Sort-based Query-adaptive Loading of R-trees

Technical Report

Daniar Achakeev
Department of Mathematics and Computer Science
Phillips-Universität Marburg
Marburg, Germany
achakeye@mathematik.uni-marburg.de

Bernhard Seeger
Department of Mathematics and Computer Science
Phillips-Universität Marburg
Marburg, Germany
seeger@mathematik.uni-marburg.de

Peter Widmayer
Institut für Theoretische Informatik
ETH Zürich
Zürich, Switzerland
widmayer@inf.ethz.ch

ABSTRACT

Bulk-loading of R-trees has been an important problem in academia and industry for more than twenty years. Current algorithms create R-trees without any information about the expected query profile. However, query profiles are extremely useful for the design of efficient indexes. In this paper, we address this deficiency and present query-adaptive algorithms for building R-trees optimally designed for a given query profile. Since optimal R-tree loading is NP-hard (even without tuning the structure to a query profile), we provide efficient, easy to implement heuristics. Our sort-based algorithms for query-adaptive loading consist of two steps: First, sorting orders are identified resulting in better R-trees than those obtained from standard space-filling curves. Second, for a given sorting order, we propose a dynamic programming algorithm for generating R-trees in linear runtime. Our experimental results confirm that our algorithms generally create significantly better R-trees than the ones obtained from standard sort-based loading algorithms, even when the query profile is unknown.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access methods

General Terms

Algorithms, Performance

Keywords

R-tree, Bulk-loading, Dynamic Programming, Z-Curve

1. INTRODUCTION

Index bulk-loading of very large data sets has been an important problem in database research. Loading is necessary when an index has to be built up for the first time. Moreover, clustered indexes also require periodical reload to reestablish the clustering of records. It is well known that loading indexes by inserting tuples one by one is less efficient than specially designed bulk-algorithms that run with the same complexity as external sorting. Bulk-loading is therefore an interesting option for supporting updates on indexes by buffering updates and reloading the index after the buffer is sufficiently filled up.

While there is a standard approach for loading of B-trees, many different techniques [14, 12, 11, 3, 18, 24, 17] were proposed for multidimensional indexes like R-trees. For loading R-trees, there is always a tradeoff between loading efficiency and index quality, i.e., how efficient an R-tree can support queries (and updates). Despite the large number of loading strategies, none appear to be ideal:

First, current loading strategies for R-trees do not (similar to B-trees) consider the query profile. Ignoring the query profile has only a minor impact on the quality of B-trees, but might result in poorly loaded R-trees. Consider the example of two extreme query profiles for a two-dimensional data set: One profile only contains partial exact match queries (*pemq*) with an exact match in the first dimension, while the other contains *pemq* with exact matches in the second dimension. Note that the ideal R-tree would actually be a B+-tree on the first and second dimension, respectively. All loading algorithms for R-trees ignore the query profile and would build the same R-tree although neither is designed for these extreme cases.

Second, many of the sophisticated loading algorithms seem to be quite complicated to implement. While these algorithms create R-trees with excellent worst-case performance (see [2]), integration into a system turns out to be quite difficult. It is therefore not surprising that less complex loading algorithms for R-trees are used in commercial systems like STR [18] and popular sort-based loading strategies [17]. However, the resulting R-trees are not optimized for specific queries. So far, a theoretical foundation of these methods does not exist.

In this paper, we revise the problem of loading R-trees. We aim to design algorithms for query-adaptive loading R-trees optimized in respect to a given query profile. Here, we focus on sort-based techniques because of their conceptual simplicity. This makes these techniques very appealing for commercial systems where other approaches are difficult to use due to their implementation complexity.

In this paper we:

1. Design a novel (sort-based) algorithm that takes the underlying query profile into account.
2. Show NP-hardness of the optimal R-tree loading.

3. Propose algorithms based on dynamic programming (DP) for generating optimal R-trees given a specific sort order. One of them requires linear runtime and space.

4. Present a new approach to determine sorting orders that improve the index for non-square windows.

5. Demonstrate the practical performance of our algorithms. Our resulting R-trees provide a better quality than competitors, even in the case of unknown query profile.

In Section 2, we provide a detailed discussion of related work and the major differences to our work. In Section 3, 4 and 5, our algorithmic framework is presented. We present first the NP-hardness of the optimal R-tree loading problem and a new sort-based heuristic before we discuss the computation of appropriate sorting orders. An extensive experimental evaluation on a standardized test framework is presented in Section 6. Last but not least, we provide a conclusion and an outlook to future work.

2. RELATED WORK

This section reviews previous bulk-loading methods for R-trees under the assumptions that a set of N d-dimensional rectangles is given and that the performance is measured by the number of accesses to nodes of fixed size. Each of the nodes is able to keep at most B rectangles. Note that B decreases linear in d.

The most generic method for loading R-trees is to apply standard insertion algorithms to each of the input rectangles. The loading time is then $O(N \log_B N)$, while the query performance solely depends on the underlying insertion algorithm. Insertion algorithms are designed in such a way that a goal function should be optimized for a split. In [21, 17], a cost model was introduced revealing that the perimeter and the area are the two crucial performance indicators. However, [6] shows that an optimal split of a node does not lead to globally optimal R-trees. This cost model provides the basis for our investigations.

Roussopoulos and Leifker [24] introduced the problem of loading a R-tree from scratch and presented a sort-based loading technique with complexity $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where M denotes the available main memory. After sorting the rectangles according to a one-dimensional criterion, an R-tree can be built bottom-up like it is known from B+-trees. Because the sorting order has a considerable impact on the search efficiency, Kamel and Faloutsos [17] proposed a double-transformation: first a rectangle is mapped to a multidimensional point and then a space-filling curve like the Hilbert-curve is used to generate a one-dimensional value. In order to improve query performance, heuristics like the one proposed in [13] can be used for local data reorganization.

STR [18] is also a sort-based loading algorithm that is conceptually different from the simple sort-based algorithms mentioned above. d different sort and partitioning phases are used, one for each dimension. The partitions after the last sort correspond to the leaf pages of the target R-tree. The advantages of sort-based loading strategies are their simplicity of implementation yet a good query performance. Therefore, they are the only methods currently used in DBMS and GIS. However none of these methods can guarantee the quality of the generated R-tree regarding a cost model.

The Top-down Greedy Splitting(TGS) bulk-loading method [14] constructs the tree in a top down manner by applying cost-optimized binary splits in a greedy manner. The cost function with the best experimental results [14] minimizes the area of the bounding boxes. The partitioning is performed by iterative binary steps where in each step multiple sorting orders are examined to detect the split with minimum area. In [2], it has been confirmed in experiments that the average search performance of R-trees generated by TGS are almost always better than the ones generated by other loading methods. Only for artifical data sets with highly varying aspect ratio, the priority R-tree has been superior to TGS. A main disadvantage of TGS is its high loading cost (due to the binary partitioning) that can be substantially higher than the cost for external sorting. Due to its binary steps, it is difficult to parallelize TGS in a scalable manner. Other top-down partitioning techniques like Quick-Load [11] avoid expensive binary partitioning steps, but the design of an efficient parallel version is still an open problem.

Loading techniques based on buffer-trees [12, 3] can be considered as a hybrid of top-down and bottom-up strategies. The basic idea is to delay insertions by temporarily storing input rectangles in buffers attached to the nodes. If buffers are filled up, the batch of insertions is reactivated and the rectangles continue their traversal down to the leaves. In order to achieve better search quality it is suggested using a sort-based loading strategy for buffer emptying above the leaves. While the loading efficiency is the same as for external sorting, the underlying split algorithm (except for the leaf level) determines the query performance.

The priority R-tree (PR-tree) [2] is the first loading method, whose target R-trees provide worst-case guarantees for window queries, while the loading can be performed with the same complexity as external sorting. It also has been shown [2] that the practical performance of the PR-tree is also good for two-dimensional data. However, in most cases it is not as good as for the R-trees of TGS, which is the only cost-model sensitive loading technique so far. In fact, the PR-tree is not primarily designed for improving the average-case performance according to a cost model and a query profile. Moreover, its high implementation complexity might prevent it from being considered in a real system.

The theoretical foundations of the loading problem has been addressed in [22] where the NP-hardness of the bucket optimization problem has been proven, but only for a specific artificial cost function that substantially differs from the ones that are commonly used for R-trees [21, 17, 27]. This is contrary to our work where NP-hardness is shown for the cost function [21, 17] minimizing the area of bounding boxes.

None of the previous methods have been designed for *query-adaptive loading* of R-trees. Query-adaptive loading refers to the problem of generating R-trees whose average performance is minimized regarding a given static profile. This is

in contrast to adaptive indexing techniques like splay-trees [25] and database cracking [15], which apply structure adaptations during runtime of the queries. Different adaptive R-trees have been proposed in the literature [7, 26, 8], but all of them require a mix of queries and insertions to obtain the full benefits of adaptivity.

One of our approaches to query-adaptive loading relies on space-filling curves (SFCs) to obtain an one-dimensional ordering of the rectangles and on an optimal assignment of rectangles to pages. Most of the other approaches to using SFCs for sorting multidimensional data like [17] shuffle the bits in a symmetric manner, which is most suitable when every dimension provides the same selectivity. Orenstein and Merett presented a more flexible framework for shuffling bits that allows the definition of different sorting orders [20]. Based on their framework, we present shuffling strategies that adapt to the underlying query profile. A theoretical foundation for generation of query-adaptive space-filling curves was developed in [5], but without considering the specific problem of bulk-loading. In addition to sorting, we also address the problem of data partitioning over a set of pages. The common packing strategy [17] to fill up pages to the maximum leads to suboptimal query performance. In contrast, our new partitioning strategy relies on the dynamic programming framework used for generating optimal histograms [16].

The dynamic programming scheme proposed in [16] is also used in [28] for computing a set of k minimal bounding rectangles (MBR) from a 2-dimensional point set. The goal was to reduce communication costs for mobile devices by approximating the spatial query result by a set of MBRs with a minimal information loss f_i. The authors showed that computing such representations is NP-hard even for d=2. One of their heuristics first sorts the query output using the Hilbert order and then apply the partitioning method of [16]. In contrast to bulk-loading, space constraints are disregarded. In this work, we show that these constraints allow for the design of more efficient algorithms.

3. QUERY-ADAPTIVE LOADING

3.1 Preliminaries

In this paper, we address the problem of R-tree loading for a d-dimensional set of N rectangles $\{r_1, \ldots, r_N\}$. R-trees consist of pages with maximum capacity B and minimum occupation $b \leq \lceil B/2 \rceil$. We consider the case $d = 2$, the generalized case for $d > 2$ is only discussed when necessary.

For query-adaptive loading, we assume that a query profile QP for range queries is given. QP provides a (statistical) model that is derived from a collection of representative queries. For the sake of simplicity, we consider the query profile for range queries that is given by the average size of the range in each dimension. For $d = 2$, $QP = (sx, sy)$, where sx and sy is the average size of the range query in the first and second dimension, respectively. We assume that queries, more precisely their centers, are uniformly distributed in the underlying domain. This assumption is obviously not satisfied in a real application. The standard approach to overcome this deficiency is to use multidimensional histograms and to maintain these parameters for each histogram cell independently [1, 23]. This approach has already been used successfully for the analysis of R-trees [27].

3.2 Basic Idea

For a given query profile $QP = (sx, sy)$, our goal is to generate R-trees whose average number of leaf accesses is minimized for queries derived from QP, as they dominate the overall cost for sufficiently large range queries. Moreover, upper levels of the trees are often located in memory, while leaf pages are generally not.

Our goal is to create optimal R-trees level by level, bottom-up. However, as we show in Section 3, the problem of generating optimal R-trees is NP-hard and, therefore, sort-based heuristics are examined traversing the following five steps:

1. **Determination of Sort Order:** For a given QP determine a sort order that minimizes the cost.
2. **Sorting:** Sort the rectangles with respect to the determined order.
3. **Partitioning:** Partition the sorted sequence into subsequences of size between b and B and store each of them in a page.
4. **Generation of Index Entries:** For each page, compute the bounding box of its partitions and create the corresponding index entry.
5. **Recursion:** If the total number of index entries is less than B, store them in a newly allocated root. Otherwise, start the algorithm with the generated index entries (bounding boxes) from Step 4.

Step 2 and Step 4 are very similar to the traditional sort-based loading of R-trees [24]. The crucial optimization occurs in the first and third step. Step 1 computes a sort order from the query profile. We exploit the fact that a space-filling curve (SFC) does not require a symmetric treatment of dimension, but allows more flexibility. As an example, consider that only partial exact match queries orthogonal to the x-axis should be supported. Thus, the sort order should be only influenced by the x-value, which corresponds to a SFC where all bits of the x-axis should precede the bits of the y-axis. The problem we address in Section 5 is therefore to compute the shuffle order from the query profile. In step 4, the rectangles are then assigned to pages such that the capacity constraints of the R-tree are met. Filling up pages to the maximum (or as generally suggested to a constant degree) does not lead to R-trees optimized with respect to the given query profile. High storage utilization is only useful for fairly large queries, while the performance of smaller queries suffer. In Section 4, we present a heuristic partitioning algorithm that is optimized according to the underlying query profile. Both steps make use of a cost model that is derived from our query profile. The cost model is presented in detail in section 3.3.

3.3 Cost Model

Our work is based on cost models proposed by [17, 21]. As given in [21], range queries are classified according to aspect ratio, location and size. If the aspect ratio is set to 1:1, there

are two possibilities for each location and size property. For location property, query rectangles follow either data or uniform distribution. The size of queries can be defined either as area or number of objects. Combining these two properties results in 4 different range query models (WQM_{1-4} as defined in [21]). The simplest WQM_1 models the uniform distribution of query rectangles with a equal area and aspect ratio 1:1. In the following, we illustrate the case for $d = 2$.

Assume that the domain corresponds to the two-dimensional unit square $[0, 1)^2$. A rectangle $r_i = (cx_i, cy_i, dx_i, dy_i)$ is represented by its center (cx_i, cy_i) and its extension (dx_i, dy_i). For a window query $WQ_{q,s}$ given by its center $q = (qx, qy)$ and its extension $s = (sx, sy)$, the probability of a rectangle r_i intersecting the window is $(dx_i + sx) \cdot (dy_i + sy)$. The average number of rectangles intersecting the query window is then given by:

$$\sum_{i=1}^{N} (dx_i + sx) \cdot (dy_i + sy) \quad (1)$$

Note that for point queries with $s = (0, 0)$, the equation computes the sum of MBR volumes. We obtain the expected number of leaf accesses, which is a typical performance indicator for R-trees, by applying the formula to the set of bounding boxes of the leaves.

4. OPTIMAL PARTITIONING

In this section, we show that the problem of partitioning a set of rectangles R is NP-hard, given that each bucket p_i from partition $P_{b,B} := p_1, \ldots, p_n$ has $b \leq |p_i| \leq B$ rectangles, so that for a given weight function $w : p_i \to \mathbb{R}^+$, the sum of weights is minimized. Let $MBB(p)$ be a minimal bounding box of bucket p. According to the cost model, the weight function $w := area(MBB(p))$ is an $MBB(p)$'s area. Based on these results, we develop a heuristic approach that optimally solves the partitioning problem for a given sorting order and $area(MBB(p))$. The justification for a heuristic approach lies in the NP-hardness of the problem.

THEOREM 1. *The problem of partitioning $P_{b,B}$ for N given rectangles that minimizes the sum of minimum bounding boxes areas of the buckets is NP-hard.*

To prove the theorem, it suffices to consider the special case of $B = 3, b = 2$ (as $b = \lceil B/2 \rceil$) and a 2-dimensional space. The proof uses a polynomial time reduction from the version of planar 3SAT [19, 28] in which for each variable, also an edge can be embedded in the plain. The edge is suited between the positive and the negative literal of a variable. We start by turning a given planar embedding (that is, a planar graph G) of a planar 3SAT formula into a rectangle set. Roughly speaking, we create for each variable a gadget of three rectangles that serves to select a truth value (item 1 below). For each clause, we create a rectangle (item 2). For each edge in the planar 3SAT graph, we create a chain of large and small rectangles that propagates the choice of truth value (item 2). Whenever a literal appears in more than one clause, we create a docker gadget to let the chosen truth value propagate to all clauses in which it appears (item 3). All gadgets use rectangles of two sizes, *big* and *small* ones. For concreteness, a big rectangle can be 2 by 2 units, and a small 1 by 1.

Figure 1: Left: Variable and clause gadget; Right: Edge chain gadget

1. For both the positive and the negative literal of each variable, we create a big rectangle. We make both rectangles of a variable appear almost next to each other, with a single big rectangle (we call this a *xor*) in between and touching them, positioned as in Fig. 1 left. The existence of a connecting edge in G between both literals makes sure we can put these three rectangles next to each other, and nothing else interferes. For each clause c_j a big rectangle will be placed in the plane at a location to be described.

2. For each edge in G between a literal l_i and a clause c_j (recall that this reflects that a literal appears in a clause) an alternating sequence of one big and two small rectangles will be placed to connect them (see Fig. 1 right, rectangle borders are meant to touch and be shared in all figures). It is important to note that such a *rectangle chain* starts and ends with a pair of small rectangles. Further, note that the chain can bend so that it can follow any embedded 3SAT edge.

3. Whenever a literal is contained in more than one clause, i.e. whenever more than one edge attaches to the literal, then *docker* rectangles for these edges are attached as is shown in Fig. 2 ("long" shaded rectangles). Furthermore, the construction makes sure that edge chains and clauses are spread nicely apart, so that in particular none of them intersect a bounding box of a variable gadget and these bounding boxes do not intersect each other.

The number of rectangles in the construction can be made polynomial in the 3SAT instance size with k variables and m clauses. The decision version of the partitioning problem now asks for a partition $P_{2,3}$ with bound A on the sum of bounding box areas that equals the sum of all rectangle areas in the construction, plus a certain amount of "unavoidable wasted space" that arises because each xor-rectangle needs to be boxed together with another rectangle (since $b = 2$), and an adjacent literal is the only cost effective option. For the concrete rectangle sizes above, the wasted space is 4 units per variable. Close inspection of the construction now reveals that partitioning within the given bound is possible if and only if the given formula is satisfiable. In more detail:

"⇒" If the given 3SAT-formula is satisfiable, we immediately get a rectangle partition with total area A as follows, based on a satisfying assignment:

1. We associate x_i with xor_i if the satisfying assignment sets x_i to TRUE, and $\bar{x}_i$ with xor_i otherwise.

2. We associate triples of rectangles in the edge chain gadget from each true literal towards its clause, i.e. until

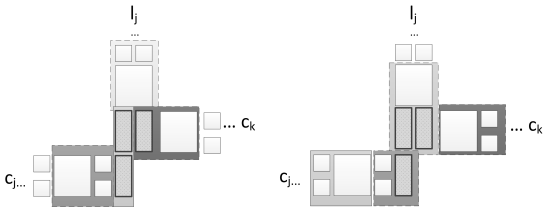

Figure 2: Ripple effect with TRUE/FALSE assignment. Shaded rectangles are *dockers* and serve to branch from the variable to multiple clauses.

only two small rectangles remain next to the clause rectangle. Each triple consists of two small rectangles towards the literal and a big rectangle towards the clause(see Fig. 2). For reasons of space, we do not discuss dockers in detail in this proof sketch. In this way, the clause rectangle is provided with two small rectangles for boxing together without waste of area.

3. We associate each FALSE literal with its adjacent two small rectangles in the chain, and continue along the chain by associating triples of rectangles, with the big rectangle towards the false literal and both small towards the clause (Fig. 2). This leaves no pair of small rectangles for associating with the clause rectangle, corresponding to the fact that the literal cannot make the clause true. Note that we can assume that each literal appears in some clause, because otherwise we could just simplify the 3SAT formula accordingly.

4. The construction guarantees that at each clause, there is at least one pair of small rectangles that can be associated with the clause rectangle (since we started from a satisfying assignment). We take an arbitrary one of the pairs and associate it with the clause. Each other such pair forms a box by itself.

This partition achieves area A because none of its bounding boxes contains "wasted space", i.e., area with no input rectangle, except for wasted space at each true literal.

"⇐" A close inspection of the construction reveals that area A can be reached only if there exists a satisfying assignment for the given 3SAT formula. In particular:

1. xor_i needs to be associated either with x_i or $\bar{x_i}$. It cannot stay alone, due to the lower bound $b = 2$, and it cannot be packed together with both without exceeding the allowable wasted space. Whenever xor_i is associated with l_i, we set l_i to true (and $\bar{l_i}$ to false).

2. Along an edge chain, a FALSE literal needs both adjacent small rectangles, or else it would violate the lower bound or create wasted space. This effect ripples along the edge chain , ending with no pair of small rectangles next to the clause and therefore it does not allow the clause to be grouped without wasted space to it, in line with the fact that the literal cannot make the clause true.

3. We just mention that a docker propagates this ripple effect or else it creates wasted space, without explaining the details.

To summarize, the given planar 3SAT formula is satisfiable if and only if we can partition the rectangle set with bound A on the sum of areas.

4.1 Sorted Set Partitioning

In this section we consider the problem of query-adaptive partitioning a sorted sequence $r_1, \ldots, r_N$ of rectangles such that each bucket of the partition corresponds to a page of the R-tree. This approach is a heuristic that is based on the specific sorting order, since the computation of an optimal partition is NP-hard for $area(MBB(p))$. Every bucket corresponds to a contiguous subsequence $p_{i,j} = r_i, \ldots, r_j$ such that $b \leq j - i + 1 \leq B$ is satisfied. A valid partition P consists of the subsequences $p_{i,j}$ such that each rectangle belongs to exactly one of them. Let S_N denote the set of all valid partitions and let $S_{N,m}$ be the partitions that consist of exactly m buckets. While the standard sort-based loading strategy stores a fixed number of rectangles per page, we do not require equal numbers of objects per pages in our approach. This gives us flexibility to optimize the partition according to a given query profile QP again. Let $MBB(p)$ be the bounding box of a contiguous sequence p of rectangles. Based on the cost model (see Equation 1) we consider the following optimization problems:

1. **Storage-bounded Loading:** Compute a partition $Sm_{opt} \in S_{N,m}$ that minimizes the cost function for the set $\{MBB(p)|p \in S, S \in S_{N,m}\}$.

2. **Query-optimal Loading:** Compute a partition $S_{opt} \in S_N$ that minimizes the cost function for the set $\{MBB(p)|p \in S, S \in S_N\}$.

Note that query-optimal loading results in better partition, but the worst-case, storage utilization of the resulting R-trees can be as low as b/B. Storage-optimal loading allows to choose the desired storage utilization $(N/(m \cdot B))$ in advance by setting m.

Let $QP = (sx, sy)$ be a given query profile and $C_{QP}(S) = \sum_{p \in S} area^+(MBB(p), QP)$ be the sum of areas extended with average side length from query profile QP. More formally, $area^+(r, QP) = (dx + sx) \cdot (dy + sy)$ for a rectangle $r = (cx, cy, dx, dy)$. $C_{QP}(S)$ denotes denotes the cost of a partition $S \in S_N$ for a given query profile QP. This function has a nice property that allows the design of an efficient algorithm to compute the optimum. Consider a split of a partition S into two arbitrary partitions S_l and S_r. Then, the following property holds for our cost function:

$$C_{QP}(S) = C_{QP}(S_l) + C_{QP}(S_r)$$

In particular, equality is satisfied for the optimal partition S_{opt}. Note that, S_l and S_r must also be optimal partitions of their associated rectangles. In fact, this observation allows us to use the paradigm of dynamic programming in a similar way as for computing optimal histograms [16, 28]. For the first i rectangles and k contiguous sequences, computation

of the minimum cost $opt^*(i,k)$ is geiven by the following recursion:

$$opt^*(i,k) = \min_{b \le j \le B} \{opt^*(i-j, k-1) + C_{QP}(p_{i-j+1,i})\} \quad (2)$$

In order to compute $opt^*(N,m)$, we apply the recursive formula for all $1 \le i \le N$ and $1 \le k \le m$, in increasing order of k, and for any fixed k in increasing order of i. We store all computed values of $opt^*(i,k)$ in a table. Thus, when a new $opt^*(i',k')$ is calculated using Equation 2, any $opt^*(i,k)$ that may be needed can be read from the table. After computing the optimal cost, we can read out the contiguous sequences of the input rectangles from the dynamic programming table. Thus, the following theorem holds:

THEOREM 2. *The optimal partition $S_{N,m}$ of N rectangles into m buckets, each of them containing between b and B contiguous rectangles, can be computed in $O(N^2 \cdot B)$ time and $O(N \cdot m)$ space.*

Next, we consider query-optimal loading, the problem of computing the optimal partition without user-defined storage utilization. At first glance, the problem appears to be harder because the solution space is larger. However, the opposite is true because the parameter m has no effect on the optimal solution anymore. This results in the following simplified recursion:

$$gopt^*(i) = \min_{b \le j \le B} \{gopt^*(i-j) + C_{QP}(p_{i-j+1,i})\} \quad (3)$$

In order to compute $gopt^*(N)$, we compute the recursive formula for all $1 \le i \le N$ in increasing order of i. We store all computed values of $gopt^*(i)$ in a table. Thus, when a new $gopt^*(\acute{i})$ is calculated using Equation 3, any $opt^*(i)$ that may be needed can be read from the table. As in the case for opt^*, we obtain the result sequences from the table. Thus, the following theorem holds:

THEOREM 3. *The optimal partition S_N of N rectangles into buckets, each of them containing between b and B contiguous rectangles, can be computed in $O(N \cdot B)$ time and O(N) space.*

Theorem 3 shows that optimal loading is possible in as little as linear time. The required CPU-time is much lower compared to the optimal solution of space-bounded loading. Note that storage utilization of R-trees generated by query-optimal loading largely depends on the underlying query profile. If the query size is large, the optimal partitioning also causes high storage utilization.

Note, that opt^* and $gopt^*$ compute only the best partition for a given sequence for one level at time. To build an optimal R-tree that include all levels, we can generalize $gopt^*$ for $k-$levels. As for *gopt* we compute the best partitioning for subsequences of size $\Theta(B^l)$ for levels $l = 1, \ldots$. To limit the processing time, we adapt the approach known from weight balanced B-trees[4]. We define parameter a as a branching parameter. Let $b = 1/3B, a = 1/4B$ and $l = 1 \ldots$ then following function computes partitioning: if $l > 0$

$$g^*(i,j,l) = \min_{\frac{1}{3}Ba^l \le k \le \frac{4}{3}Ba^l} \{g^*(i, j-k, l) + g^*(j-k+1, j, l-1) + C_{QP}(p_{j-k+1,j})\}$$

else

$$g^*(i,j,0) = \min_{\frac{1}{3}B \le k \le B} \{g^*(i, j-k, 0) + C_{QP}(p_{j-k+1,j})\}$$

As for $gopt^*$ and opt^* we use a table to hold intermediate costs. Thus, the following theorem holds:

THEOREM 4. *The optimal weight-balanced R-tree with capacity parameters B, b and branching parameter $a = \frac{1}{4}B$ can be computed from a sorted sequence of rectangles in $O(N^3 \cdot \frac{1}{4}B^2 \cdot (\frac{1}{4})^{\log_B(N)})$ time and $O(N^2 \cdot \log_B(N))$ space.*

As a result, a subtree on level $l > 0$ holds $\left[\frac{1}{3}B(\frac{B}{4})^l, \frac{4}{3}B(\frac{B}{4})^l\right]$ elements in its leaf nodes and $\left[\frac{B}{16}, B\right]$ entries per index node. However, we need at least a quadratic space and time for a computation , since all possible subsequences should be considered. Consequently, the solution is not practical anymore. In our experiments we observed only a marginal improvement in comparison to $gopt^*$, since only a small subset of a data can be processed efficiently at time.

4.2 Practical Considerations

In the following, we provide some useful informations for processing a large set of rectangles. Because computing opt^* requires quadratic space,it is unlikely that the whole intermediate data sets can be processed in memory. In this case, the data set is processed as follows: we cut the data in sufficient big equi-sized chunks and apply opt^* on each of them independently. In our experiments, we observed that B^2 (where B is equal the number of rectangles in a page) is sufficient to obtain near-optimal results. For the computation of $gopt^*$, the same strategy can be applied. However, since only the last B entires are required by $gopt^*$, a buffer of B entries is sufficient for processing.

After the first level has been constructed, the index entries of the next level can be re-sorted again. However, we noticed that for a given query profile, the produced sequence of MBRs already preserves the order of the input rectangles so that we skip the extra sorting step to reduce the total build-up time.

5. OPTIMIZATION OF SORT ORDER

The quality of our partitioning algorithms depends on the chosen sorting order. Our experiments show that traditional Hilbert and Z-Curve perform very well in combination with the proposed partitioning for square query rectangles. In this section we provide an algorithm for determining the sorting order of our bulk-loading framework for the cases where the average query shape is non-square. The sorting order is defined by a SFC whose input corresponds to an appropriate shuffling of d bit sequences, where each of them of constant length L represents a dimension of the d-dimensional unit cube $[0,1)^d$. As before, we assume two-dimensional data

(a) (b) (c)

Figure 3: Impact of Sorting Orders

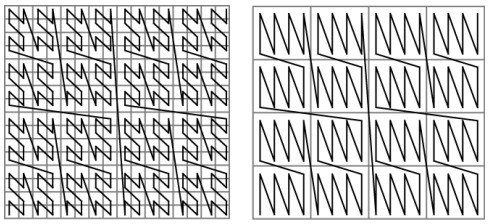

Figure 4: Left a grid with $GR = 8$, right a grid with $GR = 4$.

and discuss the general case only if necessary. Due to its flexibility, we use the Z-curve as our SFC in the following.

Our goal is to adapt to the underlying query profile $QP = (sx, sy)$. In order to model non-square window queries, we introduce here the *aspect ratio* given by $a = sy/sx$. The effect of the aspect ratio is illustrated in Fig. 3(a) where a set of range queries with a high aspect ratio is plotted. The bounding boxes of the R*-tree leaves are plotted in Fig. 3(b), while the plot of the boxes obtained from our sort-order optimized algorithm is given in Fig. 3(c). The R*-tree does not take any query profile into account and attempts to generate boxes with a quadratic shape, while our new loading algorithm adapts its boxes to the shape of the query. This query-adaptive partition causes an substantial improvement in performance compared to the standard R*-tree.

The basic idea is to introduce a two-part SFC. The first part corresponds to a SFC being defined on a non-symmetric binary grid. Each dimension of a grid is partitioned in binary manner into equi-sized intervals. The grid resolution GR is given by the total number of bits required for determining whether a point belongs to a cell. Note that the volume of the cell is 2^{-GR}. The second part combines the remaining $d \cdot L - GR$ bits in lexicographic order (see Fig 4). Note that this design of the two-part SFC allows adapting to the common cases discussed previously. In case of $a = 1$, we fully exploit the first part of our SFC, i.e., $GR = d \cdot L$, while for partial match queries, we only use the second part with an appropriate lexicographic order (given priority to the most selective dimensions). The fundamental questions are how the asymmetry of the grid is determined and how GR has to be set for a given query profile. Our goal is to design a grid such that the number of grid cells is minimized given that the volume $V = x \cdot y = 2^{-GR}$ is fixed. Here, x and y denotes the size of bounding intervals of the cell. Let a query profile be $Q = (sx, sy)$, with $sy = a \cdot sx$. These simplified assumptions allows us to use Equation 1 for estimating the average number of cells intersecting a window query. The $LC2(x, y)$ expresses the number of cells as a function of x and y.

$$LC2(x, y) = 2^{GR} \cdot (x \cdot y + x \cdot sy + y \cdot sx + sx \cdot sy) \quad (4)$$

Equation 4 can be rewritten by substituting x by V/y, sx by $\frac{sy}{a}$ and $x \cdot y$ by the constant V. Note that the average utilization is constant for different sort orders. This provides the following cost function:

$$LC(y) = 2^{GR} \cdot (V + sy \cdot (\frac{V}{y} + \frac{y}{a}) + sy^2 \cdot a) \quad (5)$$

$$LC'(y) = 2^{GR} \cdot sy \cdot (\frac{1}{a} - \frac{V}{y^2}) \quad (6)$$

Computing the root of the derivative of equation 5 yields the minimum. It directly follows that $y_{opt} = \sqrt{V \cdot a}$ minimizes $LC(y)$. In addition, we obtain $x_{opt} = \sqrt{\frac{V}{a}}$ and that the aspect ratio of the optimal cells is also equal to a again. Note that we ignore here that our optimum is not realized on the grid and some rounding is actually necessary.

In case of $d > 2$, we introduce $d-1$ aspect ratios $a_1, \ldots, a_{d-1}$ with $a_i = \frac{s_{i+1}}{s_i}$. Let $V = \prod_{1 \le i \le d} x_i$ be the average volume of a page region and x_i be the length of the i-th side of the page region. Then, LC is minimized for $x_d = (V \cdot a_{d-1})^{1/d}$, $x_i = (V \cdot \frac{a_{i-1}}{a_i})^{1/d}$ for $1 < i < d$ and $x_1 = (\frac{V}{a_1})^{1/d}$.

Let us now discuss how to set the parameter GR or equivalently the concrete size of a grid cell. There are at least two intuitive options. One is to set the average query volume equal to the average query size. Then a query hits at most four cells. As shown in [5], this minimizes the number of contiguous pieces of the SFC that intersect the query region. However, our goal is to minimize node accesses, thus we use the average size of the optimal bounding boxes of R-tree leaves (which means the optimal one obtained from the cost function 1) to determine the grid cell. The results of our experiments indicate that this option is superior to the first option. Note that the optimal bounding box offers the same aspect ratio as the window query. We use this property to initialize our algorithm with this box rather than using $d - 1$ aspect ratios and the parameter GR (see Alg. 1 for details). The input of AdaptiveShuffle consists

Algorithm 1: Algorithm AdaptiveShuffle

Input: Average edge lengths of the boxes of the leaves $(len_1, \ldots, len_d)$ with $len_i \le len_{i+1}$, d-dimensional array A of bitstrings with L bits per bitstring

Output: bitstring of length $L \cdot d$

$from = L, resString = \emptyset$;
for $k = d, \ldots, 1$ **do**
  $to = L - \left\lceil \log_2 \frac{1}{len_k} \right\rceil$;
  $resString = +SymShuffle(A, k, from - 1, to)$;
  $from = to$;
for $k = 1, \ldots, d$ **do**
  $resString = +SuffixString(A_k, L - \left\lceil \log_2 \frac{1}{len_k} \right\rceil)$;
return *resString* ;

of a d-dimensional array A of bit sequences of fixed length

L and a d-dimensional array len representing the shape of the optimized boxes. A_i denotes the value of the i-th dimension. In order to simplify the description of the algorithm, we assume that $len_i < len_{i+1}$ is satisfied, $1 \leq i < d$, without loss of generality. Each part of the two-step SFC consists of a for-loop. In the first for-loop, the routine $SymShuffle$ shuffles a certain number of bits of the first k dimensions in a symmetric manner until the selectivity of the k-th dimension is fully exploited. The symbol "=+" denotes appending the right string to the result string. This loop is iteratively performed for $k = d, d-1, \ldots, 1$. Note that the parameter GR can be computed by $GR = \sum_{1 \leq k \leq d} \left\lceil \log_2 \frac{1}{len_k} \right\rceil$. The second for-loop simply calls $SuffixString$ to append the unused bits of the k-th dimension to the result string, $k = 1, \ldots, d$. Let us consider an example for $d = 3$, $L = 6$, $A_1 = (x_5, \ldots, x_0)$, $A_2 = (y_5, \ldots y_0)$, $A_3 = (z_5, \ldots, z_0)$ and $len = (\frac{1}{16}, \frac{1}{8}, \frac{1}{2})$. From these settings, we obtain the following result string:

$$x_5, y_5, z_5, x_4, y_4, x_3, y_3, x_2, x_1, x_0, y_2, y_1, y_0, z_4, z_3, z_2, z_1, z_0$$

Note that we first interleave bits from all dimensions. After the first cycle, the z-dimension is not involved anymore. After three cycles, the asymmetric grid with resolution $GR = 8$ is generated and the remaining bits are then simply appended to the result.

6. EXPERIMENTS

In this section, we compare different sort-based loading algorithms in a set of experiments and show the improvements of our query-adaptive technique. We first describe data files and query sets used in our experiments. Then, we present improvements achieved by our algorithms and compare the influence of order optimization and the partitioning strategies on both our and also related loading algorithms. In addition, we discuss the validity of our assumptions, which have influenced the design of our loading algorithms.

6.1 Data file and Query Profiles

In our experiments, we used an adaptation of the test framework developed for the evaluation of the RR*-tree [10]. The framework consists of 28 different data sets, either points or rectangles. They belong to eight groups *abs, bit, dia, par, ped, pha, uni, rea.* Each of the first seven groups contain three artificially generated data sets with 2,3, and 9-dimensional data following the same distribution in every dimension. Each of the artificial data sets contains at least 1 million objects from $[0, 1]^d$. For example, the group *uni* consists of 3 files of $1'000'000$ two-, three- and nine-dimensional uniformly distributed points. The eighth group consist of seven real data sets with 2,3,5,9,16,22, and 26 dimensions. For example, the 2-dimensional *rea* data set consists of $1'888'012$ bounding boxes of streets of California. A full description of data generation and sources is given in [10].

In the original test framework, three range-query sets $qr1$, $qr2$ and $qr3$ were considered for each data set. Except for the group *ped*, the query sets were generated as follows: The queries of $qr1$, $qr2$ and $qr3$ refer to square-shaped windows and deliver 1, 100 and 1000 results on average, respectively. Note that in difference to previous performance comparisons, the cardinality of the response sets is limited (at most twice the average) to avoid the dominating influence of a few queries with very large response sets. All queries followed the underlying data distributions. According to the query taxonomy [21], these query sets are of type WQM_4 (queries follow data distribution and query size is based on answer number). For group *ped*, queries were generated in a more traditional way. The square-shaped range of $qr1$, $qr2$ and $qr3$ cover $k/1'000'000$ of the entire data space, k = 1, 100, 1000. In addition, *ped* queries were uniformly distributed (type WQM_1).

In order to examine the query-adaptivity of our techniques, we modified the generation of 2-dimensional query profiles $qr2$ and $qr3$ by introducing the aspect ratio a as a new parameter. There are now $qr2_a$ and $qr3_a$, $a = 1, \ldots, 20$, where $a = 1$ refers to the original profiles. We retain the original methodology for generating query profiles $qr2_a$ and $qr3_a$ limiting the response set cardinality to 100 and 1000, respectively.

Except for *ped*, the generation process is based on posing nearest neighbor queries with the weighted distance measure $L_\infty(p1, p2) = \max(|p1x - p2x|, \frac{1}{a}|p1y - p2y|)$, $p1, p2 \in [0, 1]^2$. For *ped*, we considered range queries with query profile $(\sqrt{(k/(a \cdot 1'000'000))}, \sqrt{(a \cdot k/1'000'000)})$.

6.2 Examined Algorithms

Table 1 provides a summary all methods used. As a reference method, we used the traditional sort-based loading termed Z-loading and H-loading using Z-ordering and H-ordering, respectively. Both of the loading techniques are parameterized with storage utilization set to 80%. Note that in our experiments, higher storage utilization did not improve the query performance. ZAS-loading refers to Z-ordering combined with our adaptive shuffling technique. Z-GO stands for globally optimized partitioning technique applied to Z-ordered input, whereas H-GO is based on H-ordering. H-SO uses our partitioning with a guaranteed storage utilization of 80%.

We also examined STR [18] and TGS [14] because of their popularity. Storage utilization was again set to 80%. In addition, we also present an improved version of STR, termed STR-GO, which combines STR with our globally optimized partitioning method. STR-GO performs as STR for the first $d-1$ dimensions, but uses our partitioning technique for the last dimension. This is directly applicable because the data objects are distributed among the leaf pages regarding the d-the dimension. The performance of bulk loaded R-trees and tuple-by-tuple loaded R*-trees[9] is also compared.

All algorithms are implemented in Java. Experiments were conducted on a 64 bit Intel Core2Duo (2 x 3.33 Ghz) machine with 8 Gb memory running Windows 7. In order to illustrate the performance on several different storage devices, we conducted experiments on a magnetic disk (Seagte ST35000418As), SSD (Intel X25) and in main memory. For experiments on disk and SSD, we used 4KB pages with a capacity B = 128 and minimum occupation b = 42 for d=2. For sorting, we used 10 MB of main memory. The raw I/O device interface is used to avoid the interference with other system buffers. For our in-memory experiments, we used different settings for the page capacity that was found to be

Shortcut	Sorting Order	Partitioning
Z	symmetric Z-order	naive
H	H-order	naive
ZAS	adaptive Z-order	naive
Z-GO	symmetric Z-Order	$gopt^*(i)$
ZAS-GO	adaptive Z-order	$gopt^*(i)$
H-GO	H-order	$gopt^*(i)$
H-SO	Hilbert-Order	$opt^*(i,k)$
STR	not applicable	naive
STR-GO	not applicable	$gopt^*(i)$
TGS	not applicable	n/a

Table 1: Algorithms

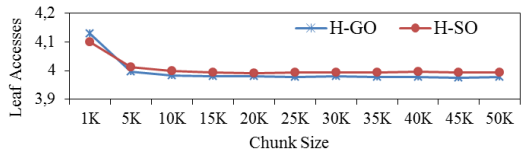

Figure 5: Query performance of partitioning algorithms for varying the chunk size

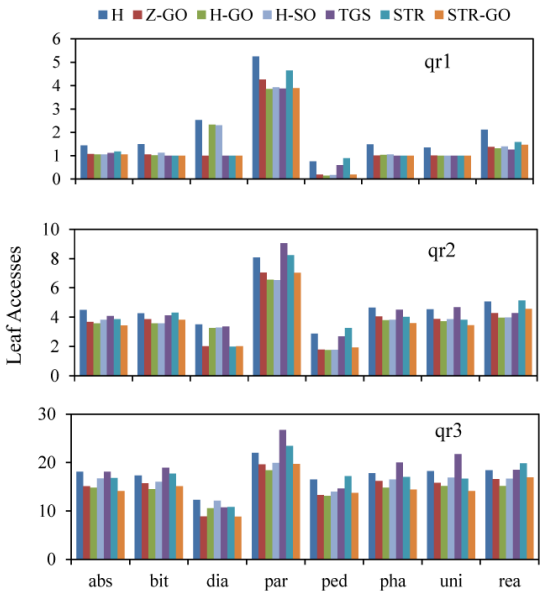

Figure 6: Avg. number of leaf accesses per query for d=2.

the overall optimum: B= 12 and b = 4.

Algorithm efficiency is measured by I/O and CPU time. We consider the number of leafs touched during query traversal as a default I/O metric, however, we do not count repeated accesses to the same leaf. As confirmed in our experiments, this is a good performance indicator, since index nodes are located in large main memories.

In Section 4.2 we introduced a simple approximation scheme for our partitioning algorithms. Rather than running the algorithm on the entire data set, we prepartition the data into equi-sized chunks and apply the algorithms to each of the chunks. Figure 5 depicts quality of the approximation as a function of chunk size for the California data set using $qr2$. We observed that a chunk size of B^2 (= 16384) is sufficient to obtain near-optimal results. Similar results are achieved for other data sets. Note that the function is not decreasing strictly monotonically because the queries do not obey the uniform assumption of the query model. This also explains that for a chunk size of 1K the SO strategy is slightly superior to GO. For the rest of the experiments, we use chunks of size B^2 for our partitioning methods.

6.3 Sorted Set Partitioning

This section discusses the improvements achieved by our partitioning strategies. We consider square-shaped queries with aspect ratio $a = 1$ only. In addition to the methods based on space-filling curve, we also report the results of TGS, STR and STR-GO. Figure 6 depicts the I/O performance for eight 2-dimensional data sets and query files $qr1$, $qr2$ and $qr3$. Note that all loading methods that use our partitioning strategies are superior to H-loading. Moreover, STR-GO performs better than its original counterpart. For TGS we observed similar effects as reported in [14]. TGS performs well for point queries $qr1$, but its performance deteriorates with an increasing query region. It is noteworthy that there is no significant difference between H-GO and Z-GO except for *dia*, where Z-GO is clearly superior. The most significant improvements over H-loading are achieved for point queries on the 2-dimensional data set *ped*. This data set is the only for which the queries are uniformly distributed. Note that this is in full agreement with the goal function used in our optimization. This also explains the large difference in performance between STR and STR-GO. We observed that the impact of the query size is marginal for storage bounded algorithms H-SO and Z-SO in comparison to the H-GO and Z-GO counterparts. Thus, minimizing the area (which is only optimal for point queries) achieves already good results for all query profiles $qr1$, $qr2$ and $qr3$.

The query size influences the relative R-tree performance. This is not surprising, as for larger regions, the storage utilization will have greater impact (than the clustering capability of the loading techniques). This is also in agreement with the analytical results obtained from the cost model. For example, R-trees generated from H-GO-loading perform small queries on the California data set (*rea*) with only 60% of the disk accesses compared to H-loaded R-tree. For queries $qr3$ with 1000 results the performance difference is only 20%. We achieved similar results for the 3-dimensional data sets. H-loading is superior to STR-GO for only some of the data files, but inferior to Z-GO and H-GO in all cases. The average normalized results for two, three and nine dimensions are reported in Table 2 (performance is expressed as the ratio of average number of leaf accesses for the specific and the H-loaded R-tree). The results indicate slight improvements for higher dimensions.

As expected, the number of leaf pages occupied by our R-trees generated from Z-GO and H-GO in relation to the number of leaves of H-loaded R-trees is higher for small queries. For larger queries, it is typically below 100%, i.e., the storage utilization is higher than 80% for the R-trees generated by Z-GO and H-GO.

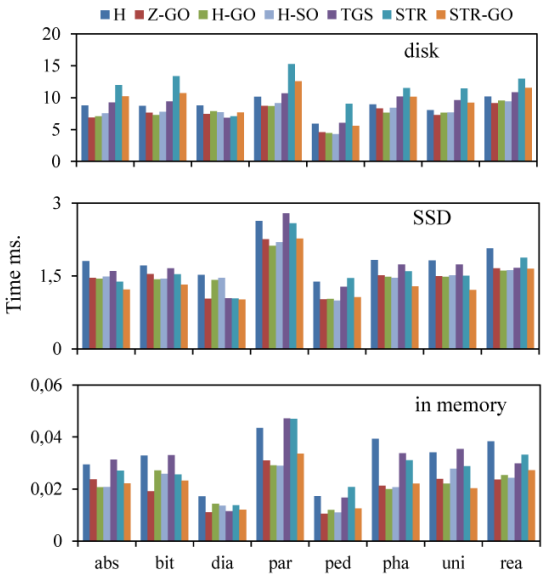

Figure 7: Avg. time per query for $qr2$ and d=2.

	d=2	d=3	d=9	d=2,3,9
Z-GO	75.5 %	71.6 %	66.45 %	71.2 %
H-GO	76.2 %	73.3 %	68.3 %	72.6 %

Table 2: Avg. query performance of Z-GO and H-GO-loaded R-trees over square-shaped queries for different dimensionalities in leaf accesses (results are normalized to H-loaded R-trees).

Further, we analyzed average query execution time for d=2 and query file qr_2. Figure 7 shows the average time per query for disk, SSD and main memory. Query time measured for a disk includes the I/O time for leaf accesses, while the index nodes are likely to reside in memory or disk cache. In particular, we observed a positive effect of sort-based loading using H-loading combined with our partitioning on the average dis access cost. The way how data is written to disk exhibits high clustering within a level, since blocks are written according to the SFC order. Therefore, there are fewer random I/Os than for TGS and STR (see Fig. 7). In order to illustrate the impact of physical clustering, we also compared the query performance with the R*-tree (see Fig. 8). As illustrated, significant improvements of up to factor of five can be achieved particularly because of the clustering when indexes are bulk-loaded. Moreover, we observed also similar effects for in memory R-trees. For SSDs, however, there are no positive effects from sequential I/O patterns. As a consequence, the average query time is highly correlated to the number of node accesses, see the plots in the mid of Figure 3 and 4.

In the following we discuss the effects of the uniformity assumptions of our cost model. Recall that except for *ped* the query distributions follow the data distribution. The question is therefore how our simplified analytical cost model is related to the real cost. In Fig. 6, the cost of our analytical model is plotted as a function of the number of leaf accesses required for processing the queries from profile $qr1$, $qr2$ and

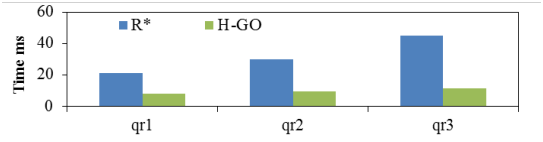

Figure 8: Avg. time per query for R*-tree and H-GO for California set (d=2).

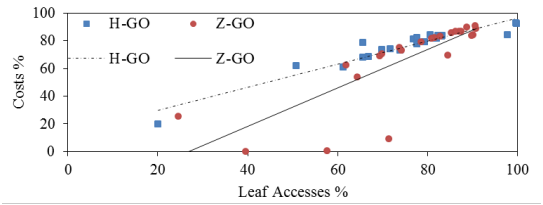

Figure 9: Correlation between the number of leaf accesses and the analytical costs

$qr3$ on the 2-dimensional data sets (both graph dimensions are normalized to H-loading). The graph shows a clear correlation between the cost measures . This supports that our cost model is indeed a good predictor for the actual cost. There are only three outliers corresponding to the extreme *dia* dataset, for which the real cost of the queries is substantially lower than the estimated costs model of our model.

Finally, the average total loading time of the algorithms for d=2 is depicted in Table 3. The cardinality of the data sets was limited to 1'000'000 rectangles. The total loading times of H/Z-loading, STR, H-GO, H-SO exhibit low standard deviation (see column std), while TGS is sensitive to the data distribution. H-GO loading time was clearly dominated by the time of external sort while the partitioning step itself has only little impact (see build time). This differs from H-SO, where the time for the partitioning step dominates sorting, also STR is more expensive as data has to be sorted twice for d = 2.

6.4 Order Optimization

In this section, we primarily discuss the benefits of adaptive shuffling for better adaptivity to the underlying query profile. For the following discussion, we consider the results obtained from R-trees generated for the 2-dimensional uniformly distributed data set and query sets $qr2_a$, $a = 1, \ldots, 20$. Fig. 10 shows the average number of leaf accesses for $qr2_a$ queries as a function of the aspect ratio a. For each setting of a, we present the performance of five loading techniques ZAS, Z-GO, ZAS-GO, H-GO and H. Note that $a = 1$ represent the case of square-shaped queries. For $a = 1$ the performance of

alg.	sort time	build time	total time	std
H	25,64	0.68	26.4	2.09
H-GO	25,64	7.40	33.12	2.03
H-SO	25,64	77.67	103.11	2.56
TGS	n/a	n/a	245.18	124.33
STR	n/a	n/a	55.47	7.76

Table 3: Avg. loading time (in sec.) of 1'000'000 2-dimensional rectangles.

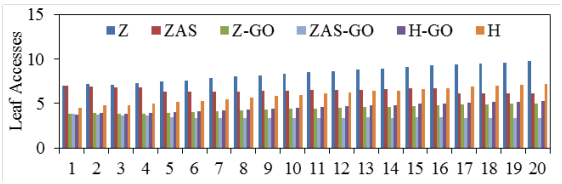

Figure 10: Query results for *uni* set (d=2)

ZAS is identical to Z-loading. In agreement with previous experiments found in the literature, H-ordering is superior to Z-ordering. However, Z-GO and ZAS-GO are superior to H-loading and only slightly inferior to H-GO. For $a = 20$, the situation has changed dramatically. The performance of H-ordering has slightly decreased to 75% of Z-ordering, while ZAS-GO is clearly the most efficient technique. It is also evident from the comparison of ZAS, Z-GO and ZAS-GO that both of our techniques contribute to the substantial improvements that are observed for ZAS-GO. Moreover, GO in combination with Z-ordering provides slightly better results than H-ordering with GO.

The average volume V of leaf box is to be known in order to design our two-part space-filling curve. Assuming a uniform data distribution we can estimate the average leaf box volume by using the ratio of B and N. For $qr2_8$, the empirically determined optimal global value GR is compared with the estimated one. Our cost model returns $GR = 12$ for uniformly distributed data in all cases, which is in agreement with half of our experimental results (*abs, pha* and *uni*). The optimal value $GR = 14$ for *bit* slightly deviates from the estimated one. For *dia*, $GR = 0$ is the best value as the data records are located on the diagonal (it is sufficient to organize the data according to one of the axes). For *par*, *ped* and *rea*, the optimal value for GR was greater than 12 because the distributions are clearly non-uniform. In particular, data sets *par* and *ped* have a very high variance of volume and perimeter. This kind of data distributions is difficult to deal with and query adaptivity often yields no improvement.

To address this issue and verify our assumption, we used histograms as an option for deriving the values for sorting parameter GR. Histograms were also used for approximating dx_i value distributions for non-uniform data and query distributions. For each bucket p_i, our sort-based two step algorithm is processed independently with local parameters GR as well as average dx_i for p_i. Data summaries are held in memory and serve as a look-up function during the sorting and partitioning steps. We used the MinSkew-Histogram[1] with 100 buckets to represent the 2-dimensional data distribution. We observed that using histograms improves substantially the performance over global estimated parameter GR for $qr2_8$ profile (e.g. by 35% for *par*, by 15% for *ped* and by 46% for *rea*), while the degree of improvement depends not only on the chosen histogram method but also on a histogram parameter settings. Therefore, we want to study more deeply the histogram and bulk-loading interaction in our future work.

7. CONCLUSIONS

In this paper, we reconsidered the problem of sort-based bulk-loading of R-trees. We demonstrate the importance of query profiles for search efficiency of generated R-trees. We designed new loading algorithms based on two innovative techniques. The first consists of a new sorting technique of rectangles based on non-symmetric Z-order curve design, while the second generates an optimal partitioning for a given sequence of rectangles. Both techniques are optimized according to a commonly used cost model for range queries. Our optimal partitioning techniques are broadly applicable and beneficial. They can be easily integrated into other loading techniques like STR, which is a popular loading method in commercial database systems. They can also be combined with standard Hilbert-loading even when the query profile is unknown. In this case, we suggest to use the partitioning that minimizes the area of the bounding boxes of the leaves.

Our experimental results obtained from a standardized test framework clearly reveal the advantages of our techniques in comparison to standard loading techniques (STR, Hilbert-loading, Z-loading, TGS). Our techniques creates R-trees with consistently better search efficiency than those created by pure Hilbert-loading, while for some data files large improvements in query performance (about factor 5) were achieved. Interestingly, due to our new partitioning methods, there is no noticeable differences anymore in the performance of R-trees build form rectangles sequences following either Hilbert-ordering or Z-ordering. Thus, we suggest using Z-ordering because of its conceptual simplicity.

In future work we aim to develop better cost models that are more accurate for non-uniform query distributions. Based on such cost models, we would be able to reorganize indexes in a proactive way to adapt them to current and future workloads.

8. ACKNOWLEDGMENTS

We would like to thank Anne Sophie Knöller for insightful feedback and reviewing this piece of work.

9. REFERENCES

[1] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD '99*, pages 13–24, New York, NY, USA, 1999. ACM.

[2] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4:9:1–9:30, March 2008.

[3] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 560–, Washington, DC, USA, 1996. IEEE Computer Society.

[5] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theor. Comput. Sci.*, 181:3–15, July 1997.

[6] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes

by an optimal pair of boxes. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 475–486, London, UK, 1992. Springer-Verlag.

[7] B. Becker, H.-W. Six, and P. Widmayer. Spatial priority search: An access technique for scaleless maps. In J. Clifford and R. King, editors, *SIGMOD '91*, pages 128–137. ACM Press, 1991.

[8] L. Becker, H. Partzsch, and J. Vahrenhold. Query responsive index structures. In *GIScience '08*, pages 1–19, 2008.

[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 *SIGMOD '90*, pages 322–331, New York, NY, USA, 1990. ACM.

[10] N. Beckmann and B. Seeger. A revised r*-tree in comparison with related index structures. In *SIGMOD '09*, pages 799–812, New York, NY, USA, 2009. ACM.

[11] J. V. d. Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB '01*, pages 461–470, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[12] J. V. d. Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB '97*, pages 406–415, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[13] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *VLDB '94*, pages 558–569, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[14] Y. J. García R, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *GIS '98*.

[15] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[16] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB '98*.

[17] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM '93*, pages 490–499, New York, NY, USA, 1993. ACM.

[18] S. Leutenegger, M. A. Lopez, and J. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.

[19] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[20] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84*, pages 181–190, New York, NY, USA, 1984. ACM.

[21] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *PODS '93*, pages 214–221, New York, NY, USA, 1993. ACM.

[22] B.-U. Pagel, H.-W. Six, and M. Winter. Window query-optimal clustering of spatial objects. In *PODS '95*, pages 86–94, New York, NY, USA, 1995. ACM.

[23] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[24] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD Conference*, pages 17–31, 1985.

[25] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[26] Y. Tao and D. Papadias. Adaptive index structures. In *VLDB '02*, pages 418–429, 2002.

[27] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS '96*, pages 161–171, New York, NY, USA, 1996. ACM.

[28] K. Yi, X. Lian, F. Li, and L. Chen. The world in a nutshell: Concise range queries. *IEEE Trans. Knowl. Data Eng.*, 23(1):139–154, 2011.