Cost-based Unbalanced R-Trees

Kenneth A. Ross*
Dept. of Computer Science,
Columbia University
New York, NY 10027-7003
kar@cs.columbia.edu

Inga Sitzmann and Peter J. Stuckey
Dept. of Computer Science,
The University of Melbourne,
Parkville 3052, Australia.
{inga,pjs}@cs.mu.oz.au

Abstract

Cost-based unbalanced R-trees (CUR-trees) are a cost-function based data structure for spatial data. CUR-trees are constructed specifically to improve the evaluation of intersection queries, the most basic selection query in an R-tree. A CUR-tree is built taking into account a given query distribution for the queries and a cost model for their execution. Depending on the expected frequency of access, objects or subtrees are stored higher up in the tree. After each insertion in the tree, local reorganizations of a node and its children have their expected query cost evaluated, and a reorganization is performed if this is beneficial. No strict balancing of the trees applies allowing the tree to unfold solely based on the result of the cost evaluation.

We present our cost-based approach and describe the evaluation and reorganization operations based on the cost function. We present a cost model for in-memory access costs and we present three different query models. In our experiments, we compare the performance of the CUR-tree to the R-tree and the R^*-tree. The CUR-tree is able to significantly improve intersection query performance, without unacceptably increasing the cost to build the tree. The use of R-trees for in-memory data reflects the high (and growing) cost of bringing data from RAM into the CPU cache relative to the cost of other computation.

1 Introduction

Applications based on spatial data, e.g., Geographic Information Systems (GIS), Computer-Aided Design (CAD), Satellite Image Bases and Medical Applications, are rapidly expanding. Spatial Database Management Systems have to master the challenge to provide efficient access to the data used by these applications. The large volume of the data and the variety of very specialized queries require fast and flexible data structures.

One of the most typical spatial queries are intersection queries which involve finding all objects intersecting a query rectangle (range queries) or point (point queries). Other types of spatial queries include finding objects within a certain distance from a query object (distance queries) or finding pairs of objects that intersect or are within a certain distance from each other (join queries). In this paper we concentrate on intersection queries. Typical spatial data structures, supporting these queries for in-memory and disk-based data, include interval trees [Ede83a, Ede83b], priority search trees [McC85], quadtree-based structures, and the R-tree and its variants [Gut84, BKSS90]. Surveys of the various spatial data structures presented in the literature can be found in [GG98] and [Sam89].

In recent years, a significant amount of research has been done to address two research issues: (i) How can spatial index structures be made more efficient by using general measures not involving specific characteristics of the data, queries, or system [KF94, YLL98, GG97]? and (ii) How can the

* The work of Kenneth Ross was supported in part by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374. Part of this work was performed while the author was visiting the University of Melbourne.

performance of an existing spatial index in a system be estimated using the properties of the data [TSS00, FK94] and of the expected queries [PSTW93]?

In this paper, we suggest reversing the order of these two steps. Instead of evaluating an index a posteriori to estimate its performance, we suggest using a cost model a priori to build an index optimized with respect to system characteristics and query patterns reflected in a cost function. We present a spatial data structure which directly takes into account the expected cost of accessing data while building the data structure.

The cost-based unbalanced R-tree (CUR-tree) is based on R-trees but does not have to follow strict rules about tree shape, e.g., balancing. Instead, a cost function determines the expected intersection query cost for a given query model under a certain cost model for possible tree arrangements. The tree with the smallest average query cost is selected.

1.1 Main Memory Use of R-Trees

Main memory sizes have been increasing rapidly, to the point where many applications can fit their data entirely within main memory. Even when the entire data does not fit, CPU and main-memory performance are often the main performance bottleneck in commercial disk-based database systems [GL01]. Thus, the focus of database performance research is beginning to shift from optimizing I/O performance to optimizing CPU and memory performance. Thus, we focus on main-memory performance in this paper.

Why do we focus on R-trees for in-memory performance? R-trees were designed for disk-based systems in which transfering a block of data from disk to memory was the *dominant* cost. Looking at the next level of the memory hierarchy, we can make a similar argument for data transfer between main memory and the data cache. At present, a data cache miss in the cache-level closest to the main memory incurs a delay of perhaps a hundred cycles on modern architectures. While this does not yet represent a dominant cost, we need to look a few years ahead. During the recent past, CPU speeds have been increasing at 60% per year, while RAM speed has been increasing at a more modest 10% per year [CLH98]. As a result, the relative cost of a cache miss is increasing at about 45% per year, and is expected to continue to increase at this rate for years to come. Eventually, this data transfer cost will become dominant. This architectural picture explains our choice of R-trees. Other in-memory techniques are available (see Section 8), but they are far more difficult to tune for cache performance than R-trees.

Our cost function models the cost of accessing R-tree structures in main memory. Extending the CSB+-tree technique presented in [RR00] makes R-trees a competitive in-memory data structure, as they are highly optimized with respect to cache accesses while still maintaining their dynamic behaviour. The main idea of [RR00] is to remove pointers from B+-tree index nodes: some nodes are stored contiguously, meaning that arithmetic on memory addresses can be used to locate child nodes. The net effect is that the branching factor is dramatically increased, leading to fewer node accesses.

1.2 Paper Organization

The paper is organized as follows. We give a motivating example in Section 2. Section 3 describes R-trees and the cost involved in querying R-trees under several cost and query models. Section 4 introduces CUR-trees. Possible reorganizations for a CUR-tree are described in Section 5. Section 6 discusses how we build a CUR-tree. Experimental results compare the performance of the CUR-tree to the R- and R*-tree in Section 7. Related work is described in Section 8 and a summary and conclusions are given in Section 9.

2 A Motivating Example

R-trees are a balanced multi-way tree structure where all objects are stored in leaf nodes. Non-leaf nodes consist of pointers to the children of the node and minimum bounding boxes describing

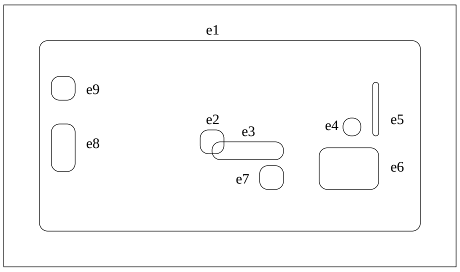


Figure 1. A collection of spatial objects

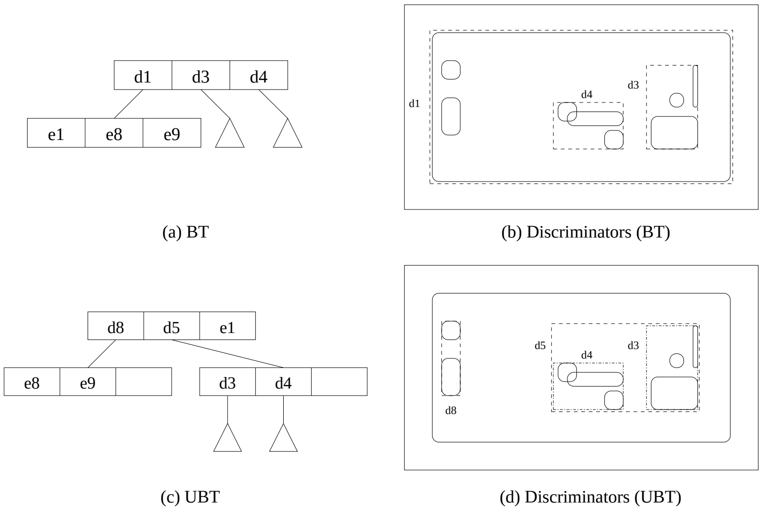


Figure 2. Storing large items higher up the tree

the entries in the child nodes.

Figure 1 shows a number of objects $e_1, \ldots, e_9$. Object e_1 covers a large part of the query space and contains the other objects $e_2, \ldots, e_9$. Object e_1 will typically intersect with a significantly larger proportion of queries than, for example, object e_4. We assume that the query-able space is the bounding box of all data objects.

The balanced R-tree (BT) must store $e1$ at the leaf level, as illustrated in Figure 2 (a). In this tree, both the root and the leftmost leaf will be retrieved for a large proportion of queries. Figure 2 (b) shows the objects as round corner boxes with solid lines and the bounding boxes used as discriminators as dashed line boxes. An unbalanced R-tree (UBT) may store $e1$ at the root level, as illustrated in Figure 2 (c). This results in only one node access to read $e1$. Furthermore, a significantly smaller discriminator $d8$ can be used to describe the remaining objects $e8$ and $e9$ as shown in Figure 2 (d). Here second level discriminators are shown as dot-dash boxes.

But what is it specifically that makes storing an item higher up the tree worthwhile? And when is the advantage of storing the item high up the tree outweighed by the need to push other items further down (the entire trees underneath $d3$ and $d4$ in Figure 2 (d))?

Our answer to these questions is to use a cost-model of query evaluation to understand the access costs of data on a particular system and a query-model to predict queries which are most likely. The cost-model and the query-model can be combined into a cost function which predicts the average intersection query cost in a tree under the given parameters and reorganizes it accordingly.

For example, the cost function of (intersection) query evaluation for R-trees in main memory consists of two parts: (i) the cost L to retrieve an index node, and (ii) the cost C to compare the query box versus bounding boxes of items or subtrees. Suppose a function $\mathsf{prob}(d)$ determines what proportion of queries intersect a bounding box d. In the example, if we examine the cost of accessing BT, the root node will always be accessed for a cost of L and three comparisons will always take place for a cost of $3C$. The remaining cost depends on which of the discriminators $d1$, $d3$ and $d4$ intersect the query.

The average cost of accessing BT is then $L + 3C + c1 + c3 + c4$ where $c1$, $c3$ and $c4$ are the average costs of accessing the trees pointed to by $d1$, $d3$ and $d4$. Ignoring costs to retrieve the objects themselves, the cost $c1$ is $\mathsf{prob}(d1) \times (L + 3C)$ since it will be accessed by the proportion of queries $\mathsf{prob}(d1)$ and when accessed involve one node access and three comparisons. As $d1$ covers about 90 per cent of the query space, queries will intersect $d1$ with $\mathsf{prob}(d1) = 0.9$. Then the average cost for BT is $1.9L + 5.7C + c3 + c4$.

Similarly, we can determine the average cost of accessing UBT as $L+3C+c8+c5$. The cost $c5$ to access $d5$ is $\mathsf{prob}(d5)\times(L+2C)+c3+c4$. The total average cost for UBT assuming $c8 = 0.1\times(L+2C)$ and $\mathsf{prob}(d5) = 0.33$ is therefore $1.43L + 3.86C + c3 + c4$ which is clearly superior.

This example illustrates how a cost function determines which organization of the tree will yield better intersection query performance than the other. It also shows that determining the cost of a node access is possible with information already available. No additional cost information needs to be stored in the tree.

3 Cost and Query Model Based Analysis of R-Trees

3.1 The R-Tree

The R-tree [Gut84, BKSS90] is a data structure for n-dimensional data. Geometric objects are stored in leaf nodes of the form $[(d_1, oid_1), \ldots, (d_n, oid_n)]$ where each d_i is the minimum bounding box describing the object pointed to by the object identifier oid_i. Non-leaf nodes contain entries of the form $[(d_1, t_1), \ldots, (d_n, t_n)]$ where each d_i is the minimum rectangle that encloses all entries in the subtree t_i. Let M be the maximum number of entries per node and $m \leq M/2$ be the minimum number of entries per node. For disk-based data, the size of M usually corresponds to the number of entries that fit a disk block. For in-memory data, the size of a node can exceed the size of a cacheline.

Further rules which determine the shape of the R-tree are:

1. All leaf nodes appear on the same level.
2. Every node which is not the root node contains between m and M entries.
3. The root node has at least two entries unless it is a leaf node.

If insertion of an entry in a node results in an overfull node, the node is split. Splitting algorithms of linear and quadratic complexity have been presented in the literature [Gut84, BKSS90, YLL98, KF94]. For an R-tree node t (and later for CUR-tree nodes) we use notation $t.n$ to refer to n, the number of entries in a node, $t[i]$ to refer to the pair (d_i, t_i), while $t[i].d$ and $t[i].t$ refer to d_i and t_i respectively.

3.2 The Cost of an Intersection Query

The cost of accessing data in an R-tree can be estimated using a query model which predicts the distribution of queries and a cost model which reflects the cost of processing a node for a particular system. The query model is reflected in a function $\mathsf{prob}(d)$ which gives the probability that the rectangle d will match a query (i.e., intersect the query box for a range query). The raw cost of processing a node with n entries is given by a function $\mathsf{cost}(n)$.

The average cost of processing a node t with n entries and minimum bounding box d can therefore be summarized as

$$\mathsf{prob}(d) \times \mathsf{cost}(n)$$

since on average only the proportion $\mathsf{prob}(d)$ of queries access the node. The average cost acost of an intersection query for a tree rooted at node t and bounding box d in an arbitrary query is simply the sum of the costs of the nodes of the tree.

$$\mathsf{acost}(t) = \mathsf{prob}(d) \times \mathsf{cost}(t.n) + \Sigma_{i=1}^{t.n} \mathsf{acost}(t[i].t)$$

If a subtree t points to an object, we assume that $\mathsf{acost}(t) = 0$. Note that we can ignore the costs for retrieving an object since any arrangement of the R-tree will result, for a particular query, in the same set of objects being retrieved and thus the same retrieval cost.

3.3 The Cost Model

The cost model for R-trees in this paper approximates the cost of querying in-memory R-trees. Accessing a node of an R-tree in main memory involves (i) reading the node into the cache and (ii) comparing each entry with the query to decide whether to continue search in the corresponding subtree. If E entries fit into one cacheline, the cost of a node with n entries consists of the cost to read the node from RAM into $\lceil \frac{n}{E} \rceil$ cachelines each at cost L and n comparisons each at cost C.

$$\mathsf{cost}(n) = \lceil \frac{n}{E} \rceil \times L + n \times C$$

The number of cache hits and misses that occur in an environment where several index structures compete for cache is difficult to predict. We therefore do not attempt to simulate real cache behaviour, but approximate it by assuming no cache hits occur and give an overestimate of the cost. In general, computation can be overlapped with data transfer between main memory and the cache. However, in our context there is very little potential for overlap within the search process. A cacheline must be cache-resident before any computation on its data items can take place, and the next cache-lines to be read depend on that computation. Thus we add the computation and memory latency components, consistent with the assumption of no overlap.

3.4 Query Models

We define query models for the two most common types of intersection queries: point queries and range queries. We also give examples for more specialized query models. The query models reflect the probability that an item d matches a query q of a given type of query.

3.4.1 Uniform Point Query Model

The point query model reflects the assumptions about queries commonly made in spatial data structures. Points which are uniformly distributed over the query space are used to query the data. Consider again the collection of spatial objects in Figure 1. As explained in the motivating example, object $e1$ is very likely to match a query as it covers a large area of the query space. The probability that an entry d contains a query point p can be measured as $\mathsf{area}(d)/\mathsf{area}(w)$ where w is a rectangle describing the total area of the data. In this cost model, the function $\mathsf{prob}(d)$ is simply the function $\mathsf{area}(d)/\mathsf{area}(w)$, calculating the percentage of the whole area covered by the entry.

3.4.2 Range Query Model

Range queries use rectangles to query the data. A common assumption is that the query rectangles are distributed uniformly over the query space and have a fixed side length l. If r is a rectangle, let $r.low[i]$ and $r.high[i]$ denote the lower and upper value of r in dimension i. The probability that a query rectangle q with side length l intersects an entry d is the probability that $q.low[i]$ intersects the interval $(\mathsf{max}(d.low[i] - l, w.low[i]), d.high[i])$ in each dimension i.

Therefore, the probability that an entry d intersects with a query rectangle q with side length l

in an 2-dimensional query space w can be calculated as

$$\prod_{i=1}^{2} \frac{d.high[i] - \mathsf{max}(d.low[i] - l, w.low[i])}{w.high[i] - w.low[i]}$$

The range query model is a generalization of the point query model. Clearly, for length $l = 0$, the formula above is the function $\mathsf{prob}(d)$ of the point query model.

3.4.3 Skewed Point Query Model

As in real applications queries are not always distributed uniformly over the workspace, we extend our query models to include point queries with any kind of distribution $\mathcal{D}(x, y)$. The probability that a discriminator d in a workspace $[0, 1] \times [0, 1]$ intersects with a point query is given by

$$\int_{d.xl}^{d.xu} \int_{d.yl}^{d.yu} \mathcal{D}(x, y)\, dy\, dx$$

We present experimental results in Section 7 for $\mathcal{D}(x, y) = (\frac{x^2 \times y^2}{9})$ over the range $[0..1, 0..1]$. This represents highly skewed query pattern where queries are much more likely in the top right-hand corner of the query space.

Query models can be used to specifically improve the performance of one particular type of query. As many query distributions may contain queries of many different types, it is important to note that query models can be derived for any query distribution, not just distributions with a single type of query. We chose the three query models as examples to demonstrate the possibilities of the CUR-tree.

4 Cost-based Unbalanced R-trees

As R-trees have to be used on a wide range of systems with different cost parameters, evaluating different kinds of queries on data with a variety of distributions, the R-tree structure is too general to cater for all these possibilities. The strict restrictions on the shape exclude many possible better arrangements of the data. The algorithms for splitting do not take specific characteristics of system or query into account. We suggest using a dynamic, self-organizing index structure which is based on R-trees but whose shape is only influenced by a cost function.

4.1 The Structure of CUR-Trees

CUR-trees are multi-way search trees based on R-trees. A node in the CUR-tree is of the form $[(d_1, t_1), \ldots, (d_m, t_m)]$ where each d_i is a minimum bounding box describing an object or a collection of objects. The pointer t_i points to either a subtree if d_i describes a collection of objects or is the object identifier of a single object. There is no explicit minimum number of entries per node. The maximum number of entries per node is given as a multiple N of the number E of entries that fit into a cacheline.

The rules of the CUR-tree structure are:

1. Each node can store pointers to both objects and other nodes
2. Every node contains at most $N \times E$ entries
3. The arrangement of the tree should be locally optimal with respect to the cost function

If inserting another entry into the node will overflow the node, the node is split or entries are demoted or promoted. The operation will be chosen that results in the best tree arrangement according to the cost function.

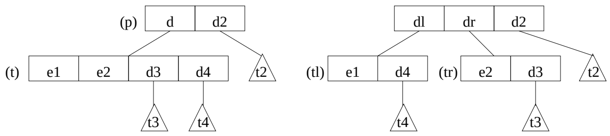


Figure 3. Split of a node

4.2 Operations on CUR-trees

In contrast to R-trees, operations on CUR-trees are invoked not only when a node overflows. Instead, upon insertion or deletion, every node on the insertion path is evaluated to see whether its entries can be better arranged with respect to the cost function. The operations used in the reorganization process are *split*, *demotion*, and *promotion*.

5 Reorganization of CUR-Trees

5.1 Split

The split operation splits a node into two nodes and redistributes its entries using the cost evaluation function. The distribution is picked which creates the cheapest tree according to the cost function acost. In Figure 3, the node t with discriminator d is split into nodes tl and tr with discriminators dl and dr. An extra entry is added to the parent node p.

We can determine when a split is advantageous by determining the change in cost for accessing the tree before and after the split. Let p be the parent node of the split, with discriminator pd. The change in acost for p is

$$\begin{aligned}&\mathsf{prob}(pd) \times \mathsf{cost}(p.n+1) - \mathsf{prob}(pd) \times cost(p.n)\\&+\mathsf{prob}(dl) \times \mathsf{cost}(tl.n) + \mathsf{prob}(dr) \times \mathsf{cost}(tr.n)\\&-\mathsf{prob}(d) \times \mathsf{cost}(t.n)\end{aligned}$$

In practice, the split operation needs to determine which possible distributions it will consider. Then, for each split considered, it calculates the change in cost for the parent tree. A split using the best distribution is then performed if it is advantageous.

5.1.1 Exponential Split

The simplest splitting algorithm is to consider all $2^{t.n}$ possible distributions. For larger node sizes, this algorithm is impractical.

5.1.2 Quadratic Split

The quadratic split is based on a generalization of the quadratic R-tree algorithm [Gut84] to arbitrary query measures. Two seeds are picked for the two new nodes *left* and *right*. The entries are chosen as seeds whose minimum bounding box would have the highest prob-value if put into one node. Subsequently, the entry is chosen from the remaining entries which shows largest preference for one group, i.e., increases the prob-value of that group least, and added to that group. This step is repeated until all entries have been distributed into the two groups. To create more than just one possible distribution, we also evaluate intermediate distributions: we use the group *left* for one node and add all remaining entries to the group *right* and vice versa. That is after each step we consider another two possible distributions defined by the pairs $(left, t - left)$ and $(t - right, right)$ for each value that *lefts* and *rights* take in the process. This results in $n - 1$ split distributions for a node with n entries.

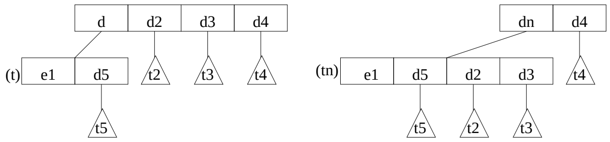


Figure 4. Demotion of a number of entries

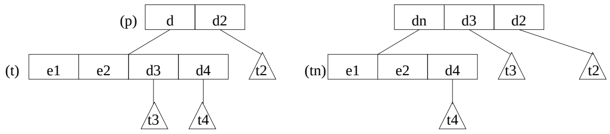


Figure 5. Promotion of an entry

5.1.3 Linear Split

The linear split is a generalization of the linear R-tree algorithm. The algorithm picks the two seeds in one linear scan of the entries. Entries that are furthest apart from each other are determined for every dimension as a possible seed pair. The dimension is chosen for the split that shows the largest separation between the two possible seeds. After the seeds have been determined, another scan through the entry set assigns each entry to either *left* or *right*. The set is chosen which shows the least increase of the prob-value. Again, after each step we consider another two possible distributions defined by the pairs $(\mathit{left}, t - \mathit{left})$ and $(t - \mathit{right}, \mathit{right})$ for each value that *lefts* and *rights* take in the process.

5.2 Demotion

Demotion moves a single entry or group of entries to a lower level in the tree. This can be an already existing node or a newly created node, depending on which arrangement results in an overall cheaper tree. In Figure 4, the entries $(d2, t2)$ and $(d3, t3)$ are demoted into the subtree t giving tn.

We can determine when a demotion is advantageous by determining the change in cost for accessing the tree before and after the demotion. Let p, with discriminator pd, be the parent of node t, with discriminator d, into which k nodes are demoted. The resulting new node is tn. Then the change in acost for p is

$$\begin{aligned}&\mathsf{prob}(pd) \times \mathsf{cost}(p.n - k) - \mathsf{prob}(pd) \times \mathsf{cost}(p.n)\\&+\mathsf{prob}(dn) \times \mathsf{cost}(t.n + k) - \mathsf{prob}(d) \times \mathsf{cost}(t.n)\end{aligned}$$

Possible groups of entries for demotion are created by using the *left* and *right* groups created by the splitting algorithm.

5.3 Promotion

Promotion moves an entry to a higher level in the tree. Entries with a large probability of being accessed should be stored high up in the tree to minimize the costs for searching and accessing them. In Figure 5, the entry (d_3, t_3) is deleted from node t and inserted in its parent node p.

We can determine if a promotion is advantageous by determining the change in cost of the resulting tree. Let p be the node into which a node t with discriminator d is promoted, with discriminator pd, the change in acost for p is

$$\begin{aligned}&\mathsf{prob}(pd) \times \mathsf{cost}(p.n + 1) - \mathsf{prob}(pd) \times \mathsf{cost}(p.n)\\&+\mathsf{prob}(dn) \times \mathsf{cost}(t.n - 1) - \mathsf{prob}(d) \times \mathsf{cost}(t.n)\end{aligned}$$

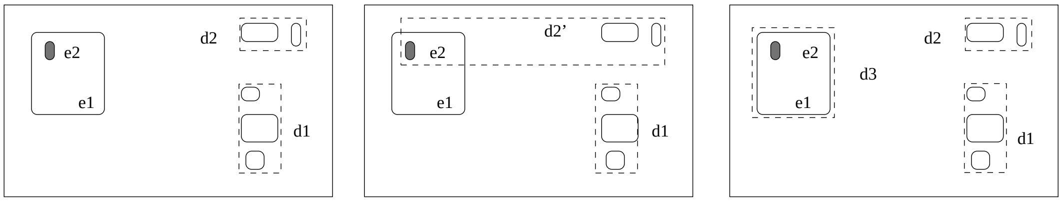


Figure 6. Insertion with and without insertion into objects

Note that promotion is a special case of a split. Imagine the split operation results in an arrangement with all entries but one in the old node and only one node in the new node. In this case, the entry can be directly inserted into the parent node.

6 Building the CUR-tree

The intersection query performance of CUR-trees depends on the decisions made about two main aspects during insertion: (i) the best node for insertion has to be found and (ii) operations on that node and all nodes on the insertion path have to optimize the cost of every node of the path.

6.1 General Insertion Algorithm

Inserting a new entry (d, e) in a CUR-tree t occurs by traversing the tree t until a suitable node for insertion has been found. Once the new entry is inserted, each node in the insertion path from the new entry to the root is updated and evaluated to see if some reorganization operation: split, promotion or demotion would improve the node.

```
insert(t, (d, e))
    choose the entry i for which prob(t[i].d ⊔ d) − prob(t[i].d) increases least
    if t[i] is an object (d', e')
        replace t[i] by a new node containing {(d', e'), (d, e)}
    else
        t[i] := insert(t[i].t, (d, e))
    if t[i] changed locally (ignoring changes in its children)
        return reorganize(t)
    else return t
```

Note that the insertion always finishes by creating a new node containing one object already in the tree and the entry to be inserted (object insertion). To avoid unnecessary computation, reorganization is only performed on node t if the information in the node changes after the insertion in its child.

6.2 Choosing the Entry for Insertion

To determine where a new entry should be inserted, the increase of the prob-value for each child of a node after inserting the new entry is measured. The entry is preferred that shows the least increase. If two entries show the same increase of their prob-value, ties are resolved by choosing the entry with the smallest total prob-value. Note that we consider the possibility of inserting a new entry into an object (which will effectively create a new node with two entries) at every level of the tree. This is important because this option can lead to much better CUR-trees. See the example in Figure 6.

The leftmost figure shows five geometric objects (depicted as rectangles with round corners), two discriminators $d1$ and $d2$ (rectangles) grouping the five entries, a large entry $e1$ and a grey new entry $e2$. The large object $e1$ is stored on the same level as the discriminators $d1$ and $d2$. When inserting the new entry $e2$ under the condition that only subtrees can be considered for insertion, insertion would result in the scenario depicted in the middle figure. One of the discriminators, $d2$ in the example, has to be enlarged significantly to contain the new entry $e2$. By doing so, the cost of $d2$ is increased significantly and space in the discriminator has been wasted as $e2$ is only a relatively small object. As the entry $e1$ contains $e2$, it is better to create a new discriminator $d3$ that contains both entries $e1$ and $e2$. This is shown in the right figure.

6.3 Reorganization

Reorganization determines which, if any, of the operations demote, promote or split produces the best local tree arrangement. The operation which results in the smallest local cost is executed. We restrict reorganization so that each reorganization considered must involve something newly created in the insertion, to eliminate unnecessary computation. This is based on the assumption that other beneficial reorganizations should already have taken place. In our implementation we also determine the kinds of reorganization operation which could be beneficial at a node from the changes made to its children.

If the current node is overfull, we restrict reorganizations of that node to those that remove the overfull condition and do not create it in a node below. The reorganization can however make the parent node overfull.

7 Experiments

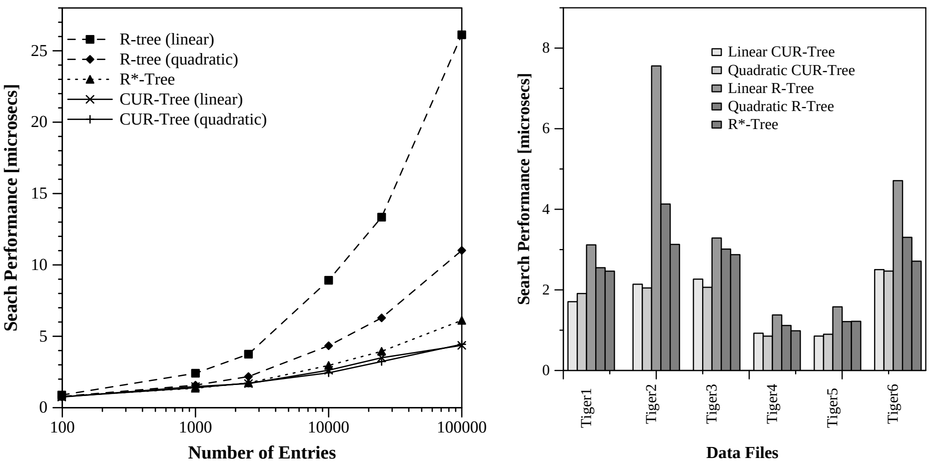
We implemented the CUR-tree and compared its performance in a series of experiments to an ordinary R-tree and the R*-tree. In this section, we present graphs showing the performance of CUR-trees, R-trees and R*-trees on a real system for real test data.

More experimental results including experiments with an optimized evaluation of split, demotion and promotion operations can be found in [RSS00]. In that report, we also describe the operations in greater detail.

7.1 Implementation Details

We implemented a common in-memory traversal algorithm for R-trees, R*-trees and CUR-trees. The algorithm reads in the tree structure (which is generated by either the R-tree/R*-tree algorithms or our cost-based algorithms) and performs a large number of queries on it. We store all coordinates as 16-bit values. (This is reasonable for many applications; for example, it allows two decimal places for longitude and latitude.) We use a variant of the CSB+-tree technique [RR00] to store one pointer per node (to a contiguous group of children) rather than one pointer per entry. A level-2 cache line on our target architecture (Sun Ultra) is 64 bytes. Thus a node-size equal to the cache-line size allows the storage of 7 boxes, together with a 32-bit pointer, an 8-bit counter (denoting how many entries are present) and an 8-bit bitmap. The bitmap is used to distinguish between bounding boxes of objects, and bounding boxes of subtrees. In our experiments, $N \times E = 7$.

To generate tree structures, we used the R-tree code by A. Guttman and the R*-tree code by N. Beckmann, and our own CUR-tree code. To compare insertion times fairly we reimplemented the R*-tree insertion in the same environment as the CUR-tree code, since Beckmann's code performs additional operations for simulating buffer behaviour which are not performed by the R-tree or CUR-tree code.



(a) Point data (b) TIGER data

Figure 7. Average query cost (μs) for point queries

7.2 Test data

For the experiments described in this section, we used an artificially created point data set and six files of the TIGER/Line Data collection. The point data sets consist of uniformly distributed points. The file sizes range from 100 entries to 100,000 entries. The TIGER/Line data sets vary in size from 3,743 entries to 234,250 entries. They are characterized by a skewed object distribution and a high density of objects in the populated areas. CUR-trees promise to be particularly useful in these condition with decreasing improvement for uniform data.

7.3 Search Performance

7.3.1 The cost values

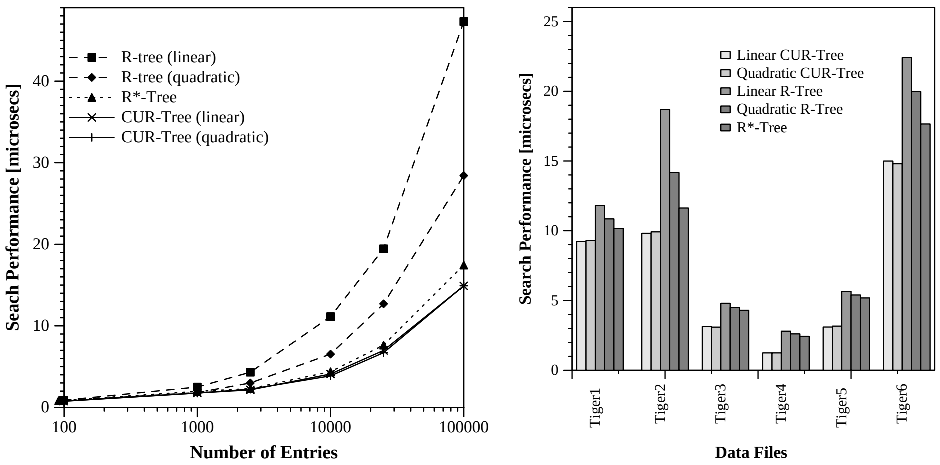
The cost values used in our experiments are based on the actual cost of reading a node into the cache and the cost of comparisons in the cache using a real system. We used the memory-cpu latency of 170ns of our target architecture (Sun Ultra-5 with 270 MHz and 256 MByte RAM) for L [Sun99]. We measured the comparison cost C to be 85ns per entry. The comparison cost includes both the cost of a comparison itself and the cost of some supplementary computation such as incrementing counter variables.

While these values are used in the cost-model for predicting the search cost of the trees produced, the experimental results are all given by actual search time per query as measured by the in-memory traversal algorithm.

7.3.2 Point query model

In our first experiment, we compared the actual search performance of a CUR-tree under the point query model with an ordinary R-tree and R*-tree.

For both point data and TIGER data, the CUR-trees clearly outperform the R-trees. Even compared with the R*-tree, the CUR-trees show better performance. Figure 7 (a) shows the average time for a point query on the point data for linear and quadratic CUR-trees, linear and quadratic R-trees and the R*-tree. The largest improvement is shown by the linear CUR-tree which outperforms the linear R-tree by a factor of seven. The quadratic CUR-tree reduces the search time by a factor of four compared to the quadratic R-tree. Compared to R*-trees, the CUR-trees still reduce the search time by about 35 per cent. The quality of the splits is not as important for the relative performance



(a) Point data (b) TIGER data

Figure 8. Average query cost (μs) for range queries

of the CUR-trees as it is for the R-trees. The performance of the R-tree depends strongly on the split, whereas only slight performance differences can be observed for the CUR-trees. The R*-tree shows better performance because of dynamic reinsertion of objects which helps to reorganize the tree.

For the TIGER data, the results are depicted in Figure 7 (b). Again, the linear CUR-tree improves on the linear R-tree clearly for all data sets, with an improvement of a factor four for the set Tiger2. The quadratic CUR-tree shows similar performance and improves on the quadratic R-tree by up to a factor of two. Both CUR-trees improve on the performance of the R*-tree by between 20 and 35 per cent.

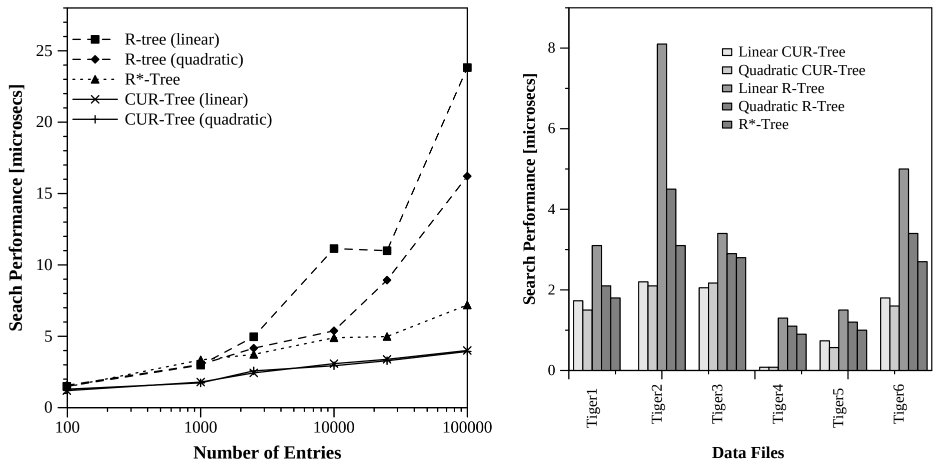
7.3.3 Range query model

The results for range queries are shown in Figure 8. The CUR-trees built under the range query query model also improve significantly on linear and quadratic R-trees and show smaller improvement compared to the R*-tree.

The results for the point data set in Figure 8 (a) show, that the linear CUR-tree outperforms the linear R-tree by a factor of three. The quadratic CUR-tree only takes half of the search time of the quadratic R-tree. Compared to the R*-tree, the CUR-trees show a slight improvement of about 12 per cent. For the TIGER data sets, the linear CUR-tree outperforms the linear R-tree by up to a factor of two. For the quadratic algorithm, the CUR-tree improves on the R-tree by between 20 to 30 per cent. The R*-tree is outperformed by both CUR-trees by about 10 per cent.

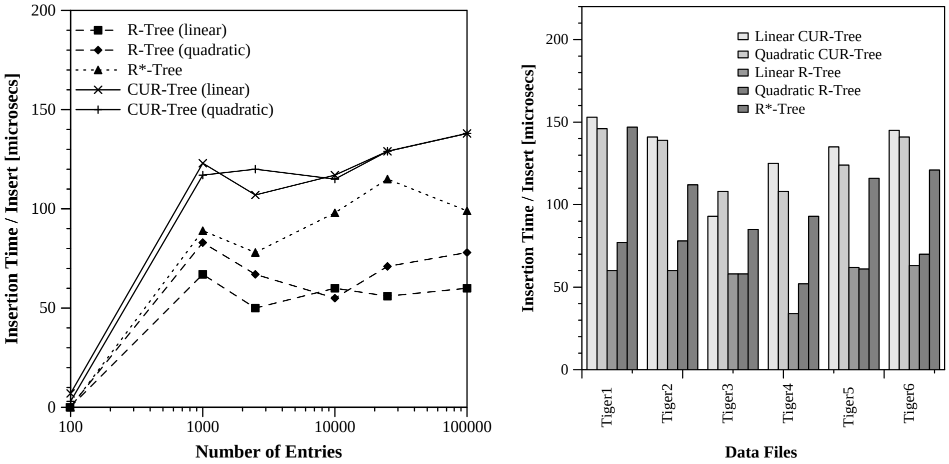
7.3.4 Skewed Point Query Model

We experimented with a skewed distribution of points for point queries. Figure 9 depicts results for the distribution $\mathcal{D}(x,y) = (\frac{x^2 \times y^2}{9})$. For point data, the query time for skewed point queries in the CUR-tree was half the time of the R*-tree. Compared to the R-trees, the CUR-tree outperforms the quadratic R-tree by a factor of almost five and the linear R-tree by a factor of seven. Differences between the CUR-tree algorithms were insignificant. For the TIGER data sets, improvement compared to the R*-tree ranges between 4 per cent for Tiger5 and a factor of 10 for Tiger4, as shown in Figure 9(b). Compared to the linear and quadratic R-tree, improvements also range from 4 per cent for Tiger5 and a factor of 10 for Tiger4 and shows larger improvement for the remaining data sets.



(a) Point data (b) TIGER data

Figure 9. Average query cost (μs) for skewed queries



(a) Point data (b) TIGER data

Figure 10. Average insertion times (μs)

7.4 Insertion Times

The insertion times of the CUR-tree compared to the insertion times of the R-trees and R*-trees are shown in Figure 10. As expected, CUR-tree insertion is slower than R-tree insertion because of the amount of evaluation carried out in each step. Compared to the R*-tree, on the other hand, the insertion times of CUR-trees are only slightly higher. For the point data insertion, insertion of a point into the CUR-trees takes on average twice as long as for R-trees. Compared the R*-tree, the CUR-tree is only about 30 per cent slower. For the TIGER data sets, as shown in Figure 10 (b), CUR-tree is again slower than R-tree insertion by a factor of two. The R*-tree insertion is up to 30 per cent faster, but shows the same insertion times for the Tiger1 data set.

Taking into account the improvement of performance of the CUR-tree even over the R*-tree and the greater flexibility given by the CUR-trees, the higher insertion times are justifiable.

7.5 Experiments with CUR*-trees

Considering the performance improvement shown by R*-trees compared to R-trees for range and point queries, one might expect that incorporating R*-tree techniques into CUR-trees might lead to further improvement. The strength of R*-trees is caused by dynamically reinserting entries as the default overflow treatment. Reinserting an entry might result in placing an entry at a more suitable place in the tree, improving the overall tree arrangement. We implemented reinsertion for CUR-trees (CUR*-trees). Our experiments showed no significant performance improvement. In fact, analyzing the reinsertion process, we found that deleted entries are reinserted in their old place in the CUR*-tree. This shows that CUR-trees have no need for further reorganization through reinsertion of entries. The cost-based insertion algorithm already produces good tree arrangements.

8 Related Work

R-trees are the basis of our work as they are suited for both point and rectangular data, and are suitable for dynamic contexts (where insertions and deletions are interleaved with intersection queries). Other spatial data structures such as interval trees [Ede83a, Ede83b] and priority search tree [McC85] are specialized for intersection queries, and have better asymptotic behaviour. But because of the more complex index structures they are more difficult to update, and hence not as useful in a dynamic context. They are also more difficult to tune for good cache performance.

The closest work to CUR-trees is [RAA97] which uses promotion and demotion to move data objects around the R-tree. In this approach overfull nodes are fixed by either promotion or demotion of single entries or split of the node. A performance improvement of up to 45 per cent compared to regular R*-tree has been achieved by this method. In contrast to our approach, the method only promotes and demotes objects rather than subtrees, hence it is identical to a regular R*-tree for point data. The trees built are always height-balanced R*-trees (although objects can appear in non-leaf nodes). No query model is taken into account, so the only impetus for reorganization is an overfull node. The concept of treating large objects separately has also been investigated in [GG97] where, to avoid fragmentation in cell trees, large objects are stored in separate buckets.

Analysis of the cost of spatial queries in R-trees has been presented in [TSS00]. The authors concentrate on estimating selection or join cost using parameters such as density of data and average extent. In contrast we use cost estimates that are particular to the CUR-tree that is being built, hence the cost models are completely different. A more accurate estimation for skewed point data is presented in [FK94] by describing the distributions as a fractal dimension.

Pagel et al. [PSTW93] introduced a performance measure which analyzes cost of accessing spatial data structures under intersection queries for different query models. They concentrate on bucket (for R-trees leaf-node) accesses, since the analysis is independent of the data structure holding the spatial data. Hence their analysis is in a sense orthogonal to ours which ignores the object retrieval phase.

9 Conclusions

We have introduced CUR-trees, a methodology for building R-tree-like spatial index structures that combines a probability based query model with a cost model. Using the estimated query cost CUR-trees reorganize their structure to improve intersection query evaluation. In all experiments, CUR-trees showed a significantly improved performance for both synthetic and real test sets. The overhead of reorganization in building a CUR-tree leads to insertion times which are still acceptable considering the improvement of search performance.

Our work shows the potential of making data structures more dynamic with regard to cost function. So far, we have only investigated intersection queries in an R-tree. We will extend our work to investigate the join costs for R-trees.

References

[BKSS90] 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/PODS 1990*, pages 322–331, 1990.

[CLH98] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical Report 98, University of Wisconsin-Madison, Computer Science Department, University of Wisconsin-Madison Madison, Wisconsin 53706, 1998.

[Ede83a] H. Edelsbrunner. A new approach to rectangle intersections: Part I. *International Journal of Computer Mathematics*, 13(3–4):209–219, 1983.

[Ede83b] H. Edelsbrunner. A new approach to rectangle intersections: Part II. *International Journal of Computer Mathematics*, 13(3–4):221–229, 1983.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *SIGMOD/PODS 1994*, pages 4–13, 1994.

[GG97] Oliver Günther and Volker Gaede. Oversize Shelves: A Storage Management Technique for Large Spatial Data Objects. *International Journal of Geographical Information Science*, 1(11):5–32, 1997.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *Computing Surveys*, 30(2):170–231, 1998.

[GL01] Goetz Graefe and Per-Ake Larson. B-tree indexes and cpu caches. In *ICDE 2001*, April 2001.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD/PODS 1984*, pages 47–57, 1984.

[KF94] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *VLDB 1994*, pages 500–509, 1994.

[McC85] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *SIGMOD/PODS 1993*, pages 214–221. ACM Press, 1993.

[RAA97] K.V. Ravi Kanth, Divyakant Agrawal, and Amr El Abbadi. Indexing non-uniform spatial data. In *Proceedings of IDEAS 1997*, pages 289–298, 1997.

[RR00] Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Procs. of ACM SIGMOD Conference*, pages 475–486, 2000.

[RSS00] Kenneth A. Ross, Inga Sitzmann, and Peter J. Stuckey. Cost-based Unbalanced R-Trees. Technical report, CSSE, The University of Melbourne, 2000.

[Sam89] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1989.

[Sun99] Sun Microsystems, Inc. *The Sun Ultra 5 and Ultra 10 Workstation Architecture – Technical White Paper*, 1997–1999.

[TSS00] Yannis Theodoris, Emmanuel Stefanakis, and Timos Sellis. Efficient Cost Models for Spatial Queries Using R-Trees. *IEEE TKDE*, 12(1):19–33, 2000.

[YLL98] Yvan J. Garcia R., Mario A. Lopez, and Scott T. Leutenberger. An Optimal Node Splitting for R-trees. In *VLDB 1998*, pages 334–344, 1998.