ACM - GIS SUBMISSION

Bulk-Insertions into R-Trees *

Li Chen, Rupesh Choubey and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{lichen|rupesh|rundenst}@cs.wpi.edu
(508) 831-5815

Abstract

A lot of recent work has focussed on bulk loading of data into multidimensional index structures in order to efficiently construct such structures for large datasets. Previous work on *bulk loading* data focussed at building index structures *from scratch*, while the problem of *bulk insertions* into *existing* index structures has been largely overlooked. In this paper, we address this new problem 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 datasets into an active R-tree. This technique, called STLT (for Small-Tree-Large-Tree), considers the new dataset as an R-tree itself (small tree), identifies and prepares a suitable location in the original R-tree (large tree) for insertion, and lastly performs the insert of the small tree into the large tree. Extensive experimental studies both on synthetic and real data sets from the Sequoia 2000 storage benchmark are reported in this paper. These experiments not only compare STLT against the conventional technique, but also evaluate the suitability and limitations of STLT under different conditions, such as varying buffer sizes, ratio between existing and new data sizes, and skewedness of new data with respect to the whole spatial region. We find that STLT does especially well (up to 60% better than the existing technique) for skewed datasets as well for large sizes of the large tree and small tree, while consistently outperforming the alternate technique in all other circumstances. Our experimental results also indicate that the quality of the resulting tree constructed by STLT in terms of query performance is acceptable and in a few cases preferable over that created by the traditional tree insertion approach.

Elke A. Rundensteiner is contact author of the paper.

Index Terms — Multidimensional Index Structures, Bulk-loading, Bulk-insertion, Query Performance, Spatial Query Processing.

* 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.

1 Introduction

Spatial data can commonly be found in many diverse applications including Cartography, Computer-Aided Design (CAD), computer vision and robotics and many others. Not only is the available amount of spatial data enormous, but also the data being acquired is of increasing magnitudes. For example, the amount of data generated by satellites is said to be several gigabytes per minute. Hence, efficient storage and indexing technique for such data is needed.

While many alternate index structures have been proposed in recent years for spatial data specifically [Sam89] and for multi-dimensional data more generally, R-trees remain a popular index structure offered by numerous commercial DBMS systems. R-trees, which are an extension of B-trees, are designed to store multi-dimensional data [Gut84]. The R-tree structure was initially proposed by Guttman [Gut84] and various variations and improvements over the original structure have been suggested and implemented ever since [NHRB90] [CT97] [YT94].

Note that like with most other index structures, the data needs to be first 'loaded' into the R-tree structure in order to set up the index before the index structure can be exploited 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 loading as the OBO (one-by-one) technique. Over the last few years, more optimal insertion and node splitting techniques were provided by Theodoridis [YT94] and Ang [CT97]. These new insertion techniques somewhat improved the R-tree loading times.

The insertion of data sequentially into the R-tree may be suitable for situations when we are entering a few data items into the R-tree. It has however been found to be inefficient in the case when the entire tree needs to be set up [YT94]. Thus, as an improvement over this sequential OBO insert strategy, various *bulk loading* strategies have been proposed in recent years [SMY97] [JPB97] [SMJ97] [IC93] [ND85]. These bulk loading strategies were aimed at loading data into an initially empty R-tree and thus creating the complete R-tree *from scratch*. These techniques thus assumed that totally new data was being collected (or set up to be utilized by some application), and thus an index structure had to be constructed from scratch for this complete set of new data.

To the best of our knowledge, no work has been done on *bulk insertions* of new data assuming an already existing R-tree. The importance of this problem for numerous real-world examples is however apparent. For example, new data is obtained by satellites almost daily and some or most of this should ideally be loaded into existing index structures in order to serve applications that must query or operate over both the old as well as new data. The loading of this new data should not only be efficient, but it should also minimize down-time of the structure. In other words, 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 now also make use of the new data as quickly as possible. The construction of a new index structure from scratch to contain both the old as well as the new data, as for example the pack-and-merge approach [NMY97], 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 should be developed.

We furthermore note that the application examples mentioned above typically involve inserting skewed or localized datasets into already existing index structures. A recent transmission from a probe may return data scans from a particular region or planet of the galaxy while the overall spatial

database system may contain data from many different planets and stars from the galaxy. The problem becomes even more important when one realizes that real data are more often skewed than balanced as observed for example in [Ooi90] [CI94].

Driven by these real-world requirements, this paper looks into the problem of bulk loading of new possibly *skewed* datasets into an existing *active* R-tree. By active, we mean an R-tree which already contains (large) datasets, may have already been used for some amount of time, and which has currently active applications which forbid the possibility of scheduling a long downtime for the index structure construction process. By skewed, we mean that the dataset to be inserted is localized to some portion of the region covered by the R-tree and is not necessarily spread out over the complete region covered by the existing index structure.

In this paper, we propose a new technique, which as opposed to the current technique of inserting data one by one, bulk inserts entire new datasets. This technique, called STLT (for Small-Tree-Large-Tree), considers the new dataset as an R-tree itself (small tree), identifies and prepares a suitable location in the original R-tree (large tree) for insertion, and lastly performs the insert of the small tree into the large tree.

In this paper, extensive experimental studies are reported that not only compare STLT against the conventional technique, but also evaluate the suitability and limitations of STLT under different conditions, such as varying buffer sizes, ratio between existing and new data sizes, and skewness of new data concerning the whole spatial region. Experiments are conducted both with synthesized datasets as well as with real world datasets from the Sequoia 2000 storage benchmark. In our experiments, we find that STLT does especially well (in some cases even up to 60% better than the existing technique) for skewed datasets as well 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 STLT 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 STLT in terms of query performance is acceptable and in some cases even preferable over that created by the traditional tree insertion approach.

In summary, the contributions of this work are four-fold:

1. We identify the problem of *bulk insertions* of spatial data into *existing* index structures - in contrast to recent work on bulk loading that has primarily focused on loading data from scratch.
2. We design a technique, called STLT, specially tuned to address this bulk insertion problem.
3. We implement STLT and the conventional insertion technique in an integrated environment in order to establish a uniform testbed for evaluating and comparing them.
4. We conduct extensive experimental studies both using real-world as well as synthetic datasets. These experiments clearly demonstrate that STLT is an absolute winner, both in terms of insertion as well as retrieval performance.

The remainder of the paper is organized as follows. Section 2 deals with work related to this paper. Section 3 defines the bulk insertion problem and discusses our solution approach. Performance results for query tree loading and query-handling are presented in Section 4 which is followed by conclusions in Section 5.

2 Related Work

To the best of our knowledge, there is no prior work on bulk insertions into an already existing R-tree. However, 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 involve a bottom-up construction of the R-tree. Kamel and Faloutsos [IC93] use the Hilbert sorting technique to first order data and then build the R-tree. Leutenegger et al [SMJ97] 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. The process is repeated recursively for each of the k-dimensions. Both techniques are concerned with bulk loading of data and not bulk insertion.

The other class of algorithms focus on a top-down approach to building the R-tree. Bercken et al [JPB97] adopt a generic top-down approach of loading data into an index structure (R-tree) with the use of a memory-based temporary structure called the buffer-tree. This technique would not be very applicable when inserting new data as we tackle in this paper — unless the temporary structure was continuously to be maintained which would be expensive, as buffer pages would be wasted. Leutenegger in [SMY97] proposed another top-down technique called the TGS (Top-Down Greedy Split), for bulk loading data. The technique tries to partition data in such a manner that it minimizes a user-supplied function. The algorithm produces an optimal tree but requires large loading times.

[NMY97] talk about 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 packed into a new R-tree. The strategy has the expensive sort and the merge steps. Also, it does not exploit the skewed nature of the incoming data which our work does.

[KET96] deals with issues related to concurrency control in B-trees when batch updates are being done. The algorithm which they provide works by sorting the incoming data and inserting chunks of it into appropriate node. The technique is for B-tree index structures.

[Moi93] talks about various issues related to R-trees. The latter portion of the paper deals with batching of data prior to insertion. The paper proposes a rudimentary batching of data and sorting it prior to insertion to improve the performance. The algorithm is rudimentary and due to the expensive sorting phase is not expected to scale well.

Bulk updates and bulk loading have been studied for various other structures and in various other scenarios [Che97] [JDJ93] [LN97] [CP98]. These techniques are specific to the structure in question and are not directly applicable to our problem.

3 STLT: A Bulk Insertion Strategy

3.1 Background on R-trees

An R-tree is a height-balanced tree-structure consisting of *index* nodes and *leaf* nodes. The leaf nodes contain the minimum boundary rectangles (MBRs) of the data objects and pointers to the complex,

spatial objects themselves. The index nodes contain the MBRs enclosing the MBRs of all its children nodes. This relationship of passing up the enclosing MBR to their respective parent is continued upwards until the root. The root itself then holds the MBR that encloses the complete global region of the area modeled by the spatial application data.

3.2 Problem Definition

The aim of any loading algorithm should be to maintain an efficient structure of the R-tree so as to keep the cost of answering the queries as low as possible. The R-tree shows good search performance if the search query causes traversals of only a few paths. This is more likely to occur if the sibling MBRs at each of the levels have less overlap between them. Also, as indicated earlier, it is important to load new data into the R-tree as fast as possible. Thus, the goals of our proposed technique are both to minimize the loading time and to produce an R-tree with a good search performance.

Intuitively, the loading time is likely to be minimized if sets of data that belong together into one localized region of the R-tree could be loaded into the R-tree at once instead of exercising the insertion process over and over for each of the individual objects. This observation, which represents the underlying principle of our proposed technique, now raises several issues that must be addressed.

- First, what is an effective strategy for inserting sets of data objects all at once into an existing R-tree while minimally impacting active applications of the tree?
- 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 both insertion time as well as resulting tree quality)?
- Third, once desirable characteristics for the objects in each insertion set have been determined, what are suitable strategies for grouping incoming possibly continuous streams of spatial objects into insertion sets so that each of them possesses these characteristics?

For the first issue, we propose in this paper a new bulk-insertion strategy called 'Small Tree Large Tree' (STLT) algorithm. STLT not only demonstrates efficient insert 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 in this paper 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. Characteristics of particular interest are of course (1) the number of objects in one insertion set (size of insertion set versus size of existing tree) as well as the spatial distribution of the new objects in terms of the overall spatial region covered by the R-tree (skewness).

For the third issue, one possible solution would be to simply chop input streams into equally sized sets grouped in the order in which they arrive for insertion or alternatively, use more sophisticated clustering techniques that also take data distributions into account.

In the remainder of this paper, we focus on the first two issues, while an investigation of the third issue is beyond the scope of this paper and instead is left as future work.

3.3 The STLT Approach

We now propose the following approach to solving the problem of inserting (skewed) data into an existing R-tree. Since the data to be inserted is assumed to be localized to some portion of the R-tree rather than being distributed over the whole R-tree, the new data is first inserted into a new empty R-tree. We call this the *small tree*. This new tree can be built using any of the existing bulk loading techniques. In our case, we use the R-tree technique proposed by Guttman [Gut84]. No difference in loading time is expected even if newer, improved techniques are used since all techniques have to traverse from top to bottom to search for the appropriate leaf node to insert the data. However, as indicated earlier, alternate bulk insertion strategies may result in even slightly better performance for this step of our algorithm.

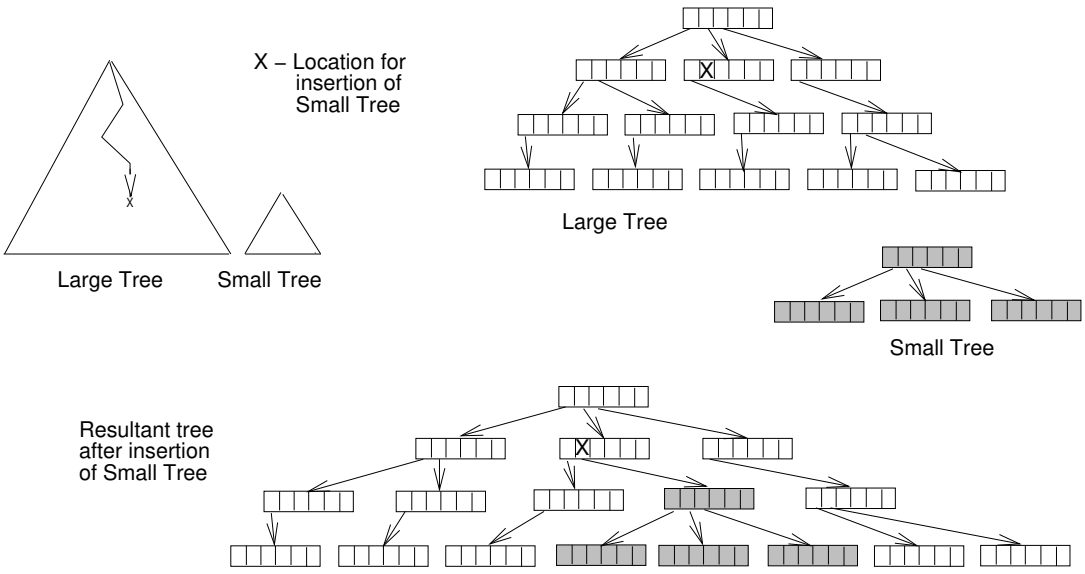

Figure 1: STLT Algorithm

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 1.

The main steps of our Small Tree Large Tree (STLT) algorithm are the following :

1. Build an R-tree (small tree) from scratch from the new dataset.
2. Find a suitable position in the original tree (big tree) for insertion of the small tree.
3. Handle unavailability of entry slot space in large tree using different heuristic techniques.
4. Insert the small tree into the identified location (or created as a result of Step 3).

Since the small tree is assumed to be relatively more skewed, clustered and packed as compared to the large tree, we do not want to decompose it into subtrees and fit them one at a time into the original tree. We instead propose various heuristics for handling overflow to allow for insertion of the whole tree all at once, as described below. These heuristics attempt some local reorganization of the large tree in order to assure that the small tree can be inserted in one operation. This reorganization cost is minimal (in practical settings, a small constant). And, it is justified especially if we are dealing with large datasets, as it allows insertion in one single traversal over the large tree. A more detailed explanation of the four steps of the STLT algorithm now follows.

3.4 The STLT Terminology and Assumptions

For the following, we assume the terminology and notation below :

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

```
ALGORITHM  STLT  ( LT_Root, NewData_file )
{
   ST_Root = BuildSmallTree( NewData_file );
                                    // build an R-tree using any
                                    // technique and initialize ST_Root
   DiffHeight = LTHeight - STHeight; // level for insertion into large tree
   if (DiffHeight < 1) {            // large tree is not larger than small tree
      return;                       // STLT not suitable for bulk insertion
   }
   else {                           // insert small tree into large tree
      InsertSTintoLT( LT_Root, ST_Root, DiffHeight );
   }
}                                   // end Procedure
```

Figure 2: The *STLT* Algorithm : Bulk Insertion of Small Tree into Large Tree.

The STLT algorithm given in Figure 2 first builds the small tree (denoted by ST) from scratch, as indicated by the function *BuildSmallTree()* invoked in the algorithm. Next, 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. A simple calculation based on the fanout of the nodes, before the small tree is constructed, could catch this and would then revert to OBO insertion. 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 *InsertSTintoLT()* procedure given in Figure 3 first checks whether *DiffHeight* is equal to one. If it is not, we have not yet approached the correct level for insertion of ST. It thus invokes the *ChooseSubtree()* procedure recursively to locate the appropriate index node at the next level of the R-tree for possible insertion of *ST_Root* until *DiffHeight* becomes one. Any of the standard search procedures for R-tree insertions would be suitable for the *ChooseSubtree()* procedure. Hence, this is not further discussed here. Once *DiffHeight* is equal to one, we invoke the *InsertSTMBR()* procedure.

```
PROCEDURE InsertSTintoLT( IndexNode, ST_Root, DiffHeight )
{
   if (DiffHeight > 1) {
      DiffHeight--;             // continue search another level down
      IndexNode = ChooseSubtree( LT, IndexNode, ST_Root );
                                // choose node at next level
      InsertSTintoLT( IndexNode, ST_Root, DiffHeight );
                                // recursive call until ST falls to suitable level
   }                            // end if
   else {                       // IndexNode is desired location for insertion
      InsertSTMBR( ST_Root, IndexNode );
   }                            // end else
   ResizeMBR( IndexNode );      // resize MBR of nodes involved for ST insertion
}                               // end Procedure
```

Figure 3: Identification of Ideal Insertion Location of Small Tree into Large Tree.

Next, as listed in Figure 4, if the large tree node identified for insertion of the small tree root (*IndexNode*) has room (i.e., has less than M entries), then we accommodate *ST_Root* in *IndexNode*. However, if *IndexNode* is full (i.e., has all M entries), we now invoke steps to handle this overflow situation.

```
PROCEDURE InsertSTMBR( ST_Root, IndexNode )
{
   if ( NumChildren( IndexNode ) < FANOUT ) { // node not full (ideal case)
      IndexNode.child[numChildren++] = ST_Root;
                                              // insert ST into indexNode
   }                                          // end if
   else {                                     // node full, handle overflow
      OverflowTreatment( ST_Root, IndexNode );
   }                                          // end else
}                                             // end Procedure
```

Figure 4: Insertion of Small Tree into Located Index Node of Large Tree.

To handle overflow, we have designed various heuristics as outlined in the *OverflowTreatment()* procedure listed in Figure 5. First, we attempt to merge siblings of the *IndexNode* in order to leave an entry slot for the small tree root. The merge is carried out by merging two closest nodes (based on their centroid distance), whose combined number of children is less than the fanout of one node. If the merge of siblings should fail, then we delete one of the sibling nodes and reinsert its data thus making room for the small tree. We choose the sibling node as candidate for deletion by considering

the distance between each child node from the center of its enclosing rectangle, and select the one that has the farthest. If all the above attempts fail, then we must resolve to the conventional technique of splitting the *IndexNode* to create a new entry slot for insertion. We then propagate multiple splits upwards as done in regular R-tree algorithm [Gut84] until either sufficient space is available (which is likely), or the large tree itself increases its height by one.

```
PROCEDURE OverflowTreatment( ST_Root, IndexNode )
{
   ( S1, S2 ) = SelectSibling( IndexNode ); // select siblings to merge
   if ( not MergeSibling( S1, S2 ) ) {      // if merge failed, try ReInsert
      Child_del = SelectFurthestChild( IndexNode );
                                            // delete farthest child and
                                            // reinsert it into LT
      if ( not ReInsert( Child_Del ) ) {
         SplitNodeQuadratic( ST_Root, IndexNode );   // else Split
      }
   }                                        // end if
}                                           // end Procedure
```

Figure 5: The Overflow Treatment Procedure.

4 Experimental Results

4.1 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, and many other supporting data structures and procedures.

Test Data : The test data comprised both real datasets and synthetic datasets. The synthetic data was generated with varying parameters to control the distribution and the nature of the objects. The real data 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. The data is derived from the TIGER/Line files distributed by the US Census Bureau.

Test Types : We carried out two major classes of tests. The first set of experiments were conducted to compare the *I/O insertion costs* of STLT with OBO for different parameters as described in Sections 4.2 to 4.6. The tests were designed to evaluate the performance of STLT over OBO in terms of I/O costs for different parameters and determine the usefulness and limitations of STLT. The second set of experiments focussed on evaluating the *quality* of the resultant trees formed by STLT 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 (Sections 4.7 and 4.9).

5.1.4 Experimental Parameters

The main parameters used in our experiments are listed below :

- *Buffer Size :* Size of the available buffer in terms of number of pages.
- *Data Sizes :* Size of large tree data and small tree data.
- *Ratio of Data Sizes :* Ratio of number of objects in the large tree as compared to the number of objects in the new dataset.
- *Ratio of Areas :* Ratio of large tree region to small tree region.
- *Location of New Dataset :* Location of the new datasets within region of the existing dataset.

The default parameter settings are listed in Table 1. For each test, we change one parameter while keeping the default values for the others fixed.

Buffer Size (# pages)	50
Large Tree Size	50000
Ratio of Large to New Data Size	50
Ratio of Large Tree Area to New Data Area	225
Distribution of New Data	Center

Table 1: Default Parameters Settings

To simulate the scenarios where the new dataset is in a region from which little or no data has previously been gathered, we introduce the concept of a 'hole'. A 'hole' is a region of the large tree which is of low (close to zero) data density (fewer MBRs) as compared to the remaining region of the large tree.

4.2 Experiments on Buffer Sizes

In this experiment, we vary the buffer size from 25 to 500 pages (with a fixed page size = 4K). We measure the page I/O costs for a given ratio of the number of objects in the large tree to the number of objects in the new dataset. As shown in Figure 6, we tested two cases, namely, a ratio of 25 and of 50.

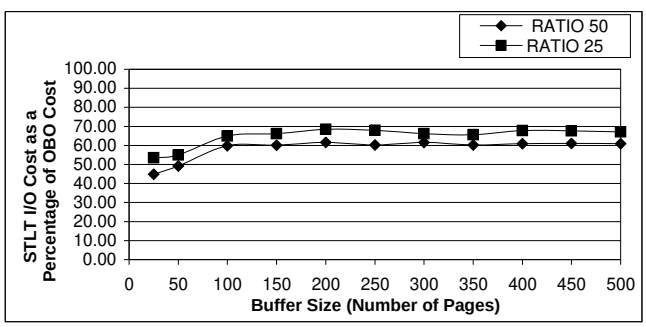

Figure 6: Effect of Different Buffer Sizes.

In Figure 6, the I/O costs for insertion by using OBO are represented by the straight line at the 100 percent cost position. The other two curves represent the STLT insertion cost as a percentage of OBO costs for the above mentioned data size ratios.

First, we observe that STLT is especially better than OBO when the buffer size is small (for example STLT has up to a 55% improvement for a buffer of 25 * 4K = 100K pages). Although STLT wins over OBO out in all the cases, its advantage over OBO is not as large for larger buffer sizes. Namely, for this fixed parameter setting, it levels out at a 40% improvement. From this test we conclude that we can save up to 55% of the I/O costs by using STLT in cases where the available buffer size is small (in this case, 400K) and upto 40% for larger buffer sizes, for the mentioned ratios.

In this experiment, we also compared different ratios of the number of objects in the large versus the small tree. The higher curve represents that the case where the data size of the large tree is 25 times that of the new data size, while for the one below the ratio equals 50. This result thus indicates that the larger the ratio, the better STLT performs.

4.3 Experiments on Data Size Ratio

In this experiment, we measure I/O costs for STLT and OBO for different size of the small tree expressed as a percentage of the large tree.

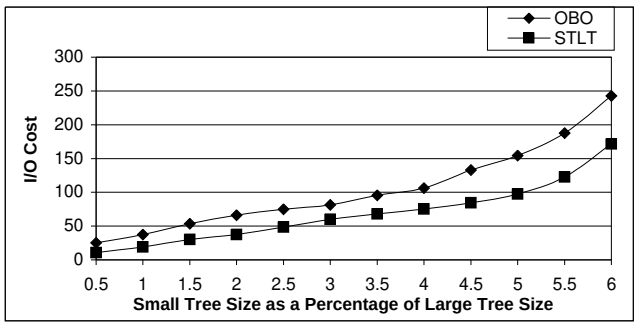

Figure 7: Relationship between STLT Cost Percentage of OBO versus Small Tree/Large Tree Size.

STLT reaches over 65% of improvement in I/O costs over the OBO when the small tree is kept sufficiently small (i.e., 0.5% size) as compared to the large tree size as seen from Figure 7. STLT consistently performs better than OBO for different sizes of the small tree. In terms of only the actual insertion costs, STLT does better by a large factor, but since in STLT the small tree is first built, some additional I/O costs are incurred which offsets STLT's advantage over OBO. The experiment clearly shows that STLT performs well even for large data sizes which is a very important result.

This experiment helps us to gain an understanding of how the ratio of data sizes affects the cost comparison. The experiment shows us that STLT performs better than OBO for small as well as large sizes of small tree.

4.4 Experiments on 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.

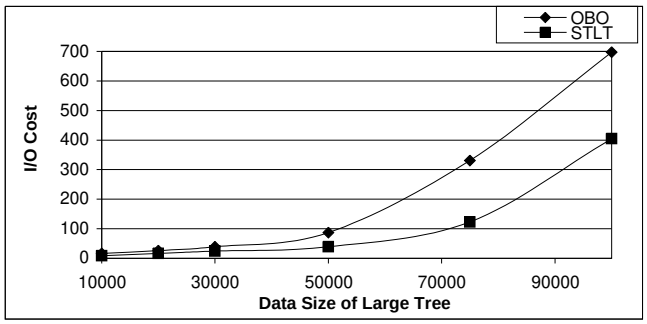

Figure 8: Relationship between I/O Insertion Cost and Data Size.

The results as seen in Figure 8 indicate that the STLT outperforms OBO all the way in absolute I/O costs. And, in fact, STLT continues to improve further over OBO with an increase in the large tree size. For a large tree size of 100,000 nodes, the OBO is over 60% worse than STLT. This is a very promising result indicating superb performance for larger existing index structures. The trend indicates that costs will be further in favor of STLT as the tree grows.

4.5 Experiments on Data Area Ratios

STLT is designed based on the assumption that the new dataset is localized and skewed as compared to the relatively evenly distributed large tree region. We now test the limitations of STLT to determine if and to what degree its performance would deteriorate, once these assumptions are dropped. For this test, we vary the ratio of the area of the new dataset to the large tree area from 0.06 percent to 4 percent, keeping the region of the large tree fixed. We keep the number of objects in the large tree to be always 50 times that of the new dataset.

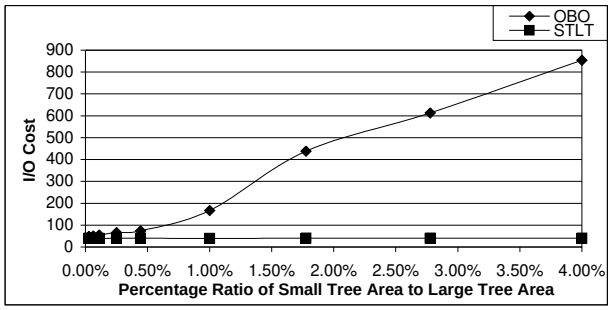

Figure 9: I/O Insertion Cost versus Area Ratio of Small Tree to Large Tree.

Our experimental results depicted in Figure 9 indicate that the cost for insertion using STLT is not affected by the percentage of the region covered by the newly inserted data. In fact, STLT costs

remain consant as seen from the lower straight line. For OBO, the I/O costs increase when the region covered by the new data is enlarged. This is because the individual objects of the new datasets might be distributed over many different paths due to their reduced skewed properties. Note that when the area of the new dataset reaches 4% of the global region, OBO already has a 2000% larger insertion cost than STLT. Considering that there could be many insertions, this could be a significant factor. While this indicates the superiority of STLT under any circumstances in terms of insertion time, we still consider STLT to be mainly applicable to the skewed dataset insertions to assure quality of the resultant tree in terms of query performance, as further evaluated in Sections 4.7 to 4.9.

4.6 Experiments on Real Datasets

We have taken real datasets derived from the TIGER/line files distributed by census bureau. We make the sizes comparable to the synthetic dataset size and obtain a series of new dataset by chopping the real dataset within the same region into chunks of different ratios. We keep other parameters the same as for the synthetic dataset test. In this experiment the large tree size and the small tree size are varied (the ratio between them is kept constant at 50) from a large tree size of 10000 to a size 100000.

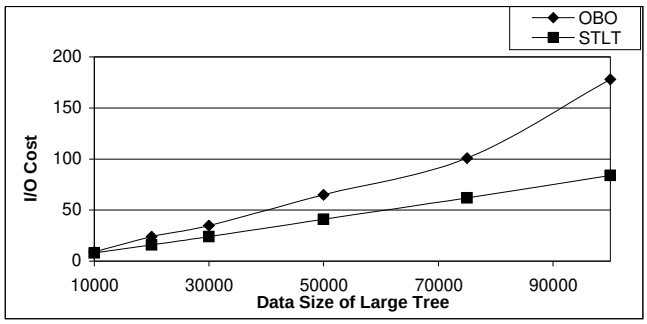

Figure 10: Relationship between I/O Cost and Data Size.

The result in Figure 10 shows the same trend as with the synthetic data, although the absolute costs are lower, and gives even slightly better percentage improvement of STLT over OBO. As the sizes of the small and large tree are increased, the difference between OBO and STLT goes up, which is similar to the trend seen in figure 8. The similarity in trend and percentage improvements show that our results with synthetic data can be considered to be indeed good predictors for real data cases.

4.7 Experiments on Tree Quality using Random Queries

In this query testing scenario, the data for one large tree and a series of small trees were generated using our synthetic data generator. Both STLT and OBO were used to insert the sequences of small trees into the same initial R-tree, thus leading to two trees with the same augmented data content. A large number of random queries (50400 queries) are then generated. The query rectangle was 10 times larger than the maximum object size in the resulting tree, and the query rectangle was placed at a random location over the large tree region.

In this experiment (Figure 11), we evaluate the quality of the R-tree constructed by either STLT or OBO by executing the same set of random queries with the query rectangle placed at a random

location over the large tree region. We measure the average number of nodes visited in order to answer the queries.

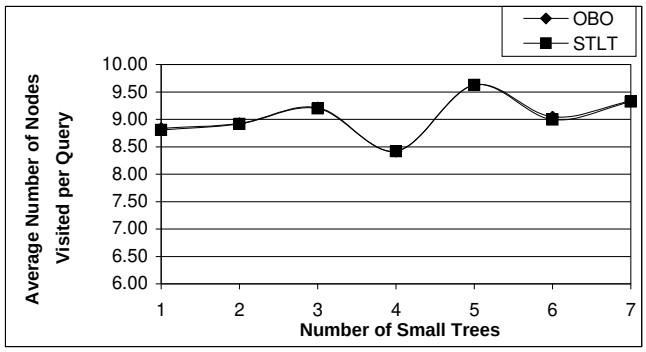

Figure 11: Number of Nodes Visited for Random Queries.

The chart shows that STLT and OBO have almost identical costs in order to answer the queries. This suggests that the quality of the tree does not deteriorate when STLT is used as compared to when OBO is used. Thus, this experiment confirms that STLT is a very viable solution as it not only offers high insertion performance, but also maintains the quality of the tree.

The query performance remains the same for STLT as compared to OBO even if a large number of small trees are inserted. Clearly, we conclude that STLT leads to trees which yield acceptable and comparable query performance for the scenarios tested in this experiment.

4.8 Experiments on Query Performance using Localized Queries

In this experiment, we model applications that may be primarily interested in the newly inserted data. Thus, we use queries with the query rectangle placed at a random location over the area where the small tree was inserted. Since the small tree by itself is likely to be of good quality (for skewed data), we expect that queries localized on that area may yield better performance for STLT as compared to OBO. We again measure the cumulative number of nodes visited in order to answer the queries.

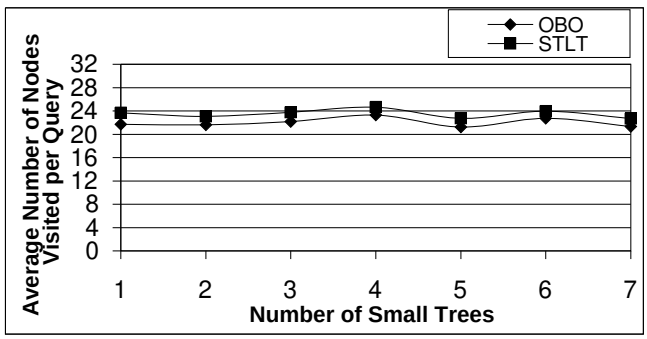

Figure 12: Number of Nodes Visited for Random Localized Queries.

As seen from Figure 12, we find that the STLT indeed does better in terms of nodes visited as compared to when running random queries - which confirms our expectation. This suggests that

STLT query performance would be definitely comparable to OBO, if not better when such localized queries are asked. The performance results indicate the strength of STLT in answering queries localized to newly inserted data, which corresponds to the scenario for which STLT was designed to begin with.

4.9 Experiments on Query Performance using Queries with Increasing Area of the Small Tree Region

In this experiment, we try to determine what area sizes are acceptable for STLT. Thus we ask random queries changing the size of the small tree area from a small percentage of the large tree (our proposed scenario) to the maximum size (covering the large tree area). We expect the performance of STLT to decrease as for larger areas, the data rectangles are being forced into a node where it does not belong.

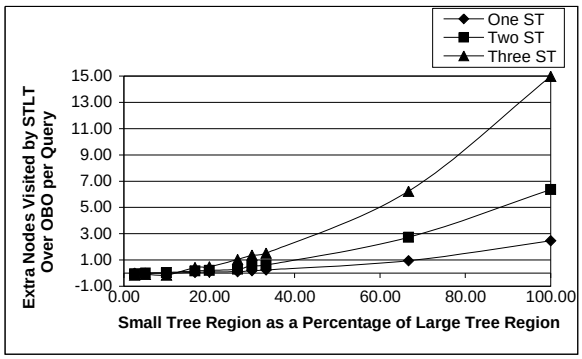

Figure 13: Number of Nodes Visited for Different Small Tree Area.

As seen from the Figure 13 we see that STLT performance deteriorates as the area of the small tree is increased. It is important to note that STLT performs as well as OBO for upto 40% small tree area size which indicates that STLT is useful for a much wider range of incoming data and is not restricted to a small skewed data insertion scenario.

5 Conclusions

In contrast to previous work on bulk loading data which focussed on building index structures from scratch, in this paper we now tackle the problem of bulk insertions into existing index structures. We have placed particular focus on R-trees – as those are an important class of index structures used widely in commercial database systems. 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 technique, called STLT (for Small-Tree-Large-Tree), considers the new dataset as an R-tree itself (small tree), identifies and prepares a suitable location in the original R-tree (large tree) for insertion, and lastly bulk-inserts this small tree into the large tree. We report experimental studies designed to not only compare STLT against the conventional technique, but also to evaluate the suitability and limitations of STLT under different conditions. In this paper, we also report on

some of the experiments we have conducted using real world datasets from the Sequoia 2000 storage benchmark, which further confirm our results.

We have found that STLT does especially well (over 60% better than the existing technique) for skewed datasets as well for large ratios of large tree sizes as compared to sizes of the new data insertions, while consistently outperforming the alternate technique in all other circumstances. In fact, STLT even reaches a 2000% improvement over OBO for less skewed insertion sets - though this is not the intended target of our techniques. Our experimental results also indicate that the quality of the resulting tree constructed by STLT in terms of query performance is acceptable and in most cases even preferable over that created by the traditional tree insertion approach. All in all, the STLT *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 resulting index access quality.

Possible future tasks include application of the general approach of STLT to other multi-dimensional index structures as well as techniques for on-the-fly clustering of data streamed to be inserted into an index structure may also be fruitful, ideally leading to optimal groupings of data into several smaller chunks (in our terminology, small trees) for bulk-insertion.

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 (Woody) 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

[Che97] Weidong Chen. Programming with logical queries, bulk updates, and hypothetical reasoning. *IEEE Transactions on Knowledge and Data Engineering*, pages 587–599, July 1997.

[CI94] C.Faloutsos and I.Kamel. Beyond uniformity and independance: Analysis of R-tree using the concept of fractal dimension. *Proceedings of SIGMOD*, pages 4–13, 1994.

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

[CT97] C.H.Ang and T.C.Tan. New linear node splitting algorithm for R-trees. *Advances in Spatial Databases*, pages 339–349, 1997.

[Gut84] A Guttman. R-trees : A dynamic index structure for spatial searching. *Proceedings of SIGMOD*, pages 47–57, 1984.

[IC93] I.Kamel and C.Faloutsos. On packing R-trees. *Proceedings of International Conference on Information and Knowledge Management*, pages 490–499, November 1993.

[JC88] J.Srivastava and C.V.Ramamoorthy. Efficient algorithms for maintenance of large database indexes. *Proceedings of IEEE International Conference on Data Engineering*, pages 402–409, 1988.

[JDJ93] J.Li, D.Rotem, and J.Srivastave. Algorithms for loading parallel gridfiles. *Proceedings of SIGMOD*, pages 347–356, 1993.

[JPB97] J.Bercken, P.Widmayer, and B.Seeger. A generic approach to bulk loading multidimensional index structures. *International Conference on Very Large Data Bases*, 1997.

[KET96] K.Pollari-Malmi, E.Soininen, and T.Ylonen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, pages 975–983, 1996.

[LN97] Scott Leutenegger and David Nicol. Efficient bulk-loading of gridfiles. *IEEE Transactions on Knowledge and Data Engineering*, pages 410–420, May 1997.

[Moi93] A Moitra. Spatio-temporal data management using R-trees. *International Journal of Geographic Information Systems*, 1993.

[ND85] N.Roussopoulos and D.Leifker. Direct spatial search on pictorial databases using packed R-trees. *Proceedings of SIGMOD*, pages 17–32, 1985.

[NHRB90] 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.

[NMY97] 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.

[Ooi90] B Ooi. Efficient query processing in geographic information systems. *Lecture Notes in Computer Science*, 1990.

[Sam89] H Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[SJJ86] S.D.Lang, J.R.Driscoll, and J.H.Jou. Batch insertion for tree structured file organizations - improving differential database representation. *Information Systems, vol. 11, no. 2*, pages 167–175, 1986.

[SMJ97] 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.

[SMY97] 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.

[YT94] Y.Theodoridis and T.Sellis. Optimization issues in R-tree construction (extended abstract). *Lecture Notes in Computer Science*, pages 270–273, 1994.