Speeding up Bulk-Loading of Quadtrees

Gísli R. Hjaltason*
Computer Science Department
Center for Automation Research
Institute for Advanced
Computer Studies
University of Maryland
College Park, Maryland 20742
grh@cs.umd.edu

Hanan Samet*
Computer Science Department
Center for Automation Research
Institute for Advanced
Computer Studies
University of Maryland
College Park, Maryland 20742
hjs@cs.umd.edu

Yoram J. Sussmann†
Computer Science Department
University of Maryland
College Park, Maryland 20742
yoram@cs.umd.edu

July 29, 1997

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. We investigate the issue of speeding up building PMR quadtrees for a set of objects and develop two approaches to achieve this goal. In an empirical study, we find that the better method of the two offers significant improvements in execution time, and present evidence of the usefulness of spatial indexing for executing spatial join queries.

1 Introduction

Traditional database systems employ indexes on alphanumeric data, usually based on the B-tree, to facilitate efficient query handling. Typically, the database system allows the users to designate which attributes (data fields) need to be indexed. However, advanced query optimizers also have the ability to create indexes on unindexed data or temporary results (i.e., results from a part of the query) as needed. In order for this to be worthwhile, the index creation process must not be too time-consuming, as otherwise the operation could be executed more efficiently without an index. In other words, the index may not be particularly useful if the execution time of the operation without an index is several times faster than the total time to execute it when the time to build the index is included.

Of course, indexes are often used even though the speed of constructing them is slow when the resulting indexed data is queried many times. In this case, the time to build the index is amortized over the number of queries made on the indexed data before a new index needs to be constructed (e.g., on account of updates). This is the case when the database is static.

In the research reported here, we focus on the situation where the database is dynamic. This is an often-neglected issue in the design of spatial databases. The problem is that most often the index is chosen on the basis of the speed with which queries can be performed and on the amount of storage that is required. The queries usually involve retrieval rather than the creation of new data. This emphasis on retrieval efficiency may lead to a wrong choice of an index when the operations are not limited to retrieval. This is especially evident for a query such as the spatial join. As an example of this query, suppose that given a road relation and a river relation, we want to find all locations where a road and river meet (i.e., locations of bridges and tunnels). This can be achieved by computing a spatial join of the two relations which is realized by joining the two relations. The join condition is one that results in extracting all tuples whose spatial attribute values have at least one point in common.

The spatial join is an interesting operation because its output has both a relational and a spatial component. In practical terms, for example in the case of line segments, we don't always want to just report the object pairs (i.e., lines or the names of the rivers and roads in our example) that intersect. In particular, we want to report their locations as well so that they can serve as input to subsequent spatial operations (i.e., a cascaded spatial join as would be common in a spatial spreadsheet). 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. Interestingly, most traditional studies of the effect of spatial indexing on the efficiency of the spatial join (e.g., [**?**, **?**, **?**, **?**, **?**, **?**]) only focused on the relational component of the output, while very few (e.g., [**?**]) included a spatial component in the output.

In this paper we examine the efficiency of building the spatial index. In particular, we focus on the PMR quadtree spatial index [**?**]. The PMR quadtree is of particular interest as we showed in an earlier study [**?**] that the PMR quadtree performs quite well for a spatial join in contrast to other spatial data structures such as the R-tree [**?**] (including variants such as the R*-tree [**?**]) and the R$^+$-tree [**?**]. This was especially true when the execution time of the spatial join included the time needed to build a spatial index for the result of the operation. In that study, the running time of the various R-tree construction algorithms could not be significantly reduced without degrading their performance in responding to the relational component of the query. The reason for this behavior is the fact that the speed of the construction of the R-tree is inversely proportional to the quality of the resulting space decomposition which also has the same relationship to the space requirements[1]. Improvement of the performance of the R$^+$-tree requires taking into account similar consider-

*The support of the National Science Foundation under Grant IRI-92-16970 and the Department of Energy under Contract DEFG0295ER25237 is gratefully acknowledged.

†The support of the National Science Foundation under Grant CCR-9501355 is gratefully acknowledged.

[1] Note that indexes such as the packed R-tree [**?**] and Hilbert-packed R-tree [**?**] (and the related Hilbert R-tree [**?**], which is really a B-tree) were not taken into account in this study as they are not constructed with the goal of minimizing coverage or overlap which are the key principles behind the R-tree.

ations. In contrast, in that study, no attempt was made to enhance the performance of the PMR quadtree construction algorithms.

Improving the performance of building a quadtree spatial index is of interest to us for a number of additional reasons. First of all, the PMR quadtree is used as the spatial index for the spatial attributes in a spatial database system built by us called SAND (Spatial and Non-Spatial Data) [**?**, **?**, **?**]. SAND employs a data model inspired by the relational algebra. The basic storage unit is an attribute, which may be non-spatial (e.g., integer or character string) or spatial (e.g., points, lines, polygons, etc.). Attributes are collected into relations, and relation data is stored as tuples in tables, each of which is identified by a *tuple ID*. SAND uses indexing to facilitate speedy access to tuples based on both spatial and non-spatial attribute values. Second, quadtree indexes have started to appear in commercial database systems such as the Spatial Data Option (SDO) from the Oracle Corporation [**?**]. Thus speeding their construction has an appeal beyond our SAND prototype.

One problem with using the PMR quadtree as an index is that despite the results of our previous comparative study [**?**], we still find that building a PMR quadtree is a time-consuming process. Our goal is to speed up the process of building a PMR quadtree from a set of objects in order to make the PMR quadtree more useful for spatial indexing in SAND. In particular, this would make it possible for the query optimizer to build indexes on the fly as the need arises. This is especially important for queries that involve spatial joins as we saw in the example above.

Strategies for efficient building have been developed for other spatial data structures, most notably for the R-tree. It should be stressed that these data structures are variants of the R-tree since their build rules are dramatically different from that of the traditional R-tree as they use a different insertion method and most assume a static database. These strategies fall into one of two groups: bottom-up and top-down. The packed R-tree [**?**] and Hilbert-packed R-tree [**?**] are examples of bottom-up methods, where the leaves of the R-tree are built first using proximity criteria to aggregate objects, and the non-leaf nodes are built on top, with bounding rectangles computed for the nodes[2]. The S-tree [**?**] (a variant of R-trees that is not strictly balanced) and the VAMSplit R-tree [**?**] are examples of top-down methods. Without going too much into detail, these top-down methods operate on a whole set of objects, subdividing the set recursively in some manner.

None of these strategies work for the PMR quadtree. In particular, a bottom-up method does not work as the leaf nodes are not all on the same level as in the R-tree (among other reasons). A top-down method is inapplicable because the splitting rules that govern the PMR quadtree require that the objects be inserted one-by-one. Notice that even if the splitting rules were relaxed, the fact the PMR quadtree is based on a disjoint decomposition of space makes it unwieldy to subdivide a set of objects based on their spatial occupancy since each object may have to be included in more than one subgroup.

We use the term *bulk-loading* to characterize the process of building a disk-based spatial index for an entire set of objects without any intervening queries. The approach taken in this paper is based on the idea of trying to fill up memory with as much of the quadtree as possible before writing some of its nodes on disk. Although our presentation and experiments are in terms of the PMR quadtree, our results hold for any variant of the quadtree. The rest of this paper is organized as follows. Section **??** describes the PMR quadtree and its implementation in SAND which serves as the prototype whose construction time is being speeded up. Section **??** presents our approach. Section **??** discusses the results of our experiments, while concluding remarks are drawn in Section **??**.

[2] Recall that these methods do not explicitly seek to minimize overlap or coverage of bounding rectangles. Instead, the hope is that the proximity criteria aggregate objects in a way that causes low overlap and coverage.

2 PMR Quadtrees and their Implementation

By the term *quadtree* [**?**, **?**] we mean a spatial data structure based on a disjoint regular decomposition of space. Each quadtree block (sometimes 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. One way of conceptualizing a quadtree is to think of it as an extended 2^d-ary tree[3]. Another way is to focus on the space decomposition, in which case it can be thought of as being an adaptive grid. Usually, there is a prescribed maximum height of the tree, or equivalently, a minimum size for each quadtree block.

The *PMR quadtree* is a dynamic spatial data structure based on the idea of a quadtree, where objects are stored only in leaf blocks that intersect them. If, upon inserting an object o into a quadtree block b, the number of objects in b exceeds a splitting threshold T and b is not at the maximum level, then b is split into 2^d sub-blocks, and the objects in b (including o) are reinserted into the newly created blocks that they intersect. Note that the sub-blocks are not split further during the insertion of o, even if they contain more than T objects. This aspect of the PMR quadtree gives rise to probabilistic behavior in the sense that the order in which the objects are inserted affects the shape of the resulting tree.

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. The part of an object that intersects a leaf block that contains it is often referred to as a *q-object*.

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 non-leaf blocks store 2^d pointers to 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 doing this 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* [**?**].

The implementation of the PMR quadtree used in the SAND spatial database is based on a general linear quadtree implementation called the *Morton Block Index* (abbreviated as *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 Morton code of the quadtree block along with the side length of the block (stored in $\log_2$ form). The Morton code of a quadtree block is constructed by bit-interleaving the coordinate values of the lower-left corner of the block. Not all possible Morton block values correspond to legal quadtree blocks. For example, for a 2-dimensional quadtree, the only quadtree block that can have a lower-left corner of $(1, 1)$ has a side length of 1. However, each Morton code can correspond to many quadtree blocks, e.g., the point with coordinate values $(0, 0)$ can be the lower-left corner of blocks of any size from 1 through 2^w. Notice that the fact that the range of the Morton codes is from 0 to 2^w for each dimension is not really a limitation, as it is a simple matter to transform coordinates in any other range into the range of a Morton code, and vice versa.

Morton codes provide a mapping from a d-dimensional point to a one-dimensional scalar, the result of which is known as a *space-filling curve*. When the d-dimensional points are ordered on the basis of their corresponding Morton codes, the order is called a *Morton* order [**?**]. It is also known as a Z-order [**?**] since it traces a 'Z' pattern in two dimensions. Many other space-ordering methods ex-

[3] An extended k-ary tree is a tree where each node is either a leaf node or contains k children.

ist, such as the Peano-Hilbert, Cantor-diagonal, and spiral orders. However, of those, only the Morton and Peano-Hilbert orders are useful for ordering quadtree blocks. The advantage of the Morton order is that it is much simpler, thereby making it computationally much less expensive, to convert between a Morton code and its corresponding coordinate values (and vice versa) than between a code based on Peano-Hilbert order and its corresponding coordinate values. In addition, various operations on Morton block values can be implemented through simple bit-manipulation operations on Morton codes. Examples include computing the Morton block values for sub-blocks, for a containing block as well as for the neighboring blocks of a quadtree block.

The MBI uses a B-tree to organize the Morton block values, employing a lexicographic sorting order on the Morton code and side length. Note that this corresponds to a Z-order on the quadtree blocks. 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. The B-tree uses a small amount of buffering of B-tree nodes, storing only the B-tree nodes from the root to the current node being searched, as well as possibly the sibling of the current node (e.g., when splitting and merging).

There are two ways of storing spatial objects in an MBI, and both are supported in our system. The first is to store the entire spatial description of the object, while the second is to store a reference ID for the object, which is actually stored in an auxiliary object table. A hybrid scheme can also be employed. The disadvantage of the first scheme is that it can lead to much wasted storage for non-point objects, as they may be represented more than once in a quadtree that is based on a regular decomposition as is the case here. The disadvantage of the second scheme is that a table lookup is necessary to determine the geometry of an object once it is encountered in a quadtree block. Nevertheless, it is unavoidable to use this scheme if we wish to associate some non-spatial data with each object (e.g., for objects representing cities, we may want to store their names and population). SAND uses the object table approach. Thus, for a spatial object o, only the tuple ID for the tuple that o is a part of is stored in the PMR quadtree.

Another design choice is whether or not to represent empty quadtree leaf blocks in the MBI. Our implementation supports both of these choices. Representing empty quadtree leaf blocks simplifies insertion procedures as well as other operations on the quadtree and makes it possible to check the MBI for consistency, since the entire data space must be represented in the index. However, for large dimensions, this may be very wasteful, since a large number of leaf blocks will tend to be empty.

3 Our Approach

Our implementation of the PMR quadtree as described in Section **??** is very flexible in several respects. The MBI supports any number of dimensions, an underlying space with a side width of up to 2^{32}, and the size of the object (in terms of the number of bytes) stored in the MBI is unlimited. The splitting threshold of the PMR quadtree is also unlimited. Nevertheless, we found its performance to be respectable for dynamic insertions and a wide range of queries. However, for loading a large number of objects simultaneously (i.e., bulk-loading), this flexibility proved to degrade performance. One reason for this inefficiency is that, in addition to the cost of B-tree operations when traversing the tree structure implied by the quadtree, node splits are very costly. This is due to the fact that when a quadtree node is split, references to objects must be deleted from the B-tree, and then reinserted with Morton block value identifiers of the newly created quadtree nodes. The deletions from the B-tree may cause merging of B-tree nodes, and the subsequent reinsertions of the objects with their new Morton block values will then cause splitting of these same nodes. This causes a lot of disk activity.

We developed two approaches in an attempt to speed up bulk-loading. One attacks the problem on the B-tree level, while the other attacks it on the PMR quadtree level. The first approach is to dramatically increase the amount of buffering done by the B-tree (the *B-tree buffering* approach). The second approach is to reduce the number of accesses to the B-tree as much as possible by storing parts of the PMR quadtree in main memory (the *quadtree buffering* approach).

3.1 B-tree Buffering

In the B-tree buffering approach, we use a buffer to store recently used B-tree nodes, and employ an LRU (least recently used) replacement policy to make space for a new node. A node locking mechanism ensures that the nodes on the path from the root to the current node are not replaced. Such extensive buffering was not included in our original implementation because it was found to offer little performance improvement for dynamic insertion as well as many query types. This is due to the fact that for such use, B-tree nodes tend to have been replaced by the time they are needed again, as quadtree blocks are requested in a largely random manner. Also, it is mostly useless for processes that access the whole quadtree in Z-order (such as done by some spatial join algorithm), as they give rise to a sequential scan of the MBI B-tree.

In order to make the most of the B-tree buffering approach, it is best that quadtree nodes be visited in Z-order, since that order corresponds to how they are sorted in the B-tree. By sorting the set of objects to insert in Z-order on their centroid, we will approach that goal (as they will tend to localize insertions within the top-most B-tree nodes, i.e., the ones storing the largest Morton block values). Sorting a set of objects prior to insertion is a small price to pay, as it is usually a much less expensive process than the cost of building the spatial index. It is a common approach for statically built spatial data structures (e.g., Hilbert-packed R-trees [**?**], S-trees [**?**], VAM-Split R-trees [**?**]).

3.2 Quadtree Buffering

In the quadtree buffering approach, we build a pointer-based quadtree in main memory, thereby bypassing the MBI B-tree. Of course, this can only be done as long as the entire quadtree fits in main memory. Once available memory is used up, parts of the pointer-based quadtree are flushed (i.e., written) onto disk (i.e., into the MBI). When all the objects have been inserted into the pointer-based quadtree, the entire tree is inserted into the MBI. In order to maintain compatibility with the MBI-based PMR structure, we use Morton block values to determine the space coverage of quadtree blocks. Note that it is not necessary to store the Morton block values in the nodes of the pointer-based structure (each node corresponds to a quadtree block), as these can be computed during traversals of the tree. However, a careful analysis of execution profiles revealed that a large percentage of the execution time was spent on bit-manipulation operations on Morton block values. Thus, we chose to store the Morton block values in the nodes, even though this increased their storage requirements.

We use a set of heuristics to choose which quadtree blocks (also referred to as nodes) to flush. The goal is to flush quadtree blocks that will not be needed later on, i.e., no subsequently inserted object intersects the block. In general, it is impossible to attain this goal for arbitrary insertion patterns. However, sorting the set of objects prior to building the spatial index makes it possible to get close to the goal. As in the case of the B-tree buffer approach, sorting the objects in Z-order does the trick. Using such an ordering would mean that the insertion activity would be rather localized in the tree, so once nothing has been inserted into a node a for a long time, a will most probably not be inserted into again.

The process of choosing quadtree blocks to flush makes use of a set of statistics that is maintained for each node a:

1. The time at which the last insertion was made into the quadtree block corresponding to a.

2. The number of q-objects in the subtree rooted at a (recall that a q-object is the part of an object that intersects a containing leaf block).

3. The number of nodes in the subtree rooted at a, not counting the node a itself (for leaf nodes, this is always 0).

After a node in the pointer-based quadtree is flushed to the MBI, its contents (i.e., child nodes for non-leaf nodes or object lists for leaf nodes) are deallocated, and the node is marked as an MBI node. The node is kept around in case there are subsequent insertions into it, in which case the MBI-based PMR quadtree insertion routine is invoked for that quadtree block. The amount of memory used by a subtree is proportional to the number of q-objects and nodes in the subtree. Thus, the flushing process is guaranteed to free a certain percentage Q, termed the *flushing quotient*, of the q-objects or nodes. The decision process is recursive on the pointer-based quadtree nodes and starts at the root. Initially, the number of q-objects, N_q, and nodes, N_n, to free is set at Q times the total number of q-objects and nodes in the memory-based structure. Each time a node is flushed, N_q and N_n are reduced as appropriate, terminating the process if either value reaches zero (actually, N_n never reaches zero unless it is a multiple of 2^d, so either it can be made a multiple of 2^d initially, or the terminating condition can be made that it is less than 2^d).

A recursive invocation of the flushing method proceeds as follows, where n is the node under consideration. If n is a leaf node, it is flushed. For a non-leaf node, if the number of q-objects and nodes in the subtree rooted at n is less than or equal to N_q and N_n, respectively, then the whole subtree rooted at n is flushed to the MBI. Otherwise, the child nodes of n are considered in the order of their last insertion time (i.e., nodes that have not been inserted into the longest are considered first). The flushing process is applied recursively to the child nodes that have not yet been flushed and whose number of q-objects is at least $\frac{1}{2^d}$ times the number of q-objects in the whole subtree rooted at n (recall that n has 2^d child nodes implying that this quantity is the average number of q-objects in the child nodes). The latter rule is a heuristic that tends to flush nodes that will not be inserted into again, given that objects are inserted in Z-order, as it makes sure that only child nodes with an above average number of q-objects are processed. Note that often, only the first few child nodes of n are looked at, since the process terminates once enough q-objects and nodes have been freed from memory.

Figure **??** presents the pseudo-code for the flushing method. In the figure, procedure SELECTFLUSHNODES calls procedure SORTCHILDLIST to sort the child nodes in the order of the time of last insertion into them. For low dimensions this does not add much overhead. However, it is not really necessary to sort all the nodes, especially since often only the first few are really processed. Thus in our implementation, we determine the sort order of the nodes incrementally as needed.

In addition to controlling how much memory is freed, the flushing quotient provides a means for controlling how deep into the tree the recursive flushing process descends in its search for nodes to flush. In other words, a small flushing quotient will cause a deeper descent into the tree than will a large flushing quotient. Also, note that we don't require both N_q and N_n to fall down to zero. The reason is that this tends to cause too many nodes to be flushed (i.e., objects are highly likely to be inserted into many of the flushed nodes). The reason for basing the heuristic process on both the number of q-objects and nodes is that it provides more consistent results (in terms of the amount of memory that is freed) than using only one, especially for small buffer sizes.

In the flushing process, we use the time of last insertion to impose an order on child nodes that are visited when processing a non-leaf node. Since this method is useful when objects are sorted in Z-order, it seems that we might as well visit the child nodes in Z-order as well. Nevertheless, our method of ordering the child nodes to visit according to the time that they were last visited is better than visiting them in Z-order, even when the data is inserted in Z-order, although the difference in execution time is usually small (i.e., on the order of 3-5%). However, for unsorted or nearly-sorted data (as often occurs with real-life data), our method offers significant improvements (typically over 30%).

4 Empirical Results

We implemented both of our approaches to speeding up the bulk-loading of quadtrees and ran experiments with two-dimensional line data, both real-world and randomly generated. The real-world data consists of three data sets from the TIGER/Line File [**?**]:

1. Washington DC: 19,185 line segments.

2. Prince George's County, MD: 59,551 line segments.

3. Roads in Washington DC metro area: 200,482 line segments.

The randomly generated data sets have 64,000, 128,000 and 256,000 line segments and were constructed by generating random infinite lines in a manner that is independent of translation and scaling of the coordinate system [**?**]. These lines are clipped to the map area to obtain line segments, and then subdivided further at intersection points with other line segments so that at the end, line segments meet only at endpoints.

In our experiments, we chose to store the entire geometry of the objects in the PMR quadtree. The side length of the space containing the data was $2^{15} = 32768$ and the splitting threshold was set at 8. Larger splitting thresholds make the quadtree buffering approach even more attractive. However, as 8 is a commonly used splitting threshold, this is the value we used.

The programs we used were compiled with the GNU C++ compiler with full optimization (–O3), and the experiments were conducted on a SUN SPARCstation 5 Model 70 (rated at 60 SPECint92 and 47 SPECfp92) with 32MB of memory.

4.1 Findings

Tables **??** through **??** show the execution time for building PMR quadtrees for the six data sets using six insertion methods and three different line segment insertion orders. In the tables, "Reg" denotes the regular insertion method that does not use buffering. "BB-large" and "BB-small" denote B-tree buffering with a large buffer and a small buffer (100 nodes occupying 400K), respectively. The large buffer size fills almost all available memory, and is large enough to hold the entire quadtree except for the two largest data sets. Similarly, "QB-large" and "QB-small" denote quadtree buffering with a large and a small buffer (100K), respectively. Again, the large buffer for quadtree buffering is large enough to hold entire quadtree for all but the largest data sets. The reason for including the cases using the large buffers is to reveal the maximum speedup that can be achieved with buffering, as in this extreme case no flushing of quadtree nodes needs to be done until the whole tree has been built. This gives a useful yardstick for assessing the performance of our flushing heuristics when using a small buffer. While 100K may not seem like a very small buffer, it is nevertheless a small fraction of the size of the index, or less than 4% for the DC data set, and even less for the others. Furthermore, in an era when 32MB of main memory is considered small, 100K is not very "large". Finally, "Both-small" is the result of using both buffering methods simultaneously with a small buffer

```
global nodesToFree, qObjsToFree

procedure SELECTFLUSHNODES(node) →
  if (node is a leaf node, or node is a non-leaf node with at most nodesToFree nodes
      and qObjsToFree q-objects in the subtree rooted at node) then
    FLUSHSUBTREETOMBI(node);
    nodesToFree ← nodesToFree − node.nodesInSubtree
    qObjsToFree ← qObjsToFree − node.qObjCount
  else if (node is a non-leaf node) then
    childList ← SORTCHILDLIST(node)
    for (childNode in childList) do
      if (childNode is not already flushed and contains at least 1/2^d of the q-objects in
          subtree of node) then
        SELECTFLUSHNODES(childNode)
        if (nodesToFree < 2^d or qObjsToFree ≤ 0) then
          terminate
        endif
      endif
    enddo
  endif

procedure INSERTOBJECT(object, qTree) →
  ...
  if (memory is low)
    nodesToFree ← qTree.root.nodeCount * flushQuotient
    qObjsToFree ← qTree.root.qObjCount * flushQuotient
    SELECTFLUSHNODES(qTree.root)
  endif
  ...
```

Figure 1: Pseudo-code for what is done when memory is low upon inserting an object.

size for both. We used a flushing quotient of .25 for the quadtree buffering method.

In Table **??**, the line segments were inserted in the order that they are encountered in the data set. In Table **??**, the line segments were inserted in Z-order based on their centroid[4]. In Table **??**, the line segments were inserted in a random order. For each buffered method, the tables indicate its execution time (given in seconds) as well as its relative performance with respect to that of the "Reg" method (the % columns). All the insertion methods can be seen to benefit from insertion in Z-order, but to a varying degree. This is also the case with the random data sets for the insertion method that does not use any buffering. Recall that Z-order corresponds to the order imposed by Morton block values in the B-tree thereby lessening the attractiveness of using other space ordering methods (e.g., the Peano-Hilbert order) to sort the line segments. As we expected, both B-tree and quadtree buffering perform worse on unsorted and randomly inserted objects when using small buffer sizes.

Figure **??** summarizes the speedup in the insertion time for the five buffering methods in comparison to insertion without buffering when the line segments are inserted in Z-order. We can see that quadtree buffering is up to 9 times faster than not using any buffering when the quadtree buffers are large, while being about 5 times faster when the quadtree buffers are small. The time needed to sort the line segments ranged from around 4 seconds for the DC map up to around 40 seconds for the largest maps. Figure **??** summarizes the insertion time improvements for the five methods when the sorting time is taken into account in comparison with inserting unsorted line segments without any buffering.

[4] For the Rand256K data set, a 200K buffer was used for "QB-small", as a 100K buffer proved to be not enough to give good speedup (almost 50% of the time of "Reg"). 200K is only about 0.5% of the size of the index.

	Reg	BB-large		BB-small		QB-large		QB-small		Both-small	
Data set	Time	Time	%	Time	%	Time	%	Time	%	Time	%
DC	208.9	149.0	71.3	151.3	72.4	32.0	15.3	115.7	55.4	96.8	46.3
PG	798.4	497.6	62.3	506.6	63.5	103.5	13.0	502.3	62.9	401.1	50.2
Roads	3028.9	1764.2	58.2	1820.7	60.1	376.7	12.4	2078.2	68.6	1410.2	46.6
Rand64K	1040.0	610.0	58.7	681.1	65.5	132.3	12.7	810.8	78.0	612.9	58.9
Rand128K	2329.1	1312.2	56.3	1501.6	64.5	299.8	12.9	1862.9	80.0	1369.6	58.8
Rand256K	5861.0	3727.3	63.6	3783.7	64.6	1850.4	31.6	5027.4	85.8	3473.0	59.3

Table 1: Inserting unsorted segments.

	Reg	BB-large		BB-small		QB-large		QB-small		Both-small	
Data set	Time	Time	%	Time	%	Time	%	Time	%	Time	%
DC	236.3	145.4	61.5	146.1	61.8	31.8	13.4	40.3	17.1	38.5	16.3
PG	840.5	482.0	57.3	484.5	57.6	101.2	12.0	147.6	17.6	159.6	19.0
Roads	3421.5	1747.1	51.1	1739.7	50.8	383.7	11.2	616.1	18.0	554.9	16.2
Rand64K	985.9	599.1	60.8	606.9	61.6	132.7	13.5	202.5	20.5	182.7	18.5
Rand128K	2240.6	1287.2	57.5	1283.9	57.3	288.8	12.9	530.1	23.7	384.6	17.2
Rand256K	5406.9	2909.1	53.8	2852.2	52.8	654.9	12.1	1115.1	20.6	1042.2	19.3

Table 2: Inserting segments sorted in Z-order of their centroids.

It is not intuitively obvious what is a good value for the flushing quotient for the quadtree buffering method. In order to shed some light on this issue, Figure **??** summarizes a set of experiments that

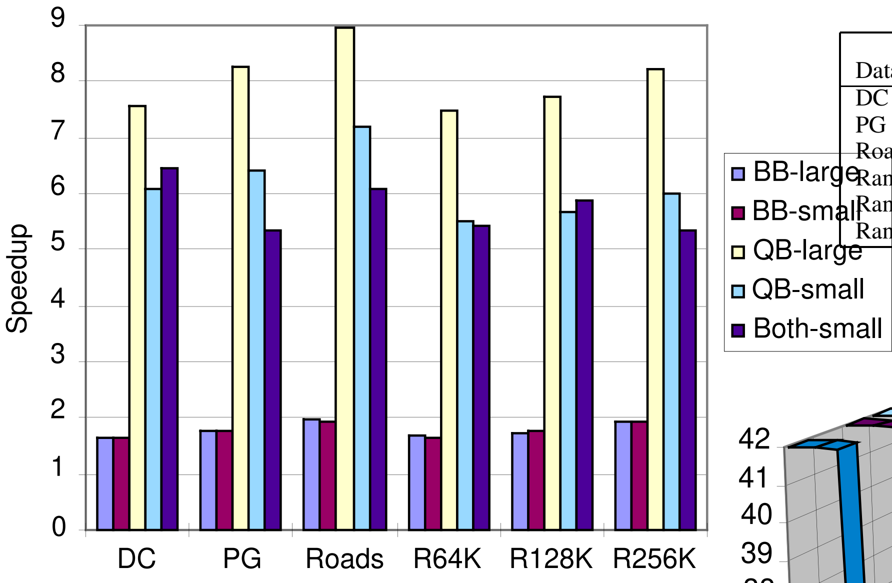

Figure 2: Improvements of the buffering methods over no buffering when the line segments are sorted in Z-order of their centroids prior to being inserted.

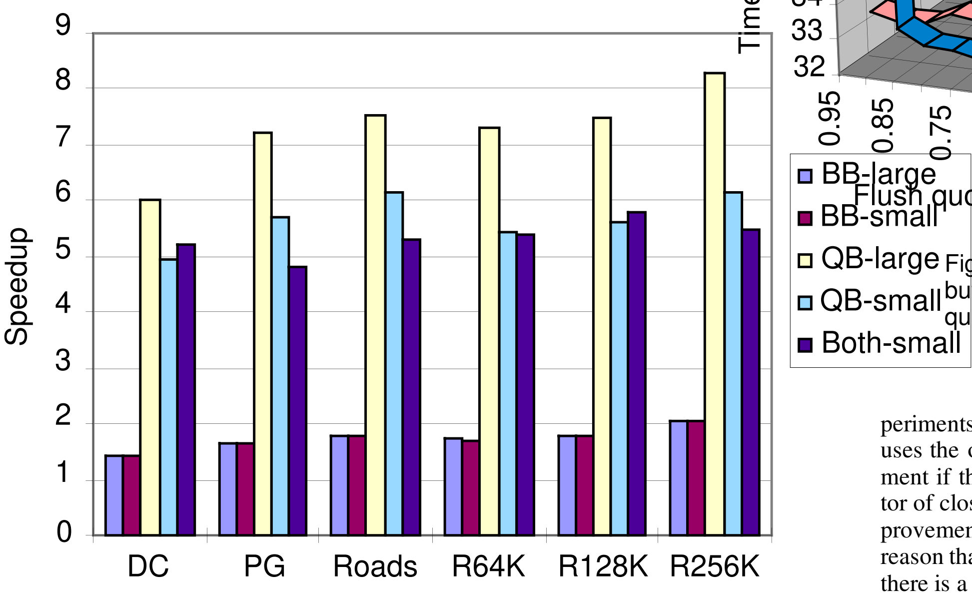

Figure 3: Improvements of the buffering methods over no buffering, taking sorting time into account.

use the DC data set, where the flushing quotient as well as the buffer size are varied. The entire pointer-based quadtree occupies about 2500K if it is all built in memory for this data set, so only a about 4% of the tree can fit in a buffer size of 100K. The figure suggests that a flushing quotient ranging between .25 and .35 is good for small buffer sizes. Although a flushing quotient in this range gave fairly stable results, the behavior is rather chaotic for other values when using small buffer sizes. This leads us to believe that there is a room for improvement in the heuristics used in our flushing algorithm.

In all of the experiments that we described, we stored the entire geometry of the objects in the PMR quadtree. We also ran some experiments with the quadtree buffering in the SAND system, which uses the object table approach. We found a factor of 3 improvement if the cost of building the object table is included, or a factor of close to 5 if that cost is not included, in contrast with an improvement of a factor of 8 or more in our earlier experiments. The reason that the difference is less for the object table method is that there is a cost involved in accessing the object table, which resides on disk, and that cost is the same whether or not buffering is employed for the quadtree. However, if the object table fits in memory, then the speedup is similar to what we saw earlier. Even if the object table does not fit in memory, storing parts of it in a buffer would also improve performance. For objects sorted in Z-order, a very small buffer would be adequate, since a set of objects that participates in a quadtree node split are usually clustered together. Another possible approach to make quadtree buffering more effective in SAND would be to employ a hybrid scheme that stores both the geometry of the object as well as a pointer into an object table (recall that the pointer into the object table is necessary in case the object has some non-spatial data associated with it). In fact, the hybrid scheme should yield the same speedup as we saw in our previous experiments. Both of these approaches (i.e, use a buffer on Z-ordered object table or storing object geometry in a quadtree) should speed up many query algorithms as they make it less costly to access the geometry of objects.

	Reg	BB-large		BB-small		QB-large		QB-small		Both-small	
Data set	Time	Time	%	Time	%	Time	%	Time	%	Time	%
DC	262.0	145.9	55.7	190.1	72.6	32.4	12.4	195.1	74.5	160.5	61.3
PG	934.4	494.4	52.9	725.7	77.7	105.6	11.3	807.7	86.4	679.5	72.7
Roads	3902.9	1749.3	44.8	3185.9	81.6	1016.5	26.0	3375.6	86.5	2950.7	75.6
Rand64K	1132.9	606.3	53.5	933.5	82.4	138.3	12.2	928.2	81.9	855.2	75.5
Rand128K	2588.9	1306.0	50.4	2208.1	85.3	347.3	13.4	2180.4	84.2	2083.6	80.5
Rand256K	8656.5	5207.1	60.2	7777.4	89.8	10276.5	118.7	7990.9	92.3	7838.7	90.6

Table 3: Inserting line segments in random order

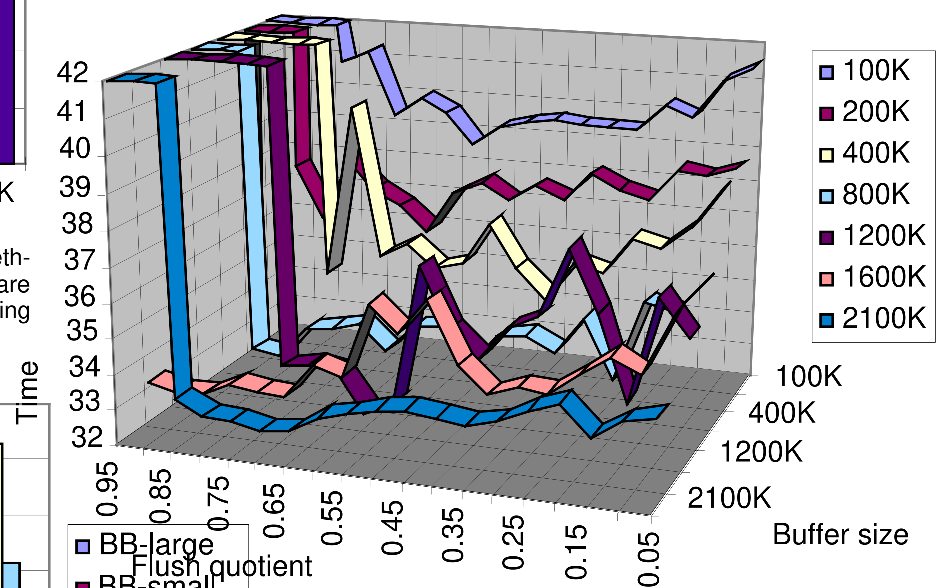

Figure 4: Inserting the DC data set with quadtree buffering when the buffer size and the flushing quotient are varied.

4.2 Performance of Spatial Join

In order to test the applicability of our quadtree buffering approach, we performed a small experiment with the spatial join example mentioned in Section **??**: given a collection of line segments representing roads and another representing rivers, find all locations where a road and a river intersect. We used the road data set already mentioned (200,482 line segments), and a river data set from the same geographic area (37,495 line segments). Below, we give a description of the experiments and the time to execute each one:

- Building a spatial index on the road and river data set (taking into account the time to sort in Z-order):
    1. without any buffering: 3029 (road) and 472 (river) seconds.
    2. with B-tree buffering using a small buffer (100 nodes): 1766 (road) and 266 (river) seconds.
    3. with quadtree buffering using a small buffer (100K): 652 (road) and 73 (river) seconds.
- Performing the spatial join with a spatial index on both data sets: 45 seconds.
- Performing the spatial join with a spatial index on the road data set but not on the river data set: 533 seconds or almost 9 minutes.
- Performing the spatial join with a spatial index on the river data set but not on the road data set: 2212 seconds or almost 37 minutes.
- Performing the spatial join with no index on either of the data sets: over 8 hours.
- Building a spatial index on the output of the spatial join (in our experiment, we did the build after the join had been computed, but building the index simultaneously with the join would yield very similar results):
    1. without any buffering: 183 seconds.
    2. with B-tree buffering (100 node buffer): 109 seconds.
    3. with quadtree buffering (large buffer): 24 seconds.
    4. with quadtree buffering (100K buffer): 70 seconds.
- Building a spatial index on the output of the spatial join after sorting the output in Z-order (taking into account the sorting time):
    1. without any buffering: 152 seconds.
    2. with B-tree buffering (100 node buffer): 93 seconds.
    3. with quadtree buffering (100K buffer): 28 seconds.

If either of the data sets is not indexed prior to the query, then it is much better to build an index using quadtree buffering and then execute the query with an index on both data sets. Using B-tree buffering when building the index offers small advantage, while building the index using no buffering is at best approximately the same or much worse than running the query without an index.

Now, let us focus more closely on merits of quadtree buffering. If an index existed for the road data set but not for the river data set, then it is 4.5 times faster to build the index with quadtree buffering and run the query (118 seconds as opposed to 533 seconds). For the converse case (i.e., no index on road data set), it is over 3 times faster to build an index on roads with quadtree buffering and run the query (697 seconds as opposed to 2212 seconds). If neither data set has an index, it would take 770 seconds (about 13 minutes) to build an index on both using quadtree buffering and to perform the spatial join with the indexes, which is more than an order of magnitude faster than computing the join without any indexes. Even the slowest quadtree building method (no buffering) offers significant benefit over performing the query witout any indexes, as it takes "only" about one hour to build the two indexes. Interestingly, it takes longer to build the indexes, even using quadtree buffering, than to perform the query with them.

Another interesting point is that the overhead of building an index on the result of the spatial join is reasonably small when using quadtree buffering. If a somewhat large buffer space is available (in this case 1.5MB), then this can be done during the query execution (for a 24 seconds, or about 43%, overhead). Otherwise, it is better to build the index after the spatial join has been executed (for a 28 seconds, or about 62%, overhead). In contrast, building the quadtree of the output of the spatial join with B-tree buffering or without buffering is two to four times more expensive than the spatial join itself.

5 Concluding Remarks

We have shown that quadtree buffering enabled us to build a quadtree in time that was an order of magnitude smaller than our original PMR quadtree implementation. Even when using modest amounts of memory for the buffering, the improvement was considerable (a factor of 5). The B-tree buffering approach offered some improvements over the original implementation that did not use any buffering, but not nearly as much as the quadtree buffering approach, even for large buffer capacity when disk activity is kept at a minimum. This demonstrates that the linear quadtree storage method is highly CPU intensive, at least for insertions. A significant portion of the CPU time is spent in computing operations on Morton block values, a cost that was avoided in the quadtree buffering approach by storing the Morton block values in the pointer-based quadtree structure. Other factors that explain the difference in performance are the higher overhead involved in traversing the quadtree through the MBI as well as the repeated splitting and merging of B-tree nodes resulting from quadtree node splits due to deletion and reinsertions of Morton block values.

Future work includes investigating whether our buffering strategies for bulk-loading may be used to speed up dynamic insertions and queries. Also, the fact that our system can build PMR quadtrees efficiently will enable us to build a spatial query processor for SAND that exploits this to construct spatial indexes for temporary results (e.g., results from other, possibly non-spatial, queries), or for un-indexed spatial relations, prior to spatial operations on them. This is particularly important for complex operations such as spatial joins.

References

[1] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman. The *s*-tree: An efficient index for multidimensional objects. (also Springer Verlag Lecture Notes in Computer Science XXX), July 1997.

[2] W. G. Aref and H. Samet. An approach to information management in geographical applications. In *Proceedings of the Fourth International Symposium on Spatial Data Handling*, volume 2, pages 589–598, Zurich, Switzerland, July 1990.

[3] W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases — Second Symposium, SSD'91*, pages 299–318, Zurich, Switzerland, August 1991. (also Springer Verlag Lecture Notes in Computer Science 525).

[4] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 190–197, Vienna, Austria, April 1993.

[5] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[6] T. Brinkhoff and H. P. Kriegel. The impact of global clustering on spatial database systems. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 168–179, Santiago, Chile, September 1994.

[7] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference*, pages 237–246, Washington, DC, May 1993.

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

[9] C. Esperança and H. Samet. An overview of the SAND spatial database system. *Communications of the ACM*, to appear 1997.

[10] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[11] O. Günther. Efficient computation of spatial joins. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 50–59, Vienna, Austria, April 1993.

[12] A. Guttman. R–trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[13] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 606–618, Zurich, September 1995.

[14] I. Kamel and C. Faloutsos. On packing R-trees. *Second International Conference on Information and Knowledge Management (CIKM)*, pages 490–499, November 1993.

[15] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, Santiago, Chile, September 1994.

[16] M. Lindenbaum and H. Samet. A probabilistic analysis of trie-based sorting of large collections of line segments. Computer Science Department TR-3455, University of Maryland, College Park, MD, April 1995.

[17] M. L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 209–220, Minneapolis, MN, June 1994.

[18] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. IBM Ltd., Ottawa, Canada, 1966.

[19] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the SIGMOD Conference*, pages 270–277, San Francisco, May 1987.

[20] Oracle Corporation. Advances in relational database technology for spatial data management. Oracle spatial data option technical white paper, September 1996.

[21] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 181–190, Waterloo, Canada, April 1984.

[22] D. Rotem. Spatial join indices. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 500–509, Kobe, Japan, April 1991. IEEE Computer Society, IEEE Computer Society Press.

[23] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the SIGMOD Conference*, pages 17–31, Austin, TX, May 1985.

[24] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

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

[26] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[27] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Submitted for publication, currently available on the WWW at http://vision.ucsd.edu/papers/simret/, 1996.