VLDB Journal (2002) 11: 190–137 / Digital Object Identifier (DOI) 10.1007/s00778-002-0067-8

Speeding up construction of PMR quadtree-based spatial indexes

Gísli R. Hjaltason*, **Hanan Samet**

Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742, USA; e-mail: hjs@cs.umd.edu

Edited by R. Sacks-Davis. Received: July 10, 2001 / Accepted: March 25, 2002
Published online: September 25, 2002 –

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 quadtree-based spatial indexes, specifically the PMR quadtree, which can index arbitrary spatial data. We assume a quadtree implementation using the "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. 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. 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 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 a main memory buffer is very small compared to the size of the resulting quadtrees.

Keywords: Spatial indexing – Bulk-loading – I/O

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 less 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 all the 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 [46].

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 evi-

This work was supported in part by the National Science Foundation under Grants IRI-97-12715, EIA-99-00268, EIA-99-01636, EAR-99-05844, IIS-00-86162, and EIA-00-91474, the Department of Energy under Contract DEFG2095ER25237, and the Italian National Group for Mathematical Computer Science (GNIM).

* *Present address*: School of Computer Science, University of Waterloo, Ontario N2L 3G1, Canada; e-mail: gisli@db.uwaterloo.ca

dent 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 [34]), 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 in this context because an earlier study [32] showed that the PMR quadtree performs quite well for spatial joins compared to other spatial data structures such as the R-tree [29] (including variants such as the R^*-tree [12]) and the R^+-tree [57]. This was especially true when the execution time of the spatial join included the time needed to build spatial indexes[1]. Improving the performance of building a quadtree spatial index is of interest to us for a number of additional reasons. First of all, the PMR quadtree is used as the spatial index for the spatial attributes in a prototype spatial database system built by our research group called SAND (Spatial and Non-Spatial Data) [6,7,22], 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 [48]. 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 [56] and the BANG file [24]. 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.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the PMR quadtree, 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 presents some analytical observations. Section 8 discusses the results of our experiments, while concluding remarks are made in Section 9.

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 [50], and Klein, Parzygnat, and Tharp [40] 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 (e.g., see [53]). Huang and Viswanathan [33] 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 [19] for the M-tree, a dynamic distance-based indexing structure.

Although targeting a different usage scenario, the B^+-tree bulk-update methods of O'Neil et al. [47] and Jagadish et al. [36] have some similarities with our methods. These methods assume a heavy stream of insertions intermixed with comparatively rare queries. Both make use of an internal memory buffer, portions of which are periodically moved to disk, which is also true of our bulk-loading method (see Sect. 5). In addition, these methods use merging to support bulk-insertions in a somewhat analogous manner as our bulk-insertion method (see Sect. 6). Furthermore, the external merge sort variant that we introduce in Sect. 5.4.2 is closely related to the "stepped merge" algorithm of Jagadish et al. [36] (although developed independently). However, our sorting algorithm is able to achieve near-optimality due to its more restricted usage assumptions, beside the minor difference that the algorithm of Jagadish et al. [36] builds a B^+-tree for each "sorted run" to support intervening queries, while our algorithm need not do so.

In recent years, many bulk-loading algorithms for spatial indexing structures have been introduced. Most of the attention

[1] Note that fast construction techniques for the R-tree, such as the packed R-tree [52] and Hilbert-packed R-tree [37], were not taken into account in this study as they tend to result in a worse space partitioning from the point of view of overlap than the standard R-tree construction algorithms.

has been focused on the R-tree and related structures. Among the exceptions are two algorithms for the grid file. Li, Rotem and Srivastava [43] introduced a dynamic programming algorithm that operates in a parallel domain, and primarily aims at obtaining a good grid partitioning. A much faster solution was introduced by Leutenegger and Nicol [42], 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 [12,29]. Some of these methods use a heuristic for aggregating objects into the leaf nodes [37,41,52], while others explicitly aim at producing good partitioning of the objects and thus a small level of overlap [4,14,25,59]. Roussopoulos and Leifker [52] 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 [52] 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 [37] 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 [41] 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. [21] 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 [28] 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.

[2] The exact order proposed by Roussopoulos and Leifker [52] 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.

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., [12]) 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 [59], which uses a variant of a k-d tree splitting strategy to obtain the space partitioning. García, López, and Leutenegger [25] 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. [4] 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 [14] 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 [9,15]. 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 [8], 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 [15], 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. [9] 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 [15] 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 [15] does not result in an R-tree structure identical to that resulting from the corresponding dynamic insertion method, whereas the algorithm in [9] does (assuming reinsertions [12] are not used). In addition, the method of [9] supports bulk-insertions (as opposed to just initial bulk-loading as in [15]) and bulk-queries, and in fact, intermixed insertions and queries.

With the exception of [9], 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 [18,39,51]. The cubetree [51] is an R-tree-like structure for on-line analytical processing (OLAP) applica-

tions that employs a specialized packing algorithm. The bulk-insertion algorithm proposed by Roussopolous, Kotidis, and Roussopolous [51] 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 [37] and possibly other R-tree packing algorithms. Kamel, Khalil, and Kouramajian [39] propose a bulk-insertion method in which new leaf nodes are first built following the Hilbert-packed R-tree [37] 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 [18], 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 [18,39] 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 fanout 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 Sect. 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 [51] (although our merging process is very different).

In addition to the numerous bulk-loading and bulk-insertion algorithms proposed for the R-tree, several different proposals exist for improving dynamic insertions [5,11,12, 26,38]. 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. [12], and the polynomial time optimal node splitting methods of Becker et al. [11] and García, López, and Leutenegger [26]. In addition, [12] and [26] also introduced heuristics for improving storage utilization. Ang and Tan [5] developed a linear time node splitting algorithm that they claim produces node splits that are better than the original node splitting algorithms [29] and competitive with that of the R*-tree. The Hilbert R-tree of Kamel and Faloutsos [38] employs the same heuristic as the Hilbert-packed R-tree [37], 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 [58] introduced the PK-tree, a multidimensional indexing structure based on regular partitioning. In [60], 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.

The main topic of this paper is a bulk-loading technique for PMR quadtrees. This subject has been previously addressed by Hjaltason, Samet, and Sussman [31]. The bulk-loading technique presented in this paper is an improvement on the algorithm in [31]. 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 [31] 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 [31] 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* [54,55] we mean a spatial data structure based on a disjoint regular partitioning of space; that is, a partitioning where each partition operation divides a region into mutually disjoint sub-regions of equal size and shape, and all partition operations result in the same number of sub-regions. 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 partitioning of two-dimensional 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., see Fig. 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. For a given quadtree block, we use the term *partition level* to indicate the level of the block in this tree view. Another way to view the quadtree is to focus on the space partitioning, in which case the quadtree can be thought of as being an adaptive grid (e.g., see Fig. 1a). Usually, there is a prescribed maximum partition level (i.e., a limit on the height of the tree), or equivalently, a minimum size

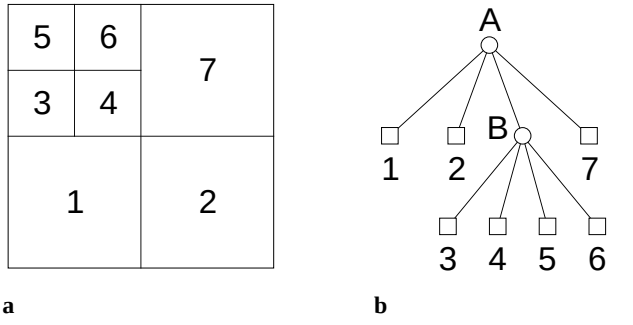


Fig. 1. a The block partitioning and **b** tree structure of a simple quadtree, where leaf blocks are labeled with numbers and nonleaf blocks with letters

for a quadtree block. For two-dimensional quadtrees, such as the one shown in Fig. 1, we often use the compass directions to refer to particular *quadrants* (i.e., one of the four sub-regions resulting from a partitioning). Thus, for example, the block labeled "1" in the figure is a South-West (abbreviated "SW") quadrant of the root block.

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 [54], 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 [54]).

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* [27].

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 4–8 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 [13] 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 [58], 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.

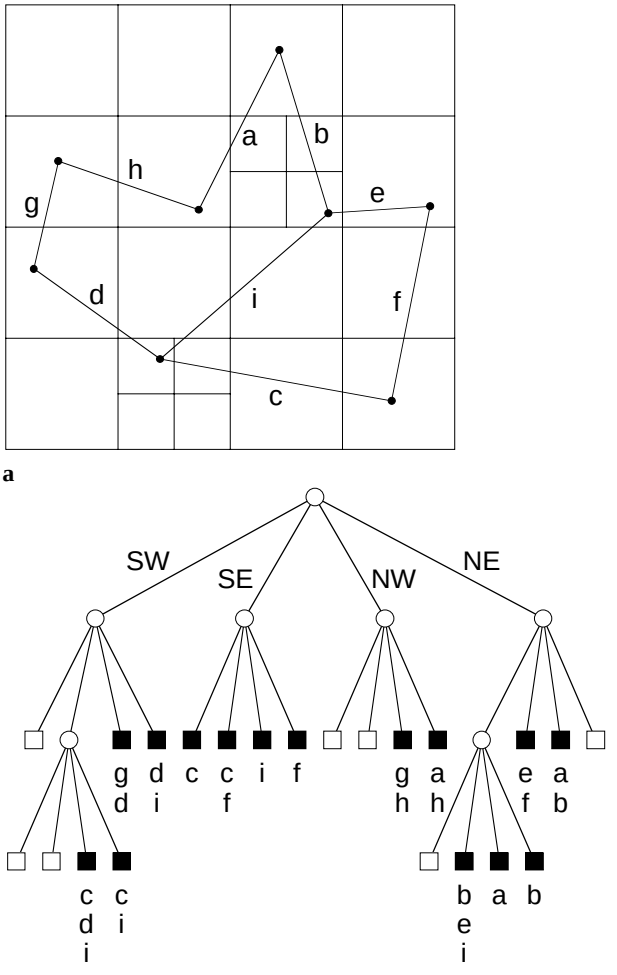
3.2 PMR quadtrees

The *PMR quadtree* [46] is a quadtree-based dynamic spatial data structure for storing objects of arbitrary spatial type. A sample PMR quadtree for a collection of line segments is shown in Fig. 2, where we show both the space partitioning and the resulting tree structure. Since the PMR quadtree gives rise to a disjoint partitioning 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 Fig. 2a is split into three q-edges as it intersects three leaf nodes, so that there are three references to a in leaf nodes of the tree structure shown in Fig. 2a.

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 partitioning 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 Fig. 2, if line segment g were inserted after line segment i instead of after line segment f, then the partitioning 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 exploit this observation later on, by re-ordering the objects to allow a more efficient quadtree construction process (see Sect. 5.1).

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 adaptable to any other disk-based representation of quadtrees. Nevertheless, for concreteness, it is helpful to review some of the details of our system.



a

b

Fig. 2a,b. A PMR quadtree for line segments with a splitting threshold of 2, where the line segments have been inserted in alphabetical order. **a** Spatial rendering of the line segments and the resulting quadtree space partitioning; **b** a tree access structure for **a**

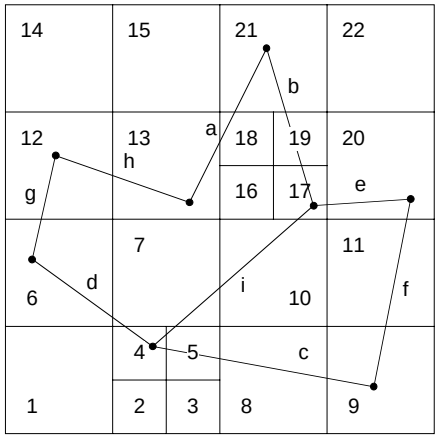
3.3.1 Morton codes

The MBI represents quadtree blocks using *Morton codes*, an encoding of the sequence of splits that result in the block, or, equivalently, of the path in the tree representation of the quadtree that leads to the block. In particular, in d-dimensional space, the child blocks resulting from a partitioning can be represented with the numbers $0, 1, \ldots, 2^{d-1}$. Thus, the Morton code of a block consists of a sequence of d bit numbers, where the length of the sequence represents the partition level of the block. For example, for the two-dimensional quadtree shown in Fig. 1, the leaf block labeled "4" can be encoded with the sequence "2,1", assuming that quadrants resulting from a split are numbered in the order SW, SE, NW, and NE, respectively. To define the particular "split encoding" (i.e., numbering of child blocks resulting from a split) used for Morton codes, think of the two halves resulting from the split along each axis as representing 0 and 1, respectively, with the half with lower coordinate values receiving the lower number. In other words, the split along each axis is represented with a bit. Given some fixed ordering of the axes, the encoding, then, is the binary number obtained by the concatenation of the bits representing the split along all axes. For two dimensions, assuming that the x axis is ordered before the y axis, this results in the encoding given above (i.e., S and W represent 0, while N and E represent 1, so SW=00b=0, SE=01b=1, NW=10b=2, and NE=11b=3, where "b" indicates binary).

For efficiency of implementation, the split sequences of Morton codes are typically represented as integers of a fixed precision. In particular, if w is the user-determined maximum partition level, w bits for each dimension are required, so the total number of bits for the split sequences is $d \cdot w$; in our implementation, w can be any value between 1 and 32. Of course, for a block at partition level l, the number of bits required to represent its split sequences is $d \cdot l$, in which case the split sequence number is padded by setting the trailing $d \cdot (w - l)$ bits in the split sequence number to zero. For example, if $w = 4$, block 4 in Fig. 1 has the split sequence "2,1", so it is represented with the integer 10010000b, where the last four binary digits are padding. Unfortunately, in this scheme, there is no way to tell the partition level of a block from the split sequence number alone, since the d bit padding sequence of zeros is a legal split encoding (i.e., indicating the first child block). For example, the split sequence number 00000000b applies to both block 1 and the root block in Fig. 1. Thus, in our implementation, Morton codes of quadtree blocks are represented as pairs (s, l), where s is the fixed-width split sequence number and l is the partition level; internally, we actually store $w - l$ which represents the side length of the block.

Typically, the region of space covered by a quadtree (i.e., by the root block) is a hypercube, or a square in two dimensions – that is, the side lengths along all dimensions are equal. However, the side lengths do not necessarily have to be equal, so, in general, the quadtree data space has a hyper-rectangular shape. Clearly, all quadtree blocks will have the same side length proportions as the root block. Furthermore, the space coverage of any quadtree block b can be efficiently determined given the space coverage of the root block and the Morton code of b. In particular, the *natural coordinate system* of a quadtree with maximum partition level of w is such that the lower left corner of the root block is at the origin and its side length is 2^w (i.e., the side length of quadtree blocks of minimum size is 1). In this coordinate system, the split sequence number of the Morton code (s, l) of b is the result of applying bit interleaving to the coordinate value of the bottom left corner of b, whose side length is 2^{w-l}. Thus, the hypercube-shaped region covered by b in the natural coordinate system is obtained by "de-interleaving" the split sequence number s to obtain the lower left corner, and using the partition level l to obtain the side length. Any other space coverage of the root block simply means that an appropriate scaling and translation is applied on this hypercube to obtain the actual space coverage of b.

Figure 3a illustrates the Morton code order imposed on the quadtree blocks for the quadtree in Fig. 2. The contents of the MBI for this PMR quadtree are partially shown in Fig. 3b, where the order in the list corresponds to Morton code order. For further illustration of actual Morton codes, assume again that $w = 4$ (i.e., the side length of the data space in the natural coordinate system is $2^4 = 16$). The split sequence of the block labeled 18 in Fig. 3a is "3,0,2", which is represented with the integer 11001000b = 200 (last two binary digits are padding), so the Morton code for the block is $(200, 3)$. To obtain the space coverage of this block in the natural coordinate



a

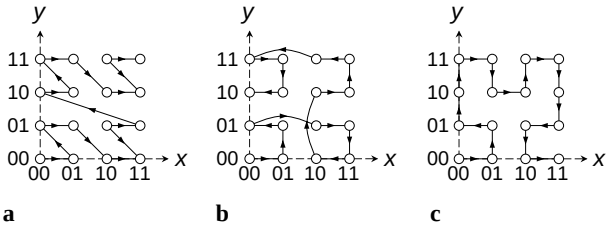
block	object
4	c
4	d
4	i
5	c
5	i
6	d
6	g
7	d
7	i
8	c
9	c
10	i
11	f
12	g
12	h
13	a
13	h
...	...

b

Fig. 3. a The PMR quadtree for the line segments in Fig. 2, with the quadtree blocks numbered in Morton code order. **b** Some of the corresponding items stored in the linear quadtree

system, we de-interleave the split sequence number to yield the lower-left corner of $(1000\mathrm{b}, 1010\mathrm{b}) = (8, 10)$, and get the side length from the partition level, yielding $2^{4-3} = 2^1 = 2$. If block 18 had to be split, the split sequence numbers of the child blocks would be 11001000b, 11001001b, 11001010b, and 11001011b. In other words, only d bits of the original split sequence number are modified. Similarly, the split sequence number of the parent block of block 18 is 11000000b.

Below, when we talk of the Morton code of a point p, we mean the result of applying bit interleaving to the coordinate values of p, after mapping into the natural coordinate system of the quadtree. Alternatively, the Morton code of p is the split sequence number for the quadtree block of side length 1 that contains p (assuming that such a quadtree block exists). Hence, Morton codes represent a mapping from d-dimensional points to one-dimensional scalars. When the d-dimensional points are ordered on the basis of their corresponding Morton codes, the order is called a *Morton* order [45], an example of a *space-filling curve*. This order is also known as a Z-order [49] 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, and each of these can be used to define an encoding. The most commonly used encoding methods for quadtree blocks are Morton, Hilbert, and Gray codes (Hilbert codes are based on Peano-Hilbert order and Gray codes [23] are related to Morton codes; see [1, 35] for a more detailed descriptions of these and other encoding methods, and for studies of their relative "goodness"). Figure 4 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 codes for quadtree blocks, e.g., computing the Morton code for subblocks, can be implemented through simple bit-manipulation operations. Nevertheless, Hilbert and Gray codes have the advantage that they preserve locality somewhat better, which



a b c

Fig. 4a–c. Ordering imposed by code values in a 4 by 4 grid when using **a** Morton code, **b** Gray code, and **c** Hilbert code

may reduce query cost [2]. 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.

3.3.2 B-tree

The MBI uses a B-tree to organize the quadtree contents[4], with Morton codes serving as keys. When comparing two Morton codes, we employ lexicographic ordering on the split sequence number and partition level. When only quadtree leaf nodes are represented in the MBI, which is the case for most quadtree variants, comparing only the split sequence number is sufficient, as the MBI will contain at most one block size for any given sequence number. For a quadtree leaf node with k objects, the corresponding Morton code 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

[4] 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 to some degree by a slightly greater storage requirement for the B^+-tree.

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 [16].

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 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. In other words, starting at the root node, we visit child nodes that intersect the object to insert, and add the object to any leaf nodes that are encountered. Thus, the CPU cost for inserting an object is roughly proportional to the depth of the leaf nodes intersecting it, due to the intersect tests that are performed during the traversal.

Fortunately, the cost of insertions can be considerably reduced by making use of the regularity of quadtree partitioning, effectively allowing us to short-circuit the traversal and yielding a cost that is roughly proportional to the number of leaf nodes intersecting the object. The key insight is that based only on the geometry of an object, we can compute the (hypothetical) quadtree block that minimally encloses the object. This is illustrated in Fig. 5a, where we indicate potential quadtree partition boundaries with broken lines. Hence, the insertion traversal can be initiated at the partition level of the minimum enclosing quadtree block (e.g., see Fig. 5b). Frequently, however, the object is completely enclosed by an existing quadtree leaf node, in which case the minimum enclosing quadtree block is inside, or coincides with, the existing leaf node (e.g., see Fig. 5c).

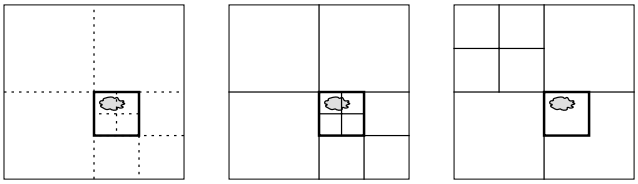


Fig. 5. a Computation of the minimum bounding block for an object, denoted by heavy lines. Broken lines indicate potential quadtree block boundaries. The minimum bounding block can **b** coincide with a nonleaf node or **c** be enclosed by a leaf node

A PMR quadtree insertion algorithm based on the idea of minimum enclosing quadtree block is shown in Fig. 6. Procedure InsertObject uses the functions ComputeEnclosingBlock and FindEnclosingNode to locate the smallest node in the quadtree index that contains *object*, and invokes Insert on that node. (At worst, *node* is the root node of the quadtree, in case *object* straddles the partition boundaries for the root 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 Split procedure splits the given leaf node, thereby turning it into a nonleaf node, and reinserts the objects into the appropriate child nodes; observe that the child nodes are not split even if the splitting threshold is exceeded. The first **foreach** loop in procedure Split makes use of the minimum enclosing block, as computed by ComputeEnclosingBlock, for objects that are fully enclosed in one of the child nodes (as determined by ChildContaining). The second **foreach** loop then reinserts objects that remain on *objList*, namely any object that intersects more than one of the child nodes and/or that is not fully enclosed by *node* itself.

The algorithm shown in Fig. 6 can be used for either a pointer-based implementation or a linear quadtree implementation of a PMR quadtree (e.g., the Morton Block Index), given an appropriate definitions of blocks and nodes and of the various utility routines. Thus, in the MBI implementation, *node* and *enclosingBlock* are both represented with a Morton code. Furthermore, the functions ChildContaining, ComputeEnclosingBlock, Intersects, and Size merely operate on object geometries and Morton codes, while the other routines (i.e., AddToLeaf, FindEnclosingNode, IsLeaf, MakeNonLeaf, ObjectCount, and ObjectList) obtain their results by accessing the MBI B-tree. In particular, AddToLeaf inserts into the B-tree, MakeNonLeaf deletes

```
procedure InsertObject(object) →
  enclosingBlock ← ComputeEnclosingBlock(object)
  node ← FindEnclosingNode(enclosingBlock)
  Insert(node, object)

procedure Insert(node, object) →
  if (IsLeaf(node)) then
    AddToLeaf(node, object)
    if (ObjectCount(node) > threshold) then
      Split(node)
  else
    foreach (childNode of node) do
      if (Intersects(object, childNode)) then
        Insert(childNode, object)

procedure Split(node) →
  objList ← ObjectList(node)
  MakeNonLeaf(node)
  /* make use of minimum enclosing block if possible */
  foreach (object in objList) do
    enclosingBlock ← ComputeEnclosingBlock(object)
    if (Size(enclosingBlock) < Size(node)) then
      childNode ← ChildContaining(node, enclosingBlock)
      AddToLeaf(childNode, object)
      Delete(objList, object)
  /* apply intersect check for remaining objects in objList */
  foreach (childNode of node) do
    foreach (object in objList) do
      if (Intersects(object, childNode)) then
        AddToLeaf(childNode, object)
```

Fig. 6. PMR quadtree insertion algorithm

from the B-tree, FindEnclosingNode and IsLeaf perform 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 code.

In a naive PMR quadtree insertion algorithm the InsertObject procedure would simply invoke Insert on the root node, and the first **foreach** loop in Split would not be used. By using the minimum enclosing blocks, the algorithm in Fig. 6 achieves significant CPU cost savings, due to a reduction in the number of intersection tests. Nevertheless, these savings are 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. Although not treated in the experimental section (Sect. 8), we observed a speedup of approximately 70–110% in terms of CPU cost over the naive insertion algorithm when bulk-loading 2D line segment data (Sect. 8.2), and around 40–80% for loading the same data using dynamic insertions.

5 Bulk-loading PMR quadtrees

PMR quadtrees represented with the linear quadtree method, such as our MBI implementation described in Sect. 3.3, perform well for dynamic insertions (especially with the insertion algorithm described in Sect. 4) and a wide range of queries. Nevertheless, we found that bulk-loading large data sets into MBI based PMR quadtrees with dynamic insertions takes a considerable amount of time relative to the size of the data set. As is true for most indexing structure, the primary reason for sub-optimal performance of dynamic insertions in this setting is the fact that successive insertions typically involve different disk blocks in the external representation, a B-tree in the case of the MBI, assuming arbitrary ordering of the data set. Thus, unless the entire structure fits into the B-tree buffer, each insertion is likely to require reading a B-tree block that has already been written to disk, so each B-tree block will eventually be written multiple times to disk. In addition to this excessive I/O, we identified several areas that exhibited considerable overhead in CPU time, the chief of which is the high cost of splitting quadtree nodes. In particular, when a quadtree node is split, references to objects must be deleted from the B-tree, and then reinserted with the Morton codes of the newly created quadtree nodes. Thus, in addition to local reorganizations within B-tree blocks, such sequences of deletions and insertions to the B-tree can cause repeated merging and splitting of the same B-tree blocks.

Our bulk-loading method addresses inefficiencies of dynamic insertions in terms of both I/O and CPU cost. The basic idea is to reduce the number of accesses to the B-tree as much as possible by storing parts of the PMR quadtree in main memory. The end result is that there are only insertions into the B-tree (i.e., no deletions), and those insertions occur in a strictly sorted order, which allows building the B-tree with minimal number of I/Os and with no CPU cost overhead for reorganizations. As shown below, the above properties can only be achieved by pre-sorting the data objects in a certain manner. Of course, the cost of pre-sorting must therefore be taken into account in the overall cost of our bulk-loading method. This is done in both the analysis presented in Sect. 7 and in the experiments conducted in Sect. 8.

The remainder of this section is organized as follows: In Sect. 5.1 we present an overview of our bulk-loading approach. Next, in Sect. 5.2, we present the details of our flushing algorithm, which frees up space if none is left in the main memory buffer. In Sect. 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 Sect. 5.4. Finally, in Sect. 5.5 we show how the MBI B-tree can be built efficiently and with a high storage utilization.

5.1 Overview

In our bulk-loading 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; we use the term *quadtree buffer* to refer to the memory block used for the memory-resident portion of the quadtree. In order to maintain compatibility with the MBI-based PMR structure, we use Morton codes to determine the space coverage of the memory-resident quadtree blocks. Note that it is not necessary to store the Morton codes 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 codes[5]. Thus, we chose to store the Morton codes 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 Fig. 7, 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 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, and we can flush quadtree blocks that cover points with lower Morton codes. As demonstrated in Fig. 7, this strategy can be thought of as a variation of plane sweep, where the customary sweep line is replaced by a piecewise linear curve (e.g., the thick boundary in the figure).

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.

[5] For most other encoding methods for quadtree blocks, such as Hilbert and Gray codes, this overhead can be expected to be even greater.

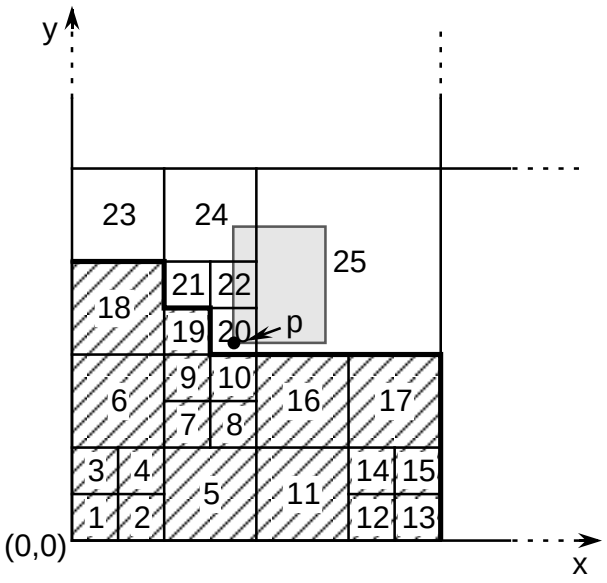


Fig. 7. A portion of a hypothetical quadtree, where the leaf nodes are labeled in Z-order. The shaded rectangle is the bounding rectangle of the next object to insert

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

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 Fig. 7).
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 Fig. 7, in increasing order of the labels):
   (a) if the quadtree block intersects p (e.g., the leaf block labeled 20 in Fig. 7), then terminate the process;
   (b) otherwise, insert the leaf block into the MBI.

Figure 8 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 Fig. 8 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 Fig. 7, 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 quadtree buffer,

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.

```
procedure INSERTOBJECT(object) →
  if (available memory falls below a threshold) then
    p ← lower left corner of the bounding rectangle of object
    FLUSHNODES(root, p)
  /* remainder of procedure same as in Fig. 6 */
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)
procedure FLUSHSUBTREETOMBI(node, freeNode) →
  if (node has already been flushed) then
    return
  if (ISLEAF(node)) then
    foreach (object in node) do
      MBIINSERT(node, object)
  else
    foreach (childNode of node) do
      FLUSHSUBTREETOMBI(childNode, true)
  if (freeNode) then
    FREENODE(node)
  else
    mark node as having been flushed and turn into empty leaf node
```

Fig. 8. Pseudo-code for flushing process

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 particular, let m_p be the Morton code of p, and let $m_{\rm lo}$ and $m_{\rm hi}$ be the smallest and largest Morton codes, respectively, for a quadtree block b ($m_{\rm lo}$ is the Morton code of its lower-left corner, while $m_{\rm hi}$ 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 $(0, 0)$, $m_{\rm hi}$ is the Morton code for the point $(3, 3)$. Testing for intersection of b and p is equivalent to checking the condition $m_{\rm lo} \leq m_p \leq m_{\rm hi}$. 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 $m_{\rm lo}$ and m_p must match (the $2w_b$ low-order bits of $m_{\rm lo}$ are all 0 and those of $m_{\rm hi}$ are all 1).

5.3 Reinsert freeing

The problem with the flushing algorithm presented in Sect. 5.2 is that it may fail to flush any leaf nodes, and thus not free up any memory space. In the example in Fig. 7 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). This condition rarely applies to a large number of objects, at least not for low-dimensional data and reasonable quadtree buffer sizes as discussed in Sect. 7. 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. In addition, 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 Sect. 5.5 (i.e., adapted to tolerate slightly out-of-order insertions). Furthermore, our bulk-insertion algorithm would not be applicable (although a usually more expensive variant could be used; see Sect. 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 Sect. 5.4 we discuss how to extend a sorting algorithm to handle reinsertions). This is illustrated in Fig. 9 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 code. 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.

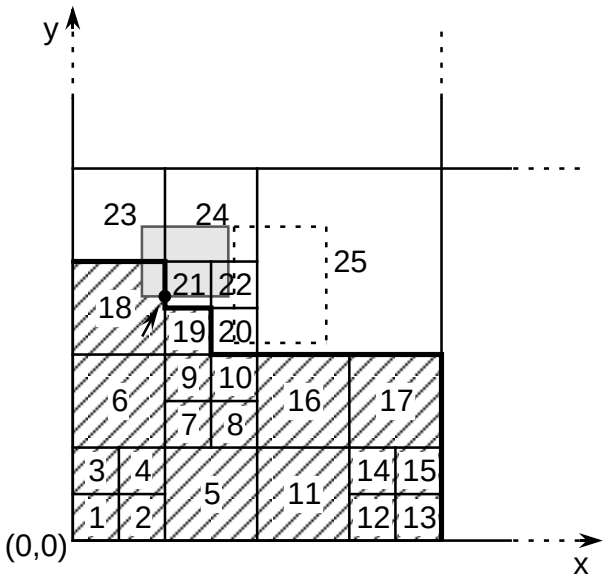


Fig. 9. An example of an object that is to be reinserted (shaded rectangle). The striped region represents quadtree nodes that have been flushed, while the broken lines indicate the bounding rectangle of the object that was inserted last

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 Fig. 9. 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 Sect. 8.6). Another possible strategy is to visit the leaf nodes in decreasing Morton order (i.e., the ones with the highest Morton codes 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 the number of invocations of flushing or reinsertion freeing is reduced (i.e., cost savings of fewer reinsertions of the second strategy were overwhelmed by the cost increase of more traversals when performing flushing or reinsertion freeing).

An important point is that an object can only be reinserted a limited number of times, thus ensuring that we do not reinsert the same object indefinitely. This is guaranteed by the property that reinsertions always produce sort keys that are greater than the sort key of the object that was last delivered by the sorting phase, since we do not reinsert objects occurring in the leaf node intersecting the sort key of the object that was inserted last (e.g., objects occurring in leaf node 20 in Fig. 9 are not reinserted). Thus, this property and the correctness of the sorting phase guarantee that objects are delivered by the sorting phase in a strictly non-decreasing order of sort keys and that an object is never reinserted twice with the same sort key. Another way to view this is that some progress always occurs between two successive reinsertions for the same object.

The total number of insertions (original and reinsertions) for an object o is bounded from above by the number of quadtree nodes (leaf and nonleaf) in the final quadtree that are intersected by o. A tighter bound can be obtained by assuming that we apply a somewhat more expensive method of constructing sort keys during reinsertions than the one described above. In particular, let b be the block used to compute the sort key for object o for some reinsertion of o. To compute the new sort key s for o, we compute the minimum bounding rectangle of the portion of o that is inside b, and use the lower-left corner of this rectangle in computing s. This construction guarantees that when o is eventually (re)inserted into the memory-resident quadtree with the sort key s, the then-current leaf node b_l that intersects s is either a leaf node in the final quadtree, or b_l is a nonleaf node in the final quadtree having at least two child nodes that intersect o. Thus, the number of insertions for o is at most twice the number of leaf nodes intersecting o.

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 Sect. 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., [4,37,40,41,52, 59,60]).

We implemented two external sorting algorithms suitable for our application. The first algorithm is a variation of the standard distribution sort [3], 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). In addition, the algorithm is difficult to adapt to support reinsertions (Sect. 5.3) in an efficient manner. The second algorithm that we implemented is external merge sort [3]. This algorithm

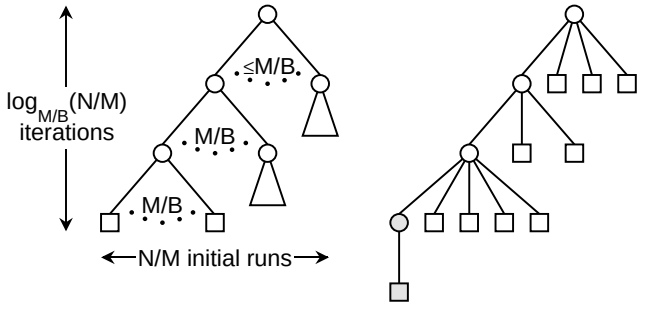


Fig. 10a,b. Depiction of external merge sorting: **a** regular, and **b** with reinsertions. In **a**, squares represent runs created from the input while circles represent merged runs. In **b**, the white squares represent active runs, the white circles represent future merged runs, and the shaded square represents a partial run being created in memory

has the advantage of being provably optimal, having an I/O cost of $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, where N is the number of data objects, M the number of objects that fit into an internal memory buffer used for sorting, and B is the number of objects in a block transfer (typically the size of a disk page). Furthermore, in the presence of reinsertions, it is at worst only slightly sub-optimal. 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 [3] 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[6], so $R = M/B$. A depiction of the process is shown in Fig. 10a. The squares represent runs created from the input while circles represent merged runs. The run represented by the circle at the "root" contains the entire sorted data set.

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 modified algorithm is only slightly sub-optimal. In particular, if N_s is the number of objects plus the number of reinsertions, the modified algorithm achieves a comparable I/O performance as sorting N_s objects from scratch. The general scenario for the modified sorting algorithm is as follows: Initially, the sort process receives a set of N objects, each with some sort key, which are placed into the *sort set*. Subsequently, the sort process must respond to `getNext` and `reinsert` requests, until no object remains in the sort set. On `getNext` requests, the sort process delivers the object with the smallest sort key in the sort set (and removes it from the set), while on `reinsert` requests, an object that has been previously delivered is inserted again into the sort set with a different sort key. We use the term *object instance* to denote an object with a particular sort key; hence, each reinsertion results in a new object instance, and the total number of object instances is N_s. Clearly, if the sort key in a reinsertion is no smaller than the sort key of the object that was last delivered (which is the case for our bulk-loading algorithm), the sort keys of the objects delivered by the sort process are monotonically non-decreasing. Furthermore, assuming that `reinsert` requests only occur for objects not currently present in the sort set (which is also the case for our bulk-loading algorithm), the number of object instances present in the sort set never exceeds N.

The basic idea behind our modified merge sort algorithm is to use a portion of the sort buffer (i.e., the internal memory buffer of size M used by the algorithm) to store newly reinserted objects, which are maintained in a heap structure termed the *reinsert heap*, allowing fast retrieval of the object with the smallest sort key. The remainder of the sort buffer is used to buffer the merging of a set of *active* runs, since each run being merged requires buffer space of B objects. In particular, the sort proceeds as in the original merge sort algorithm until reaching the final iteration, where $m' \leq M/B$ runs are being merged. These m' runs become the initial set of active runs, which are used by the algorithm to respond to `getNext` requests, in the same way that runs are merged in the original algorithm. Furthermore, as objects get inserted into the reinsert heap due to reinsertions, the reinsert heap also participates in this merging. This process can go on until the sort buffer is full upon a `reinsert` request, i.e., when the reinsert heap contains $M - a \cdot B$ objects, where a is the number of active runs. At this point, there are two options: 1) write the reinsert heap to disk as a sorted run; or 2) merge the active runs into a single sorted run. With the first option, we obtain $a+1$ active runs and an empty reinsert heap, while with the second we obtain a single active run, with room in the sort buffer for the reinsert heap to grow. We adopt the convention that option 1 is taken only if the reinsert heap contains at least $M/2$ objects, so option 2 is taken only if the number of active runs is at least $\frac{1}{2}M/B$.

The algorithm that we sketched above can be expected to perform well as long as the number of reinserted objects in the sort set is not too large, which is usually the case in our bulk-loading algorithm. Unfortunately, in extreme cases, the number of I/Os can be much larger than the $O(\frac{N_s}{B}\log_{M/B}\frac{N_s}{B})$ that we are aiming for. In particular, in the proof of the optimality of merge sort, we make use of the fact that each object gets written into approximately $\log_{M/B}(N/M)$ increasingly large sorted runs. However, in the algorithm above, a large number of reinsertions may cause many object instances to be written into substantially more than $\log_{M/B}(N_s/M)$ runs, leading to greatly sub-optimal behavior. To see why, consider the scenario that the original m' runs have been merged into one, and subsequent overflows of the sort buffer have caused the creation of so many new active runs that the sort buffer is

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

full while the reinsert heap contains less than $M/2$ objects. Thus, upon the next `reinsert` request, we would merge all the active runs into one, which will include objects from the original m' runs (i.e., that have not been reinserted). Depending on the number of reinsertions, the above scenario may occur arbitrarily often, thereby causing object instances to be written arbitrarily often into new runs.

The cause of the above dilemma is that the algorithm results in runs consisting of object instances of different "ages", where the age of an object instance is defined as the number of times that the object instance has been written into a new run. To resolve the dilemma, our modified sorting algorithm maintains a hierarchy of active runs, as depicted in Fig. 10b, where the level of an active run depends on the age of the object instances in the run. Thus, the partial run being formed in the reinsert heap is at the lowest level (the shaded square in Fig. 10b), indicating an age of zero, the runs created by writing out the reinsert heap at the level above, and so on. Furthermore, the m' original runs are approximately at level $\lfloor \log_{M/B}(N/M) \rfloor$. When a merge is necessary (i.e., by the conditions outlined above), the algorithm merges runs at the level in the hierarchy containing the greatest number of runs. Notice that the active runs are continuously being read from in response to `getNext` requests. This means that object instances do not necessarily travel up the entire hierarchy, and that runs at lower levels may become depleted and thereby removed from the hierarchy.

An advantage 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 large as M/B, but never smaller 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 $\log_{\frac{M}{2hB}} \frac{N_s}{B} < 2 \log_{M/B} \frac{N_s}{B}$. 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 Sect. 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 using B-tree/B$^+$-trees bulk-loading algorithms that exploit the sorted insertion order (e.g., [50, 53]). In essence, such algorithms simply fill the leaf nodes of the tree in order, which also leads to ordered insertion into the non-leaf nodes. 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 nearly 100%. Thus, we can achieve substantially better storage utilization than that typically resulting from building B-trees, which is about 69% for random insertions [61].

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 Sect. 5.3 (i.e., flushing nodes that may be needed later using a heuristic) can cause out of order insertions. As another example, in Sect. 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 fall beyond the range of keys already in the tree (which is the case when inserting into a previously empty B-tree).

For our experiments, we implemented a B-tree packing algorithm that is similar to that of [50] (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; in contrast, the algorithm of [40] is not applicable in this context, as it requires knowing the number of records to insert). However, our algorithm has the advantage that it always maintains a fully connected tree structure, which enabled us to adapt it to gracefully handle situations where B-tree insertions occur somewhat out of order[7]. Of course, it is not possible to ensure 100% storage utilization in the face of out-of-order insertions. In our experiments, we found that a reasonable compromise was achieved by aiming for 85% storage utilization in the algorithm in such cases (which affects how nodes are split). A similar approach can be taken when using B$^+$-trees, leading to an algorithm related to the B$^+$-tree bulk-loading algorithm described in [53].

6 Bulk-insertions for PMR quadtrees

Our 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

[7] When insertions occur in order, only the rightmost B-tree node on each level is affected by insertions, so no tree traversals are necessary. When our algorithm detects that an insertion occurs that does not fall into the current node, it traverses to the proper leaf node and makes it the current node in case the succeeding insertions fall into that node.

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 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 11 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 Fig. 11a, 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 Fig. 11b). 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 Fig. 11c). In this case, b_n is split, and b_o is recursively checked against the new child nodes of b_n (in Fig. 11c, case 1 would apply to the new SW child of b_n).

6.2 Algorithm

Our merge algorithm is shown in Fig. 12. The algorithm modifies procedures FLUSHNODES and FLUSHSUBTREETOMBI from Fig. 8, 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

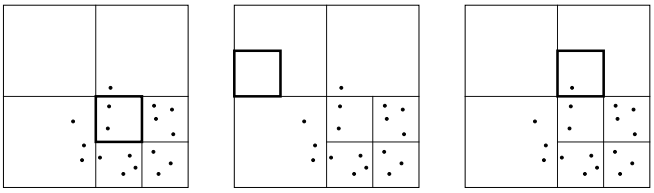


Fig. 11a–c. A simple PMR quadtree T_n consisting of points and the three cases that arise when merging with an existing quadtree T_o with our bulk-insertion algorithm: **a** A leaf node in T_o coincides with a nonleaf node or a nonempty leaf node in T_n, **b** a leaf node in T_o is contained in an empty 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

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 codes. 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 Fig. 12 that empty leaf nodes are not represented in the disk-based quadtree. In addition, 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

The cost of bulk-inserting a data set into an existing quadtree is at least as large as the cost of bulk-loading the combined data set minus the cost of bulk-loading the original data set, since our bulk-insertion algorithm is based on our bulk-loading algorithm. In other words, letting $c_L(S)$ denote the cost of bulk-loading a quadtree index with a data set S, and $c_I(S_1, S_2)$ denote the cost of bulk-inserting the data set S_2 into an existing index for data set S_1, the relation

$$c_I(S_1, S_2) \geq c_L(S_1 + S_2) - c_L(S_1)$$

holds. Furthermore, define the "excess" cost of bulk-inserting S_1 as the cost $c_I(S_1, S_2) - (c_L(S_1 + S_2) - c_L(S_1))$. Clearly,

```
procedure FlushNodes(node, p, oldTree) →
  if (not IsLeaf(node)) then
    MergeSubtrees(node, oldTree)
    /* remainder of procedure is same as in Fig. 8 */

procedure FlushSubtreeToMBI(node, freeNode, oldTree) →
  MergeSubtrees(node, oldTree)
  /* remainder of procedure is same as in Fig. 8 */

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 Fig. 11a) */
      do
        Insert(node, CurLeafObject(oldTree))
        NextLeafNode(oldTree)
      while (EqualCoverage(node, CurLeafNode(oldTree)))
    elseif (IsLeaf(node)) then
      if (IsEmpty(node) then
        /* current in oldTree is same size or smaller (see Fig. 11b) */
        do
          MBIInsert(CurLeafNode(oldTree))
          NextLeafNode(oldTree)
        while (Contains(node, CurLeafNode(oldTree)))
      else
        /* current in oldTree is smaller (see Fig. 11c) */
        Split(node)
```

Fig. 12. Pseudo-code for quadtree merging

based on the above observations, the excess cost is nonzero, and the lower its value, the better the bulk-insertion algorithm.

We believe that our bulk-insertion algorithm is very efficient in terms of the excess cost. From the standpoint of CPU cost, the excess is primarily due 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, intersection tests, which are a major component of the CPU cost, should not significantly contribute to the excess CPU cost. Furthermore, besides the cost of accessing the intermediate quadtree, the bulk of the CPU cost of MergeSubtrees is involved in work that must also be performed when bulk-loading the combined data set, 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, the excess cost comes from writing out the intermediate quadtree (during bulk-loading) and reading it back in (during bulk-insertion). 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.

Our bulk-insertion algorithm essentially merges a new quadtree being built in memory with an existing disk-resident quadtree, and writes out a new combined disk-resident quadtree. It is easy to transform this "merge-based" algorithm into an "update-based" algorithm that instead updates 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 Sect. 5.5 is less effective when adapted to handle updates of an existing B-tree. Thus, the excess I/O cost is often higher than with our method due to worse storage utilization, and, in addition, the excess CPU cost is typically significantly higher due to the cost of updating the existing B-tree nodes. A further advantage of the merge-based algorithm over the update-based one 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 Sect. 8.7, where we report expected results for both variants, the update-based variant is sometimes more efficient than our merge-based one 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 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. In addition, 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 Analytic observations

In this section we make some observations about the execution cost of our bulk-loading algorithm. The discussion is for the most part informal, and is meant to give insight into general trends, rather than being a rigorous treatment. We make the simplifying assumption that the objects occupy a fixed amount of storage (such as is the case for elementary geometric objects like line segments and rectangles, but unlike for complex ones like polygons). We also assume a disk-based representation of the quadtree that has similar characteristics as the Morton Block Index (MBI), described in Sect. 3.3.

Our experiments, as reported in Sect. 8, 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. First, however, we introduce the symbols used in evaluating the costs, and discuss important issues that affect the cost.

7.1 Preliminaries

The I/O and CPU cost of our algorithm can be attributed to three activities: 1) sorting; 2) construction of quadtree partitioning; and 3) B-tree loading. Of these, the second activity does not directly involve I/O operations. As we shall see, the cost of each activity depends on a) its input size, b) the number of internal memory buffers available to the overall process, and c) the unit of block transfer for I/O operations. The following table defines symbols that denote quantities of relevance in these cost factors:

	The number of:
N	data objects
N_s	object instances (counting reinsertions) for sorting
N_q	q-objects
N_b	B-tree entries
M	objects that fit into internal memory buffers
M_s	objects that fit into sorting buffer
M_q	objects that fit into quadtree buffer
M_b	objects that fit into B-tree buffer
B	objects that fit into a disk page (or the desired unit of block transfers)
B_s	objects in a disk page for sorting
B_b	entries in a B-tree node

Clearly, N is the input size for the overall bulk-loading process, but the input size of sorting and the partitioning activities is N_s, while it is N_b for the B-tree loading. Below, we examine the relationship between N and the different N_* values (i.e., N_s, N_q, and N_b). As we shall see, given some reasonable assumptions on the distribution of the location and sizes of the objects, the different N_* values are asymptotically the same as N. Furthermore, the different M_* and B_* values can also be shown to be asymptotically equivalent to M and B, again, given some assumptions (but more modest ones than for N). In particular, assuming the same disk page size in the B-tree as for sorting, B_b is slightly smaller than B_s, since each entry in the B-tree occupies somewhat more space (at least for the nonleaf nodes, which store pointers to nodes at the next lower level). However, the difference in entry sizes is only by a small constant, so both B_b and B_s can be simplified to B without affecting asymptotic results. By the same token, the differences in entry sizes can be ignored for the M_* values. Moreover, by assuming that the total buffer space is equally divided among the different activities (i.e., the sorting activity is performed concurrent with the quadtree building activity, as we suggest in Sect. 5.4), we can simplify asymptotic bounds by using M for M_s, M_q, and M_b, the buffer sizes of each individual activity. Nevertheless, in our experiments (see Sect. 8), we found that a relatively modest buffer space was sufficient for the quadtree buffer, while for the B-tree loading, buffering of only as much as a single node at each level is sufficient (see Sect. 5.5). Thus, most of the total buffer space can be allotted to sorting.

The primary reason for the fact that analyzing our algorithm is difficult is that its behavior depends on the distribution of the geometric positions and sizes of the spatial objects. In particular, this applies to the relationship between N and the quantities N_s, N_q, and N_b. Full analysis of the expected behavior requires complex modeling of the geometric distribution, which is outside the scope of this paper. Furthermore, the geometric distribution for specific applications may differ considerably from that assumed by the analysis. Thus, instead, we make in Sects. 7.1.1 and 7.1.2 some informal observations about the relationships.

7.1.1 Relationship between N, N_q, and N_b

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_b, 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[8], 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,

[8] 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.

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_b relative to N for actual data sets. In Sect. 8 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 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_b 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_b 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 data of this nature, but the same can be said about most other spatial index structures (such as the R-tree).

7.1.2 Relationship between N and N_s

In Sect. 5.3, we point out that an object intersecting q leaf nodes can be subject to no more than $2q$ insertions into the memory-resident quadtree (original and reinsertions), assuming that an appropriate method of computing sort keys for reinsertions is applied. Thus, the total number of insertions for N data objects is $O(N_q)$. As we outlined in Sect. 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. As we argue in Sect. 7.1.1, N_q is typically proportional to N, in which case N_s is also proportional to N based on the above argument. Even in situations where N_q is much higher than N, reinsertions can still be expected to be relatively rare, since reinsertions only occur if the flushing algorithm fails to free any memory. The informal analysis below, although simplistic, suggests that the latter situation does not arise frequently.

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 Fig. 7. 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 Fig. 7 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 500 kB and a data file size of about 2.3 GB.

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 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 9 MB 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.

The experiments reported in Sect. 8 corroborate the above argument, as we observed no reinsertions except when we explicitly aimed at producing them (Sect. 8.6). In the latter experiments, we used a real-world data set of 260K line segments (employing a very small quadtree buffer) and a synthetic one of 10,000 highly overlapping line segments (each of which intersected about 110 quadtree nodes on the average). For the real-world data set, we found that N_s was only about 8% greater than N. However, for the synthetic data set, N_s was more than 8 times greater than N. Nevertheless, a

[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.

multiple of 8 is actually modest in this case being that N_q was 110 times greater than N, implying that the sorting cost was overwhelmed by the cost of quadtree construction and B-tree loading.

7.2 I/O cost

With the groundwork laid down in Sects. 7.1 and 5.4, it is a straight-forward exercise to establish the I/O cost of our bulk-loading method. In particular, recall that only the sorting and B-tree loading activities perform I/O operations. We account for each one separately. First, as we outlined in Sect. 5.4.2, the I/O cost of sorting N_s objects in the face of reinsertions is $O(s\frac{N_s}{B}\log_{M/B}\frac{N_s}{B})$, where s is less than 2 for all practical values of N, M, and B. Second, with the use of B-tree packing, as presented in Sect. 5.5, the I/O cost of the B-tree loading is $O(\frac{N_b}{B})$, since each B-tree[10] node in the MBI is written out only once (with a constant storage utilization) and never read. The overall I/O cost of our bulk-loading algorithm is therefore $O(\frac{N_b}{B} + s\frac{N_s}{B}\log_{M/B}\frac{N_s}{B})$. This simplifies to $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, the lower bound on the I/O cost of indexing (e.g., see [9]), under the fairly reasonable assumptions outlined in Sect. 7.1 and the assumption that s is a constant.

One way to verify the above cost formula is to perform experiments with data sets of various different sizes, and then attempt to fit the cost formula to the actual experiment results. Given the results for the synthetic line segment data presented in Sect. 8.2, we found that the I/O cost (or, more precisely, the portion of the execution time that was due to I/Os) showed an excellent fit to the formula $aN\log N + bN + c$, where the coefficient b appeared to have more significance than a, being nearly ten times greater. For the real-world line segment data in the same section, the fit was also good, with the coefficient b overwhelming a in significance. In other words, the actual I/O cost appeared to be nearly linear in N, with a smaller term that was proportional to $N\log N$. While the number of data points used in the curve fitting was admittedly too small (i.e., three in both cases) to draw a firm conclusion, it does nevertheless provide some indication.

7.3 CPU cost

In analyzing the CPU cost of our algorithm, we treat each of the three activities separately (i.e., sorting, quadtree partitioning, and B-tree loading, as mentioned in Sect. 7.1). First, observe that the CPU cost of sorting N objects with the external merge sorting algorithm given in Sect. 5.4.1 is roughly proportional to the number of comparison operations. The average number of comparison operations per object when constructing the initial runs is $O(\log M)$. Each merge step gives rise to $O(\log\frac{M}{B})$ 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 (Sect. 5.4.2), sorting remains nearly optimal, or close to $O(N_s\log N_s)$.

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], or roughly $O(N_q\log N_q)$ without assuming constant number of q-objects. 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 $O(N_q)$. 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 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 Sect. 5.5, the CPU cost is proportional to the number of inserted items, N_b. This is due to the fact that each inserted item goes directly to its final destination, without being subsequently moved, for a constant cost for each insertion into a B-tree leaf node. The cost of inserting an item that is destined for a B-tree nonleaf node is proportional to the node level relative to the leaves (e.g., $O(\log N_b/B)$ for insertions into the B-tree root node). However, since the number of items on each level decreases geometrically in B, the average cost per insertion remains constant. Hence, the total CPU cost of B-tree packing is $O(N_b)$.

To summarize, we saw that the asymptotic CPU cost was roughly $O(N_s\log N_s)$ for sorting the objects, $O(N_q\log N_q)$ for constructing the quadtree in memory (given our simplifying assumptions), and $O(N_b)$ for building the B-tree. Thus, we see that in an ideal situation (i.e., if the data distribution

[10] The same would hold for the B$^+$-tree.

[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.

is not too skewed and the assumptions outlined in Sect. 7.1 hold), we can expect the total CPU cost of our bulk-loading algorithm to be roughly $O(N \log N)$, or equal to the lower bound on the CPU cost of indexing.

We also verified the above derivation by correlating with experimental results, in the same manner as we showed in Sect. 7.2. For the CPU cost, however, we found that the coefficient b when fitting to $aN \log N + bN + c$ was an even stronger influence than we found for the I/O cost, exceeding a by a wide margin. This suggests that the CPU cost for this kind of data is essentially proportional to N.

8 Empirical results

Below, we detail the results of a number of experiments which show the performance of the PMR quadtree bulk-loading technique presented in this paper. The remainder of this section is organized as follows: In Sect. 8.1, we present various details about the experimental setup. In Sect. 8.2 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 Sect. 8.3 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 Sects. 8.4 and 8.5 we show how well our method does with other types of data, multidimensional points and two-dimensional polygons, again using SAND. In Sect. 8.6, we study the performance of the algorithm when no node can be flushed and reinsert freeing must be used. In Sect. 8.7 we examine how well our bulk-insertion algorithm for PMR quadtrees performs. In Sect. 8.8, we establish how our bulk-loading algorithm compares to two bulk-loading techniques for R-trees. Finally, in Sect. 8.9 we summarize the conclusions drawn from our experiments.

8.1 Experimental setup

We implemented the techniques that we presented in Sects. 5 in C++ within an existing linear quadtree testbed (described in Sect. 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 [31]. 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 64 MB 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[12]. The use of raw disk partitions is another reason we obtained lower speedup than in [31], 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 to 8. Larger splitting thresholds make our 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 4 kB, while node capacity varied between 50 and 400 entries, depending on the experiment. The data files used in the experiments are available online [30].

The sizes of the data sets we used in our experiments were perhaps modest compared to some modern applications. However, we compensated for this by using a modest amount of buffering, limiting the space occupied by the pointer-based quadtree to 128 kB. 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 512 kB. A sort buffer size of 256 kB 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 1 MB) 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 Sect. 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.

8.2 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 [17]. 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 [44]. 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 end points. 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.

[12] In other words, in our experiments, I/O operations block the CPU until their completion.

Table 1. Details on line segment maps

Data set	Number of line segments	Avg. q-edges per segment	File size (kB)	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 2. Summary of PMR quadtree loading methods used in experiments

Method	B-tree buffering	Quadtree bulk-loading	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 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.

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 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 codes, 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 Sect. 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 the bulk-loading method, the B-tree packing algorithm (see Sect. 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 Sect. 5.4.

Table 3 shows the execution time for loading PMR quadtrees for the six data sets using the five loading methods. Figure 13 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 bulk-loading 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 bulk-loading ("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 bulk-loading. Curiously, the average cost for Roads is smaller for all

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

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

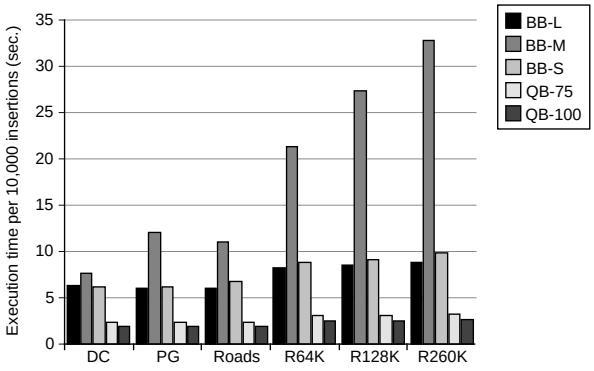


Fig. 13. Execution time per 10,000 line segments for building quadtrees for the six data sets

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).

A better representation of the experiment results for comparing the five different loading methods is shown in Fig. 14. The figure shows the speedup of "QB-100", quadtree bulk-loading 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 Fig. 15. 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 bulk-loading. This shows that the savings in execution time yielded by quadtree bulk-loading are, for the most part, caused by reduced CPU cost (the time for performing I/O is only 1.3–2.9 s per 10,000 insertions for all but "BB-M"). For "BB-M", the proportion of time spent on I/O gradually

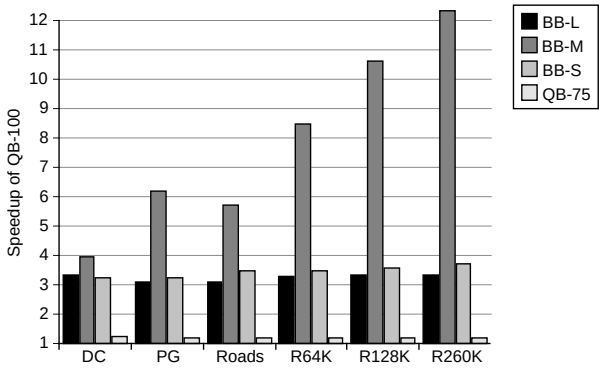


Fig. 14. Speedup of "QB-100" compared to the other four loading methods for line segment data

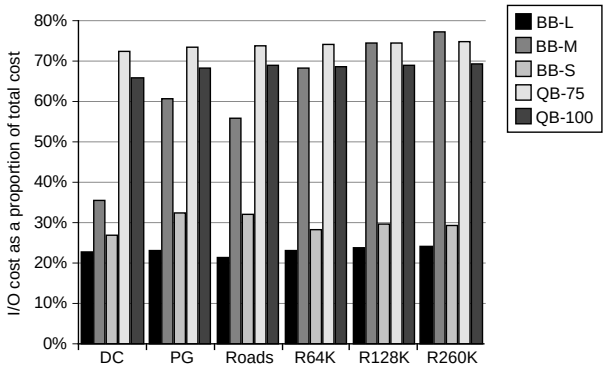


Fig. 15. Proportion of execution time spent on I/O operations for the five loading methods for line segment data

increases with larger data sizes as B-tree buffering becomes less effective on unsorted data.

8.3 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 Sect. 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 geometry 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 4 kB 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

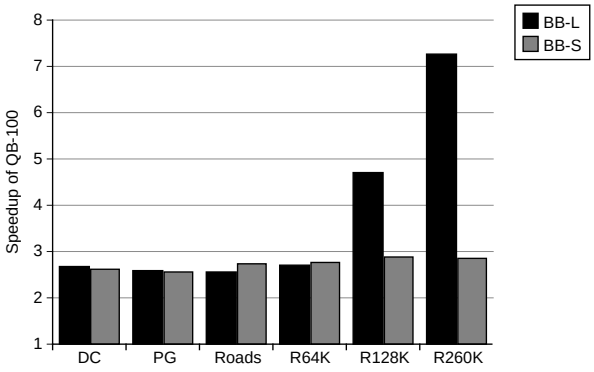


Fig. 16. Speedup of "QB-100" compared to the other methods for line segment data, using object table approach

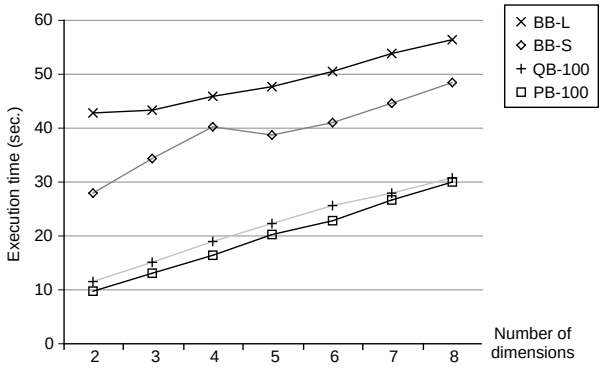


Fig. 17. Execution time for building PMR quadtrees for point data sets of varying dimensionality (using "BB-L", "BB-S", and "QB-100", described in Table 2) and a quadtree bulk-loading algorithm specialized for point data (denoted "PB-100")

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 16 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.

8.4 Multidimensional point data

Next, we examine the effect of the dimensionality of the space on the performance of our bulk-loading methods, 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 [20]. We used SAND for these experiments, storing the point geometry directly in the index. We

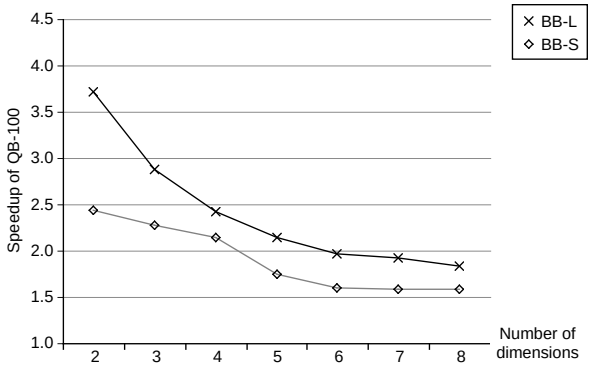


Fig. 18. Speedup of "QB-100" compared to "BB-L" and "BB-S" for point data sets of varying dimensionality

compare using the loading methods "BB-L", "BB-S", and "QB-100" in Table 2, in addition to a quadtree bulk-loading algorithm specialized for point data (denoted below by "PB-100"). Figure 17 shows the execution time of building the quadtree, while Fig. 18 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 bulk-loading 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. Observe that with our techniques, bulk-loading point data into a PMR quadtree takes nearly the same time as with a quadtree bulk-loading method specialized for point data ("PB-100").

8.5 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 40 MB of disk space. We performed this

experiment in SAND with the same loading methods as before. This time, we used a 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 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"[13]. 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 8 MB), 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 Fig. 19 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 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 Fig. 19). 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).

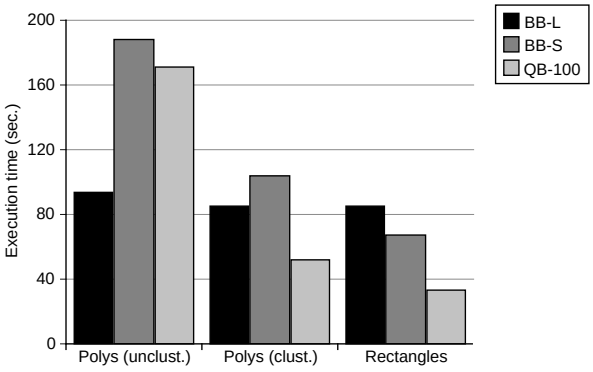


Fig. 19. Execution time for building PMR quadtrees for polygon data set (labels of bars denote loading methods from Table 2). "Polys (unclust.)" denotes building the quadtree on an unclustered polygon relation, "Polys (clust.)" denotes building it on an spatially clustered polygon relation, while "Rectangles" denotes building it on the bounding rectangles of the polygons

8.6 Reinsert freeing

In Sect. 5.3 we described a strategy we termed reinsert freeing that is used if the flushing algorithm fails to free any memory. Although reinsertion freeing may seem somewhat complicated, we actually found it to be fairly simple to implement. Furthermore, as shown in the next set of experiments, reinsertion freeing adds a relatively small overhead to the bulk-loading process.

In these experiments, 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 quadtree buffer size in the bulk-loading method to only 8 kB. 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 8,000 nodes or about 32 MB. 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 bulk-loading was better at lower splitting thresholds, however). For both

[13] 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.

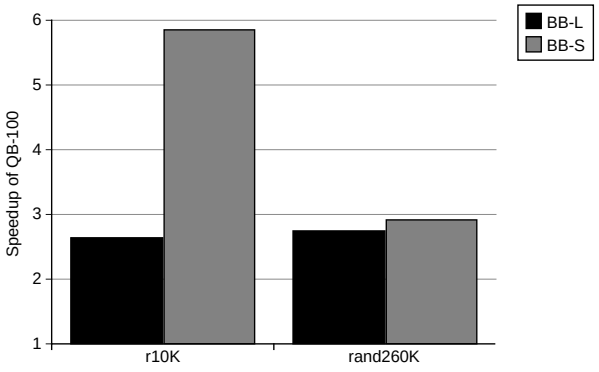


Fig. 20. Speedup of "QB-100" compared to the other methods when re-insertions are needed (labels of bars denote loading methods from Table 2)

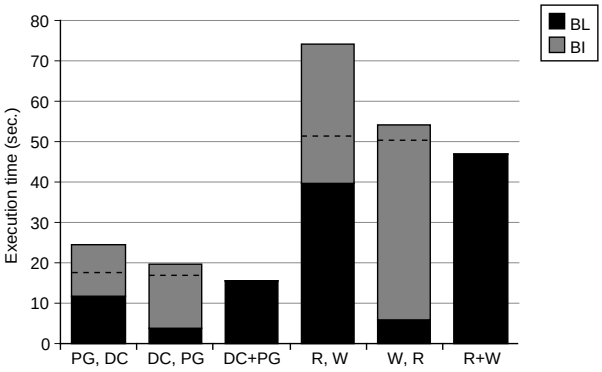


Fig. 21. Execution time for bulk-loading (indicated by the bars labeled "BL") and bulk-insertions (indicated by the bars labeled "BI") for two pairs of data sets. The portions of the bars above the broken lines indicate the excess I/O cost, i.e., the I/O overhead of the combined bulk-loading/bulk-insertion operations compared to bulk-loading the combined data set. "R" and "W" denote the "Roads" and "Water" data sets, respectively

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, Fig. 20 shows that quadtree bulk-loading 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 3,000 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 bulk-loading 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 Fig. 14, where reinsertions were not needed.

8.7 Bulk-insertions

The next set of experiments investigates the performance of PMR quadtree bulk-insertions (see Sect. 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 128 kB) for the combined quadtree, which allowed writing to disk multiple nodes at a time[14]; another solution would be to store the existing quadtree and the combined quadtree on different disks.

Figure 21 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 excess I/O cost (see Sect. 6), i.e., the cost of writing (during the bulk-load) and reading (during the bulk-insertion) the intermediate PMR quadtree. Clearly, the excess I/O cost represents nearly all the excess cost of the bulk-insertion algorithm in terms of execution time. Interestingly, the remainder of the excess cost 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 excess I/O cost is proportional to size of the existing data set in relation to the combined data set).

In Sect. 6.3 we discussed an alternative, update-based, 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 22 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 pack-

[14] Such multi-block I/Os are commonly used in bulk-loading and bulk-insertion methods to amortize the cost of disk seeks over multiple blocks; e.g., [47].

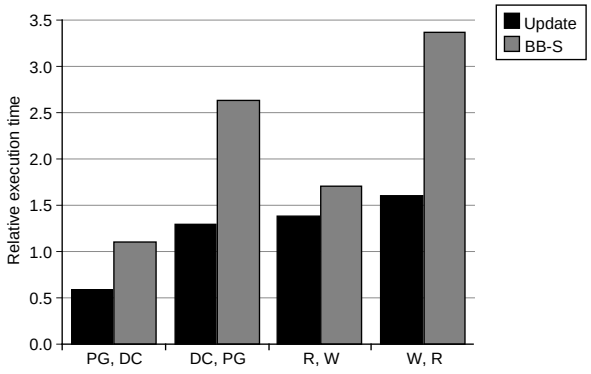


Fig. 22. Execution time of two alternative bulk-insertion methods relative to the merge-based PMR quadtree bulk-insertion algorithm. "Update" denotes the update-based variant of our algorithm, while "BB-S" denotes dynamic insertions (see Table 2)

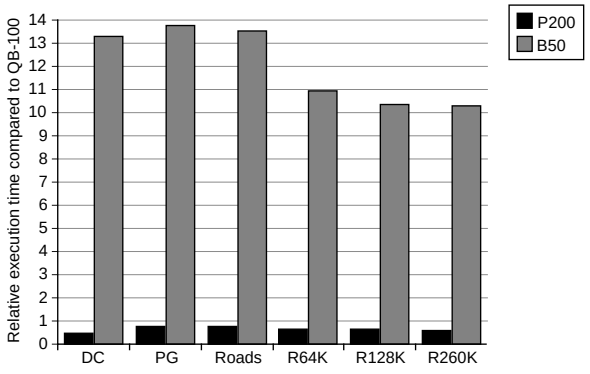


Fig. 23. Relative performance of two R-tree bulk-loading algorithm compared to "QB-100" ("P200" denotes the Hilbert-packed R-tree algorithm with a fan-out of 200, while "B50" denotes the buffer-tree approach with a fan-out of 50)

ing approach (see Sect. 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% (larger values caused more B-tree node splits). Note that in Fig. 22, 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 bulk-insertion algorithm may be preferable.

8.8 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 [37] with the space partitioning improvements of [21][15]; and 2) a very simplified version of the buffer-tree approach of [9, 15]. 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 [9, 15], 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 27 MB) was significantly less than the size of physical memory (64 MB). In order to obtain good space partitioning, we used the R*-tree [12] insertion rules, except that no reinsertions were performed as they are not supported by the buffer-tree approaches. Since 4 kB 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 [12], 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 23 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 ten to fourteen times 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 partitioning, which is a distinguishing feature of the PMR quadtree, is relatively low when using our bulk-loading algorithm.

[15] 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.

8.9 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 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. Nevertheless, our approach was nearly as fast as a quadtree bulk-loading method specialized for point data, indicating 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[16] 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 somewhat worse spatial filtering and thus potentially higher query cost.

We verified that our 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 partitioning 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., [9,15] with R*-tree insertion rules) can be much slower than our algorithm.

9 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.

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 reasonably "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 2–4, depending on the data distribution and other factors (see Sect. 8.9).

An important utility of bulk-loading methods is that they enable quickly building indexes on un-indexed data in preparation of performing complex operations such as joins. In order to test the utility of our bulk-loading technique for this purpose, we performed a small experiment with the spatial join example mentioned in Sect. 1: given a collection of line segments representing roads and another representing rivers, find

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

all locations where a road and a river intersect[17]. When an index existed for one data set but not the other, we observed speedups ranging from 60% to 75% by bulk-loading an index prior to running the query ranged rather than perform it with just one index. When neither data set had an index, building indexes for both sets and using them to answer the query was more than an order of magnitude faster than evaluating the query using a naive nested loop algorithm. Of course, in the latter situation, it may be faster to use a fast spatial join algorithm specially meant for non-indexed data sets (e.g., [10]) rather than bulk-loading both data sets. Nevertheless, the bulk-loading approach has the advantage that it also speeds up future queries involving the bulk-loaded data set, assuming that the produced index is retained.

Future work includes investigating whether our buffering strategies for bulk-loading may be used to speed up dynamic insertions and queries. In addition, we wish to investigate situations in which a query engine can exploit fast PMR quadtree index construction in order to speed spatial operations on intermediate query results (possibly from non-spatial subqueries), or for un-indexed spatial relations. This is particularly important for complex operations such as spatial joins.

References

1. Aasno T, Ranjan D, Roos T, Welzl E, Widmayer P (1997) Space-filling curves and their use in the design of geometric data structures. Theor Comput Sci 181(1):3–15
2. Abel DJ, Mark DM (1990) A comparative analysis of some two-dimensional orderings. Int J Geogr Inf Syst 4(1):21–31
3. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Comm ACM 31(9):1116–1127
4. Aggarwal C, Wolf J, Yu P, Epelman M (1997) The S-tree: an efficient index for multidimensional objects. In: Scholl M, Voisard A (eds) Advances in Spatial Databases – 5th International Symposium, SSD'97, Lecture Notes in Computer Science, vol. 1262. Springer, Berlin Heidelberg New York, 1997, pp. 350–373
5. Ang CH, Tan TC (1997) New linear node splitting algorithm for R-trees. In: Scholl M, Voisard A (eds) Advances in Spatial Databases – 5th International Symposium, SSD'97, Lecture Notes in Computer Science, vol. 1262. Springer, Berlin Heidelberg New York, 1997, pp. 339–349
6. Aref WG, Samet H (1990) An approach to information management in geographical applications. In: Proc. 4th International Symposium on Spatial Data Handling vol. 2, pp. 589–598, Zurich, Switzerland
7. Aref WG, Samet H (1991) Extending a DBMS with spatial operations. In: Günther O, Schek HJ (eds) Advances in Spatial Databases – 2nd Symposium, SSD'91, Lecture Notes in Computer Science, vol. 525. Springer, Berlin Heidelberg New York, 1991, pp. 299–318
8. Arge L (1996) Efficient external-memory data structures and applications. PhD thesis, Computer Science Department, University of Aarhus, Aarhus, Denmark. Also in: BRICS Dissertation Series, DS-96-3
9. Arge L, Hinrichs KH, Vahrenhold J, Vitter JS (1999) Efficient bulk operations on dynamic R-trees. In: Goodrich MT, McGeoch CC (eds) Proc. 1st Workshop on Algorithm Engineering and Experimentation (ALENEX'99), Baltimore, Md., USA, Lecture Notes in Computer Science, vol. 1619. Springer, Berlin Heidelberg New York, 1999, pp. 328–348
10. Arge L, Procopiuc O, Ramaswamy S (1998) Scalable sweeping-based spatial join. In: Gupta A, Shmueli O, Widom J (eds) Proc. 24th International Conference on Very Large Data Bases VLDB, pp. 570–581, New York
11. Becker B, Franciosa PG, Gschwind S, Ohler T, Thiemt G, Widmayer P (1992) Enclosing many boxes by an optimal pair of boxes. In: Finkel A, Jantzen M (eds) Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science STACS,ENS Cachan, France, Lecture Notes in Computer Science, vol. 577. Springer, Berlin Heidelberg New York, 1992, pp. 475–486
12. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. ACM SIGMOD Conference pp. 322–331, Atlantic City, N.J., USA
13. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Comm ACM 18(9):509–517
14. Berchtold S, Böhm C, Kriegel HP (1998) Improving the query performance of high-dimensional index structures by bulk-load operations. In: Advances in Database Technology – EDBT'98, Proc. 6th International Conference on Extending Database Technology pp. 216–230, Valencia, Spain
15. van den Bercken J, Seeger B, Widmayer P (1997) A generic approach to bulk loading multidimensional index structures. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proc. 23rd International Conference on Very Large Data Bases VLDB pp. 406–415, Athens, Greece
16. Brinkhoff T, Kriegel HP (1994) The impact of global clustering on spatial database systems. In: Proc. 20th International Conference on Very Large Data Bases VLDB Bocca J, Jarke M, Zaniolo C (eds) pp. 168–179, Santiago, Chile
17. Bureau of the Census (1989) Tiger/Line precensus files. Washington, D.C., USA
18. Chen L, Choubey R, Rundensteiner EA (1998) Bulk-insertions into R-trees using the Small-Tree-Large-Tree approach. In: Laurini R, Makki K, Pissinou N (eds) Proc. 6th International Symposium on Advances in Geographic Information Systems pp. 161–162, Washington, D.C., USA
19. Ciaccia P, Patella M (1998) Bulk loading the M-tree. In: Proc. 9th Australasian Database Conference (ADC'98) pp. 15–26, Perth, Australia
20. Ciaccia P, Patella M, Zezula P (1997) M-tree: An efficient access method for similarity search in metric spaces. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proc. 23rd International Conference on Very Large Data Bases VLDB) pp. 426–435, Athens, Greece
21. DeWitt DJ, Kabra N, Luo J, Patel JM, Yu JB (1994) Client-server paradise. In: Bocca J, Jarke M, Zaniolo C (eds) Proc. 20th International Conference on Very Large Data Bases VLDB) pp. 558–569, Santiago, Chile
22. Esperança C, Samet H (1997) Orthogonal polygons as bounding structures in filter-refine query processing strategies. In: Scholl M, Voisard A (eds) Advances in Spatial Databases – 5th International Symposium, SSD'97, Lecture Notes in Computer Science, vol. 1262. Springer, Berlin Heidelberg New York, 1997, pp. 197–220
23. Faloutsos C (1986) Multiattribute hashing using gray codes. In: Proc. ACM SIGMOD Conference pp. 227–238, Washington, D.C., USA
24. Freeston M (1987) The BANG file: a new kind of grid file. In: Proc. ACM SIGMOD Conference pp. 260–269, San Francisco, Calif., USA

[17] We used the data sets described in Sect. 8.7, "Roads" with 200,482 line segments and "Water" with 37,495 line segments.

25. García YJ, López MA, Leutenegger ST (1998) A greedy algorithm for bulk loading R-trees. In: Laurini R, Makki K, Pissinou N (eds) Proc. 6th International Symposium on Advances in Geographic Information Systems pp. 163–164, Washington, D.C., USA
26. García YJ, López MA, Leutenegger SST (1998) On optimal node splitting for R-trees. In: Gupta A, Shmueli O, Widom J (eds) Proc. 24th International Conference on Very Large Data Bases VLDB pp. 334–344, New York
27. Gargantini I (1982) An effective way to represent quadtrees. Comm ACM 25(12):905–910
28. Gavrila DM (1994) R-tree index optimization. In: Waugh TC, Healey RG (eds) Proc. Sixth International Symposium on Spatial Data Handling pp. 771–791, Edinburgh, Scotland, UK, International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information. Also in: University of Maryland Computer Science TR-3292
29. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proc. ACM SIGMOD Conference pp. 47–57, Boston, Mass., USA
30. Hjaltason GR, Samet H (2002) Data files for bulk-loading experiments. Available at: http://db.uwaterloo.ca/∼gisli/bldata.tgz
31. Hjaltason GR, Samet H, Sussmann Y (1997) Speeding up bulk-loading of quadtrees. In: Proc. 5th International ACM Workshop on Advances in GIS pp. 50–53, Las Vegas, Nev., USA
32. Hoel EG, Samet H (1995) Benchmarking spatial join operations with spatial output. In: Dayal U, Gray PMD, Nishio S (eds) Proc. 21st International Conference on Very Large Data Bases VLDB) pp. 606–618, Zurich, Switzerland
33. Huang S, Viswanathan V (1990) On the construction of weighted time-optimal B-trees. BIT 30(2):207–215
34. Iwerks G, Samet H (1999) The spatial spreadsheet. In: Huijsmans DP, Smeulders AWM (eds) Proc. Third International Conference on Visual Information Systems VISUAL99) pp. 317–324, Amsterdam, The Netherlands
35. Jagadish HV (1990) Linear clustering of objects with multiple attributes. In: Proc. ACM SIGMOD Conference pp. 332–342, Atlantic City, N.J., USA
36. Jagadish HV, Narayan PPS, Seshadri S, Kanneganti R (1997) Incremental organization for data recording and warehousing. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds)Proc. 23rd International Conference on Very Large Data Bases VLDB) pp. 16–25, Athens, Greece
37. Kamel I, Faloutsos C (1993) On packing R-trees. In: Proc. Second International Conference on Information and Knowledge Management CIKM) pp. 490–499, Washington, D.C., USA
38. Kamel I, Faloutsos C (1994) Hilbert R-tree: An improved R-tree using fractals. In: Proc. 20th International Conference on Very Large Data Bases VLDB) Bocca J, Jarke M, Zaniolo C (eds) pp. 500–509, Santiago, Chile
39. Kamel I, Khalil M, Kouramajian V (1996) Bulk insertion in dynamic R-trees. In: Kraak MJ, Molenaar M (eds) Proc. 7th International Symposium on Spatial Data Handling pp. 3B31–3B42, Delft, The Netherlands. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information
40. Klein TM, Parzygnat KJ, Tharp L (1991) Optimal B-tree packing. Inf Syst 16(2):239–243
41. Leutenegger ST, López MA, Edgington J (1997) STR: A simple and efficient algorithm for R-tree packing. In: Gray A, Larson PAA (eds) Proc. 13th IEEE International Conference on Data Engineering, pp. 497–506, Birmingham, UK
42. Leutenegger ST, Nicol DM (1997) Efficient bulk-loading of gridfiles. IEEE Trans Knowl Data Eng 9(3):410–420
43. Li J, Rotem D, Srivastava J (1993) Algorithms for loading parallel grid files. In: Proc. ACM SIGMOD Conference pp. 347–356, Washington, D.C., USA
44. Lindenbaum M, Samet H, Hjaltason GR (2000) A probabilistic analysis of trie-based sorting of large collections of line segments in spatial databases. Computer Science Department TR-3455.1, University of Maryland, College Park, MD., USA
45. Morton GM (1966) A computer oriented geodetic data base and a new technique in file sequencing. IBM, Ottawa, Canada
46. Nelson RC, Samet H (1987) A population analysis for hierarchical data structures. In: Proc. ACM SIGMOD Conference pp. 270–277, San Francisco, Calif., USA
47. O'Neil PE, Cheng E, Gawlick D, O'Neil EJ (1996) The log-structured merge-tree LSM-tree). Acta Inf 33(4):351–385
48. Oracle Corporation (1996) Advances in relational database technology for spatial data management. Oracle spatial data option technical white paper, Oracle Corporation
49. Orenstein JA, Merrett TH (1984) A class of data structures for associative searching. In: Proc. 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS) pp. 181–190, Waterloo, Ontario, Canada
50. Rosenberg AL, Snyder L (1981) Time- and space-optimality in B-trees. ACM Trans Database Syst 6(1):174–193
51. Roussopolous N, Kotidis Y, Roussopolous M (1997) Cubetree: Organization of and bulk incremental updates on the data cube. In: J Peckham (ed) Proc. ACM SIGMOD Conference pp. 89–111, Tucson, Ariz., USA
52. Roussopoulos N, Leifker D (1985) Direct spatial search on pictorial databases using packed R-trees. In: Proc. ACM SIGMOD Conference pp. 17–31, Austin, Tex., USA
53. Salzberg B (1988) File structures: an analytic approach. Prentice-Hall, Englewood Cliffs, N.J., USA
54. Samet H (1990) Applications of spatial data structures: computer graphics, image processing. Addison-Wesley, Reading, Mass., USA
55. Samet H (1990) The design and analysis of spatial data structures. Addison-Wesley, Reading, Mass., USA
56. Seeger B, Kriegel HP (1990) The buddy-tree: an efficient and robust access method for spatial data base systems. In: McLeod D, Sacks-Davis R, Schek HJ (eds) Proc. 16th International Conference on Very Large Databases VLDB pp. 590–601, Brisbane, Australia
57. Stonebraker M, Sellis T, Hanson E (1986) An analysis of rule indexing implementations in data base systems. In: Proc. 1st International Conference on Expert Database Systems pp. 353–364, Charleston, S.C., USA
58. Wang W, Yang J, Muntz R (1998) PK-tree: a spatial index structure for high dimensional point data. In: Tanaka K, Ghandeharizadeh S (eds) Proc. 5th International Conference on Foundations of Data Organization and Algorithms FODO) pp. 27–36, Kobe, Japan
59. White DA, Jain R (1996) Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, Calif., USA
60. Yang J, Wang W, Muntz R (1997) Yet another spatial indexing structure. Computer Science Department Technical Report 970040, University of California at Los Angeles (UCLA), Los Angeles, Calif., USA. Available at: http://www.cs.unc.edu/∼weiwang/paper/TR-97040ps
61. Yao AC (1978) On random 2-3 trees. Acta Inf 9(2):159–168