GBI: A Generalized R-Tree Bulk-Insertion Strategy *

Rupesh Choubey, Li Chen and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{rupesh|lichen|rundenst}@cs.wpi.edu

Abstract. A lot of recent work has studied strategies related to bulk loading of large data sets into multidimensional index structures. In this paper, we address the problem of *bulk insertions* into *existing* index structures with particular focus on R-trees – which are an important class of index structures used widely in commercial database systems. We propose a new technique, which as opposed to the current technique of inserting data one by one, bulk inserts entire new incoming datasets into an active R-tree. This technique, called GBI (for Generalized Bulk Insertion), partitions the new datasets into sets of clusters and outliers, constructs an R-tree (small tree) from each cluster, identifies and prepares suitable locations in the original R-tree (large tree) for insertion, and lastly performs the insertions of the small trees and the outliers into the large tree in bulk. Our experimental studies demonstrate that GBI does especially well (over 200% better than the existing technique) for randomly located data as well as for real datasets that contain few natural clusters, while also consistently outperforming the alternate technique in all other circumstances.

Index Terms — Bulk-insertion, Bulk-loading, Clustering, R-Tree, Index Structures, Query Performance.

* This work was supported in part by the University of Michigan ITS Research Center of Excellence grant (DTFH61-93-X-00017-Sub) sponsored by the U.S. Dept. of Transportation and by the Michigan Dept. of Transportation. Dr. Rundensteiner thanks IBM for the Corporate IBM partnership award, and Li Chen thanks IBM for the Corporate IBM fellowship as well as mentoring from the IBM Toronto Labora.

1 Introduction

1.1 Background and Motivation

Spatial data can commonly be found in diverse applications including Cartography, Computer-Aided Design, computer vision, robotics and many others. The amount of available spatial data is of an ever increasing magnitudes. For example, the amount of data generated by satellites is said to be several gigabytes per minute. Hence, efficient storage and indexing techniques for such data are needed.

Among many index structures proposed for spatial data, R-trees remain a popular index structure employed by numerous commercial DBMS systems [Inf] [Map]. The R-tree structure was initially proposed by Guttman [Gut84] and various variations and improvements over the original structure have been suggested ever since [BKSS90] [AT97] [TS94].

Generally, index structures need to be set up by loading data into them before they can be utilized for query processing. Initially, the basic *insert* operation proposed in [Gut84] was used to load sets of data into an R-tree, i.e., each object was inserted one by one into the index structure. In this paper, we will refer to this traditional technique of data sequential loading as the OBO (one-by-one) technique. The insertion of data sequentially into the R-tree has been found to be inefficient in the case when the entire tree needs to be set up [TS94]. Thus as an improvement, various *bulk loading* strategies have been proposed in recent years [LLG97, LLE97, BWS97, KF93]. These bulk loading strategies were aimed at creating a complete R-tree *from scratch*. These techniques thus assumed that totally new data was being collected (or existing data files were set up to be utilized by some application), and thus an index structure had to be constructed from scratch for this new dataset.

An upsurge of interest in spatial databases is how to efficiently manipulate existing massive amounts of spatial data, especially the problem of *bulk insertions* of new data assuming an already existing R-tree. The importance of this problem for numerous real-world examples is apparent. For example, new data obtained by satellites needs to be loaded into existing index structures. Active applications using the spatial data should continue functioning while being minimally impacted by the insertion of new data and in addition should be given the opportunity to make use of the new data as quickly as possible. The construction of a new index structure each time from scratch to contain both the old as well as the new data is not likely to scale well with an increasing size of the existing original index structure. Instead, new techniques specially tuned to this problem at hand are needed.

1.2 The Proposed Bulk-Insertion Approach

In our earlier work [CCR98a], we focussed on the problem of bulk insertion when the incoming dataset was skewed to a certain subregion of the original data. By skewed, we mean that the dataset to be inserted is localized to some portion of the region covered by the R-tree instead of being spread out over the whole region. In this paper, we extend the work and now provide a solution that deals with the general problem of bulk insertion of datasets of any nature instead of just skewed datasets. Both works look into the problem of bulk insertion of new datasets into an *existing active* R-tree. By active, we mean an R-tree which already contains (large) datasets, may have been used for some amount of time, and which has currently active applications that forbid the possibility of scheduling a long down-time for the index structure construction process.

However, in contrast to the prior STLT (Small-Tree-Large-Tree) technique [CCR98a], we now propose a new technique that is designed to handle bulk-insertion cases for different characteristics of the incoming datasets both in terms of the size of the new versus existing datasets as well as completely localized, partially skewed, somewhat clustered, or even completely uniform datasets. This technique, called GBI (for Generalized-Bulk-Insertions), partitions the new dataset into a set of clusters, constructs R-trees from the clusters (small trees), identifies and prepares suitable locations in the original R-tree (large tree) for insertions, and lastly performs the insertions of the small trees into the large tree.

Extensive experiments are conducted both with synthesized datasets as well as with real world datasets from the Sequoia 2000 storage benchmark to test applicability of our new technique. The results for both are comparable, thus indicating the appropriateness of the synthesized data for our tests. In our experiments, we find that GBI does especially well (in some cases even more than 200% better than the existing technique) for non-skewed large datasets as well as for large ratios of large tree to small tree data insertion sizes, while consistently outperforming the alternate technique in practically all other circumstances. Our experimental results also indicate that the GBI not only reduces the time taken for loading the data, but it also provides reasonable query performance on the resultant tree. The quality of the resulting tree constructed by GBI in terms of query performance is comparable to that created by the traditional tree insertion approach.

In summary, the contributions of this work include:

1. Design a general solution approach, GBI, to address the problem of *bulk insertions* of spatial data into *existing* index structures - in contrast to recent work on bulk loading data from scratch [LLE97] [LLG97] [BWS97] [KF93] and our previous work that deals with skewed datasets only [CCR98a].
2. Select and modify McQueen's k-means clustering algorithm to achieve the clustering desired by GBI.
3. Implement GBI and the conventional insertion technique in an UNIX environment using C++ in order to establish a uniform testbed for evaluating and comparing them.
4. Conduct experimental studies both using real-world as well as synthetic datasets. These experiments demonstrate that GBI is a winner, both in terms of substantial cost savings in insertion times as well as in keeping retrieval costs down.

The paper is organized as follows. Section 2 reviews related work. Section 3 defines the bulk insertion problem. Section 4 discusses our solution approach. Performance results for query tree loading and query-handling are presented in Section 5. This is followed by conclusions in Section 6.

2 Related Work

The general topic of bulk loading data into an initially empty structure has been the focus of much recent work. Two distinct categories of bulk loading algorithms have been proposed.

The first class of algorithms involves the bottom-up construction of the R-tree. Kamel and Faloutsos [KF93] use the Hilbert sorting technique to first order data and then build the R-tree. Leutenegger et al. [LLE97] proposed the STR (Sort-Tile-Recursive) technique in which a k-dimensional set of rectangles is sorted on an attribute and then divided into slabs. Both techniques are concerned with bulk loading and not with bulk insertion.

The other class of algorithms focuses on a top-down approach to build the R-tree. Bercken et al. [BWS97] adapt a strategy using a memory-based temporary structure called the buffer-tree. This technique is not likely to be very applicable when inserting new data over time – unless the temporary structure was continuously to be maintained along with the actual R-tree. This would be expensive, as buffer pages would be wasted. Arge et al. [AHVV99] presented a new buffer strategy for performing bulk operations on dynamic R-trees. Their method uses the buffer tree idea, which takes advantage of the available main memory and the page size of the operating system, and their buffering algorithms can be combined with any of the existing index implementations. However, their bulk insertion strategy is conceptually identical to the repeated insertion algorithm while we will present in this paper a conceptually unique bulk insertion strategy that can potentially be combined with their buffering algorithms.

[RRK97] discuss bulk incremental updates in the data cube. A portion of their work deals with bulk insertions of data which is collected over a period of time into R-trees. Their approach uses the sort-merge-pack strategy in which the incoming data is first sorted, then merged with the existing data from the R-tree and then a new R-tree is built from scratch. The strategy resolves back to eventually loading the tree up from scratch, whereas our approach avoids this for large existing trees prohibitly expensive step. [Moi93] suggests batching of data and sorting it prior to insertion. However, sorting phase is expensive and requires the data to be collected beforehand, while our algorithm tries to avoid the sorting phase. Bulk updates and bulk loading have been studied for various structures and in various scenarios [Che97, LRS93, LN97, CP98]. These techniques are typically specific to the structure in question and are not directly applicable to our problem.

Ciaccia et al. [CP98] proposed methods of bulk loading the M-tree which is possibly the work closest to ours. The proposed bulk loading algorithm performs a clustering of data objects into a number of sets, obtains sub-trees from the sets, and then performs reorganization to obtain the final M-tree. Our initial STLT algorithm is similar to the proposed algorithm in that STLT also constructs trees from subsets of data objects. However, in STLT, the choice of an appropriate location removes the need for reorganization of the tree in order to re-establish the balance of the tree. Again, the problem handled here was of bulk loading data and not of bulk insertion which is the focus of this paper. Heuristics of sizes of subtrees to insert were not developed, as done in our work [CCR98c].

Clustering algorithms have been the focus of extensive studies for long. Innumerable clustering algorithms with varying characteristics have been developed, such as [DO74, Spa82, War63, Rom84]. [Spa82] classifies clustering algorithms into the two classes of hierarchical and non-hierarchical algorithms. We do not attempt to improve any of the clustering algorithms nor suggest a new one. On the contrary, we simply select one of the algorithms based on our needs and apply it in our GBI solution.

3 Four Issues of Bulk Insertion

There are two conflicting goals for our proposed technique of bulk inserting data into an R-tree, the first being that the quality of the structure should be as good as possible and the second being that the time to load the new data into the R-tree should be minimized. This observation raises several issues to be addressed by our work:

- First, what is an effective strategy for inserting sets of data in bigger chunks rather than one-by-one so as to minimize down time for applications that use the R-tree structure?
- Second, which characteristics should these sets of to-be-inserted data objects (insertion sets) ideally possess so that the bulk-insertion strategy works most efficiently (in terms of both insertion time as well as resulting tree quality)?
- Third, how to group incoming possibly continuous streams of spatial objects into insertion sets so that each of them possesses desirable characteristics?
- Lastly, is it possible to design a framework which allows multiple solutions which have different balances between data loading times and the resultant tree quality?

For the first issue, we propose a new bulk-insertion strategy called the 'Generalized Bulk Insertion' (GBI) algorithm (see Section 4). GBI not only demonstrates efficient insertion times, but it also does have minimal impact on existing applications (1) by requiring no down time of the existing index by avoiding to build a new R-tree from scratch, and (2) by locking as few portions as possible of the R-tree for as short as possible a time (often only one single index node).

For the second issue, we conduct an extensive study that identifies numerous possible characteristics of the to-be-inserted data set in order to determine their impact on the insertion time as well as final tree quality (see Section 5). Characteristics of particular interest are (1) the number of objects in one insertion set (size of insertion set versus size of existing tree) and the spatial distribution of the new objects (skewness).

For the third issue, we could simply chop input streams into equally sized sets or as we do in our work employ a suitable clustering technique that takes data distributions into account (see Section 4). A new dataset is fed into a clustering tool that allows tuning of parameters to control the desired compactness and density of the clusters to be generated. Then clusters as well as outliers are ready for the next bulk insert phase, using STLT and OBO, respectively.

For the fourth issue, we provide a solution framework representing a mechanism for realizing multiple strategies for bulk inserting the new data. These parameters allow solutions which range from insertion of all the data in one shot to the insertion of data items one at a time. The middle portion of this range where data is inserted in multiple sets provides a compromise between loading time and tree quality. We provide a set of heuristics for selecting among the strategies for insertion [CCR98c].

4 GBI: A Generalized Bulk Insertion Strategy

4.1 The GBI Framework

As depicted in Figure 1, the 'Input Feed' module of GBI takes all the data to be inserted into the existing tree and gives it to the 'Input Feed Analyzer'. Analysis of the input data is done by the latter module, which passes the result of its analysis (number of incoming data, area of coverage, etc.) to the 'Cluster Detector'. The latter identifies clusters of suitable dimensions and accordingly separates the data into different clusters and a set of outliers. The generated clusters are used to construct a series of separate R-trees (small trees) by the 'Small Tree Generator' module. Note that this step can be done in parallel in order to improve performance. The 'Strategy Selector' module determines what strategy of insertion should be adopted, i.e., whether to gather all the data into one large single set and insert it in one shot or to group the data into multiple sets. This choice reflects the tradeoff between fast insertion of data and higher quality of the resultant tree. The small trees constructed are inserted into the existing large tree by the 'STLT Insertion' module. The outliers are inserted into the existing R-tree using the traditional (OBO) insert function.

The main steps of our Generalized Bulk Insertion (GBI) framework are the following:

1. Using our proposed heuristics, based on incoming data size and its area of coverage, determine values of the clustering modules' parameters.
2. Execute the clustering algorithm on the new dataset and obtain a set of clusters and outliers.
3. Build an R-tree (small tree) from one of the generated clusters.
4. Find a suitable position in the original tree (big tree) for insertion of the newly built small tree.
5. Handle unavailability of entry slot space in large tree using different heuristic techniques.

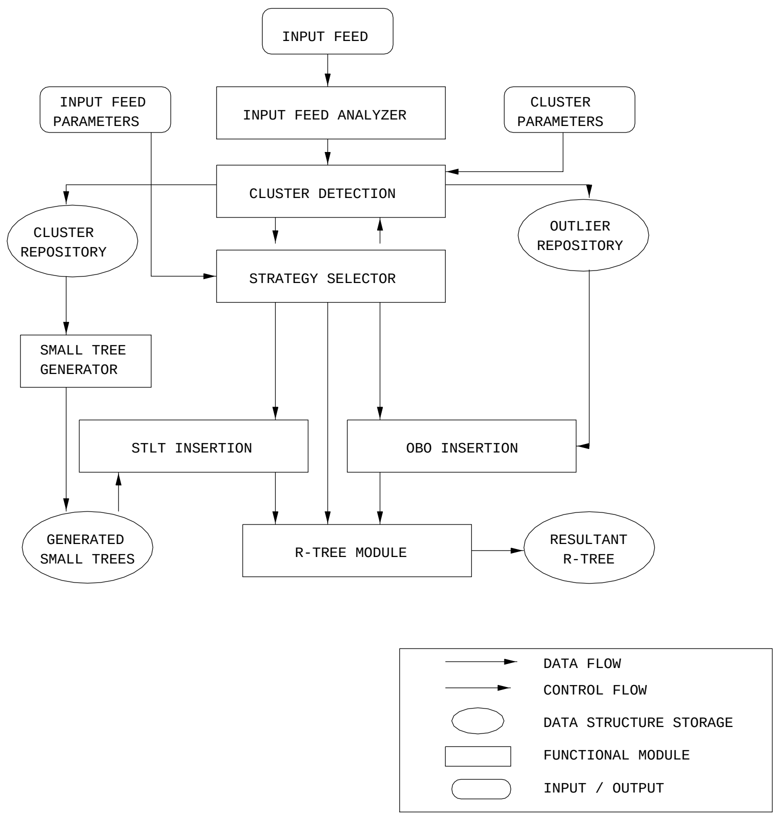

Fig. 1. GBI Framework

6. Insert the small tree into the identified location (or created as a result of Step 5).
7. Repeat steps 3 through 6 for each of the generated clusters.
8. Insert the outliers (also generated from step 2) using a traditional insertion algorithm.

A detailed explanation of the key steps of the GBI framework follows.

4.2 The GBI Terminology and Assumptions

For the following, we assume the terminologies and notations below :

$NumClusters$ = number of clusters generated by cluster generator
ST = newly built small tree
LT = original R-tree before insertion
ST_Root = root node of the small tree
LT_Root = root node of the large tree
$IndexNode$ = an index node of the large tree
M = maximum number of slot entries in a node
$DiffHeight$ = difference in heights between large and small trees

An explanation of each of the parameters employed in the clustering algorithm is as follows :

k : Initial number of clusters. This may increase or decrease depending on the number of clusters actually present and the ordering in which input data arrives or is considered. The final number of generated clusters is mainly dependent on the input dataset.

C : Coarsening value which determines how close one cluster can be to another. Any two clusters whose centroids are at a distance less then this value will be merged.

R : Refining value which determines how close a data point should be to a cluster to make the cluster a candidate for insertion of the data point. Any data to be inserted that is at distances greater than R from all the clusters will be inserted in its own separate cluster. If there are more than one clusters at distance less than R, then the data will be inserted in the closest cluster. The cluster where the data is inserted has its centroid recomputed and the possibility of cluster merges is examined.

fmin : Minimum flush value which determines how large a cluster should be (in terms of the number of data elements present) in order to consider it a candidate for a small tree. If the number of elements in a cluster is less than fmin, then the data elements are inserted into the outlier list.

fmax : Maximum flush value which determines the maximum size allowed for a cluster.

4.3 GBI Framework Description

First, the entire new dataset is fed to the clustering tool that then breaks up the dataset into appropriate clusters and a set of outliers. The generated clusters are controlled by a few parameters which determine the number of data items in a cluster and also the compactness or density of the generated clusters. The GBI algorithm given in Figure 2 and as visually depicted in Figure 3 first identifies clusters and outliers from the given input dataset and then builds a small tree (denoted by ST) from each of the clusters, as indicated by the function *BuildSmallTree()* invoked in the algorithm. Next, considering one small tree at a time, we compute the difference in heights of the large tree and small tree as this would determine how many levels we need to go down in the large tree in order to locate the appropriate place for insertion of the small tree. If the new data is larger than the existing data, then the proposed technique is not meaningful.

```
ALGORITHM GBI(LT_Root)
{
   SetClusteringParameters(f, C, R, K); // Initial tuning values
   while a pause ( > setted time period ) of inputing data {
      NumClusters = InvokeClusteringAlgo(DatasetFromInputFeed);
      repeat (for NumCluster clusters) {
         ST_Root = BuildSmallTree(next ClusterFile);                                          // Build an R-tree
         DiffHeight = LTHeight - STHeight; // Insertion level
         if (DiffHeight < 1)               // Large tree <= small tree
            adopt OBO insertion;
// not suitable for GBI bulk insertion
         else                              // Insert small tree
            InsertSTintoLT(LT_Root, ST_Root, DiffHeight);
      }
      InsertUsingOBO(OutliersSet);
   }
}
```

Fig. 2. The GBI Framework: Bulk Insertion of New Data into Large Tree

Then we instead simply apply one of the bulk-loading strategies to build a new tree containing both old and new data from scratch. Otherwise, we invoke the *InsertSTintoLT()* function to insert ST into an appropriate IndexNode in the big tree. The previous step is repeated for each of the small trees. Finally, once all clusters are exhausted, the outliers are inserted into the large tree one-by-one.

4.4 GBI Clustering Module

For the clustering, we use a variant of the MACQueen's k-Means Method [And73] with a suitable extension. This algorithm is chosen because it can be easily modified to allow for clusters of fixed maximum or minimum sizes. It can also be made to adapt for continuous incoming data. Figure 3 shows how clusters and outliers are formed from a given set of input data elements and how some of the formed clusters can be potentially merged (based on the values of the parameters of the clustering algorithm).

1. Select the proper values of the parameters : number of clusters k, coarsening value C, refining value R, and flush values *fmin* and *fmax*.
2. For the input, let each of the first k data unit be a cluster of size one.
3. Determine the minimal distance C between clusters. Merge the clusters with the distance of their centroids less than C until all the clusters are at a distance greater than C from one another.
4. Take the next data element and determine the cluster closest to it.
5. Decide to insert the data into its closest cluster if it is at a distance of less than R to that cluster, then recompute its centroid and merge clusters that

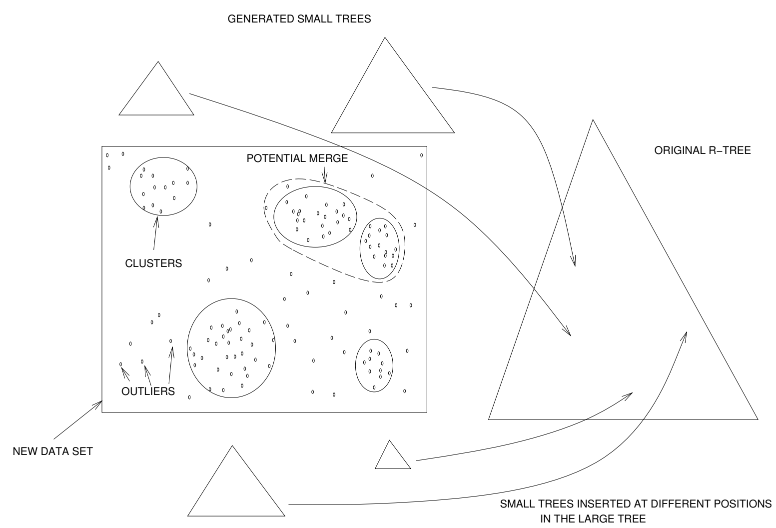

Fig. 3. Graphical Depiction of GBI Process

their distance become less than C. If the closest cluster is at a distance greater than R, take the data as a new cluster of one member.

6. Repeat Step 4 and 5 for the remaining data given by the Input Feed module.
7. Take each of the cluster centroids as fixed seed points and reallocate each data to its nearest seed point.
8. For each cluster, if the number of data items it contains is greater than f, then consider it as a cluster, otherwise insert it in the outlier list.

4.5 GBI - STLT Module

A detailed description of the STLT (small-tree-large-tree) strategy, including precise algorithms for the module to insert one small tree into an existing large tree, is given in [CCR98a], while below we give a brief summary of its basic ideas below. Let the height of the newly built small R-tree be h_r and the height of the original R-tree be H_R. We consider the root rectangle of the small R-tree (enclosing rectangle of all new data rectangles) as a data rectangle. In other words, we use the standard *insert* operation to find a suitable place to insert the newly built R-tree into the existing R-tree (referred to as the *large tree*) as if it were an individual data item. We try to insert it into the level $l = H_R - h_r$ of the original R-tree. Our goal here is to assure that the bottom level of the small R-tree is on the same level as that of the original R-tree as seen in Figure 4. This is in order to ensure that the resultant tree remains balanced which is a fundamental requirement for the structure to be an R-tree.

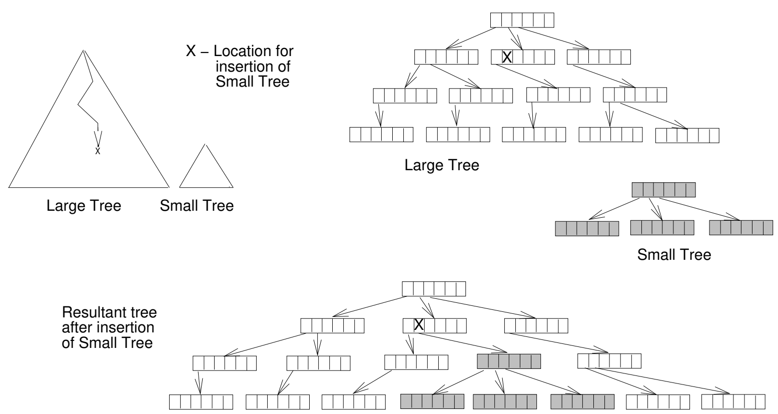

Fig. 4. Insertion of One Small Tree into the Large Tree

5 Experimental Evaluation of the GBI Framework

5.1 Experimental Setup

Testbed Environment: Our performance studies are conducted on a testbed on a SUN Sparc-20 workstation running the UNIX operating system. The testbed includes the original R-tree with Quadratic splitting [Gut84], an I/O buffer manager, modified MACQueen's k-means clustering module, input feed analyzer, strategy selector module and other supporting data structures and procedures.

Test Data: The test data comprised both real datasets and synthetic datasets. Most of our tests are based on synthetically generated testing datasets to be able to verify the usefulness of GBI under different extreme settings. The synthetic data was generated with varying parameters to control the distribution and the nature of the objects. The real data from the TIGER/Line files distributed by the US Census Bureau consists of a dataset of streets (131,461 objects) and a dataset of rivers and railway tracts (128,971 objects) from an area in California.

Test Types: We carried out two major classes of tests. The first type of experiments were conducted to compare the *I/O insertion costs* of GBI with OBO for different parameters. The tests were designed to evaluate the performance of GBI and compare that of OBO in terms of I/O costs for different parameters and determine the usefulness and limitations of GBI. The second type of experiments focussed on evaluating the *quality* of the resultant trees formed by GBI or OBO style of insertions. The tests comprised asking queries on the resultant trees to measure the number of nodes visited to answer the queries and the I/O cost incurred in answering the queries.

5.2 Experiments Measuring Insertion Cost and Query Cost for Different Data Area Ratios

This experiment is used to evaluate the performance of GBI as compared to OBO, when the area of the new dataset is varied from a small percentage of the large tree until it equals the area covered by the original tree. A set of 5000 data elements is inserted and the insertion times are measured. A set of 50000 random queries is asked on each of the trees generated after OBO insertion and after GBI insertion to evaluate the query performance. The number of elements in the large tree is 100000.

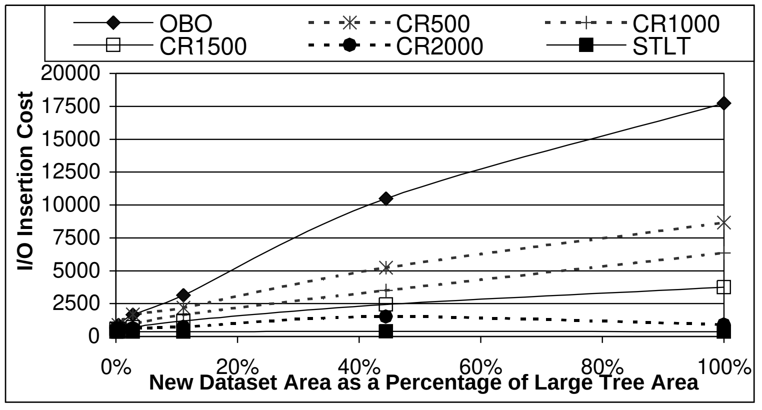

Fig. 5. I/O Insertion Cost for Different Area Ratio of Original Tree Dataset to New Dataset.

As shown in Figure 5, GBI wins over OBO for most ‘C’ and ‘R’ values. For larger ‘C’ and ‘R’ values, the improvement in insertion times is maximal. This is because, for larger values, many clusters are formed and there are few outliers. In such cases, the insertion cost is the sum of building the small trees and then inserting them one by one, which is less than inserting data elements one by one.

An important result is that as the new data becomes less and less skewed, the savings in terms of insertion times becomes more significant. This shows that the GBI algorithm proves to be useful for non-skewed random data. The reason for this improvement is to some degree the fact that for less skewed data, OBO is not able to exploit the locality of pages in the buffer as most elements belong to different pages. Hence, the I/O cost of insertion for OBO increases as the new dataset becomes less localized.

On the other hand, Figure 6 shows that the query performance of the GBI algorithm is comparable to the OBO query performance for smaller ‘C’ and ‘R’ values. This is because the clusters formed are much tighter if the values of ‘C’ and ‘R’ are lower. The figure also shows that GBI improves on the STLT [CCR98a] query performance while yielding significant savings in the insertion

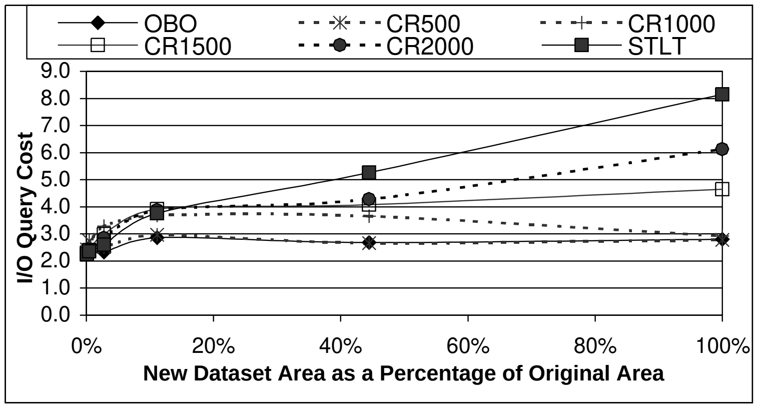

Fig. 6. Query Cost for Different Area Ratio of Original Tree Dataset to New Dataset.

time. This is done by ensuring that the data to be inserted into the original tree is clustered and thus each small tree does not cover too large an area of the large tree region.

5.3 Experiments Measuring Insertion Cost for Different Data Sizes

In this experiment, we keep the ratio of the large tree data size to the new data size fixed (at a value of 50) and instead vary absolute data sizes by increasing both small and large tree sizes at the same percentage. The area covered by both the original tree and the new dataset is the same.

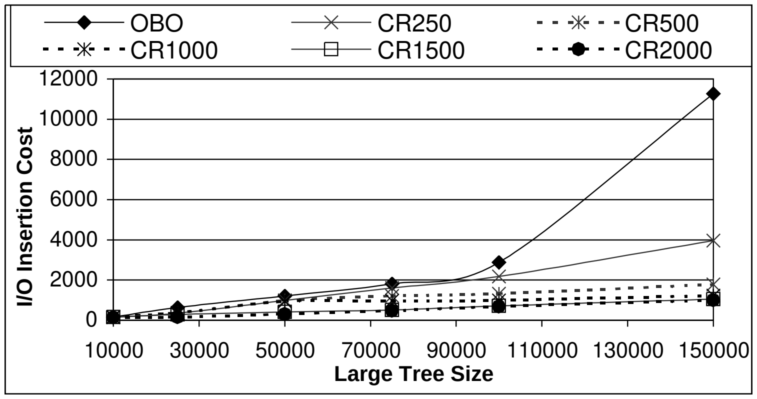

Fig. 7. I/O Insertion Cost for Different Ratios of Original Tree Size to the New Dataset Size

As seen from Figure 7, as the relative sizes of the original tree and the new dataset increase, the insertion cost for OBO increases fairly rapidly whereas the costs for GBI increase less rapidly. Thus, GBI yields more improvement for relatively larger sizes of original tree and new dataset. This experiment shows that GBI is scalable and is not constrained to small sized datasets.

5.4 Experiments on Determining Effect of 'C' and 'R' Values

In this experiment, we analyze the effect of varying 'C' and 'R' values on the insertion times and the resulting tree quality in terms of query performance. We insert a set of 5000 data elements randomly located over the region of the large tree and measure the insertion costs and query costs for 50000 random queries. By randomly located, we mean that the data elements are randomly spread out all over the area enclosed by the original tree elements which is 30000 by 30000.

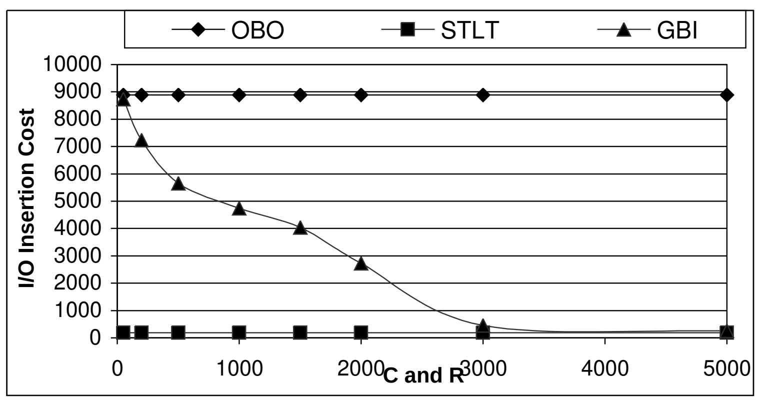

Fig. 8. I/O Insertion Cost for Different 'C' and 'R' Values

As can be seen from Figure 8, as the 'C' and 'R' values increase, the insertion time decreases. This is because for larger 'C' and 'R' values, larger size clusters are formed and there are fewer data elements which are inserted individually thus improving on the insertion times tremendously (100-fold improvement for 'C' and 'R' values greater than 3000). For low 'C' and 'R' values, there are fewer clusters which are used to construct small trees and most of the data not being part of a cluster but being an outlier is inserted using the OBO technique. For very small values of 'C' and 'R', the technique yields no small trees and hence there is no difference in OBO and the GBI insertion costs.

The query performance corresponding to the resultant trees from the above experiment is shown in Figure 9. This shows clearly that GBI yields trees of good quality when the 'C' and 'R' values are kept low. This is because low 'C' and 'R' values yield more dense clusters which keep the query costs low. For higher 'C'

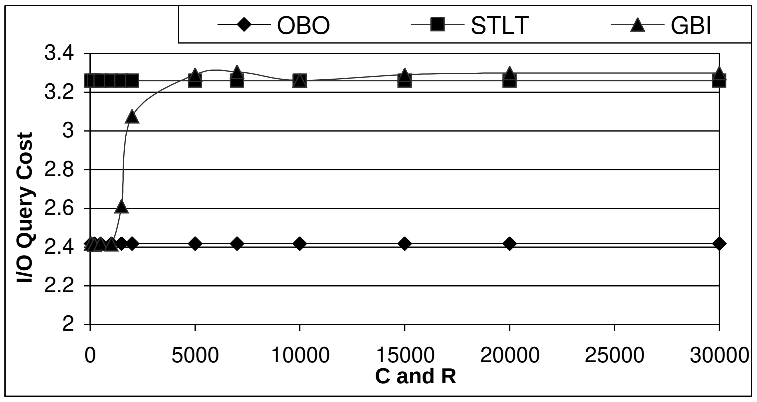

Fig. 9. I/O Query Cost for Different 'C' and 'R' Values

and 'R' values, the clusters formed are less dense and thus the trees generated using them are of slightly lower quality (in terms of intersecting MBRs) and thus the query performance of GBI becomes worse. This experiment yields a maximum of one extra I/O on the average for the tree constructed using GBI as compared to the one constructed using OBO. Clearly, settings for 'C and 'R' values must be empirically selected to find a trade-off between maximizing tree retrieval quality while minimizing tree insertion times.

5.5 Experiments on Real Datasets using Tuned GBI

Based on the experiments listed thus far and additional ones that can be found in our technical report [CCR98c], we have empirically determined values for initializing the key parameters of the GBI framework, such as values for *C, R, flushmin, flushmax*, etc. (see Section 4.1 for explanation of these parameters). A justification of this tuning of GBI, then called the GBI* heuristic strategy, can be found in [Cho99], while below we give some insight into the performance achieved by this tuned GBI*. The experiments are done with real data extracted from the Sequioa 2000 storage benchmark. The experiment uses different sizes of the large tree and new dataset but the ratio is kept fixed at 50.

Figure 10 clearly shows that GBI* wins out over OBO in terms of insertion costs. As the sizes of the datasets increase, the savings in insertion cost increase as well. Figure 11 displays the query costs for GBI* and OBO which are very similar thus indicating that the GBI* generated tree is of acceptable quality.

6 Conclusions

In contrast to previous work on bulk loading data which primarily focussed on building index structures from scratch, in this paper we tackle the problem of

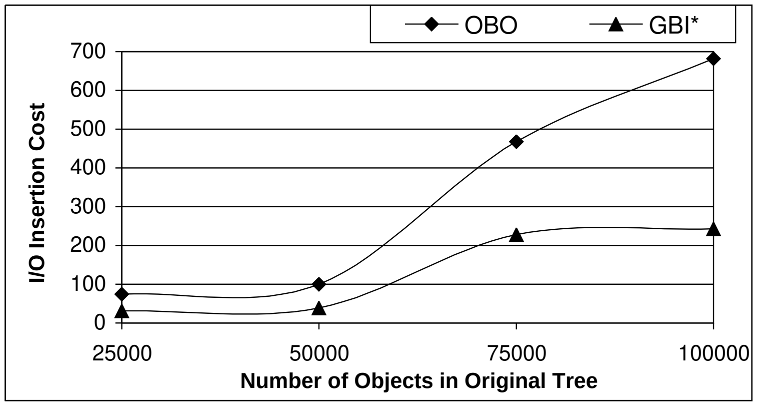

Fig. 10. Comparison of Insertion Costs of GBI* and OBO for Real Datasets

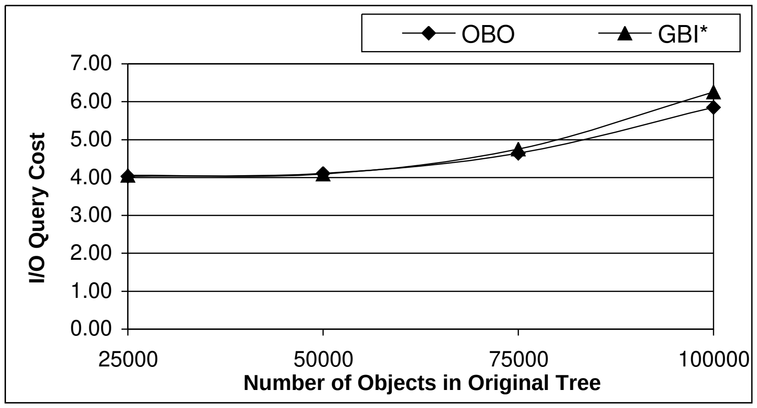

Fig. 11. Comparison of Query Costs of GBI* and OBO for Real Datasets

bulk insertions into existing active index structures. We extend our earlier work [CCR98b] where we addressed the problem for skewed datasets only. While our current work focuses on R-tree, we contend that the overall strategy of group-first and then insert-as-bulk is a general approach and thus could also be explored for other multi-dimensional index structures.

The proposed framework, called GBI (for Generalized Bulk Insertion), considers the new dataset as a set of clusters and outliers, constructs from the clusters a set of R-trees (small trees), identifies and prepares suitable locations in the original R-tree (large tree) for insertion of each of the small trees, and lastly bulk-inserts each small tree into the large tree.

We have reported experimental studies designed to not only compare GBI against the conventional technique, but also to evaluate the suitability and limitations of the GBI framework under different conditions. We have found that

GBI does especially well (over 200% better than the existing technique) for non-skewed datasets as well for large ratios of large tree sizes as compared to sizes of the new data insertions (see Figure 7), while consistently outperforming the alternate technique in all other circumstances. Our experimental results also indicate that the quality of the resulting tree constructed by GBI in terms of query performance is acceptable when compared to the resulting tree created by the traditional tree insertion approach. All in all, the GBI *bulk insertion* strategy has thus been found to be a viable and significant optimization over the conventional one-by-one insertion approach, gaining both in terms of update costs as well as preserving the resulting index access quality.

Possible future tasks include application of the general approach of GBI to other multi-dimensional index structures as well as experimental evaluation of alternate approaches against GBI such as from-scratch bulk-loading techniques.

Acknowledgments. The authors would like to thank first and fore-most Yun-Wu Huang, Ning Jing, and Matthew Jones who have developed most of the experimental system that served as platform for our work, including the R-tree index structure, storage manager, and buffer management modules. We are especially grateful to Yun-Wu Huang who continued to help with making our way through the prototype code and in many other uncountable ways. Lastly, we thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research.

References

[AHVV99] L. Arge, K. Hinrichs, J. Vahrenhold, and J. Vitter. Efficient Bulk Operations on Dynamic R-trees. *Workshop on Algorithm Engineering and Experimentation ALENEX 99*, pages 92–103, 1999.

[And73] M. R. Anderberg. *Probability and Mathematical Statistics.* Academic Press, New York, San Francisco, London, 1973.

[AT97] C.H. Ang and T.C. Tan. New Linear Node Splitting Algorithm for R-trees. *Advances in Spatial Databases*, pages 339–349, 1997.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree : An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of SIGMOD*, pages 322–331, 1990.

[BWS97] J. Bercken, P. Widmayer, and B. Seeger. A Generic Approach to Bulk Loading Multidimensional Index Structures. *International Conference on Very Large Data Bases*, pages 406–415, 1997.

[CCR98a] L. Chen, R. Choubey, and E. A. Rundensteiner. Bulk Insertions into R-trees using the Small-Tree-Large-Tree Approach. *Proceedings of ACM GIS Workshop*, pages 161–162, 1998.

[CCR98b] L. Chen, R. Choubey, and E.A. Rundensteiner. Bulk Insertions into R-Trees. *WPI, Tech. Rep. CS-WPI-98-05*, February 1998.

[CCR98c] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A Generalized R-Tree Bulk Insertion Strategy. *WPI Technical Report TR-98-15-STLT*, 1998.

[Che97] W. Chen. Programming with Logical Queries, Bulk Updates, and Hypothetical Reasoning. *IEEE Transactions on Knowledge and Data Engineering*, pages 587–599, July 1997.

[Cho99] R. Choubey. R-Tree Bulk Insertion Strategies. *Master Thesis in progress, Worcester Polytechnic Institute*, 1999.

[CP98] P. Ciaccia and M. Patella. Bulk Loading the M-tree. *Proceedings of the Australasian Database Conference*, February 1998.

[DO74] B. Duran and P. Odell. *Cluster Analysis - A Survey*. Springer-Verlag, Berlin, Heidelberg, New York, 1974.

[Gut84] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of SIGMOD*, pages 47–57, 1984.

[HJR97a] Y.W. Huang, N. Jing, and E.A. Rundensteinder. Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. *International Conference on Very Large Data Bases*, pages 396–405, 1997.

[HJR97b] Y.W. Huang, M. Jones, and E.A. Rundensteiner. Symbolic Intersect Detection: A Method for Improving Spatial Intersect Joins. *Proc. of the International Symposium on Spatial Databases*, pages 165–177, 1997.

[HNR97] Y.W. Huang, N.Jing, and E.A. Rundensteiner. A cost model for estimating the performance of spatial joins using R-tree. *International Working Conference on Scientific and Statistical Database Management*, pages 30–38, August 1997.

[Inf] Informix Corporation ("http://www.informix.com"). *Informix*.

[KF93] I. Kamel and C. Faloutsos. On Packing R-trees. *Proceedings of International Conference on Information and Knowledge Management*, pages 490–499, November 1993.

[LLE97] S. Leutenegger, M. Lopez, and J. Edgigton. STR: A Simple and Efficient Algorithm for R-tree Packing. *Proceedings of IEEE International Conference on Data Engineering*, pages 497–506, 1997.

[LLG97] S. Leutenegger, M. Lopez, and Y. Garcia. A Greedy Algorithm for Bulk Loading R-trees. Technical report, University of Denver Computer Science Technical Report # 97-02, 1997.

[LN97] S. Leutenegger and D. Nicol. Efficient Bulk-Loading of Gridfiles. *IEEE Transactions on Knowledge and Data Engineering*, pages 410–420, May 1997.

[LRS93] J. Li, D. Rotem, and J. Srivastave. Algorithms for Loading Parallel Gridfiles. *Proceedings of SIGMOD*, pages 347–356, 1993.

[Map] MapInfo Corporation. *SpatialWare - http://www.mapinfo.com/spatialware/spatial20.html*.

[Moi93] A. Moitra. Spatio-Temporal Data Management Using R-trees. *International Journal of Geographic Information Systems*, 1993.

[Rom84] H. Charles Romesburg. *Cluster Analysis for Researchers*. Lifetime Learning Publications, Belmont, California, 1984.

[RRK97] N. Roussopoulos, M. Roussopoulos, and Y. Kotidis. Cubetree : Organization of and Bulk Incremental Updates on the Data Cube. *Proceedings of SIGMOD*, pages 89–99, 1997.

[Spa82] H. Spath. *Cluster Analysis Algorithms for Data Reduction and Classification of Objects*. Ellis Horwook Publishers, Chichester, 1982.

[TS94] Y. Theodoridis and T. Sellis. Optimization Issues in R-tree Construction (extended abstract). *Lecture Notes in Computer Science*, pages 270–273, 1994.

[War63] J. H. Ward. Hierarchical Grouping Analysis For Applications. *Journal of American Statistics Association*, 58:236–244, 1963.