Bulk Operations for Space-Partitioning Trees

Thanaa M. Ghanem Rahul Shah Mohamed F. Mokbel Walid G. Aref Jeffrey S. Vitter
Department of Computer Sciences, Purdue University
{ghanemtm,rahul,mokbel,aref,jsv}@cs.purdue.edu

Abstract

The emergence of extensible index structures, e.g., GiST (Generalized Search Tree) [25] and SP-GiST (Space-Partitioning Generalized Search Tree) [3], calls for a set of extensible algorithms to support different operations (e.g., insertion, deletion, and search). Extensible bulk operations (e.g., bulk loading and bulk insertion) are of the same importance and need to be supported in these index engines. In this paper, we propose two extensible buffer-based algorithms for bulk operations in the class of space-partitioning trees; a class of hierarchical data structures that recursively decompose the space into disjoint partitions. The main idea of these algorithms is to build an in-memory tree of the target space-partitioning index. Then, data items are recursively partitioned into disk-based buffers using the in-memory tree. Although the second algorithm is designed for bulk insertion, it can be used in bulk loading as well. The proposed extensible algorithms are implemented inside SP-GiST; a framework for supporting the class of space-partitioning trees. Both algorithms have I/O bound $O(NH/B)$, where N is the number of data items to be bulk loaded/inserted, B is the number of tree nodes that can fit in one disk page, H is the tree height in terms of pages after applying a clustering algorithm. Experimental results are provided to show the scalability and applicability of the proposed algorithms for the class of space-partitioning trees. A comparison of the two proposed algorithms shows that the first algorithm performs better in case of bulk loading. However the second algorithm is more general and can be used for efficient bulk insertion.

1 Introduction

With the increasing number of computer applications that rely on large multi-dimensional data sets, it becomes essential to provide efficient multi-dimensional access methods. Examples of these applications include multimedia databases, computer aided design (CAD), geographic information systems (GIS), and cartography. Numerous types of tree-based multi-dimensional access methods are proposed over the last three decades (e.g., see [23]). The main objective of these multi-dimensional access methods is to support efficient answers to similarity search queries, nearest neighbor queries, and range queries. Usually, the performance of an index structure for the *insert* operation is not satisfactory when inserting large amounts of data. To overcome this drawback in multi-dimensional access methods, a bulk loading algorithm is needed. A bulk loading algorithm has more knowledge than the standard insertion procedure where all the data is known a priori rather than tuple by tuple. Using this extra knowledge, the bulk loading algorithm has the opportunity to produce (1) better balanced structure, (2) better storage utilization, and (3) better query answering performance. Another related problem is *bulk insertion*. In contrast to bulk loading, where an index is built from scratch, bulk insertion aims to update an existing structure with a large set of data.

In this paper, we propose two new extensible bulk loading and bulk insertion algorithms for the class of *space-partitioning* trees; a class of hierarchical data structures that recursively decompose the space into disjoint partitions. The proposed bulk loading and insertion algorithms are implemented inside SP-GiST (Space-Partitioning Generalized Search Tree) [2, 3]; an extensible database index structure for the class of space partitioning trees. SP-GiST is a software engineering solution that allows fast realization of instances of space-partitioning index trees inside a commercial database system. SP-GiST provides generic index search and index maintenance operations, online node clustering into disk pages, concurrency control and recovery for all index instances for the class of space-partitioning trees. SP-GiST has interface parameters and methods that allow it to represent instance indexes of the class of space-partitioning trees and reflect the structural and behavioral differences among

them. The contribution of this paper can be summarized as follows:

1. We propose two new generic algorithms for bulk operations in the class of space-partitioning trees. The algorithms are implemented inside SP-GiST.

2. We use the I/O complexity model [45] to give bounds for the proposed algorithms in terms of the external tree height and number of data items to be bulk loaded/inserted.

3. We provide extensive experimental results which show that these algorithms in fact perform much better in practice than the worst case bounds.

The rest of the paper is organized as follows: Section 2 highlights related work for bulk loading and bulk insertion algorithms. The main characteristics of the class of space-partitioning trees are discussed in Section 3. An overview of the SP-GiST framework is presented in Section 4. The proposed extensible algorithms for bulk loading and bulk insertion for the class of space-partitioning trees are presented in Section 5 along with the analytical study of their performance. Section 6 provides an extensive set of experiments that gives the performance of the proposed algorithms. Finally, Section 7 contains concluding remarks.

2 Related Work

The class of space-partitioning trees can be further divided into data-driven and space-driven trees. For the data-driven trees, bulk loading algorithms benefit from data sorting. In [40], a bulk loading algorithm is proposed for the point quadtree [20]. The key idea is to build an optimized point quadtree so that, given a node A, no subtree of A accounts for more than one half of the nodes rooted at A. This requires sorting the data and taking the median as the root while dividing the remaining data into four groups. In addition, if the tree fails to meet a predefined balanced criteria, the tree is partially rebalanced. In [22], the *adaptive* k-d tree is proposed as a bulk loading approach for the k-d tree [8], where data is stored only in leaf nodes while interior nodes contain the median of the set of the children nodes.

A top-down approach for bulk loading space-driven quad tree variants is proposed in [27] and later is enhanced in [26]. The main idea is to fill up the memory with as much of the quadtree as possible before flushing some of its nodes into a disk-based index. The input data is sorted in such a way that the portions that are written out to the disk would not be inserted into again. Sorting input is performed via the Z-order [39] of the lower-left corner of data objects.

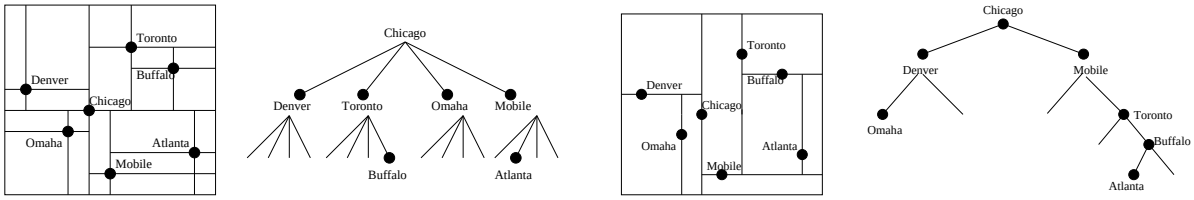
Several algorithms are developed for bulk loading in the R-tree [24]. The main idea is to pre-sort the data before bulk loading. The reader is referred to [29, 33, 42] for further details. Other bulk loading algorithms include a sort-based approach [32] for bulk loading a B-tree [7], a top-down approach [9, 11] for bulk loading an X-tree [10], a sample-based approach [14] for bulk loading an M-tree [15]. However, these bulk loading algorithms are specific to the index structure in question. Most of the bulk insertion algorithms are concerned with the R-tree. Sort-based approaches are proposed in [37, 12, 13, 30, 41], while in [5], the idea of the buffer-tree [4] is utilized for bulk insertion.

The emergence of new extensible multi-dimensional index structures, e.g., GiST [25] and SP-GiST [3], calls for extensible bulk loading algorithms to support a broad class of multi-dimensional indexes. In [16, 17], generic algorithms that utilize the buffer-tree [4] for bulk loading the class of *Grow and Post* trees [34] (e.g., the class of trees supported by GiST) is presented. Another extensible bulk loading algorithm is proposed in [1] for the class *weight-balanced trees* (e.g., the k-d tree [8], the BBD-tree [6], and the BAR-Tree [19]). To the best of our knowledge, this is the first attempt to develop bulk loading/insertion algorithms for the general class of space partitioning trees, which may not be height balanced.

3 The Class of Space-Partitioning Trees

The class of space-partitioning trees is a class of hierarchical data structures that recursively decompose the space into disjoint partitions. The class of space-partitioning trees can be further divided into *data-driven* space-partitioning trees and *space-driven* space-partitioning trees. In data-driven space-partitioning trees, the space is decomposed based on the input data. Examples of data-driven space-partitioning trees are the point quad tree [20] and the k-d tree [8]. In space-driven space-partitioning trees, the space decomposition does not depend on the order of data insertion. Examples of space-driven space-partitioning trees include the trie index [21], the region quadtree [20], the MX quadtree [31], the PM quadtree [44], the PR quadtree [38] and the PMR quadtree [36]. The class of space-partitioning trees can be structurally distinguished from other tree classes in the following:

1. Space-partitioning trees decompose the space re-



(a) Point Quadtree

(b) k-d tree

Figure 1. Example of a point quadtree and a k-d tree.

cursively. Each time, a fixed number of disjoint partitions is produced.

2. Due to its limited fanout (e.g., the quadtree has only a fanout of four), space-partitioning trees are unbalanced trees that can be skinny and long.

3. Two different types of nodes exist in a space-partitioning tree. Non-leaf nodes are index nodes while leaf nodes are data nodes.

In the following, we give a brief overview of some of the commonly used space-partitioning trees. For more details, the reader is referred to [43].

The trie [21]: A trie is a tree structure for storing strings in which there is one node for every common prefix. Two types of nodes can be distinguished; index nodes are non-leaf nodes of the trie that are used for storing common prefixes, and data nodes are leaf nodes in the trie that are used to store the data. The trie is commonly used to store words in a dictionary. The trie is a space-driven decomposition structure, where at each level, the space is partitioned into a number of partitions equal to the number of alphabets and an additional blank. For example, a trie for English words partitions the space at each level up to 27 disjoint classes.

The Patricia Trie [35]: The Patricia trie is a space efficient implementation of the trie, where all nodes with one child are merged with their parents. The merge is applied to both leaf and non-leaf nodes. Most practical systems implement the Patricia trie instead of the trie.

The Quadtree: Quadtrees can be based on either a space-driven or a data-driven decomposition. Space-driven quadtrees, also known as quadtries [43] have the same structure as the trie. Examples of the space-driven quadtrees are, the MX quadtree [28], the MX-CIF quadtree [31], the PR quadtree [38], the PM quadtree [44], and the PMR quadtree [36]. Figure 1a gives an example of point quadtree as an example of data-driven quadtrees.

The k-d tree [8]: The k-d tree is a multi-dimensional search tree, useful for answering range queries about a set of points in the k-dimensional space. The k-d tree is an improvement over the point quadtree where it reduces the branching factor at each node and the storage requirements. Levels of the tree are split into two subspaces along successive dimensions of the multi-dimensional points. The k-d tree is a binary search tree with the distinction that at each level, a different dimension (key) is tested to determine the dimension in which a branch is made. For the k-dimensional space, at level L, dimension number L mod k + 1 is used, where the root is at level 0. Therefore, the first dimension is used at the root, the second dimension at level 1, and so on, until all dimensions have been used. The dimensions are used again beginning at level k. As in the point quadtree, the decomposition in the k-d tree depends on the data values and hence is data-driven. Figure 1b gives an example of a k-d tree.

4 SP-GiST: A Framework for Supporting the Class of Space-Partitioning Trees

In this section, we give a brief overview of SP-GiST (Space-partitioning Generalized Search Trees). The reader is referred to [2, 3] for further detail. SP-GiST is a general index framework for the class of space-partitioning trees. Different space-partitioning trees can be realized inside SP-GiST through a number of *interface parameters* and *external methods*. In addition, SP-GiST has *internal methods* that reflect the similarity between space-partitioning trees for insertion, deletion, and search that are already implemented inside the SP-GiST index engine. The user of SP-GiST provides only the external methods and interface parameters, while the internal methods are hard coded into the SP-GiST index engine. In the following, we give an overview of some of the important interface parameters, external methods, and internal methods of SP-

	Patricia trie	k-d tree
Parameters	ShrinkPolicy = Tree Shrink, BuckeSize = B NoOfSpacePartitions = 26 NodePredicate = letter or blank Key Type = String	ShrinkPolicy = Leaf Shrink, BuckeSize = 1 NoOfSpacePartitions = 2 NodePredicate = "left", "right", or blank Key Type = Point
Consistent(E,q,level)	If (q[level]==E.letter) OR (E.letter ==blank AND level > length(q)) Return True, else Return False	If (level is odd AND q.x satisfies E.p.x) OR (level is even AND q.y satisfies E.p.y) Return True, else Return False
PickSplit(P,level)	Find a common prefix among words in P Update level = level + length of the common prefix Let P predicate = the common prefix Partition the data strings in P acording to the character values at position "level" If any data string has length < level, Insert data string in Partition "blank" If any of the partitions is still over full Return True, else Return False	Put the old point in a child node with predicate "blank" Put the new point in a child node with predicate "left" or "right" Return False

Table 1. Realization of the Patricia trie and the k-d tree inside SP-GiST.

GiST.

SP-GiST Interface parameters:

- *NodePredicate:* This parameter gives the predicate to be used in the index nodes of a space-partitioning tree. For example, a quadrant in a quadtree or a letter in a trie are predicates that are associated with an index node.
- *KeyType:* This parameter gives the type of data in the leaf-level of the tree. For example, the MX quadtree uses a predefined data type of *Point* while the trie use a key type of *Word*.
- *NumberofSpacePartitions:* This parameter gives the number of disjoint partitions produced at each decomposition. For example, this parameter is set to four in the case of the quadtree and to 26 in the case of a trie that contains English alphabets.
- *Resolution and ShrinkPolicy:* These parameters limit the number of space decompositions and are set depending on the required granularity.
- *BucketSize:* This parameter gives the maximum number of data items a data node can hold.

SP-GiST External Methods

- *Consistent(Entry E, Query Predicate q, level):* A *Boolean* function that is used by the search method as a navigation guide through the space-partitioning tree.
- *PickSplit(P, level, splitnodes, splitpredicates):* This method defines a way of splitting the entries into a number of partitions and returns a *Boolean* value indicating whether further partitioning should take place or not.
- *Cluster ():* This method defines how tree nodes are clustered into disk pages.

SP-GiST Internal Methods

- *Insert:* This method is used to insert a new data entry to the existing tree structure. The *insert* method uses the interface parameter *PathShrink* and the external methods *PickSplit* and *Consistent*.
- *Search:* Searching in space-partitioning trees starts from the root. The search item is checked againist all branches using the external method *Consistent*.
- *Delete:* Deleted items in SP-GiST are marked deleted but are not physically removed from the tree. A rebuild is used from time to time as a *clean* procedure.

The realization of any space-partitioning tree T inside SP-GiST is achieved by providing the interface parameters and external methods for T. For example, Table 1 gives the realization of the *Patricia trie* and k-d tree inside SP-GiST. Examples for the realization of other space-partitioning trees inside SP-GiST can be found in [2, 3].

5 Extensible Algorithms for Bulk Loading Space-Partitioning Trees

In this section, we present two extensible buffer-based bulk loading and bulk insertion algorithms for the class of space-partitioning trees (termed, the *Direct Buffering Bulk Loading* algorithm (DBBL) and the *Buffer Tree Bulk Insertion* algorithm (BTBI)). Both algorithms use disk-based buffers to distribute data items. The BTBI algorithm is more general than the

DBBL algorithm in the sense that BTBI is designed to handle bulk insertions as well as bulk loading. However, as we will see in the performance section, BTBI performs slightly worse than DBBL in terms of the I/Os incurred. However, BTBI is a very effective when we want to insert a large number of data items into an already operational tree. The main idea of both bulk loading algorithms is to build an in-memory tree of the desired index structure using the standard insertion procedure. Then, the in-memory tree is used to distribute the remaining data items into disk-based buffers. Bulk loading algorithms are applied recursively to the disk-based buffers.

5.1 Direct Buffering Bulk Loading Algorithm

The *Direct Buffering Bulk Loading* algorithm (DBBL, in short) partitions the available memory into two equal partitions, namely the memory part and the buffer part. Data items are loaded into an in-memory space-partitioning tree in the memory part until it is exhausted. The space-partitioning tree is clustered into memory pages using the clustering algorithm by Diwan et. al. [18]. Then, the DBBL algorithm utilizes the buffer part by associating a buffer to each clustered tree memory page. Only one page of the associated buffers resides in memory, while, the rest of the buffer is in disk. In other words, half of the memory is reserved for clustering the in-memory tree, while the other half is reserved by memory buffers; one buffer per clustered tree page. The remaining data items, if any, are distributed among memory buffers using the in-memory tree. The DBBL algorithm works recursively on each buffer until all buffers are processed.

Pseudo Code. Figure 2 gives the pseudo code of the DBBL algorithm implemented inside the SP-GiST engine [3]. A *Buffer List* is used to keep track of a list of buffers that contains input data. Initially, the *Buffer List* contains only the input file F. For any buffer B in the *Buffer List*, the best-case scenario is when all data items in B can fit in memory. In this case, an in-memory tree with all data items can be built inside memory and flushed once to disk (Step 2 in Figure 2). However, if data cannot fit in memory, the DBBL buffering algorithm starts by reading as much data as can fit in half the memory (Step 3 in Figure 2). Such data is used to build an in-memory tree using the standard insertion procedure that is implemented as an internal method inside SP-GiST (*SP-GiST.Insert()*). A distinct property in space-partitioning trees is that the node size is much less than the page size. Thus, the insertion procedure in SP-GiST uses the clustering

Algorithm *The Direct Buffer Bulk Loading algorithm for SP-GiST, DBBL*
Input:

- F: The data input file.
- R: Root of the tree (initially, empty).

Begin

- Initialize *Buffer List* to contain only F.
- While *Buffer List* is not empty

1. B is the first buffer in the *Buffer List*.
2. If all items in B can fit in memory. Then, build an in-memory tree using the standard insertion procedure, and flush it to disk. Then, Go To Step 7.
3. While available memory is less than half full and B has data
   - Read record x from B.
   - Insert x in the in-memory tree, rooted at R, using SP-GiST.INSERT (R,x) method.
4. Associate a buffer page p with each page from the in-memory tree.
5. While B has data
   - Read record x from B.
   - Call SP-GiST.LocateLeaf(R,x) to locate the leaf node L in which x should be inserted.
   - Insert x in the buffer B_x that is associated with the page contains L.
     //Only one page of B_x is in-memory, the
     //rest of the buffer (if needed) is on disk
   - If x is the first item to be inserted in B_x, add B_x to the *Buffer List*.
6. Flush the in-memory tree into disk.
7. Delete B from the *Buffer List*.

End.

Figure 2. Pseudo code for the DBBL algorithm.

algorithm by Diwan et. al. [18] to pack tree nodes into pages. Diwan's clustering algorithm is optimal in the sense that it results in a minimum tree height in terms of clustered pages. Applying the clustering algorithm for each inserted item is done in memory, thus node clustering into pages does not incur any I/O overhead. SP-GiST uses the dynamic version of the clustering algorithm [18]. Thus, clustering is incremental as data is inserted in the tree. Incremental clustering avoids extra CPU overhead.[1]

For the other half of the memory, the DBBL algorithm associates an empty buffer page to each clustered

[1] We could recluster the entire tree to attain the strictly minimum height, in one pass over the tree once the bulk loading algorithm is executed. However, this does not seem to affect the external height much.

page (Step 4 in Figure 2). The remaining records in B (if any), are distributed among memory buffers using the clustered in-memory tree. (Step 5 in Figure 2). To insert a data item x into a memory buffer, the DBBL algorithm calls the SP-GiST internal function *SP-GiST.LocateLeaf(R,x)* to locate the final destination page L for x. Then, x is inserted in the buffer associated with page L. The correctness of this step comes out from the properties of Diwan's clustering algorithm [18] where the clustering algorithm guarantees that there is only one tree in every clustered page. This means that for each page, there is a root node from which we can reach for every other node on this page. If the insertion of x results in a new buffer B_x, we add B_x to the *Buffer List*. Finally, the in-memory tree is flushed into disk (Step 6 in Figure 2) and the buffer B is deleted from the *Buffer List* (Step 7 in Figure 2). The DBBL algorithm recursively operates in all buffers until the *Buffer List* is empty.

Example. Figure 3 gives an example of bulk loading nine data items into a *trie* using the DBBL algorithm. The memory can accommodate up to six pages. The first three entries (*"aaa","aac","cbc"*) consume three memory pages (i.e., half the memory space). Page boundaries are represented as dotted rectangles in Figure 3. One buffer is associated with each clustered tree page. The remaining six data items are partitioned into the three buffers (see Figure 3a) using the clustered tree. For example, when we insert the data item "*bca*", we go through the root page (i.e., Page I). Data item "*bca*" cannot go through page II or Page III since Page II is for data items with Prefix "*aa*", while Page III is for data items with Prefix "*c*". In this case, Page I is the final destination for the data item "*bca*". Thus, "*bca*" is inserted into the buffer associated with Page I (i.e., Buffer I). Similarly, the data item "*cca*" is inserted into buffer III; the associated buffer with Page III. Notice that some of the buffers may be empty (e.g., buffer II), while other buffers may need extra disk space (e.g., buffer III). In this example, we assume that the memory buffer can accommodate up to three entries. The DBBL algorithm works recursively on the memory buffers. Figures 3b and 3c give the in-memory subtrees that result from applying the DBBL algorithm on Buffers I and III, respectively. Notice that, in these figures, there is no need to divide the whole memory into two partitions since the memory can accommodate all data items and hence there will be no need for buffers. For example, all the data items in Buffer I are used to build an in-memory tree of three pages (Figure 3b), while the data items in Buffer III are used to build an in-memory tree of six pages (Figure 3c). The final

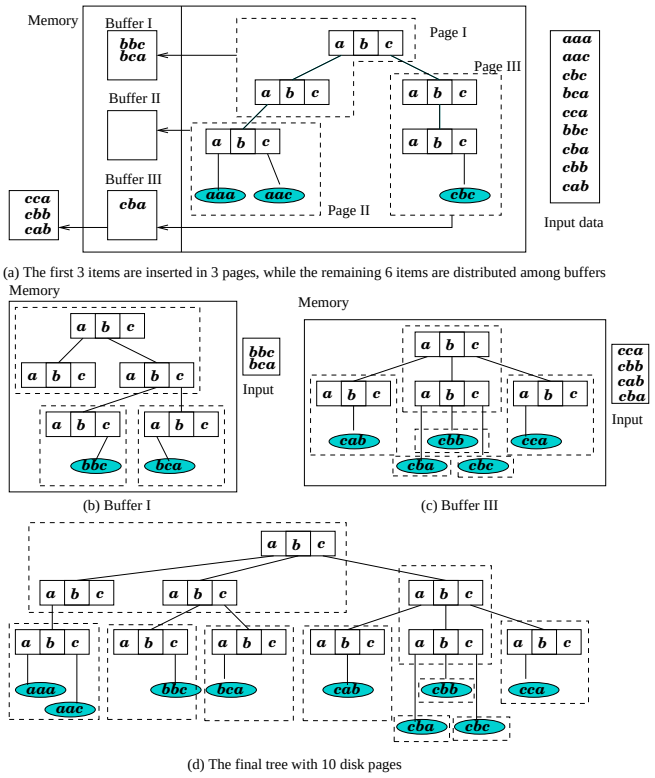


Figure 3. Example of trie bulk loading using the DBBL algorithm.

disk-based tree is given in Figure 3d where the whole tree takes ten disk pages.

Analysis. We analyze DBBL in terms of its I/O complexity. Let N be the amount of data, B be the amount of data that fits in a single disk page (i.e., I/Os are in blocks of size B), M be the amount of data that can fit in the internal memory. For complexity purposes, N will also denote the number of nodes in the tree, B will also denote the number of tree nodes that fit in a page and M will denote the number of tree nodes that fit into internal memory. Notice that the size of the tree nodes is constant that we can hide under the liberty of big-O notation. Let H be the external height of the tree, which is the maximum number of pages encountered on any path from root to leaf.

We account for the number of read page and write page operations. In each recursive call, we bulk load data items till we have an in-memory tree of size $M/2$. Then, the in-memory tree is clustered into $O(M/2B)$ pages and the remaining items are distributed onto disk based buffers. Then, the clustered pages of the tree are written into the disk to form part of the tree. At the

next level of recursion, a page from the tree is reloaded into memory along with its corresponding buffer and the algorithm is applied recursively. Every page in the data structure is written once and is read once resulting in $O(N/B)$ I/Os. The main cost of I/O comes from writing and reading buffers which happens in blocks of size B. If H is the external height of our final tree, than every data item is read and is written at most H times. Since I/Os happen in blocks of size B, we get $O(NH/B)$ as the worst case I/O complexity bound.

However, we will see that even if H is large, our algorithm achieves the number of I/Os much closer to N/B in terms of the constant factor. This is because $O(NH/B)$ is the worst case bound assuming the tree is lop-sided. In the average case, we can assume that the data items fall almost equally into $M/2B$ buffers in each recursive call. Hence, at each next recursion level the buffer size reduces by the factor of $M/2B$. In this case, the depth of the recursion is only $\log_{M/2B} N/B$ nested recursive calls. Hence, no data item will be written to buffers more than $O(\log_{M/B} N/B)$ times. Thus, in the average case, we achieve an I/O bound of $O(N/B \log_{M/B} N/B)$ for this algorithm, which is also an optimally tight bound because sorting is lower bounded by this complexity in the I/O model.

5.2 The Buffer Tree Bulk Insertion Algorithm

The *Buffer Tree Bulk Insertion Algorithm* (BTBI, for short) is more general than DBBL in the sense that BTBI can be used for bulk insertions into an existing tree. We adopt the idea of buffer trees [5] during bulk insertions into SP-GiST. The main difference over [5] is that space-partitioning trees may not be balanced and hence need non-trivial clustering. As in DBBL, we associate a buffer per index page. However, data items in the buffers are pushed down only one page at a time. BTBI can work in two forms: (1) Assuming all the data to be inserted is available – in this case we assume that there are at least as many new data items being inserted as there are already in the tree; (2) When insertions arrive as a stream – in this case the insertions are batched at every level of the tree and insertions descend the pages of the tree only when there is a sufficient number of insertions available to amortize the cost. Whenever a sufficient number of data items accumulate at the buffers in the leaf level, we cluster them to form a new part of the tree.

Pseudo Code. Figure 4 gives the pseudo code for the *Buffer Tree Bulk Insertion* algorithm (BTBI) when all the items to be inserted are already in a file. BTBI can be used for both bulk loading and bulk insertion. The input root tree R may be null (bulk loading) or contains an already existing space-partitioning tree (bulk insertion). If the root page R is the only page in the tree, then BTBI starts by building an in-memory space-partitioning tree using the *SP-GiST.INSERT()* internal method. Note the difference between DBBL and BTBI in this step. While DBBL utilizes only half the memory to build the tree, BTBI utilizes the whole memory space for building the tree. Once the memory is exhausted, BTBI associates a disk-based buffer for each clustered tree page. Initially, the input file F is considered the buffer associated with the root page (the page containing the root node).

If a disk-based buffer b is associated with a leaf page P, BTBI is recursively called with b as its input file and P as its root page (Step 1 in Figure 4). However, if the disk-based buffer b is associated with an intermediate page P, BTBI loads the page P in memory and associates an in-memory buffer, say q_i, with each page, say p_i, fanning out from P (Steps 2 and 3 in Figure 4). Thus, the available memory is used for building in-memory buffers. Unlike DBBL, where the avail-

Algorithm *Buffer Tree Bulk Insertion algorithm for SP-GiST, BTBI*
Input:

- F: The data input file.
- R: Root page of the tree T.

Begin

- If R is the only page in the tree
  - While memory is available and F has data
    - Read record x from F.
    - Insert x in the in-memory tree, rooted at R, using SP-GiST.INSERT(R,x) method.
- Associate an empty disk buffer with each page of T.
- Initialize the buffer at R with remaining data items in F.
- For each disk buffer b associated with page P
  1. If P is a leaf page
     - BTBI(b,P); Continue.
  2. Load page P in-memory.
  3. Associate in-memory buffer pages q_i with each page p_i fanning out from the in memory tree.
  4. While b has data
     - Read record x from b.
     - Locate q_i into which x goes. Insert x into q_i
  5. Append q_i's to the disk buffers corresponding to respective p_i's.

End.

Figure 4. Pseudo code for the BTBI algorithm.

able memory is utilized for the tree and the in-memory buffers at the same time, in BTBI, the available memory alternates between building the tree and using the buffers. The data items in b are distributed among the in-memory buffers q_i. The distribution is performed according to the tree at Page P. Notice that the clustering algorithm [18] guarantees that there exists only one root node Q for each clustered page P. Finally, the entries in the in-memory buffers q_i's are appended to the corresponding disk-based buffers p_i's.

Example. Figure 5 gives an example of bulk loading a trie using BTBI. The input data set is given in Figure 5a, which is the same input data set for the example in Figure 3. The memory size is set to accommodate four pages. Only the first four data items can be inserted in memory (Figure 5b). The page boundaries are plotted in dotted rectangles. Then, the in-memory tree is flushed into disk. The remaining five input data items are considered the data in the buffer associated with the root node. A line with double arrows connects each buffer with its associated page. Figure 5c gives the distribution of the remaining five data items into the existing tree. The root page is loaded into memory, while the remaining data items are distributed into two buffers: Buffer I and Buffer II. Figures 5d and 5e give the resulting sub-tree after inserting the data items from Buffer II and Buffer I, respectively. Notice that when we distribute data items from buffer I, we need to have buffer III that contains only one item (Figure 5e). Buffer III would result in the sub-tree given in Figures 5f. The final tree is depicted in Figure 5g. Notice that the tree has the same final shape as the one in Figure 3d. However, the page clustering is different. The difference in page clustering is a result of having different sub-trees in both algorithms.

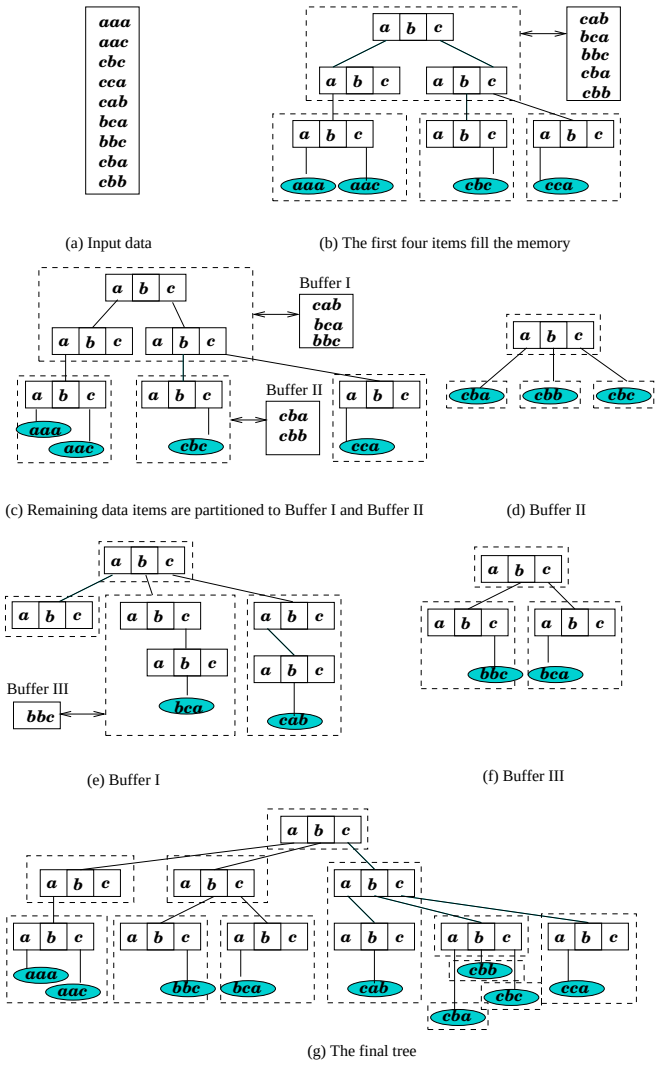
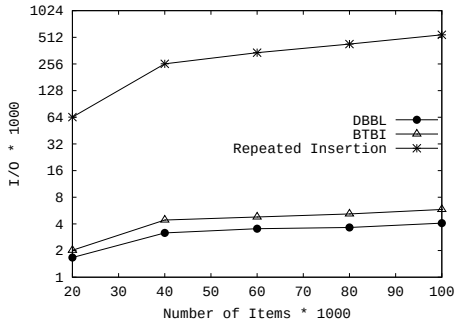


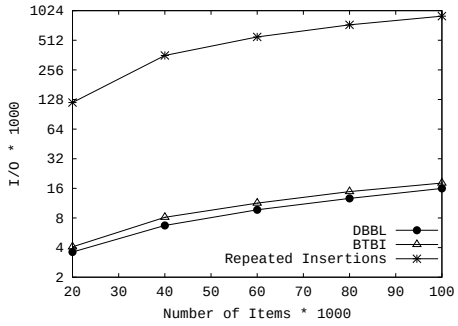
Figure 5. Example of trie bulk loading using BTBI Algorithm.

Analysis. Assume that there are N items to be inserted into an already functional tree containing N items. Here, each insertion descends its usual path in the tree towards the leaf. A step consists of pushing down insertions in blocks of size B across a given page. We divide the cost of I/O into two parts. One part is attributed to the pages in the tree while the other part is attributed to the buffers. During the process of pushing down the data items in a buffer, if the number of insertions crossing through a particular page is $X > B$, then it takes X/B I/Os to read the buffer and 1 I/O to read the page. While writing a buffer, if the buffer has $Y > B$ data items, then the cost of this write is attributed to the buffer else the cost of write is attributed to the next page to which this buffer is associated. Thus, every page has at most three I/Os associated with it: one for reading the page and one each for writing and reading its buffer page when its buffer page has less than B data items. This accounts for $O(N/B)$ I/Os. The remaining I/Os are for the buffer reads and writes that are in blocks of $O(B)$ data items. Since every data item in this process is read and is written at most H times, therefore the total number of I/Os is $O(NH/B)$.

In the case when insertions arrive as a real time stream, before pushing insertions across a page we lazily wait till we collect a sufficient number of data items to be pushed down, so as to amortize the cost of I/O. Typically, we wait till the buffer has at least B^2 items before pushing it down. Assume that the fanout



(a) Patricia trie



(b) KD-tree

Figure 6. Bulk loading Vs. Repeated insertion.

(i.e. the number of downward pages accessible) from each node is proportional to the number of data items stored in the nodes. Then, there can be at most $O(B)$ buffers into which the data is distributed. We only distribute when the size of the buffer becomes more than B^2, this distribution will result in at most $O(B)$ I/Os. Therefore, by amortizing the cost over the number of data items, each data item causes $O(1/B)$ I/Os during one distribution phase. Since each data item can be pushed down at most H pages, we can bulk insert N items in $O(NH/B)$ I/Os in the worst case. When the stream ends or when we have at least N items inserted, we can resort to the earlier case and flush all the buffers down.

6 Performance Evaluation

In this section, we study the performance of the proposed algorithms for bulk operations for the class of space-partitioning trees. Both the DBBL and BTBI algorithms are implemented inside the SP-GiST index engine. The implementation is in C++ running SunOS 5.6 (Sparc). Two new internal methods are added to SP-GiST, namely, the *BulkLoad()* and *BulkInsert()*.

6.1 Bulk Loading

In this section, we study the performance of both DBBL and BTBI for bulk loading space-partitioning trees. Figures 6a and 6b compare the number of I/O's from using DBBL, BTBI, and the repeated standard insertion procedure for bulk loading the *Patricia trie* and the k-d tree, respectively. We choose the *Patricia trie* as an example of the space-driven space-partitioning trees and the k-d tree as an example of the data-driven space-partitioning trees. The realization of both the *Patricia trie* and k-d tree inside SP-GiST is given in Table 1 Similar performance is achieved when applying DBBL and BTBI for other space-partitioning trees implemented inside SP-GiST. The repeated insertion procedure is implemented by calling the *Insert* internal method of SP-GiST N consecutive times to insert N data items. The number of bulk loaded items varies from 20K to 100K. For both the *Patricia trie* and the k-d tree, bulk loading algorithms achieve around 1% of the I/O's required by the repeated insertion procedure. Notice that the number of I/O's in Figure 6 is drawn in log scale. Due to the excessive time and I/O's for the standard insertion procedure, for the rest of the experiments, we ignore the repeated insertion procedure and focus on the performance of DBBL and BTBI. Both the *Patricia trie* and k-d tree have similar performance (other space-partitioning tress are no exception, where they result in similar performance). Thus, for the following experiments, we focus on bulk loading/insertion in the *Patricia trie*. However, all the experimental results and analysis are applicable for other space-partitioning trees.

Figure 7 gives a comparison between DBBL and BTBI when bulk loading a *Patricia trie*. The number of bulk loaded items varies from 60K to 300K. DBBL gives better performance than BTBI for all data input sizes. The main reason for the better performance of DBBL is that DBBL forwards data items to their final buffer positions in the in-memory tree. On the other hand, BTBI forwards data items in the tree one level at a time. Thus, BTBI performs more I/O's than DBBL.

Figures 8a and 8b give the effect of increasing the memory size on the performance of DBBL and BTBI, respectively. The experiments are performed for bulk loading a *Patricia trie* with fanout seven. Similar performance is achieved when bulk loading the k-d tree

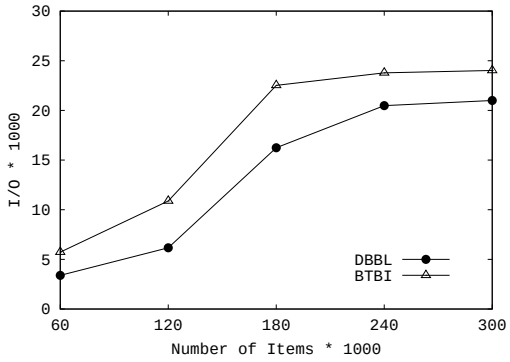
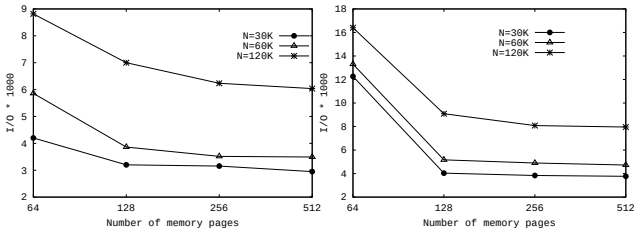


Figure 7. DBBL Vs. BTBI

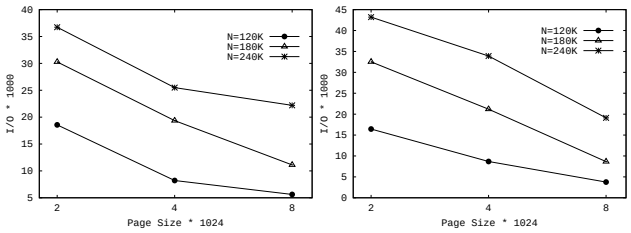


(a) DBBL (b) BTBI

Figure 8. Effect of memory size.

implementation inside SP-GiST. The page size is set to 4K, while the memory size varies from 64 to 512 pages. For each memory size, we run the experiment to bulk load N items, where N is set to 30K, 60K, and 120K data items. For all values of N, initially, the increase of the memory size decreases the number of I/O's for both algorithms. However, after memory size 128 pages, increasing the memory size does not give any preferences in the number of I/O's. This result matches with the analysis performance in Section 5. DBBL has an average case I/O of $O(N/B \log_{M/B} N/B)$, which indicates that the number of I/Os is affected by the memory size and is proportional to $1/\log M$ (which is a hyperbolic curve on log scale). From the Figure, BTBI performs more I/Os when the memory size is small because when $M < B^2$, BTBI may need to access buffers repeatedly during the distribution process.

Figures 9a and 9b give the effect of increasing the page size on the performance of DBBL and BTBI, respectively. The experiments are performed for bulk loading a *Patricia trie* with fanout seven. The page size varies from 2K to 8K. For each page size, we run the experiment to bulk load N items, where N is set to 120K, 180K, and 240K data items. The number of



(a) DBBL (b) BTBI

Figure 9. Effect of page size.

I/O's decreases with the increase in page size. This result matches with the performance analysis in Section 5. Both algorithms are affected by the page size in $O(1/B)$, where B is measured in terms of the number of data items that can fit in the page.

6.2 Bulk Insertion

In this section, we study the performance of the BTBI algorithm for bulk insertion. All experiments in this section are performed for a *Patricia trie* implementation inside SP-GiST (with fanout seven). Similar performance results are obtained when applying BTBI for other space-partitioning trees. In the first experiment (see Figure 10), we compare BTBI with the repeated insertion procedure. The number of the data items N in the initial *Patricia trie* varies from 10K to 60K. The number of items to be bulk inserted is N. The memory size is set to accommodate 400 pages, with the page size set to 4K. BTBI outperforms the repeated insertion procedure for all input data sizes. The main reason is the obvious one; the repeated insertions may read the pages of the tree back and forth many times, while in bulk insertion, every page in the tree is read only once as the data items are forwarded through the tree pages.

Figure 11 illustrates the scalability of BTBI in terms of the number of data items to be bulk inserted. The page size is set to 4K, while the memory size is set to 400 pages. The number of data items N in the initial tree varies from 10K to 60K. BTBI is applied to bulk insert N, $2N$, and $4N$ data items. BTBI is scalable to the number of data items, where increasing the number of inserted data items results in a slight increase in the number of I/O's. The difference in the number of I/O's between N insertions and $2N$ insertions is less than the double. The main reason is that only a few more buffer pages are used in the case of $2N$ insertions.

Figure 12 gives the effect of the page size on the

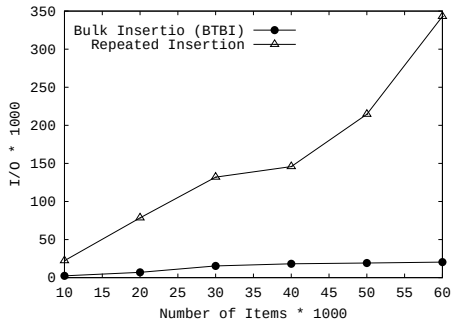


Figure 10. BTBI Vs. Repeated Insertion.

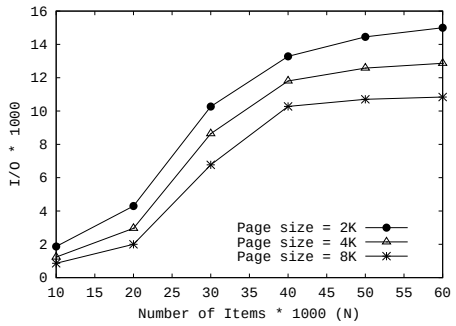


Figure 12. Effect of page size.

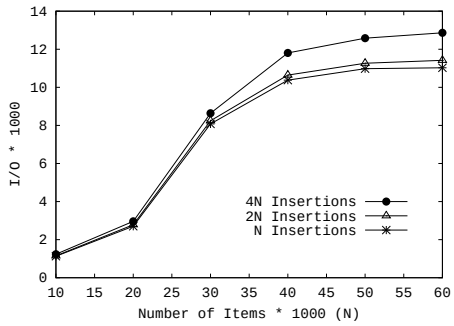


Figure 11. Effect of number of insertions.

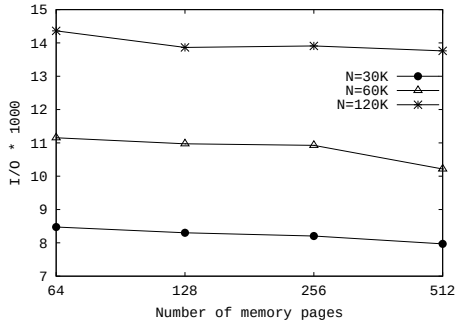


Figure 13. Effect of memory size.

performance of BTBI. The memory size is set to 400 pages. The number of inserted items is $4N$ where N varies from 10K to 60K. We plot three lines that correspond to different page sizes (2K, 4K, and 8K). The number of I/O's required to load the same number of items increases with the decrease in page size.

Figure 13 gives the effect of memory size on the performance of BTBI. Generally, increasing the memory size more than 128 pages does not affect the performance of BTBI. This result is similar to the one obtained in Figure 8 where the I/O performance of BTBI as a function of the memory size M is $O(1/\log M)$.

7 Conclusion

In this paper, we present two extensible bulk loading and bulk insertion algorithms for the class of space-partitioning trees; a class of hierarchical data structures that recursively decompose the space into disjoint partitions. The proposed algorithms are implemented inside SP-GiST (Space-Partitioning Generalized Search Tree); a framework for supporting the class of space-partitioning trees. The main idea of the proposed algorithms is to utilize part of the data items to build an in-memory tree of the target index structure. The remaining data items are partitioned into either in-memory or disk-based buffers. The algorithms work recursively on each buffer. A detailed implementation and realization of the proposed bulk loading and bulk insertion algorithms inside SP-GiST are presented. Analytical study of both algorithms ensure a worst case I/O upper bound of $O(NH/B)$, where N is the number of data items to be bulk loaded/inserted, B is the number of tree nodes that can fit in one disk page, H is the tree height in terms of pages after applying a clustering algorithm. Experimental results show the scalability of both algorithms in terms of the number of data items. The first proposed algorithm outperforms the second in case of bulk loading. However, the second algorithm is general enough to be applicable to bulk insertion as well.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A Framework for Index Bulk Loading and Dynamization. In *Intl. Colloq. on Automata, Lang. and Prog., ICALP*, pages 115–127, July 2001.

[2] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *SSDBM*, pages 49–58, 2001.

[3] W. G. Aref and I. F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partition-

ing Trees. *Journal of Intelligent Information Systems, JIIS*, 17(2–3):215–240, dec 2001.

[4] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, Denmark, aug 1996.

[5] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient Bulk Operations on Dynamic R-Trees. *Algorithmica*, 33(1):104–128, 2002.

[6] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM, JACM*, 45(6):891–923, 1998.

[7] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–89, 1972.

[8] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM, CACM*, 18(9):509–517, 1975.

[9] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations. In *EDBT*, pages 216–230, Mar. 1998.

[10] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, Sept. 1996.

[11] C. Böhm and H.-P. Kriegel. Efficient Bulk Loading of Large High-Dimensional Indexes. In *Proc. of the Intl. Conf. on Data Warehousing and Knowledge Discovery, DeWak*, pages 251–260, 1999.

[12] L. Chen, R. Choubey, and E. A. Rundensteiner. Merging R-Trees: Efficient Strategies for Local Bulk Insertion. *GeoInformatica*, 6(1):7–34, 2002.

[13] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A Generalized R-Tree Bulk-Insertion Strategy. In *SSD*, pages 91–108, July 1999.

[14] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proc. of Australasian Database Conf.*, Feb. 1998.

[15] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*, pages 426–435, Aug. 1997.

[16] J. V. den Bercken and B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *VLDB*, pages 461–470, Sept. 2001.

[17] J. V. den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB*, pages 406–415, Aug. 1997.

[18] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. In *VLDB*, pages 342–353, Sept. 1996.

[19] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: combining the advantages of k-d trees and octrees. In *Proc. of the ACM Symp. on Disc. Algo., SODA*, pages 300–309, 1999.

[20] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval of composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[21] E. Fredkin. Trie memory. *Communications of the ACM, CACM*, 3(9):490–499, 1960.

[22] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software, TOMS*, 3(3):209–226, 1977.

[23] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comp. Surveys*, 30(2):170–231, 1998.

[24] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Indexing. In *SIGMOD*, pages 47–57, Boston, MA, June 1984.

[25] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, pages 562–573, Sept. 1995.

[26] G. R. Hjaltason and H. Samet. Improved Bulk-Loading Algorithms for Quadtrees. In *GIS*, 1999.

[27] G. R. Hjaltason, H. Samet, and Y. J. Sussmann. Speeding up Bulk-Loading of Quadtrees. In *GIS*, 1997.

[28] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1(2):145–153, Apr. 1979.

[29] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *VLDB*, Sept. 1994.

[30] I. Kamel, M. Khalil, and V. Kouramajian. Bulk Insertion in Dynamic R-Trees. In *Proc. of the Intl. Symp. on Spatial Data Handling, SDH*, pages 31–42, 1996.

[31] G. Kedem. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proc. of the Design Automation Conference, DAC*, pages 352–357, 1982.

[32] T. M. Klein, K. J. Parzygnat, and A. L. Tharp. Optimal B-tree packing. *Information Systems*, 16(2):239–243, 1991.

[33] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, Apr. 1997.

[34] D. B. Lomet. Grow and Post Index Trees: Roles, Techniques and Future Potential. In *SSD*, Aug. 1991.

[35] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Coded in Alphanumeric. *Journal of the ACM, JACM*, 15(4):514–534, 1968.

[36] R. C. Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. In *Proceedings of the ACM SIGGRAPH*, pages 197–206, Aug. 1986.

[37] S. R. Ning An, Kothuri Venkata Ravi Kanth. Improving Performance with Bulk-Inserts in Oracle R-Trees. In *VLDB*, Sept. 2003.

[38] J. A. Orenstein. Multidimensional Tries Used for Associative Searching. *Information Processing Letters*, 14(4):150–157, 1982.

[39] J. A. Orenstein and T. Merrett. A Class of Data Structures for Associative Searching. In *PODS*, 1984.

[40] M. H. Overmars and J. van Leeuwen. Dynamic Multi-Dimensional Data Structures Based on Quad- and K - D Trees. *Acta Informatica*, 17(3):267–285, 1982.

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

[42] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In *SIGMOD*, pages 17–31, May 1985.

[43] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[44] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics, TOG*, 4(3):182–222, 1985.

[45] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.