Improved Bulk-Loading Algorithms for Quadtrees*

Gísli R. Hjaltason and Hanan Samet
Computer Science Department, Center for Automation Research, Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742
{grh,hjs}@cs.umd.edu

Abstract

Spatial indexes, such as the PMR quadtree, are important in spatial databases for efficient execution of queries involving spatial constraints, especially when the queries involve spatial joins. In this paper we report recent improvements in bulk-loading PMR quadtrees, which index arbitrary spatial objects, and present a new algorithm for bulk-loading PR quadtrees, which index point data. Our algorithms assume that the quadtree is implemented using a linear quadtree, a disk-resident representation that stores objects contained in the leaf nodes of the quadtree in a linear index (e.g., a B-tree) ordered on the basis of a space-filling curve. We show with experiments that our algorithms yield significant performance improvements for bulk-loading quadtrees.

1 Introduction

Spatial indexes are designed to facilitate spatial database operations that involve retrieval on the basis of the values of spatial attributes. A central problem in the implementation of spatial indexes is the time needed to build them in the sense that the index is only worthwhile if the time to execute the operation without the index is slower than the sum of the time to build the index and the time to execute the operation. Of course, if the database is static, then we can afford to spend more time on building the index as the index creation time can be amortized over the number of queries made on the indexed data. However, we are interested in the case that the database is dynamic. This situation arises when the output of an operation, such as a spatial join, results in the creation of new data which can be used as input to subsequent operations. In this case, when we evaluate the efficiency or appropriateness of a particular spatial index, we must also take into account the time needed to build an index on the result of the operation.

*The support of the National Science Foundation under Grant IRI-97-12715 is gratefully acknowledged.

As an example of a spatial join where the output is useful, suppose that given a road relation and a river relation we want to find all locations where a road and river meet (i.e., the actual physical locations of bridges and tunnels rather than just the object pairs of roads and rivers that have a point in common which is the join condition). The locations serve as input to subsequent spatial operations (i.e., a cascaded spatial join as would be common in a spatial spreadsheet [8]). Therefore, we also need to construct a map for the output, which means that we need to construct a spatial index. In other words, the time to build the spatial index plays an important role in the overall performance of the index in addition to the time required to perform the spatial join itself whose output is not always required to be spatial.

In this paper we examine the efficiency of building the spatial index by use of bulk-loading which is the process of building the index for a set of objects without any intervening queries. In recent years, numerous bulk-loading algorithms for spatial indexes have been introduced. Most of the attention has been focused on the R-tree, and related structures (e.g., [1, 2]). Here we improve upon an algorithm previously developed by us for bulk-loading a disk-based PMR quadtree [7]. Although our presentation and experiments are in terms of the PMR quadtree, our results hold for any variant of the quadtree. In fact, our approach can be adapted to speed up the construction of many other spatial data structures based on regular partitioning, such as the buddy-tree [12] and the BANG file [4]. In our bulk-loading algorithm for the PR quadtree, the fact that point data has no spatial extent enables us to build the leaf nodes of the quadtree in a bottom-up manner (loosely speaking). This is in contrast to the PMR quadtree bulk-loading algorithm, which must proceed in a top-down manner (see [6] for more details including pseudo code listings and a technique for extending the algorithms for bulk-insertions).

The rest of this paper is organized as follows. Section 2 describes the PR and PMR quadtrees and their implementation. Section 3 presents our bulk-loading approach for the PMR quadtree. Section 4 describes a bulk-loading method for the PR quadtree. Section 5 discusses the results of our experiments, while concluding remarks are drawn in Section 6.

2 Quadtrees

By the term *quadtree* [11] we mean a spatial data structure based on a disjoint regular partitioning of space. Each quadtree block (also referred to as a *cell*) covers a portion of space that forms a hypercube in d-dimensions, usually with a side length that is a power of 2. Quadtree blocks may be further divided into 2^d sub-blocks of equal size.

The *PMR quadtree* [9] is a quadtree variant intended for storing objects of arbitrary spatial type. Figure 1 shows a PMR quadtree for a collection of line segments. Since the PMR quadtree gives rise to a disjoint decomposition of space, and objects are stored only in leaf blocks, this implies that non-point objects may be stored in more than one leaf block. Thus, the PMR quadtree would be classified as applying *clipping*, as we can view an object as being *clipped* to the region of each intersecting leaf block. The part of an object that intersects a leaf block that contains it is often referred to as a *q-object*; for line segments, we usually talk of *q-edges*. For example, segment a in Figure 1 is split into three q-edges as it intersects three leaf nodes.

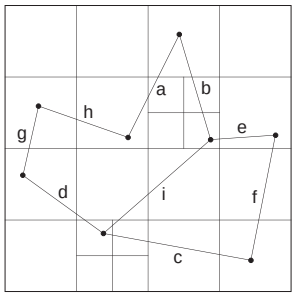

Figure 1: A PMR quadtree for line segments with a splitting threshold of 2, where the line segments have been inserted in alphabetical order.

A key aspect of the PMR quadtree is its splitting rule. If the insertion of an object o causes the number of objects in a leaf block b to exceed the *splitting threshold* T and b is not at the maximum level, then b is split and the objects in b (including o) are inserted into the newly created blocks that they intersect. The sub-blocks are not split further at this time, however, even if they contain more than T objects.

Quadtrees can be implemented in many different ways. One method, inspired by viewing them as trees, is to implement each block as a record, where nonleaf blocks store 2^d pointers to child block records, and leaf blocks store a list of objects. However, this pointer-based approach is ill-suited for implementing disk-based structures. A general methodology for solving this problem is to represent only the leaf blocks in the quadtree. The location and size of each leaf block is encoded in some manner, and the result is used as a key into an auxiliary disk-based data structure. This approach is termed a *linear quadtree* [5].

The linear quadtree is used in a number of spatial database systems including SAND, a prototype spatial database system built by our group. In SAND, the implementation of the PMR quadtree is based on a general linear quadtree implementation called the *Morton Block Index* (abbreviated *MBI*). The size of the space covered by an MBI has side length of 2^w with 0 as the origin for each dimension, and the minimum side length of a quadtree block that can be represented is 1. The MBI encodes quadtree blocks using a pair of numbers, termed a *Morton block value*. The first number is the *Morton code* of the lower-left corner of the quadtree block, while the second number is the side length of the block (stored in $\log_2$ form). The Morton code of a point is constructed by bit-interleaving its coordinate values.

The MBI uses a B-tree to organize the quadtree contents, with Morton block values serving as keys. When comparing two Morton block values, we employ lexicographic ordering on the Morton code and the side length. When only representing quadtree leaf nodes in the MBI, which is the case for most quadtree variants, only comparing the Morton code value is sufficient, as the MBI will contain at most one block size for any given Morton code value. For a quadtree leaf node with k objects, the corresponding Morton block value is represented k times in the B-tree, once for each object. In the B-tree, we maintain a buffer of recently used B-tree nodes, and employ an LRU (least recently used) replacement policy to make space for a new B-tree node.

3 Bulk-Loading PMR Quadtrees

The algorithm in [7] for bulk-loading PMR quadtrees was based on trying to fill up memory with as much of the quadtree as possible before *flushing* some of its nodes into the disk-based index. The key idea was to sort the input data in such a way that the portions that were written out to the disk would not be inserted into again. This strategy was shown to achieve dramatic speedup compared to the regular quadtree insertion algorithm. Nevertheless, it had two drawbacks. First, the way in which it chose leaf nodes to flush, and thereby result in their insertion in the MBI, did not always result in their being flushed in sorted order (based on Morton block values) although they were almost sorted. The problem was that since we could not guarantee that the nodes were flushed in sorted order, we could not use packing methods to insure that the MBI B-tree nodes were completely full. Thus it was often the case that an MBI B-tree node would overflow and hence be split thereby resulting in two nodes that are 50% full yet these two nodes might never be inserted into again due to the sorted order. Thus storage utilization in the B-tree was very poor. Second, the algorithm relied on a parameter termed "flushing quotient" to guide flushing. Unfortunately, although we found that the algorithm performed well for a range of parameter values, it was unclear how to choose the optimal value or how robust the algorithm was for any given value.

3.1 Improved Flushing Algorithm

To address these drawbacks, we developed a new flushing algorithm. In the new algorithm, the objects are always inserted in Z-order based on the lower-left corner, i.e., the one closest to the origin, of their bounding rectangle (as opposed to the centroid in [7]). If available memory has been exhausted (i.e., the pointer-based quadtree occupies too much memory) upon inserting a new object o, then the algorithm uses o to guide the flushing of nodes from memory rather than making use of the flushing quotient. This is illustrated in Figure 2, where all quadtree blocks in the striped region can be flushed. The advantage of our new algorithm is that it inserts leaf nodes in Z-order into the MBI, thus leading to items being inserted in strict key order into the corresponding B-tree. This can be exploited to pack the B-tree nodes to capacity, e.g., with the algorithm of [10]. Thus, we can achieve nearly 100% storage utilization in the B-tree.

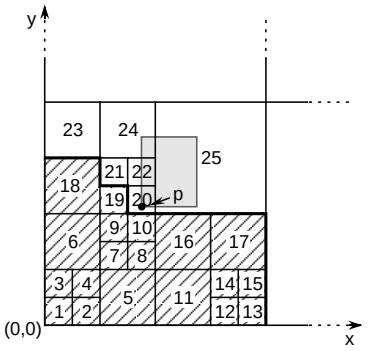

Figure 2: A portion of a hypothetical quadtree, where the leaf nodes are labeled in Z-order. The shaded rectangle is the bounding rectangle of the next object to insert.

Another advantage of flushing the leaf nodes in Z-order is that this makes it very efficient to bulk-insert into an existing PMR quadtree. Essentially, it involves merging the ordered stream of leaf nodes with the leaf nodes in an existing PMR quadtree (which are stored in Z-order).

The flushing algorithm outlined above may fail to free any memory. This occurs if a large number of objects intersects the boundary between flushed and unflushed leaf nodes (e.g., the boundary of of the striped region in Figure 2). We have developed a fall-back method that is invoked when this situation occurs, termed "reinsertion freeing". This involves the removal of a selected set of leaf nodes from the pointer-based quadtree, and scheduling the objects contained in them for re-insertion at a later stage. One way of implementing this is to make the sorting phase and the bulk-load phase operate in tandem, in which case the re-inserted objects are sent back to the sorting phase with a new sort key.

3.2 Improved Insertion Algorithm

Like insertion algorithms for most hierarchical data structures, the PMR quadtree insertion algorithm is defined with a top-down traversal of the quadtree. Thus, the CPU cost for inserting an object is roughly proportional to the depth of the leaf nodes intersecting it. The single largest contributor to the CPU cost of the algorithm (besides the cost of updating the B-tree in the MBI implementation) is the intersection test performed during the top-down traversal. When inserting an object, the number of intersection tests is bounded from above by $2^d \cdot D_{\max} \cdot q$, where $D_{\max}$ is the maximum depth of a leaf node and q is the number of leaf nodes intersected by the object (recall that each nonleaf node has 2^d children). However, the average is typically more like $2^d \cdot D_{\text{ave}}$, where D_{ave} is the average leaf node depth, which is on the order of $\log_{2^d} N$ if the data distribution is not too skewed. Another significant contributor to CPU cost in the MBI implementation is the computation of child blocks (in a pointer-based quadtree, this cost can be avoided since the Morton block values or some other representation for the quadtree regions can be stored in the nodes). The number of these computations is similar to the number of intersection tests. Thus, they contribute considerably to the CPU cost, especially if this computation is not highly optimized.

The number of intersection tests, as well as the number of Morton code computations, can be dramatically reduced by exploiting the structure of the quadtree. The key insight is that based only on the geometry of an object, we can compute the quadtree block that minimally bounds the object. This is illustrated in Figure 3a, where we indicate potential quadtree partition boundaries with broken lines. We can look up the Morton block value of this block in the B-tree of the MBI, which will locate a quadtree leaf block containing the object, if any exists. Two cases can arise: the minimally enclosing quadtree block can be inside (or coincide with) an existing leaf node (e.g., Figure 3b), or there may be more than one leaf node contained in the minimal enclosing quadtree block (e.g., Figure 3c). Thus, in the improved PMR quadtree insertion algorithm, instead of starting the top-down traversal at the root, we start at the smallest existing node enclosing the minimal enclosing quadtree block of the inserted object. The same technique can also be used to speed up PMR quadtree node splits.

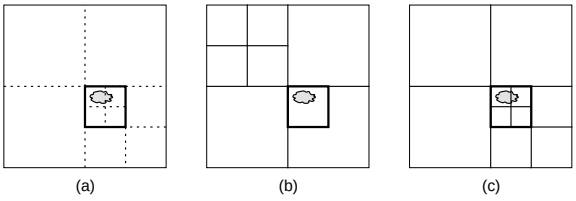

Figure 3: (a) Computation of the minimum bounding block for an object, denoted by heavy lines. Broken lines indicate potential quadtree block boundaries. The minimum bounding block can (b) be enclosed by a leaf node or (c) coincide with a nonleaf node.

The reduction in the number of intersection tests performed by the improved insertion algorithm depends on D'_{ave}, the

average depth of the quadtree nodes (leaf or nonleaf) in the final quadtree that minimally enclose each object. For example, in Figure 3b, the object is minimally enclosed by a leaf node at depth 1 (i.e., the leaf node is a child of the root), whereas in Figure 3c, the object is minimally enclosed by a nonleaf node at depth 2. For an object o minimally enclosed by a node n' at depth D', the original PMR quadtree insertion algorithm must perform $2^d D'$ intersection tests to determine that o is contained in n'. In contrast, our improved algorithm avoids all of these intersection tests, and thus achieves an average reduction of $2^d D'_{\rm ave}$ per object in the number of intersection tests. If o is contained in a leaf node n at depth D, the number of intersection tests performed is at least $2^d(D-D')$, since all child nodes of the nonleaf nodes on the path from n' to n must be tested for intersection with o (e.g., in Figure 3c, the leaf nodes containing o are one level down from n', so only $2^2 = 4$ intersection tests are needed). Hence, the number of intersection tests performed by the improved algorithm on the average per object can be expected to be approximately $p(D_{\rm ave} - D'_{\rm ave})$, where $D_{\rm ave}$ is the average depth of leaf nodes, $2^d \leq p \leq 2^d q$, and q is the average number of q-objects per object. If the objects are very small compared to the size of the data space, $D'_{\rm ave}$ will be nearly as high as $D_{\rm ave}$, so the number of intersection tests will be small. In the extreme case of point objects, no intersection tests are needed and $D_{\rm ave} \approx D'_{\rm ave}$[1]. Experiments [6] on typical line maps showed that our improved algorithm reduces the number of intersection tests by a factor of 3 to 5.

4 Bulk-Loading PR Quadtrees

The bulk-loading method for quadtrees described in Section 3 can be used to bulk-load a PMR quadtree for any type of spatial objects. However, it is possible to do better for point data if we use the PR quadtree [11] (or, more accurately, the bucket PR quadtree) instead of the PMR quadtree. In the PR quadtree (see Section 2), a fixed bucket capacity is established for the leaf nodes instead of a splitting threshold. The method we describe is related to the bulk-loading method for PK-trees described in [13]. Our description is in terms of a PR quadtree stored in an MBI (see Section 2), but can easily be adapted to any other representation. Thus, the quadtree blocks are represented with Morton block values.

When bulk-loading the PR quadtree, we assume that the data is sorted in Morton code order prior to being inserted, as we do in our PMR quadtree bulk-loading method. Rather than first building a pointer-based quadtree in main memory, however, we can directly construct the leaf blocks of the quadtree. Briefly, the algorithm works by adding points, one by one, to a list of candidates for the current leaf node, expanding the node's region as needed (see Figure 4). If adding a new point causes overflow (i.e., more than c points, where c is the bucket capacity) or causes the node's region to intersect a previously created node, then we construct a new leaf node in the MBI with the largest possible subset of the candidates (see Figure 5). For more details, see [6].

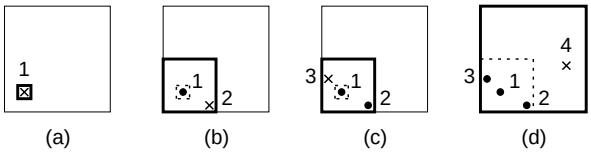

Figure 4: Example of insertions into candidate list, of points 1-4 (in order), demonstrating expansion of the current leaf node region (shown with heavy lines). The square with broken lines denotes the current leaf node region prior to its last expansion.

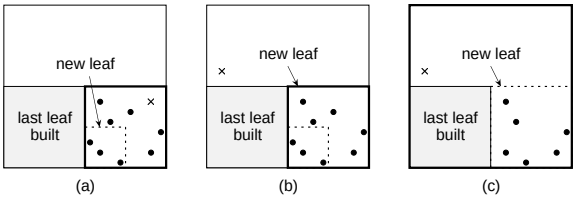

Figure 5: Conditions for constructing a new leaf node where the most recently inserted point is denoted by x assuming a bucket capacity of 8: (a) candidate list overflows and the new point is in the current leaf node region, (b) candidate list overflows and the new point is not in the current leaf node region, and (c) expansion of the current leaf node region causes overlap of the leaf node that was last built.

5 Empirical Results

5.1 Experimental Setup

We implemented the algorithms presented in Sections 3 and 4 in C++ within an existing PMR quadtree and PR quadtree testbed. The source code was compiled with the GNU C++ compiler with full optimization (–O3) and the experiments were conducted on a Sun Ultra 1 Model 170E workstation, rated at 6.17 SPECint95 and 11.80 SPECfp95 with 64MB of memory. In order to better control the run-time parameters, we used a raw disk partition. This ensures that the execution times reflect the true cost of I/O, which would otherwise be partially obscured by the file caching mechanism of the operating system. B-tree node size was set to 4KB, while node capacity varied between 50 and almost 300, depending on the experiment. The maximum depth of the quadtree was set to 16 and the splitting threshold to 8.

The sizes of the data sets we used were perhaps modest compared to some modern applications. However, we compensated for this by using a modest amount of buffering. In

[1] $D_{\rm ave}$ and $D'_{\rm ave}$ are typically not exactly equal for points, since $D_{\rm ave}$ is an average over leaf nodes while $D'_{\rm ave}$ is an average over objects. Alternative, and perhaps more accurate, definitions of $D_{\rm ave}$ that make it equal to $D'_{\rm ave}$ for points are as follows: 1) over all q-objects, the average depth of the leaf node containing them, or 2) over all objects, the average depth of the smallest leaf node intersecting them.

our PMR quadtree bulk-loading algorithm, the space occupied by the pointer-based quadtree was limited to 128K. This proved more than adequate and a larger buffer did not improve performance. A buffer of 512K bytes was allocated to the sorting process. Interestingly, a smaller a buffer size of 256K increased running time only slightly (typically less than 3% of the total time). Only one B-tree node at each level had to be buffered when using our bulk-loading algorithms, due to the efficient B-tree packing algorithm that we used. However, for dynamic insertions, we used a B-tree buffer of 1MB (i.e., for 256 nodes).

In reporting the results of the experiments, we use execution time. This takes into account the cost of reading the data, sorting it, establishing the quadtree structure, and writing out the resulting B-tree. The reason for using execution time, rather than such measures as number of comparisons or I/O operations, is that no other measure adequately captures the overall cost of the loading operations. For each experiment, we averaged the results of a number of runs (usually 10), repeating until achieving consistent results.

In our experiments, we used both non-point data and point data. The point data consisted of two-dimensional line segment data, both real-world and synthetic. The real-world data consists of three data sets from the TIGER/Line File [3]. The first two contain all line segment data for Washington, DC and Prince George's County, MD, abbreviated below as "DC" (19,185 line segments) and "PG" (59,551 line segments). The third contains roads in the entire Washington, DC metro area, abbreviated "Roads" (200,482 line segments). We also used three synthetic line segment data sets, containing 64K, 128K, and 260K non-intersecting line segments. The point data sets that we used were synthetic, consisting of 100K points each, in dimensions ranging from 2 to 8. The sets of points form 10 normally-distributed clusters, whose centers are uniformly distributed in the space.

5.2 Findings

Figure 6 shows the execution time for building PMR quadtrees using dynamic insertions and with the PMR quadtree bulk-loading algorithm (using our improved PMR quadtree insertion algorithm for both). The execution times are adjusted for map size, and reflect the average cost per 10,000 inserted line segments. The bulk-loading algorithm achieves a speedup ranging from a factor of 4 to 12 compared to dynamic insertions, i.e., when line segments are inserted one by one into the disk-based PMR quadtree index. Much of this speedup is due to reduced CPU cost (typically by a factor of at least 5). Moreover, as the size of the data sets increases, the effectiveness of B-tree buffering in the dynamic insertion algorithm is reduced, and most of the excess execution time becomes due to resultant disk I/O. On the other hand, as can be seen in the figure, the insertion cost per line segment for our bulk-loading algorithm grows very slowly with the size of the data set.

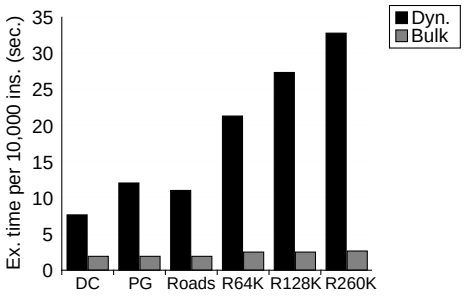

Figure 6: Execution time per 10,000 line segments for building quadtrees for the six data sets.

Figure 7 shows the speedup in execution time resulting from the reduction in the number of intersection tests when using the improved PMR quadtree insertion algorithm when bulk-loading PMR quadtrees for the line segment data sets. The speedup is considerable, ranging from 30% up to nearly 50%. The speedup in CPU time is about twice that shown in the figure, since performing I/Os takes about half the execution time when not using our improved insertion algorithm (recall that our technique does not affect I/O cost). Figure 8 shows the corresponding speedup when bulk-loading PMR quadtrees for point data sets of varying dimensionality. For the two-dimensional data set, the speedup is about 50%. More importantly, the speedup grows as the dimensionality increases, and reaches a factor of nearly 8 for the eight-dimensional point data set.

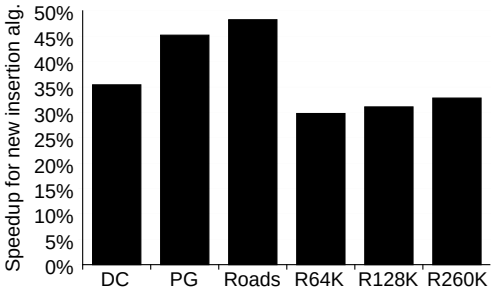

Figure 7: Speedup in terms of execution time resulting from the reduction in the number of intersection tests when using the improved PMR quadtree insertion algorithm for bulk-loading PMR quadtrees for line segment data.

Figure 9 compares the execution time when bulk-loading PMR quadtrees for the point data (with our improved PMR quadtree insertion technique) with that for bulk-loading a PR quadtree with the algorithm presented in Section 4. The execution time appears to grow linearly with the dimension for both bulk-loading algorithms. This is to be expected, since the size of the point data as well as the time to compute geometric operations grows linearly with the dimension. The PR quadtree bulk-loading algorithm is slightly faster for all dimensions, but the difference between the two techniques

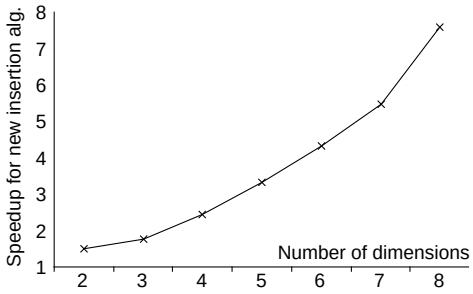

Figure 8: Speedup in terms of execution time resulting from the reduction in the number of intersection tests when using the improved PMR quadtree insertion algorithm for bulk-loading PMR quadtrees for point data of varying dimensionality.

gradually decreases as the number of dimensions increases. This difference corresponds to the overhead (in terms of execution time) in the PMR quadtree bulk-loading algorithm due to the use of the pointer-based quadtree and the associated flushing process. From the figure we see that the overhead is minor. However, the relative parity of the two bulk-loading algorithms is only achieved when the improved PMR quadtree insertion algorithm is used. Without it, the execution time for the PMR quadtree bulk-loading algorithm grows exponentially with the dimension, as Figure 8 indicates.

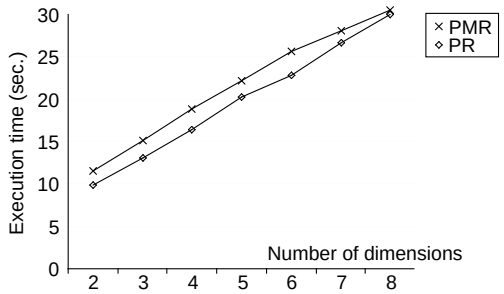

Figure 9: Execution time for bulk-loading PMR quadtrees and PR quadtrees for point data sets of varying dimensionality.

6 Concluding Remarks

In this paper we described improvements to an existing bulk-loading algorithm for the PMR-quadtree, which is capable of indexing arbitrary spatial data. The improvements result in a more robust algorithm, higher storage utilization, and allow the efficient implementation of bulk-insertion. In addition, we presented a new technique for speeding up insertions into PMR quadtree. It is applicable both to our improved bulk-loading algorithm as well as to the traditional dynamic insertion algorithm for the PMR quadtree. This new technique dramatically reduces the number of intersection tests necessary for locating leaf nodes that contain an object. We also presented a new algorithm for bulk-loading the PR quadtree, a quadtree variant for storing point data. Our experiments confirmed the utility of our techniques.

Future work includes investigating whether our buffering strategies for bulk-loading may be used to speed up dynamic insertions and queries. Also, we wish to identify situations where a query engine can exploit fast spatial index construction in order to speed spatial operations on intermediate query results or for un-indexed spatial relations. This is particularly important for complex operations such as spatial joins.

References

[1] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proc. 1st ALANEX Workshop*, Baltimore, MD, Jan. 1999.

[2] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. 23rd VLDB Conf.*, pp. 406–415, Athens, Greece, Aug. 1997.

[3] Bureau of the Census. *Tiger/Line precensus files*. Washington, DC, 1989.

[4] M. Freeston. The BANG file: a new kind of grid file. In *Proc. SIGMOD Conf.*, pp. 260–269, San Francisco, CA, May 1987.

[5] I. Gargantini. An effective way to represent quadtrees. *CACM*, 25(12):905–910, December 1982.

[6] G. R. Hjaltason and H. Samet. Speeding up construction of quadtrees for spatial indexing. Comp. Sci. Dep. TR-4033, Univ. of Maryland, College Park, MD, July 1999.

[7] G. R. Hjaltason, H. Samet, and Y. Sussmann. Speeding up bulk-loading of quadtrees. In *Proc. 5th ACM-GIS Workshop*, pp. 50–53, Las Vegas, NV, Nov. 1997.

[8] G. Iwerks and H. Samet. The spatial spreadsheet. In *Proc. 3rd Conf. on Visual Info. Sys. (VISUAL99)*, Amsterdam, The Netherlands, June 1999.

[9] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proc. SIGMOD Conf.*, pp. 270–277, San Francisco, CA, May 1987.

[10] A. L. Rosenberg and L. Snyder. Time- and space-optimality in B-trees. *ACM TODS*, 6(1):174–193, 1981.

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

[12] B. Seeger and H. P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proc. 16th VLDB Conf.*, pp. 590–601, Brisbane, Australia, Aug. 1990.

[13] J. Yang, W. Wang, and R. Muntz. Yet another spatial indexing structure. Comp. Sci. Dept. TR 970040, Univ. of California, Los Angeles, CA, 1997.