s, the input R-tree R_i built from this cluster will be too shallow to be a direct child of N_t. An example is depicted in Fig. 7(a). For the resulting R-tree to be a valid R-tree, R_i should be inserted into one of the entries of the target node N_t. However, data items clustered in N_s are likely to be spread over the area of N_t, as shown in Fig. 7(b). If we insert R_i into one of the entries of N_t, MBR of that entry will become too large and its query performance will be degraded. Therefore, an input R-tree R_i should not be inserted as a whole as in the previous cases, but be processed in another way.

A naïve solution is to reinsert all the data items stored in R_i into the subtree whose root node is N_t. Although this increases the number of inserting operations, we can take advantage of the locality, since we are inserting data items in a relatively small area into a small subtree which may fit in memory.

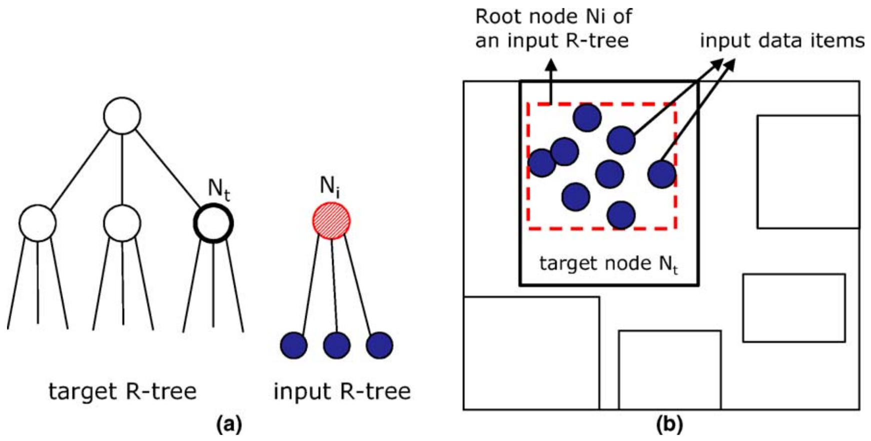

Fig. 6. (a) Data items are skewed that the root node N_i of an input R-tree and the target node N_t are on the same level and (b) N_t fully encloses the input R-tree by definition of a seed tree.

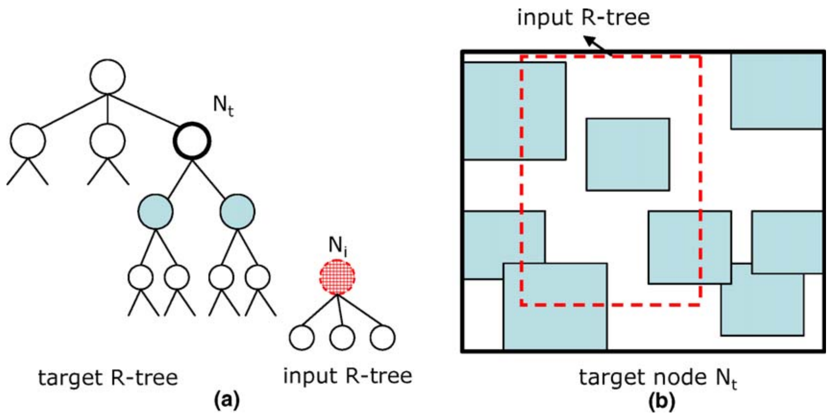

Fig. 7. (a) Data items are sparse and thus the input R-tree is too shallow to be an entry of N_t and (b) N_t fully encloses the input R-tree by definition of a seed tree.

Nonetheless, we can solve this problem more intelligently during the seeded clustering step. After clustering is done and right before creating input R-trees from these clusters, we can predict the height of each input R-tree. This is because we build an input R-tree using a bulk loading method. Since the bulk loading method attempts to fill every node, we can estimate the height of the resulting input tree using the number of data items in a cluster without actually creating an input R-tree.

From this, we can find out whether the input R-tree will fit the target node N_t or not. If the height of an input R-tree is found out to be $h_i < l_t$, then we extend the seed tree by one level for the path from root node to N_s and perform clustering again. Clustering is done for data items clustered in N_s only. We do not have to restart clustering from the root node of a seed tree but from the node N_s. The extended tree is shown in Fig. 8.

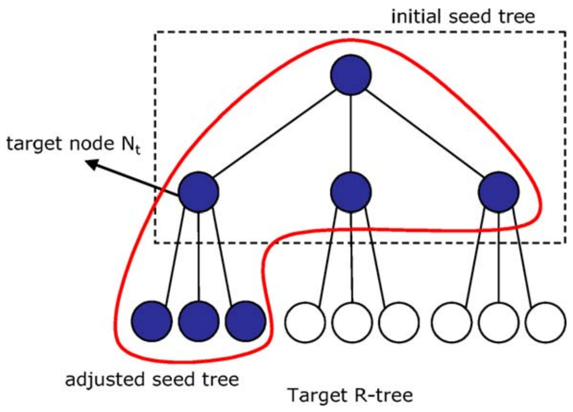

Fig. 8. Dynamically adjusted seed tree.

In this figure, N_s is a leaf node of the initial seed tree. After the first seeded clustering step is done, we find out whether the number of input items clustered in N_s is too small and the path from the root to N_s should be adjusted. For such paths, we *dynamically increase the height of a seed tree* and perform the clustering again with the data items clustered in N_s only. We can predict which path should be adjusted after the first round of the seeded clustering step. This can be done recursively and the recursion stops when the $l_t = 0$, in which case an input R-tree always fit in N_t.

7. Issues in inserting input R-trees

7.1. Repacking for local reduction in the overlapped area

Since there can be a large overlap between input R-tree(s) and the target node N_t, it is desirable to reduce overlap between them to make the target R-tree efficient for query processing. There was a work that performs post-optimization of existing R-trees by occasionally execute restructuring process while inserting data objects [13]. It improved point query performance while incurring a little increase in build cost. We also perform restructuring called *repacking*. As previously shown in Fig. 5, it is likely for an input R-tree to overlap the MBRs of the entries of the target node N_t. Since input data items in a cluster are apt to spread over the area of the target node N_t, many entries may overlap the input R-tree. Therefore, we unpack all the entries that overlap the input R-tree and the root node of the input R-tree itself and repack them to get new MBRs. By doing this, we can locally reduce overlap between nodes.

Detailed algorithms are given in Algorithms 1 and 2.

Algorithm 1. Post-process: Local reduction in the overlapped area

```
Function PostProcess ($N_t$, $R_i$)
$R_i$: an input R-tree that was inserted
$N_t$: node into which an input R-tree is inserted
```

```
begin
  $N_i \leftarrow$ the root node of $R_i$
  EntriesToRepack $\leftarrow N_i \cup$ entries of $N_t$ that overlap with $N_i$
  RepackedNodes $\leftarrow$ REPACK (EntriesToRepack)
  replace EntriesToRepack with RepackedNodes
end
```

Algorithm 2. Repack entries

```
Function REPACK (EntriesToRepack)
begin
  $ent \leftarrow \emptyset$
  foreach $e \in$ EntriesToRepack do
    $n \leftarrow$ node pointed to by $e$
    $ent \leftarrow ent \cup$ entries of $n$
  endfch
  return create nodes that enclose the elements of $ent$ using bulk loading method
end
```

7.2. Node underflow

There is another important property of an R-tree that must be satisfied. In an R-tree, every node except the root node must have at least the minimum number of entries m [1]. *Node underflow* occurs when a node contains less than m entries [14]. 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 because the root can have less than m entries. However, the target R-tree after inserting this input R-tree breaks the rule because the root of the input R-tree is now an internal node of the target R-tree which should have no less than m entries.

We solve this problem during the repacking step. When the input R-tree fits in the target node as described in Section 6.1, we unpack the root node of an input R-tree and entries of the target node that overlap it. After unpacking, we repack the entries to create new MBRs. During this, we can control the number of MBRs created using the number of nodes we unpacked. Suppose we unpacked $n + 1$ (including the root node of an input R-tree) nodes. If the root node of an input R-tree underflows, we can create n or $n + 1$ new nodes depending on the sum of entries to pack. Since the original target node had at least n entries, creating n new nodes does not make the target node underflow. On the contrary, if we created $n + 1$ MBRs because we cannot create n nodes out of the unpacked entries, the target node can overflow. In this case, we just split the target node as usual.

For the case described in Section 6.2, there is no underflow. This is because we insert subtrees of the root node of an input R-tree into the target node and each root node of subtrees is an internal node of an R-tree (input R-tree). Thus it does not underflow. The case in Section 6.3 can be transformed to one of the two cases described above.

8. Experiments

In this section, we present the results of an extensive experimental study to show the validity and the effectiveness of our approach.

8.1. Experimental setup

We have implemented a disk based R*-tree in C++ on windows 2003 server. We used the direct I/O feature of windows to prevent OS from buffering the disk pages. In the implementation of an R*-tree, we directly managed the LRU buffer. We restricted the buffer size so that only a limited size of the main memory is used. A node corresponds to a 4KB disk block and can hold approximately 100 two-dimensional entries per node. We compared our experimental results with the standard insertion method of an R-tree and GBI [3]. We presented the insertion cost and the query cost in average number of disk I/O. We also presented time spent for performing insertion. Since OS buffering prevents evaluating the net time spent, we used windows to always read/write disk pages from the disk. From now on, we will use SCI to represent *Seeded Clustering Individual insertion* and SCB to represent bulk insertion by seeded clustering algorithm.

We used two synthetic data sets having different characteristics as well as real data set.

TIGER/Line data: For the real data set, we used the standard benchmark data used in spatial databases, namely rectangles obtained from the TIGER/Line data set [15]. We used 2,249,727 streets of California [16]. As we need a data set for a target R-tree and the input data set, we randomly selected an input data set out of these data. Target data items and input data items are exclusive (see Fig. 9).

Data with uniform distribution: We created a data set with 500,000 data items for the target R-tree. In the uniformly distributed data set, the upper left corner of each square is uniformly distributed over the unit square. The area of the square is uniformly distributed between 0 and 2 times the average area. The value of the average area of a square is determined by the density

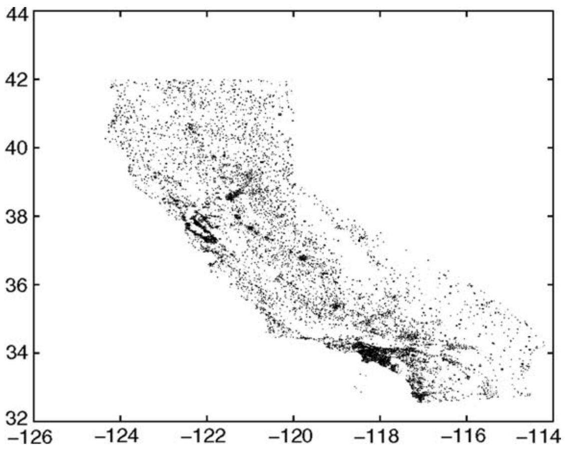

Fig. 9. TIGER/Line street data of California area.

of the data set, where density equals the sum of the areas of all the rectangles in the data set [8]. The lower right corner is chosen to give the desired area unless it exceeds the bounds of the unit square, in which case the coordinate(s) that exceeds 1.0 is set to 1.0. We used data density of 0.5.

Data with Zipfian distribution: We created a data set having non-uniform distribution, or the Zipfian distribution. The data items are skewed to the upper left part of the unit square. As the items are skewed to the side of the unit square, the input R-trees are likely to have different height depending on their location. The size and the density of the data sets used are the same as that used in the uniformly distributed data [17].

8.2. Insertion cost and query cost for the different data sets

In order to examine the effect of the seeded clustering, we first performed experiments where we compared the performance of our two methods (SCI and SCB) with R*-tree and GBI. We performed experiments with a buffer of 5% the size of the initial target R-tree. Initial target R-tree had 500,000 data objects for the synthetic data set and 1,249,848 data objects for TIGER/Line. We inserted from 10% to 80% of the number of data items stored in the initial target R-tree as input data items. Usually the buffer size of the operating systems are about 10–20% of the total memory size. It is reasonable for us to use 5% of the initial tree (not of the total memory) because in disk-based index we can assume that the data size are usually much larger than the available memory size.

Fig. 10 shows the insertion cost with uniformly distributed input data set. In Fig. 10(a), insertion cost in number of disk I/O's is given and in Fig. 10(b), R*-tree is omitted to zoom in the other methods. Fig. 10(a) shows that both of our algorithms dramatically improve the insertion cost in terms of the number of disk I/Os over normal R-tree. Depending on the size of input data set, our algorithms reduce the number of I/Os by a factor of 8–40. This can be explained as follows: in SCB and SCI, we process input data items in units of a cluster, so that the disk accesses are localized which causes all the disk pages accessed to be kept in memory. For GBI, since it does nothing after inserting small trees (corresponding to input R-trees), it shows the least disk accesses as we can see in Fig. 10(b).

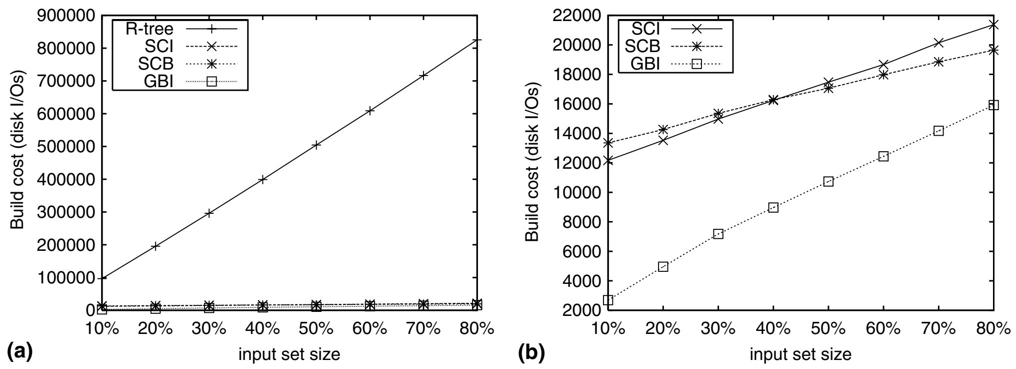

Fig. 10. Insertion cost in disk I/Os for uniformly distributed input data: (a) for all four methods and (b) GBI, SCI and SCB zoomed in.

We also presented the total time spent to perform bulk insertion in Fig. 11. In most method, elapsed time reflects the result of Fig. 10. However, we can find out that the overhead of clustering in GBI showed longer time spent than SCI or SCB. Since GBI clusters input data set using general clustering methodology, it requires more processing cost than SCI and SCB.

In Table 1, we presented the throughput (the number of insertions per second) of each method for each data set. We can clearly find out how effective and scalable our SCB is. In each data set, by using SCB, we can insert about 3–8 times more than GBI wt can insert about 40–80 times more data in the same time using SCB than R-tree.

In Fig. 12, SCB shows an outperforming result against other methods as expected. This is because of the repacking during insertion. Without repacking, an input R-tree will unavoidably show increased overlap with nodes in the target R-tree. In range queries given in Fig. 12(b), influence of reduced overlap decreases since it is likely for a query region to span overlapping nodes even if the area of overlapping region was reduced. However, even in the range query result, SCB

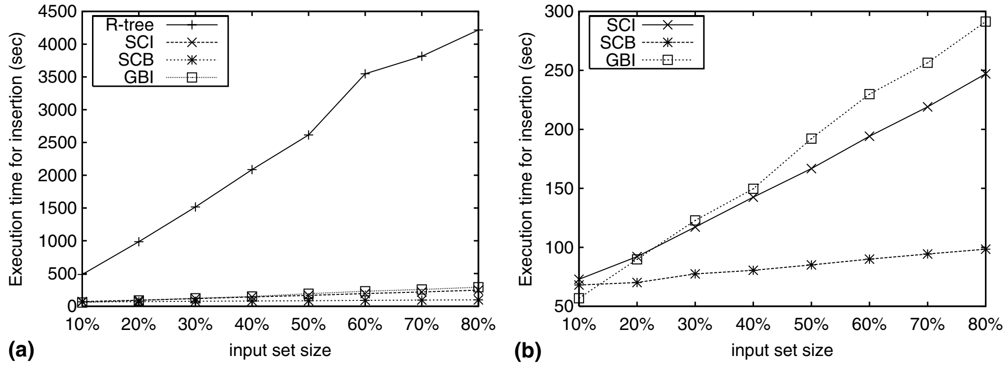

Fig. 11. Insertion cost in time for uniformly distributed input data: (a) for all four methods and (b) GBI, SCI and SCB zoomed in.

Table 1
The throughput (# of insertions per second) of each method for each data set

Data set	Method	10%	20%	30%	40%	50%	60%	70%	80%
Uniform	R-tree	206	203	198	192	191	169	183	190
	GBI	1759	2226	2443	2673	2602	2610	2730	2746
	SCI	1372	2171	2560	2807	2999	3091	3195	3237
	SCB	1470	2850	3872	4970	5874	6669	7422	8128
Skew	R-tree	185	180	192	191	170	186	135	149
	GBI	1527	2133	2372	2358	2471	2429	2529	2379
	SCI	1084	1816	2215	2466	2585	2601	2834	2938
	SCB	1222	2358	3252	4267	5015	5760	6272	6982
TIGER/Line	R-tree	3929	3508	3003	3006	3052	2752	2440	2137
	GBI	1936	2252	2454	2448	2591	2610	2733	2642
	SCI	4275	4461	4442	4360	4288	4262	4214	4274
	SCB	6015	8901	11,189	12,470	14,178	15,440	16,422	16,820

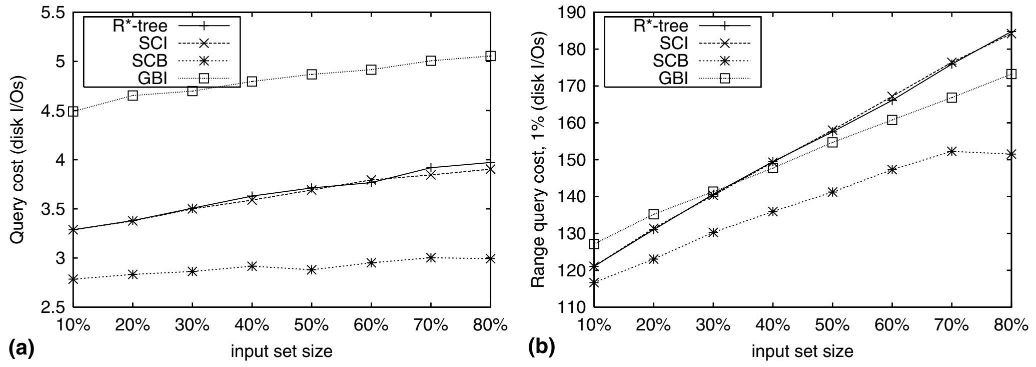

Fig. 12. Query cost in average disk I/Os for uniformly distributed data: (a) Point queries and (b) range queries of sizes 1% of the space.

shows improvement over other methods. We also presented graphs for skewed data set and TIGER/Line data set from Figs. 13–17.

We also include an experiment that the input data set and the existing data set had different distribution. This is included because each of the experiments we present so far used the same distribution for the input data set and the existing data set. In this experiment, we used the existing data set that had the upper left area (one fourth of the total area had no data points). As the input data set, we used only the data items in the uncovered area at upper left corner. We used five sets of input data items. Each covers 1%, 4%, 9%, 16% and 25% of the total area. The result is shown in Fig. 18. SCB still shows better result against the R-tree and the GBI.

8.3. Insertion cost depending on the buffer size

In this section, we made an experiment on insertion cost depending on the buffer size to see the buffering effect of our methods. Since SCI and SCB first cluster input data set using seeded clustering, we can localize the insertion of the input data items by processing in unit of a cluster. A cluster covers relatively small area that the nodes accessed during insertion are likely to fit in

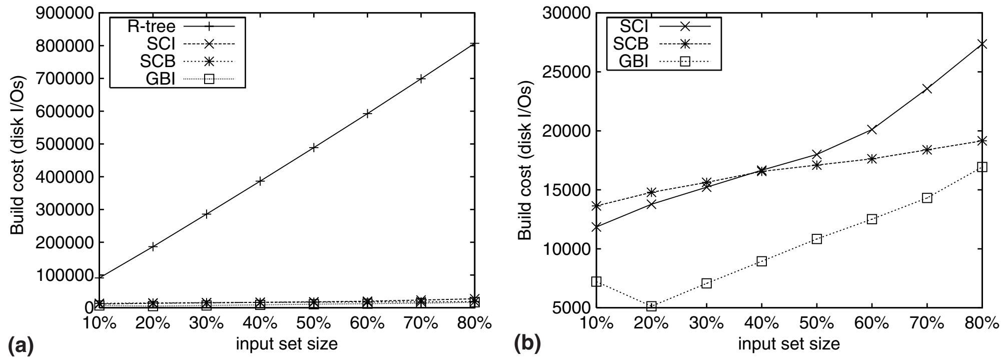

Fig. 13. Insertion cost in disk I/Os for skewed input data: (a) for all four methods and (b) GBI, SCI and SCB zoomed in.

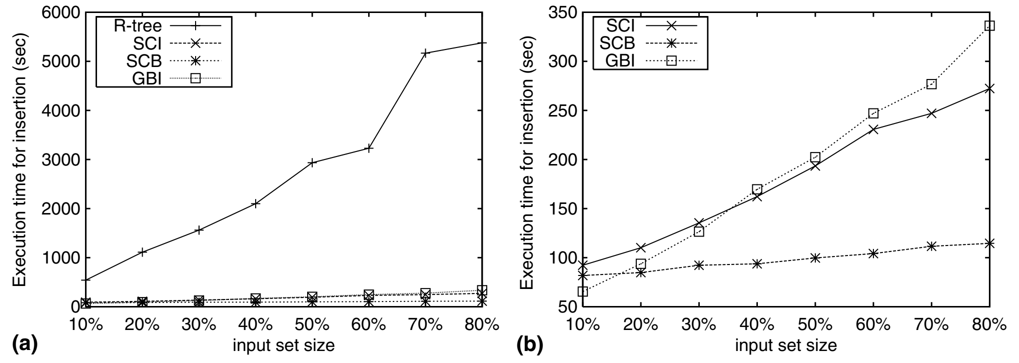

Fig. 14. Insertion cost in time for skewed input data: (a) for all four methods and (b) GBI, SCI and SCB zoomed in.

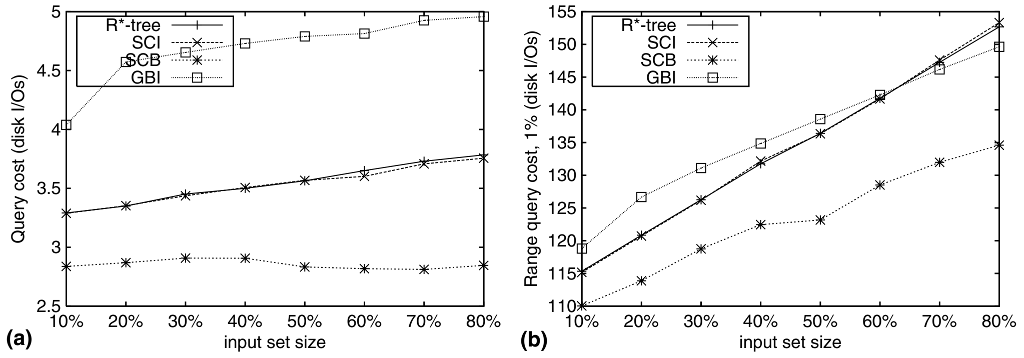

Fig. 15. Query cost in average disk I/Os for skewed input data: (a) point queries and (b) range queries of sizes 1% of the space.

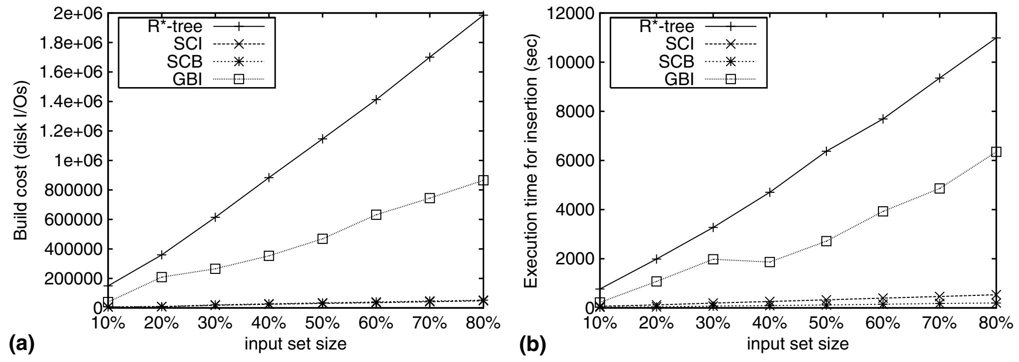

Fig. 16. Insertion cost for TIGER/Line input data set: (a) insertion cost in disk I/Os and (b) execution time.

buffer. By the experiment given in this section, only 3% of the total pages of the target R-tree are sufficient for SCI and SCB to efficiently perform insertion. Fig. 19 shows insertion cost of R*-tree,

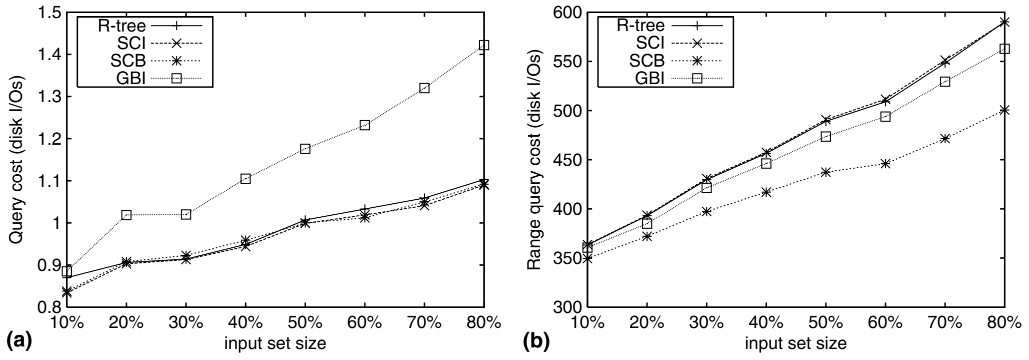

Fig. 17. Query cost in average number of disk I/Os for TIGER/Line input data set: (a) point queries and (b) range queries of sizes 1% of the space.

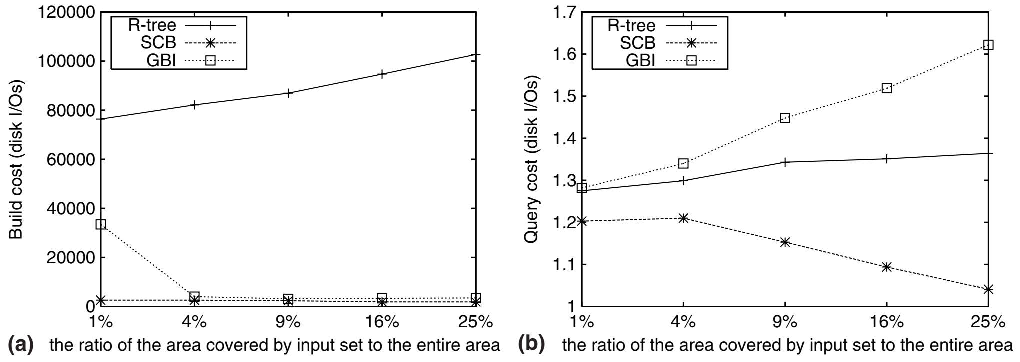

Fig. 18. Upper left corner (a fourth of the total area) is left out for initial target R-tree. Five sets of the input data items are inserted into the target R-tree. Each set has the data items of the upper left corner (covers 1%, 4%, 9%, 16% and 25% of the total area, respectively): (a) insertion cost for each method and (b) query cost for each method.

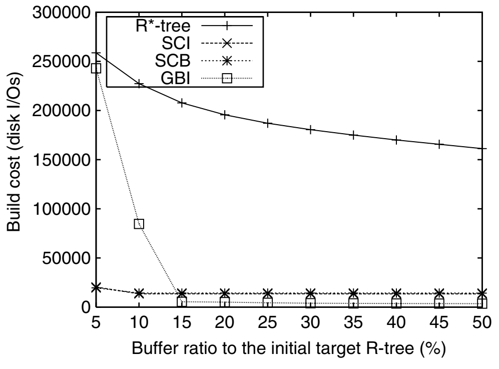

Fig. 19. Insertion cost depending on the buffer size. (uniformly distributed data).

GBI, SCI and SCB for uniformly distributed data set. Buffer size varies from 5% to 50% of the size of the target R-tree.

9. Conclusion

In this paper, we have proposed an effective bulk insertion method in the environment where data are continuously added to a spatial databases. We have proposed the *Seeded Clustering* technique which quickly and effectively clusters input data items by utilizing the structure of the target R-tree. By clustering, input data objects are grouped according to their locality and we can localize the access of nodes during the insertion of a cluster. We have proposed two bulk insertion algorithms SCI (Seeded Clustering Individual insertion) and SCB (Bulk Insertion by Seeded Clustering). SCI improves the insertion cost dramatically over the standard insertion method of an R^*-tree. However, its query performance gain was limited. SCB creates an R-tree from each of the clusters and inserts them one at a time by bulk which highly improves both insertion cost and query cost. SCB uses a repacking algorithm which reorganizes the structure of an R-tree after insertion. This requires additional CPU cost but it is worth while since the gain is considerable. Repacking also makes our algorithm scalable. As the data items are continuously added, our SCB method yields a good quality tree so that it outperforms previous bulk insertion methods in terms of query performance.

Acknowledgement

This work was sponsored in part by the Ministry of Information and Communications, Korea, through the ITRC program and the BK 21 program, Korea Science and Engineering Foundation (KOSEF) through the Brain Pool program, US National Science Foundation CAREER Award (IIS-9876037), NSF Grant No. IIS-0100436, and NSF Research Infrastructure program EIA-0080123.

References

[1] A. Guttman, R-Trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM-SIGMOD Conference, 1984, pp. 47–57.

[2] L. Chen, R. Choubey, E.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, 1998, pp. 161–162.

[3] R. Choubey, L. Chen, E.A. Rundensteiner, GBI: A Generalized R-tree Bulk-Insertion Strategy, in: Advances in Spatial Databases, 1997, pp. 91–108.

[4] I. Kamel, M. Khalil, V. Kouramajian, Bulk insertion in dynamic R-trees, in: Proceedings of the 4th International Symposium on Spatial Data Handling (SDH'96), 1996, pp. 3B.31–3B.42.

[5] L. Arge, K.H. Hinrichs, J. Vahrenhold, J.S. Vitter, Efficient bulk operations on dynamic R-trees, Algorithmica 33 (1) (2002) 104–128.

[6] L. Arge, The buffer tree: a new technique for optimal i/o-algorithms, Lecture Notes in Computer Science 955 (1993) 334–345.

[7] M.-L. Lo, C.V. Ravishankar, Spatial joins using seeded trees, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 1994, pp. 209–220.
[8] I. Kamel, C. Faloutsos, On packing R-trees, in: Proceedings of the second International Conference on Information and Knowledge Management, 1993, pp. 490–499.
[9] S.T. Leutenegger, J.M. Edgington, M.A. Lopez, STR: A Simple and Efficient Algorithm for R-Tree Packing, in: Proceedings of the IEEE Data Engineering, 1997, pp. 497–506.
[10] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed R-trees, in: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, 1985, pp. 17–31.
[11] Y.J. García, M.A. Lopez, S.T. Leutenegger, A Greedy Algorithm for Bulk Loading R-Trees, in: ACM-GIS '98, Proceedings of the 6th International symposium on Advances in Geographic Information Systems, November, 1998, pp. 163–164.
[12] J.V. den Bercken, B. Seeger, P. Widmayer, A Generic Approach to Bulk Loading Multidimensional Index Structures, in: Proceedings of 23rd International Conference on Very Large Data Bases, August, 1997, pp. 406–415.
[13] Y. Garcia, M.A. Lopez, S.T. Leutenegger, Post-optimization and Incremental Refinement of R-trees, in: Proceedings of the 1999 ACM GIS International Conference on Geographical Information System, 1999, pp. 91–96.
[14] N. Beckmann, H.-P. Kriegel, R. Schneider, B. 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, 1990, pp. 322–331.
[15] TIGER/Line Files, Technical Documentation, US Bureau of Census, Washington, DC (2000). Available from: <http://www.census.gov/geo/www/tiger/>.
[16] Y. Theodoridis, The r-tree-portal (2003). Available from: <http://www.rtreeportal.org>.
[17] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger, Quickly generating billion-record synthetic databases, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 1994, pp. 243–252.

Taewon Lee is a Ph.D. candidate in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea. He received his M.S. and B.S. degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1997 and 1999, respectively. His current research interests include spatial access methods, spatial and temporal databases, high dimensional index structures, and mobile data managements.

Bongki Moon is an Associate Professor of Computer Science at the University of Arizona. His current research areas of interest are XML indexing and query processing, information streaming and filtering, spatial and temporal databases, and parallel and distributed processing. He received his Ph.D. degree in Computer Science from University of Maryland, College Park, in 1996, and his M.S. and B.S. degrees in Computer Engineering from Seoul National University, Korea, in 1985 and 1983, respectively. He was a communication systems research staff at Samsung Electronics Corp. and Samsung Advanced Institute of Technology, Korea, from 1985 to 1990. He received the National Science Foundation CAREER Award in 1999.

Sukho Lee received his BA degree in Political Science and Diplomacy from Yonsei University, Seoul, Korea, in 1964 and his M.S. and Ph.D. in Computer Sciences from the University of Texas at Austin in 1975 and 1979, respectively. He is currently a professor of the School of Computer Science and Engineering, Seoul National University, Seoul, Korea, where he has been leading the Database Research Laboratory. He served as the president of Korea Information Science Society in 1994. He served as the honorary chair in the International Symposium on Database Systems for Advanced Applications (DASFAA), 2004. His current research interests include database management systems, spatial database systems, and multimedia database systems.