VLDB SUBMISSION #268

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. In this paper, we present an analytical model of STLT. Extensive experimental studies both on synthetic and real data sets from the Sequoia 2000 storage benchmark are also reported. They 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 concerning the whole spatial region. We find that STLT does especially well (up to 80% 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 quality of the resulting tree constructed by STLT in terms of query performance is acceptable and in many cases even preferable over that created by the traditional tree insertion approach.

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 really large, but also the data being acquired constantly is of ever increasing magnitude. For example, the amount of data generated by satellites in said to be several gigabytes per minute. Hence efficient storage and indexing technique for such data is needed. The index structure should have two important properties. One, it should be a dynamic structure supporting fast insertions and deletions of data in order to allow for the structure to be quickly updatable — without hindering existing applications that are concurrently using the index structure for their query purposes. Two, it should provide high- performance access in order to effectively support query processing and other activities over the spatial data.

While many alternate index structures have been proposed in recent years for spatial data specifically 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 [A.G84]. The R-tree structure was initially proposed by Guttman [A.G84] 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 [A.G84] was used to load sets of data into an R-tree, i.e., each object was inserted one by one into the index. Hence, in this paper, we will refer to this traditional technique of data loading as the OBO (one-by-one) technique. The original structure provided by Guttman [A.G84] did not use the best strategies for insertion and more optimal insertion and node splitting techniques were provided subsequently 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 extremely 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 completely new data was being collected (or set up to be utilized by some application), and thus an index had to be constructed 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 operation while being minimally impacted by the insertion of new data and in addition while been given as quickly as possible the opportunity to now also make use of the new data. The construction of a new index structure from scratch that now contains both the old as well as the new data, as for example proposed in [NMY97] as the pack-and-merge approach, 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 [B.O90] [CI94].

Driven by these real-world requirements, the problem that this paper looks into is that of bulk loading of new possibly skewed data 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 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. While we set out to design a technique specifically targeted to these characteristics, in this paper we also evaluate the performance of the proposed approach in terms of boundary conditions, for example the new data being evenly distributed over the complete spatial region.

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.

We also present an analytical model we have developed for evaluating STLT and contrasting it to OBO. Extensive experimental studies are also 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, skewness of new data concerning the whole spatial region, and data density. Experiments are conducted both with synthesized datasets to test the boundary conditions of the applicability of our new technique as well as with real world datasets from the Sequoia 2000 storage benchmark. The results for both are comparable, thus indicating the appropriateness of the synthesized data for our tests. In our experiments, we find that STLT does especially well (in some cases even up to 300% 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. The technique does especially well in terms of query performance for queries which are asked only over the newly inserted data. Such queries would be common for systems where new data is brought in frequently and queries are asked over the new data. One example of such a system may be incremental data mining algorithms which query primarily the 'new' data, while still requiring some access to the original data. Another example may be GIS systems where users are particularly interested in exploring and learning about the newly acquired spatial data.

In summary, the contributions of this work are five-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 establish an analytical model, which while being preliminary is still able to shed some light on the conditions under which STLT is likely to perform well (and not so well).
4. We implement STLT and the conventional insertion technique in an integrated environment in order to establish a uniform testbed for evaluating and comparing them.
5. We conduct extensive experimental studies both using real-world as well as synthetic datasets. These experiments allow us to not only compare STLT against the conventional OBO technique but also to identify the strengths and limitations of STLT itself.

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. Section 4 presents the analytical model we have developed for evaluating our algorithm. Performance results for query tree loading and query-handling are presented in Section 5, followed by conclusions in Section 6.

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.

Kamel and Faloutsos [IC93] use the Hilbert sorting technique to first order data and then build the R-tree bottom-up. In this technique, the data is sorted on the hilbert-value of some point and the sorted data is then used to construct the tree recursively in a bottom-up manner. This technique proves to be better in terms of query performance than repeated OBO technique . However, it assumes a static database and thus focuses on doing all insertions all at once. Furthermore, it would not apply to the problem of inserting data into an existing tree if that tree were not initially built using the Hilbert technique.

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 approach avoids any sorting of the data and instead works by using the buffer-tree, which differs from the target index structure (R-tree) in two ways. First, its internal nodes have additional buffers where records can be stored. Second, multiple insertions are processed simultaneously. Using the buffer-tree, each level of the R-tree is built. The amortized insertion cost of the buffer-tree is lower than the insertion cost of the original index structure. This reduction in the I/O cost is mainly due to the use of a temporary structure. It thus 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.

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. The technique is shown to do better than the Hilbert sorting technique for uniform and mildly skewed data. Again, the target of this work was bulk-loading and not bulk-insertions.

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. The algorithm performs better than STR and Hilbert sort for skewed data and does considerably better for point queries than range queries. A major drawback of the algorithm is that it requires up to 2.6 times more loading time than the other competing approaches and is best suited for cases where the bulk loading does not need to be performed frequently.

While several experimental studies on R-tree performance have appeared in the literature, analytical models that account for realistic assumptions — in particular that incorporate buffer behavior — are rare. Leutenegger et al [SM96] is one work that studied the effect of buffering on the performance of R-trees. The paper dealt with a buffer model which analyzed the number of disk accesses required for spatial queries using R-trees. The model was used to evaluate the quality of the R-tree in terms of the query performance. In [HJR97], Huang et al. present the first analytical model for R-tree based joins, again taking buffer effects into account. In this current paper, we now made use of these previously introduced techniques when developing our analytical model to compare STLT against OBO in terms of insertion performance.

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 objects themselves. The index nodes contain the MBRs of all its child nodes. Thus, if an *index* node has five children nodes, it would store the MBR of its five children nodes. This relationship of passing up the enclosing MBR to their respective parent is continued upwards until the root, allowing us during query retrieval to make the right choices of which paths to search down on. The root itself then holds the MBR that encloses the complete global region of the area modeled by the spatial application data.

A query on an R-tree is processed by starting from the root and recursively traversing downwards through paths which are likely to contain spatial objects relevant to the query as indicated by the MBR information associated with the respective path. In an R-tree, answering a query might involve accessing multiple paths down the tree as the R-tree allows intersecting MBRs, i.e., in the worst case the complete R-tree may have to traversed. However, the R-tree has been found in practice to have extremely well behavior for real-world data, generally never reaching this worst case.

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 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 chunks 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 chunks of data all at once into an existing R-tree while minimally impacting active applications of the tree?
- Second, which characteristics should these chunks of to-be-inserted data objects 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 those most desirable 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 application by (1) requiring no down time of the existing index by avoiding to build a new R-tree from scratch, (2) 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 (skewedness); as our STLT is designed to apply to insertion of relatively skewed sets of objects into a possibly large existing tree - for which re-construction would simply be too costly.

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 from the data source. In this case, insertion sets of the optimal size could be easily constructed without any overhead. Alternatively, more sophisticated clustering techniques could be employed that also take data distributions into account, however, wold require some additional overhead to perform the clustering.

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 Our Approach - The STLT Algorithm

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 quadratic R-tree technique proposed by Guttman [A.G84] due to the availability of this source code. However, as indicated earlier, alternate bulk insertion strategies may result in even slightly better performance for this step of our algorithm.

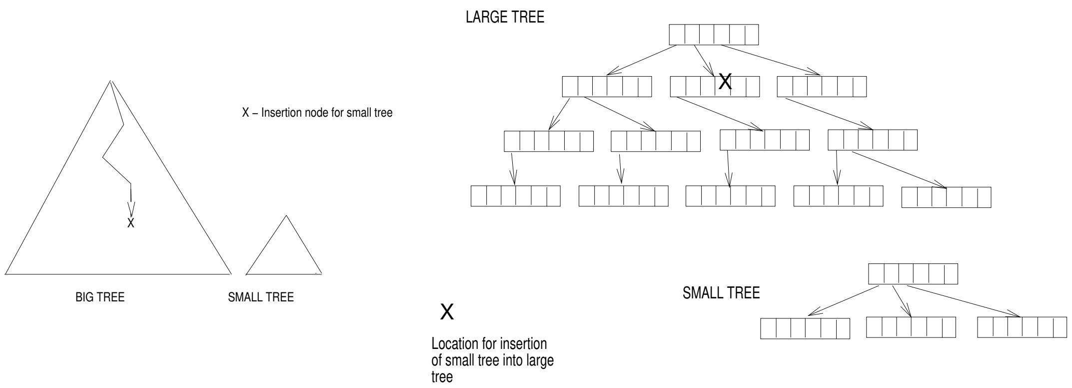

Figure 1: STLT Algorithm

Let the height of the newly built small R-tree be h_r and the height of the original R-tree be H_R. Next 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 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 involved in our Small Tree Large Tree (STLT) algorithm are summarized as follows :

1. Build an R-tree (small tree) from scratch from the new skewed 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 proposed 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.

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

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

```
ALGORITHM  STLT  ( LT, NewData_file )
{
   ST = BuildSmallTree( NewData_file );  // using any R-tree technique
   ST_Root = SetEnclose( ST );           // calculate root MBR for small tree
   DiffHeight = LTHeight - STHeight;     // level for insertion into large tree
   if (DiffHeight < 1)                   // large tree is not larger than small tree
      then return;                       // thus STLT not suitable for bulk insertion
   else
      InsertSTintoLT( LT_Root_Node, ST_Root, DiffHeight );
   endif;                                // insert small tree into large tree
}
```

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

The STLT algorithm first builds the small tree (denoted by ST) from scratch, as indicated by the function *BuildSmallTree()* invoked in the algorithm given in Figure 2. 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, and we instead simply applying one of the bulk-loading strategy to build a new tree containing both old and new data from scratch. Otherwise, we invoke the *InsertSTintoLT()* function, with the height difference DiffHeight as a parameter, to find an appropriate node $IndexNode$ in the big tree in which the new entry ST_Root should be placed.

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.

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 InsertSTintoLT( IndexNode, ST_Root, DiffHeight )
{
   if (DiffHeight <> 1)         // continue search another level down
   {
      DiffHeight--;
                                // choose for next level insertion node for ST
      chosenNode = ChooseSubtree( LT, IndexNode, ST_Root );
                                // recursive call until ST falls to suitable level
      InsertSTintoLT( chosenNode, ST_Root, DiffHeight );
                                 // resize MBR of nodes involved for ST insertion
      child[chosenNode].SetEnclose( ST_Root );

   } // endif
   else                          //   chosenNode is desired insert location
         InsertSTMBR( ST_Root, chosenNode );
   endif;
} // end Procedure
```

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

```
PROCEDURE InsertSTMBR( ST_Root, IndexNode )
{
   if ( numChildof( IndexNode ) < FANOUT ) // node not full (ideal case)
   {
      child[numChild++] = ST_Root;          // insert ST into indexNode
      SetEnclose( IndexNode, ST_Root );     // resize MBR of indexNode to hold ST_Root
   }
   else
       Invoke OverflowTreatment( ST_Root, IndexNode );
   endif;
} //
```

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 *OverFlow Treatment* procedure listed in Figure 5. First, we attempt to merge siblings 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 children 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 new entry slot for them. We then propagate multiple splits upwards as done in regular R-tree algorithm [A.G84] 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
      MergeSuccess = MergeSibling( S1, S2 );
      if ( not MergeSuccess )                  // if merge failed, try ReInsert
      {
                                               // choose  furthest child
         Child_del = SelectFurthestChild( IndexNode );
                                               // delete farthest child and
                                               // reinsert it into LT
         ReInsertSuccess  = ReInsert( Child_Del );
         if ( not ReInsertSuccess )
            SplitNodeQuadratic( IndexNode );   // else Split
         endif;
      } //endif;
}
```

Figure 5: The Overflow Treatment Procedure.

4 Analytical Model

4.1 Notations

We now establish an analytical model to compare the performance of the STLT algorithm with alternate solutions. We use the following notation :

- N = Total number of objects in the existing tree index structure (prior to insertions)
- n = Total number of objects to be inserted
- P = Maximum number of objects that can be stored in one data page
- F = Fanout of tree structure
- CP = Average percentage capacity of data objects in one data page
- CF = Average percentage of fanout of an index node of tree structure

Hence the average number of objects that is stored in one data page is $CP * P$, denoted by AP. The actual fanout of index nodes of the tree structure is $CF * F$, denoted by AF.

4.2 About the Buffer Context

In our implementation, we use a standard LRU buffer manager. The buffering of portions of the tree in main memory can significantly affect I/O costs due to the likelihood of somewhat localized traversals over the index structure. To our knowledge there is no known analytical model on the analysis of I/O costs of bulk insertions that takes the buffer impact into account.

To compare our STLT bulk insertion technique with the conventional OBO approach, we approach this problem by first presenting an I/O model assuming a zero buffer size. In other words, each and every access to an R-tree node is counted as an I/O. However, in the context of bulk insertion or bulk loading, the number of zero-buffer expected I/O cost is not likely to be practical for performance prediction in a buffered environment as more localized access pattern that make direct use of he pages still in buffer are likely to occur. To model the buffer impact, we then extent our basic model. Similar to previous efforts on establishing an analytical model for spatial joins [HJR97], we use a probability distribution function to measure the probability of a given tree node access to actual still be in the buffer. Based on this probability and the zero-buffer expected I/O cost, the estimated I/O cost for bulk insertions into an R-Tree using STLT and OBO techniques can then be computed.

4.3 Measuring the Zero-Buffer Expected I/Os (ZEIO)

In a zero-buffer environment (worst case scenario), each insertion using the OBO technique requires us to traverse downwards the large tree to find a leaf node for insertion. Each traversal down takes on an average $O(\ log_{AF}\ N/AP)$, because a data rectangle will be in at most $O(\ log_{AF}\ N/AP)$ pages on its insertion path. Thus for a set of n inserts, we have $COST_{OBO} = O(\ n * log_{AF}\ N/AP)$.

For the STLT algorithm, we must first perform the insertion of all new data into the small tree. For simplicity, we assume that the construction of the small tree is done using the OBO technique – though of course any alternate bulk loading technique could be employed for this task. We thus use a worst case estimation here, though we do not expect that this will too significantly favor the OBO over the STLT as the cost is fairly minimal.

The construction of the small tree can be approximated by n insertions of objects into a tree of size n. Again, this corresponds to an upper bound as the small tree is initially empty. Thus, the construction cost of n objects into the small tree can be approximated by $O(\ nlog_{AF}\ n/AP)$. Then we must insert the small tree into the large tree as a whole. This corresponds to one single OBO insertion at the cost of $O(\ log_{AF}\ N/AP)$. If the initial insert attempt fails, then the various overflow handling techniques given in Section 3.4 would be employed. All of them are likely to take at most $O(\ log_{AF}\ N/AP)$ time, namely, with the worst case being the split all the way up the tree. So the insertion cost for STLT is the sum of these two parts, namely, $COST_{STLT} = O(\ nlog_{AF}\ n/AP\ +\ log_{AF}\ N/AP)$. The main contributor for this cost is the construction of the small tree.

Based on the above analysis of the I/O costs modeled by a zero buffer assumption, the percentage of I/O cost using STLT compared to the OBO technique is :

$$PCT_{S/O} = O((n\ log_{AF}\ n/AP\ +\ log_{AF}\ N/AP)\ /\ (n\ log_{AF}\ N/AP)) = O(1\ -\ log_{(N/AP)}\ R),$$

where R denotes the ratio of the number of objects in large tree to the number of objects in the small tree. We can see from the formula that the fanout factor has nothing to do with the percentage improvement.

Furthermore, the larger the large tree is (N) as well as the larger the ratio between N and n is, the smaller the $PCT_{S/O}$ value becomes. This implies that STLT will show improved performance over the OBO for large existing trees (N) as well as for large ratios N/n. The latter indicates that we would prefer to insert a small tree in tree chunks that are considerably smaller than the large tree.

4.4 Adding the Buffer Model to the Estimation

When the buffer is taken into consideration, we need an empirical buffer model for I/O cost estimation. Our buffer manager uses an LRU replacement policy for page faults. It can be observed that the BHP (buffer hit probability) exhibits steady behavior after the buffer has warmed-up. Also, it is clear from experimental observations, that small buffer size improves the I/O cost for STLT to a larger degree than bigger buffer sizes. Therefore, a relatively small buffer is a good estimate of the steady state BHP. At first glance, we can get a rather coarse prediction indicator for BHP. Buffer Hit Probability $(BHP) = |BUF| \ / \ (N/AP)$ because there are N/AP pages to be inserted and buffer size is BUF.

In a real environment, we expect this to be higher than that indicated number because the constantly used top few levels of the R-tree are often pinned in memory, which are always most recently used. So the I/O cost can be approximated by $ZEIO * (1 - BHP)$. Hence, for larger buffer sizes, the BHP will be larger and hence the I/O cost will be smaller for both STLT and OBO.

5 Experimental Results

5.1 Experimental Setup

5.1.1 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 [A.G84], an I/O buffer manager, and many other supporting data structures and procedures. The major modules, all written in C++, are depicted in Figure 6.

5.1.2 Test Data

The test data comprised both real datasets and synthetic datasets. 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. Most of our tests are based on synthetically generated testing datasets to be able to verify the usefulness of STLT under different extreme settings. The data is derived from the TIGER/Line files distributed by the US Census Bureau. The synthetic data was generated with varying parameters to control the distribution and the nature of the objects. Our experiments allow us to gain an understanding of bulk loading for skewed datasets based on different scenarios of experimental evaluations, and thus to establish guidelines to select the most suitable strategy for performing *bulk insertions* under given special circumstances.

5.1.3 Test Types

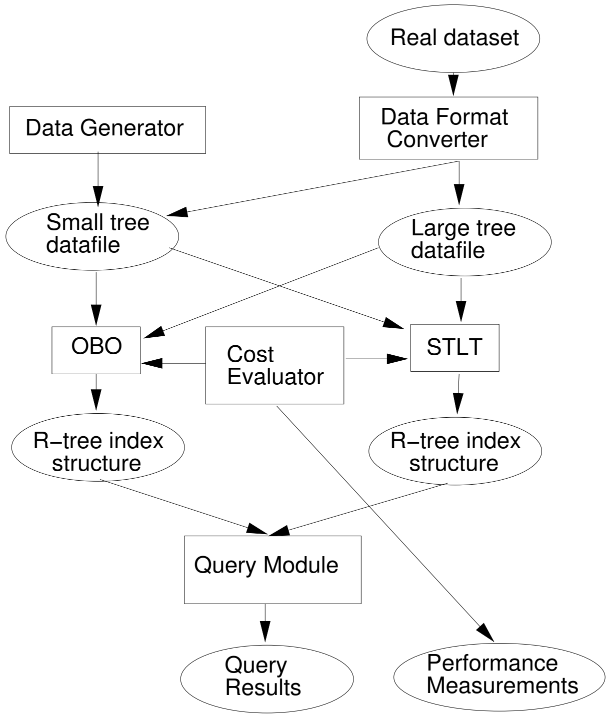

Figure 6: Implementation Setup

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 Section 5.2 to 5.11. 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 5.12 and 5.13).

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 Area :* Ratio of large tree region to small tree region.
- *Location of New Dataset :* Location of the new datasets within region of the existing dataset.
- *Shape of New Datasets :* If the new dataset is skewed, then it can be square-like or needle-like.
- *Density of Hole :* Number of objects of the large tree in the hole per square unit.

- *Average Size of Objects Compared to the Region Size :* Ratio of the actual object size as compared to the large tree region size.

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	100
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
New Data Location	Equal
Data Density of Hole	Sparse
Data Object Size	Small

Table 1: Default Parameters Settings

5.2 Experiment on Buffer Size

In this experiment, we varied the buffer size from 10 to 500 pages (with a fixed page size = 4K). We measured 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 7, we tested two cases, namely, a ratio of 25 and of 50.

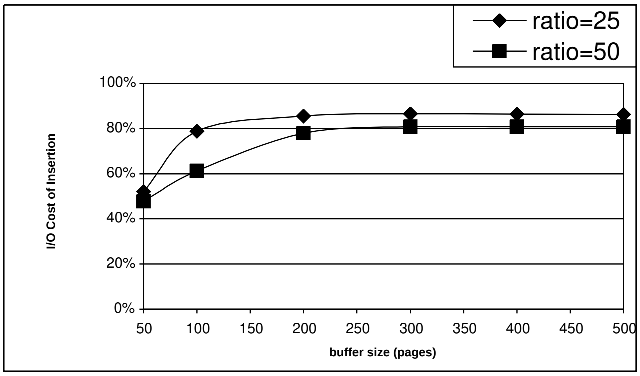

Figure 7: Effect of Different Buffer Sizes.

In Figure 7, the I/O costs for insertion by using OBO is represented by the straight line at the 100 percent cost position. The two curves represent the STLT insertion cost as a percentage of OBO 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 100 * 4K = 400K pages). Although STLT wins over OBO out in all the cases, its advantage over OBO is not as large with the increasing size of the buffer. Namely, for this fixed parameter setting, it levels out at a 20% 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 experiment, we also compared different ratios of 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. This then confirms our expectations as given by the analytical model that STLT favors larger ratios (See Section 4).

5.3 Experiment on Data Size Ratio

This experiment helps us to gain an understanding of how the ratio of data sizes affects the cost comparison. The results of this experiment are in sync with our analytical model indicating that the I/O cost is greatly reduced by STLT as compared to OBO when the large tree data size is significantly larger than the new data set. STLT reaches over 65% of improvement in I/O costs over the OBO when the small tree is kept sufficiently small as compared to the large tree size as seen from Figure 8.

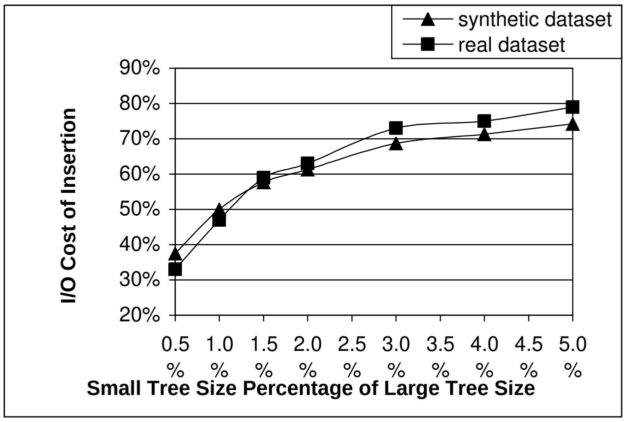

Figure 8: Relationship between STLT Cost Percentage of OBO versus Small Tree/Large Tree Size.

5.4 Experiment 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 at the same percentage.

The results as seen in Figure 9 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 80% 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 and further in favor of STLT as the tree grows.

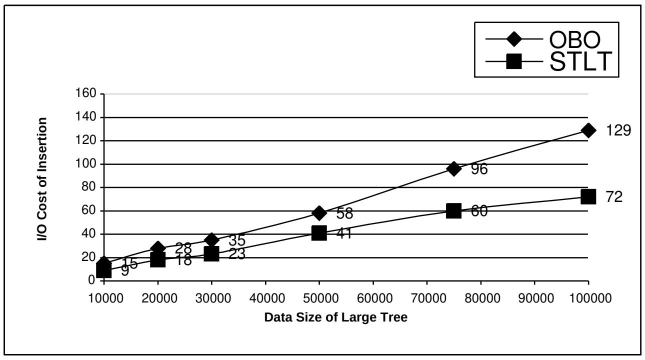

Figure 9: Relationship between I/O Insertion Cost and Data Size.

5.5 Experiment on Data Area Ratio

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 to test the limitations of STLT to determine if and to what degree 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. We keep the number of objects in the large tree to be always 50 times that of the new dataset.

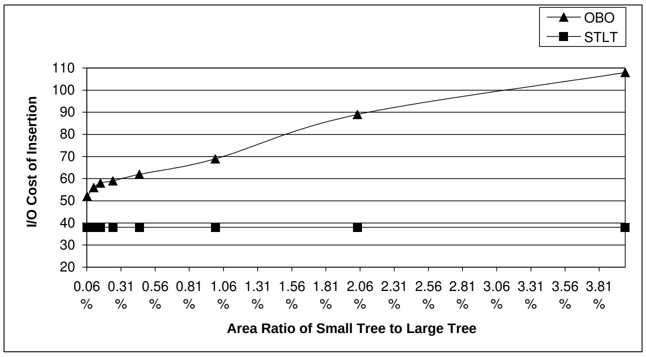

Figure 10: I/O Insertion Cost versus Area Ratio of Small Tree to Large Tree.

Our experimental results depicted in Figure 10 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 correspond to the lower straight line. For OBO, the I/O costs increase when the region covered by the new data is enlarged 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 300% 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, 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.

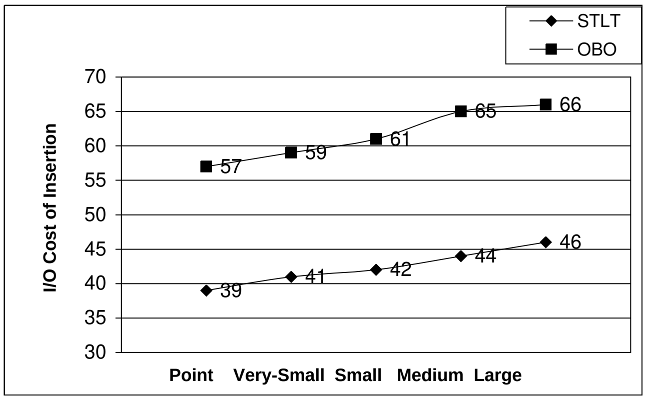

Figure 11: Relationship between I/O Cost and Data Object Size.

5.6 Experiment on the Size of Objects

From Figure 11, we see that the insertion I/O cost for both STLT and OBO increase when the data objects become bigger (have larger MBRs). However, larger objects continue to favor the STLT technique and the percentage improvement of STLT over OBO seems to increase slightly with increase in the size of the objects.

5.7 Experiment on Hole Location

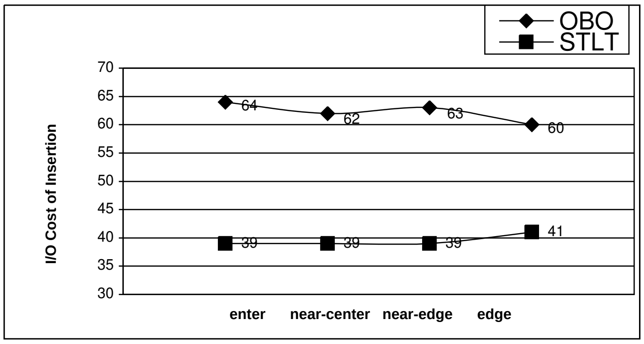

Figure 12: Relationship between I/O Cost of Insertion versus Location of Hole.

In this test the location for the new datasets is varied from the center of the large tree moving outwards towards the edge of the region. From the experimental results in Figure 12, we find that STLT has roughly the same cost, i.e. it is fairly independent of location, while OBO performs slightly better at the edge than in the center of the complete region. The reason for that might be that in OBO, if new datasets are clustered in the center, there are probably more neighboring rectangles than at an edge. So at the center, there would be a larger possibility for objects being inserted to be attracted to multiple paths. We conclude that datasets around the center would yield the worst results for OBO making the STLT technique even more desirable. However, this is not a major decisive factor.

5.8 Experiment on Hole Shape

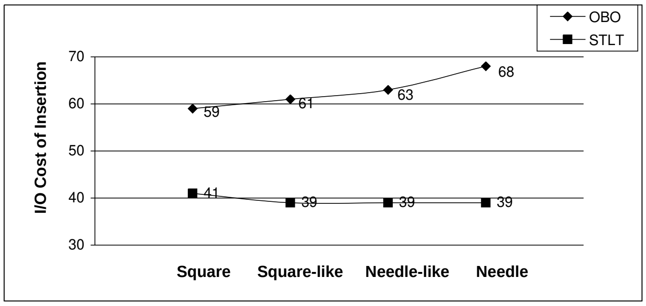

Figure 13: Relationship between I/O Cost of Insertion versus Shape of Hole.

In this test the shape of the hole is varied from a square shape to a needle-like shape. The shape of the hole appears to have an impact on costs. Figure 13 shows us that the insertion costs stay constant for STLT when varying the shape of the hole from having a square to a needle-like shape. We observe that OBO does better for new data which has an aspect ratio closer to 1 (square shape) versus those which have a large (or small) aspect ratio (needle-shape). This is probably due to the fact that the lopsided perimeters of the needle-shaped objects increase the likelihood of divergence of the insertion path. Also locality of buffering for point-like shapes can be expected leading to a reduction in I/O costs for point-like shapes.

5.9 Experiment on Hole Region in Relationship to New Data Region

We now evaluate the effect of the location for the new dataset to be in a less densely populated area of the large region (a "hole"). We vary the new data position to be totally within the hole, exactly overlapping the hole, intersecting the hole, covering the hole and disjoint from the hole.

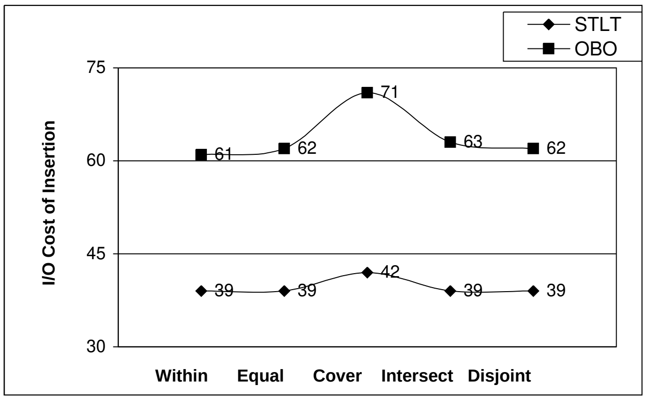

Figure 14: I/O Costs for Different Relative Locations of the Hole and Small Tree.

Our results in Figure 14 show that the cover relation case yields the worst case for OBO and would hence be the most preferred case for STLT. This is because, when the new dataset region covers the hole region, since its peripheral areas fall into a dense large tree area, the objects at its periphery would be distributed over more large R-tree nodes, thus increasing the I/O costs for OBO.

5.10 Experiment on Data Density of Hole

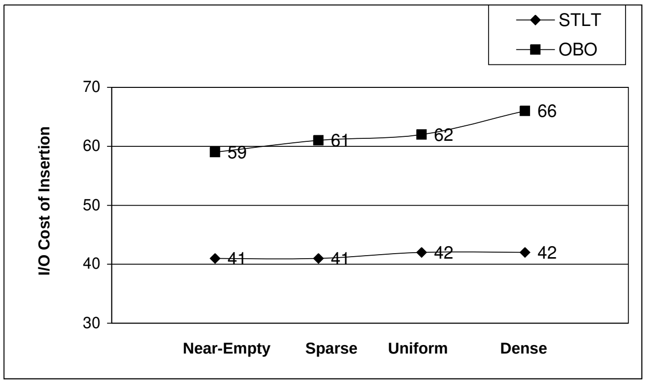

Figure 15: Relationship between I/O Cost and Data Density in the Hole.

The density of data in the hole may also influence the insertion costs. The experiment shows (see Figure 15) that OBO seems to get worse as the density of the hole increases whereas STLT remains fairly unaffected.

5.11 Experiment on Real Dataset

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.

This is another experiment where the large tree size is fixed and the ratio of the large tree size to the small tree size is varied. The result in Figure 8 shows the same trend as with the synthetic data and gives even slightly better percentage improvement of STLT over OBO in some cases. For the bigger ratio cases of large tree size to the small tree size, the experiment for real dataset does better than the synthetic dataset. However the performance of STLT seems to deteriorate a little in the small ratio cases. This might be because the real dataset is also skewed in some manner and it may be suitable for the small dataset to be inserted as a small tree which may no longer hold true when the new dataset becomes larger.

5.12 Experiments on Tree Quality using Random Queries

In this query testing scenario, the data for a 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 data content. A large number of random queries (700 queries) are then generated. The testing parameters are the same as for the insertion tests above (Sections 5.2 - 5.11). The query rectangle was 5 times larger than the maximum object size in the resulting tree.

In this experiment (Figure 16), we evaluate the quality of the R-tree constructed by either STLT or OBO by executing random queries with the query rectangle placed at a random location over the large tree region. We measure the cumulative number of nodes visited in order to answer the queries.

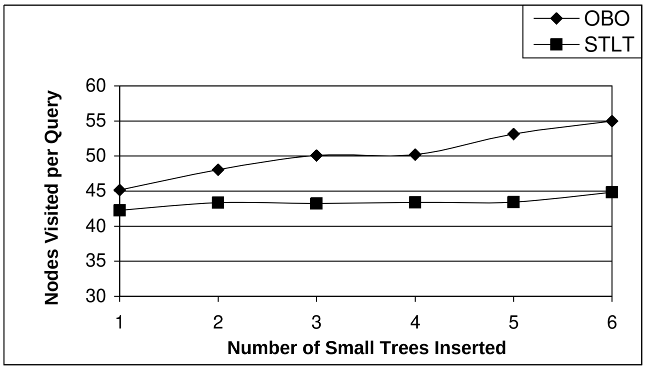

Figure 16: Number of Nodes Visited for Random Queries.

STLT does better in terms of the number of nodes visited. This suggests that the quality of the tree is slightly better. This could be because we restricted our experimental setup to parameters settings for which the STLT was designed initially, to small sizes of data and number of objects in the small tree is a fairly small ratio of the large tree and the objects have skewed distribution. The 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.

Increasing the number of small trees that are inserted over time shows some cost increase in the performance of the queries. The slight performance increase may be attributed to growing tree sizes, plus the MBRs of the top nodes are likely to expand forcing down queries to search over more paths. Clearly, we conclude that STLT leads to trees which yield acceptable and comparable query performance for reasonable sized new datasets.

5.13 Experiment 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 would 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.

As seen from Figure 17, we find that the STLT does better in terms of the number of nodes visited as compared to random queries - which confirms our expectation. This suggests that STLT query performance would be definitely comparable to OBO, if not better if localized queries were asked. The performance results indicate the strength of STLT in answering queries localized to newly inserted data.

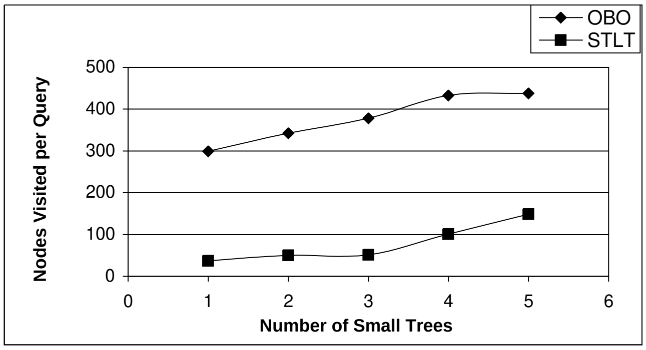

Figure 17: Number of Nodes Visited for Random Queries.

6 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. The overall strategy of group-first and then insert-as-bulk is however a general approach and thus could also be explored for other multi-dimensional index structures that are tree-structure based.

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. Besides an analytical model, we also 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 80% 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 practically all 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 some 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.

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

[A.G84] A.Guttman. R-trees : A dynamic index structure for spatial searching. *Proceedings of SIGMOD*, pages 47–57, 1984.

[B.O90] B.Ooi. Efficient query processing in geographic information systems. *Lecture Notes in Computer Science*, 1990.

[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.

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

[HJR97] Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. A cost model for estimating the performance of apstial joins using r-tree. *International Working Conference on Scientific and Statistical Database Management*, August 1997.

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

[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.

[SM96] S.Leutenegger and M.Lopez. The effect of buffering on the performance of r-trees. Technical report, University of Denver Computer Science Technical Report # 97-02, 1996.

[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.