CAR-TR-917 IRI-97-12715
CS-TR-4033 DEFG0295ER25237
July 1999

Speeding Up Construction of Quadtrees for Spatial Indexing

Gísli R. Hjaltason and Hanan Samet

Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742-3275
grh@cs.umd.edu and hjs@cs.umd.edu

Abstract

Spatial indexes, such as those based on the 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 present a number of techniques for speeding up the construction of two quadtree-based spatial indexes, the PMR quadtree and the PR quadtree. The PMR quadtree can index arbitrary spatial data, whereas the PR quadtree is specialized for multidimensional point data. The quadtrees are 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 based on a space-filling curve. For the PMR quadtree, we present two complementary techniques: an improved insertion algorithm and a bulk-loading method. The bulk-loading method can be extended to handle bulk-insertions into an existing PMR quadtree. For the PR quadtree, we present a bulk-loading method, which also can be extended to handle bulk-insertions. We make some analytical observations about the I/O cost and CPU cost of our PMR quadtree bulk-loading algorithm, and conduct an extensive empirical study of all the techniques presented in the paper. Our techniques are found to yield significant speedup compared to traditional quadtree building methods, even when the size of the main memory buffer is very small compared to the size of the resulting quadtrees. The usefulness of speeding up quadtree construction is demonstrated by studying a spatial join operation that requires the construction of a spatial index for its operands as well as its spatial output. In this case, the performance of the spatial join was significantly improved by the presence of the spatial indexes.

To appear in the *VLDB Journal*.

Keywords: spatial indexing, buffering, I/O, spatial join, query processing

This work was supported in part by the National Science Foundation under Grant IRI-97-12715, the Department of Energy under Contract DEFG2095ER25237, and the Italian National Group for Mathematical Computer Science (GNIM).

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 un-indexed relations or intermediate query results 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 faster than the total time to execute it when the time to build the index is included. 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. The same issues arise in spatial databases, where attribute values may be of a spatial type, in which case the index is a spatial index (e.g., a quadtree).

In the research reported here, we address the problem of constructing and updating spatial indexes in situations where the database is dynamic. In this case, the time to construct or update an index is critical, since database updates and queries are interleaved. Furthermore, slow updates of indexes can seriously degrade query response, which is especially detrimental in modern interactive database applications. There are three ways in which indexes can be constructed or updated for an attribute of a relation (i.e., a set of objects). First, if the attribute has not been indexed yet (e.g., it represents an intermediate query result), an index must be built from scratch on the attribute for the entire relation (known as *bulk-loading*). Second, if the attribute already has an index, and a large batch of data is to be added to the relation, the index can be updated with all the new data values at once (known as *bulk-insertion*). Third, if the attribute already has an index, and a small amount of data is to be added (e.g., just one object), it may be most efficient to simply insert the new objects, one by one, into the existing index. In our work, we present methods for speeding up construction and updating of quadtree-based spatial indexes for all three situations. In particular, we focus on the PMR quadtree spatial index [39], and to a lesser degree on the PR quadtree multidimensional point index.

The issues that arise when the database is dynamic have often been neglected in the design of spatial databases. The problem is that 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 complex query operations such as the spatial join. As an example of a spatial join, 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 join of the two relations, where the join predicate is true for road and river pairs that have at least one point in common. Since computing the spatial join operation is expensive without spatial indexes, it may be worthwhile to build a spatial index if one is not present for one of the relations. Furthermore, the output of the join may serve as input to subsequent spatial operations (i.e., a cascaded spatial join as would be common in a spatial spreadsheet [29]), so it may also be advantageous to build an index on the join result. In this way, the time to build spatial indexes can play an important role in the overall query response time.

The PMR quadtree is of particular interest because an earlier study [27] showed that the PMR quadtree performs quite well for spatial joins in contrast to other spatial data structures such as the R-tree [25] (including variants such as the R^*-tree [10]) and the R^+-tree [50]. This was especially true when the execution time of the spatial join included the time needed to build spatial indexes[1]. Improving the perfor-

[1]Note that fast construction techniques for the R-tree, such as the packed R-tree [46] and Hilbert-packed R-tree [30], were not taken into account in this study as they tend to result in a worse space decomposition from the point of view of overlap than the standard R-tree construction algorithms.

mance 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 prototype spatial database system built by us called SAND (Spatial and Non-Spatial Data) [5, 6, 20], which employs a data model inspired by the relational algebra. 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 [40]. Therefore speeding their construction has an appeal beyond our SAND prototype.

In this paper, we introduce a number of techniques for speeding up the construction of quadtree-based spatial indexes. Many of these techniques can be readily adapted to other spatial indexes that are based on regular partitioning, such as the buddy-tree [49] and the BANG file [22]. We present two complementary techniques for the PMR quadtree, an improved insertion algorithm and a bulk-loading method for a disk-based PMR quadtree index. The improved PMR quadtree insertion algorithm can be applied to any quadtree representation, and exploits the structure of the quadtree to quickly locate the smallest quadtree node containing the inserted object, thereby greatly reducing the number of intersection tests. The approach that we take in the PMR quadtree bulk-loading algorithm 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 (termed "flushing"). A key technique for making effective use of the internal memory quadtree buffer is to sort the objects by their spatial occupancy prior to inserting them into the quadtree. This allows the flushing algorithm to flush only nodes that will never be inserted into again. Our treatment of PMR quadtree bulk-loading has several other elements, including alternative strategies for freeing memory in the quadtree buffer and a technique for achieving high storage utilization. In addition, we show how our bulk-loading method can be extended to handle bulk-insertions into an existing quadtree index.

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. Nevertheless, the two algorithms share the requirement that the input data be sorted in a particular way. Note also that the PR quadtree bulk-loading algorithm can be extended to handle bulk-insertions.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the PR and PMR quadtrees, and the disk-based quadtree representation used in SAND. Section 4 introduces an improved PMR quadtree insertion algorithm. Section 5 presents our PMR quadtree bulk-loading approach. Section 6 discusses how the PMR quadtree bulk-loading algorithm can be extended to handle bulk-insertions. Section 7 describes our PR quadtree bulk-loading method (for point data). Section 8 presents some analytical observations, mainly regarding the PMR quadtree bulk-loading approach. Section 9 discusses the results of our experiments, while concluding remarks are made in Section 10.

2 Related Work

Methods for bulk-loading dynamic access structures have long been sought. The goal of such methods is to reduce the loading time, the query cost of the resulting structure, or both. The B-tree, together with its variants, is the most commonly used dynamic indexing structure for one-dimensional data. Rosenberg and Snyder [44], and Klein, Parzygnat, and Tharp [33] introduced methods for building space-optimal B-trees, i.e., ones having the smallest number of nodes, or equivalently, the highest possible average storage utilization. Their methods yield both a lower load time, and lower average query cost due to the improved storage utilization. Both methods rely on pre-sorting the data prior to building the tree; a similar approach can be used to bulk-load B^+-trees. Huang and Viswanathan [28] took a more direct approach to reducing query cost, while possibly increasing loading time. However, no experiments were reported. They introduce a dynamic programming algorithm, inspired by existing algorithms for binary

search trees, that builds a tree that yields the lowest expected query cost, given the access frequencies of key values. Another example of bulk-loading algorithms for non-spatial structures is the one by Ciaccia and Patella [17] for the M-tree, a dynamic distance-based indexing structure.

In recent years, numerous bulk-loading algorithms for spatial indexing structures have been introduced. Most of the attention has been focused on the R-tree and related structures. Among the exceptions are two algorithms for the grid file. Li, Rotem and Srivastava [36] introduced a dynamic programming algorithm that operates in a parallel domain, and primarily aims at obtaining a good grid partitioning. Leutenegger and Nicol [35] introduced a much faster solution, which results in grid file partitions that are in some ways better.

Most bulk-loading strategies that have been developed for the R-tree have the property that they result in trees that may be dramatically different from R-trees built with dynamic insertion rules [10, 25]. Some of these methods use a heuristic for aggregating objects into the leaf nodes [30, 34, 46], while others explicitly aim at producing good partitioning of the objects and thus a small level of overlap [3, 12, 42, 52]. Roussopoulos and Leifker [46] introduced a method (termed the packed R-tree) that uses a heuristic for aggregating rectangles into nodes. First, the leaf nodes in the R-tree are built by inserting the objects into them in a particular order. The nonleaf nodes are built recursively in the same manner, level by level. The order used in the packed R-tree method [46] is such that the first object to be inserted into each leaf node is the remaining object whose centroid has the lowest x-coordinate value, whereas the rest of the objects in the node are its $B-1$ nearest neighbors, where B is the node capacity[2]. Kamel and Faloutsos [30] devised a variant of the packed R-tree, termed a Hilbert-packed R-tree, wherein the order is based purely on the Hilbert code of the objects' centroids. Leutenegger, López, and Edgington [34] proposed a somewhat related technique, which uses an ordering based on a rectilinear tiling of the data space. The advantage of packing methods is that they result in a dramatically shorter build time than when using dynamic insertion methods. Unfortunately, the heuristics they use to obtain their space partitioning usually produce worse results (i.e., in terms of the amount of overlap) than the dynamic ones. This drawback is often alleviated by the fact that they result in nearly 100% storage utilization (i.e., most R-tree nodes are filled to capacity). DeWitt et al. [19] suggest that a better space partitioning can be obtained with the Hilbert-packed R-tree by sacrificing 100% storage utilization. In particular, they propose that nodes be initially filled to 75% in the usual way. If any of the items subsequently scheduled to be inserted into a node cause the node region to be enlarged by too much (e.g., by more than 20%), then no more items are inserted into the node. In addition, a fixed number of recently packed leaf nodes are combined and resplit using the R*-tree splitting algorithm to further improve the space partitioning. Gavrila [24] proposed another method for improving the space partitioning of R-tree packing, through the use of an optimization technique. Initially, an arbitrary packing of the leaf nodes is performed, e.g., based on one of the packing algorithms above. Next, the algorithm attempts to minimize a cost function over the packing, by moving items from one leaf node to a nearby one.

The bulk-loading strategies for the R-tree that aim at improved space partitioning have in common that they operate on the whole data set in a top-down fashion, recursively subdividing the set in some manner at each step. They differ in the particular subdivision technique that is employed, as well as in other technical details, but most are specifically intended for high-dimensional point data. Since building R-trees with good dynamic insertion methods (e.g., [10]) is expensive, these methods generally achieve a shorter build time (but typically much longer than the packing methods discussed above), as well as improved space partitioning. One example of such methods is the VAMSplit R-tree of White and Jain [52], which uses a variant of a k-d tree splitting strategy to obtain the space partitioning. García, López, and

[2]The exact order proposed by Roussopoulos and Leifker [46] for the packed R-tree appears to be subject to a number of interpretations. Most authors citing the packed R-tree describe it as using an order based solely on the x-coordinate values of the objects' centroids which produces node regions that are highly elongated in the direction of the y-axis, whereas this is not exactly what was originally proposed.

Leutenegger [42] present a similar technique, but they introduce the notion of using a user-defined cost function to select split positions. The S-tree of Aggarwal et al. [3] is actually a variant of R-trees that is not strictly balanced; the amount of imbalance is bounded, however. The technique presented by Berchtold, Böhm, and Kriegel [12] also has some commonality with the VAMSplit R-tree. However, their splitting method benefits from insights into effects that occur in high-dimensional spaces, and is able to exploit flexibility in storage utilization to achieve improved space partitioning. A further benefit of their technique is that it can get by with only a modest amount of main memory, while being able to handle large data files.

Two methods have been proposed for bulk-loading R-trees that actually make use of dynamic insertion rules [8, 13]. These methods are in general applicable to balanced tree structures which resemble B-trees, including a large class of multidimensional index structures. Both techniques are based on the notion of the buffer-tree [7], wherein each internal node of the tree structure contains a buffer of records. The buffers enable effective use of available main memory, and result in large savings in I/O cost over the regular dynamic insertion method (but generally in at least as much CPU cost). In the method proposed by van den Bercken, Seeger, and Widmayer [13], the R-tree is built recursively bottom-up. In each stage, an intermediate tree structure is built where the lowest level corresponds to the next level of the final R-tree. The nonleaf nodes in the intermediate tree structures have a high fan-out (determined by available internal memory) as well as a buffer that receives insertions. Arge et al. [8] achieve a similar effect by using a regular R-tree structure (i.e., where the nonleaf nodes have the same fan-out as the leaf nodes) and attaching buffers to nodes only at certain levels of the tree. The advantages of their method over the method in [13] are that it is more efficient as it does not build intermediate structures, and it results in a better space partition. Note that the algorithm in [13] does not result in an R-tree structure identical to that resulting from the corresponding dynamic insertion method, whereas the algorithm in [8] does (assuming reinsertions [10] are not used). In addition, the method of [8] supports bulk-insertions (as opposed to just initial bulk-loading as in [13]) and bulk-queries, and in fact, intermixed insertions and queries.

With the exception of [8], all the methods we have mentioned for bulk-loading R-trees are static, and do not allow bulk-insertions into an existing R-tree structure. A few other methods for bulk-insertion into existing R-trees have been proposed [16, 32, 45]. The cubetree [45] is an R-tree-like structure for on-line analytical processing (OLAP) applications that employs a specialized packing algorithm. The bulk-insertion algorithm proposed by Roussopolous, Kotidis, and Roussopolous [45] works roughly as follows. First, the data set to be inserted is sorted in the packing order. The sorted list is merged with the sorted list of objects in the existing data set, which is obtained directly from the leaf nodes of the existing cubetree. A new cubetree is then packed using the sorted list resulting from the merging. This approach is also applicable to the Hilbert-packed R-tree [30] and possibly other R-tree packing algorithms. Kamel, Khalil, and Kouramajian [32] propose a bulk-insertion method in which new leaf nodes are first built following the Hilbert-packed R-tree [30] technique. The new leaf nodes are then inserted one by one into the existing R-tree using a dynamic R-tree insertion algorithm. In the method presented by Chen, Choubey, and Rundensteiner [16], a new R-tree is built from scratch for the new data (using any construction algorithm). The root node of the new tree is then inserted into the appropriate place in the existing R-tree using a specialized algorithm that performs some local reorganization of the existing tree based on a set of proposed heuristics. Unfortunately, the algorithms of [16, 32] are likely to result in increased node overlap, at least if the area occupied by the new data already contains data in the existing tree. Thus, the resulting R-tree indexes are likely to have a worse query performance than an index built from scratch from the combined data set.

None of the bulk-loading techniques discussed above are applicable to quadtrees. This is primarily because quadtrees use a very different space partitioning method from grid files and R-trees, and because they are unbalanced and their fan-out is fixed. Additional complications arise from the use of most disk-

resident representations of quadtrees (e.g., the linear quadtree), as well as from the property that each non-point object may be represented in more than one leaf node (sometimes termed "clipping"; see Section 3). Nevertheless, some analogies can be drawn between our bulk-loading methods and some of the above methods. For example, like many of the above algorithms, we rely on sorting the objects in our algorithm and we use merging to implement bulk-insertions as done in the cubetree [45] (although our merging process is very different).

In addition to the numerous bulk-loading and bulk-insertion algorithms proposed for the R-tree, there have been several different proposals for improving dynamic insertions [4, 9, 10, 43, 31]. Most have been concerned with improving the quality of the resulting partitioning, at the cost of increased construction time, including the well known R*-tree method of Beckmann et al. [10], and the polynomial time optimal node splitting methods of Becker et al. [9] and García, López, and Leutenegger [43]. In addition, [10] and [43] also introduced heuristics for improving storage utilization. Ang and Tan [4] developed a linear time node splitting algorithm that they claim produces node splits that are better than the original node splitting algorithms [25] and competitive with that of the R*-tree. The Hilbert R-tree of Kamel and Faloutsos [31] employs the same heuristic as the Hilbert-packed R-tree [30], maintaining the data rectangles in strict linear order based on the Hilbert codes of their centroids. This is done by organizing them with a B$^+$-tree on the Hilbert codes, augmented with the minimum bounding rectangle of the entries in each node. Thus, updates in the Hilbert R-tree are inexpensive, while it often yields query performance similar to that of the R*-tree (at least in low dimensions).

Recently, Wang, Yang, and Muntz [51] introduced the PK-tree, a multidimensional indexing structure based on regular partitioning. In [53], they proposed a bulk-loading technique for the PK-tree, which is based on sorting the data in a specific order, determined by the partitioning method. Their method resembles our bulk-loading techniques in that a space-filling curve is used to order the data prior to building the tree. In fact, our PR quadtree bulk-loading algorithm (Section 7) can be viewed as an adaptation of their method. However, it is not applicable for building a PMR quadtree for non-point objects, since each object may be represented in more than one leaf node.

One of the topics of this paper is a bulk-loading technique for PMR quadtrees. This subject has been previously addressed by Hjaltason, Samet, and Sussman [26]. The bulk-loading technique presented in this paper is an improvement on the algorithm in [26]. In particular, our flushing algorithm (which writes to disk some of the quadtree nodes from a buffer) is guided by the most recently inserted object, whereas the one in [26] relied on a user-defined parameter. Unfortunately, it was unclear how to choose the optimal parameter value or how robust the algorithm was for any given value. Moreover, the heuristic employed by the flushing algorithm in [26] did not always succeed in its goal, and sometimes flushed nodes that intersected objects that had yet to be inserted into the quadtree. A further benefit of our improved approach is that it permits a much higher storage utilization in the disk-based quadtree, which reduces the I/O cost for constructing the quadtree as well as for performing queries.

3 Quadtrees and their Implementation

In this section, we first briefly discuss the general concept of quadtrees. Next we define the PMR quadtree, followed by a description of the implementation of quadtrees in SAND.

3.1 Quadtrees

By the term *quadtree* [47, 48] 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; i.e., the sub-blocks of a block are obtained by halving the block

along each coordinate axis. Figure 1 shows a simple quadtree decomposition of space. One way of conceptualizing a quadtree is to think of it as an extended 2^d-ary tree, i.e., a tree in which every nonleaf node has 2^d children (e.g., Figure 1b). Thus, below we use the terms quadtree node and quadtree block interchangeably. In this view, the quadtree is essentially a *trie*, where the branch structure is based on space coverage. Another way to view the quadtree is to focus on the space decomposition, in which case the quadtree can be thought of as being an adaptive grid (e.g., Figure 1a). Usually, there is a prescribed maximum height of the tree, or equivalently, a minimum size for each quadtree block.

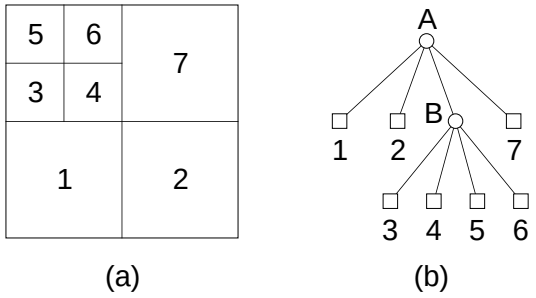

Figure 1: (a) The block decomposition and (b) tree structure of a simple quadtree, where leaf blocks are labeled with numbers and nonleaf blocks with letters.

Many different varieties of quadtrees have been defined, differing in the rules governing node splitting, the type of data being indexed, and other details. An example is the PR quadtree [47], which indexes point data. Points are stored in the leaf blocks, and the splitting rule specifies that a leaf block must be split if it contains more than one point. In other words, each leaf block contains either one point or none. Alternatively, we can set a fixed bucket capacity c, and split a leaf block if it contains more than c points (this is termed a bucket PR quadtree in [47]).

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 are encoded in some manner, and the result is used as a key into an auxiliary disk-based data structure, such as a B-tree. This approach is termed a *linear quadtree* [23].

Quadtrees were originally designed for the purpose of indexing two- and three-dimensional space. Although the definition of a quadtree is valid for a space of arbitrary dimension d, quadtrees are only practical for a relatively low number of dimensions. This is due to the fact that the fan-out of internal nodes is exponential in d, and thus becomes unwieldy for d larger than 5 or 6. Another factor is that the number of cells tends to grow sharply with the dimension even when data size is kept constant[3], and typically is excessive for more than 4 to 8 dimensions, depending on the leaf node capacity (or splitting threshold) and data distribution. For a higher number of dimensions, we can apply the k-d tree [11] strategy of splitting the dimensions cyclically (i.e., at each internal node, the space is split into two equal-size halves), for a constant fan-out and improved average leaf node occupancy. The resulting space partitioning can be effectively structured using the PK-tree technique [51], for example. In the remainder of this paper, we will usually assume a two-dimensional quadtree to simplify the discussion. Our methods are general, however, and work for arbitrary dimensions.

[3]This is due to the fact that average leaf node occupancy tends to fall as the number of dimensions increases.

3.2 PMR Quadtrees

The *PMR quadtree* [39] is a quadtree-based dynamic spatial data structure for storing objects of arbitrary spatial type (e.g., see Figure 2 which 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 2 is split into three q-edges as it intersects three leaf nodes.

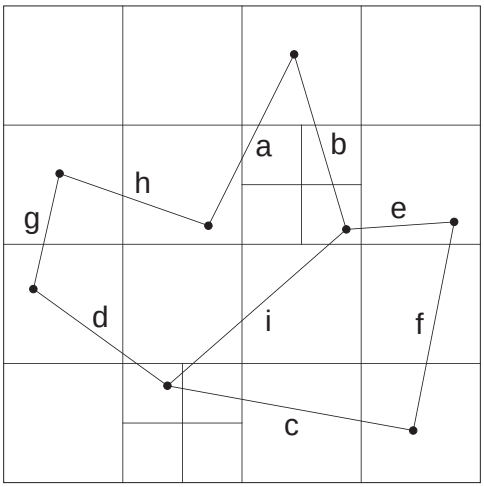

Figure 2: A PMR quadtree for line segments with a splitting threshold of 2, where the line segments have been inserted in alphabetical order.

A key aspect of the PMR quadtree is its splitting rule, i.e., the condition under which a quadtree block is split. The PMR quadtree employs a user-determined *splitting threshold t* for this purpose. If the insertion of an object *o* causes the number of objects in a leaf block *b* to exceed *t* and *b* is not at the maximum decomposition level, then *b* is split and the objects in *b* (including *o*) are inserted into the newly created sub-blocks that they intersect. These sub-blocks are not split further at this time, even if they contain more than *t* objects. Thus, a leaf block at depth *D* can contain up to $t + D$ objects, where the root is at depth 0 (there is no limit on the number of objects in leaf nodes at the maximum depth). The rationale for not immediately splitting newly formed leaf blocks is that this avoids excessive splitting. This aspect of the PMR quadtree gives rise to a probabilistic behavior in the sense that the order in which the objects are inserted affects the shape of the resulting tree. As an example, in Figure 2, if line segment g were inserted after line segment i instead of after line segment f, then the decomposition of the SE quadrant of the SW quadrant of the root, where c, d, and i meet, would not have taken place. Nevertheless, it is rarely of importance which of the possible quadtree shapes arise from inserting a given set of objects. We will exploit this later on, by re-ordering the objects to allow a more efficient quadtree construction process.

3.3 Quadtree Implementation in SAND

The implementation of quadtrees used in the SAND spatial database is based on a general linear quadtree implementation called the *Morton Block Index* (abbreviated *MBI*). Our bulk-loading methods are applicable to any linear quadtree implementation, and should be easily adapted to any other disk-based repre-

sentation of quadtrees. Nevertheless, for concreteness, it is helpful to review some of the details of our system.

3.3.1 Morton Codes and Morton Block Values

The MBI encodes quadtree blocks using a pair of numbers, termed a *Morton block value*. The first number is the *Morton code* of the corner of the quadtree block closest to the origin (i.e., the lower-left corner in two dimensions), 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. Coordinate values are constrained to be w-bit integers, where w is a user-determined value between 0 and 32. Thus, a Morton code for d-dimensional space occupies $d \cdot w$ bits. Furthermore, the side length of the space covered by the MBI is 2^w, and coordinate values range from 0 to $2^w - 1$ in each dimension[4]. Since the minimum side length of a quadtree block that can be represented is 1, the maximum height of the quadtree is w. Not all possible Morton block values correspond to legal quadtree blocks. For example, for a two-dimensional quadtree, the only quadtree block that can have a lower-left corner of $(1,1)$ has a side length of 1. On the other hand, a Morton code can correspond to many quadtree blocks, e.g., the point with coordinate values $(0,0)$ can be the lower-left corner of a block of any size. Observe that the number of dimensions, d, is not limited by the MBI, although very high values are not practical.

Morton codes provide a mapping from d-dimensional points to one-dimensional scalars, 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 [38]. It is also known as a Z-order [41] since it traces a 'Z' pattern in two dimensions. Many other space-ordering methods exist, such as the Peano-Hilbert, Cantor-diagonal, and spiral orders. However, of these, only the Morton and Peano-Hilbert orders are practical for ordering quadtree blocks. The codes derived from the Peano-Hilbert order are usually called Hilbert codes. Morton codes can also be transformed into so-called Gray codes, in which two successive code values differ only in one bit [21]. Figure 3 presents an example of the ordering resulting from these three encoding methods. The advantage of Morton codes over Hilbert codes and Gray codes is that it is computationally less expensive to convert between a Morton code and its corresponding coordinate values (and vice versa) than for the other two encoding schemes, especially compared to the Hilbert code. In addition, various operations on Morton block values can be implemented through simple bit-manipulation operations on Morton codes; e.g., computing the Morton block values for sub-blocks. Nevertheless, Hilbert and Gray codes have the advantage that they better preserve locality (e.g., the Euclidean distance between the locations of two points with successive code values is lower on average than for Morton codes), which may reduce query cost [1]. However, for the most part, operations on the quadtree are independent of the actual encoding scheme being used, and in particular, this is true of our bulk-loading method. Thus, in most of this paper, any mention of Morton codes (or Z-order) can be replaced by Hilbert or Gray codes (or the ordering induced by them). When warranted, we mention issues arising from the use of Hilbert or Gray codes.

Figure 4a illustrates the Morton code order imposed on the quadtree blocks for the quadtree in Figure 2. The contents of the MBI for this PMR quadtree are partially shown in Figure 4b, where the order in the list corresponds to Morton code order. To illustrate actual Morton block values, assume that the side length of the data space is $2^4 = 16$. The coordinate values of the lower-left corner of the block labeled 15 are $(2,12)$, or $(0010\text{b}, 1010\text{b})$ ("b" indicates binary). Thus, the Morton code value of this block is 10001100b (i.e., the bit order for the coordinate values is *yxyxyxyx*), which equals 140. The size of this block is $2 = 2^1$, so the Morton block value is $[140,1]$. Observe that the two least significant bits of the Morton code are 0, which is the case for all blocks of size $2 = 2^1$. In general, for a block of size 2^s,

[4]This limited range of coordinate values is not a real drawback, as it is a simple matter to transform coordinate values in any other range into the range of a Morton code, and vice versa.

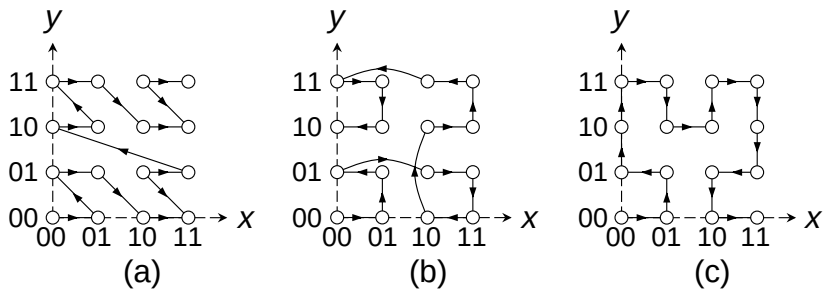

Figure 3: Ordering imposed by code values in a 4 by 4 grid when using (a) Morton code, (b) Gray code, and (c) Hilbert code.

the $s \cdot d$ least significant bits are 0, where d is the dimensionality. If block 15 had to be split, the Morton code values of the child blocks would be 10001100b, 10001101b, 10001110b, and 10001111b. In other words, only d bits of the original Morton code are modified. Similarly, the Morton code of the parent block of block 15 is 10000000b.

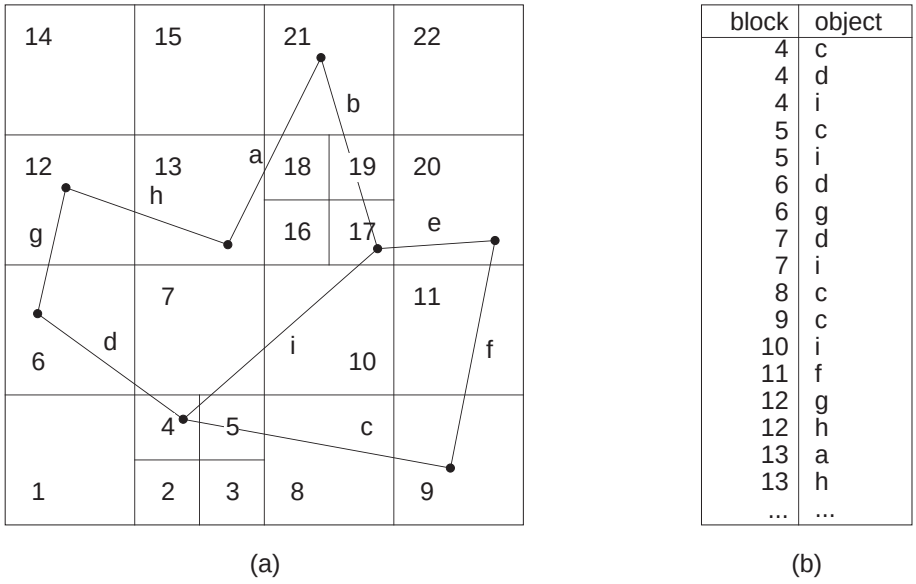

Figure 4: (a) The PMR quadtree for the line segments in Figure 2, with the quadtree blocks numbered in Morton code order. (b) Some of the corresponding items stored in the linear quadtree.

3.3.2 B-tree

The MBI uses a B-tree to organize the quadtree contents[5], 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

[5]The MBI can also be based on a B^+-tree. This has some advantages, notably when scanning in key order. However, the difference is not very significant, and is offset by a slightly greater storage requirement for the B^+-tree.

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 room for a new B-tree node. In addition, we employ a node locking mechanism in order to ensure that the nodes on the path from the root to the current node are not replaced; this is useful in queries that scan through successive items in the B-tree, since the nodes on the path may be needed later in the scan.

3.3.3 Object Representation

The amount of data associated with each object in the MBI is limited only by the B-tree node size. This flexibility permits different schemes for storing spatial objects in quadtree indexes implemented with the MBI. One scheme is to store the entire spatial description of the object, while another scheme 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, wherein we store both the spatial description of the object and an object ID. The disadvantage of the first scheme is that it potentially leads to much wasted storage for non-point objects, as they may be represented more than once in the PMR quadtree. The drawback 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, we must use that scheme (or the hybrid one) 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 populations).

As previously mentioned, SAND employs a data model inspired by the relational algebra. The basic storage unit is an attribute, which may be non-spatial (e.g., integers or character strings) or spatial (e.g., points, line segments, polygons, etc.). Attributes are collected into relations, and relational data is stored as tuples in tables, each of which is identified by a *tuple ID*. In SAND relations, the values of spatial attributes (i.e., their geometry) are stored directly in the tuples belonging to the relation. When the PMR quadtree is used to index a spatial attribute in SAND, the tuple ID of the tuple storing each spatial object must be stored in the quadtree (i.e., we use the second scheme described above). For simple fixed-size spatial objects (such as points, line segments, rectangles, etc.), SAND also permits storing the geometric representation in the index (i.e., resulting in a hybrid scheme). This allows performing geometric computations during query evaluation without accessing the tuples. Alternatively, a separate object table associated with the index can be built for only the values of the spatial attribute. Object IDs in that table are then represented in the index, while the tuple ID is stored in the object table. This is advantageous when the size of the spatial attribute values (in bytes) is small compared to the size of a whole tuple. A further benefit is that this object table can be clustered by spatial proximity, such that nearby objects are likely to be located on the same disk page. Spatial clustering is important to reduce the number of I/O operations performed for queries, as stressed by Brinkhoff and Kriegel [14].

3.3.4 Empty Leaf Nodes

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 some 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 can be very wasteful, since a large number of leaf blocks will tend to be empty.

4 Improved PMR Quadtree 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. Below, we introduce a technique that dramatically reduces the CPU cost of insertions. However, before we get into our improved insertion algorithm, we present the traditional PMR quadtree insertion algorithm in Figure 5. This algorithm can be used for either a pointer-based implementation or a linear quadtree implementation of a PMR quadtree (e.g., the Morton Block Index). Of course, the definitions of the various utility routines (i.e., ADDTOLEAF, ISLEAF, MAKENONLEAF, OBJECTCOUNT, and OBJECTLIST) would be different, as would the representation of *node*. In the MBI implementation, *node* is represented with a Morton block value, and these routines obtain their results by accessing the B-tree. In particular, ADDTOLEAF inserts into the B-tree, MAKENONLEAF deletes from the B-tree, ISLEAF performs a lookup, while OBJECTCOUNT and OBJECTLIST perform a lookup followed by a linear scan. Observe that in the case of a linear quadtree implementation, the nonleaf nodes are not physically present in the MBI. However, the insertion algorithm is based on a top-down traversal of the tree and thus simulates their existence by constructing their corresponding Morton block values.

```
procedure INSERTOBJECT(object) →
  INSERT(root, object)

procedure INSERT(node, object) →
  if (INTERSECTS(object, node)) then
    if (ISLEAF(node)) then
      ADDTOLEAF(node, object)
      if (OBJECTCOUNT(node) > threshold) then
        SPLIT(node)
      endif
    else
      foreach (childNode of node) do
        INSERT(childNode, object)
      endfor
    endif
  endif

procedure SPLIT(node) →
  objList ← OBJECTLIST(node)
  MAKENONLEAF(node)
  foreach (childNode of node) do
    foreach (object in objList) do
      if (INTERSECTS(object, childNode)) then
        ADDTOLEAF(childNode, object)
      endif
    endfor
  endfor
```

Figure 5: Pseudo-code for PMR quadtree insertion.

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 by the INTERSECTS function. It is implemented by first converting the Morton block value for *node* into object space coordinates. The number of intersection tests when inserting an object 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, i.e., the determination of *childNode* from the Morton block value *node* (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 6a, 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 6b), or there may be more than one leaf node contained in the minimal enclosing quadtree block (e.g., Figure 6c).

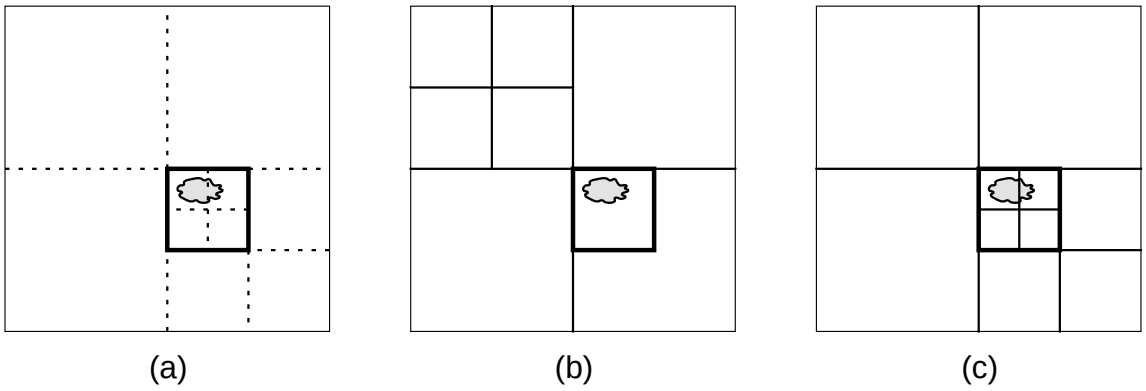

Figure 6: (a) Computation of the minimum bounding block for an object, denoted by heavy lines. Broken lines indicate potential quadtree block boundaries. The minimum bounding block can (b) be enclosed by a leaf node or (c) coincide with a nonleaf node.

4.1 Algorithm

An algorithm based on the idea of minimum enclosing quadtree block is shown in Figure 7. In particular, procedures INSERTOBJECT and SPLIT in Figure 7 replace the procedures with the same name in Figure 5. Again, the same algorithm can be applied to any representation of the PMR quadtree. Procedure INSERTOBJECT uses the functions COMPUTEENCLOSINGBLOCK and FINDENCLOSINGNODE to locate the smallest node in the quadtree index that contains *object*. If this node is a leaf node, *object* is directly added to it (subject to a split if the node contains more objects than the splitting threshold). Otherwise, INSERT (from Figure 5) is invoked on the child nodes of *node*. The task of locating the smallest node containing the object is divided into two functions since it naturally decomposes into two subtasks. The first, COMPUTEENCLOSINGBLOCK, is based only on the geometry of the object and computes its minimum enclosing quadtree block, while the second, FINDENCLOSINGNODE, accesses the quadtree index to locate an actual quadtree node.

The number of intersection tests is also reduced in procedure SPLIT in Figure 7 through the use of COMPUTEENCLOSINGBLOCK. If *enclosingBlock*, returned by COMPUTEENCLOSINGBLOCK for an object, is smaller than the leaf node, then we know that *enclosingBlock* is properly contained in *node* and only intersects one of its child nodes. This child node is determined by the function CHILDCONTAINING. When the nodes are encoded with Morton block values, CHILDCONTAINING can be computed using simple bit manipulations. Once *childNode* has been determined, *object* is added to it and deleted from the list of objects[6]. After the first **foreach** loop in procedure SPLIT is completed, the objects that remain on *objList* have enclosing blocks that are equal to or larger than *node*. Since these objects can intersect more than one child node of *node*, we apply the regular split method to them (i.e., SPLIT in Figure 5).

```
procedure INSERTOBJECT(object) →
  enclosingBlock ← COMPUTEENCLOSINGBLOCK(object)
  node ← FINDENCLOSINGNODE(enclosingBlock)
  if (ISLEAF(node)) then
    ADDTOLEAF(node, object)
    if (OBJECTCOUNT(node) > threshold) then
      SPLIT(node)
    endif
  else
   foreach (childNode of node) do
     INSERT(childNode, object) /* see Figure 5 */
   endfor
  endif

procedure SPLIT(node) →
  objList ← OBJECTLIST(node)
  MAKENONLEAF(node)
  foreach (object in objList) do
    enclosingBlock ← COMPUTEENCLOSINGBLOCK(object)
    if (SIZE(enclosingBlock) < Size(node)) then
      childNode ← CHILDCONTAINING(node, enclosingBlock)
      ADDTOLEAF(childNode, object)
      DELETE(objList, object)
    endif
  endfor
  /* apply regular split method to objects remaining in objList */
  foreach (childNode of node) do
    foreach (object in objList) do
      if (INTERSECTS(object, childNode)) then
        ADDTOLEAF(childNode, object)
      endif
    endfor
  endfor
```

Figure 7: PMR quadtree insertion with dramatically lower CPU cost.

[6]Recall that in the PMR quadtree, the child nodes resulting from a split are not split again as a result of reinserting the objects in the split node, even if the threshold is exceeded.

Of the procedures and functions first used in INSERTOBJECT and SPLIT in Figure 7, only procedure FINDENCLOSINGNODE depends on the actual quadtree representation. Recall that FINDENCLOSINGNODE looks for a node in the quadtree index that spatially encloses *enclosingBlock* (or is equal to it). In the case of the MBI, this is accomplished by a single access to the B-tree. In particular, we search for the leaf block having the largest Morton block value smaller than or equal to that of *enclosingBlock*. If one is found (e.g., Figure 6b), FINDENCLOSINGNODE returns its Morton block value; otherwise, *enclosingBlock* corresponds to a nonleaf node in the quadtree so its value is returned (e.g., Figure 6c). The ISLEAF test in procedure INSERTOBJECT can be executed by making use of information returned by FINDENCLOSINGNODE, so that no additional B-tree accesses are needed. Observe that if empty leaf nodes are not represented in the MBI, the definition of FINDENCLOSINGNODE is slightly more complicated, since *enclosingBlock* may fall into an empty leaf block. In addition to reducing the number of intersection tests, the improved insertion algorithm also results in fewer invocations of ISLEAF (in procedure INSERT in Figure 5), and thus fewer B-tree lookups. However, the saving that this results in is mostly in CPU cost, since the B-tree nodes that get accessed will frequently already be in the B-tree buffer.

When applied to pointer-based quadtrees, procedure FINDENCLOSINGNODE must descend the pointer-based quadtree from the root until it encounters a node whose region encloses the region computed by COMPUTEENCLOSINGBLOCK. If Morton block values are used to represent the node regions (either stored within the tree or computed on the fly), the descent can be guided by the Morton block value returned by COMPUTEENCLOSINGBLOCK. In particular, at a nonleaf node, the next child to visit can be determined from bits in the Morton code of the minimum enclosing Morton block value. Thus, the descent is relatively inexpensive.

4.2 Discussion

The reduction in the number of intersection tests performed by the INSERT and SPLIT procedures in 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 6b, 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 6c, 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 $2^d D'_{\text{ave}}$ 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 6c, 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 $p(D_{\text{ave}} - D'_{\text{ave}})$, where D_{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'_{ave} will be nearly as high as D_{ave}, so the number of intersection tests will be small. In the extreme case of point objects, no intersection tests are needed and $D_{\text{ave}} \approx D'_{\text{ave}}$[7].

Figure 8 shows values of D_{ave} and D'_{ave} for six line segment data sets used in our experiments (for more details, see Section 9). For these data sets, we found that p ranged from 6 to 6.5, but in general

[7] D_{ave} and D'_{ave} are typically not exactly equal for points, since D_{ave} is an average over leaf nodes while D'_{ave} is an average over objects. Alternative, and perhaps more accurate, definitions of D_{ave} that make it equal to D'_{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.

its value probably depends on the data distribution. As an example of the reduction in intersection tests, the average depth of minimally enclosing nodes for the "PG" data set is more than 7, so the number of intersection tests for each object is reduced by $4 \cdot 7 = 28$. The value of $D_{\text{ave}} - D'_{\text{ave}}$ is about 1.4 for "PG", and the number of intersection tests actually performed for each object is about $6 \cdot 1.4 \approx 8.5$ on the average. On the other hand, the original insertion algorithm performs about $28 + 8.5 = 36.5$ intersection tests per object. Thus, the improved algorithm reduces the number of intersection tests by a factor of more than 4. For the other data sets, the reduction factor ranged from 3 to 5.

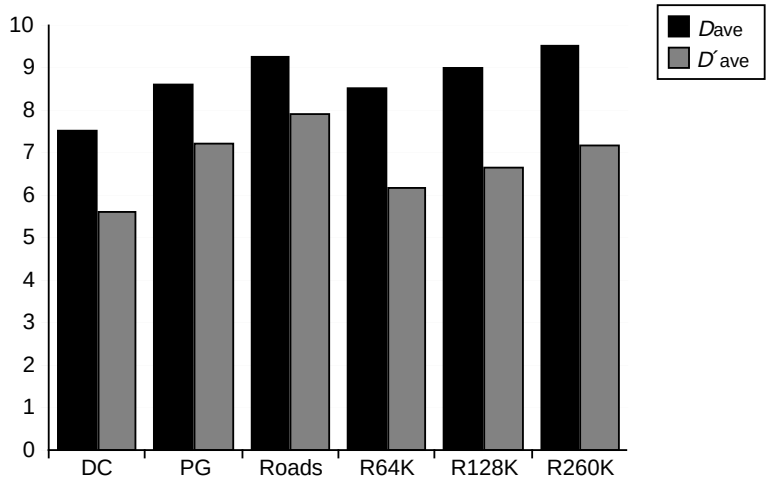

Figure 8: Average depth of leaf nodes (D_{ave}) and minimally enclosing nodes (D'_{ave}).

The CPU cost saving due to the reduction in number of intersection tests is tempered by the cost of invoking COMPUTEENCLOSINGBLOCK (whose CPU cost is similar to that of INTERSECTS). This is especially true for procedure SPLIT, since COMPUTEENCLOSINGBLOCK must be recomputed for each object, and the intersection tests must be invoked anyway if the enclosing block is larger than or equal to the leaf node being split. To reduce unnecessary invocations of COMPUTEENCLOSINGBLOCK we can retain the value computed by the COMPUTEENCLOSINGBLOCK invocation in INSERTOBJECT, so it need not be computed again in SPLIT. Of course, this is usually not practical as it increases the storage requirement for the objects. Nevertheless, this technique is useful in our bulk-loading algorithm, since only a limited number of nodes is kept in memory, while the nodes that have been written to disk are never split again.

5 Bulk-Loading PMR Quadtrees

Our implementation of the PMR quadtree as described in Section 3.3 is very flexible in several respects, and we found its performance to be respectable for dynamic insertions and a wide range of queries. However, for loading a large number of objects at once (i.e., bulk-loading), its performance was somewhat lacking. As we looked for reasons for the poor performance, we identified several sources of inefficiencies, both in terms of CPU cost and I/O cost. The main reason for excessive CPU cost is the high cost of quadtree node splitting. 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 can 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. Such B-tree

reorganizations are expensive in terms of CPU time if frequent enough, and add a considerable overhead as we shall see in Section 9.

The basic idea of our bulk-loading 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. Our approach can be characterized as buffering quadtree nodes, which contrasts to the normal buffering of B-tree nodes. Thus, we sometimes refer to our approach as *quadtree buffering*.

The remainder of this section is organized as follows: In Section 5.1 we present an overview of our quadtree buffering approach. Next, in Section 5.2, we present the details of our flushing algorithm, which frees up space if none is left in the main memory buffer. In Section 5.3 we describe an alternative method for freeing memory which is used if the flushing algorithm fails to do so. Our bulk-loading approach requires sorted input, so we discuss two efficient external sort algorithms in Section 5.4. Finally, in Section 5.5 we show how the MBI B-tree can be built efficiently and with a high storage utilization.

5.1 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 onto disk (i.e., inserted into the MBI). When all the objects have been inserted into the pointer-based quadtree, the entire tree is inserted into the MBI and the quadtree building process is complete. In order to maintain compatibility with the MBI-based PMR structure, we use Morton block values to determine the space coverage of the memory-resident quadtree blocks. Note that it is not necessary to store the Morton block values in the nodes of the pointer-based structure, as they can be computed during traversals of the tree. However, a careful analysis of execution profiles revealed that a substantial percentage of the CPU 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.

How do we choose which quadtree blocks to flush when available memory has been exhausted? Without some knowledge of the objects that are yet to be inserted into the quadtree, it is impossible to determine which quadtree blocks will be needed later on, i.e., which quadtree blocks are not intersected by any subsequently inserted object. However, carefully choosing the order in which the objects are inserted into the tree provides exactly such knowledge. This is illustrated in Figure 9, which depicts a quadtree being built. In the figure, the shaded rectangle represents the bounding rectangle of the next object to insert. If the objects are ordered in Z-order based on the lower-left corner of their minimum bounding rectangle (i.e., the corner closest to the origin), we are assured that none of the quadtree blocks in the striped region will ever be inserted into again, so they can be flushed to disk. The reason why this works is that the lower-left corner of a rectangle has the lowest Morton code value of all points in the rectangle. Thus, using this order, we know that all points contained in the current object, as well as in all subsequently inserted objects, have a higher Morton code value, and we can flush quadtree blocks that cover points with lower Morton code values.

When using Hilbert or Gray codes, we also would use the lowest code value for points in the minimum bounding rectangle of an object as a sort code. However, in this case the lowest code value occurring in a rectangle is typically not in the lower-left corner, but can occur anywhere on its boundary. Thus, the lowest code value is somewhat more expensive to compute when using Hilbert or Gray codes than when using Morton codes. One way to do so is to recursively partition the space, at each step picking the partition having the lowest code value that intersects the rectangle.

The flushing process is described in greater detail in Section 5.2. Under certain conditions, this flushing method fails to free any memory, although this situation should rarely occur. In Section 5.3 we explain why, and present two alternative strategies that can be applied in such cases.

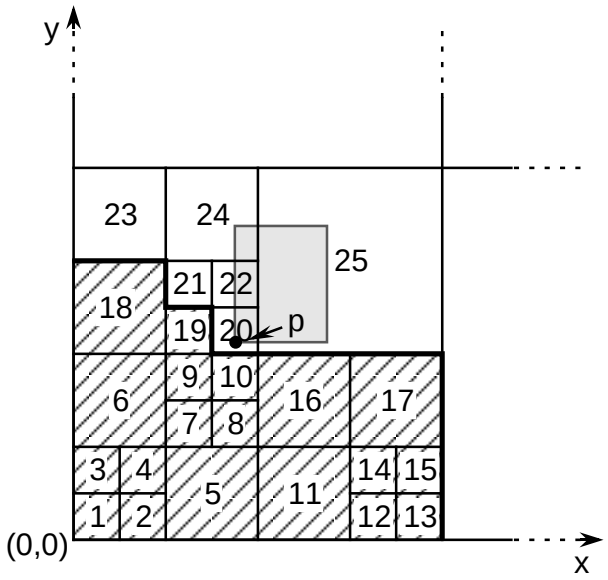

Figure 9: A portion of a hypothetical quadtree, where the leaf nodes are labeled in Z-order. The shaded rectangle is the bounding rectangle of the next object to insert.

5.2 Flushing Algorithm

Informally, the flushing algorithm can be stated as follows:

1. Let p be the lower-left corner of the bounding rectangle of the object to insert next (see Figure 9).
2. Visit the unflushed leaf blocks in the pointer-based quadtree in increasing order of the Morton code of their lower-left corner (e.g., for Figure 9, in increasing order of the labels).
   (a) if the quadtree block intersects p (e.g., the leaf block labeled 20 in Figure 9), then terminate the process;
   (b) otherwise, insert the leaf block into the MBI.

Figure 10 presents a more precise portrayal of the algorithm in terms of a top-down traversal of the pointer-based quadtree. The flushing algorithm is embodied in the function FLUSHNODES in Figure 10 and is invoked by INSERTOBJECT when the pointer-based quadtree is taking too much space in memory. For each nonleaf node, FLUSHNODES recursively invokes itself exactly once, for the child node whose region intersects p, while it invokes FLUSHSUBTREETOMBI to flush the subtrees rooted at all unflushed child nodes that occur earlier in Morton code order. Thus, FLUSHNODES traverses the pointer-based tree down to the leaf node whose region intersects p. For example, in Figure 9, the function traverses the tree down to the node labeled 20, while it flushes the entire subtrees containing nodes 1 through 10 and nodes 11 through 17, as well as the leaf nodes labeled 18 and 19. The FLUSHSUBTREETOMBI function removes the given subtree from the buffer memory, and marks it flushed. That way, we will know in subsequent invocations whether a given quadtree node is merely empty, or has already been flushed. When all objects have been inserted into the quadtree, FLUSHSUBTREETOMBI is invoked on the root node, resulting in the final tree on disk.

The function CONTAINS used in procedure FLUSHNODES can be efficiently implemented using the Morton code of p, which can be computed before flushing is initiated (i.e., in procedure INSERT). In

```
procedure INSERTOBJECT(object) →
  if (memory is low) then
    p ← lower left corner of the bounding rectangle of object
    FLUSHNODES(root, p)
  endif
  /* remainder of procedure same as in Figure 5 or Figure 7 */

procedure FLUSHNODES(node, p) →
  if (not ISLEAF(node)) then
    foreach (unflushed childNode of node) do
      /* child nodes are visited in Morton code order */
      if (CONTAINS(childNode, p)) then
        /* childNode is on the path from root to leaf containing p */
        FLUSHNODES(childNode, p)
        return /* exit function */
      else
        /* childNode has a smaller Morton code than p */
        FLUSHSUBTREETOMBI(childNode, false)
      endif
    endfor
  endif

procedure FLUSHSUBTREETOMBI(node, freeNode) →
  if (node has already been flushed) then
    return
  endif
  if (ISLEAF(node)) then
    foreach (object in node) do
      MBIINSERT(node, object)
    endif
  else
    foreach (childNode of node) do
      FLUSHSUBTREETOMBI(childNode, true)
    endfor
  endif
  if (freeNode) then
    FREENODE(node)
  else
    mark node as having been flushed and turn into empty leaf node
  endif
```

Figure 10: Pseudo-code for flushing process.

particular, let m_p be the Morton code of p, and let m_{lo} and m_{hi} be the smallest and largest Morton codes, respectively, for a quadtree block b (m_{lo} is the Morton code of its lower-left corner, while m_{hi} is the Morton code of the "pixel" in the upper-right corner). For example, for the block of size 4 by 4 with lower-left corner $(0,0)$, m_{hi} is the Morton code for the point $(3,3)$. Testing for intersection of b and p is equivalent

to checking the condition $m_{\text{lo}} \leq m_p \leq m_{\text{hi}}$. This test can be efficiently implemented with bit-wise operations. Specifically, if the size of b is $2^{w_b} \times 2^{w_b}$, then all but the low-order $2w_b$ bits of m_{lo} and m_p must match (the $2w_b$ low-order bits of m_{lo} are all 0 and those of m_{hi} are all 1).

5.3 Reinsert Freeing

The problem with the flushing algorithm presented in Section 5.2 is that it may fail to flush any leaf nodes, and thus not free up any memory space. In the example in Figure 9 this would occur if all the nodes in the striped region have already been flushed. In this case, the objects that remain in the pointer-based quadtree intersect leaf nodes labeled 20 or higher, but the lower-left corners of their minimum bounding rectangles fall into leaf nodes labeled 20 or lower (due to the insertion order). Thus, if r is a bounding rectangle of one of these objects, then either r intersects the boundary of the striped region or the lower-left corner of r falls into the leaf node labeled 20 (i.e., the unflushed leaf node with the lowest Morton code value). This condition rarely applies to a large number of objects, at least not for low-dimensional data and reasonable buffer sizes as discussed in Section 8. Nevertheless, we must be prepared for this possibility.

If the flushing algorithm is unable to free any memory, then we cannot flush any leaf nodes without potentially choosing nodes that will be inserted into later. One possibility in this event is to flush some of these leaf nodes anyway, chosen using some heuristic, and invoke the dynamic insertion procedure on any subsequently inserted objects that happen to intersect the flushed nodes. The drawback of such an approach is that we may choose to flush nodes that will receive many insertions later on. Also, this means that we lose the guarantee that B-tree insertions are performed in strict key order, thereby reducing the effectiveness of the B-tree packing technique introduced in Section 5.5 (i.e., adapted to tolerate slightly out-of-order insertions). Furthermore, our PMR quadtree bulk-insertion algorithm would not be applicable (although a usually more expensive variant could be used; see Section 6.3). The strategy we propose instead, termed *reinsert freeing*, is to free memory by removing objects from the quadtree (allowing empty leaf nodes to be merged) and scheduling them for reinsertion into the quadtree at a later time. This strategy avoids the drawbacks mentioned above, but increases somewhat the cost of some other aspects of the bulk-loading process as described below.

In reinsert freeing, we must make sure that objects to be reinserted get inserted back into the quadtree at appropriate times. We do this by sending the objects back to the sorting phase, with a new sort key (in Section 5.4 we discuss how to extend a sorting algorithm to handle reinsertions). This is illustrated in Figure 11 where the shaded rectangle is the bounding rectangle of an object that is to be reinserted (broken lines indicate the bounding rectangle of the last inserted object). The object intersects nodes labeled 18 and 21 through 24. Since node 21 is the existing node with the lowest Morton code that intersects the object, the appropriate time for inserting the object back into the quadtree is when all nodes earlier than node 21 in Morton order have already been inserted into. Thus the location used to form the new sort key of the object should intersect node 21. One choice is to compute the lower-left intersection point of the bounding rectangle and the region for node 21, shown with a dot and pointed at by the arrow. Alternatively, to avoid this computation, we could simply use the lower-left corner of node 21 as the new sort key. Observe that in either case, the new sort key is larger than the original sort key for the object. As the example illustrates, we must make sure to reinsert each object only once, even though it may occur in several leaf nodes, and the sort key is determined from the leaf node intersecting the object having the smallest Morton block value. Notice that when the object in the figure is eventually inserted again into the quadtree, it is not inserted into node 18, since that node has already been flushed.

A second issue concerning reinsert freeing is how to choose which objects to remove from the quadtree. Whatever strategy is used, it is important that we not reinsert the objects occurring in the leaf node b intersecting the lower-left corner of the most recently inserted object; e.g., the leaf node labeled 20 in

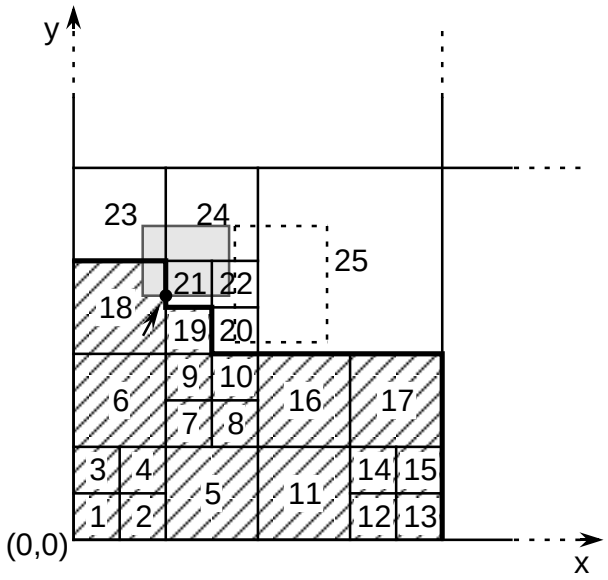

Figure 11: An example of an object that is to be reinserted (shaded rectangle). The striped region represents quadtree nodes that have been flushed, while the broken lines indicate the bounding rectangle of the object that was inserted last.

Figure 11. A simple, but effective, strategy is to remove all objects except those occurring in leaf node *b*, and merge all child nodes of non-leaf nodes not on the path from the root to *b*. Thus, the only nodes retained in the pointer-based quadtree are the nodes on the path from the root to *b*, and their children. This is the strategy that we use in our experiments (see Section 9.2.6). Another possible strategy is to visit the leaf nodes in decreasing Morton order (i.e., the ones with the highest Morton code values first), and remove the objects encountered until some fraction (say, 50%) of the quadtree buffer has been freed. One complication in this strategy is that once we have made enough buffer space available, we must then remove the objects chosen for reinsertion from the leaf nodes that remain in the buffer. Although perhaps somewhat counter-intuitive, we found that the second strategy (which frees only a portion of the buffer) usually led to a higher number of reinsertions than the first (which frees nearly the entire buffer), unless a large fraction of the buffer was freed. At best, the reduction in the number of reinsertions of the second strategy was only marginal, and even in those cases, the first strategy was usually slightly faster since reinsertion freeing was invoked less often.

An important point is that an object can only be reinserted a limited number of times, thus guaranteeing that we do not reinsert the same objects indefinitely. To see this, observe that the leaf node intersecting the sort key used last for an object to be reinserted will always have been flushed (e.g., leaf node 18 in Figure 11). This is guaranteed by the fact we do not remove objects occurring in the leaf node intersecting the search key of the object inserted last (e.g., objects occurring in leaf node 20 in Figure 11 are not reinserted). Thus, some progress always occurs between two successive reinsertions for the same object (i.e., some leaf nodes will have been flushed). The total number of insertions (original and reinsertions) for an object is never more than than $q + a'$, where q is the number of corresponding q-objects and a' is the number of ancestors of the leaf nodes containing the q-objects, not including the ancestors that completely enclose the object.

5.4 Sorting the Input

Our bulk-loading approach requires the input to be in a specific order for it to be effective when the entire quadtree cannot fit in the amount of memory allotted to the bulk-loading process. The input data will usually not be in the desired order, so it must be sorted prior to bulk-loading. Since we cannot assume that the data fits in memory, we must make use of an external memory sorting method. Whatever method is used, instead of writing the final sorted result to disk, it is preferable that the sorting phase and quadtree building phase operate in tandem, with the result of the former pipelined to the latter. This avoids the I/O cost of writing the final sorted result, and permits dealing with reinsertions (see Section 5.3).

Sorting a large set of objects can be expensive. However, as we will see in our experiments, sorting a set of objects prior to insertion is often a much less expensive process than the cost of building the spatial index. More importantly, the savings in execution time brought about by sorting far outweigh its cost. Note that some form of sorting is commonly employed when bulk-loading spatial access structures (e.g., [3, 30, 33, 34, 46, 52, 53]). Aggarwal and Vitter [2] established a lower bound on the I/O cost of external sorting, $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, where

- N is the number of data objects;
- M is the number of objects that fit into an internal memory buffer used for sorting;
- B is the number of objects that fit into a disk page (or some other unit of block transfers).

We implemented two external sorting algorithms suitable for our application. The first algorithm is a variation of the standard distribution sort [2], where we employ an application-specific partitioning scheme. This is the algorithm that we used in most of our experiments, where we found it to have very good performance. Unfortunately, our partitioning scheme is not always guaranteed to distribute sufficiently evenly to yield optimal cost (although it works well for typical data sets). Also, the algorithm is difficult to adapt to support reinsertions (Section 5.3) in an efficient manner. The second algorithm that we implemented is external merge sort [2]. This algorithm has the advantage of being provably optimal. Furthermore, in the presence of reinsertions, it is at worst only slightly suboptimal. Below, we briefly describe the external merge sort algorithm and how it can be modified to handle reinsertions.

5.4.1 Merge Sort

The external merge sort algorithm [2] first sorts the data in memory, generating short sorted *runs* on disk. These are then merged to generate longer runs, until we have a single sorted run. More precisely, the initial runs are of length M, and there are approximately N/M of them. In each merge pass, groups of R runs are merged together, reducing the number of runs by a factor of R. During a merge, B objects from each run must be kept in memory[8], so $R = M/B$. A depiction of the process is shown in Figure 12a.

As we mentioned above, this algorithm is I/O optimal. Each iteration decreases the number of runs by a factor of M/B, so we need about $\log_{M/B}(N/M)$ iterations until we have a single run. The initial formation of runs as well as each iteration require about N/B I/Os, so we have a total of $O(\frac{N}{B}(1+\log_{M/B}(N/M)) = O(\frac{N}{B}\log_{M/B}(N/B))$ I/Os.

5.4.2 Handling Reinsertions

The merge sort algorithm can be modified to handle reinsertions so that the result is only slightly suboptimal. In particular, if N^* is the number of objects plus the number of reinsertions, the modified algorithm achieves comparable I/O performance to sorting N^* objects from scratch. Observe that while

[8] Buffer space for $2B$ objects is needed for each run when using asynchronous I/O and double buffering.

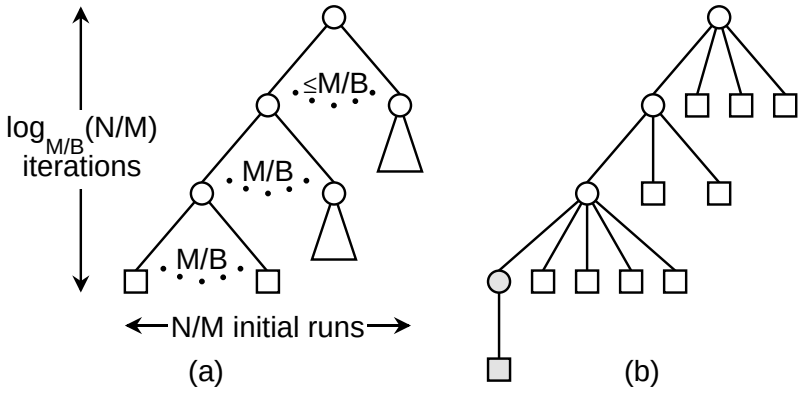

Figure 12: Depiction of external merge sorting: (a) regular, and (b) with reinsertions. In (a), squares represent runs created from the input while circles represent merged runs. In (b), the white squares represent active runs, the white circles represent future merged runs, and the shaded square represents a partial run being created in memory.

N^* is larger than N in the presence of reinsertions, the number of "pending" objects (i.e., ones that have not yet been delivered in their proper order) is never larger than N. Figure 12b illustrates our modified merge sort algorithm. The sort proceeds as before until reaching the final iteration, where we deliver the result of merging up to M/B runs to the bulk-load process. From this point on, we maintain a hierarchy of runs, as depicted in the figure. The runs at level 1 — the children of the root in Figure 12b — initially are the sort runs resulting from the final iteration, the runs at the bottom are created from reinserted objects, while the runs at intermediate levels are created by merging all existing runs at the level below. Notice that the active runs are continuously being read from to produce the input to the bulk-load process. This means that reinserted objects do not necessarily travel up the entire hierarchy, and runs at lower levels may become depleted and thereby removed from the hierarchy. A space of B objects from the sort buffer is allotted to each active run, so there can be a total of at most M/B active runs at all levels. We impose a lower limit of $M/2$ on the size of the runs being created at the bottom level, so we require up to half of the sort buffer to be available for that purpose. When no buffer space is available when an object is reinserted, the action depends on the total number of active runs: 1) if active runs total $\frac{1}{2}M/B$ or more, merge the runs at the level containing the largest number of runs, creating one run at the next highest level; 2) otherwise, create a new run at the lowest level from the reinserted objects in the buffer, which will number at least $M/2$.

A benefit of our method is that the allocation of the sort buffer is dynamically adapted to the number of reinsertions and the number of active runs at each level. When merging, the number of runs being merged may be as high as M/B, but never fewer than $\frac{M}{2hB}$, where h is the height of the hierarchy, initially about $\log_{M/B}(N/M)$. In order for our method to be optimal, the number of runs being merged each time must be sufficiently high. In particular, $\log(\frac{M}{2hB}) = \log(M/B) - \log(2h)$ must be $O(\log(M/B))$, or in other words, $\log h = \log\log_{M/B}(N/M)$ must be a constant. Unfortunately, this is not quite the case, but for all practical purposes it is. For example, even if M is only 10 times larger than B, h is less than 16 as long as N is less than 10^{16} times larger than M (for comparison, note that a terabyte is around 10^{12} bytes), so $\log_2 h$ is less than 4. Thus, $\log\frac{M}{2hB} < 5$, and $\log_{\frac{M}{2hB}}\frac{N^*}{B} < 2\log_{M/B}\frac{N^*}{B}$. In other words, the number of I/Os is at most doubled given the assumptions, which are virtually guaranteed to hold.

5.5 B-tree Packing

As a byproduct of sorting the input and using the flushing algorithm described in Section 5.2, the leaf blocks will be inserted into the MBI, and thus the B-tree, in strict Morton code order. Since Morton codes are the sort key of the B-tree, this has the unfortunate effect that most of the nodes in the B-tree become only about half full. The reason for this is that the conventional B-tree node splitting algorithm splits a node so that the two resulting nodes are about half full. However, since insertions occur in strict key order, the node receiving entries with smaller key values will never be inserted into again, and thus will remain only half full. Therefore, in general all nodes will be half full, except possibly the right-most nodes on each level (assuming increasing keys in left-to-right order). This low storage utilization increases build time, since more nodes must be written to disk, and decreases query efficiency, as more nodes must be accessed on average for each query.

The seemingly negative behavior of inserting in strict key order can easily be turned into an advantage, by splitting nodes unevenly. In other words, instead of splitting an overflowing node so that each resulting node is about half full, we split the node so that the node storing entries with lower key values receives more entries than the other. In this way, we can precisely determine the storage utilization of all but the right-most nodes on each level, setting it to be anywhere between 50% to 100% (but see below). Thus, we can achieve substantially better storage utilization than that typically resulting from building B-trees, which is about 69% for random insertions [54].

The algorithm for packing the B-tree as sketched above is shown in Figure 13. The procedure PACKINSERT is invoked to insert items. As long as the items arrive approximately in key order, PACKINSERT always inserts them into the rightmost leaf node in the B-tree. A pointer to the rightmost leaf node can be maintained by the algorithm, thereby making it immediately accessible. Procedure PACKSPLIT performs the uneven splitting of nodes, with the global variable *splitFraction* controlling the distribution of entries among the two result nodes. In the algorithm, we assume that an overflowing node contains one more record than the capacity of the disk pages. Thus, if *splitFraction* is 100%, we split a node into one full node and one empty node, with the record with the largest key being inserted into the parent as a discriminator. The drawback here is that we can end up with a B-tree in which the right-most nodes at some of the levels are empty, containing nothing but a pointer to a child node (if a nonleaf node). To alleviate this, we can either always split in such a way that there is at least one record in the right node (and thus one less than the maximum in the left node), or we can move entries to empty nodes from their siblings (via rotation) once all records have been inserted into the tree.

```
procedure PACKINSERT(item) →
  node ← rightmost leaf node in B-tree
  INSERTINTONODE(node, item)
  if (OVERFLOW(node)) then
    PACKSPLIT(node)
  endif

procedure PACKSPLIT(node) →
  splitIndex ← splitFraction · maxEntries
  parent ← SPLITNODE(node, splitIndex)
  if (OVERFLOW(parent)) then
    PACKSPLIT(parent)
  endif
```

Figure 13: Pseudo-code for B-tree packing.

The only actions performed on the B-tree by the algorithm are to insert into the rightmost leaf node and to split the rightmost node at a level in the tree. In other words, any node other than a rightmost node at a level is never changed by the algorithm. Thus, by merely buffering one B-tree node at each level, we can ensure that each node in the B-tree is written only once to disk. Another benefit of inserting in sorted order is that we avoid repeated traversals of the B-tree, thereby reducing CPU time as no key comparisons are needed.

As mentioned above, our flushing algorithm is guaranteed to lead to B-tree insertions that are strictly in key order. In other circumstances, insertions into the B-tree are mostly in key order but sometimes slightly out of order. For example, the alternative to reinsert freeing mentioned in Section 5.3 (i.e., flushing nodes that may be needed later using a heuristic) can cause out of order insertions. As another example, in Section 6.3, we discuss a variant of our bulk-insertion approach that involves updating an existing B-tree. There, the insertions are strictly in key order, but usually do not get inserted into the rightmost B-tree leaf node. The packing algorithm can be adapted to handle these situations, by locating the B-tree node that should receive the item being inserted, instead of always assuming that it should be inserted into the rightmost B-tree leaf node. In order to avoid unnecessary B-tree traversals, the algorithm can keep track of the B-tree leaf node into which the last insertion was made, and check if the item being inserted falls into the range of items stored in that leaf node. The drawback to this modification of the algorithm is that the B-tree node receiving fewer items as a result of splits may remain underfull, which may be undesirable. Nevertheless, the average storage utilization is often improved by the uneven node splits, at least if *splitFraction* is not too high (e.g., we have found that 85% often works well).

Although there are differences in some details, the algorithm in Figure 13 is similar to that of [44]. Their algorithm was presented in terms of compacting a 2-3 tree, a precursor of B-trees, but it can easily be adapted to building a B-tree from sorted data. The main difference between our algorithm and theirs is that they maintain an array of nodes that have not yet been fully constructed (at most one for each level), and these nodes are not yet connected to the main tree. In contrast, our algorithm always maintains a fully connected tree structure, which is an advantage if B-tree insertions potentially occur out of order. The B-tree packing algorithm of [33] is not applicable in our scenario, since it requires knowing in advance the number of records to insert. In addition, it is more complicated to implement than ours.

6 Bulk-Insertions for PMR Quadtrees

Our PMR quadtree bulk-loading algorithm can be adapted to the problem of bulk-inserting into an existing quadtree index. In other words, the goal is to build a PMR quadtree for a data set that is a combination of data that is already indexed by a disk-resident PMR quadtree (termed *existing data*) and data that has not yet been indexed (termed *new data*). This may be useful, for example, if we are indexing data received from an earth-sensing satellite, and data for a new region has arrived. Frequently, the new data is for a region of space that is unoccupied by the existing data, as in this example, but this is not necessarily the case. The method we describe below is equally well suited to the case of inserting into previously unoccupied regions and to the case of new data that is spatially interleaved with the existing data.

6.1 Overview

Recall that our flushing algorithm writes out the quadtree leaf nodes in Morton code order. This is also the order in which leaf nodes are stored in the B-tree of the MBI. The idea of our bulk-insertion algorithm is to build a quadtree in memory for the new data with our PMR quadtree bulk-loading algorithm. However, the flushing process is modified in such a way that it essentially merges the stream of quadtree leaf nodes for the new data with the ordered stream of quadtree leaf nodes in the PMR quadtree for the existing data. The merging process is somewhat more complicated than this brief description may imply. In particular,

in order to merge two leaf nodes they must be of the same size, and the content of the resulting merged leaf node must obey the splitting threshold. Below, we use the terms *old quadtree* when referring to the disk-resident PMR quadtree for the existing data, *new quadtree* when referring to the memory-resident PMR quadtree for the new data, and *combined quadtree* when referring to the disk-resident PMR quadtree resulting from the merge process (which indexes both the existing data and the new data). Similarly, we use *old leaf node* and *new leaf node* for leaf nodes in the old and new quadtrees, respectively.

Figure 14 illustrates the three cases that arise in the merging process, where the new data is denoted by dots (the old data is not shown). The square with heavy borders denotes a leaf block from the old quadtree, while the squares with thin borders denote leaf blocks in the new quadtree. The first case arises when an old leaf node b_o coincides with a node b_n in the new quadtree, where b_n is either a nonempty leaf node or a nonleaf node, implying that b_o intersects new data (see Figure 14a, where b_n is a nonempty leaf node). Thus, the objects contained in b_o must be inserted into the subtree rooted at b_n, subject to the splitting threshold. The second case arises when an old leaf node b_o is contained in (or coincides with) an empty leaf node b_n in the new quadtree (see Figure 14b). When this occurs, the contents of b_o can be written directly into the combined quadtree, without the intermediate step of being inserted into the new quadtree. The third case arises when an old leaf node b_o is contained in a larger nonempty leaf node b_n in the new quadtree (see Figure 14c). In this case, b_n is split, and b_o is recursively checked against the new child nodes of b_n (in Figure 14c, case 1 would apply to the new SW child of b_n).

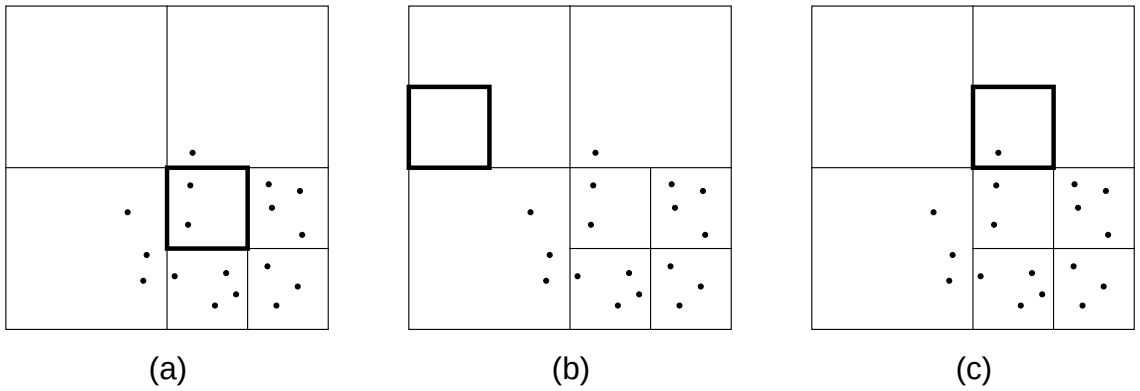

Figure 14: A simple PMR quadtree T_n consisting of points and the three cases that arise when merging with an existing quadtree T_o with our bulk-insertion algorithm: (a) A leaf node in T_o coincides with a nonleaf node or a nonempty leaf node in T_n, (b) a leaf node in T_o is contained in an empty leaf node in T_n, and (c) a leaf node in T_o is contained in a larger non-leaf node in T_n. Squares with a heavy border correspond to leaf nodes in T_o, but the objects in T_o are not shown.

6.2 Algorithm

Our merge algorithm is shown in Figure 15. The algorithm modifies procedures FLUSHNODES and FLUSHSUBTREETOMBI from Figure 10, while the actual merging is coordinated by procedure MERGESUBTREES. The parameter *oldTree* in the procedures is a reference to the old quadtree. The old quadtree is accessed in MERGESUBTREES by the functions CURLEAFNODE and CURLEAFOBJECT, which return the current node region and object, respectively, for the current leaf node item, and by the procedure NEXTLEAFNODE, which advances the current leaf node item to the next one in the order of Morton block values. Observe that two successive leaf node items can be two objects belonging to the same leaf

node. For simplicity of the presentation, we assume in Figure 15 that empty leaf nodes are not represented in the disk-based quadtree. Also, we do not explicitly test for the condition that the entire content of the existing quadtree has already been read, assuming instead that the current leaf node region is set to some special value when that happens so that it does not intersect any of the leaf nodes in the memory-resident quadtree. The three cases arising in merging enumerated above are represented in MERGESUBTREES. The first case triggers the first **do** loop, where objects in the old quadtree are inserted into the new memory-resident quadtree (which may cause node splits). The second case triggers the second **do** loop, where leaf node items are copied directly from the old quadtree and into the combined quadtree. The third case triggers an invocation of SPLIT, which splits the new leaf and distributes its content among the child nodes as appropriate. Procedure MERGESUBTREES will be invoked later on the child nodes. Since MERGESUBTREES is invoked on nodes in the new quadtree in top-down fashion, CURLEAFNODE(*oldTree*) is never larger than *node*, and the leaf node splitting (for case 3) ensures that, eventually, either case 1 or case 2 will apply to every leaf node in the old quadtree.

6.3 Discussion

One way of evaluating the efficiency of our bulk-insertion algorithm is to compare bulk-loading a quadtree from scratch on the combined data set to first bulk-loading the old data and then bulk-inserting the new data. From the standpoint of CPU cost, we believe that our algorithm is very efficient in this regard. Nevertheless, there is some overhead, mainly related to B-tree operations on the intermediate B-tree (i.e., writing it during bulk-loading and reading during bulk-insertion), as well as memory allocation and handling of nodes in the new quadtree that are also present in the old tree. However, the number of intersection tests, which are a major component of the CPU cost, would not be increased much over bulk-loading the combined data set. Furthermore, the bulk of the CPU cost of MERGESUBTREES is involved in updating the disk-resident combined quadtree and the memory-resident new quadtree, and accessing the disk-resident old quadtree, while other operations performed by it take little time if implemented efficiently (typically less than 5% of the total CPU cost of MERGESUBTREES in our tests). From the standpoint of I/O cost, performing both bulk-load and bulk-insert operations carries the overhead of writing out the intermediate quadtree (during bulk-loading) and reading it back in (during bulk-insertion), when compared to bulk-loading the combined data set. This can be expected to be partially offset by slightly lower I/O cost of sorting the two smaller data sets as opposed to the combined set.

In our quadtree merging algorithm, we chose to write out a new combined disk-resident quadtree. It would be easy to modify our algorithm to instead update the old disk-resident quadtree: 1) after inserting objects from the old quadtree into the new memory-resident quadtree, the corresponding B-tree entries would be deleted, 2) instead of the second **do** loop (where entries in the old quadtree are copied into the combined quadtree), we would look up the next B-tree entry that does not intersect *node*. Unfortunately, in the worst case, we would still need to read and modify every B-tree node. Furthermore, the B-tree packing technique discussed in Section 5.5 is less effective when adapted to handle updates of an existing B-tree. Thus, the overall I/O cost overhead is often higher than with our method due to worse storage utilization, in addition to the CPU cost incurred for updating the existing B-tree nodes. A further advantage of our approach over updating the old quadtree is that the old quadtree index can be used to answer incoming queries while the bulk-insertion is in progress, without the need for complex concurrency control mechanisms. Nevertheless, as we shall see in Section 9.2.7, this update-based variant is sometimes more efficient than our merge approach when the new data covers previously unoccupied regions in the existing quadtree.

A drawback of our quadtree merging approach is that it results in a quadtree structure that corresponds to first inserting all the new data and then the existing data (due to the INSERT invocations in the first **do** loop). Since the structure of a PMR quadtree depends on the insertion order, the resulting

```
procedure FLUSHNODES(node, p, oldTree) →
  if (not ISLEAF(node)) then
    MERGESUBTREES(node, oldTree)
    /* remainder of procedure is same as in Figure 10 */
  endif

procedure FLUSHSUBTREETOMBI(node, freeNode, oldTree) →
  MERGESUBTREES(node, oldTree)
  /* remainder of procedure is same as in Figure 10 */

procedure MERGESUBTREES(node, oldTree) →
  if (CONTAINS(node, CURLEAFNODE(oldTree))) then
    if (SIZE(node) = SIZE(CURLEAFNODE(oldTree)) and
        not (ISLEAF(node) and ISEMPTY(node))) then
      /* node regions are equal (see Figure 14a) */
      do
        INSERT(node, CURLEAFOBJECT(oldTree))
        NEXTLEAFNODE(oldTree)
      while (EQUALCOVERAGE(node, CURLEAFNODE(oldTree)))
    elseif (ISLEAF(node)) then
      if (ISEMPTY(node) then
        /* CURLEAFNODE(oldTree) is same size or smaller (see Figure 14b) */
        do
          MBIINSERT(CURLEAFNODE(oldTree))
          NEXTLEAFNODE(oldTree)
        while (CONTAINS(node, CURLEAFNODE(oldTree)))
      else
        /* CURLEAFNODE(oldTree) is smaller (see Figure 14c) */
        SPLIT(node)
      endif
    endif
  endif
```

Figure 15: Pseudo-code for quadtree merging.

structure may be different than when first inserting the existing data and then the new data. However, this should not be much of a concern, as the difference is usually slight: only a small percentage of the quadtree blocks will be split more in one tree than in the other. Another potential problem is that the size of the memory-resident quadtree (in terms of occupied memory) may increase during the merging, before any parts of it can be freed. To see this, let b_n be the non-empty leaf node in the new memory-resident quadtree with the smallest Morton code (among unflushed leaf nodes). Without merging, b_n would be the first leaf node to be flushed. Also, let b_o be the next leaf node in the old quadtree, and assume that the region of b_o intersects that of b_n. Before b_n can be flushed and its content freed from memory, the memory-resident quadtree can grow in two ways: 1) if the region of b_n is larger than that of b_o, then b_n is split, and 2) if b_o is non-empty, then its contents are inserted into the memory-resident quadtree. Since the numbers of objects in b_n and b_o are limited, the amount of memory consumed by these actions should not be very large. Furthermore, most or all the extra memory consumed is freed soon afterwards. Thus,

it should be sufficient to allow for only a small amount of extra memory to handle such cases and thus prevent a memory overflow situation.

7 Bulk-Loading PR Quadtrees

The bulk-loading method for quadtrees described in Section 5 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 [48] (or, more accurately, the bucket PR quadtree) instead of the PMR quadtree. In the PR quadtree (see Section 3.1), 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 [53]. Our description is in terms of a PR quadtree stored in an MBI (see Section 3.3), but can easily be adapted to any other representation. Thus, the quadtree blocks are represented with Morton block values.

7.1 Overview

When bulk-loading the PR quadtree, we assume that the data is sorted in Morton code order prior to being inserted, just as we do in our PMR quadtree bulk-loading method. However, rather than first building a pointer-based quadtree in main memory, 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. 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.

Figure 16 illustrates the insertion of a sequence of points 1–4 into the candidate list (in increasing order), and how the current leaf node region is expanded to encompass new points. In the figure, the square with a heavy border denotes the current leaf node (being built in memory), while the square with a broken border denotes the previous leaf node, i.e., the current leaf node prior to its last expansion. The most recently inserted point is denoted with an $\times$ symbol, while the other candidate points are shown as dots. Figure 16a shows what happens when a point is inserted into an empty candidate list: the current leaf node region is set to the smallest possible quadtree region around the point, i.e., of size 1 by 1. In Figures 16b and 16d, the inserted point is not contained in the current leaf node region. Thus, the current leaf node region is expanded so that it contains the new point. As we shall see below, the previous region may be needed later, so it must be remembered. In Figure 16c, the new point is contained in the current leaf node region, so no expansion takes place. Observe that the current leaf node region is always the smallest enclosing quadtree block containing the points in the candidate list.

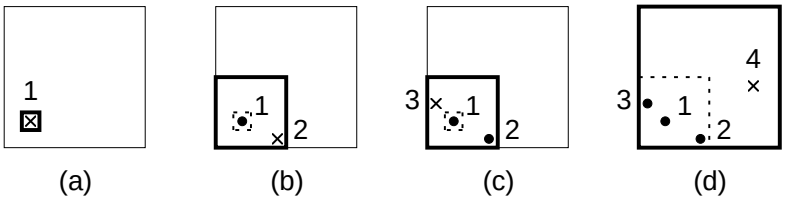

Figure 16: Example of insertions into candidate list, of points 1-4 (in order), demonstrating expansion of the current leaf node region (shown with heavy lines). The square with broken lines denotes the current leaf node region prior to its last expansion.

Figure 17 illustrates the conditions that lead to the construction of a new leaf node in the MBI. The shaded square in the figure denotes the last leaf node that was built. In this example, we assume a bucket

capacity of 8. Thus, in Figures 17a and 17b we have exceeded the bucket capacity, as the new point leads to the candidate list containing nine points. In Figure 17a, the new point is contained in the current leaf node region. Since we have an overflow in that region, we must use the previous leaf node region (shown with broken lines) to construct a new leaf node, containing the three points inside it. In Figure 17b, on the other hand, the new point is outside the current leaf node region, so we can build a new leaf node containing the eight points in the current leaf node region. In Figure 17c, we do not have an overflow, so the current leaf node region gets expanded to contain the new point. However, this results in the current leaf node region overlapping the previously constructed leaf node. This is not permitted, so we construct a new leaf node for the six points inside the current leaf node region as it was prior to the expansion. For all three cases, the points in the candidate list that are outside the newly constructed leaf node are reinserted into the candidate list (which is first emptied), in the same manner as described above (i.e., recall Figure 16). However, for the case illustrated in Figure 17a, we could optimize the process slightly by immediately constructing leaf nodes for the two points above and the two points to the right of the new leaf node (i.e., the NW and SE quadrants of the current leaf node region, which is denoted by a heavy boundary).

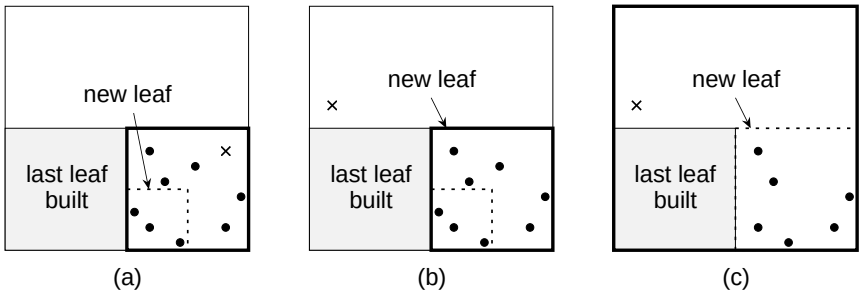

Figure 17: Conditions for constructing a new leaf node where the most recently inserted point is denoted by x: (a) candidate list overflows and the new point is in the current leaf node region, (b) candidate list overflows and the new point is not in the current leaf node region, and (c) expansion of the current leaf node region causes overlap of the leaf node that was last built.

When a new leaf node is constructed for a set of points *A*, the leaf node region is the smallest one covering the points. However, the PR quadtree is defined so that the leaf node regions are maximal, i.e., as large as possible without intersecting other leaf nodes. Figure 18 illustrates the case where the leaf node region for a set of points is not maximal. Thus, the leaf node region must be expanded until it is the largest possible region that does not overlap any points not in *A*. As shown in Figure 18, we use the last leaf node to be constructed and the most recently inserted point in the candidate list to guide how far to expand the leaf node region. In the figure, we expand the leaf node region once, to the square drawn with heavy lines. If we expanded it once more, it would overlap both the last leaf node and the most recently inserted point. However, overlap with either one suffices to halt the expansion.

7.2 Algorithm

The algorithm is shown in detail in Figure 19. For simplicity, we assume in the figure that there are never more than *c* points in quadtree blocks of the minimum size (which have a side length of 1). The algorithm can easily be extended to handle the extreme case when this assumption does not hold. The algorithm employs the following global variables to maintain its state:

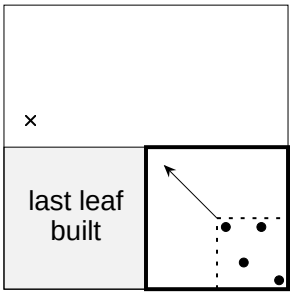

Figure 18: Expansion of the current leaf node region to cover maximal area. The most recently inserted point is denoted by x.

- *candidateList*: an array of up to $c + 1$ candidate points for the current block (always occurring in Morton order),
- *listLen*: the number of points in *candidateList*,
- *currentLeafBlock*: the smallest quadtree block enclosing points on *candidateList* (i.e., the square with heavy border in Figures 16 and 17),
- *smallerLeafBlock*: a smaller block than *currentLeafBlock* containing a subset of the points in *candidateList* (i.e., the broken square in Figures 16 and 17),
- *smallerCount*: the number of points in *smallerLeafBlock*,
- *lastLeafBlock*: the last quadtree leaf block inserted into the MBI (i.e., the shaded square in Figures 17 and 18),
- *leafCount*: the number of quadtree leaves that have been constructed (i.e., inserted into the MBI).

The build process is initialized by setting both *listLen* and *leafCount* to 0. Once procedure INSERTPOINT has been invoked for all points in the data set, the candidate list will contain up to *c* points. To build the final leaf with those points, we invoke "BUILDLEAF(*listLen*, *currentLeafBlock*, *NIL*)", where *NIL* indicates a null point value.

The algorithm uses several functions for manipulating and testing Morton block values: MINIMUMENCLOSING, EXPANDTOCONTAIN, and CONTAINS. These are most efficiently implemented if the Morton codes of the inserted points are computed in advance (this need only be done once for each inserted point). In this case, they merely involve simple bit manipulations and comparisons. The procedure MBIINSERT inserts an item into the Morton Block Index. Procedure INSERTPOINT implements the control structure of the algorithm. An inserted point is added to the *candidateList* array. If it is the sole element, *currentLeafBlock* is initialized to only contain the point. If the number of points in *candidateList* exceeds *c*, we build a new leaf by invoking BUILDLEAF. The number of points in the new leaf depends on whether or not the inserted point is contained in the current leaf block (e.g., Figures 17a and 17b, respectively). If the number of points in the candidate list is between 2 and *c*, we test whether the inserted point is contained in the current leaf block. If not, we remember the current leaf block (which may be needed later), and extend the region of the current leaf block to include the inserted point (e.g., Figure 16d). If extending the current leaf block makes it overlap the previously created leaf block (e.g., Figure 17c), we invoke BUILDLEAF using the previous value of *currentLeafBlock*. BUILDLEAF must start with enlarging the leaf block region as much as possible, in order to adhere to the definition of the

```
procedure INSERTPOINT(point) →
  listLen ← listLen+1
  candidateList[listLen] ← point
  if (listLen = 1) then
    currentLeafBlock ← MINIMUMENCLOSING(point)
  elseif (listLen > c) then
    if (CONTAINS(currentLeafBlock, point)) then
      BUILDLEAF(smallerCount, smallerLeafBlock, point) /* see Figure 17a */
    else
      BUILDLEAF(c, currentLeafBlock, point) /* see Figure 17b */
    endif
  elseif (not CONTAINS(currentLeafBlock, point)) then
    smallerLeafBlock ← currentLeafBlock
    smallerCount ← listLen-1
    currentLeafBlock ← EXPANDTOCONTAIN(currentLeafBlock, point)
    if (leafCount > 0 and CONTAINS(currentLeafBlock, lastLeafBlock)) then
      BUILDLEAF(smallerCount, smallerLeafBlock, point) /* see Figure 17c */
    endif
  endif

procedure BUILDLEAF(pointCount, leafBlock, point) →
  /* make leafBlock as large as possible (see Figure 18) */
  parentBlock ← PARENT(leafBlock)
  while (not CONTAINS(parentBlock, point) and
          not (leafCount > 0 and CONTAINS(parentBlock, lastLeafBlock))) do
    leafBlock ← parentBlock
    parentBlock ← PARENT(parentBlock)
  endwhile
  /* insert first pointCount candidates into MBI */
  for (i = 1..pointCount) do
    MBIINSERT(leafBlock, candidateList[i])
  endfor
  lastLeafBlock ← leafBlock
  leafCount ← leafCount+1
  /* recursively reinsert remaining points */
  oldListLen ← listLen
  listLen ← 0
  for (i = pointCount+1..oldListLen) do
    INSERTPOINT(candidateList[i])
  endfor
```

Figure 19: Pseudo-code for the PR quadtree bulk-loading algorithm.

PR quadtree (e.g., Figure 18). Next, it inserts the requested points into the MBI, while recursively invoking INSERTPOINT on the remaining points in the candidate list. This is necessary in order to construct the proper value for *currentLeafBlock*.

The above algorithm can be extended to handle bulk-insertions into an existing PR quadtree, by using a merge process analogous to that for the PMR quadtree bulk-insertion algorithm (see Section 6). As INSERTPOINT generates new leaf nodes, the contents of some nodes in the existing PR quadtree have to be inserted into the candidate list, while existing leaf nodes not containing any of the new points can be copied directly into the new PR quadtree (as in MERGESUBTREES in Figure 15). In addition, INSERTPOINT and BUILDLEAF must make sure that *currentLeafBlock* and *leafBlock*, respectively, are not extended so much as to contain the next leaf node in the existing PR quadtree (in the same way as they prevent the leaf blocks from containing *lastLeafBlock*).

8 Analytic Observations

In this section we make some observations about the execution cost of our PMR quadtree bulk-loading algorithm. Many of the considerations apply to the PR quadtree bulk-loading algorithm as well. The discussion is for the most part informal, and is meant to give insight into general trends, rather than being a rigorous treatment. Our experiments suggest that I/O cost and CPU cost both contribute significantly to the total execution cost (although the I/O cost contribution is usually higher). Therefore, we discuss each separately below.

8.1 I/O Cost

The performance of bulk-loading algorithms is frequently characterized by their I/O cost [8, 13]. Such an analysis seeks to evaluate the number of I/O operations (reads and writes) performed by the algorithm, each affecting a disk block that contains a maximum of B records. The algorithm is assumed to use an internal memory buffer accommodating M records, and the number of data records to load is N. Below, we make some observations on the I/O cost of our bulk-loading algorithm when used with a linear quadtree such as the Morton Block Index (MBI).

Besides the cost of reading the actual data file, the I/O cost of our bulk-loading method has two components: sorting I/O cost and quadtree I/O cost. In the case of the MBI, the quadtree I/O is really B-tree I/O, so this is the designation we use below. Before we proceed, we must point out that the values of B, M, and N for each of these components is different. First, the values of B for the B-tree are slightly lower than for sorting (assuming a constant disk page size in bytes), since each entry in the B-tree occupies somewhat more space. They differ by a constant factor, however, so this does not affect asymptotic results. Second, if the sorting phase and the tree building phase are executed simultaneously, with the result of the first pipelined to the second, both will require their own internal memory buffer. Thus, each component really has a buffer of $M/2$ records, assuming we allocate the same amount to each component. Also, as with B, the values of M for the two components are different due to different record sizes (assuming the same buffer size in bytes). For both of these issues, the difference in M is constant, and thus can be ignored. Third, the value of N is generally higher for the B-tree than for sorting, since the former represents the number of q-objects rather than objects. In addition, empty quadtree leaf blocks may be represented in the B-tree (recall from Section 3.3 that this is optional). Neither of these factors can be ignored. For the present, we will use N' to denote the number of entries in the B-tree. Later, we attempt to relate N' to N, the number of data objects.

Sorting N items in external memory can be done in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O operations (see Section 5.4). Reinsertions may add to this cost. Recall from Section 5.3 that the total number of insertions into the quadtree (original and reinsertions) for object o is no more than $q + a'$, where q is the number of corresponding q-objects and a' is the number of ancestors of the leaf nodes containing the q-objects, not counting those ancestors that completely enclose the object. Since a' is no higher than wq, where w is the maximum height of the quadtree, the total number of insertions is at most wN_q, where N_q denotes the

number of q-objects. However, since the q-objects for an object typically share most of their ancestors, we can expect the total number of reinsertions to be $O(N_q)$. As we outlined in Section 5.4.2, the I/O cost of sorting N objects and reinserting $O(N_q)$ objects is $O(sN_q/B \log_{M/B} \frac{N_q}{B})$, where s is less than 2 for all practical values of N, M, and B.

In our PMR quadtree bulk-loading approach, each B-tree node in the MBI is written only once and never read, due to the use of B-tree packing (see Section 5.5). This means that the B-tree I/O cost is between N'/B and $2N'/B$, depending on the split fraction, and thus $O(N'/B)$.

The overall I/O cost of our bulk-loading algorithm is therefore $O(\frac{N'}{B} + s\frac{N_q}{B} \log_{M/B} \frac{N_q}{B})$. Below, we argue that reinsertions are unlikely to occur, so the presence of s and N_q (instead of N) in the formula generally greatly overestimates the sorting cost. Furthermore, the values of N' and N_q are often on the same order as N. Therefore, in many cases, the actual I/O cost of the bulk-loading algorithm is about the same as that of external sorting, i.e., $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$.

8.1.1 When are Reinsertions Needed?

As we saw above, the sorting cost can increase substantially in the presence of reinsertions. However, reinsertions only occur if the flushing algorithm fails to free any memory. The informal analysis below, although simplistic, suggests that this will rarely happen.

Recall that the flushing algorithm is unable to free any memory if all the objects stored in the pointer-based quadtree intersect the boundary (referred to as *flushing boundary* below) between flushed and unflushed nodes; e.g., the boundary of the striped region in Figure 9. This condition never arises if the data objects are points and is unlikely to occur if the "space" between adjacent data objects is generally larger than their size. In general, however, we must make some assumptions about the distribution of the locations and sizes of non-point objects to be able to estimate the number of objects that intersect the flushing boundary. We will make the simplifying assumption that the data objects are all of the same size, and are equally spaced in a non-overlapping manner so that they cover the entire data space. In other words, for a two-dimensional object, the bounding rectangle is approximately a square with area $\frac{L^2}{N}$, and thus side lengths $\frac{L}{\sqrt{N}}$, where L is the side length of the square-shaped data space. The length of the flushing boundary is at most $2L$, since starting from its top-left corner, the boundary is monotonically non-decreasing in the x axis and non-increasing in the y axis (refer to Figure 9 for an example)[9]. Given the assumptions above, the number of objects intersected by the flushing boundary is at most $\frac{2L}{L/\sqrt{N}} = 2\sqrt{N}$, since the boundary is piecewise linear. For that many objects, the quadtree buffer would be full if $M \leq 2\sqrt{N}$. Put another way, given a buffer size of M, the buffer can be expected to never fill if $N \leq M^2/4$. For example, with a buffer capacity of 10,000 objects, we can expect the buffer never to fill for a data file of up to 50 million objects. If each object occupies 50 bytes, these numbers correspond to a buffer size of about 500K and a data file size of about 2.3GB.

In general, for d dimensions, the object's bounding hyper-rectangles (which are nearly hyper-cubes in shape) have a volume of about L^d/N, so each of their $d-1$ dimensional faces has a $d-1$ dimensional volume of approximately $(L^d/N)^{\frac{d-1}{d}} = L^{d-1}/N^{\frac{d-1}{d}}$. The flushing boundary has a $d-1$ dimensional volume of at most dL^{d-1}, so the number of objects intersected by it can be expected to be less than $\frac{dL^{d-1}}{L^{d-1}/N^{\frac{d-1}{d}}} = dN^{\frac{d-1}{d}}$. Unfortunately, if N is smaller than d^d, this value is larger than N. However, for the relatively low-dimensional spaces for which quadtrees are practical, N is typically much larger than d^d so $dN^{\frac{d-1}{d}}$ is smaller than N. Furthermore, it is not common to be working with non-point objects in spaces of higher dimensionality than 3. For three-dimensional space, we can expect a buffer of size M

[9] It is possible to show that the maximum length is even less than this ($3L/2$) and the average length is still less (L), but the bound $2L$ suffices for our purposes.

to never fill if $N \leq (M/3)^{3/2}$. For example, a buffer capacity of 10,000 objects can be expected to be enough to handle data files of up to approximately 190,000 objects (about 9MB for objects of 50 bytes each). Although this may not seem as dramatic as in the two-dimensional case, the difference between N and M is still more than an order of magnitude.

8.1.2 Relationship between N, N_q, and N'

The I/O cost of the bulk-loading algorithm given above was in terms of N_q and N', the number of q-objects and the number of B-tree items, respectively. In order to get a better picture of the I/O cost, it is useful to establish the relationship between the three quantities N, N_q, and N'. In this section we explore this issue.

First, consider N, the number of objects, and N_q, the number of q-objects. Note that for points, $N_q = N$. For non-point objects, the value of N_q depends on many factors, including 1) the splitting threshold, 2) the relative sizes of objects, 3) how closely clustered the objects are, 4) the complexity of the boundaries of objects, and 5) the degree of overlap. As an extreme example, if all the objects were squares (hypercubes for $d > 2$) that covered the entire data space, then the space would be maximally partitioned into the smallest allowable cells. In other words, we would get 2^{wd} leaf nodes, where w is the maximum height of the quadtree, assuming N is at least $w + t$, where t is the splitting threshold value. Thus, each object is broken up into 2^{wd} q-objects, and $N_q = 2^{wd}N$. As another example, if the data objects are square-shaped (cube- or hypercube-shaped for $d > 2$), all of the same size, the largest number of q-objects for a square is 6, or $2 \cdot 3^{d-1}$ in general (assuming $t \geq 2^d$); the average number will depend on t. In this example, the ratio between N and N_q is still exponential in d. However, non-point data is rarely used in spaces with dimensionality above 3.

As to the relationship between N_q and N', the difference between the two is the number of empty quadtree leaf nodes, if we choose to represent them in the B-tree. Unfortunately, there can be a large number of empty leaf nodes in the tree. As an extreme example, suppose that all the objects lie in a single cell of the minimum size. This would cause node splits at all levels of the tree until we have all the objects in a single leaf node at the lowest level. Thus, given a two-dimensional quadtree with a maximum depth of w, we would have $3w$ empty leaf nodes for the single non-empty leaf node. We can extend this example to a tree of k non-empty leaf nodes having as many as $3(w - \lfloor \log_4 k \rfloor)k$ empty leaf nodes[10], or in general for a d-dimensional quadtree, $(2^d - 1)(w - \lfloor \log_{2^d} k \rfloor)k$ empty leaf nodes. In quadtrees that give rise to such a high number of empty leaf nodes, most internal nodes have $2^d - 1$ empty leaf nodes as child nodes while only one child is either a non-empty leaf node or an internal node. Thus, such quadtrees are rather contrived and unlikely to actually occur. A more reasonable assumption is that for the majority of quadtree nonleaf nodes, at least two child nodes are non-empty. Given this assumption, an upper bound of about 2^{d+1} empty leaf nodes for each non-empty leaf node can be established. Since the number of empty leaves tends to grow sharply with d, it is inadvisable to store empty quadtree nodes in the B-tree for quadtrees of dimension more than 3 or 4.

It is interesting to consider the values of N_q and N' relative to N for actual data sets. In Section 9 we use six data sets consisting of non-overlapping two-dimensional line segment data, three of which are real-world data and three of which are synthetic. With a splitting threshold of 8, the value of N_q was at most about $2N$ for the real-world data sets, while it was about $2.63N$ for the synthetic data sets. The number of empty leaf nodes was rather small, ranging from 2.2% to 4.7% of N for the real-world data sets and 3.2% to 3.8% for the synthetic ones. With a splitting threshold of 32, the value of N_q ranged from $1.3N$ to $1.6N$, while the number of empty leaf nodes was negligible. In the experiments, we also used a real-world data set comprising two-dimensional polygons representing census tracts in the US. The

[10] This is realized by having k trees with one non-empty leaf node, all of height $w - \lfloor \log_4 k \rfloor$, and a complete quadtree of height $\lfloor \log_4 k \rfloor$ down to the roots of these k trees.

spatial extent of these polygons had a wide range, the polygon objects touched each other's boundaries, and their boundaries were often very complex (up to 3700 points per polygon, with an average of about 40). Thus, this data set represents an extreme in the complexity of non-overlapping two-dimensional data. With a splitting threshold of 8, both N_q and N' were about $4N$, while with a splitting threshold of 32 they were less than $2N$ (more precisely, about $1.9N$). Thus, the values obtained for N_q and N' were still relatively close to the value of N, at least for the larger splitting threshold. Finally, we experimented with highly overlapping synthetic line segment data. Not surprisingly, the number of q-objects for each object is very high for such data. Even with a relatively large splitting threshold of 32, the value of N_q was about $110N$. This strongly suggests that quadtrees are not very suitable for such data, but the performance of other spatial index structures will also degrade for such data.

8.2 CPU Cost

Three factors contribute to the CPU cost of the algorithm: 1) sorting the objects, 2) building the pointer-based quadtree, and 3) building the B-tree of the MBI. For each of these factors, the techniques that we outlined are very efficient.

The CPU cost of the external merge sorting algorithm given in Section 5.4.1 is roughly proportional to the number of comparison operations. Using the symbols N, M, and B as described in Section 5.4, the average number of comparison operations per object when constructing the initial runs is $O(\log M)$. In each merge step, we need $O(\log \frac{M}{B})$ comparisons for each object on average since at most M/B runs are merged each time. Thus, recalling that the number of merge steps is $O(\log_{M/B} \frac{N}{M}) = O(\frac{\log(N/M)}{\log(M/B)})$, the overall number of comparisons per object on average is $O(\log M + \frac{\log(N/M)}{\log(M/B)} \log \frac{M}{B}) = O(\log M + \log \frac{N}{M}) = O(\log N)$, and the total cost is $O(N \log N)$, which is optimal. Even in the presence of reinsertions (Section 5.4.2), sorting remains nearly optimal.

Assuming for the moment that the original insertion algorithm is used instead of our improved one, the total cost of building the pointer-based quadtree is roughly proportional to the number of intersection tests. Recall that the intersection tests are needed to determine whether an object should be inserted into a certain node. If o_q is a q-object of object o that intersects a leaf node n, the number of intersection tests on o is at least $2^d \cdot D_n$, where D_n is the depth of n. Thus, in the worst case, the total number of intersection tests needed on o is $2^d \cdot D_{\max}$ times the number of q-objects for o. To analyze this further, we resort to a gross simplification: assume that the objects are non-overlapping equal-sized squares in two dimensions, and that they are uniformly distributed over the data space. In this simple scenario, the number of q-objects for an object is $O(1)$, while the number of empty leaf nodes tends to be very low. Thus, the expected number of leaf nodes (and thus all nodes) is roughly proportional to N. Since the objects are uniformly distributed, the leaf nodes will tend to be at a similar depth in the tree, so the average height is approximately proportional to $\log N$. Therefore, the total number of intersection tests is $O(N \log N)$[11]. Note that in our improved PMR quadtree insertion algorithm, the total number of intersection tests is typically much smaller, and can potentially be as small as $O(N)$. Nevertheless, some work is still expended in traversing the pointer-based quadtree down to the leaf level for each object.

When traversing the pointer-based quadtree during flushing, most of the nodes visited are deleted from the tree, and thus are never encountered during subsequent flushing operations. The visited nodes that are retained (or at least a similar number of nodes) are also visited by the insertion operation that initiated the flushing, so the cost of visiting them is accounted for in the cost of the insertion operation. Thus, the total additional cost of tree traversal during flushing is proportional to the number of quadtree nodes ($O(N)$ in the simplified scenario above). During flushing, some work is also expended for every

[11] Of course, for arbitrary dimensions, a 2^d factor would be involved. However, recall that the quadtree is only used for relatively modest values of d.

q-object in the flushed nodes. However, this work is accounted for in the cost of building the B-tree.

In the B-tree packing algorithm introduced in Section 5.5, the CPU cost is proportional to the number of inserted items. To see this, observe that procedures PACKINSERT and PACKSPLIT in Figure 13 both expend a constant amount of work for each invocation (if the split fraction is not 100%, the cost of PACKSPLIT is proportional to B, but the amortized cost per object in the split node is still constant). PACKINSERT is only invoked once per item, while PACKSPLIT is invoked h times for an item that eventually is stored in a node at height h. If the total number of B-tree items is N, then the number of items at height h is about N/B^h, where B is the number of items in each B-tree node. Therefore, the number of invocations for items at height h is approximately hN/B^h. Thus, the total number of invocations of PACKSPLIT is roughly $\sum_{h=1}^{h_{\max}} hN/B^h \leq N\sum_{h=1}^{\infty} h/B^h = N/(B-2+1/B)$, which is $O(N)$ for $B \geq 3$. Hence, the total CPU cost of B-tree packing is $O(N)$.

To summarize, we saw that the asymptotic CPU cost was $O(N \log N)$ for sorting the objects, $O(N \log N)$ for constructing the quadtree in memory (given our simplifying assumptions), and $O(N)$ for building the B-tree. Thus, we see that in an ideal situation (i.e., if the data distribution is not too skewed), we can expect the total CPU cost of our bulk-loading algorithm to be approximately $O(N \log N)$.

9 Empirical Results

9.1 Experimental Setup

We implemented the techniques that we presented in Sections 5 and 7 in C++ within an existing linear quadtree testbed (described in Section 3.3). Our quadtree implementation has been highly tuned for efficiency, but this primarily benefits dynamic PMR quadtree insertions (i.e., when inserting directly into the MBI). Thus, the speedup due to bulk-loading would be even greater than we show had we used a less tuned implementation. This is partly the reason why we obtained lower speedup than reported in [26]. 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 machine, 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 execution times reflect the true cost of I/O, which would otherwise be partially obscured by the file caching mechanism of the operating system. The use of raw disk partitions is another reason we obtained lower speedup than in [26], since the reduction in CPU cost is much greater than the reduction in I/O cost. The maximum depth of the quadtree was set to 16 in most of the experiments, and the splitting threshold in the PMR quadtree (bucket capacity in the PR quadtree) to 8. Larger splitting thresholds make our PMR quadtree bulk-loading approach even more attractive. However, as 8 is a commonly used splitting threshold, this is the value we used. B-tree node size was set to 4KB, while node capacity varied between 50 and 400 entries, depending on the experiment.

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 our PMR quadtree bulk-loading algorithm, we limited the space occupied by the pointer-based quadtree to 128K. The flushing algorithm was always able to free substantial amounts of memory (typically over 90% but never less than 55%), except in experiments explicitly designed to make it fail. In all other experiments, this level of buffering proved more than adequate and a larger buffer did not improve performance. The sort buffer was limited to 512K. A sort buffer size of 256K increased running time only slightly (typically less than 3% of the total time). For the B-tree, we explored the effect of varying the buffer size, buffering from 256 B-tree nodes (occupying 1MB) up to the entire B-tree. For the bulk-loading methods, however, only one B-tree node at each level needed to be buffered, as described in Section 5.5.

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. As a result, the size of the 99% confidence interval for each experiment was usually less than 0.4% of the average value, and never more than about 1%. In particular, the confidence intervals are always smaller than the differences between any two loading methods being compared.

9.2 Findings

Below, we detail the results of a number of experiments which show the performance of the two bulk-loading techniques presented in this paper, for PMR and PR quadtrees (Sections 5 and 7, respectively), as well as our technique for improving the performance of PMR quadtree insertions (Section 4). With the exception of Section 9.2.5, the improved insertion method is used in all experiments involving the PMR quadtree, both in our bulk-loading algorithm and when performing dynamic insertions (i.e., updating the MBI directly). Unless otherwise specified, the experiments in this section use the PMR quadtree and the bulk-loading algorithm presented in Section 5.

The remainder of this section is organized as follows: In Section 9.2.1 we go into considerable detail on bulk-loading two-dimensional line segment data, as well as describe the specifics of the PMR quadtree loading methods used in these and subsequent experiments. In Section 9.2.2 we repeat the same experiments in SAND, our prototype spatial database system, in order to examine the effects of using the object table approach. In Sections 9.2.3 and 9.2.4 we show how well our method does with other types of data, multidimensional points and two-dimensional polygons, again using SAND. The performance of the PR quadtree bulk-loading algorithm (for multidimensional points) is also presented in Section 9.2.3, and compared with using the PMR quadtree bulk-loading algorithm. In Section 9.2.5 we investigate how much our improved PMR quadtree insertion algorithm improves the performance of the PMR quadtree bulk-loading algorithm and of dynamic insertions. In Section 9.2.6, we study the performance of the algorithm when no node can be flushed and reinsert freeing must be used. In Section 9.2.7 we examine how well our bulk-insertion algorithm for PMR quadtrees performs. In Section 9.2.8, we establish how our bulk-loading algorithm compares to two bulk-loading techniques for R-trees. Finally, in Section 9.2.9 we summarize the conclusions drawn from our experiments.

9.2.1 2D Line Segment Data

In the first set of experiments, we used two-dimensional line segment data, both real-world and synthetic. In these experiments, we stored the actual coordinate values of the line segments in the quadtree. The real-world data consists of three data sets from the TIGER/Line File [15]. The first two contain all line segment data — roads, rail lines, rivers, etc. — for Washington, DC and Prince George's County, MD, abbreviated below as "DC" and "PG". The third contains roads in the entire Washington, DC metro area, abbreviated "Roads". The synthetic data sets were constructed by generating random infinite lines in a manner that is independent of translation and scaling of the coordinate system [37]. 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. Using these data sets enables us to get a feel for how the quadtree loading methods scale up with map size on data sets with similar characteristics.

Table 1 provides details on the six line segment maps: the number of line segments, the average number of q-edges per line segment, the file size of the input files (in KB), and the minimum and maximum number of nodes in the MBI B-trees representing the resulting PMR quadtrees. Recall that a q-edge is a piece of a line segment that intersects a leaf block. The average number of q-edges per line segment

is in some sense a measure of the complexity of the data set, and a sparse data set will tend to have a lower average. The number of items in the resulting B-tree is equal to the number of q-edges plus the number of white nodes. Notice the large discrepancy in the B-tree sizes, reflecting the different storage utilizations achieved by the different tree loading methods. In the smallest trees, the storage utilization is nearly 100%. In the trees built with the dynamic PMR quadtree insertion method, the storage utilization ranged from 65% to 69%, and thus these trees were about 45% larger than the smallest trees.

Data set	Number of line segments	Avg. q-edges per segment	File size (KB)	MBI B-tree size (nodes)	
				Min	Max
DC	19,185	2.08	384	301	532
PG	59,551	1.86	1176	843	1529
Roads	200,482	1.76	3928	2691	4859
Rand64K	64,000	2.61	1264	1259	2152
Rand128K	128,000	2.62	2512	2525	4322
Rand260K	260,000	2.63	5088	5146	8674

Table 1: Details on line segment maps.

Table 2 summarizes configurations used for loading the PMR quadtree in the experiments. Three of them use dynamic quadtree insertion (i.e., updating the MBI directly) with varying levels of buffering in the MBI B-tree (denoted "BB-L", "BB-M", and "BB-S"), while two use our quadtree buffering bulk-loading method (denoted "QB-75" and "QB-100"). In one of the B-tree buffering configurations, "BB-S", we sorted the objects in Z-order based on their centroids prior to insertion into the quadtree. This has the effect of localizing insertions into the B-tree within the B-tree nodes storing the largest existing Morton code values, thus making it unlikely that a node is discarded from the buffer before it is needed again for insertions. Thus, the sorting ensures that the best use is made of limited buffer space. The drawback is that the storage utilization tends to be poor, typically about 20% worse than with unsorted insertions. Since deletions occur in the B-tree and insertions do not arrive strictly in key order, the regular B-tree packing algorithm could not be used. When we adapted the B-tree packing approach to handle slightly out-of-order insertions (see Section 5.5), and set it to yield storage utilization similar to that of unsorted insertions, the speedup was at best only slight. Nevertheless, we do not make use of this in our experiments, since it has the undesirable property of causing underfull nodes. For quadtree buffering, the B-tree packing algorithm (see Section 5.5) was set to yield approximately 75% ("QB-75") and 100% ("QB-100") storage utilization. In this experiment, as well as most of the others, we used the distribution sort algorithm mentioned in Section 5.4.

Method	B-tree buffering	Quadtree buffering	Sorting
BB-L	yes (unlimited)	no	no
BB-M	yes (1024 nodes)	no	no
BB-S	yes (256 nodes)	no	yes
QB-75	limited	yes ($\approx$ 75% B-tree storage utilization)	yes
QB-100	limited	yes ($\approx$ 100% B-tree storage utilization)	yes

Table 2: Summary of PMR quadtree loading methods used in experiments.

Table 3 shows the execution time for loading PMR quadtrees for the six data sets using the five loading methods. Figure 20 presents this data in a bar chart, where the execution times are adjusted for map

size; i.e., they reflect the average cost per 10,000 inserted line segments. Two conclusions are immediately obvious from this set of experiments. First, the large difference between "QB-75" and "BB-L", which both write each B-tree block only once ("QB-75" due to B-tree packing and "BB-L" due to unlimited B-tree node buffering) and have a similar B-tree storage utilization, shows clearly that quadtree buffering achieves large savings in CPU cost. Second, the dramatic increase in execution time between "BB-S" and "BB-M", in spite of the latter using four times as large a B-tree buffer, demonstrates plainly that unsorted insertions render buffering ineffective, especially as the size of the resulting B-tree grows with respect to the buffer size. The reason why the execution time of "BB-M" is lower for the real-world data sets than the synthetic ones is that the real-world data sets have some degree of spatial clustering, while the synthetic data sets do not. The cost of sorting in "BB-S" is clearly more than offset by the saving in B-tree I/O, even though the storage utilization in the B-tree becomes somewhat worse. Within the same loading method, the average cost tends to increase with increased map size. This is most likely caused by increased average depth of quadtree leaf nodes, which leads to a higher average quadtree traversal cost and more intersection tests on the average for each object. The rate of increase is smaller for quadtree buffering ("QB-75" and "QB-100"), reflecting the fact that quadtree traversals are more expensive in the MBI than in the pointer-based quadtree used in quadtree buffering. Curiously, the average cost for Roads is smaller for all five loading methods than that of R64K, even though the size of the R64K data set is smaller, and so is the average depth of leaf nodes in the resulting quadtree (8.53 for R64K vs. 9.24 for Roads). The reason for this appears to be primarily the larger average number of q-edges per inserted line segment for the R64K data set (see Table 1).

Data set	BB-L	BB-M	BB-S	QB-75	QB-100
DC	12.24	14.62	11.87	4.47	3.68
PG	35.62	71.49	37.15	13.80	11.53
Roads	120.78	221.55	134.38	46.14	38.92
R64K	52.49	136.18	56.07	19.37	16.04
R128K	109.41	349.48	116.62	39.47	32.85
R260K	229.31	853.34	254.58	82.31	68.81

Table 3: Execution times (in seconds) for building quadtrees for the six data sets.

A better representation of the experiment results for comparing the five different loading methods is shown in Figure 21. The figure shows the speedup of "QB-100", quadtree buffering with nearly 100% B-tree storage utilization, compared to the other four methods. Compared to "BB-L" and "BB-S", the speedup of "QB-100" is by a factor of between three and four, and the speedup increases with the size of the data set. Compared to "BB-M", the speedup is by a factor of at least four, and up to over 12 when "BB-M" performs the most B-tree I/O. Overall, "QB-75" was about 20% slower than "QB-100", which was to be expected since the MBI B-tree produced by "QB-75" is about 33% larger.

The proportion of the execution time spent on I/O operations is shown in Figure 22. We obtained these numbers by recording the I/O operations performed while building a PMR quadtree, including reading the data, and then measuring the execution time needed to perform the I/O operations themselves. For the loading methods that use sorting, we include the I/O operations executed by the sort process. For B-tree buffering, except for "BB-M", the relative I/O cost is small, or only about 20-30%, compared to between 65% and 75% for quadtree buffering. This shows that the savings in execution time yielded by quadtree buffering are, for the most part, caused by reduced CPU cost (the time for performing I/O is only 1.3 to 2.9 seconds per 10,000 insertions for all but "BB-M"). For "BB-M", the proportion of time spent on I/O gradually increases with larger data sizes as B-tree buffering becomes less effective on unsorted

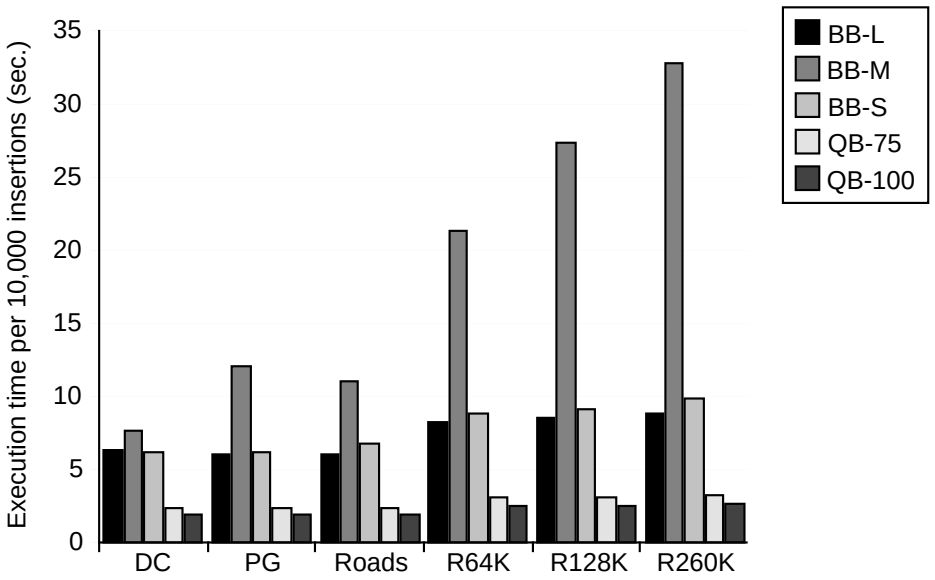

Figure 20: Execution time per 10,000 line segments for building quadtrees for the six data sets.

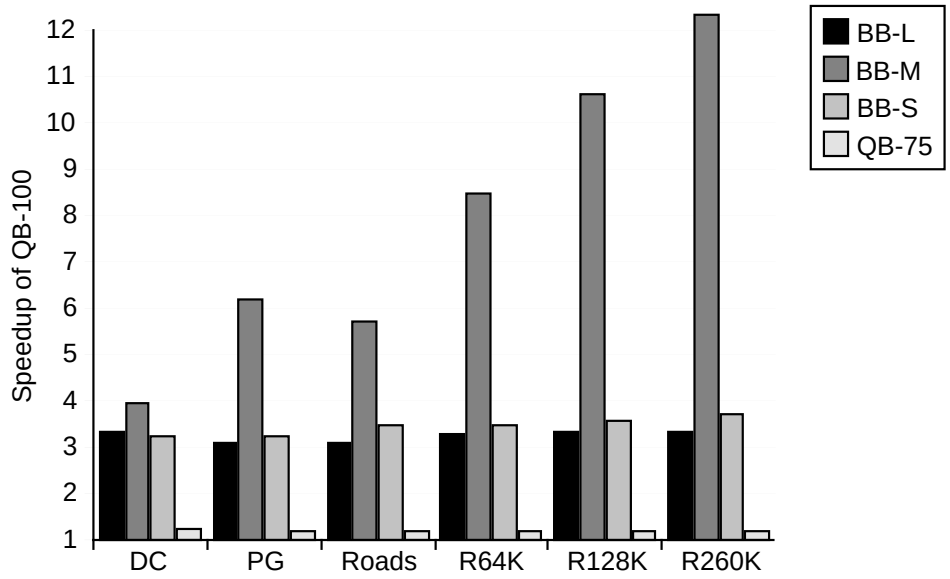

Figure 21: Speedup of "QB-100" compared to the other four loading methods for line segment data.

data.

9.2.2 Line Segment Data in SAND

In the first set of experiments, we stored the actual geometry of the objects in the PMR quadtree. As mentioned in Section 3.3, our quadtree implementation also allows storing the geometry outside the quadtree. The second set of experiments was run within SAND, our spatial database prototype, using the same data. This time, we stored only tuple IDs for the spatial objects in the quadtree, rather than the geome-

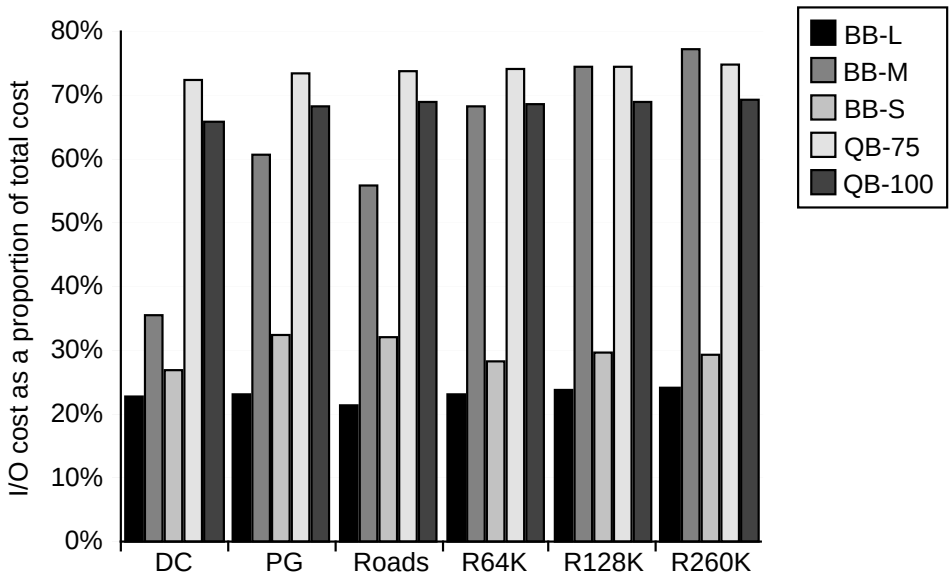

Figure 22: Proportion of execution time spent on I/O operations for the five loading methods for line segment data.

try itself. Storing the geometry in the quadtree with SAND yields results similar to that of our previous experiments, the difference being that SAND also must store the tuple ID, thereby making for slightly larger B-tree entries and lower fan-out. An additional difference is that in the experiments above, we used 4-byte integers for the coordinate values of the line segments, while SAND uses 8-byte floating point numbers for coordinate values. For this set of experiments, we used the configurations "BB-L", "BB-S", and "QB-100", described in Table 2. In keeping with the modest buffering in the latter two, we only buffered 128 of the most recently used disk pages for the relation tuples, where each disk page is 4KB in size, while for "BB-L" we used a buffer size of 512 disk pages. The PMR quadtree indexes were built on an existing relation, which consisted of only a line segment attribute, and where the tuples in the relation were initially inserted in unsorted order. Since the objects were not spatially clustered in the relation table, objects that are next to each other in the Morton order are typically not stored in close proximity (i.e., on the same disk page) in the relation table. This had the potential to (and did) cause excessive relation disk I/O during the quadtree construction process when we inserted in Morton order (i.e., in "BB-S" and "QB-100"). A similar effect arises for objects in a leaf node being split, regardless of insertion order. Thus, in "BB-S" and "QB-100" we built a new object table for the index, into which the objects were placed in the same order that they were inserted into the quadtree; this effectively clusters together on disk pages objects that are spatially near each other. When measuring the execution time for the quadtree construction, we took into account the time to construct the new object table.

Figure 23 shows the speedup of "QB-100" compared to "BB-L" and "BB-S" for building a PMR quadtree index in SAND for the line segment data, using the object table approach described above. This time, the speedup for "QB-100" compared to "BB-S" is somewhat smaller than we saw earlier, being a little less than 3 instead of 3 to 4 before, but the same general trend is apparent. The smaller speedup is due to the fact that the execution cost of activities common to the two is higher now than before, since the coordinate values in these experiments were larger (8 bytes vs. 4 bytes before), leading to a higher I/O cost for reading and writing line segment data. On the other hand, "BB-L" is now considerably slower in comparison to "QB-100" for the "R128K" and "R260K" data sets, which is caused by a much larger amount of relation I/O, in spite of "BB-L" having four times as large a buffer. This clearly demonstrates

the value of using a spatially clustered object table, as is the case in "QB-100" and "BB-S". Interestingly, the clustering was obtained as a by-product of sorting the objects in Z-order, providing a further example of the importance of this sorting order.

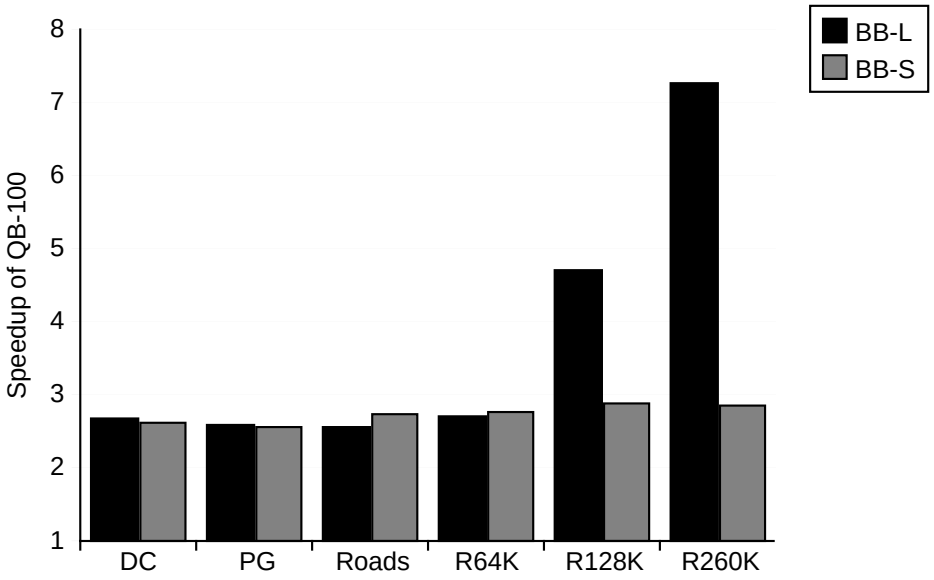

Figure 23: Speedup of "QB-100" compared to the other methods for line segment data, using object table approach.

9.2.3 Multidimensional Point Data

Next, we examine the effect of the dimensionality of the space on the performance of our bulk-loading methods (for both the PMR quadtree and the PR quadtree), using synthetic point data sets of 100,000 points each, in dimensions ranging from 2 to 8. The sets of points form 10 normally-distributed clusters with the cluster centers uniformly distributed in the space [18]. We used SAND for these experiments, storing the point geometry directly in the index. We compare using the loading methods "BB-L", "BB-S", and "QB-100" in Table 2, in addition to the PR quadtree bulk-loading algorithm described in Section 7 (denoted below by "PB-100"). Figure 24 shows the execution time of building the quadtree, while Figure 25 shows the speedup of "QB-100" compared to "BB-L" and "BB-S". The speedup is considerable for the lowest dimensions (factors of about 4 and 2.5 for "BB-L" and "BB-S", respectively), but becomes less as the number of dimensions grows. However, this is not because quadtree buffering is inherently worse for the larger dimensions. Rather, it is because the cost that is common to all loading methods (disk I/O, intersection computations, etc.) keeps growing with the number of dimensions. Figure 26 shows the speedup of "PB-100" compared to "QB-100". The speedup is initially about 17% but gradually decreases as the number of dimensions increases.

9.2.4 Complex Spatial Types (Polygons)

In the next set of experiments we built PMR quadtrees for a polygon data set consisting of approximately 60,000 polygons. The polygons represent census tracts in the United States and contain an average of about 40 boundary points each (which meant that each data page contained only about six polygons on the average), but as much as 3700 for the most complex ones, occupying over 40MB of disk space. We performed this experiment in SAND with the same loading methods as before. This time, we used a

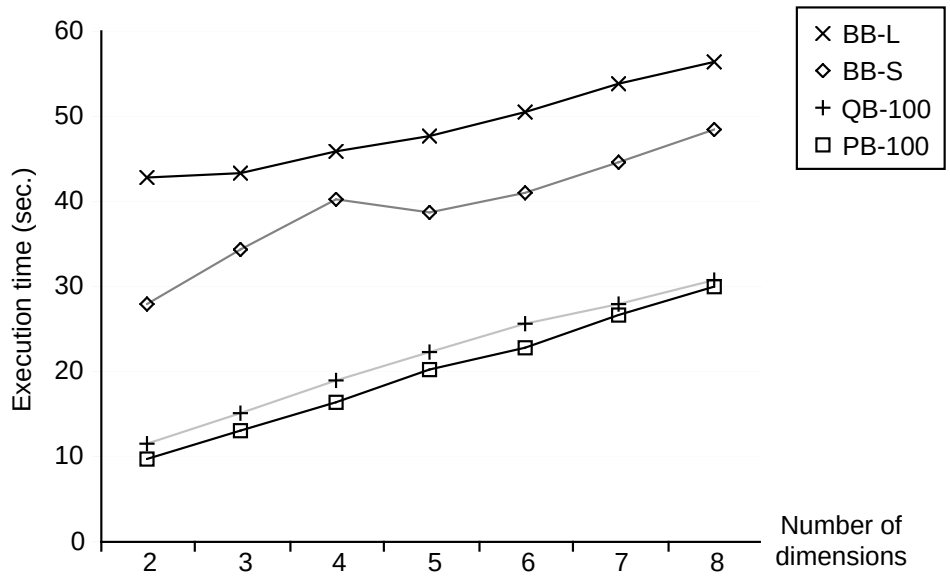

Figure 24: Execution time for building PMR quadtrees for point data sets of varying dimensionality (using "BB-L", "BB-S", and "QB-100", described in Table 2) and a PR quadtree with the bulk-loading algorithm from Section 7 (denoted "PB-100").

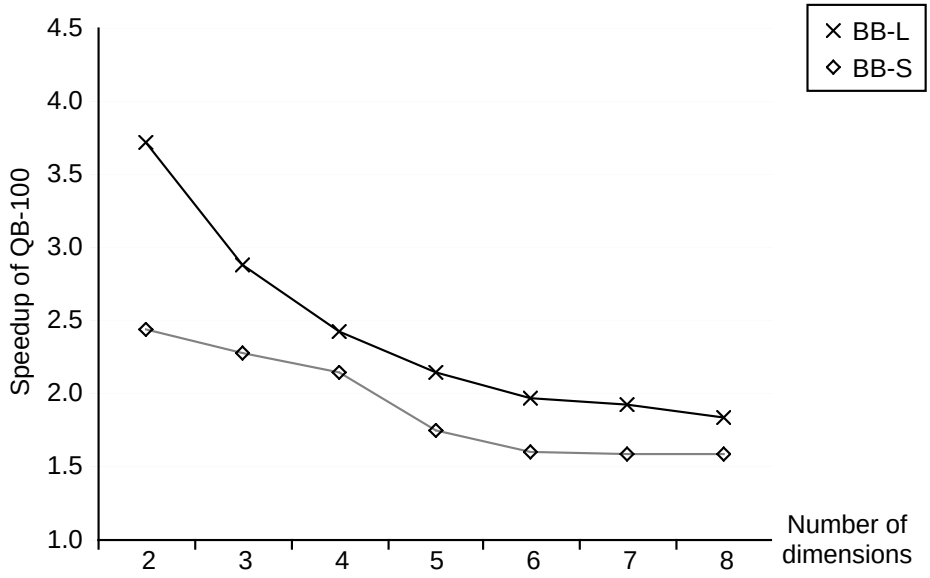

Figure 25: Speedup of "QB-100" compared to "BB-L" and "BB-S" for point data sets of varying dimensionality.

splitting threshold of 32, leading to an average of about two q-objects for each object. In contrast, the complex boundaries of the polygons led to an excessively large number of q-objects for a splitting threshold of 8, about four for each object on the average (however, the speedup achieved by our bulk-loading algorithm over the dynamic insertion method was better with the lower threshold value). As polygons have different numbers of edges, we had to use the object table approach, where we only store object

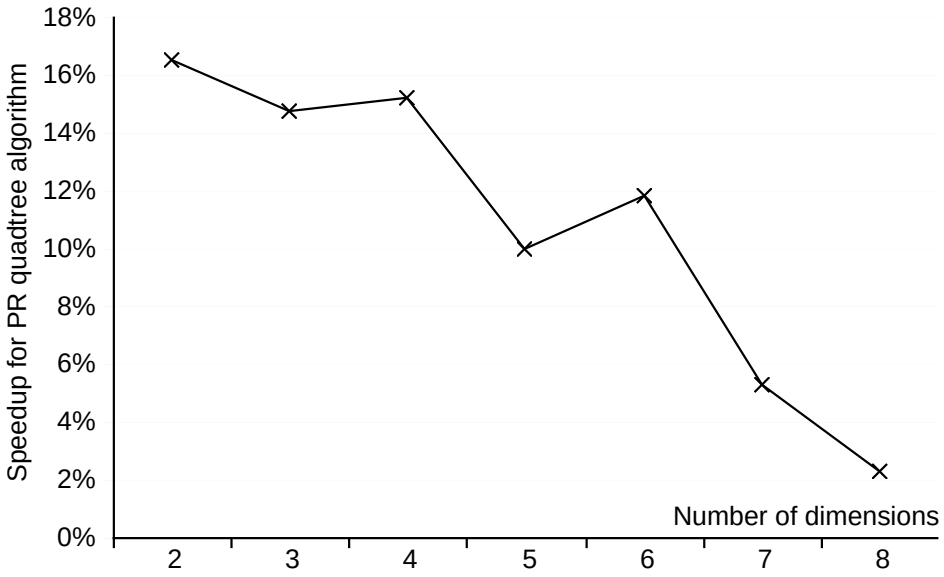

Figure 26: Speedup of the PR quadtree bulk-loading algorithm compared to the PMR quadtree bulk-loading algorithm for point data sets of varying dimensionality.

references in the quadtree.

In the first experiment with the polygon data, the polygon relation was not spatially clustered. In this context, spatial clustering denotes the clustering obtained by sorting the objects in Z-order, as is done by "BB-S" and "QB-100". For this data, more I/Os were required for building a spatially clustered object table for "BB-S" and "QB-100" than when accessing the unclustered relation table directly. To see why this is so, we observe that when building a new clustered object table for a large data set, the sorting process involves reading in the data, writing all the data to temporary files at least once, reading it back in, and then finally writing out a new object table. Thus, at least four I/Os are performed for each data page, half of which are write operations. In contrast, when the unclustered relation is accessed directly, the data items being sorted are the tuple IDs, so the sorting cost is relatively small. Nevertheless, in our experiment, this caused each data page to be read over three times on the average for "BB-S" and "QB-100"[12]. The difference between the polygon data and the line segment data, where building a new clustered object table was advantageous, is that in the polygon relation there is a low average number of objects in each data page. Thus, the average I/O cost per object is high for the polygon data when building a new object table, whereas the penalty for accessing the unclustered object table directly is not excessive as there are relatively few distinct objects stored in each page. As a comparison, when using "BB-L" to build the PMR quadtree, which does not sort the data and for which we used a large relation buffer of 2048 data pages (occupying 8MB), the overhead in data page accesses was only about 17% (i.e., on the average, each page was accessed about 1.17 times).

The first column ("Polys (unclust.)") in Figure 27 shows the execution times for the experiment described above. The large amount of relation I/O resulted in "QB-100" being nearly twice as slow as "BB-L". Nevertheless, "QB-100" was slightly faster than "BB-S" (by 10%). In order to explore the additional cost incurred by "QB-100" and "BB-S" for repeatedly reading many of the data pages (due to the sorted

[12]Each data page is read once when preparing to sort the polygons, since their bounding rectangles must be obtained. The remaining two I/Os per page (out of the three we observed on the average for each data page) occur when each polygon is initially inserted into the quadtree or when a node is split.

insertions), we measured the cost of building a PMR quadtree when the polygon relation was already spatially clustered ("Polys (clust.)") as well as building it on the bounding rectangles of the polygons ("Rectangles" in Figure 27). In the former case, we did not need to sort the data again for "QB-100" and "BB-S", thus only incurring 29% overhead in data page accesses, while in the latter case, each polygon was accessed only once, i.e., to compute its bounding rectangle. The geometry of the bounding rectangles was stored directly in the quadtree. Of course, the PMR quadtrees for the bounding rectangles are somewhat different from those for the polygons themselves, since some leaf nodes may intersect a bounding rectangle but not the corresponding polygon. In both cases, "QB-100" and "BB-S" take much less time to build the PMR quadtree, and the speedup of "QB-100" compared to "BB-S" is by a factor of 2. However, the speedup of "QB-100" over "BB-L" is not quite as high when building the quadtree on the clustered polygon relation (by a factor of 1.7) as when building it on the bounding rectangles (by a factor of 2.5).

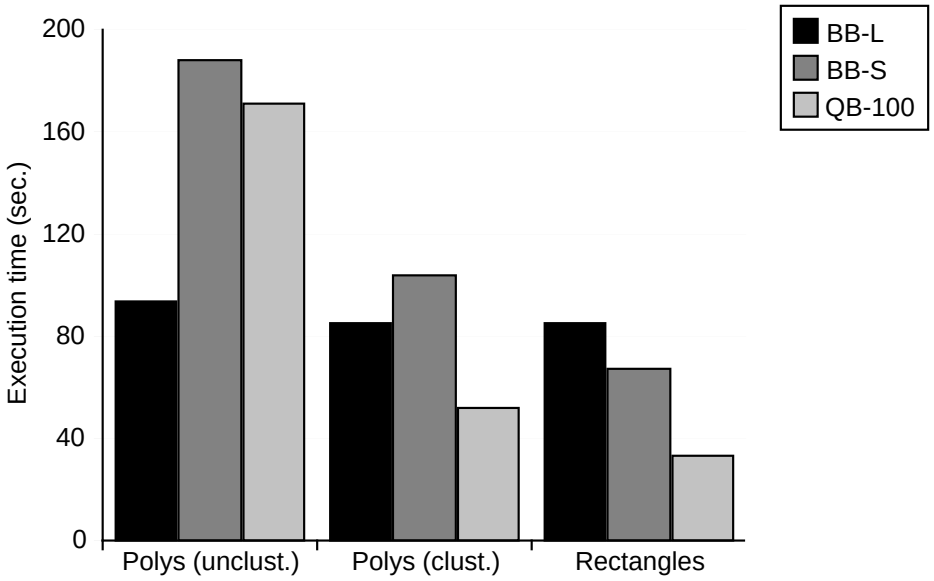

Figure 27: Execution time for building PMR quadtrees for polygon data set (labels of bars denote loading methods from Table 2). "Polys (unclust.)" denotes building the quadtree on an unclustered polygon relation, "Polys (clust.)" denotes building it on an spatially clustered polygon relation, while "Rectangles" denotes building it on the bounding rectangles of the polygons.

9.2.5 Improved PMR Quadtree Insertion Algorithm

In Section 4 we presented a technique for improving the performance of PMR quadtree insertions, which significantly reduces the number of intersection tests. Figure 28 shows the speedup in execution time that results from using our technique with the line segment data sets when building a PMR quadtry with dynamic insertions ("BB-S") as well as with our bulk-loading algorithm ("QB-100"). The speedup is considerable, ranging from 30% to nearly 55% for "BB-S" and slightly less for "QB-100". Observe that, due to sorting, "BB-S" performs fewer I/Os than the dynamic insertion algorithm typically performs (without sorting) with a similar B-tree buffer size, so the speedup for dynamic insertions without sorting can be expected to be somewhat less in most cases. For "QB-100", 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 29 shows the

speedup in execution time when building a PMR quadtree for the point data sets of varying dimensionality (see Section 9.2.3). For the two-dimensional data set the speedup is about 50% when using "QB-100". More importantly, as the dimensionality increases, the speedup grows, reaching a factor of nearly 8 for the eight-dimensional point data set. The speedup of dynamic insertions ("BB-S") for the point data is somewhat less than for the bulk-loading algorithm, but still substantial.

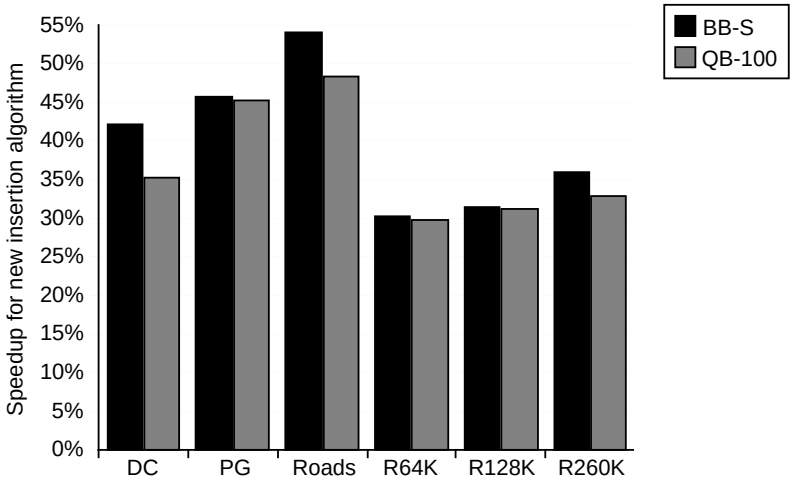

Figure 28: Speedup in terms of execution time resulting from the reduction in the number of intersection tests when using the improved PMR quadtree insertion algorithm for building PMR quadtrees for line segment data using dynamic insertions ("BB-S") and the bulk-loading algorithm ("QB-100").

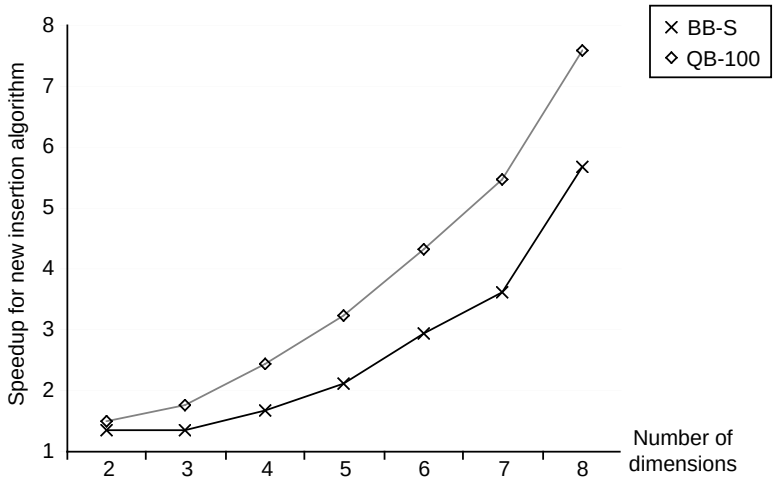

Figure 29: Speedup in terms of execution time resulting from the reduction in the number of intersection tests when using the improved PMR quadtree insertion algorithm for building PMR quadtrees for point data of varying dimensionality using dynamic insertions ("BB-S") and the bulk-loading algorithm ("QB-100").

9.2.6 Reinsert Freeing

In Section 5.3 we described a strategy we termed reinsert freeing that is used if the flushing algorithm fails to free any memory. The next set of experiments explores how well reinsert freeing performs. We used two synthetic line segment data sets, and stored their geometry in the PMR quadtree. The first data set, R260K, was described earlier. In order to cause the flushing algorithm to fail when building a PMR quadtree for R260K, we set the buffer size for quadtree buffering to only 8K. The second data set, R10K, consists of 10,000 line segments whose centroids are uniformly distributed over the data space, and whose length and orientation are also uniformly distributed. Thus, this data set exhibits a large degree of overlap and therefore a large number of q-edges, causing the MBI B-tree to occupy a large amount of disk space. For instance, the B-tree resulting from building a quadtree for R10K with "QB-100" occupied over 8000 nodes or about 32MB. For R10K, we used a splitting threshold of 32, as a lower splitting threshold led to an even higher number of q-edges (the speedup achieved by quadtree buffering was better at lower splitting thresholds, however). For both data sets, we used the merge sort algorithm to sort the objects, since it is better suited for handling reinsertions.

The number of reinsertions for R260K was about 21,000, while it was over 72,000 for R10K (i.e., each object was reinserted over seven times on the average). In spite of such a large number of reinsertions, Figure 30 shows that quadtree buffering yields significant speedup over B-tree buffering. In fact, B-tree buffering was so ineffective for R10K, that we increased the buffer size of "BB-S" to about 3000 B-tree nodes, which is about 25% of the number of nodes in the resulting B-tree. For a data set of 20,000 line segments constructed in the same way as R10K, the speedup for "QB-100" compared to "BB-L" was by a factor of more than 8, so it is clear that quadtree buffering with reinsertions scales up well with data size, even if the data has extreme amount of overlap. With "QB-100", it took about 4.5 times as long to build the PMR quadtree for the 20,000 line segment data set as for R10K, but the larger data set also occupied nearly four times as much disk space. For the more typical data set, R260K, the speedup achieved by "QB-100" is only slightly lower than what we saw in Figure 21, where reinsertions were not needed.

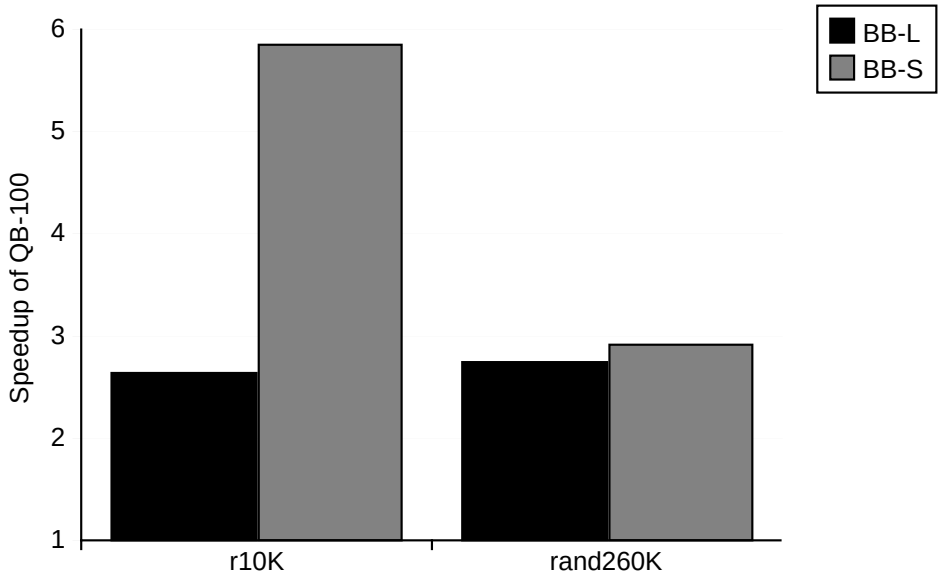

Figure 30: Speedup of "QB-100" compared to the other methods when reinsertions are needed (labels of bars denote loading methods from Table 2).

9.2.7 Bulk-Insertions

The next set of experiments investigates the performance of PMR quadtree bulk-insertions (see Section 6). We used two pairs of line segment data sets. In the first, comprising the "DC" and "PG" line segment data sets, the new objects cover an unoccupied area in the existing quadtree. In the second, the new objects are interleaved with the objects in the existing quadtree. In this pair, the line segments denote roads ("Roads" with 200,482 line segments) and hydrography ("Water" with 37,495 line segments) in the Washington, DC, metro area. For the bulk-insertions, we found that interleaved read and write operations (to the existing quadtree and the combined quadtree, respectively) caused a great deal of I/O overhead due to disk head seeks. To overcome this effect, we used a small B-tree buffer of 32 nodes (occupying 128KB) for the combined quadtree, which allowed writing to disk multiple nodes at a time; another solution would be to store the existing quadtree and the combined quadtree on different disks.

Figure 31 shows the execution time required to bulk-load and bulk-insert the pairs of data sets in either order, as well as to bulk-load the combined data set. In the figure, the notation X, Y means that first X is bulk-loaded, and then Y is bulk-inserted into the quadtree containing X, while the notation $X + Y$ means that the union of the two sets is bulk-loaded. The execution times of the bulk-load ("BL") and bulk-insertion ("BI") operations are indicated separately on the bars in the figure. In addition, the topmost portion of each bar, above the broken line, indicates the I/O overhead of the combined bulk-load and bulk-insertion operations, i.e., the cost of writing (during the bulk-load) and reading (during the bulk-insertion) the intermediate PMR quadtree. Clearly, the I/O cost overhead represents nearly all the additional cost of bulk-loading and bulk-inserting compared to bulk-loading the combined data set. Interestingly, the remaining overhead was very similar in all cases, amounting to 7-11% of the execution time of bulk-loading the combined data sets. Since the pairs of data sets had different relative space coverage and size, this demonstrates that the performance of our bulk-insertion algorithm is largely independent of the space coverage of the bulk-inserted data in relation to the existing data, as well as the relative sizes of the existing and new data sets (with the exception that the I/O overhead is proportional to size of the existing data set in relation to the combined data set).

In Section 6.3 we discussed a variant of our bulk-insertion algorithm that updates the existing quadtree, as opposed to the merge-based approach that builds a new quadtree on disk. Figure 32 shows the performance of the update-based bulk-insertion variant relative to the merge-based bulk-insertion algorithm, as well as that of using dynamic insertions into the existing quadtree using "BB-S". In an attempt to make a fair comparison we made the alternative methods as efficient as possible. In particular, for the update-based bulk-insertion variant, we used the adapted B-tree packing approach (see Section 5.5), with a split fraction of 90%, and the existing quadtree had a storage utilization of 90%. For "BB-S", the existing quadtree had a storage utilization of 75% (higher values caused more B-tree node splits). Note that in Figure 32, we only take into account the bulk-insertion of the new data set and not the bulk-loading of the existing one. The two alternative approaches for bulk-insertion, that both update the existing quadtree, are clearly much more sensitive to the relative space coverage of the new data set with respect to the existing one than our merge-based algorithm. In particular, when the new data set occupies an area that is not covered by the existing data set (as for "PG,DC"), the update-based methods work much better than when the new data set is interleaved with the existing data (as for "R,W"). In the latter case, a higher fraction of the nodes in the MBI B-tree are affected by the update operations, thus leading to more I/O. In addition, the update-based methods are also less effective when the new data set is larger than the existing data set. Nevertheless, if we know that bulk-insertions involve data sets that are mostly into unoccupied regions of a relatively large existing quadtree, then the update-based variant of our PMR quadtree bulk-insertion algorithm may be preferable.

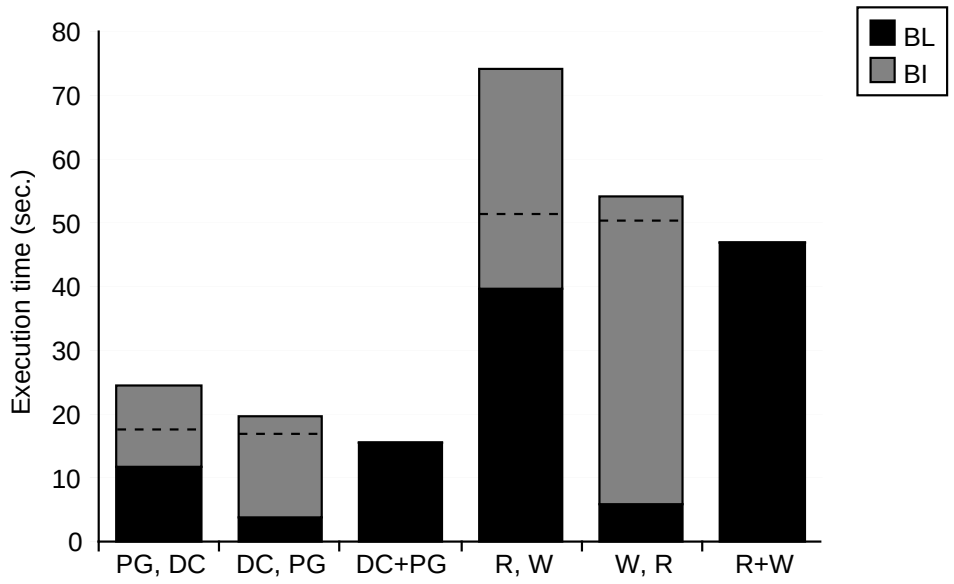

Figure 31: Execution time for bulk-loading (indicated by the bars labeled "BL") and bulk-insertions (indicated by the bars labeled "BI") for two pairs of data sets. The portions of the bars above the broken lines indicate the I/O overhead of the combined bulk-loading/bulk-insertion operations compared to bulk-loading the combined data set. "R" and "W" denote the "Roads" and "Water" data sets, respectively.

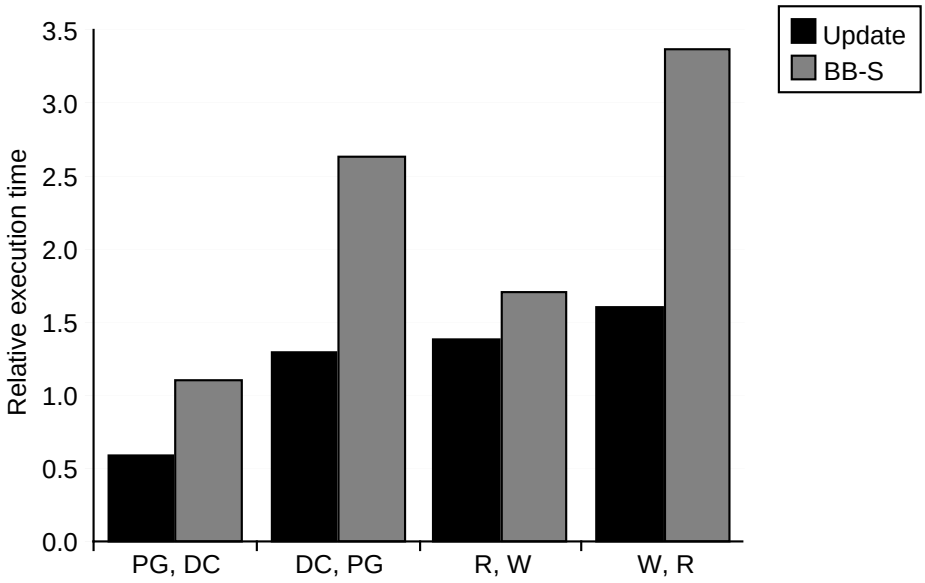

Figure 32: Execution time of two alternative bulk-insertion methods relative to the merge-based PMR quadtree bulk-insertion algorithm. "Update" denotes the update-based variant of our algorithm, while "BB-S" denotes dynamic insertions (see Table 2).

9.2.8 R-tree Bulk-Loading

It is interesting to compare the performance of our bulk-loading algorithm to that of existing bulk-loading algorithms for another commonly used spatial data structure, the R-tree. We chose two bulk-loading algorithms for the R-tree: 1) Hilbert-packed R-tree [30] with the space partitioning improvements of [19][13], and 2) a very simplified version of the buffer-tree approach of [8, 13]. For ease of implementation we used an unlimited buffer size for the buffer-tree approach, thus building the entire R-tree in memory. The nodes were written to disk once the tree was fully constructed. The CPU time of our approach is at most equivalent to that of [8, 13], while the I/O cost is much less. Note that virtual memory page faults were not a major issue, since the size of the R-trees (at most 27MB) was significantly less than the size of physical memory (64MB). In order to obtain good space partitioning, we used the R*-tree [10] insertion rules, except that no reinsertions were performed as they are not supported by the buffer-tree approaches. Since 4K is the physical disk page size in our system, we used R-tree nodes of that size, which allow a fan-out of up to 200. However, a fan-out of 50 is recommended in [10], and this is what we used in the buffer-tree approach. A fan-out of 200 led to a much worse performance, by more than an order of magnitude. For the Hilbert-packed R-tree, on the other hand, we use a fan-out of 200, as lower levels of fan-out lead to a higher I/O cost. The two methods are at two ends of a spectrum with respect to execution time. For the Hilbert-packed R-tree, nearly all the time is spent doing I/O, whereas for the buffer-tree approach, nearly all the execution time is CPU time. It is important to note that the quality of the space partitioning obtained by the Hilbert-packed R-tree approach is generally not as high as that obtained by the R*-tree insertion method. This is in marked contrast to our quadtree PMR quadtree bulk-loading algorithm, which produces roughly the same space partitioning as dynamic insertions (the variation is due to different insertion order).

Figure 33 shows the execution time performance of the two methods for bulk-loading R-trees for the data sets listed in Table 1 relative to the execution time of "QB-100". The buffer-tree technique with R*-tree partitioning ("B50") took 10-14 as much time as building the PMR quadtree. However, building the Hilbert-packed R-tree ("P200" in the figure) took less time, or about 50%-80% as much as building a PMR quadtree. This was partly due to the small CPU cost of the Hilbert-packed R-tree method, but primarily due to the fact that in the PMR quadtree each object may be represented in more than one leaf node and thus stored more than once in the MBI's B-tree. Thus we see that the price of a disjoint space decomposition, which is a distinguishing feature of the PMR quadtree, is relatively low when using our bulk-loading algorithm.

9.2.9 Summary

Our experiments have confirmed that our PMR quadtree bulk-loading algorithm achieves considerable speedup compared to dynamic insertions (i.e., when updating the MBI directly). The speedup depended on several factors. One is the effectiveness of buffering the B-tree used in dynamic insertions. When the nodes in the B-tree were effectively buffered, our bulk-loading algorithm usually achieved a speedup of a factor of 3 to 4. This speedup was achieved, for the most part, by a dramatic reduction in CPU time. In fact, in some experiments, only about 25-35% of the execution time of our bulk-loading algorithm was attributed to CPU cost. However, when B-tree buffering is ineffective so that B-tree nodes are frequently brought into the buffer and written out more than once in dynamic insertions, our bulk-loading approach can achieve substantially higher speedups (up to a factor of 12 in our experiments). In situations requiring the use of reinsert freeing, our bulk-loading algorithm was at worst only slightly slower than in situations

[13] We only used the first of their improvements, wherein each node is not quite filled to capacity if the addition of an object causes the bounding rectangle of the node to enlarge too much. The use of re-splitting would involve more CPU cost, while the I/O cost would stay the same or increase.

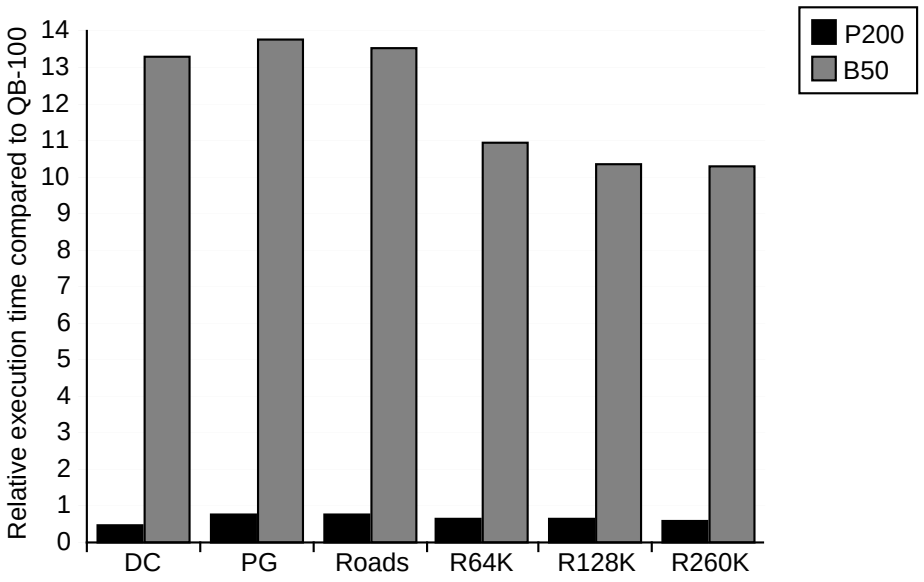

Figure 33: Relative performance of two R-tree bulk-loading algorithm compared to "QB-100" ("P200" denotes the Hilbert-packed R-tree algorithm with a fan-out of 200, while "B50" denotes the buffer-tree approach with a fan-out of 50).

where the flushing algorithm was sufficient.

Another factor affecting the speedup of the bulk-loading algorithm is the relative importance of cost factors common to any PMR quadtree construction method, such as the cost of reading the input data and of intersection tests. As these common cost factors become a larger portion of the total cost, the potential for speedup diminishes. Indeed, we found that for point data, the speedup achieved by our PMR quadtree bulk-loading approach diminishes as the number of dimensions increases.

As we expected, our PR quadtree bulk-loading algorithm outperformed the PMR quadtree bulk-loading algorithm for point data. However, the speedup in execution time was less than 20% at best, and decreased with a higher number of dimensions. The relatively small speedup achieved by the PR quadtree bulk-loading algorithm over the PMR quadtree bulk-loading algorithm indicates that the overhead (in terms of execution time) due to the use of the pointer-based quadtree and the associated flushing process in the PMR quadtree bulk-loading algorithm is minor.

Our experiments with complex polygon data showed that a lack of spatial clustering[14] in a spatial relation has an especially detrimental effect on the amount of I/O when the spatial objects occupy a large amount of storage space (which means that few objects fit on each data page). Without spatial clustering on the polygon relation, the PMR quadtree bulk-loading algorithm took about twice as long to build the quadtree as doing dynamic insertions. The difference in performance was due to the fact that we allotted a much larger buffer space to the latter, besides the fact that it is less affected by the lack of spatial clustering since the objects are not sorted prior to inserting them into the quadtree. Nevertheless, when the polygon relation was spatially clustered as well as when building the quadtree based on the bounding rectangles of the polygons, the speedup of our bulk-loading algorithm was about a factor of 2 when a comparable amount of buffer space was used. In situations where the relation to index is not spatially clustered (and performing clustering is not desired), using bounding rectangles may yield overall savings in execution time (for building the quadtree and executing queries), even though it means that the quadtree provides

[14]Recall that in this context, spatial clustering denotes the clustering obtained by sorting the objects in Z-order.

somewhat worse spatial filtering and thus potentially higher query cost.

The improved PMR quadtree insertion algorithm (that reduces the number of intersection tests) was shown to yield significant speedup, both for dynamic insertions as well as in our PMR quadtree bulk-loading algorithm. For line segment data, the speedup when it was used in the PMR quadtree bulk-loading algorithm ranged between 30-50%. For point data, the speedup was 50% for two-dimensional data and grew with the number of dimensions up to nearly a factor of 8 for eight-dimensional data.

We verified that our PMR quadtree bulk-insertion algorithm is very efficient. Compared to bulk-loading the combined data set, most of the extra cost of first bulk-loading the existing data and then bulk-inserting the new data lies in I/O operations, while the overhead due to larger CPU cost was minor. Furthermore, our bulk-insertion algorithm is more robust and generally more efficient than an update-based variant of the algorithm that updates the existing quadtree instead of merging the existing quadtree with the quadtree for the new data. Nevertheless, the update-based variant is more efficient in certain circumstances, namely when the amount of new data is relatively small and covers an unoccupied region in the existing quadtree.

Our bulk-loading algorithm for PMR quadtrees compared favorably to bulk-loading algorithms for R-trees. In particular, the price paid for the disjoint decomposition provided by the PMR quadtree is relatively low. An R-tree algorithm having very low CPU cost (the Hilbert-packed R-tree) was at most about twice as fast as our algorithm. Most of the difference can be explained by higher I/O cost for PMR quadtree bulk-loading due to the presence of multiple q-objects per object. When we used the object table approach in the PMR quadtree, in which the actual objects are stored outside the quadtree (i.e., each object is stored only once regardless of the number of q-objects), the fastest R-tree bulk-loading algorithm was typically only 5-30% faster than our PMR quadtree bulk-loading algorithm. Moreover, R-tree bulk-loading algorithms that expend more CPU time to achieve better space partitioning (e.g., [8, 13] with R*-tree insertion rules) can be much slower than our algorithm.

9.3 Performance of Spatial Join

In order to test the utility of our PMR quadtree bulk-loading approach, we performed a small experiment with the spatial join example mentioned in Section 1: 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 ("Roads" with 200,482 line segments), and a data set for the hydrography of the same geographic area ("Water" with 37,495 line segments). Below, we give a description of the experiments, and in Table 4 we tabulate the time to execute each one. In the table, "BB-S" and "QB-100" denote quadtree loading methods as summarized in Table 2.

1. Build a spatial index on the Roads and Water data sets.

2. Perform the spatial join with a spatial index on both data sets. This is done by simultaneously stepping through the MBI for each index.

3. Perform the spatial join with a spatial index on the Roads data set but not on the Water data set. This method looks up intersecting line segments in the Roads index for each line segment in the Water data set. In order to reduce the number of I/Os to the Roads index, the line segments in the Water data set are sorted in Morton order of their centroids.

4. Experiment 3 with the roles of Roads and Water reversed. In other words, we have a spatial join with a spatial index on the Water data set but not on the Roads data set.

5. Perform the spatial join with no index on either of the data sets. This is done with a nested loop method. In the buffered variation, all line segments in the Water data sets were read into memory at the start of the query to avoid re-reading them.

6. Build a spatial index on the output of the spatial join. Actually, in our experiment, we built the index after the join had been computed, but building the index simultaneously with the join would yield the same results.

Data set	BB-S	QB-100
Roads	134	38.9
Water	17.8	5.75
Join output (points)	2.55	0.89

(a)

Join method	Time
Both indexed	8.92
Only Roads indexed	25.9
Only Water indexed	75.8
Neither indexed (buffered)	1420
Neither indexed (unbuffered)	33900

(b)

Table 4: Execution time (given in seconds) for (a) building indexes and (b) computing a spatial join for the Roads and Water data sets.

If either of the data sets is not indexed prior to executing the query, then we have two alternatives. The first is to build an index (possibly on both data sets) and then execute the query with the two indexes. The second alternative is to run the query without building any new indexes. The cost of these alternatives is shown in Table 4, where part (a) shows the cost of building indexes and part (b) shows the cost of processing the join query, with or without indexes. Now, let us focus on the case of building indexes using quadtree buffering prior to computing the spatial join. If an index existed for the Roads data set but not for the Water data set, then the speedup achieved by building the index prior to running the query is about 76% (14.7 seconds as opposed to 25.9 seconds). For the converse case (i.e., no index on the Roads data set), it is about 59% faster to build an index on Roads and run the query (47.8 seconds as opposed to 75.8 seconds). If neither data set has an index, then it would take 53.6 seconds to build an index on both 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. As a comparison, using B-tree buffering (i.e., "BB-S"), the performance is about the same if an index must be built on Water (26.7 vs. 25.9), but building an index on Roads and computing the join takes nearly twice as long as computing the join with only the index in Water (143 vs. 75.8). However, a speedup of nearly 8 times is achieved if an index must be built on both (161 vs. 1420). Interestingly, even though building two indexes and performing the query is faster than performing a query without indexes, it takes much longer to build the two indexes than to perform the query with them.

10 Concluding Remarks

There are three typical situations in which an index must be updated: 1) a new index must be built from scratch on a set of objects (bulk-loading), 2) a batch of objects must be inserted into an existing index (bulk-insertion), and 3) one object (or only a few) must be inserted into an existing index (dynamic insertions). In this paper we have presented techniques for speeding up index construction for the PMR quadtree spatial index in all three situations. Furthermore, we introduced bulk-loading and bulk-insertion techniques for the PR quadtree multidimensional point index.

In an informal analysis of the PMR quadtree bulk-loading algorithm, we presented persuasive evidence that both its I/O and CPU costs are asymptotically the same as that of external sorting for rea-

sonably "well-behaved" data distributions. Indeed, our experiments verified that the execution time per object grows very slowly with the size of the data sets. Moreover, the speedup of the bulk-loading algorithm over the dynamic algorithm (which updates the disk-resident quadtree directly for each insertion) is substantial, up to a factor of 12 for the data sets we used. When the dynamic algorithm was enhanced to better take advantage of buffering, the speedup was still significant, typically a factor of 2 to 4, depending on the data distribution and other factors (see Section 9.2.9).

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 query engine for SAND that exploits this to construct spatial indexes for intermediate query results (possibly from non-spatial subqueries), 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] D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, January 1990.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[3] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman. The *S*-tree: an efficient index for multidimensional objects. In *Advances in Spatial Databases — Fifth International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 350–373, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262).

[4] C. H. Ang and T. C. Tan. New linear node splitting algorithm for R-trees. In *Advances in Spatial Databases — Fifth International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 339–349, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262).

[5] 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*, vol. 2, pages 589–598, Zurich, Switzerland, July 1990.

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

[7] L. Arge. *Efficient external-memory data structures and applications*. BRICS dissertation series, DS-96-3, University of Aarhus, 1996.

[8] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALANEX '99)*, Baltimore, MD, January 1999.

[9] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, A. Finkel and M. Jantzen, eds., pages 475–486, ENS Cachan, France, February 1992. (Also Springer-Verlag Lecture Notes in Computer Science 577).

[10] 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 ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[11] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[12] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Advances in Database Technology — EDBT'98, 6th International Conference on Extending Database Technology*, pages 216–230, Valencia, Spain, March 1998.

[13] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 406–415, Athens, Greece, August 1997.

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

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

[16] L. Chen, R. Choubey, and E. A. Rundensteiner. Bulk-insertions into R-trees using the Small-Tree-Large-Tree approach. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, R. Laurini, K. Makki, and N. Pissinou, eds., pages 161–162, Washington, DC, November 1998.

[17] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)*, Perth, Australia, February 1998.

[18] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 426–435, Athens, Greece, August 1997. (Source code available at `http://www-db.deis.unibo.it/~patella/MMindex.html`).

[19] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. B. Yu. Client-server paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases*, J. Bocca, M. Jarke, and C. Zaniolo, eds., pages 558–569, Santiago, Chile, September 1994.

[20] C. Esperança and H. Samet. Orthogonal polygons as bounding structures in filter-refine query processing strategies. In *Advances in Spatial Databases — Fifth International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 197–220, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262).

[21] C. Faloutsos. Multiattribute hashing using gray codes. In *Proceedings of the ACM SIGMOD Conference*, pages 227–238, Washington, DC, May 1986.

[22] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Conference*, pages 260–269, San Francisco, CA, May 1987.

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

[24] D. M. Gavrila. R-tree index optimization. In *Proceedings of the Sixth International Symposium on Spatial Data Handling*, T. C. Waugh and R. G. Healey, eds., pages 771–791, Edinburgh, Scotland, September 1994.

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

[26] G. Hjaltason, H. Samet, and Y. Sussmann. Speeding up bulk-loading of quadtrees. In *Proceedings of the 5th International ACM Workshop on Advances in GIS*, pages 50–53, Las Vegas, NV, November 1997.

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

[28] S.-H. S. Huang and V. Viswanathan. On the construction of weighted time-optimal B-trees. *BIT*, 30(2):207–215, 1990.

[29] G. Iwerks and H. Samet. The spatial spreadsheet. In *Proceedings of the Third International Conference on Visual Information Systems (VISUAL99)*, A. Smuelders, ed., Amsterdam, The Netherlands, June 1999.

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

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

[32] I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the Seventh International Symposium on Spatial Data Handling*, M. J. Kraak and M. Molenaar, eds., pages 3B.31–3B.42, Delft, The Netherlands, August 1996.

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

[34] S. T. Leutenegger, M. A. Lòpez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 497–506, Birmingham, U.K., April 1997.

[35] S. T. Leutenegger and D. M. Nicol. Efficient bulk-loading of gridfiles. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):410–420, May/June 1997.

[36] J. Li, D. Rotem, and J. Srivastava. Algorithms for loading parallel grid files. In *Proceedings of the ACM SIGMOD Conference*, pages 347–356, Washington, DC, May 1993.

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

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

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

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

[41] 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.

[42] Y. J. García R., M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, R. Laurini, K. Makki, and N. Pissinou, eds., pages 163–164, Washington, DC, November 1998.

[43] Y. J. García R., M. A. López, and S. T. Leutenegger. On optimal node splitting for R-trees. In *Proceedings of the 24th International Conference on Very Large Data Bases*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 334–344, New York, August 1998.

[44] A. L. Rosenberg and L. Snyder. Time- and space-optimality in B-trees. *ACM Transactions on Database Systems*, 6(1):174–193, March 1981.

[45] N. Roussopolous, Y. Kotidis, and M. Roussopolous. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of the ACM SIGMOD Conference*, pages 89–111, Tucson, AZ, May 1997.

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

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

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

[49] B. Seeger and H. P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, D. McLeod, R. Sacks-Davis, and H. Schek, eds., pages 590–601, Brisbane, Australia, August 1990.

[50] 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.

[51] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In *Proceedings of the 5th International Conference of Foundations of Data Organization (FODO)*, pages 27–36, Kobe, Japan, November 1998.

[52] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, CA, 1996. (see `http://vision.ucsd.edu/papers/simret`).

[53] J. Yang, W. Wang, and R. Muntz. Yet another spatial indexing structure. Computer Science Department Technical Report 970040, University of California, Los Angeles, CA, 1997. (see `http://dml.cs.ucla.edu/~weiwang/paper/TR97040.ps`).

[54] A. C. Yao. On random 2-3 trees. *Acta Informatica*, 9(2):159–168, 1978.