Efficient Bulk Operations on Dynamic R-trees

Lars Arge* Klaus H. Hinrichs† Jan Vahrenhold‡ Jeffrey Scott Vitter§

Abstract

In recent years, there has been an upsurge of interest in spatial databases. A major issue is how to efficiently manipulate massive amounts of spatial data stored on disk in multidimensional *spatial indexes* (data structures). Construction of spatial indexes (*bulk loading*) has been studied intensively in the database community. The continuous arrival of massive amounts of new data makes it important to efficiently update existing indexes (*bulk updating*).

In this paper, we present a simple, yet efficient technique for performing bulk update and query operations on multidimensional indexes. We present our technique in terms of the so-called R-tree and its variants, as they have emerged as practically efficient indexing methods for spatial data. Our method uses ideas from the *buffer tree* lazy buffering technique and fully utilizes the available internal memory and the page size of the operating system. We give a theoretical analysis of our technique, showing that it is efficient both in terms of I/O communication, disk storage, and internal computation time. We also present the results of an extensive set of experiments showing that in practice our approach performs better than the previously best known bulk update methods with respect to update time, and that it produces a better quality index in terms of query performance. One important novel feature of our technique is that in most cases it allows us to perform a batch of updates and queries simultaneously. To be able to do so is essential in environments where queries have to be answered even while the index is being updated and reorganized.

*Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708, USA. Supported in part by the U.S. Army Research Office through MURI grant DAAH04–96–1–0013 and by the National Science Foundation through ESS grant EIA–9870734 and CAREER grant EIA–9984099. Email: large@cs.duke.edu.

†Westfälische Wilhelms-Universität, Institut für Informatik, 48149 Münster, Germany. Email: khh@math.uni-muenster.de.

‡Westfälische Wilhelms-Universität, Institut für Informatik, 48149 Münster, Germany. Part of this work was done while a visiting scholar at Duke University. Email: jan@math.uni-muenster.de.

§Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708, USA. Supported in part by the U.S. Army Research Office through MURI grant DAAH04–96–1–0013 and by the National Science Foundation through grants CCR–9522047, EIA–9870734, and CCR–9877133. Email: jsv@cs.duke.edu.

1 Introduction

In recent years, there has been an upsurge of interest in spatial databases in the commercial and research database communities. Spatial databases are systems designed to store, manage, and manipulate spatial data like points, polylines, polygons, and surfaces. Geographic information systems (GIS) are a popular incarnation. Spatial database applications often involve massive data sets, such as for example EOS satellite data [12]. Thus the need for efficient handling of massive spatial data sets has become a major issue, and a large number of disk based multidimensional index structures (data structures) have been proposed in the database literature (see, e.g., [5, 14, 23, 33] for recent surveys). Typically, multidimensional index structures support insertions, deletions, and updates, as well as a number of proximity queries like window or nearest-neighbor queries. Recent research in the database community has focused on supporting *bulk operations*, in which a large number of operations are performed on the index at the same time. The increased interest in bulk operations is a result of the ever-increasing size of the manipulated spatial data sets and the fact that performing a large number of single operations one at a time is simply too inefficient to be of practical use. The most common bulk operation is to create an index for a given data set from scratch—often called *bulk loading* [13].

In this paper, we present a simple lazy buffering technique for performing bulk operations on multidimensional indexes and show that it is efficient in theory as well as in practice. We present our results in terms of the R-tree and its variants [9, 16, 17, 19, 28], which have emerged as especially practically efficient indexing methods for spatial data. However, our technique applies not only to R-trees but also to a large class of hierarchically structured multidimensional indexes, namely the so-called class of *grow-and-post trees* [22].

1.1 Model of computation and previous results on I/O-efficient algorithms

Since objects stored in a spatial database can be rather complex they are often approximated by simpler objects, and spatial indexes are then built on these approximations. The most commonly used approximation is the *minimal bounding box*: the smallest d-dimensional rectangle that includes the object. Thus an important indexing problem is to maintain a dynamically changing set of d-dimensional rectangles on disk such that, for example, all rectangles containing a query point can be found efficiently. For simplicity we restrict our attention to the two-dimensional case; the boxes are called *minimal bounding rectangles*. For convenience we refer to these rectangles as being input data; we assume that each bounding rectangle contains a pointer to the place on disk where the real data object is stored.

For our theoretical considerations we use the standard two-level I/O model [1] and define the following parameters:

$$\begin{aligned} N &= \#\text{ of rectangles,} \\ M &= \#\text{ of rectangles fitting in internal memory,} \\ B &= \#\text{ of rectangles per disk block,} \end{aligned}$$

where $N \gg M$ and $1 \leq B \leq M/2$. An *input/output operation* (or simply *I/O*) consists of reading a block of contiguous elements from disk into internal memory or writing a block from internal memory to disk. Computations can only be performed on rectangles in internal memory. We measure the efficiency of an algorithm by the number of I/Os it performs, the amount of disk space it uses (in units of disk blocks), and the internal memory computation time.[1] More sophisticated measures of disk performance involve analysis of seek and rotational latencies and caching issues [27]; however, the simpler standard model has proven quite useful in identifying first-order effects [32].

[1] For simplicity we concentrate on the two first measures in this paper. It can be shown that the asymptotic internal memory computation time of our new R-tree algorithms is the same as for the traditional algorithms.

I/O efficiency has always been a key issue in database design but has only recently become a central area of investigation in the algorithms community. Aggarwal and Vitter [1] developed matching upper and lower I/O bounds for a variety of fundamental problems such as sorting and permuting. For example, they showed that sorting N items in external memory requires $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. Subsequently, I/O-efficient algorithms have been developed for several problem domains, including computational geometry, graph theory, and string processing. The practical merits of the developed algorithms have been explored by a number of authors. Recent surveys can be found in [4, 5, 33]. Much of this work uses the *Transparent Parallel I/O Environment* (TPIE) [6, 31], a set of templated C++ functions and classes that allow for simple, efficient, and portable implementation of I/O algorithms.

1.2 Previous results on bulk operations on R-trees

The R-tree, originally proposed by Guttman [17], is a height-balanced multiway tree similar to a B-tree [7, 11]. The leaf nodes contain $\Theta(B)$ data rectangles each, while internal nodes contain $\Theta(B)$ entries of the form $(Ptr,\ R)$, where Ptr is a pointer to a child node and R is the minimal bounding rectangle covering all rectangles in the subtree rooted in that child. An example of an R-tree is depicted in Figure 1.

An R-tree occupies $O(N/B)$ disk blocks and has height $O(\log_B N)$; insertions can be performed in $O(\log_B N)$ I/Os. There is no unique R-tree for a given set of data rectangles and minimal bounding rectangles stored within an R-tree node can overlap. In order to query an R-tree to find all rectangles containing a given point p, all internal nodes whose minimal bounding rectangle contains p have to be visited. Intuitively, we thus want the minimal bounding rectangles stored in a node to overlap as little as possible. An insertion of a new rectangle can increase the overlap and several heuristics for choosing which leaf to insert a new rectangle into, as well as for splitting nodes during rebalancing, have been proposed [9, 16, 19, 28].

Bulk loading an R-tree with N rectangles using the naive method of repeated insertion takes $O(N\log_B N)$ I/Os, which has been recognized to be abysmally slow. Several bulk loading algorithms using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os have been proposed [10, 13, 18, 21, 26, 30]. These algorithms are more than a factor of B faster than the repeated insertion algorithm. Most of the proposed algorithms [13, 18, 21, 26] work in the same basic way; the input rectangles are sorted according to some global one-dimensional criterion (such as x-coordinate [26], the Hilbert value of the center of the rectangle [13, 18], or using an order obtained from a rectangular tiling of the space [21]) and placed in the leaves in that order. The rest of the index is then built recursively in a bottom-up, level-by-level manner. The algorithm developed in [30] also builds the index recursively bottom-up but utilizes a lazy buffering strategy. The method proposed in [10] works recursively in a top-down way by repeatedly trying to find a good partition of the data. This algorithm is designed for point

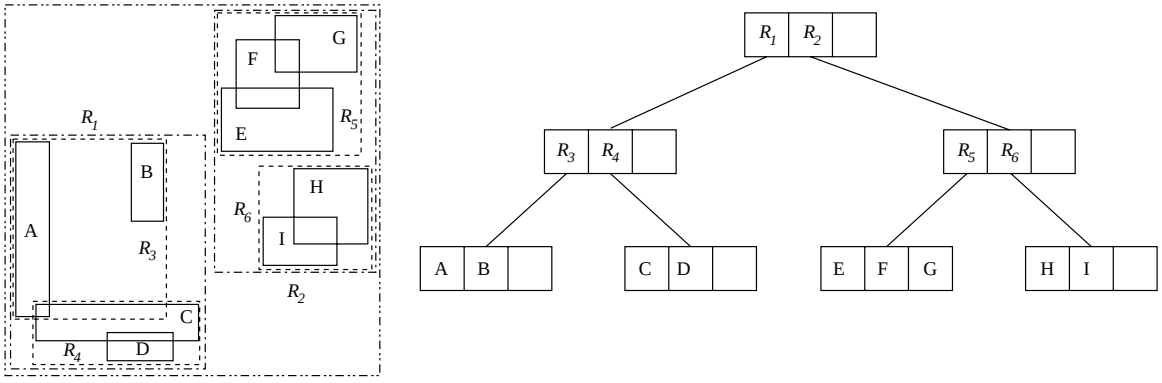


Figure 1: R-tree constructed on rectangles A, B, C, ..., I (Blocksize 3).

data instead of rectangle data but by transforming rectangles to points in higher-dimensional space it can still be used for rectangle data.

Even though a major motivation for designing bulk loading algorithms is the slowness of the repeated insertion algorithm, another and sometimes even more important motivation is the possibility of obtaining better space utilization and query performance.[2] Most of the bulk loading algorithms mentioned above are capable of obtaining almost 95% space utilization (meaning that only $\frac{1}{0.95}\lceil\frac{N}{B}\rceil$ disk blocks are used), while empirical results show that average utilization of the repeated insertion algorithm is between 50% and 70% [9]. However, empirical results also indicate that packing an R-tree too much can lead to poor query performance, especially when the data is not uniformly distributed [13]. Bulk loading algorithms typically produce R-trees with better query performance than the repeated insertion algorithm. But no one algorithm is best for all cases (all data distributions) [10, 21]: On mildly skewed low-dimensional data the algorithm in [21] outperforms the algorithm in [18], while the opposite is the case on highly skewed low-dimensional data [21]. Both algorithms perform poorly on higher-dimensional point data [10], where the algorithm developed in [10] achieves the best query performance. Interestingly, in terms of I/O the repeated insertion algorithm produces an index with a similar query performance as that of the algorithm in [10]. In terms of total running time repeated insertion performs worse, probably because the algorithm in [10] produces an index where blocks containing neighboring nodes in the tree are stored close to each other on disk. van den Bercken et al. [30] have independently developed a bulk loading method that uses a lazy buffer technique similar to the one we use in this paper. We describe their algorithm in more detail in Section 2.3. A common misperception (cf., for example, [10]) is that the algorithm produces an R-tree index identical to the index obtained using repeated insertion. In reality the two R-tree indexes can be quite different; van den Bercken et al. [30] report empirical results only on the performance of the construction of multiversion B-trees [8], where the order of elements in the leaves is unique and thus the buffer algorithm will always construct the same order as the repeated insertion algorithm. This equivalence does not hold for R-tree construction and the quality of an R-tree index produced using the method has not been studied.

All algorithms mentioned above are inherently "static" in the sense that they can only be used to bulk load an index with a given static data set. None of them efficiently supports bulk updates. To perform a batch of updates we would have to run the bulk loading algorithm on the combined datasets or perform the insertions one by one using the normal insertion algorithm. In many applications these solutions are not viable, and in fact, bulk updating is mentioned in [10] as an important open problem. The most successful attempt to process a batch of updates seems to be an algorithm by Kamel et al. [20]. In this algorithm, the rectangles to be inserted are first sorted according to their spatial proximity (Hilbert value of the center) and then packed into blocks of B rectangles. These blocks are then inserted one at a time using standard insertion algorithms. Intuitively, the algorithm should give an insertion speedup of B (as blocks of B rectangles are inserted together), but it is likely to increase overlap and thus produces a worse index in terms of query performance. Empirical results presented in [20] support this intuition. Bulk loading R-trees is also used in the context of the cubetree [25], a representation for the *data cube* [15], using a collection of packed point data R-trees [26]. Bulk updates to the data cube are handled by repeatedly bulk loading separate R-trees with the new items and merging a set of (original and new) R-trees together. This algorithm implies a method for updating R-trees, namely to sort-merge the data in the old tree with the data points corresponding to the updates and then to bulk load a new R-tree with this sorted sequence. The merging takes advantage of sequential blocked

[2] A simple approach for building an R-tree using only $O(N/B)$ I/Os is to form the leaves by grouping the input rectangles B at a time and then build the tree in a bottom-up level-by-level manner. However, if the grouping of rectangles is done in an arbitrary manner, the resulting R-tree will likely have extremely bad query performance, since the minimal bounding rectangles in the nodes of the index will have significant overlaps. Such R-tree algorithms are therefore of no interest. At a minimum, constructing good R-trees is at least as hard as sorting and therefore we refer to algorithms that use $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os as *optimal*.

access, however, the entire data set must be processed, including the original data. If the number of updates is relatively small, the amortized cost per update can therefore be high.

1.3 Our results

In this paper, we present a simple lazy buffering technique for performing bulk operations on multidimensional indexes. As mentioned, we describe the technique in terms of the R-tree family.

In the first part of the paper (Section 2), we present our technique and analyze its theoretical performance. Unlike previous methods our algorithm does not need to know in advance all the operations to be performed, something which is important in applications where updates and queries arrive continuously, e.g., in pipelined database operations. Furthermore, our method is general enough to handle a wide variety of bulk operations:

- Our algorithm can *bulk insert* N' rectangles into an R-tree containing N rectangles using $O(\frac{N'}{B}\log_{M/B}\frac{N+N'}{B}+\frac{N}{B})$ I/Os in the worst case.
- Using the bulk insertion algorithm an R-tree can be *bulk loaded* in the optimal number of I/O operations $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.
- Given N' queries that require $O(Q\log_B\frac{N}{B})$ I/Os (for some Q) using the normal (one by one) query algorithm, our algorithm answers the queries in $O(\frac{Q}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B})$ I/Os.
- A set of N' rectangles can be *bulk deleted* from an R-tree containing N rectangles using $O(\frac{N'}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B}+Q(N'))$ I/Os, where $Q(N')$ is the number of I/Os needed to locate the leaves containing the N' rectangles to be deleted.

In most cases, our algorithms represent an improvement of more than a factor of B over known methods, and our technique can even handle a batch of intermixed inserts, deletes, and queries. As discussed in [25], being able to do so is extremely important in many environments where queries have to be answered while the index is being updated.

In the second part of the paper (Section 3), we present the results of a detailed set of experiments on real-life data. The data sets we use are the standard benchmark data set for spatial indexes, namely, the TIGER/Line data [29]. Our experiments were designed to test our theoretical predictions and to compare the performance of our algorithms with previously known bulk update algorithms:

- To investigate the general effects of our technique we used it in conjunction with the standard R-tree heuristics to bulk load an R-tree. We compared its performance with that of the repeated insertion algorithm. Our experiments show that we obtain a huge speedup in construction time, and at the same time the query performance remains good. Even though bulk loading using our technique does not yield the same index as would be obtained using repeated insertion (contrary to popular belief [10, 30]), the quality of the index remains about the same.
- As discussed, special-purpose bulk loading algorithms often produce significantly better indexes than the one obtained using repeated insertion, especially in terms of space utilization. We are able to capitalize on a certain laziness in our algorithm, via a simple modification, and achieve dramatically improved space utilization. The modification uses a heuristic along the lines of [13, 18]. Our bulk loading experiments with the modified algorithm show that we can obtain around 90% space utilization while maintaining or even improving the good query performance. As an added benefit the modification also improves construction time by up to 30%.

- Finally, in order to investigate the practical efficiency of our technique when performing more general bulk operations, we performed experiments with bulk insertion of a large set of rectangles in an already existing large R-tree. We compared the performance of our algorithm with that of the previously best known bulk update algorithm [20]. Our algorithm performs better than the algorithm in [20], in terms of both the number of I/Os used to do the insertions and the query performance of the resulting R-tree index. We also performed bulk query experiments that resulted in speed-ups similar to the ones obtained in our bulk loading experiments.

One especially nice feature of our algorithms is that—from a high level point of view—the set of bulk operations are performed precisely as in the standard on-line algorithms. For example, our bulk insertion algorithm is conceptually identical to the repeated insertion algorithm (except of course that the insertions are done lazily). From an implementation point of view, our algorithms admit a nice modular design because they access the underlying index (in our case, the R-tree) only via standard routing and balancing routines. Having implemented our algorithms we can thus combine them with most existing indexes very easily.

2 Performing Bulk Operations on R-trees using Buffers

In this section, we present our technique for performing bulk operations on R-trees and analyze it theoretically. In Section 2.1, we review the standard R-tree insertion and query algorithms and present the general idea in our buffer technique. In Section 2.2, we discuss the details in how a R-tree index can be bulk loaded, and in Section 2.3 how a bulk of insertions or queries can be performed on an existing index using the technique. In Section 2.4, we discuss how to perform a bulk of deletions. Final remarks (including a discussion of how our method compares with a previously proposed buffer technique [30]) are given in Section 2.5.

2.1 R-tree basics and sketch of our technique

Before presenting the main idea in our buffering technique, we first review the algorithms for inserting a new rectangle into an R-tree and for querying an R-tree with a rectangle. (Other queries can be handled with similar algorithms.) Deletes are discussed in Section 2.4.

As mentioned, the R-tree is a height-balanced tree similar to a B-tree; all leaf nodes are on the same level of the tree and a leaf contains $\Theta(B)$ rectangles. Each internal node v (except maybe for the root) has $\Theta(B)$ children. For each of its children, v contains a rectangle that covers all the rectangles in the child. We assume that each leaf and each internal node fits in one disk block. An R-tree has height $O(\log_B \frac{N}{B})$ and occupies $O(\frac{N}{B})$ blocks. Guttman [17] introduced the R-tree and several researchers have subsequently proposed different update heuristics designed to minimize the overlap of rectangles stored in a node [9, 16, 19, 28]. All variants of R-tree insertion algorithms (heuristics) [9, 16, 17, 19, 28] conceptually work in the same way (similar to B-tree algorithms) and utilize two basic functions:

- *Route*(r, v), which given an R-tree node v and a rectangle r to be inserted, returns the best (according to some heuristic) subtree v_s to insert r into. If necessary, the function also updates (extends) the rectangle stored in v that corresponds to v_s.
- *Split*(v), which given a node v, splits v into two new nodes v' and v''. The function also updates the entry for v in *parent*(v) to correspond to v', as well as insert a new entry corresponding to v''. (If v is the root, a new root with two children is created.)

Queries are handled in the same way in all R-tree variants using the following basic function:

Insert(r, v)

1. `WHILE` v `is not a leaf DO`
   $v := Route(r, v)$
2. `Add` r `to the rectangles of` v
3. `WHILE too many rectangles in` v `DO`
   $f := parent(v)$; $Split(v)$; $v := f$

Query(r, v)

1. `IF` v `is not a leaf THEN`
   $V_s := Search(r, v)$
   `Recursively call` $Query(r, v')$
   `for all` $v' \in V_s$
2. `IF` v `is a leaf THEN`
   `Report all rectangles in` v
   `that intersect` r

Figure 2: Abstract algorithms for inserting and querying in an R-tree rooted at node v.

- *Search*(r, v), which given a rectangle r and a node v, returns a set of subtrees V_s whose associated rectangles in v intersect r.

The abstract algorithms for inserting a new rectangle into an R-tree and for querying an R-tree with a rectangle are given in Figure 2. When performing a query many subtrees may need to be visited. Thus it is not possible to give a better than linear I/O bound on the worst-case complexity of a query. An insertion can be performed in $O(\log_B \frac{N}{B})$ I/Os since only nodes on a single root-to-leaf path are visited by the (routing as well as the rebalancing) algorithm.

Our technique for efficiently performing bulk operations on R-trees is a variant of the general *buffer tree technique* introduced by Arge [2, 3]. Here we modify the general technique in a novel way, since a straightforward application of the technique would result in an R-tree with an (impractically large) fan-out of m [30]. The main idea is the following: We attach *buffers* to all R-tree nodes on every $\lfloor \log_B \frac{M}{4B} \rfloor$th level of the tree. More precisely, we define the leaves to be on level 0 and assign buffers of size $\frac{M}{2B}$ blocks to nodes on level $i \cdot \lfloor \log_B \frac{M}{4B} \rfloor$, for $i = 1, 2, \ldots$. (See Figure 3.) We call a node with an associated buffer a *buffer node*. Operations on the structure are now done in a "lazy" manner. For example, let us assume that we are performing a batch of insertions. In order to insert a rectangle r, we do not immediately use $Route(r, v)$ to search down the tree to find a leaf to insert r into. Instead, we wait until a B rectangles to be inserted have been collected and

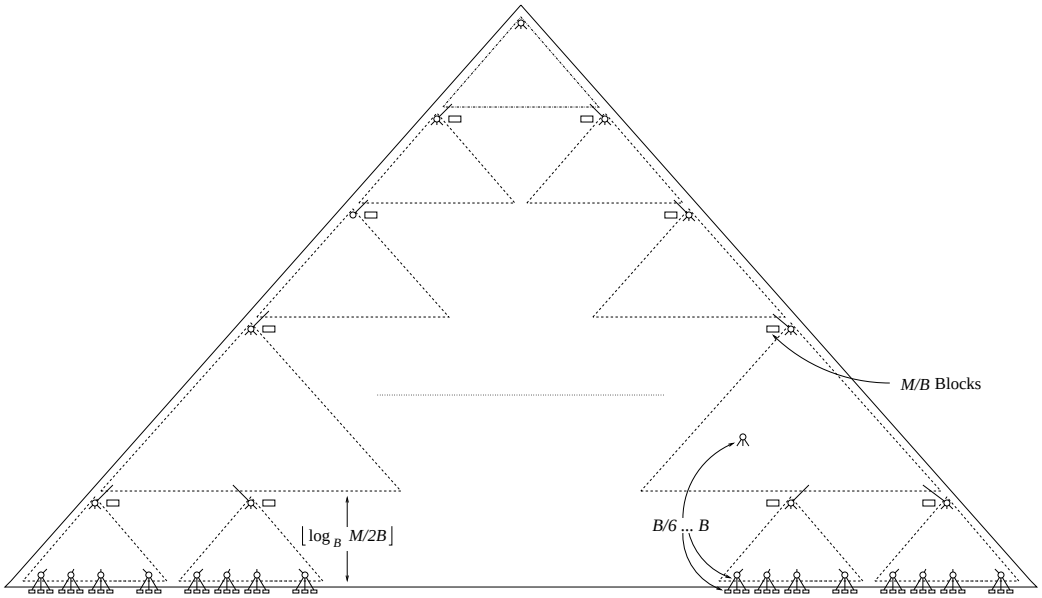


Figure 3: Buffered R-tree: Normal R-tree with buffers on every $\lfloor \log_B \frac{M}{4B} \rfloor$th level

On other than level $\lfloor \log_B \frac{M}{4B} \rfloor$:

`FOR the first` $M/4$ `rectangles` r `in` v`'s buffer DO`

1. $n := v$
2. `WHILE` n `does not have a buffer DO`
   $n := \mathit{Route}(r, n)$
3. `Insert` r `in the buffer of` n

On level $\lfloor \log_B \frac{M}{4B} \rfloor$:

`FOR all rectangles` r `in` v`'s buffer DO`

1. $n := v$
2. `WHILE` n `is not a leaf DO`
   $n := \mathit{Route}(r, n)$
3. `Add` r `to the rectangles in` n
4. `WHILE too many rectangles in` n `DO`
   $f := \mathit{parent}(n)$; $\mathit{Split}(n)$, $n := f$

Figure 4: Sketch of main idea in buffer emptying process on node v depending on the level of v.

then we store them in a block in the buffer of the root (which is stored on disk). When a buffer "runs full" (contains $M/4$ or more rectangles) we perform what we call a *buffer emptying process*: For each rectangle r in the buffer, we repeatedly use $\mathit{Route}(r, v)$ to route r down to the next buffer node v'. Then we insert r into the buffer of v', which in turn is emptied when it eventually runs full. When a rectangle reaches a leaf it is inserted there and if necessary the index is restructured using *Split*. In order to avoid "cascading" effects we only route the first $M/4$ rectangles from the buffer in a buffer emptying process. Since all buffers these rectangles are routed to are non-full (i.e., they contain less than $M/4$ rectangles), this means that no buffer overflow ($> \frac{M}{2}$ rectangles) can occur even if all rectangles are routed to the same buffer. The abstract buffer emptying process is sketched in Figure 4.

Because of the lazy insertion, some insertions do not cost any I/Os at all while others may be very expensive (by causing many buffer emptying processes). However, the introduction of buffers allows us to take advantage of the big block and internal memory sizes and perform the whole batch of insertions with fewer I/Os than we would use if we performed the insertions in the normal way. The key to this improvement is that when we empty a buffer we route many rectangles through a relatively small set of R-tree nodes to the next level of buffer nodes. In fact, the set of nodes is small enough to fit in half of the internal memory; the number of different buffer nodes the rectangles can be routed to is bounded by $B^{\lfloor \log_B \frac{M}{4B} \rfloor} \leq \frac{M}{4B}$, since the fan-out of the nodes is bounded by B. Thus the maximal number of nodes that needs to be loaded is bounded by $2 \cdot \frac{M}{4B} = \frac{M}{2B}$. This means that we can route the rectangles from the buffer *in main memory*: Before we perform the buffer emptying process sketched in Figure 4, we simply load all the relevant nodes, as well as the first $\frac{M}{4}$ rectangles from the buffer, into main memory using $O(\frac{M}{B})$ I/Os. Then we can perform the routing without using any I/Os at all, before using $O(\frac{M}{B})$ I/Os to write the R-tree nodes back to disk and $O(\frac{M}{B})$ I/Os to write the rectangles to the new buffers. (Since the number of buffers we write rectangles to is $< \frac{M}{4B}$, we only use $O(\frac{M}{B})$ I/Os to write non-full blocks.) In total we use $O(\frac{M}{B})$ I/Os to push $\frac{M}{4}$ rectangles $\lfloor \log_B \frac{M}{4B} \rfloor$ levels down, which amounts to $O(\frac{1}{B})$ I/Os per rectangle. To route all the way down the $O(\log_B \frac{N}{B})$ levels of the tree, we thus use $O((1/(B \cdot \lfloor \log_B \frac{M}{4B} \rfloor)) \cdot \log_B \frac{N}{B}) = O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os per rectangle. In contrast, the normal insertion algorithm uses $O(\log_B \frac{N}{B})$ I/Os per rectangle.

The above is just a sketch of the main idea of the buffer technique. There are still many issues to consider. In the next three subsections we discuss the details of how the buffer technique can be used to bulk load an R-tree, as well as to perform more general bulk operations.

2.2 Bulk loading an R-tree

Our bulk loading algorithm is basically the standard repeated insertion algorithm, where we use buffers as described in the previous subsection. A number of issues still have to be resolved in

order to make the algorithm I/O-efficient. For example, since a buffer emptying process can trigger other such processes, we use an external memory stack to hold all buffer nodes with full buffers (i.e., buffers containing more than $M/4$ rectangles). We push a reference to a node on this stack as soon as its buffer runs full and after performing a buffer-emptying on the root we repeatedly pop a reference from the stack and empty the relevant buffer. Special care needs to be taken when the root is not a buffer node, that is, when it is not on level $j \cdot \lfloor \log_B \frac{M}{4B} \rfloor$, for some j. In such a situation we simply store the top portion of the tree without buffers in internal memory instead of on the disk. Note that since an R-tree—being (like a B-tree) a *grow-and-post tree* [22]—only grows (shrinks) at the top, a node stays at the level of the tree it is on when it is created. This means that if a node is a buffer node when it is created it stays a buffer node throughout its lifetime.

We also need to fill in the details of how restructuring is performed, that is, how precisely the buffer of a node v on level $\lfloor \log_B \frac{M}{4B} \rfloor$ is emptied. The index is restructured using the *Split* function and we need to be a little careful when splitting buffer nodes. The detailed algorithm for emptying the buffer of a node on level $\lfloor \log_B \frac{M}{4B} \rfloor$ is given in Figure 5: We first load the R-tree rooted at v (including the leaves containing the data rectangles) into internal memory. Then all rectangles in the buffer are loaded (blockwise) and each of them is routed (using *route*) to a leaf and inserted. If an insertion causes a leaf l to overflow, *Split* is used to split l as well as all the relevant nodes on the path from l to v (these nodes are all in internal memory). If the split propagates all the way up to v we need to propagate it further up the part of the tree that is not in internal memory. In order to do so, we stop the buffer emptying process; we first write the trees rooted in the two (buffer) nodes v' and v'', produced by splitting v, back to disk. In order to distribute the remaining rectangles in the buffer of v, we then load all of them into internal memory. For each rectangle r, we use *Route* to decide which buffer to insert r into and finally we write each rectangle back to the relevant buffer on disk. It is easy to realize that we use $O(\frac{M}{B})$ I/Os on the above process, regardless of whether we stop the emptying process or not. Finally, the restructuring is propagated recursively up the index using *Split*, with the minor modification of the normal R-tree restructuring algorithm that rectangles in the buffer of a buffer node being split are distributed using *Route* as above.

Lemma 1 *A set of N rectangles can be inserted into an initially empty buffered R-tree using $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations.*

Proof: By the argument in Section 2.1 we use $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os per rectangle, that is, $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os in total, *not* counting I/Os used on emptying buffers on level $\lfloor \log_B \frac{M}{4B} \rfloor$, which in turn may result in splitting of R-tree nodes (restructuring). When emptying a buffer on level $\lfloor \log_B \frac{M}{4B} \rfloor$ we either push $M/2$ rectangles down to the leaves and the argument used in Section 2.1 applies, or we suspend the emptying process in order to rebalance the tree. In the latter case we may spend a constant number of I/Os to split R-tree nodes on each of the $O(\log_B \frac{N}{B})$ levels and $O(\frac{M}{B})$ I/Os to distribute rectangles on each of the $O((\log_B \frac{N}{B})/\log_B \frac{M}{4B}) = O(\log_{M/B} \frac{N}{B})$ levels with buffers, including the $O(\frac{M}{B})$ I/Os we used on the suspended buffer emptying. However, we only spend these I/Os when new nodes are created. During the insertion of all N rectangles, a total of $O(\frac{N}{B}/B)$ R-tree nodes and a total of $O(\frac{N}{B}/\frac{M}{B}) = O(\frac{N}{M})$ buffer nodes are created. Thus the overall restructuring cost adds up to at most $O(\frac{N}{B})$ I/Os. □

The only remaining issue is that after all the insertions have been performed it is very likely that we still have many nonempty buffers that need to be emptied in order to obtain the final R-tree. To do so we simply perform a buffer-emptying process on all buffer nodes in a breadth first manner, starting with the root.

Lemma 2 *All buffers in a buffered R-tree on N rectangles can be emptied in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations.*

- Load R-tree (including leaves) rooted in v into internal memory.
- WHILE (buffer of v contains rectangle r) AND (v is not split) DO
  1. $l := v$
  2. WHILE l is not leaf DO
     $l := Route(r, l)$
  3. Insert r in l
  4. WHILE (too many rectangles in l) AND (v is not split) DO
     $f := parent(l)$; $Split(l)$; $l := f$
- IF v did not split THEN write R-tree rooted in v back to disk.
- IF v split into v' and v'' THEN
  1. Write R-trees rooted in v' and v'' back to disk.
  2. Load remaining rectangles in buffer of v into internal memory.
  3. Use *Route* to compute which of the two buffers to insert each rectangle in.
  4. Write rectangles back to relevant buffers.
  5. $v := parent(v)$
  6. WHILE too many rectangles in v DO
     $f := parent(v)$; $Split(v)$
     IF v is a buffer node THEN
       (a) Load rectangles in buffer of v into internal memory.
       (b) Use *Route* to compute which buffer to insert each rectangle in.
       (c) Write rectangles back to buffers.
     $v := f$

Figure 5: Buffer emptying process on buffer nodes v on level $\lfloor \log_B \frac{M}{4B} \rfloor$.

Proof: The cost of emptying full buffers can be accounted for by the same argument as the one used in the proof of Lemma 1, and thus $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations are needed. The number of I/Os used on emptying non-full buffers is bounded by $O(\frac{M}{B})$ times the number of buffer nodes. As there are $O(\frac{N}{B}/\frac{M}{B})$ buffer nodes in a tree on N rectangles, the bound follows. □

The above two lemmas immediately imply the following.

Theorem 1 *An R-tree can be bulk loaded with N rectangles in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations.*

2.3 Performing bulk insertions and queries

After having discussed our bulk loading algorithm, it is easy to describe how bulk inserts can be performed efficiently on an already existing R-tree: We simply attach buffers to the tree, insert the rectangles lazily one by one, and perform a final emptying of all buffers. Using the same arguments as in the proofs of Theorem 1, we obtain the following.

Theorem 2 *A set of N' rectangles can be bulk inserted into an existing R-tree containing N rectangles in $O(\frac{N'}{B} \log_{M/B} \frac{N+N'}{B} + \frac{N}{B})$ I/O operations.*

In order to answer a large set of queries on an existing R-tree (such a situation often arises when performing a so-called spatial join [24]), we simply attach buffers to the R-tree and perform the queries in a lazy manner in the same way as when we perform insertions: To perform one query, we insert the query rectangle in the buffer of the root. When a buffer is emptied and a query needs

to be recursively performed in several subtrees we simply route a copy of the query to each of the relevant buffers. When a query rectangle reaches a leaf the relevant data rectangles are reported.

Theorem 3 *A set of N' queries can be performed on an existing R-tree containing N rectangles in $O(\frac{Q}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B})$ I/O operations, where $Q\log_B N$ is the number of nodes in the tree the normal R-tree query algorithm would visit.*

2.4 Performing bulk deletions

Conceptually, bulk deletions can be handled as insertions and queries: To delete rectangle r, we first perform a query for r using buffers lazily as before and when r (or rather, one of the several copies of r generated during the querying) reaches the relevant leaf the deletion is performed. The deletion of a rectangle may result in a leaf containing less than $B/6$ rectangles (normally called "node underflow") and requires the need for restructuring of the index. Restructuring is done by merging nodes and can propagate up the tree similar to the way splits can propagate. This corresponds to the way deletions are handled in B-trees. Many R-tree variants instead delete a node when it underflows and reinsert its children into the tree (often referred to as "forced reinsertions"). The idea behind this method is to try to obtain a better index by forcing a global reorganization of the structure instead of the local reorganization a node merge constitutes. Because of the laziness in our algorithms, we are not able to support forced reinsertion.

A couple of issues make deletes slightly more complicated to handle than insertions. First, in contrast to insertions where bounding rectangles in the internal nodes are adjusted while rectangles are pushed down the tree, such adjustments for deletions can only be done after a rectangle to be deleted has been located. Thus, after emptying the buffer of a node on level $\lfloor\log_B\frac{M}{4B}\rfloor$, we might need to travel all the way up the tree while adjusting bounding rectangles in the internal nodes. Second, recall that when splitting a buffer node in the insertion algorithm we redistributed the rectangles in the buffer between the two newly generated nodes. This cannot result in a full buffer. However, when merging buffer nodes the buffer of the resulting node might need to be emptied. In order to insure that the size of a buffer does not grow in an uncontrolled way, and that different restructuring operations do not interfere with each other, we therefore change the order in which we empty full buffers; we do not empty buffers of nodes on level $\lfloor\log_B\frac{M}{4B}\rfloor$ (possibly leading to the need for restructuring) before a buffer emptying process has been performed on all internal buffer nodes with full buffers. Similarly, emptying the buffer of a node on level $\lfloor\log_B\frac{M}{4B}\rfloor$ may lead to rebalancing and the creation of new internal nodes with full buffers. These buffers are emptied before another rebalancing operation is initiated. These modifications mean that the buffer of a node on level $\lfloor\log_B\frac{M}{4B}\rfloor$ can temporarily grow bigger than $\frac{M}{2}$ rectangles.[3]

The detailed algorithm for emptying a node v on level $\lfloor\log_B\frac{M}{4B}\rfloor$ is given in Figure 6: First the buffer and the rectangles in the subtree rooted in v are loaded into internal memory and the relevant rectangles are deleted (Step 1). The resulting set of rectangles is then inserted into a subtree rooted in one of the siblings of v; the best sibling v' to insert the rectangles into are determined using the *Route* function and then the rectangles are inserted while the tree is restructured using *Split* (Step 2–4). If v' splits the process is stopped and the remaining rectangles, as well as the rectangles in the buffer of v', are distributed between the two new buffers as normally when splitting a buffer node (Step 5). If this results in a buffer running full it is emptied as usual. After the deletion of v, $parent(v)$ may contain too few rectangles and the tree may need to be restructured. (This cannot be the case if v' was split.) As mentioned, this case is handled by merging nodes and may propagate up the tree (Step 6); we repeatedly use *Route* to find the best sibling to merge with and merge the

[3]The leaves below a node on level $\lfloor\log_B\frac{M}{4B}\rfloor$ can contain at most $\frac{M}{4}$ rectangles and thus only that many of the delete rectangles in the buffer can be "relevant". Thus, if the buffer grows to the maximal size $\frac{M}{2}$ we can remove at least $\frac{M}{4}$ of the rectangles in it, simply by removing rectangles that are not present in the leaves. This can easily be performed in $O(\frac{M}{B})$ I/Os, that is, in $O(1/B)$ I/Os per removed rectangle.

1. (* perform updates *)
   - Load the set of rectangles R in leaves of subtree rooted in v into internal memory and delete rest of subtree.
   - FOR all rectangles r in v's buffer DO
     Delete r from R (if $r \in R$)
2. (* find subtree v' to merge with *)
   - Compute MBR r for R.
   - Let $v' = Route(r, parent(v))$.
   - Load subtree rooted in v' into internal memory.
3. WHILE (R not empty) and (v' is not split) DO (* insert R in tree rooted in v' *)
   - Delete rectangle r from R.
   - $l := v'$
   - WHILE l is not a leaf DO
     $l = Route(r, l)$
   - Insert r in l
   - WHILE too many rectangles in l DO
     $f := parent(l)$; $Split(l)$; $l := f$
4. Write R-tree rooted in v' back to disk.
5. IF v' was split into v' and v'' THEN (* redistribute buffer *)
   - Write R-tree rooted in v'' back to disk.
   - Load set of rectangles R' in buffer of v' into internal memory.
   - Use *Search* to compute buffer to insert each rectangle from R' and R in.
   - Write rectangles back to relevant buffers.
   - IF buffer of any node v''' full THEN empty buffer of v'''.
6. WHILE too few rectangles in $parent(v')$ DO (* rebalance by merging *)
   - $v' = parent(v')$.
   - Let $v'' = Route(MBR(v'), parent(v'))$.
   - Merge v' and v'' into new node v'.
   - IF v' is a buffer node THEN
     (a) Insert rectangles from buffer of v'' into buffer of v'.
     (b) IF buffer of v' full THEN empty buffer of v'.
   - IF too many rectangles in v' THEN
     $split(v')$
     IF v' is a buffer node THEN
       (a) Load rectangles in buffer of v' into main memory
       (b) Use *Search* to compute which buffers to insert each rectangle into.
       (c) Write rectangles back to the two buffers.
7. WHILE v' is not the root DO (* Update routing rectangles *)
   - $f := parent(v')$
   - Adjust rectangle in f according to change in v'
   - $v' := f$

Figure 6: Buffer emptying process on buffer node v on level $\lfloor \log_B \frac{M}{4B} \rfloor$

two nodes, remembering to merge the buffers if the nodes are buffer nodes. If this produces a full buffer it is emptied. We do not empty other leaf buffers before all relevant internal nodes have had their buffers emptied and the current restructuring operation is finished. Thus we are sure that the

buffer emptying cannot result in another restructuring operation on the same node. If the merge results in a node with too many rectangles, we split the node, and if it is a buffer node, we also redistribute the rectangles in the buffer. In the latter case, the restructuring is finished. Finally, if the restructuring stops before reaching the root, we traverse the path to the root and update rectangles (Step 7) as discussed above.

Lemma 3 *A buffer emptying process on a node on level* $\lfloor \log_B \frac{M}{4B} \rfloor$ *(including I/Os used on rebalancing but not counting I/Os used on emptying full buffers) can be performed in* $O(\log_B \frac{N}{B} + \frac{M}{B}(1+X)$ *I/O operations, where* X *is the number of removed buffer nodes.*

Proof: Steps 1–5 can be performed in $O(\frac{M}{B})$ I/Os. In Steps 6 and 7 $O(1)$ I/Os are spent on merging or updating each of the $O(\log_B \frac{N}{B})$ nodes on a path to the root and $O(\frac{M}{B})$ I/Os extra I/Os are spent every time two buffer nodes are merged. □

Using this lemma we can then prove the following.

Lemma 4 *A set of* N' *rectangles can be bulk deleted from an R-tree containing* N *rectangles using* $O(\frac{N'}{B} \log_{M/B} \frac{N}{B} + \frac{N}{B} + Q(N'))$ *I/O operations, where* $Q(N')$ *is the number of I/O operations needed to locate the rectangles in the tree using batched query operations.*

Proof: All buffer emptying processes performed on other than level $\lfloor \log_B \frac{M}{4B} \rfloor$ buffer nodes during insertion of the deletes into the root buffer, as well as during the final emptying of all buffers, can be accounted for as in the proof of Theorem 3.

According to Lemma 3, we spend $O(\log_B \frac{N}{B} + \frac{M}{B})$ I/Os, plus $O(\frac{M}{B})$ I/Os per deleted buffer node, when emptying a buffer node on level $\lfloor \log_B \frac{M}{4B} \rfloor$. The number of buffer nodes is bounded by $O(\frac{N}{B}/\frac{M}{B})$ by the start of the algorithm so the last term contributes a total of $O(\frac{N}{B})$. The rest of the I/O cost can be amortized over the M rectangles in the buffer, so that each rectangle is charged $O(\frac{1}{M}(\log_B \frac{N}{B} + \frac{M}{B}) = O((\log_B \frac{N}{B})/B\frac{M}{B} + \frac{1}{B}) = O((\log_B \frac{N}{B})/B \log_B \frac{M}{B}) = O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os and the lemma follows. □

The only nonstandard R-tree operation used by our delete algorithm (and the only new operation compared with those used by the insertion algorithm) is the merging of two nodes. This operation can easily be implemented using an operation for inserting a whole subtree at a given level (and in a given node) of the tree. This operation is one of the standard R-tree operations used in forced reinsertion. Thus, as mentioned in the introduction, our algorithm possesses the nice property that it only accesses the index through the standard routines.

2.5 Further remarks

In the previous sections, we have discussed how to perform insertions, deletions, and queries using buffers. Our technique can easily be modified such that a batch of *intermixed* updates and queries can be performed efficiently (even if they are not all present at the beginning of the algorithm, but arrive in an on-line manner). Being able to do so is extremely important in many on-line environments, where queries have to be answered while the index is being updated [25]. Buffers are attached and the operations are performed by inserting them block-by-block into the buffer of the root. Buffer emptying is basically performed as discussed in the previous sections. Only the buffer emptying process for nodes on level $\lfloor \log_B \frac{M}{4B} \rfloor$ has to be modified slightly. Since the modifications are similar to the ones used in the basic buffer tree we refer the interested reader to [2] for details.

As mentioned in the introduction, there exists a variety of approaches for bulk loading multi-dimensional indexes. Most of these algorithms are optimized for a specific application area, e.g., geographic information systems, VLSI design, or multimedia databases, where the data to be processed has specific distributions and/or dimensionality. The goal of the present work was to design

a general technique for bulk update operations on multidimensional indexes, and we do not regard our proposed technique as a competitor to specialized bulk loading algorithms. Instead, we can build upon these algorithms in the following way: when bulk loading a multidimensional R-tree from scratch, we can choose the best bulk loading algorithm (heuristic) for the problem at hand, build the index using this technique, and finally (re-)attach buffers in order to perform subsequent bulk operations efficiently. Apart from our new technique, the only known methods for performing bulk operations other than bulk loading are the algorithms by Kamel *et al.* [20] and the method that can be derived from the algorithm by Roussopoulos *et al.* [25]. As discussed in the introduction, the amortized update cost of the latter method can be very high. The algorithm in [20] on the other hand has a good amortized performance and we therefore include a practical comparison of the bulk insertion performance of this algorithm and our proposed technique in Section 3.4. Table 1 contains a comparison of the capabilities of the different bulk techniques.

Technique	Bulk Operation				
	Loading	Insertions	Deletions	Queries	Intermixed
Berchtold *et al.* [10][4]	√	—	—	—	—
Kamel and Faloutsos [19]	√	—	—	—	—
Kamel *et al.* [20]	√	√	—	—	—
Leutenegger *et al.* [21]	√	—	—	—	—
van den Bercken *et al.* [30]	√	—	—	—	—
Roussopoulos *et al.* [25]	√	√[5]	√[5]	—	√[5]
(this paper)	√	√	√	√	√

Table 1: Comparison of different bulk processing techniques.

As mentioned, a buffering method similar to our approach has previously been presented by van den Bercken, Seeger, and Widmayer [30]. However, while our approach supports all kinds of bulk operations, the approach in [30] only supports bulk loading. To bulk load an R-tree, the algorithm in [30] first constructs an R-tree with fanout $\frac{M}{B}$ by attaching buffers to *all* nodes and performing insertions in a way similar to the one used in our algorithm. The leaves in this R-tree are then used as leaves in the R-tree with fanout B that will eventually be constructed. The rest of this tree is produced in a bottom-up manner by successively applying the buffer algorithm to the set of rectangles obtained when replacing the rectangles in each node on the just constructed level with their minimal bounding rectangle. The number of I/Os used on performing the bulk loading is dominated by the construction of the leaf level, which asymptotically is the same as the algorithm we develop. In practice, however, the additional passes over the data, even though they involve data of geometrically decreasing size, represent a significant percentage of the total time. Bulk loading using our method corresponds to the leaf level phase of the algorithm in [30] and will therefore be more efficient in practice Furthermore, the bottom-up construction is inherently off-line, since all data needs to be known by the start of the algorithm, and the algorithm is therefore unsuitable for bulk operations other than bulk loading. Another advantage of our bulk loading technique being on-line can be illustrated by considering *pipelining*; Consider four relations A, B, C, and D, each of which is indexed by an R-tree. In order to process the multi-way join $A \bowtie B \bowtie C \bowtie D$, one can perform two two-way joins, e.g., $A \bowtie B$ and $C \bowtie D$, and then join the resulting data sets. If the join operator requires the input relations to be indexed, two indexes have to be constructed for the intermediate results of $A \bowtie B$ and $C \bowtie D$. Using our technique this construction can be started while the first two joins are processed, whereas all off-line algorithms need to wait for the first two

[4] Does not work directly with rectangular data, see Section 1.2

[5] Method can be derived from the original algorithm [25] but involves bulk loading a new data structure.

joins to be computed. In summary, both the method proposed in [30] and our method utilize a lazy buffering approach, but our technique is more efficient, on-line, and it allows for much more general bulk operations.

3 Empirical Results

In this section, we discuss our implementation of the algorithms presented in the last section and give empirical evidence for their efficiency when compared to existing methods. In Section 3.1, we describe our implementation and the experimental setup. Section 3.2 is dedicated to an empirical analysis of the effects of buffering, and in Section 3.3 we discuss how to improve our algorithms using heuristics similar to the ones used in [13, 18]. Finally, in Section 3.4, we compare the I/O cost and query performance of our bulk insertion algorithms with that of the one proposed in [20].

3.1 Our implementation

In order to evaluate the practical significance of our buffer algorithm, we implemented the original repeated insertion algorithm for R-tree construction [17], the bulk insertion algorithm developed in [20], and our proposed buffered bulk insertion algorithm. We also implemented the standard rquery algorithm [17]. Our implementations were done in C++ using TPIE [31, 6]. TPIE supports both a *stream-oriented* and a *block-oriented* style of accessing secondary storage. A TPIE stream represents a homogeneous list of objects of an arbitrary type, and the system provides I/O-efficient algorithms for scanning, merging, distributing, and sorting streams. The block-oriented part of TPIE supports random accesses to specific blocks. TPIE supports several methods for actually performing the I/Os. All these methods work on standard UNIX files. For example, one method uses the standard I/O system calls `fread` and `fwrite`, while another relies on memory mapping (the `mmap` and `munmap` calls). In our experiments, we used the method based on memory mapping and when we refer to the number of I/Os performed by an algorithm we refer to TPIE's count of how many `mmap` operations were performed (both reads and writes involve mapping a block). The actual physical number of I/Os performed is very likely to be less than this count, as the operating system can choose to keep a block in internal memory even after it is unmapped.

A conceptual benefit of our algorithms is that from an abstract point of view they are similar to the normal algorithms where the operations are performed one by one. Our algorithms admit a nice modular design because they only access the underlying R-tree through the standard routing and restructuring procedures. We used the block-oriented part of TPIE to implement a standard R-tree [17] that served as a base for realizing the different update approaches. Using the stream-oriented part of TPIE we stored all the buffers of an index in one separate stream. As a consequence of this modular design we are able to attach buffers to any existing index, regardless of how it has been created. After all of the bulk operations have been performed (and all buffers have been emptied) we can even decide to detach the buffers again without affecting the updated index. Therefore our buffer algorithm can be regarded as a generic "black box" that takes an arbitrary index and returns an updated version while only accessing the public interface of the index.

Our experiments were performed on a machine with a block size of 4 Kbytes (Sun SparcStation20 running Solaris 2.5) which allowed for an R-tree fanout of 100. However, following recommendations of previous empirical studies [9, 17], we only used a maximal fanout of 50. Similarly, we used a minimal fanout of 50/6 which has previously been found to give the best query performance. For simplicity, we added buffers in our implementation to all nodes instead of only to nodes on every $\lfloor \log_B \frac{M}{4B} \rfloor$th level. Theoretically, by using a buffer size of only B blocks we obtain a bulk loading I/O bound of $O(\frac{N}{B} \log_B \frac{N}{B})$. In practice, this is not significantly different from the theoretical $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ bound obtained in Section 2.

As test data we used the standard benchmark data used in spatial databases, namely rectangles

State	Category	Size	Objects	Category	Queries	Results
Rhode Island (RI)	Roads	4.3 MB	68,278	Hydrography	701	1,887
Connecticut (CT)	Roads	12.0 MB	188,643	Hydrography	2,877	8,603
New Jersey (NJ)	Roads	26.5 MB	414,443	Hydrography	5,085	12,597
New York (NY)	Roads	55.7 MB	870,413	Hydrography	15,568	42,489

Table 2: Characteristics of test data.

obtained from the TIGER/Line data set [29] for the states of Rhode Island, Connecticut, New Jersey and New York. In all our bulk loading and updating experiments we used rectangles obtained from the road data of these states. Our query experiments consisted of overlap queries with rectangles obtained from hydrographic data for the same states. The queries reflect typical queries in a spatial join operation. In order to work with a reasonably small but still characteristic set of queries, we used every tenth object from the hydrographic data sets. In the construction and updating experiments, we used the entire road data sets. The sizes of the data sets are given in Table 2. The third column shows the size of the road data set *after* the bounding rectangles have been computed, that is, it is the actual size of the data we worked with.

For simplicity we in the following only present the results of query experiments performed without buffers. When buffers were used, we observe speed-ups similar to the speed-ups observed in the bulk loading experiments presented in the next section. For example, when querying an R-tree built upon the the RI (CT) road data set with all rectangles obtained from the corresponding hydrography data set without using buffers the overall number of I/O operations was around $60,000$ ($310,000$). In contrast, performing the queries with buffers reduced the I/O count to less than $4,000$ ($16,000$)—an improvement by at least a factor of 15!

3.2 Effects of adding buffers to multidimensional indexes

In order to examine the general effects of adding buffers to a multidimensional index, we first performed a set of bulk loading experiments where we compared the performance of our algorithm (in the following referred to as BR for buffer R-tree) with that of the standard repeated insertion algorithm. (Since the repeated insertion algorithm takes unsorted data and inserts it, we will use UI when referring to it.) We performed experiments with buffers of size $\lfloor B/4 \rfloor$, $\lfloor B/2 \rfloor$ and $\lfloor 2B \rfloor$ blocks. With our choice of $B = 50$ this corresponds to buffers capable of storing 600, 1250 and 5000 rectangles, respectively.

Figure 7 (a) shows that our algorithm (BR) dramatically outperforms the repeated insertion algorithm (UI) as expected. Depending on the buffer size, we reduce the number of I/O operations by a factor of 16–24. Our experiments on query performance given in Figure 7 (b) show that the introduction of buffers affects the structure of the resulting index. More specifically, too large a buffer size decreases query performance for the following reason: When inserting objects one-by-one the routing rectangles in the tree are updated after each operation and thus rectangles are always routed the best possible way (according to the chosen heuristics). When using buffers on the other hand, the inserted rectangles have to be routed through the buffers to the leaves before they affect the routing rectangles. The larger buffers the later the rectangles reach the leaves and the later the routing rectangles are updated. Thus some later rectangles may be routed to non-optimal subtrees. However, our experiments also show that by carefully tuning the buffer size we are able to produce an index of the same quality as the index produced by UI.

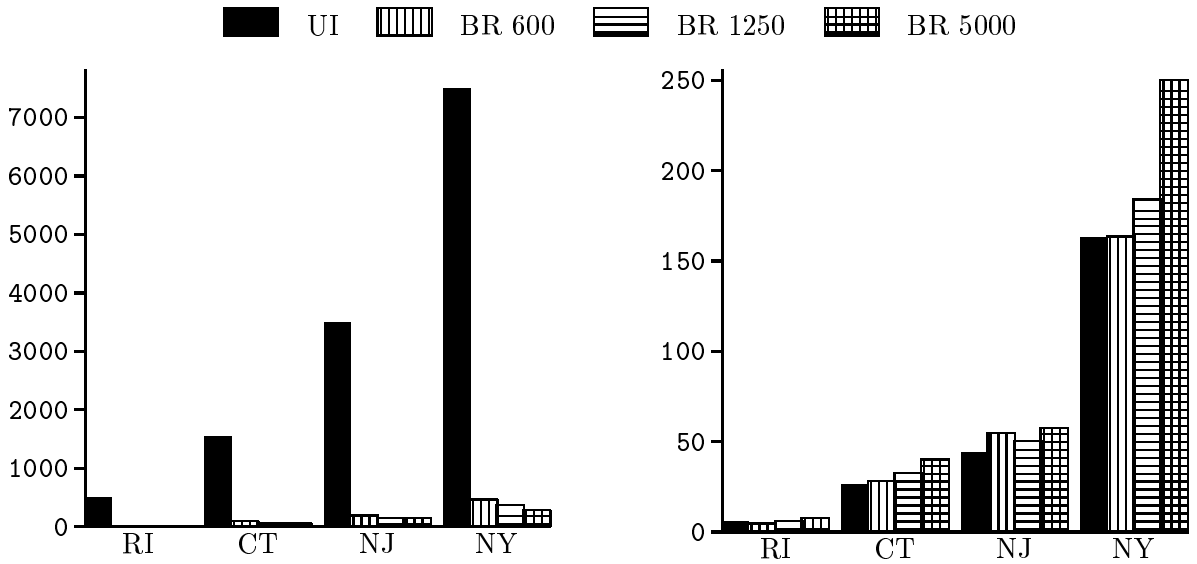
Our conclusion from this first set of experiments is that buffers can indeed be used to speed up spatial index algorithms without sacrificing query performance. Since our main objective was to design and test algorithms capable of performing more general bulk operations than just bulk loading, we did not empirically compare our bulk loading algorithm with other such algorithms (for

a conceptual comparison see Section 2.5). The purpose of this first set of experiments was just to examine the effects of buffering on the behavior of index structures where the query performance is sensitive to the order in which data elements are inserted. Specialized bottom-up R-tree bulk loading algorithms are likely to outperform our algorithm because of the smaller number of random I/Os (as opposed to sequential I/Os) performed in such algorithms. For this reason, a wise choice for bulk loading—as discussed in Section 2.5—would be to construct the index using the best bulk loading heuristic for the specific problem at hand and then attach buffers to speed up bulk insertions, bulk queries, and bulk deletes.

3.3 Improving the space utilization

As discussed in the introduction, special-purpose bulk loading algorithms often produce significantly better indexes than the ones obtained using repeated insertion, especially in terms of space utilization. There exist several heuristics for improving the space utilization of two-dimensional R-trees from less than 70% [9] to almost 95% [13, 18, 21]. One key question is therefore whether we can take advantage of some of these heuristics in our buffer algorithm in order to improve space utilization, and without sacrificing the conceptual advantage of not having to know all updates by the start of the algorithm.

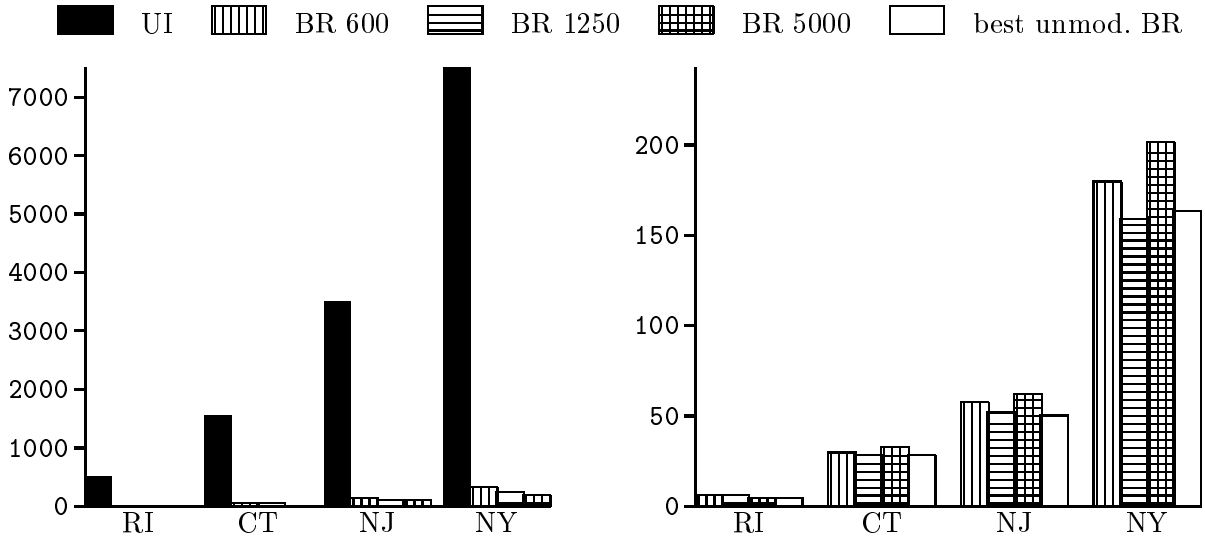
It turns out that by modifying our buffer emptying algorithm for nodes on the level just above the leaves we are indeed able to combine the advantages of our buffering method and the so-called Hilbert heuristic [18]: As in the algorithm discussed in Section 2 we start by loading all leaves into internal memory. Instead of repeatedly inserting the rectangles from the buffer using the standard algorithm, we then sort all the rectangles from the buffer and the leaves according to the Hilbert values of their center. The rectangles are then grouped into new leaves that replace the old ones. Following the recommendations in [13] we are careful not to fill the leaves completely. Instead we fill them up to 75% of capacity and include the next candidate rectangle only if it increases the area of the minimal bounding rectangle of the rectangles in the leaf by no more than 20%. Since the space utilization of the tree is mainly determined by how much the leaves are filled, this modification improves our space utilization significantly. The main reason for selecting the Hilbert heuristic was that this heuristic is very well suited for the kind of two-dimensional data we are working with [18]. The general idea in our improvement works for other heuristics as well. For



(a) Cost of building the R-tree

(b) Cost of querying the R-tree

Figure 7: Effects of buffering on build and query performance (I/O in thousands)



(a) Cost of building the R-tree

(b) Cost of querying the R-tree

Figure 8: Building and querying with modified buffer emptying algorithm (I/O in thousands)

high-dimensional point data for example, we could easily replace the Hilbert heuristic by a (variant of) the method by Berchtold *et al.* [10].

In order to evaluate the modified algorithm, we repeated the bulk loading experiments from Section 3.2. The results of these experiments were very encouraging: With all buffer sizes we improved space utilization to approximately 90% and at the same time we improve the construction time by around 30% compared with our original algorithm. We also noticeably reduced the query time of the algorithm using the largest buffer size and for each data set we were able to produce at least one index matching the query performance obtained by the original buffer algorithm. In most cases we were even able to produce a better index than the one produced using repeated insertions. The results are summarized in Figure 8. The detailed I/O costs of the experiments, as well as of the experiments from Section 3.2, are presented in Table 3.

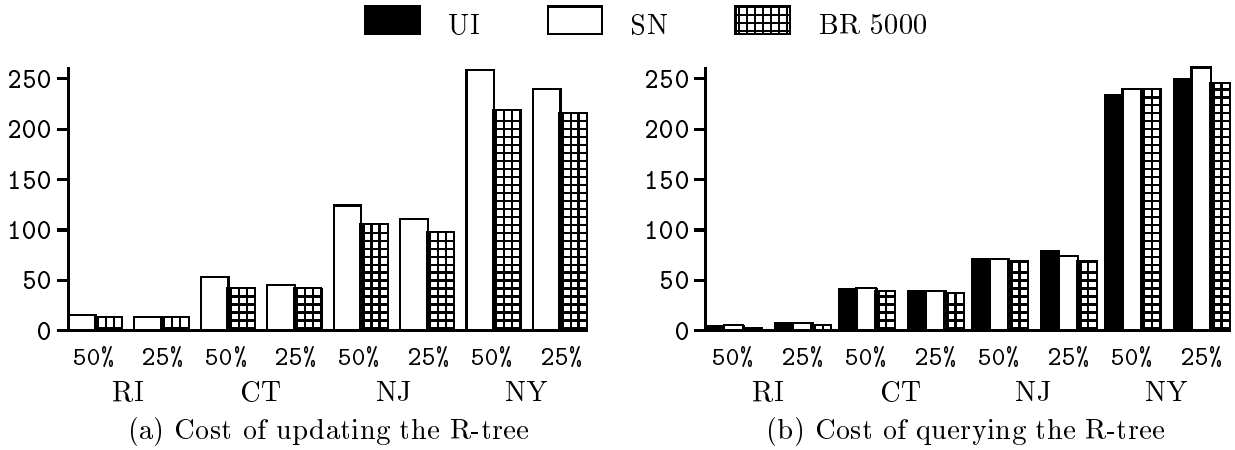
3.4 Bulk updating existing R-trees

Finally, in order to investigate the practical efficiency of our technique when performing more general bulk operations, we performed experiments with bulk insertion of a set of rectangles into an already existing R-tree. We compared the performance of our buffer algorithm (with a buffer size of 5000 rectangles) with the naive repeated insertion algorithm (UI), as well as with the bulk update algorithm of Kamel et al. [20]. As mentioned, this algorithm sorts the rectangles to be bulk inserted according to the Hilbert value of their centers and groups them into blocks. These blocks are then inserted using the repeated insertion algorithm. (Since this algorithm sorts the rectangles and groups them into nodes, we refer to it as SN.) To allow for a fair competition we used the I/O-efficient external sorting algorithm in TPIE to sort the rectangles. We allowed the sort to use 4 Mbytes of main memory. This should be compared to the maximal internal memory use of around 300 Kbytes of the buffer algorithm. The 4 Mbytes represent what is typically allocated to a single process in a database environment. Increasing the allocated memory would result in a certain speed-up for the sorting part of the algorithm, while insertions and queries would not be affected by the size of the available memory. Note that allocating a larger amount of memory would mean that the algorithms would not really be on-line since a large number of inserts would be collected before being performed.

For each state we constructed two base R-trees with 50% and 75% of the objects from the road

Data Set	Buffer Size	Building		Querying		Packing	
		Standard	Modified	Standard	Modified	Standard	Modified
RI	0	495,909	495,909	5,846	5,846	56%	56%
	600	32,360	23,801	5,546	5,858	60%	88%
	1,250	26,634	16,140	6,429	6,632	60%	89%
	5,000	21,602	11,930	8,152	5,322	59%	90%
CT	0	1,489,278	1,489,278	27,699	27,699	56%	56%
	600	94,286	74,244	28,569	28,947	59%	88%
	1,250	76,201	50,983	32,586	27,591	59%	90%
	5,000	62,584	38,588	40,600	32,352	60%	91%
NJ	0	3,376,188	3,376,188	43,156	43,156	56%	56%
	600	211,467	167,401	53,874	56,844	59%	88%
	1,250	168,687	117,971	50,183	50,743	59%	90%
	5,000	142,064	92,578	57,571	61,485	59%	91%
NY	0	7,176,750	7,176,750	160,235	160,235	56%	56%
	600	448,645	345,038	160,232	175,424	59%	88%
	1,250	357,330	250,704	180,205	155,338	59%	89%
	5,000	299,928	203,165	243,704	196,972	59%	90%

Table 3: Summary of the I/O costs for all construction experiments.



(a) Cost of updating the R-tree

(b) Cost of querying the R-tree

Figure 9: Bulk updating results for UI, SN and BR (buffer 5000) (I/O in thousands)

data set using BR and inserted the remaining objects with the three algorithms.[6] Experiments showed that the algorithms SN and BR outperform UI with respect to updating by the same order of magnitude, and that our algorithm (BR) additionally improves over SN. (Refer to Figure 9 (a).) As far as query performance is concerned, our experiments showed that SN results in worse indexes than the repeated insertion algorithm (UI). On the other hand, the indexes generated by our algorithm (BR) are up to 10% better than the indexes produced by SN, and they match or even improve the query performance obtained by indexes updated using UI. (See Figure 9 (b).) The major problem with SN with respect to query performance is that it does not take into consideration the objects already present in the R-tree; all new leaves are built without looking at the index to be updated.

[6] To avoid clustering effects, we did not build the R-trees using the first 50% (resp., 75%) of the data, but instead we skipped every second object (resp., every fourth object) in the data set when building the trees.

Data Set	Update Method	Update with 50% of the data Building	Querying	Packing	Update with 25% of the data Building	Querying	Packing
RI	UI	259,263	6,670	64%	145,788	8,148	66%
	SN	15,865	7,262	92%	14,490	8,042	91%
	BR	13,484	5,485	90%	13,301	6,981	91%
CT	UI	805,749	40,910	66%	428,163	39,016	69%
	SN	51,086	40,593	92%	44,236	39,666	91%
	BR	42,774	37,798	90%	42,429	35,968	90%
NJ	UI	1,777,570	70,830	66%	943,992	66,715	71%
	SN	120,034	69,798	92%	106,712	71,383	91%
	BR	101,017	65,898	91%	95,823	66,030	91%
NY	UI	3,736,601	224,039	66%	1,988,343	238,666	71%
	SN	246,466	230,990	92%	229,923	249,908	91%
	BR	206,921	227,559	90%	210,056	233,361	90%

Table 4: Summary of the I/O costs of all update experiments.

The detailed I/O costs of all update experiments are given in Table 4. Our experiments show that our bulk update algorithm outperforms the previously known algorithms in terms of update time, while producing an index of at least the same quality. The overall conclusion of our experiments is that our buffer technique is not only of theoretically interest, but also a practically efficient method for performing bulk updates on dynamic R-trees.

4 Conclusions

In this paper, we have presented a new buffer algorithm for performing bulk operations on dynamic R-trees that is efficient both in theory and in practice. Our algorithms allows for simultaneous updates and queries which is essential in many on-line environments. Furthermore, all operations can be pipelined. One key feature of our algorithm is that from a high level point of view the bulk operations are performed precisely as if buffers were not used. For example, our bulk insertion algorithm is conceptually identical to the repeated insertion algorithm. From an implementation point of view another key feature of our algorithm is that it admits a nice modular design because it only accesses the underlying index through the standard routing and restructuring routines. Having implemented our buffering algorithms we can thus combine them with the most efficient existing index implementation for the problem class at hand.

References

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

[2] L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In S. G. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, 4th International Workshop, WADS '95*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 1993.

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

[4] L. Arge. External-memory algorithms with applications in geographic information systems. In M. J. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer, 1997.

[5] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2001. (To appear).

[6] L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[7] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19.2 of *SIGMOD Record*, pages 322–331. ACM Press, June 1990.

[10] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Advances in Database Technology – EDBT '98, 6th International Conference on Extending Databases Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 1998.

[11] D. E. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[12] R. F. Cromp. An intelligent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In *S. R. Tate ed., Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community, CESDIS Technical Report Series, TR–93–99*, pages 75–84, 1993.

[13] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server Paradise. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 558–569. Morgan Kaufmann, 1994.

[14] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997. A preliminary version appeared in the *Proceedings of the Twelfth International Conference on Data Engineering* (1996), pages 152–159.

[16] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 606–615. IEEE Computer Society, 1989.

[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD '84, Proceedings of Annual Meeting*, volume 14.2 of *SIGMOD Record*, pages 47–57. ACM Press, June 1984.

[18] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM '93)*, pages 490–499, 1993.

[19] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 500–509. Morgan Kaufmann, 1994.

[20] I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In M. Kraak and M. Molenaar, editors, *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH '96)*, pages 3B.31–3B.42, 1996.

[21] S. T. Leutenegger, M. A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In A. Gray and P.-Å. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 497–506. IEEE Computer Society Press, 1997.

[22] D. B. Lomet. Grow and post index trees: Role, techniques and future potential. In O. Günther and H.-J. Schek, editors, *Advances in Spatial Databases – Second International Symposium (SSD'91)*, volume 525 of *Lecture Notes in Computer Science*, pages 183–206. Springer, 1991.

[23] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. J. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer, 1997.

[24] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25.2 of *SIGMOD Record*, pages 259–270. ACM Press, June 1996.

[25] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In J. M. Peckman, editor, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26.2 of *SIGMOD Record*, pages 89–99. ACM Press, June 1997.

[26] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In S. B. Navathe, editor, *Proceedings ACM-SIGMOD International Conference on Management of Data*, volume 14.4 of *SIGMOD Record*, pages 17–31. ACM Press, December 1985.

[27] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[28] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518. Morgan Kaufmann, 1987.

[29] *TIGER/Line™ Files, 1995 Technical Documentation.* Prepared by the U.S. Bureau of Census, Washington, DC, July 1996. Accessible via URL `http://www.census.gov/geo/www/tiger/tl95doc.html` (accessed 06 Feb. 1999).

[30] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 406–415. Morgan Kaufmann, 1997.

[31] D. E. Vengroff. A transparent parallel I/O environment. In *Proceedings of the DAGS Symposium*, 1994.

[32] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, 1996.

[33] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, in press. An earlier version entitled "External Memory Algorithms and Data Structures" appeared in *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, 1–38.