CAR-TR-718 DACA76-92-C-0009
CS-TR-3292 June 1994

R-tree Index Optimization

D.M. Gavrila

Computer Vision Laboratory
Center for Automation Research
University of Maryland
College Park, MD 20742-3275

Abstract

The optimization of spatial indexing is an increasingly important issue considering the fact that spatial databases, in such diverse areas as geographical, CAD/CAM and image applications, are growing rapidly in size and often contain on the order of millions of items or more. This necessitates the storage of the index on disk, which has the potential of slowing down the access time significantly. In this paper, we discuss ways of minimizing the disk access frequency by grouping together data items which are close to one another in the spatial domain ("packing"). The data structure which we seek to optimize here is the R-tree for a given set of data objects.

Existing methods of building an R-tree index based on space-filling curves (Peano, Hilbert) are computationally cheap, but they do not preserve spatial locality well, in particular when dealing with higher-dimensional data of non-zero extent. On the other hand, existing methods of packing based on all dimensions of the data, such as the several proposed dynamic R-tree insertion algorithms, do not take advantage of the fact that all the data objects are known beforehand. Furthermore, they are essentially serial in nature.

In this paper, we regard packing as an optimization problem and propose an iterative method of finding a close-to-optimal solution to the packing of a given set of spatial objects in D dimensions. The method achieves a high degree of parallelism by constructing the R-tree bottom-up. In experiments on data of various dimensionalities and distributions, we have found that the proposed method can significantly improve on the packing performance of the R* insertion algorithm and the Hilbert curve. It is shown that the improvements increase with the skewness of the data and, in some cases, can even amount to an order of magnitude in terms of decreased response time.

Keywords: spatial databases, R-tree, optimization

The support of the Advanced Research Projects Agency (ARPA Order No. 8459) and the U.S. Army Topographic Engineering Center under Contract DACA76-92-C-0009 is gratefully acknowledged.

1 Introduction

There has been a great deal of interest over the years in extending traditional, alpha-numeric databases to handle multi-dimensional spatial data. Applications of such spatial databases have traditionally been found in Geographical Information Systems (GIS) and Computer Aided Design (CAD) packages. In a GIS, for example, that contains maps of all the roads, lakes and rivers of a country X, there is a need to facilitate queries such as requesting all the roads within Y miles of city Z. More recently, spatial data structures have been used to aid in the retrieval of images from large image databases by shape similarity [2, 13]. In order to process these types of queries quickly, an efficient indexing mechanism for spatial data objects is required, according to their location in space.

A recent survey of spatial data structures is given in [16]. Many of the more successful approaches rely on the principle of hierarchical decomposition of space. The idea is to index progressively smaller regions of space using a B-tree-like data structure, such that search can be focused at a high level towards the relevant regions. An example of such a data structure is the R-tree and its variants [1, 6, 18], where the data space is successively decomposed into (hyper) rectangles. Another example is the Cell-tree [5], where the primitive index region is a polygon. All these data structures are suited for handling spatial data dynamically. In addition to retrieval, they allow runtime insertion and deletion of objects in the database.

If some of the current databases are considered to be large, future databases are expected to be huge. For example, the U.S. Bureau of the Census has been building the TIGER database to store a detailed map of the country; its size is currently approximately 19 Gb. In the near future, NASA's Earth Observation System database is expected to include more than 10^{10} Mb of image data. The volume of such databases containing millions of data objects necessitates the storage of the index structure on disk. This has the potential of slowing down the access time considerably. In this paper, we discuss ways of minimizing the disk access frequency by grouping together data objects which are spatially close to one another ("packing"). The data structure we seek to optimize is the R-tree for the case of a given set of data objects. Packing can thus be done to build the final index on a static database, or to periodically reconfigure (part of) the index on a dynamic database whenever time constraints allow.

Our approach is to view packing as an optimization problem and use an iterative method to find a close-to-optimal solution. The iterative method used is based on the minimization of an objective

function, which captures what efficient packing means. The same type of objective function can be used to pack data structures whose primitive index region is not a (hyper) rectangle, by defining an appropriate distance metric. We pack each level of the R-tree, and construct the R-tree bottom-up, rather than starting at the root and inserting data items one by one in a top-down fashion. By doing so, we obtain an approach which is well suited for parallel implementation.

Unlike previous work on packing methods based on space-filling curves [10, 15], in our approach clustering takes place in D-dimensional space. It takes into account both the position and the spatial extent of the data in all D dimensions. We therefore obtain superior packing performance which becomes more evident with increasing dimensionality and skewness of the data, as we shall see. On the other hand, compared to the existing D-dimensional dynamic insertion algorithms of the R-tree [1, 6, 18], our approach takes advantage of the fact that all the data is known beforehand in performing the packing.

This paper is organized as follows. Section 2 gives an overview of previous work on the R-tree and previous approaches to constructing the index tree. In Section 3 we present our approach to the packing problem; this is followed by a discussion in Section 4. We have performed a number of experiments on data of different dimensionalities and distributions in order to compare the performance of our approach with some of the most promising existing methods of constructing the R-tree, the Hilbert curve and the R* insertion algorithm. We show in simulations that considerable improvement is possible in terms of decreased response time, if we optimize the packing as proposed. The experiments are described in Section 5, after which we conclude in Section 6.

2 Indexing on the R-tree

2.1 The R-tree

The R-tree [6] is a height-balanced tree similar to the B-tree. In the R-tree, leaf nodes contain index records of the form *(I, tuple-id)* where *tuple-id* uniquely determines a tuple in the database and I determines a bounding (hyper) rectangle of the indexed spatial object. The actual data objects can have arbitrary shapes. Non-leaf nodes contain entries of the form *(I, child-pointer)* where *child-pointer* refers to the address of a lower node in the R-tree and I is the smallest bounding rectangle that contains the bounding rectangles of all of its children nodes. If (m, M) is the degree of the R-tree, each node contains between $m \leq M/2$ and M entries (the "node fill requirements"), with the possible exception of the root. See Fig. 1a for an example of a $(1, 3)$ R-tree. Fig. 1b shows

a different $(1,3)$ R-tree on the same data set. The R-tree is well suited for storage on secondary memory (i.e. disk). Each node of the R-tree is placed on a separate disk page. This makes the R-tree particularly useful for applications involving very large data bases where the index is too large to fit in main memory.

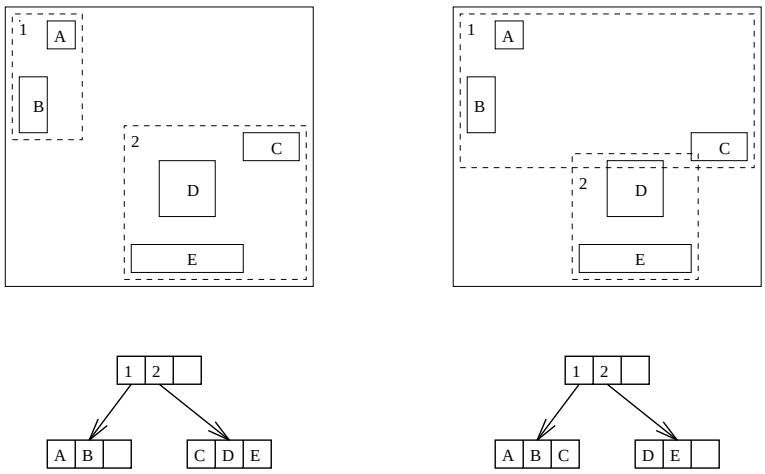

Figure 1: a) and b) Two different R-trees on the same data set

A search in an R-tree starts at the root and descends the tree in a manner similar to a search in a B-tree. Due to the non-zero size of the query window, and due to possible overlap between bounding rectangles at each level of the tree, multiple paths from the root downwards may need to be traversed. The R-tree can be updated dynamically, by insertion or deletion of data objects at the leaves. We will review the various existing R-tree construction methods in Section 2.3. For further details on the various search, insertion and deletion algorithms see [1, 6, 18]. See also [17, 19] for some recent extensions of the R-tree to deal with spatio-temporal data.

2.2 Response Time

Given that there are multiple possible R-tree configurations that can index the same database of objects, one can introduce a notion of efficiency of a configuration. In our case, we seek to minimize the response time, which is the time elapsed from the moment a user enters a query until the response is completed. We assume that due to memory constraints, at least part of the R-tree is disk-resident. Since the I/O time dominates the CPU time in determining the response time of a typical query, we will take only the former into consideration. Given that each node of the R-tree is stored on a separate page, the response time will then correspond to the number of R-nodes on disk visited during a query. Here we assume that we have stored the R-tree on a single-disk system.

The response time model can be extended to deal with the storage of the index tree on parallel disk arrays by assuming unit cost for accessing P pages located on P different disks. The discussion below is then modified accordingly (see also [9]).

Using the single-disk model for the response time of an R-tree configuration, it is possible to derive its expected response time $E[T_r]$ for some simple query distributions. For example, let S_j be the extent of the data space in the j^{th} dimension, let q_j be the extent of a window query in that dimension, and let r_j^i be the extent of the i^{th} non-leaf ("directory") rectangle of the R-tree on disk in the j^{th} dimension. For the case of a query distribution with fixed windows of above sizes and uniformly distributed centers, the expected response time of an R-tree configuration is determined by [10, 14]

$$E[T_r] \;\propto\; (\sum_{i=1}^{N} \prod_{j=1}^{D} (r_j^i + q_j)) \;/\; \prod_{j=1}^{D} S_j \qquad (1)$$

The above expression computes the fraction of the volume indexed in the data space, if every i^{th} (disk resident) directory rectangle is enlarged by q_j in the j^{th} dimension. It is an approximation for the case $q_j << S_j$, in which case boundary effects are negligible. If this is not the case, the term $r_j^i + q_j$ should be restricted by the boundary of the data space [14]. For point queries, the expected response time is determined by the fraction of volume indexed. As mentioned in [10], the above formulas are independent of the details of the R-tree creation/insertion/update algorithms and hold for R-trees, R* trees, R+ trees, etc. They are also independent of the data distribution. They provide a useful way to compare two R-tree configurations without the need to run actual queries and compare performance. For example, the R-tree configuration of Fig. 1a is rated more favorably than the one in Fig. 1b by eq. (1).

In reality the query distribution might well be non-uniform. The query distribution might depend on the distribution of the stored data, with higher likelihood of access in areas with high data densities. In that case, one could envisage using a modified version of eq. (1) with weights for the volumes of the (directory) rectangles depending on the expected access frequency of that region (see also [14]). The following criteria affect the expected response time of a R-tree configuration in the 2D case [1]:

1. The areas of the directory rectangles
2. The perimeters of the directory rectangles
3. The overlap between directory rectangles
4. The storage utilization

In D dimensions, area is replaced by volume and the perimeter of a (hyper) rectangle can be defined either as the sum of its extents in the different dimensions, or as the sum of the volumes of the sides of the (hyper) rectangle. Lower-volume directory rectangles are desirable because this means less "dead" space (space which is indexed but does not contain data). In an R-tree configuration with less dead space, queries which do not index data are likely to be discontinued higher in the R-tree, reducing the number of disk accesses. Minimizing the second and third criteria while maximizing the fourth criterion is particularly useful for reducing the disk accesses when dealing with window queries. Note that in some cases the above optimization criteria can be contradictory [1]. An increase in storage utilization could for example result in an increase of the area indexed. Methods of constructing R-trees are based on heuristics to achieve these conflicting goals; we discuss them in the next subsection.

2.3 R-tree construction

Methods of building an R-tree on a given data set can be characterized by the following features:

Insertion Mode

Incremental vs. batch. In the incremental insertion mode, data objects are inserted one by one in the current tree. The main procedure is to follow a path from the root to one of the leaves and perform the actual insertion. After each insertion, a valid R-tree exists. The batch insertion approach takes a given data set and builds the tree bottom-up, starting at the leaf level and proceeding towards the root. Only at the end does a valid R-tree exist.

Dimensionality

D-dimensional vs. linear. In D-dimensional packing methods the grouping of data and directory rectangles in the R-tree is done by grouping in D-dimensional space rather than in one-dimensional (linear) space.

The above characterization is useful for two reasons. First, it allows the comparison of existing methods of constructing R-trees, and second, it suggests a novel approach to the problem, as we will show shortly. We start with the first issue and give an overview of the existing methods of building an R-tree on a given data set.

The existing dynamical insertion algorithms, the linear/quadratic R tree [6], the R+ tree [18] and the R* tree [1], can be characterized as incremental and D-dimensional. The insertion process is

similar to that of a dynamical B-tree. The data items are inserted one by one in the tree by starting at the root and following a path down to a leaf node at which point insertion is attempted. The path is determined by choosing for each node in the path the child node which would require the least volume enlargement to accommodate the new item (or some other D-dimensional criterion). If insertion is attempted at a leaf node which is not full, the new data item is added and the insertion is completed. If the leaf node is full, then some action needs to be taken to maintain the node fill requirements of the R-tree nodes. This can be handled by a *node split* in which the contents of the overfilled leaf node and the new data item are distributed over two new nodes. A node split can propagate upwards and result in the root being split, at which point the R-tree increases in level. The method used to deal with a node-split is one of the main distinguishing factors between the dynamical variants of the R-tree; see [1, 6, 18] for the details. The R* insertion algorithm [1] provides an additional mechanism to deal with an overfilled node. This involves deleting some of the entries and re-inserting them into the R-tree. The problem with a dynamic index structure like the R-tree is that the early-inserted data rectangles (most likely) will have introduced directory rectangles which no longer efficiently represent the current data, in terms of measures like eq. (1). Node splits are merely local reorganizations of the R-tree and thus not likely to be of sufficient help to deal with this problem. Re-insertion of data items gives the R-tree a basic global reorganization capability [1].

Other existing methods [10, 15] of building an R-tree can be characterized as batch-oriented and linear. They are linear since they are based on the clustering properties of space-filling curves. Space-filling curves [8] are mappings from D to 1 dimensions with the desirable property that, in general, they map points which are close together in D-dimensional space into points that also close together in one-dimensional space. In other words, they tend to preserve spatial locality under dimensionality reduction. See Fig. 2 for some of the more popular space filling curves from 2D to 1D. The same curves can be generalized to map from higher dimensions to 1D. The methods are batch oriented since they build the R-tree bottom up, level by level. At each level a space-filling curve is used to sort the rectangles using their centers or corners. The sorted list is then looped through and successive rectangles are assigned to the same R-tree node until a certain fraction of the node is filled (usually 100%, full packing). Then the next R-tree node of a level is filled using the bounding rectangles of the previous level.

It should be mentioned that packing methods based on the Dimension Sort curve (Fig. 2), as proposed by [15], do not seem to be very desirable. They perform rather poorly in practice

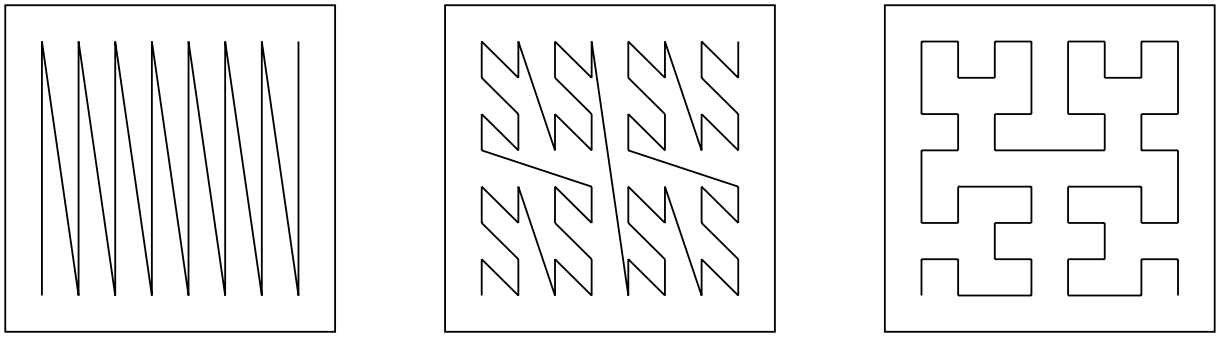

Figure 2: Space Filling Curves

because only one side of the rectangles determines the grouping. If the data are skewed they can cover considerable dead space, a problem which deteriorates for non-point data. Furthermore, they produce long thin bounding rectangles along one dimension. A range query is thus likely to intersect many of these directory rectangles, resulting in an excess of disc accesses [10]. An improvement is described in [10], where a space filling Hilbert curve is used to sort the rectangles according to the Hilbert-coordinates of their centers. It was reported in earlier work [4, 8] that the Hilbert curve compares favorably to the Gray-code and Peano space-filling curve regarding clustering performance.

Recently, it has been proposed to use the Hilbert curve in the dynamical insertion algorithm for the R-tree [11]. The algorithm becomes similar to inserting in a B-tree. This approach can be characterized as incremental and linear.

Table 1: Characterization of R-tree construction methods

	incremental methods	**batch-oriented methods**
D-dimensional Methods	Quadratic/linear R [6], R+ [18], R* [1]	*Iterative optimized R-tree*
Linear Methods	Hilbert R-tree [11]	R-tree by space-filling curves [10, 15] Parallel R-tree [7]

R-tree construction methods are compared in Table 1. The main use of incremental methods for the R-tree is to handle dynamical data. If we have a known data set to be indexed, batch

methods are preferable for two reasons. First, the incremental methods are dependent on the order in which the data objects are inserted. As mentioned earlier, previously inserted data objects are likely to lead to R-tree directory rectangles which are inadequate to represent the whole data set. In contrast, batch methods can take advantage of the fact that we know the entire data set in advance. The second reason to prefer batch methods is that they lead to massive parallelism. In a bottom-up approach different processors can be assigned to different portions of the data space to group the data objects located there. See for example [7] for a parallel algorithm to build an R-tree bottom up, based on linear sorting of rectangles across several axes. On the other hand, incremental methods are serial in nature, inserting data objects one by one.

Linear packing methods have the advantage that they are fast. However, they have the drawback that they try to map D-dimensional rectangles onto 1-D (i.e. a mapping from 2D parameters to one parameter), not taking into account the positions and the spatial extents of the rectangles in all dimensions. The discarding of parameters negatively affects the linear packing methods for two-dimensional data. Their performance decreases with increasing dimensionality of the data, when compared to D-dimensional packing methods. This will be illustrated in the experiments.

The above considerations lead us to propose a D-dimensional and batch-oriented packing method for the R-tree (see Table 1). The motivation is that in many applications of interest, such as image databases [2, 13], the data is multi-dimensional and fairly static. For example, [13] uses 26-dimensional feature vectors to encode various shape, texture and color properties of image objects for indexing purposes. We are willing to employ a more computationally intensive technique to obtain an efficient R-tree index for such databases, because the cost is incurred only once (or few times for near-static databases) and is offset by the repeated gain in the indexing performance on-line.

3 Proposed Approach

We describe here a packing method for the R-tree which is D-dimensional and batch-oriented. We build the R-tree bottom up, starting from the leaf level and proceeding towards the root, while applying the algorithm on each level. The method basically performs a K-way clustering of D-dimensional rectangles with constraints on constituent cluster sizes (i.e. the R tree node-fill requirements). For a parallel disk system, after the R-tree is built, sibling nodes can be assigned to different disk units to optimize the response time (see also [9]). The input to our packing algorithm

at level i of the R-tree is:

- a set of N "data" rectangles $r_1, \ldots, r_N$, the bounding rectangles of the rectangles of level $i+1$, or of the data objects if i is the leaf level
- the min node size m, the max node size M
- the number of sets K, the size of the partition

The output of our packing algorithm at level i of the R-tree is:

- a set of K "bounding" rectangles $R_1, \ldots, R_K$, each rectangle spatially containing between m and M of the rectangles $r_1, \ldots, r_N$

M is a parameter which is determined by the page size, by dividing the page size in bytes by the size of one rectangle record in bytes. K is determined by the average fill fraction $\overline{f}$ of the R-tree nodes we want to achieve at level i. Given

$$\frac{m}{M} \leq \overline{f} \leq 1 \tag{2}$$

we have

$$K = \left\lceil \frac{N}{\overline{f}M} \right\rceil \tag{3}$$

The K-way clustering of the data rectangles is achieved by iterative optimization. Starting with an initial partition, data rectangles are moved back and forth between neighboring sets while minimizing a certain objective function E. The objective function E incorporates a measure of what efficient packing means. It can be tailored to specific data and query distributions, along the lines discussed in Section 2.2. The optimization criterion we choose to minimize in this paper is the sum of the volumes of the directory rectangles (eq. (1)). This leads to efficient R-trees for all practical purposes. The objective function we use is

$$E = \sum_{k=1}^{K} \left(D(R_k, R_k) + \frac{1}{\binom{|S_k|+1}{2}} \sum_{r_i r_j \in S_k, i \leq j} D(r_i, r_j) + penalty(S_k) \right) \tag{4}$$

where $D(p,q)$ is a distance measure between two rectangles defined here as the volume of their bounding rectangle. The first term of the objective function is the optimization criterion. The second term depends on the spread of the data rectangles within a set. It denotes the average distance between two (not necessarily distinct) rectangles within a set. Sets in which the data

rectangles are clustered are deemed more favorable (lower E-value) than sets in which the data rectangles are further apart. We have added the second term to provide more guidance during the search process. It has been observed to be quite essential for achieving convergence of the algorithm to a good solution. Omitting it would, for example, result in data rectangles being shifted back and forth aimlessly between fully enclosing sets. Finally, the last term represents a penalty term, which encodes the constraints of the problem into the algorithm. In our case, the penalty term enforces the node fill requirements of the R-tree:

$$penalty(S_i) = \begin{cases} w \times (m - |S_i|)^v & \text{if } |S_i| < m \\ 0 & \text{if } m \leq |S_i| \leq M \\ w \times (|S_i| - M)^v & \text{if } |S_i| > M \end{cases}$$

where v and w are parameters to be set. We have given these parameters high values in our experiments, so in practice there will be no moves into an "illegal" state of the system (i.e. a state where one or more R-tree nodes would violate the node fill requirements). There might be some merit in giving these parameters low values initially and gradually increasing their importance in order to allow the system to converge faster.

Note that eq. (4) takes into account both the locations and spatial extents of the data rectangles along the axes. It can also be used for index trees other than the R-tree. For example, for the Cell-Tree, $D(p, q)$ can be redefined to be a measure of distance between polygons. A serial version of the proposed algorithm can be formulated as follows:

$t = 0$;
$STATE_0$ = assign the N rectangles to K sets according to some measure;
$E = E(STATE_0)$;
while (**continue**(t)) do {
    (S_i, r_k, S_j) = **generate-move**($STATE_t$);
    ΔE = **compute-delta-E** (S_i, r_k, S_j)
    if **accept** (ΔE) {
      $STATE_{t+1}$ = state after moving r_k
                  from S_i to S_j in $STATE_t$
      $E = E + \Delta E$ }
    $t = t + 1$
}

where **continue**, **generate-move**, **compute-delta-E**, **accept** are functions that need to be defined. The above description formulates the packing problem as a search through the space of possible partitions. The function **generate-move** comes up with possible moves in the current state. It would be a bad strategy to consider all possible moves from the current state; their number is $N \times K$ and the vast majority are not likely to decrease the objective function. Instead, the function **generate-move** selects only moves between a pair of neighboring sets, the latter defined by heuristics. Once such a set pair is chosen, only a candidate rectangle remains to be specified for a candidate move. A possible implementation of **generate-move** is described in Section 5.

The function **compute-delta-E** computes the increase of the objective function (eq. (4)) for a candidate move (S_i, r_k, S_j). This is a potential $O(M^2)$ computation, given the second term in eq. (4). By storing

$$DS_k = \sum_{r_i r_j \in S_k, i \leq j} D(r_i, r_j) \tag{5}$$

for each set S_k, the computation of **compute-delta-E** is linear in M. The reason to distinguish between **generate-move** and **compute-delta-E** is that the former could use less expensive computations to suggest candidate moves on which **compute-delta-E** would be invoked.

accept is a boolean function which accepts or discards the generated moves. If the only moves which are accepted are those which lower the objective function we have a "hill-climbing" type of search algorithm (although the term "valley-descending" would be more appropriate here to describe the minimization process). A more general approach can be based on *simulated annealing* [12], where the acceptance of moves is stochastic in order to escape local minima of the objective function. For our experiments, we chose the hill-climbing variant because of speed considerations.

As for choosing the initial state of the iterative process, note that it would be inefficient to start with a random assignment of rectangles to sets. A fair initial state can be provided quickly by a space-filling curve. After the rectangles are sorted on a linear scale, they can be partitioned into K equal groups; this represents the initial state.

Finally, we need to specify the stopping condition **continue**. Reasonable conditions are reaching a maximal iteration number, or insufficient progress in terms of decrease in the objective function between successive checkpoints.

4 Discussion

In this section we would like to discuss two issues regarding the algorithm proposed in the previous section: *scalability* and *parallelism*. To start with the former, note that the move generation is the sole part of the algorithm which is sensitive to the size of the data set. This is because there we have to compute pairs of sets which are "neighbors" in order to ensure that the candidate moves which are generated have a good chance of succeeding. Given a definition of neighboring sets such as the one used in the next section, one can assume that the average number of neighbors of a set remains constant (or increases at most sub-linearly) with increasing size of the data set, given a reasonable initial packing. Thus the problem is to find these pairs, whose number is linear (or at most sub-quadratic) in N, efficiently. An exhaustive approach involves $O(K^2)$ operations, a figure which becomes prohibitive for large values of N (for example $N > 10^6$, $M = 50$, and $f = 0.75$, gives $K > 2.6 \times 10^4$). There are two possible approaches to dealing with this problem.

The first approach is to bypass the scalability issue altogether by partitioning a large data set into groups of such size that even an exhaustive approach can be used to find neighboring sets. The optimization algorithm is then run on each sub-group separately. The initial partitioning can be done using a space-filling curve or some other simple clustering method. It can be argued that such a divide-and-conquer approach will produce a packing that is not much worse (in terms of measures like eq. (1)) than the one resulting from running the optimization on the entire data set. This is because if the subgroups are of substantial size (say $S > 10^5$) and clustered, the "border" effects due to the partitioning of the data set (i.e. a rectangle is assigned to a different sub-group than its neighboring rectangles) will be small and not affect the overall packing performance in a significant way.

The second approach is to find pairs of neighboring sets in sub-quadratic time. There is no requirement here to find *all* neighbors of a certain set; just a few suffice to enable **generate-move** to generate reasonable candidate moves. One way of doing this is to represent the sets by the centers of the corresponding bounding rectangle and to use space-filling curves to sort these linearly. The neighbors of a set can then be searched in a fixed range on this linear scale. Other ways of restricting the search for neighboring sets based on simple bucketing methods can be used as well.

Note on the second issue that the above algorithm contains a high degree of parallelism. The load of computing the N mappings of the space-filling curve for the initial state can be distributed evenly over P available processors. For the iterative phase, each processor can pick a neighboring

set pair and consider moves between them. The only restriction is that the set pairs are disjoint, i.e. if processor p_k picks set pair (S_i, S_j), no other processor can consider moves involving S_i or S_j. For the case $P \ll K/2$ this will approach linear speed-up, at least in theory.

5 Implementation and Experiments

We have performed various experiments to compare the performance of the Dimension Sort curve, the Hilbert curve, the R* algorithm and packing by iterative optimization, in building an efficient R-tree. Unlike previous papers, we have included experiments with higher dimensional data to assess the effects of dimension on the different methods. We used simulated data in order to obtain a variety of distributions with predictable properties. In every experiment we took a data file, built an R-tree using one of the methods and ran queries against it, measuring the number of disk accesses it required. We briefly discuss the parameters we chose for the different methods. In order to provide a fair comparison between the different methods, we ran the R* algorithm first and measured the number of nodes K at a level of the resulting tree, and used the same K for the batch-oriented methods in order to compare them at equal storage utilization.

We chose for the R* algorithm the parameters which gave the best overall performance according to [1]: $m/M = 0.4$, and fraction of entries reinserted at node overflow $p = 0.3$. In particular, for all packing methods we set $m = 20$ and $M = 50$. These parameters roughly correspond to page sizes in the range of 1K–8K, depending on the dimensionality of the data used in the experiments (2–10). The actual page sizes did not affect the relative performance of the different packing methods so we chose to fix the node fill requirements instead.

The multi-dimensional Hilbert curve was implemented using the algorithm described in [3]. The Hilbert curve was based on a discretization of space of size 128 along each dimension, i.e. the first power of 2 larger than the actual extent of the data space along each dimension, which was 100.

For the iterative optimization method, the initial state was computed by the Hilbert curve. During an iteration **generate-move** periodically computes a set of candidate moves in the current state and selects from that set at subsequent iterations until it has used up all moves, at which point it recomputes the set. This approach can be characterized as lazy, i.e. it tries to exert little effort in choosing good moves as long as **accept** has an accept ratio above a certain threshold. The following heuristics were used. At the beginning of the iteration process **generate-move** considers only moving rectangle r_k from S_i to S_j if condition 1 (see below) is met. If the accept ratio falls

under 5% **generate-move** considers the more computationally expensive condition 2 to generate moves. In both cases, the first sub-condition represents the definition of neighboring sets. Only for sets which met this condition was the second sub-condition tested.

condition 1:

- The bounding rectangles of S_i and S_j intersect,
- Rectangle r_k of S_i intersects the bounding rectangle of S_j.

condition 2:

- The bounding rectangles of S_i and S_j are "close" in the following sense. Let $dV_1 < 0$ be the volume decrease of the bounding rectangle of S_i resulting from shrinking its length along each dimension by factor $s = 0.2$. Let dV_2 be the minimum volume increase of the bounding rectangle of S_j required to intersect the bounding rectangle of S_i. The bounding rectangles of S_i and S_j are "close" if $dV = dV_1 + dV_2 \leq 0$.
- Moving rectangle r_k of S_i to S_j results in a decrease of the objective function (4)

The neighboring sets were computed using an exhaustive search; a maximum of ten neighboring sets were considered for each set. The iteration process was stopped when the optimization criterion (volume indexed) decreased by less than 2% between successive computations of the candidate move set by **generate-move**.

The data files each contained 50000 rectangles, with extents uniformly distributed in the range 1–5. The following data and query distributions were used:

(D1) "Uniform" The centers of the data rectangles follow a D-dimensional independent uniform distribution.

(D2) "Cluster" The centers of the data rectangles are grouped in 500 clusters of 100 objects each. Each cluster has an extent of 20 in each dimension with the cluster centers uniformly and independently distributed. The data rectangles are uniformly and independently distributed in each cluster.

(D3) "Mixed" 75% of the data rectangles are distributed "cluster" and 25% are distributed "uniform".

(Q1) "Uniform Fixed Window" The centers of the query rectangles follow a D-dimensional independent uniform distribution. The extent of the query rectangles is 20 in each dimension.

(Q2) "Data Centered Fixed Window" The centers of the query rectangles are set equal to those of the data rectangles. The extent of the query rectangles is 20 in each dimension.

Given the size of the data set and of the R-tree nodes, the index tree has three layers. In our experiments, we assume that the first two levels are in main memory. For each data file and packing method we also included the fraction of the volume indexed by the various methods under "uniform point queries". The results are given in the Appendix. The following observations can be made based on these figures:

- The Dimension Sort curve gives by far the worst performance in all cases. It results in between 15% and an order of magnitude more disk accesses than the Hilbert curve, with increasing data dimensionality and skewness.

- The R* algorithm and the Hilbert curve have similar performance on the data set. They differ by less than 40% in disk accesses. The R* algorithm is slightly better at the "uniform" distribution, while the Hilbert curve has an advantage at the "cluster" distribution. Given the lower computational cost of packing with the Hilbert curve and the potential for parallelism, it should be considered as the preferable method of obtaining a fast and reasonable packing of the R-tree for a very large, low-dimensional data set of small extent.

- Packing by iterative optimization can considerably improve on the Hilbert curve performance (and thus also on the R* algorithm). As would be expected, improvement increases with increase in data dimensionality and skewness. For the "uniform" data distribution improvements are up to 37% compared to the Hilbert curve. But for the "cluster" distribution, improvements can be as high as two orders of magnitude; see Table 8 for $D = 10$. The reason that the Hilbert curve and R* algorithm perform so poorly in this case is that they assign rectangles from different clusters to the same R-tree node. Since in the $D = 10$ case the inter-cluster distance is very large compared to the intra-cluster distance, this results in very large directory rectangles indexing mostly empty space. Optimization by the objective function of eq. (4) can improve on this, because the second term of eq. (4) will result in the transfer of the few rectangles which are not part of the same cluster in the initial partitioning by the Hilbert curve. The effects of indexing empty space are less serious when the queries are data-centered, although even in that case improvement up to a factor of four is shown possible (Table 10). For the "mixed" data distribution, the improvements lie between the figures for

the "uniform" and "cluster" data distribution and are up to a factor of three compared with the Hilbert curve and the R* insertion algorithm. There is no reason to believe that the above mentioned improvement gains will deteriorate sharply under a few update operations once the initial R-tree index has been built. The conditions under which the optimized R-tree was built were the same as the ones for the R* tree (same average space utilization). A better initial packing is likely to be beneficial to the performance of various dynamical insertion and deletion algorithms.

We ran our simulations on a SPARC 10 workstation. The CPU time involved in the various methods is as follows. Depending on the data dimensionality, it took about 1–5 minutes of CPU time to construct the leaf level of the R-tree with the Hilbert curve for the 50000 data objects. The current implementation of the proposed iterative method required from 15 minutes to 3 hours of CPU time, depending on the improvement achieved during an iteration (10% – 10000%). The stopping conditions used were also important factors; for example, a run resulting in 10000% improvement in three hours would reach 1000% improvement in less than 30 minutes. Some speed-up could be achieved by optimizing the move generation part of the algorithm, but the most gain is likely to come from parallel implementation. Finally, the R* algorithm required about 3–25 minutes of CPU time for the different dimensions of the data to build the index tree.

6 Conclusions

In this paper we have considered the problem of constructing an efficient R-tree index on a given set of spatial data objects. The main idea has been to group together the rectangles in the R-tree which are close to each other in the spatial domain. For range queries, this will minimize the disk access frequency and the resulting response time of indexing. Methods of constructing an R-tree have been characterized by the method of insertion (incremental vs. batch oriented) and by the dimensionality of the space in which they perform the grouping (D-dimensional vs. linear). We proposed a new class of R-tree construction methods which are batch oriented and D-dimensional based on the iterative minimization of an objective function, which captures the goodness of groupings of spatial objects. Although this leads to a more computationally intensive approach, the cost is off-line and is incurred only once for static data bases or a few times for near-static databases. Thus the cost can be offset by the repeated gain in indexing performance on-line. We have run experiments comparing the performances of the Dimension Sort curve, the

Hilbert curve and the R* insertion algorithm with the proposed method, and it has been shown that improvements can be as high as an order of magnitude on relatively low-dimensional (2–10) skewed data.

Acknowledgements

The author thanks C. Faloutsos for useful feedback in discussions on this topic. Furthermore, the author is grateful to L.S. Davis for his continued support.

References

[1] N. Beckman and H.P. Kriegel: "The R* tree: An efficient and robust access method for points and rectangles", *Proc. ACM SIGMOD*, pp. 322–331, 1990.

[2] E. Binaghi et al.: "Indexing and fuzzy logic-based retrieval of color images", *Visual Database Systems II*, E. Knuth and L.M. Wegner (eds.), North-Holland, Amsterdam, 1992.

[3] A.R. Butz: "Alternative algorithm for Hilbert's space-filling curve", *IEEE Transactions on Computers*, C-20, pp. 424–426, 1971.

[4] C. Faloutsos and S. Roseman: "Fractals for secondary key retrieval", *Proc. ACM PODS*, pp. 247–252, 1989.

[5] O. Gunther: "The cell tree: An index for geometric data", Memorandum UCB/ERL, M86/89, University of California, Berkeley, 1986.

[6] A. Guttman: "R-trees: A dynamic index structure for spatial searching", *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[7] E.G. Hoel and H. Samet: "Data-parallel R-tree algorithms", *Proc. 23rd Int'l. Conf. on Parallel Processing*, pp. 49–53, 1993.

[8] H.V. Jagadish: "Linear clustering of objects with multiple attributes" *Proc. ACM SIGMOD*, 1990.

[9] I. Kamel and C. Faloutsos: "Parallel R-trees", CS-TR-2820, University of Maryland, College Park, 1992.

[10] I. Kamel and C. Faloutsos: "On packing R-trees", *Proc. 2nd Int'l. Conf. on Information and Knowledge Management*, pp. 490–499, 1993.

[11] I. Kamel and C. Faloutsos: "Hilbert R-tree: An improved R-tree using fractals", *Proc. VLDB Conf.*, 1994.

[12] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi: "Optimization by Simulated Annealing", *Science*, vol. 220, pp. 671–680, 1993.

[13] W. Niblack et al.: "The QBIC Project: Querying images by content using color, texture and shape, " *SPIE Int'l. Symp. on Electronic Imaging Science and Technology, Conf. 1908, Storage and Retrieval for Image and Video Databases*, 1993.

[14] B.U. Pagel et al.: "Towards an analysis of range query performance in spatial data structures", *Proc. ACM PODS*, pp. 214–221, 1993.

[15] N. Roussopoulos and D. Leifker: "Direct spatial search on pictorial databases using packed R-trees", *Proc. ACM SIGMOD*, 1985.

[16] H. Samet: "The design and analysis of spatial data structures", Addison-Wesley, Reading, MA, 1990.

[17] R. Schneider and H.P. Kriegel: "Indexing the spatiotemporal monitoring of a polygonal object", *Proc. 5th Int'l. Symp. on Spatial Data Handling*, 1992.

[18] T.Sellis, N. Roussopoulos and C. Faloutsos: "The R+ tree: A dynamic index for multi-dimensional objects", *Proc. 13th VLDB Conf.*, 1987.

[19] X. Xu et al.: "RT-tree: an improved R-tree index structure for spatio-temporal databases", *Proc. 4th Int'l. Symp. on Spatial Data Handling*, 1990.

Appendix: Tables

Table 2: Avg. disk accesses per query: uniform data, uniform point queries

	D2	D4	D6	D8	D10
dimsort	12.7	12.0	11.4	10.7	9.89
hilbert	6.83	2.51	2.49	2.63	1.98
rstar	6.04	2.40	1.99	1.61	1.22
iteropt	6.47	1.81	1.72	1.65	1.42

Table 3: Avg. disk accesses per query: uniform data, uniform window query

	D2	D4	D6	D8	D10
dimsort	123	121	121	118	120
hilbert	98.4	29.4	21.2	19.3	15.3
rstar	94.8	29.6	19.9	15.4	12.1
iteropt	97.0	25.2	17.4	14.7	12.4

Table 4: Avg. disk accesses per query: uniform data, data centered window query

	D2	D4	D6	D8	D10
dimsort	124	121	121	112	121
hilbert	98.9	29.6	21.6	19.2	15.0
rstar	95.2	29.6	19.8	15.4	12.1
iteropt	97.6	21.2	17.7	14.7	12.2

Table 5: Avg. disk accesses per query: mixed data, uniform point queries

	D2	D4	D6	D8	D10
dimsort	11.6	9.95	8.33	7.04	6.39
hilbert	5.92	2.43	2.40	1.89	1.28
rstar	4.94	1.91	1.61	1.32	1.06
iteropt	5.47	1.68	1.21	0.667	0.406

Table 6: Avg. disk accesses per query: mixed data, uniform window query

	D2	D4	D6	D8	D10
dimsort	125	121	115	112	109
hilbert	102	30.2	19.3	14.8	11.1
rstar	97.6	28.6	19.4	15.6	14.0
iteropt	100	25.4	14.1	9.52	7.08

Table 7: Avg. disk accesses per query: mixed data, data centered window query

	D2	D4	D6	D8	D10
dimsort	146	145	140	139	140
hilbert	121	42.0	28.8	24.4	19.8
rstar	116	41.6	30.8	26.0	25.6
iteropt	119	30.0	21.8	16.8	14.4

Table 8: Avg. disk accesses per query: clustered data, uniform point queries

	D2	D4	D6	D8	D10
dimsort	9.09	6.06	3.89	2.83	2.04
hilbert	4.97	1.92	0.881	0.367	0.157
rstar	4.25	1.49	0.649	0.337	0.161
iteropt	4.62	1.11	0.103	0.00661	0.00152

Table 9: Avg. disk accesses per query: clustered data, uniform window query

	D2	D4	D6	D8	D10
dimsort	121	110	101	95.0	90.8
hilbert	101	28.8	14.6	8.12	5.32
rstar	98.2	28.0	14.7	9.66	6.76
iteropt	100	23.2	5.52	1.03	0.312

Table 10: Avg. disk accesses per query: clustered data, data centered window query

	D2	D4	D6	D8	D10
dimsort	155	148	144	140	144
hilbert	133	45.8	28.2	20.8	18.3
rstar	130	46.0	30.4	26.6	23.6
iteropt	131	37.8	14.0	6.02	4.40