Improving the R*-tree
Storage Allocation Algorithm

Arnaut de Rijk

September 1999 - August 2000
Improving the R*-tree Storage Allocation Algorithm

Master Thesis
Arnaut de Rijk

September 1999 - August 2000
Abstract:

Geographic information systems (GISs) are one of the major applications of database systems. GISs are characterized by the huge amount of spatial data that has to be handled (in the order of thousands of terabytes, i.e. petabytes). This makes fast indexing techniques indispensable. The huge size of the databases causes indexing methods to be stored on slow secondary memory (i.e. disk), because they are too big to fit into primary memory.

One of the most successful and applied indexing techniques is the R*-tree [Beckmann, 1990]. A lot of research has been conducted to improve its performance. However, one issue that has been overlooked is its storage on secondary memory. We present a hypothesis, that it is possible to exploit spatial relations between nodes to design and construct a (dynamic) R*-tree storage allocation algorithm that improves query processing by reducing seek time costs.

We designed and implemented different types of storage allocation algorithms. The empirical results obtained from an algorithm that clusters the leaves of the R*-tree on disk (Dynamic Clustered Leaves) are used to sustain our hypothesis. Compared to algorithms that do not use spatial relations, a reduction of query time up to 64% can be achieved. However, insertions and deletions are up to 54% slower and the structure occupies around 60% more disk space.
Preface

This report describes my master thesis project. The master thesis is part of the Computer Science study at the Delft University of Technology. The project was conducted at the Federal University of Rio de Janeiro (UFRJ) in Brazil. The whole project had a duration of nine months. I stayed in Brazil from September 7, 1999 until June 7, 2000.

Reading this report is worthwhile, if you are interested in subjects such as databases and storage allocation or if you are interested in what I have been doing these 9 months.

I want to thank the following people, because without them I would never been able to do this in a strange foreign country. I also would like to thank them for making my stay here in Rio de Janeiro most pleasant.

Claudio Esperança and Jano Souza de Moreira for their knowledge and excellent support in Rio de Janeiro. Waltraud Gerhardt and Edward Verbree for their support from Delft.

My roommates and new friends in Rio, Marcos, Ricardo, Lucio, Fernando, Beto, Victor, Novello, Milton and many others. My roommate and friends in the Netherlands; Pieter, Christine, Joost, Sander, Raymond, Richard, Bertil, Bart and everyone who I can’t think of right now.

And of course, my parents.

I placed a quote from Martin Walkyier, lead singer and songwriter of Skyclad, with every chapter. It adds an extra flavor and hopefully some smiles during the reading of this report.

Arnaut de Rijk, 9th of August, 2000
Summary

Over the last few years, there has been a great deal of interest in extending traditional, alphanumeric databases to handle multi-dimensional spatial data. One of the applications are geographic information systems (GISs). GISs are a rich source of problems, which are mainly caused by the huge amount data (in the order of petabytes, i.e. thousands of terabytes) that has to handled. This causes the necessity for indexing techniques that speed up the modification and query processes, since the data has to be stored on slow secondary memory (i.e. disk). The indexing structures are also huge and too big to fit into fast primary memory (i.e. RAM) and also have to be stored onto secondary memory.

One of the most successful and applied indexing technique for GISs is the R*-tree [Beckmann, 1990]. This is an improved variant of the R-tree [Guttman, 1984] that is a dynamic height-balanced tree similar to the B*-tree [Bayer, 1972]. Leaf nodes of the R*-tree contain index records, that consist of an approximation (minimum bounding box (mbb)) and an pointer to an indexed spatial object. Non-leaf nodes contain records that consist of a pointer to a child node and the mbb of that child. This structure is well suited for storage on secondary memory, because each node can be stored in a separate disk page. However, there has not been much research on how to store the R*-tree structure to secondary memory.

We focus on reducing the seek-time of a node request, when performing window querying on the tree. The principle that we used to obtain improvement is that nodes close in space should be close to each other on disc in order to reduce the seek-time. Resulting in the following hypothesis. It is possible to exploit the spatial relations between nodes to design and construct a (dynamic) R*-tree storage allocation algorithm that improves query processing by reducing seek time costs.

In order to evaluate the hypothesis we implemented several types of storage allocation algorithms, resulting in an equal number of indexing techniques.

Comparison of these indexing techniques consisted of performing experiments that consist of three phases; building and querying trees, and comparing results. The three types of algorithms that we used are: Simple algorithms (Random and Lowest Free), Level Order algorithms (Static and Dynamic), and Clustered Leaves algorithms (Dynamic, Static and Hilbert). Only the later two types use spatial relations. The first type is used for comparison.

For conducting experiments we used an existing R*-tree implementation in which we integrated the storage allocation algorithms. With these implementations and four different types of data sets, we build differently constructed types of R*-trees. On these trees we performed nine
query batches with different sized windows. Following, we compared the benchmark results from this building and querying of trees with different storage allocation algorithms.

In order to validate our hypothesis we need at least one storage allocation algorithm that uses spatial relations between nodes and improves query performance by reducing seek-time costs. We show by comparison of query performance and disk read times that the two dynamic Clustered Leaves algorithms can indeed improve query performance by using spatial relations between nodes. The dynamic variant has a better modification performance than the Hilbert variant and is therefore more suitable for fulfilling the hypothesis.

Window query performance is improved up to 160% compared to the algorithms that do not use spatial relations. However the cost of insertion and deletion of entries is increased up to 55% and the disk space occupied by the data structure is around 60% more.

Concluding, we showed that we indeed can exploit spatial relations between nodes to develop a (dynamic) R*-tree storage allocation algorithm that improves the query performance by reducing seek-time costs.
Contents

M. Sc. Thesis presentation ................................................................. i
Preface ................................................................................................. iii
Summary ............................................................................................... v
Contents ............................................................................................... vii
List of figures ....................................................................................... ix
List of tables .......................................................................................... ix

1 Introduction ....................................................................................... 1
  1.1 Objective ....................................................................................... 2
  1.2 Organization of the report ............................................................. 2

2 Theory ............................................................................................... 3
  2.1 Space filling curves ......................................................................... 3
  2.2 External memory ............................................................................. 4
  2.3 The R-tree ....................................................................................... 6
  2.4 R-tree improvements ...................................................................... 11
  2.5 Problem analysis and hypothesis .................................................. 13

3 Research method .............................................................................. 15
  3.1 Comparing indexing techniques ................................................... 15
  3.2 Comparison methods ..................................................................... 15
  3.3 Algorithms .................................................................................... 16
  3.4 Comparison criteria ...................................................................... 20

4 Experiments ...................................................................................... 25
  4.1 Development environment ........................................................... 25
  4.2 Assumptions .................................................................................. 25
  4.3 Preparation .................................................................................... 26
  4.4 Experiments ................................................................................... 32

5 Results .............................................................................................. 37
List of figures

Figure 1.1, Types of 2-dimensional range queries ................................................................. 2
Figure 2.1, Examples of space-filling curves ........................................................................3
Figure 2.2, Schematic drawing of a disk ................................................................................5
Figure 2.3, R*-tree with requested nodes by some query......................................................7
Figure 2.4, Two different R-trees on the same data set.......................................................9
Figure 5.1, Average time per query.................................................................................. 38
Figure 5.2, Average time per read node ........................................................................... 39
Figure 5.3, Average distance per query ............................................................................ 40
Figure 5.4, Average build time, normalized to fastest....................................................... 40
Figure 5.5, Average insert time per entry, normalized to fastest......................................... 41
Figure 5.6, Average delete time per entry, normalized to fastest......................................... 41
Figure 5.7, Average time per query, normalized to Dynamic Clustered Leaves ............... 43
Figure 5.8, Average insert time per entry ......................................................................... 44
Figure 5.9, Average delete time per entry ......................................................................... 44
Figure 5.10, Average build time ...................................................................................... 44
Figure 5.11, File utilization rate ....................................................................................... 44
Figure 5.12, Disk traffic .................................................................................................... 46
Figure 5.13, Average read time per node ......................................................................... 46
Figure 5.14, Average write time per node ......................................................................... 46
Figure 5.15, Average building buffer hit rate .................................................................... 48
Figure 5.16, Average query buffer hit rate ........................................................................ 48

List of tables

Table 3.1, Summary of storage allocation strategies ........................................................... 19
Table 3.2, Benchmark variables ........................................................................................ 22
Table 4.1, Characteristics data set, California ..................................................................... 29
Table 4.2, Characteristics of the data used for building trees ............................................. 30
1 Introduction

Over the last few years, there has been a great deal of interest in extending traditional, alphanumeric databases to handle multi-dimensional spatial data. Spatial data consists of points, lines, rectangles, regions, surfaces and volumes. The representation of such data is becoming increasingly important in applications in various computer areas, such as computer graphics, database management systems, computer-aided design, solid modeling, computer vision and robotics, geographic information systems (GISs), image processing, computational geometry, cartography and pattern recognition [Samet, 1989].

GISs are a rich source of important problems that require good use of external-memory techniques. These systems are used for scientific applications such as environmental impact, wildlife repopulation, epidemiology analysis, and earthquake studies and for commercial applications such as market analysis, facility location, distribution planning and mineral exploration [Haas, 1991].

In support of these applications, GISs store, manipulate and search through enormous amounts of spatial data [Cromp, 1993] [Laurini, 1992] [Samet, 1989] [Kreveld, 1995]. For example the GIS of NASA’s EOS project is expected to manipulate petabytes (thousands of terabytes) of data.

Many approaches for storing and using large amounts of data rely on the principle of hierarchical decomposition of space. This has led to the development of data-structures such as region quad trees [Finkel, 1974] and the B-tree [Bayer, 1972]. The B-tree clusters spatial objects based on their proximity. One of the advantages of the B-tree is a guaranteed 50% storage usage rate. These structures make it possible for the search for objects to focus at a high level towards the relevant regions. An example of such an index data structure is the R-tree [Guttman, 1984] and its variants, including the R*-tree [Beckman, 1990] and others [Kamel, 1994] [Sellis, 1987] [Greene, 1989], where the objects are arranged into a hierarchy of (hyper) rectangles. Another example is the Cell-tree [Gunther, 1986], 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. A survey of spatial data structures is given in [Samet, 1989] and [Samet, 1990].

The volume of (GIS) databases containing millions of data objects makes it necessary to store the index structure on secondary memory (i.e. disk). Storage on secondary memory is
significantly slower than on primary memory (RAM). In order to process queries quickly, an efficient mechanism that indexes spatial data objects according to their location in space is required.

Typical queries in spatial of databases are spatial join, point location, similar shape search, diagonal corner, 2-sided, 3-sided and general window query (Figure 1.1). For a more complete introduction on spatial databases, see [Guting, 1994].

![Figure 1.1, Types of 2-dimensional range queries.](image)

1.1 Objective

As mentioned in the paragraph above, it is very important to have an efficient indexing method. One of the most commonly used and best suited for most purposes is the R*-tree.

This structure has been intensely investigated. This researches primary focussed on the logical structure of the R*-tree, like proposing different heuristics for insertion, deletion and querying algorithms. However little attention has been devoted to the physical layout of the R*-tree nodes on disk.

The goal of this master thesis is to investigate and improve the storage allocation strategies for the R*-tree in order to improve the query performance (extensive analysis and hypothesis in paragraph 2.5).

1.2 Organization of the report

The second chapter (Theory), of this report introduces the reader to subjects that are critical for understanding this report. It also presents the analysis of the problem and the hypothesis. The third chapter (Research method) describes the methods and techniques used to test the hypothesis. The fourth chapter (Experiments) presents to the reader the proceedings of our research and experiments. The fifth chapter (Results) presents the results obtained in the experiments. The sixth chapter (Conclusions) contains the interpretations of the results, recommendations and suggestions for further research.
2 Theory

In this section we introduce various subjects that are important to this research. The following subjects are included. The first section contains an introducing to space-filling curves. This is followed by some general theory about storage allocation and external memory management. A description of the R-tree, its variants and improvements can be found afterwards. We conclude the chapter with an analysis of the problem and the definition of the hypothesis.

2.1 Space filling curves

A space-filling curve is a continuous mapping from a lower-dimensional space into a higher-dimensional one. A useful property of a space-filling curve is that it visits all the points in a $k$-dimensional grid exactly once and never crosses itself. Thus, points that are close together in the plane will tend to be close together in appearance along the curve. Space-filling curves in spatial environments are often used to order the objects in space. This is done by calculating a curve-value (key) for the objects, which is based on their coordinates. These keys are then used to establish an order among the objects.

In [Faloutsos, 1989] it was shown experimentally that the Hilbert curve achieves better clustering of spatial objects than other curves. Also, according to [Kamel, 1993], packing an R*-tree
by reordering objects based on calculated Hilbert keys of the center coordinates of minimum bounding boxes (mbbs) gives the better performance than with other curves or without ordering. Examples of space-filling curves are the Morton order, Cantor-diagonal order and the Hilbert curve (Figure 2.1). For further discussion of some of the properties of space filling curves, see [Patrick, 1968] [Butz, 1971] [Alexandrov 1979 and 1980] [Laurini, 1985].

2.2 External memory

The Input / Output communication (I/O) between the fast internal memory and the slow external memory (such as disk) can be a bottleneck in applications that process massive amounts of data [Gibson, 1996]. One approach to optimizing performance is to bypass the virtual memory system and to develop algorithms that explicitly manage data placement and movement, which we refer to as external memory algorithms (EM algorithms).

There are four techniques available for solving the performance problem of the slow external memory [Gibson, 1996]:

- Increasing storage device parallelism, to increase bandwidth between storage I/O devices and main memory
- More effective caching and reorganizing data to exploit locality
- Overlapping I/O with computation in order to reduce waiting time of the application
- More effective scheduling and reducing or rearranging the accesses made to data, possibly by changing the applications themselves.

In order to be effective, EM algorithm designers often have a simple but reasonably accurate model of the memory system's characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data is stored in concentric circles on the platters called tracks. To read or write a data item at a certain address on disk, the read/write head must mechanically seek the correct track and then wait for the desired address to pass by. The seek time to move from one random track to another is often on the order of 5-10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it pays to transfer a large collection of contiguous data items, called page. Track size is a parameter of the disk hardware and cannot be altered. For batch applications, the page size should be chosen to be a significant fraction of the track size or a small multiple of the track size, to better amortize seek time. For online applications, a smaller value is appropriate. Page size is not modifiable after formatting the disk. Formatting is time-consuming and remover all data from disk, thus it is usually done only once. Typical page sizes are between 1 and 8 KB. Disk pages are also referred to as disk blocks.
See Figure 2.2 for a schematic picture of an average disk.

Even if an application can structure its pattern of memory use to take full advantage of disk page transfer, there is still a substantial gap between internal and external memory performance. In fact the access gap is growing, since the speed of memory chips is increasing more quickly than disk bandwidth and disk latency.

EM algorithms are often designed using the parallel disk model (PDM) [Vitter, 1994]. This is a good model for theoretically developing and checking algorithms. More precise and complicated disk models than PDM, have been developed, such as the ones by [Ruemmler, 1994], [Shriver, 1998] and [Barve, 1997]. These latter ones distinguish between sequential reads and random reads and consider the effect on throughput of features such as disk buffer caches and shared buses.

![Figure 2.2, Schematic drawing of a disk.](image)
2.3 The R-tree

The R-tree [Guttman, 1984] is a height-balanced tree similar to the B*-tree [Bayer, 1972].
The B-tree is a balanced search tree, which stores multidimensional rectangles as complete objects
without clipping or transforming them. This is a good structure if much of the tree is on disk, since
the tree height and hence the number of disk accesses, can be kept small.

In the R-tree, leaf nodes contain index records of the form (l, tuple-id) where tuple-id
uniquely determines a tuple in the database and l 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 (l, child-pointer) where child-pointer refers to the address of a lower node in the
R-tree and l 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 ≤ M/2 and M
entries (the “node fill requirements”), with the possible exception of the root.

This results in the following R-tree properties:

• The root has at least two children unless it is a leaf
• Every non-leaf node has between m and M children unless it is the root
• Every leaf node contains between m and M entries unless it is the root
• All leaves appear on the same level

The R-tree performs best if M is best chosen so that the size of the node is equal to the disk
page of the file system.

2.3.1 Minimal bounding box

Since the objects stored in a spatial database can be rather complex, they are often
approximated by simpler objects and the index is built on these approximations. The most commonly
used approximation is the minimal bounding box, the smallest axes-aligned d-dimensional rectangle
that includes the whole object. In a two-dimensional case, the boxes are called minimal bounding
rectangles. The R-tree uses this approximation method.

2.3.2 R-tree querying

Window queries are one of the most commonly performed queries on data structures. A
window query on an R-tree is handled as follows. It 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. This means that when doing more than one query using the
tree, the higher-level nodes are requested more often than lower level ones. Usually it also means
that more nodes of lower levels are requested than higher level ones.

For example in the tree of Figure 2.3, the nodes requested by a given query are shown in
gray. The search starts at the top and descends to a node at a lower level if it fulfills the query
requirements. Multiple paths, like those shown in this Figure, are mostly traversed in depth-first
order.

![R*-tree with requested nodes (gray) by some query](image)

**Figure 2.3, R*-tree with requested nodes (gray) by some query**

### 2.3.3 R-tree and external memory

The R-tree is well suited for storage on external memory, because each node of the R-tree
can be stored in a separate disk page. To accomplish this, $M$ (the maximum of entries in one node)
is chosen in such a way that the size of one node is equal to the size of one disk page. In practice, it
is not likely that this can be accomplished exactly. Thus, $M$ is usually chosen so that the size of the
node is as big as possible, but smaller than or equal to the size of a disk page. Note that this results
in a small loss of space. This way of storage has some advantages, which are described in the
following.

When querying an R-tree, there are requests for nodes and of course only the nodes that
are actually required. Moreover since a node is stored in exactly in one disk page, only one read
operation is required, which is the smallest possible read operation from a disk. A note must be
made that there is some *useless* information retrieved. This consists of the entries in a node, that
are not requested by a (query) request. For relatively big query windows that overlap a lot of the
data space, most entries in a node are used. Repressing this problem by choosing $m$ bigger results
in a higher fill rate, but also causes entries to be less well-distributed in tree, which is a major
disadvantage.
Also worth mentioning is that nodes are not fully filled. Thus reading a disc page results in reading some empty entries that are also useless.

If we choose a different value for $M$ (and thus the size of the node), we lose some of the retrieval performance. Key concept here is that we don’t want to lose time for reading useless information and at the same time, we don’t want to read too much information, which slows the process. Of course, we chose the node size as a multiple of a disk page. Otherwise we would have a lower disk page rate, which causes a worse performance of the tree, because we read useless information.

Choosing $M$ bigger and thus (a multiple of the disk page size) means reading multiple disk pages. Thus finding a certain entry takes more time. On the contrary, it has the advantage that we read less useless information and that means less wasted read time.

Choosing a smaller value for the node size (less than one page) results in reading a complete disk page that is not completely filled with information.

Since the page size is a hardware factor and not changeable (only by formatting the disk), we have to choose $M$ so that the size of the node is (almost) as big as the disk page.

2.3.4 Modifying the R-tree

The R-tree can be updated dynamically by insertion and deletion of data objects. We briefly describe here how this is done.

The insertion algorithm is composed of two sub-algorithms in which crucial decisions for good retrieval performance are made. The first sub-algorithm is known as ChooseSubtree. This algorithm begins a search in the root node, descends to an appropriate child node, until it reaches the most suitable leaf node, where the new entry will be placed. The second sub-algorithm is known as Split. It is called if ChooseSubtree ends in a node filled with the maximum number of entries $M$. Split then distributes $M+1$ rectangles into two nodes containing not less than $m$ and not more than $M$ entries each. This split is done by on minimizing the overlap of the two nodes during node splitting. Alas this is not achieved well by the R-tree. An improved has been achieved by the development of the R*-tree [Beckmann, 1990], which incorporates a combined optimization of area, margin and overlap during splitting (see paragraph 2.4.1).

The deletion algorithm simply deletes the entries from the leaf node in which it is stored. This occasionally causes an underflow of the node. In other words, the number of entries in the node is smaller than $m$. These entries will be re-inserted in the tree after the underflow node is deleted. More details of the insertion and deletion algorithm can be found in [Guttman, 1984].
Remember that the bounding box of higher level nodes have to be updated after insertion or deletion of entries.

It is not possible to exactly predict how an R-tree will look like after building it for a given data set. This unpredictable behavior is caused by calculations made during the insertion and deletion of entries. These are dependent on entries in the nodes and determine the location of entries in the tree. Thus, the order in which the data entries are inserted matters. In Figure 2.4, we present two examples of different R-trees that could have been built by inserting the nodes in a different order. Note that they contain the same information.

Figure 2.4a and 2.4b, Two different R-trees on the same data set.

2.3.5 Disk allocation behavior

When inserting and deleting nodes, the R-tree has a particular behavior of allocating and de-allocating nodes. Deletion of nodes means clearing disk pages. Insertion causes the allocation of new or free pages. First, we describe the insertion behavior, followed by a brief description of the deletion.
2.3.5.1 Insertion

The R-tree algorithm searches for the “best” leaf node to insert the new entry. When this causes an overflow of the node, the algorithm has to split it and redistribute the entries over the old and a new allocated node. The new node will have to be represented in the parent node by a freshly created entry. Obviously, this also may cause an overflow in the parent node, thus causing another split. This process may cascade up until the root node.

When the root splits, the algorithm behaves in slightly different manner. It does distribute the entries over itself and a new allocated node, which has the same level as the root. However, this causes a problem, since there can be only one root node. Therefore, a new root node is created, which has two children; the former root and the other newly allocated node. This causes the height of the tree to be increased.

Node splitting occurs more often at lower levels of the tree. This means that when we do a bulk insertion of entries, the tree will require more nodes of lower levels. Thus whenever an R-tree requests a new disk page, chances are that it will be used for leaf nodes rather than higher level nodes.

Concluding, we can say that there is no predictable pattern for the level of the requested new node when doing insertions. The requests for new nodes and pages are often handled by a simple storage allocation algorithm, which just chooses a disk page using some simple rule. For example: choose the last page freed by some earlier deletion operation or, if none is available, a page at the end of the file.

2.3.5.2 Deletion behavior

When deleting a node from a tree, this will occasionally cause an underflow in a node. The remaining entries will be re-inserted in the tree after the deletion and the node will be freed. This will cause intervals of free pages to occur in the R-tree file. Continuous deletion will cause more or bigger holes and nodes spread across the file, separated by gaps. This, of course, increases the seek-time for a requested node, since they are located far from each other. The removal of these gaps can be done by reorganizing and reallocating nodes. In a dynamic environment, this is not possible because it is too time-consuming.

No study of this de-allocation behavior has been conducted. This is awkward because it is apparent that the policy used for the allocation/de-allocation of nodes influences the seek time of reading a node considerably and thus influences the query performance of the tree.
2.4 **R-tree improvements**

In this section we present an overview of several research efforts that have been spent in trying to improve several aspects of the original R-tree data structure. We focus on the most commonly used R-tree variant, R*-tree.

### 2.4.1 R-tree variants

The R*-tree [Beckman, 1990] is one of the numerous variants of the R-tree [Beckman, 1990]. Some other variants are the Hilbert R-tree [Kamel, 1994], the R+-tree [Sellis, 1987] and Green’s R-tree [Greene, 1989]. Main idea of the variants is the following. The R-tree is a dynamic structure. Thus, all approaches of optimizing the retrieval performance have to be applied during building and modifying the tree. They focus on how to improve the insertion heuristics of the original R-tree.

The Hilbert R-tree orders entries according to the Hilbert-key of its mbb center and a B-tree is build in a bottom-up manner on the totally ordered entries. Bulk loading is improved considerably, but on the contrary, query performance is worse than the R*-tree.

The R+-tree variants changes the insertion and deletion heuristics of the R-tree in such a way that at any time all nodes of one level are disjoint.

Compared to the original R-tree, the Green's variant has only a different split algorithm. This algorithm searches for an axis in the node that splits and thus divides the entries into two parts.

Like other R-tree variants, the R*-tree has an improved version of the splitting algorithm, which incorporates a combined optimization of area, margin and overlap. Also there is the use of the concept of forced re-insertion, which is analog to the deferred-splitting in B-trees [Bayer, 1972]. When a node overflows and splits, some of its children are carefully chosen and they are deleted from their current position and re-inserted. This usually results in an R-tree with a better structure, where the re-inserted nodes have a better place in the tree and fewer splits of nodes occur.

Experiments by [Beckmann, 1990] indicate that optimum retrieval performance of the R*-tree is achieved when \( m \) is chosen around 40% of \( M \). These experiments also showed that the average fill rate of a node is around 70%.

The R*-tree seems to have a better performance than other variants [Vitter, 1999].

### 2.4.2 Packed R-Trees

The approaches of [Gavrilla, 1994] and [Kamel, 1993 and 1994] are attempts to achieve a reduction of the number of nodes requested from disc by packing the R*-tree. This means putting the maximum amount of entries in a node, so that the tree will have fewer nodes. In order to be able to pack an R*-tree it is usually assumed that the data is static. In other words, the indexed data is not subject to modifications. This is justified since the insertion or deletion of an entry would cause
either a split or a node that is not completely filled, thus violating the packing criteria. Coping with this problem would probably bring severe performance loss.

The work by Gavrilla [Gavrilla, 1994] is based on packing the tree by iterative minimizing an optimization formula, that captures the goodness of groupings of spatial objects. Kamel and Faloutsos [Kamel, 1993] base their packing criteria on ordering the entries using space-filling curves. They show that the Hilbert curve yields a better query performance. They also made a dynamic version of their packing algorithm [Kamel, 1994] based on a Hilbert curve, which makes the insertion algorithm similar to the B-trees algorithm. However, the performance of the dynamic version is not very much better than the original R*-tree. In both researches, it is assumed that the time needed for querying is equivalent to the number of nodes requested.

It should be mentioned that Gavrilla probably stores the nodes in level order, with the highest level first. This is not explicitly stated but we assume that the nodes are stored in the same order used to construct them. It is also reasonable to assume that Kamel stores the nodes of the static version in curve order.

2.4.3 Other data structures

Another approach is to improve the external memory management by externalizing internal memory algorithms. Some good internal memory solutions exist for 2-dimensional range searching. The priority search tree [McCreight, 1985] is an example. There have been some attempts to make this structure external [Blankenagel, 1990] [Icking, 1987] [Ramaswamy, 1994], but still the performance is non-optimal. In [Subramanian, 1995] a structure called p-range tree is developed. The structure supports 3-sided queries and 2-dimensional queries, and a static structure for 3-dimensional queries has been developed [Vengroff, 1996]. The segment tree [Bentley, 1977] has been externalized too [Blankenagel, 1994] [Ramaswamy, 1994] but it uses more than linear space in memory.

In [Kanellakis, 1993] the dynamic interval management problem is considered, in which intervals of external memory can be inserted and deleted and, given a query interval, all current intervals that intersect it must be reported. This proves to be an I/O optimal data structure for several applications involving batched dynamic problems, including bulk loading of B-trees [Arge, 1996] and R-trees [Arge, 1999].

In [Brinkhoff, 1994] several heuristics for globally clustering data objects on disc are presented and it is shown that a significant performance improvement can be obtained by reducing disc seek time. For example, spatial join queries were processed up to 4 times as quickly, whereas window queries were sped up even more.

Other work by [Arge, 1994] and [van den Bercken, 1997] is based on the buffer tree. Most improvement in these approaches is due to improving the buffering and by managing external
memory in intervals. Specific applications are bulk loading 2-dimensional data, dynamic batched updates and queries.

For a recent survey on external memory algorithm and data structures dealing with massive data, see [Vitter, 1999].

2.5 Problem analysis and hypothesis

Several performance problems in GISs can be attributed to inefficient management of disk storage allocation. These problems are more apparent when performing spatial queries, which usually entail the use of spatial indexing structures. In this project we chose to study the disc storage management problem as it affects the processing of the R*-tree, one of the most popular spatial indexing methods used in GISs and spatial databases.

Significant research has already been done on improving several aspects of the processing of R*-trees. The focus of that researches has been on changing the insertion and deletion methods of the R*-tree to reorder the entries in the tree in order to achieve query performance improvement. In this project we focus on an aspect that has received little attention, namely, the way that the R*-tree is stored on disc. We focus on reducing the seek-time of a node request, when doing querying on the tree. In other words we want to apply the second point of improvement techniques [Gibson, 1996] (paragraph 2.2) to the R*-tree. This means using more effective caching and reorganizing data to exploit locality, in order to improve the performance of the R*-tree.

A key observation about the behavior of spatial access methods will help us in our efforts: Once a certain datum has been retrieved from disc in response to a query, another datum which is spatially close is more likely to be requested next than a datum lying far away.

Thus, if a query can cause the read of multiple nodes (pages) from disc, then these nodes should be as close as possible to each other. In other words, nodes close in space should be close to each other on disc in order to reduce the seek-time.

We would like to apply this rationale to an R*-tree implementation in such a way that it is able to work well in a dynamic environment. This is a challenging problem, because although we would like the R*-tree to have good performance even when insertion and deletion operations take place, the reorganization of nodes tends to be time-consuming.

2.5.1 Hypothesis

Summarizing from the previously described problem analyses, we formulate the following hypothesis.

It is possible to exploit the spatial relations between nodes to design and construct a (dynamic) R*-tree storage allocation algorithm that improves query processing by reducing seek time costs.
3 Research method

The research methodologies we have used in order to examine the hypothesis are presented in this section. Firstly, we present some general theory about how to compare indexing techniques. Next, we speculate about several methods, which can be used in our specific research. After this, we present the storage allocation algorithms that were included in our project. We conclude by explaining the criteria used to compare these algorithms.

3.1 Comparing indexing techniques

An index is a data structure that identifies the location at which indexed objects occur. All kinds of indexing are associated with query evaluation algorithms that access this information and update algorithms that maintain it. Therefore, when evaluating an indexing technique, we must consider not only the data structure, but rather the structure in conjunction with the necessary algorithms. In our research, we focus on these algorithms.

There are four principal ways to compare index techniques: by direct argument, by mathematical modeling, by simulation, and by experiment [Zobel, 1996]. It is common to use at least two of these methodologies.

In order to compare indexing techniques, we need criteria that consider overall speed, space requirements, CPU time, memory requirements, measures of disk traffic, ease of index construction, maintenance in the presence of additions, modifications, and deletions; applicability, extensibility, scalability, implications for concurrency, transactions, and recoverability.

Also very important when we compare indexing techniques is to prevent some commonly made mistakes. Most of these are founded on making wrong decisions, using wrong material, conducting false logic and making wrong assumptions. We will try to avoid making these mistakes.

3.2 Comparison methods

The three phases we used for comparison by experiment are: building, querying and comparing. We applied this method to each of our algorithms separately. We call this the algorithm benchmark cycle, which can be summarized as follows:

---

1 Most of the material in this section was taken from [Zobel, 1996], which contains guidelines for presenting and comparing indexing techniques.
Step 1. **Building trees.** Constructing trees and gathering building benchmark information.

Step 2. **Querying.** Querying these trees and gathering query benchmark information.

Step 3. **Comparing.** Comparing benchmark results to that of other already finished algorithms.

We implemented several storage allocation algorithms that have the potential for fulfilling our hypothesis. Each implemented algorithm corresponds to an indexing method, which is then compared against the others.

### 3.3 Algorithms

We briefly describe the storage allocation algorithms that were implemented and benchmarked. When we talk about allocating a new node, we mean that the implementation chooses some disk page where it will store the information of the node. When an algorithm removes or deletes a node, it clears the disk page by marking it free in a list of free nodes. In other words, the page remains part of the file. For more information about the list of free nodes, see paragraph 4.3.1.

#### 3.3.1 Simple

First, we present two simple storage allocation algorithms, which we will refer to henceforth as *Random* and *Lowest Free*, respectively.

These algorithms do not take advantage of the fact that we are dealing with spatial data. We implemented these algorithms because we would like to see how a simple principle (Lowest Free) of the storage allocation algorithm would effect the query performance.

We used these algorithms as a bottom line for the performance and benchmark results. We compare the other algorithms against them. We expect to see improvement by using other more sophisticated algorithms.

##### 3.3.1.1 Random

This storage allocation algorithm has a very simple concept. When requesting a new node (insertion) and thus a new disk page, it simply allocates it at the end of the file, when there is no space in the tree’s file. If there is space it randomly allocates one of the free pages within the file. When a node has to be deleted, it simply does this without following further policies.

The random algorithm is the simplest and less sophisticated we could think of. It does not take advantage of the fact that we are dealing with spatial data.

##### 3.3.1.2 Lowest Free

The basic idea of the algorithm is the following. When there is space in the file and a request for a new node (page) is made, it allocates the free node with the lowest file offset address, which is
as close as possible to the beginning of the file. When there is no space, the algorithm allocates a new node at the end of the file. When deleting a node it just deletes the node (page) by clearing it. No further policies are followed.

3.3.2 Level Order

We shall call Level Order two variants of an algorithm where the nodes of the R*-tree are stored according to a breadth-first traversal of the structure. In other words, store nodes of the highest level in the beginning of the file, followed by the nodes of one level lower until the lowest level (leaf) nodes are stored.

This “ideal” situation is impossible to maintain in a dynamic environment when nodes are inserted and deleted. Remember the disk allocation behavior of the R*-tree when allocating new nodes (paragraph 2.3.6). This, combined with the fact that the algorithm has to allocate new nodes at the end of the file, makes it obvious that it can not be maintained without some degree of reorganization.

We implemented two variants for comparison reasons, one suitable for use in a dynamic environment and one for a static environment. Improvement of the query performance of the static variant gives us an indication that it is worth investigating a dynamic version that reorganizes nodes. We implemented a dynamic version without reorganization.

The dynamic variant works as follows. When there is space in the file and we need a new page, it tries to maintain the level order of the nodes as far as it can. When there is no space in the file, a new node is allocated at the end of the file. When deleting a node it simply deletes it, without further actions.

The static variant has exactly the same behavior. Only after insertion and deletion of all the entries (and nodes) this is followed by a reorganization of all the nodes. So this creates a file with the “ideal” allocation order. Note that it also removes any gaps that might exist in the file by moving these to the end of the file.

The main reason which convinced us to implement this algorithm is that it was used in a previous work on Packed R-trees [Gavrilla, 1994]. The author of that work did not elaborate on the reasons, which led him to choose this particular layout nor does he present any proof that it is better than any other. One possible reason is that since a query is processed typically by first reading the root node and then retrieving nodes at successively lower levels, then the disk head would start at the beginning of the file and move in only one direction, which might lead to a good query performance.
3.3.3 Clustered Leaves

We call *Clustered Leaves* the approach used in three algorithm variants where leaf nodes are kept close to their parent nodes. The rationale behind this concept is that while processing typical window queries, leaf nodes are more often requested than nodes lying at higher levels (paragraph 2.3.3), especially for large window sizes. Therefore, we would like to keep nodes of the leaf level together on disk. This is based on the principle of storing nodes together that are close together in space. Recall the way queries are performed, it is likely that a query causes reading of several leaf nodes that belong to one parent (see paragraph 2.3.2). If those nodes are stored close together on disk, reading them requires less seek time.

Of course, this policy requires some technique for reorganizing nodes in a dynamic environment. To keep this reorganization as cheap as possible, we only organize leaf nodes and do not take this concept any further higher up the tree.

We may infer from this description that the created index file will contain gaps, since it often has to move all children nodes of one parent node. Therefore, we decided to implement another variant that removes these gaps. Since we do not expect this variant to be suitable for a dynamic environment, we built a static version of this one.

An insertion or deletion of one leaf node may cause the reorganization of all children nodes of one parent, because children are or can not be stored in a sequence. Reorganization consists of placing the nodes together in one interval somewhere in the file. Details of this reorganization process can be found in the pseudo code in Appendix E.

When we are performing reorganization, we can also order these children nodes using a space-filling curve. Thus, we decided to implement a Hilbert curve variant as well.

Summarizing, we constructed three versions of this algorithm.

*Dynamic variant*, without specific order of sibling leaf nodes.

*Static variant*, same as Dynamic variant, but with gap removal after total creation of a tree.

*Hilbert variant*, dynamic variant with Hilbert-curve ordered sibling leaf nodes of one parent.

3.3.4 Other algorithms

There are many storage allocation strategies, which we did not consider in our investigation. For example, we could implement a list of free nodes and apply a Last In First Out (LIFO) principle for getting free nodes. Instead of LIFO we could also use a First In First Out (FIFO) principle, or similar variations.

We did not include this type of algorithms in our research, because their allocation behavior is not founded on spatial relations, which is a necessary property of our hypothesis. Therefore this type can not be used to prove our hypothesis. In addition, we presume that their behavior is almost
similar to the simple algorithms that we described before, because they also don't use spatial
relations.

One principle that was not used on most algorithms is the use of space filling curves. We did
not do this, because we assume that its use would make the insertion and deletion very slow. It
always needs a lot of reorganizing of nodes on disk, which would make it not usable in a dynamic
environment. However, we implemented one algorithm with the Clustered Leaves principle, because
it already uses a lot of reorganization.

In addition, static variants could use space filling curves, because they have to
reorganization of nodes too. We did not do this, because we are interested in static algorithms, see
our hypothesis. Although, it might be well worth to research the use of space filling curves in a static
environment, which we recommend for further research.

3.3.5 Summary

The algorithms were implemented in the presented order. Details of the used algorithms can
be found in Appendix E.

A summary of behavior of the algorithms is given in the following table. We left out the static
variants of the Level Order and Clustered Leaves algorithms, because they behave the same as
their dynamic variants. They only perform reorganization after building of the whole tree. The Hilbert
variant of the Clustered Leaves algorithm behaves the same as the Dynamic variant, but additionally
orders nodes accordingly to Hilbert-curve.

Table 3.1, Summary of storage allocation strategies

<table>
<thead>
<tr>
<th></th>
<th>Free page(s) in file</th>
<th>No free page(s) in file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Random</td>
<td>At end of file</td>
</tr>
<tr>
<td>Deletion</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lowest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Lowest file-offset address</td>
<td>At end of file</td>
</tr>
<tr>
<td>Deletion</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level Order</td>
<td>Try to maintain level order and allocate most favorable</td>
<td>At end of file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clustered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leaves</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Leaf: Place together with siblings</td>
<td>Leaf: Place together with siblings</td>
</tr>
<tr>
<td></td>
<td>Higher level: Random</td>
<td>Higher level: At end of file</td>
</tr>
<tr>
<td>Deletion</td>
<td>Leaf: Delete and place remaining leaf nodes/pages together</td>
<td>Higher level: Just delete page</td>
</tr>
<tr>
<td></td>
<td>Higher level:</td>
<td></td>
</tr>
</tbody>
</table>
3.4 Comparison criteria

Our benchmarking is divided in two parts: tree building and querying. We are interested in how algorithms affect construction and modification of a tree, because we want to be sure that it can be used in a dynamic environment. On the other hand, we are also interested in how our algorithms affect the query performance.

3.4.1 Benchmark topics and variables

We describe next several important index characteristics and related variables that are taken into account in our benchmarking process. A summary of the used benchmark variables can be found in table 3.2.

3.4.1.1 Query evaluation speed

It is clear that we would like to develop a storage allocation algorithm that will lead to a better query performance. We are especially interested in reducing the total time needed to process a given query. Therefore, we monitored the following benchmark variables in the query part of benchmarking:

- **Average time per query.** This gives us an indication of the query evaluation speed enhancement due to the various storage allocation algorithms.
- **Average distance between read nodes per query.** This measures the absolute and accumulated block address difference between each disk block read operation and the next. The distance is counted by our software and is not the real distance that the disk head travels. However, it should give us a rough estimate of this parameter.

3.4.1.2 Index construction, insertion and deletion

Another important issue for our research is how a storage allocation algorithm will affect the index performance for insertion and deletion operations. Remember that these operations have to be reasonably fast since we aim to use the implementation in a dynamic environment. A slower insertion/deletion performance is acceptable only if query performance has improved considerably.

We included the following variables in the construction benchmarking:

- **Overall time.** This measures the time needed for constructing a tree.
- **Average insertion time of one entry.** Measures the average time taken by the implementation for inserting one entry.
- **Average deletion time of one entry.** Measures the average time taken by the implementation for deleting one entry.
3.4.1.3 Disk space

Most of the algorithms affect the total size of the index file considerably. Since we do not want to increase the file size out of proportion, we monitor the size of the file. Some increase of the file size is acceptable when it speeds up insertion, deletion or querying.

Used benchmark variable, during construction:

- *File utilization ratio*. This is calculated at the end of the construction of an index tree. It is calculated by dividing the amount of free nodes by the amount of allocated nodes.

3.4.1.4 Disk traffic

The storage allocation algorithm affects the number of pages read and written during the building phase, but not during the query phase. This is explained by the fact that the algorithms affect where the nodes are stored in disk, but not the overall data or the shape of trees. However, we counted the number of nodes read during query phase anyway, because we use this value for computing other benchmark variables, such as the *average distance traveled by disk head per read*. Of course, during the query phase no nodes are written.

Related benchmark variables:

- *Nodes read*. The number of nodes that are read from disk.
- *Nodes written*. The number of nodes that are written to disk.
- *Average time per read*. This actually measures the time needed for reading one node from disk.
- *Average time per write*. Measures the time needed for writing a node to disk.

(Note, during the query tests we do not monitor disk write variables.)

3.4.1.5 Buffer

Our implementation uses a buffer for nodes (see section 4.2.1). We monitor how many disk reads and writes it prevents. Although we do not alter the buffer policies or algorithms, the buffer efficiency impacts the overall index performance.

We benchmark the following variables during both the construction and query phases:

- *Buffer hits*. The number of nodes requested that we were still in memory and did not have to be retrieved from disk.
- *Buffer misses*. The number of nodes requested that had to be retrieved from disk.
Table 3.2, Benchmark variables

<table>
<thead>
<tr>
<th>Topic</th>
<th>During construction</th>
<th>During querying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query speed</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Construction, insertion and deletion</td>
<td>Overall construction time</td>
<td>• Time per query</td>
</tr>
<tr>
<td></td>
<td>• Insertion time per entry</td>
<td>• Distance per read node per query</td>
</tr>
<tr>
<td></td>
<td>• Deletion time per entry</td>
<td></td>
</tr>
<tr>
<td>Disk space</td>
<td>• File utilization rate</td>
<td>-</td>
</tr>
<tr>
<td>Disk traffic</td>
<td>• # Read nodes</td>
<td>• # Read nodes</td>
</tr>
<tr>
<td></td>
<td>• Average time per read node</td>
<td>• Average time per read node</td>
</tr>
<tr>
<td></td>
<td>• # Written nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Average time per written node</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>• Buffer hits</td>
<td>• Buffer hits</td>
</tr>
<tr>
<td></td>
<td>• Buffer misses</td>
<td>• Buffer misses</td>
</tr>
</tbody>
</table>

3.4.2 Other considerations

The list of benchmark variables described above is relatively short, but should be adequate for our needs. Other parameters could also be scrutinized and/or benchmarked, but we believe that doing so would be either too costly or irrelevant. Some issues, however, were taken into consideration during the programming of our implementations. These are discussed below.

3.4.2.1 Applicability and extensibility

The applicability and extensibility of the final implementation cannot be affected by the implementation of storage allocation algorithms. In other words, the same queries should remain supported and the index should remain capable of holding the same sorts of data.

Still, we want to mention that in our research we only benchmarked the performing of window queries. Although other query types could have been investigated, we believe that window queries are very typical of the searching operations performed with R-trees. Nevertheless, better performance achieved obtained for window queries does not imply improved performance for other types of queries.

3.4.2.2 Scalability

Introduction of storage allocation algorithms to the R*-tree should not change scalability. In other words, changes to the R*-tree should never affect the amount of data the R*-tree can handle.
3.4.2.3 Implications for concurrency, transactions and recoverability

Obviously, we do not intend to influence these by our modifications. We presume that this has been correctly implemented in the provided implementation. We did not affect this by our new storage allocation algorithms.

However, we would like to note that concurrent access to a disc (or database) which our R*-tree implementation uses, will affect the performance of our algorithms, because the disc head is moved by other processes and therefore undoes the achieved seek-time reduction. We recommend this for future work.

3.4.2.4 CPU time

We did not benchmark the CPU time specifically, because we could compose this by subtracting the total I/O time from the total used time, variables that are already included in the benchmark.

3.4.2.5 Memory requirements

We did not expect differences in memory use by our different algorithms. We could benchmark this, but we did not, because the algorithms do not use the memory differently. They all use the same buffer (equally sized) and the same program has to be loaded into memory since we implemented all algorithms into one program. Another reason why we did not benchmark this is that the buffer of the implementation is the biggest consumer of memory and not the rest of the implementation that includes our storage allocation algorithms.
4 Experiments

In this section we describe the proceedings and details of the experiments, its environment, setbacks and unexpected luck. In other words, we explain how we applied our research methods in practice and what was the outcome.

We start with an overview of used hardware and software. This is followed by some assumptions we had to make before we could build implementations and do experiments. After this, we describe how we implemented the storage allocation algorithms and some support programs. This is followed by the proceedings of the benchmark experiments, setbacks and unexpected luck.

4.1 Development environment

Before jumping into the details of our experiments, we give a brief overview of the hardware and software that we used.

All experiments have been conducted on an IBM Personal Computer 300GL, with an Intel Pentium II 300MHz processor, 64 MB of RAM, a 3GB hard disk. System benchmark results can be found in Appendix C.

The experiments were conducted under the Linux operating system, with the Second Extended File System (Ext2 FS).

We used the available R*-tree implementation by Claudio Esperança for implementing the storage allocation algorithms. The original implementation used a storage allocation algorithm which can be described as a Last In First Out (LIFO) algorithm. As noted before in paragraph 3.3.4, we did not include this algorithm in our research. The implementation also uses a node buffer algorithm, which uses a Least Used First Out (LUFO) principle.

The code was written in C++ and developed with the GNU C++ compiler and debugger. The code should be portable to other UNIX-like environments or any platform which supports a standard C++ compiler.

4.2 Assumptions

4.2.1 Node buffering

The original implementation uses a buffer for nodes, which is allocated in memory. It reads nodes from disk and stores them in the buffer when they are requested and not already in it. When a
node is modified it will be marked dirty. When the buffer is full it removes the least used node and thus frees a position in the buffer for another node. Only if the removed node is dirty it will be written to disk. On closing of the buffer it writes all dirty nodes to disk.

Of course, the buffering algorithm is very important to our research, since it influences the number of nodes read from and written to disk. Our research did not contemplate any investigation about how nodes can be buffered optimally in order to minimize the number of disk access operations. This topic, at least with respect to use in spatial indices, is still sparse and deserves to be further investigated in the future.

We had to use a single buffer size for use in all experiments so that the results would not be skewed towards any storage allocation technique. So we decided to use a 1Mb buffer which, in our opinion, is a reasonable buffer size and the implementation can perform well with it. In paragraph 5.2.5 we show that the buffer hit rate is between 83 and 98 percent during querying and building of trees. During our experiments, the size of a disk page was 4KB, so the buffer can store 256 nodes in the buffer.

4.2.2 Disk model

We used a model for the disk that we used during our research. There is only one disk. We assume that our files will be written to disk in one consecutive interval of disk pages on one platter. Of course this not realistic, files will be stored distributed over several intervals and platters. This depends on the OS and we have to trust that it does this correctly and as fast as possible.

In particular, it should be noted that our way of estimating the distance traveled by the disk head is not realistic. Nevertheless, it should give an indication of the real head movement, when we compare algorithms.

4.3 Preparation

In preparation to our experiments, we had to implement several components, such as the storage algorithms, benchmarking programs and to modify the original implementation accordingly.

4.3.1 Interval list structure

Before implementing our algorithms, we found that we needed a data structure that keeps track node addresses and their level.

The need for such a fast data structure becomes clear by the following example. The Level Order algorithm needs to know where the free places in the file are and where the sequences of nodes of one level are stored in order to place new node. This information about free nodes and the level of allocated nodes is requested often. It must be quickly accessible in order to permit the
implementation to work in a dynamic environment. Storing this information on disk is clearly inadequate, so it has to be stored in memory.

The original implementation did not supply fast access to the address and level of stored and free nodes. It only stores the address of the next free node in a list. Therefore, we decided to implement a list of intervals entries, as attribute of the R*-tree. An interval entry contains the number of the first and last node of sequentially stored nodes that have the same level. We include both allocated and free nodes in the list. Free nodes intervals are represented by a negative value of its level.

The original implementation uses a data structure that is stored on disk. In our new implementations, it is resident in memory. Obviously, the size of the list is dependent on the number of nodes that (once) have been allocated and thus it is variable in time.

The size is not big if we compare it to the total amount of memory used in the implementation. In a worst case situation the size of the list can be pretty large but in practice this is unlikely to happen. During our experiments the memory use of the R*-tree implementation was around 1.5MB. The memory size of the interval list was maximal 3.5KB during our experiments.

The interval list is written to disk in case the tree is closed. When a tree is opened, the list is read from disk.

Details of the Interval list data structure can be found in Appendix D.

4.3.2 Index-building algorithms

We chose to implement all algorithms as part of one implementation. This has the following advantages over the other option of compiling separate implementations. It provides the possibility to change the storage allocation algorithm of a tree at run-time. It also makes the benchmarking program and process simpler. A disadvantage is that, since all algorithms are included, the compiled program is slightly bigger and uses more memory. Nevertheless, we found the advantages more valuable so went for this option.

We implemented the algorithm is the following order: Random, Lowest Free, Level Order and Clustered Leaves. Note that after implementing an algorithm, we benchmarked and compared it to previous ones. This gave us an indication of whether we were on the right track and if we should pursue other variations of the algorithm.

Pseudo code of the algorithms can be found in Appendix E.
4.3.3 Creating data

One part of our experiments consisted of building R*-tree indices for some spatial data sets. We had to choose data sets which could reasonably represent real data encountered in typical applications. This posed a problem, because there are no widely accepted typical data sets.

We decided to use three artificially created data sets and one real data set. We chose the following artificial data distributions: Cluster, Gaussian and Uniform. These are chosen because they are commonly used in other research, such as [Beckmann, 1990], [Kamel, 1994] and [Gavrilla, 1994]. Moreover in these works, these distribution have shown that they behave reasonably similar compared to real data sets and are therefore useful for comparing indexing techniques. The Cluster distribution contains data objects that have centers that follow a uniform distribution within several sub data space of the whole. The Gaussian distribution has objects whose centers follow an independent Gaussian distribution across the data space. The Uniform data set has objects whose centers follow an independent uniform distribution.

We did not create the complete artificial data sets, but only the corresponding minimum bounding boxes (mbbs), since only these are required in order to test the R*-tree. Similarly, when we refer to the “real” data set, we mean a collection of mbbs of objects that represent real world data.

The number of objects in each artificial data set is equal to the number of objects in the real data set, i.e., approximately 130,000. We did this, in order to be able to compare the performance of the algorithm for each data set. We also kept other characteristics as equal as possible to the real data (size of the mbbs, size of the data space, etc).

The real data set that we used consists of a set of line objects, from an area of California. The data is drawn from the TIGER/Line files used by the US Bureau of the Census [Census, 1989]. Some characteristics can be found in table 4.1.

We implemented a program that provides us with the previously described data sets, which consisted of mbbs stored in separate files. We used these data files later for building and modifying trees.

We chose to store the data sets on disk to be sure that they are not changed during the whole benchmark process. We could also have used the pseudo-random generator of the C++ library, which produces static lists of randomly generated numbers. But since it is most likely that in real-life use the data is also inserted by reading data files, we chose this approach.

Because of the way trees are built, we decided to divide the data sets into two files per distribution. We did this, because for all trees we need an initial data set to insert, followed by a data set used in further insertions (after a portion of the original data is deleted). See paragraph 4.3.4 for details about the several types of tree-building procedures.
The artificially created mbbs are generated by choosing its size from a Gaussian distribution, where \( \mu \) that is the same as the average size of the mbbs of the real data set. Coordinates of all mbbs are chosen within the data space of the real data set.

We need query windows for selecting the entries that are to be removed. We created two files, one with windows that are used for bulk deletion, and another for smaller deletions.

We implemented a program called *buildmbbfiles*. It needs a (real) data set, from which it calculates values for the artificial data sets. With this, it builds for all four distributions two files with mbbs used for insertion. It also creates two files with windows that are used for deletion.

In table 4.2 we present details of the files constructed by the *buildmbbfiles* program.

<table>
<thead>
<tr>
<th><strong>Table 4.1. Characteristics data set, California</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Square size of total data space</td>
</tr>
<tr>
<td>Minimum Bounding Box of total data space</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Dimensions</td>
</tr>
<tr>
<td>Number of objects</td>
</tr>
<tr>
<td>Average object size</td>
</tr>
<tr>
<td>Object size relative to total data space</td>
</tr>
</tbody>
</table>
Table 4.2. Characteristics of the data used for building trees

<table>
<thead>
<tr>
<th>File description</th>
<th># of entries</th>
<th>Distribution of entry positions</th>
<th>Square size of mbbs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster insert entries 1</td>
<td>85980 (66,6%)</td>
<td>171 clusters of 500 entries. (Entries are uniformly distributed inside the cluster space. Cluster size is ~1% of square size of total data space.)</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Cluster insert entries 2</td>
<td>42990 (33,3%)</td>
<td>85 clusters of 500 entries. (Entries are uniformly distributed inside the cluster space. Cluster size is ~1% of square size of total data space.)</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Gaussian insert entries 1</td>
<td>85980 (66,6%)</td>
<td>Gaussian</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Gaussian insert entries 2</td>
<td>42990 (33,3%)</td>
<td>Gaussian</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Real data insert entries 1</td>
<td>85980 (66,6%)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Real data insert entries 2</td>
<td>42990 (33,3%)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Uniform insert entries 1</td>
<td>85980 (66,6%)</td>
<td>Uniform</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Uniform insert entries 2</td>
<td>42990 (33,3%)</td>
<td>Uniform</td>
<td>~ 0,0001675%</td>
</tr>
<tr>
<td>Deletion windows 1</td>
<td>8598 (10%)</td>
<td>Uniform within data space of real data set.</td>
<td>~2,0%</td>
</tr>
<tr>
<td>Deletion windows 2</td>
<td>8598 (10%)</td>
<td>Uniform within data space of real data set.</td>
<td>~0,5%</td>
</tr>
</tbody>
</table>

4.3.4 Tree-building procedures

We would like our experiments to mimic typical usage of R*-tree indices. Unfortunately, there is no generally accepted standard for index usage pattern.

With respect to the tree construction procedure, we are especially interested in how our implementation performs when inserting and deleting entries. For this purpose, we have performed three types of tree-building procedures for every implementation. All of them start with an empty tree in which a large amount of entries is inserted. We refer to this as initial building or as the initial tree.
In addition, we are interested in how the diverse storage allocation algorithms perform when we insert and delete entries. We used two types of modification patterns: bulk modifications and small modifications.

One construction pattern consists of inserting and deleting big amounts of entries. We refer to this as bulk building and the resulting tree as bulk tree. We started with a big file (tree after initial building) and deleted a significant amount of the entries, followed by inserting the same amount of entries. We start benchmarking right before the start of the bulk deletion.

For the small modification we have build a tree as follows. First, we insert a reasonable amount of entries into the tree. Then we start inserting and deleting small amounts entries. These insertions and deletions will be randomly alternated, but insertions are to be more common than deletions. Because we assume that in practice a database will tend to increase over time. We refer to this as interlaced building or as interlaced tree. We start benchmarking after the initial building and just before the interlaced modification.

Concluding, we benchmark the following construction and modification of trees:

Initial tree. This tree is build by inserting an amount of entries that is 66% of the amount of data, which is supplied to our buildmbbfiles program.

Bulk tree. We start with a copy of the initial tree. After this, we start benchmarking. Then we delete 25% of this tree, followed by insertion of the same amount of entries. In the end, the file has the same size as the initial tree.

Interlaced tree. We construct a tree using 75% of the entries that were inserted into the initial tree. After this we start benchmarking, and commence inserting and deleting entries until the size of the file reaches the same size as of the initial tree. The amount we insert or delete is randomly chosen, from one entry to 10% of the number of entries in the initial tree. We perform an insertion three times more often than a deletion.

Summarizing, we build three trees for every algorithm and for every data set. We implemented a program called buildtrees that builds these trees with the previously described data in each particular building pattern. After building and modifying, the program stores the resulting benchmark variables in a file.

4.3.5 Querying

The queries used in the benchmark consist of window operations, that is, queries which return all objects (mbbs) that overlap given query rectangles (i.e., windows). The query rectangles were chosen using a Gaussian distribution. We found it acceptable to presume that query windows in real-life are performed by such a distribution, because many geographic data sets have a bigger concentration of data in the center of the data space and they are thus likely to be queried more.
Note that this assumption is questionable and that other distributed query windows should also be tested in future work.

We performed window queries with nine different sizes ranching from 0.25% to 100% of the square size of the total data space; 100%, 50%, 25%, 10%, 5%, 2.5%, 1%, 0.5% and 0.25%.

We implemented a program called, query. It performs these queries on the created trees and stores benchmark results after performing all queries in a file.

We used the pseudo-random generator of C++ to generate query windows at runtime. The generated windows are the same for every tree because the generator is equally seeded and thus produces the same fixed list of random numbers.

4.4 Experiments

After implementing all components, we started our experiments by running the buildmbbfiles program. This created the data files from which all trees were built.

Remember that we implemented the algorithms one by one. After the implementation of an algorithm, we executed the program buildtrees. This builds 12 trees with that algorithm and stores benchmark results. After this we used the query program, which performs the previously described nine query batches of different sized windows on these 12 trees. After obtaining the results from these two programs, we compared them to other previously implemented and benchmarked algorithms. Results can be found in Chapter 5.

4.4.1 Setbacks

We encountered several setbacks during the implementation and benchmarking processes.

4.4.1.1 Bugs

Bugs are always an important issue when developing software. Of course, we encountered many small bugs when we were programming our new storage allocation algorithms. Most of time these were not difficult to fix.

On the other hand, we encountered some that were hard to track. These bugs occurred as errors when we were building a tree or querying it. Major problem was that these actions take a long time and some errors only appeared in some rare cases. Often it took a lot of time to figure out where the bug exactly was, because the error was difficult to reproduce. This is mainly caused by the big state of the program, in other words many variables and big amounts of data. Most of these bugs were found by assertion failures.

In addition, we discovered some bugs that were less obvious and passed our assertions. These bugs were mainly related to some behavior of the storage allocation algorithm which was not predicted. Executing the programs step-by-step and observing what the algorithm did, was the only
way to discover them. This also was not easy to do, because it meant staring at huge amounts of data.

One bug in the original implementation that we encountered in the provided code involved the buffer code. The bug was previously undiscovered. It was a minor mistake in the allocation of new buffer space, which caused problems with using multiple trees at the same time. It took us first a long time to figure out that it was there and not in our new algorithm code.

Another bug that we soon discovered in the buffer code was during querying of the tree. The nodes in the buffer were always set dirty and thus caused all used nodes to be written to disk. It did not take long to fix this one.

4.4.1.2 Timing in Linux

After a few runs of query program on a few trees, we discovered that the values of query times did not vary much for different storage allocation algorithms. This contradicted our belief that we should have obtained at least some small but noticeable time difference.

After a few simple tests, we discovered that our timers were not working correctly. They used standard timing functions of C++ and they are not suitable for use in a multi-threaded environment and thus do not measure I/O time. We did not suspect the timers of this, because they were included in the original implementation. We fixed the problem by using the real time clock (wall time) rather than the CPU time reported by the Linux system.

4.4.1.3 OS buffer

After the problem with the timers, we still did not get difference of the query timings. The way we performed queries, it showed that the cache of the Linux operating system (OS) interfered. The results of the first query were much slower than following queries on the same file.

The operating system Linux uses a buffer in memory for caching files. The size of this buffer is normally the total memory minus a small amount of memory that it wants to keep free. During experimenting, the size of the buffer was around, 56MB. (60MB total – 1MB kept free – 3MB for memory resident programs, that includes the OS kernel and our R*-tree implementation.)

The file size of the trees is around 5MB, thus the tree files are completely cached.

To solve the problem we looked for a way to clear this cache. We could not find any solutions through an OS command. Therefore, we decided that after doing a query we would clear this cache by copying several times a huge file (size around 64MB), with the same size as the cache. This slowed querying process considerably, but at least gave correct timings. We did not include this extra time in the benchmarking timings.

Normally we should be happy to have an increased speed due to the OS cache, but of course we need correct benchmarks of query time in our research. Because we need this to
conclude if our algorithms are working like we expected them to work. Maybe more realistic testing can be done with bigger amounts of data or under a different operating system with less or no buffering. We could not pursue these solutions, because of the limited time available to our research.

These problems made us decide to add a benchmark variable that keeps track of the movement of the disk head when we are querying. It counts the nodes that it should travel to read the next node. The measurement is done by our software, thus it is not the real physical distance that the disk head travels. Nevertheless, the value of this should mirror the timing results.

This problem with the OS buffer also interferes with our benchmarking during the building of the trees. Of course, in this phase we do not mind that the buffer speeds up the process. It is no real problem because this will happen in a real environment as well. On the other hand, it is a problem during the query phase because in real use, we would expect the cache to be relatively smaller than the tree.

A simpler and more realistic solution is the use of the R*-tree implementation in a spatial storage container (SSC). The SSC program will occupy space in memory and reduces the memory that is used for caching of files. We also can do the test with really inserting and deleting objects not only from the R*-tree index, but also from the database. Big parts of the database file would be cached by the OS leaving less space for the index file in memory.

Another solution is using bigger data sets, because the R*-tree files will be bigger when we do our experiments and possibly do not fit into the cache completely.

We did not use the two previously mentioned solutions. We did not implement a SSC, because the time restraints of our project, it simply would take too much time. We did not use bigger data sets, because within our allowed time we could not find bigger sets. We also received our results very late in the project and thus there was no time left to look for new data sets and rerun the time consuming tests. In addition, we are satisfied with the results that we obtained from our less realistic but still valuable experiments.

4.4.1.4 Artificial environment

We like to make a note about our artificial environment in which we performed our experiments. The real use of the R*-tree implementation is integrated in a Spatial Storage Container (SSC). The SSC itself also reads and writes objects to the database. This naturally moves the head of the disk. So when it accesses the R*-tree the head will be somewhere else, which reduces the effect of our storage algorithms.

We also note that a multi-user environment probably influences the effect of our algorithms as well, because most likely moves the disk head more often to various positions on disk.
However, we still expect that we can gain speed, because the R*-tree implementation reads multiple nodes from disk in one session.

4.4.1.5 Querying

Recall the query program that we used for performing window queries on our trees. When we first implemented it, it read query windows from a file that was created by the buildmbbfiles program. The program query would read one window at a time and would perform a query on a tree.

We used this for quite a long time, until we discovered that this of course interferes with our query benchmarking. Therefore, when we read a window from a file, this moves the disk head to some other position than the tree file. During real use of the tree this would never happen, because queries are normally applied by mouse or some other device that does not uses the hard disk. Therefore, we could not measure the seek-time of the querying correctly this way. We corrected this by using windows created by the pseudo-random generator of C++.

At the end of our research, another problem arose with the queries. This problem was that our implementation did not order the entries in a node.

Our implementation performs a query by searching a node from the first entry to the last entry. Thus reading the nodes that these entries point to, should be done in disk address order, for best performance and lowest seek time. The order of the entries in a node was not ordered. Therefore, we had to implement this. Now entries in a node are sorted by disk address and nodes are read in the right order.

4.4.1.6 Random number generator

In our program that builds trees, we use the C++ library's pseudo-random number generator *drand48()* for the interlaced insertions and deletions. The generator chooses when to insert or delete, and also how many entries to delete. (For constructing the other two types of trees this is not necessary.) Now remember that we want to create similar (interlaced) trees for all algorithms. Therefore, when we start constructing an interlaced tree we seed the generator with the same value and thus all algorithms perform the same pseudo-random actions.

However, the RANDOM algorithm occasionally has to randomly choose a free node to allocate. If the same pseudo-random generator function is used for that task, the generated number sequence is altered and thus this algorithm would make slightly different decisions. To prevent this from happening, we tried to use a different pseudo-random generator function named *rand()* in order to choose free nodes.

Unfortunately, we discovered that using both functions in the same program caused it to "hang". We could not figure out a solution, other than to use the same generator. Therefore, the RANDOM algorithm implementation builds different interlaced trees. This, however, should not
change the benchmark results much, so we can still compare the interlaced trees created by this algorithm with the others.

4.4.2 Unexpected luck

We discovered while implementing the Clustered Leaves algorithm, that it would not be a problem to implement a Hilbert curve variant. The algorithm always has to reorder children nodes of one parent whenever an entry is deleted or inserted. We decided that this reordering could have an additional feature, namely, having the entries ordered according to a Hilbert curve [Faloutsos, 1994]. We did not include this variant at first, because we thought it would be too time-consuming to implement it.
5 Results

In this chapter we present the results obtained from the conducted experiments. We present the results that support the evaluation of our hypothesis in section 5.1 and some additional results - not directly connected with the hypothesis - in section 5.2.

5.1 Hypothesis evaluation

In summary, our hypothesis states that we can develop a R*-tree storage allocation algorithm that exploits the relations between nodes in order to reduce the time needed for performing queries by reducing the seek time of reading nodes from disk. The algorithm also has to perform well in a dynamic environment.

Now let us see if we can prove this hypothesis by using the results obtained from our experiments. Note that we need at least one algorithm that fulfills it, in order to prove the hypothesis.

5.1.1 Approach

Recall that we proposed three types of algorithms: Simple Heuristics (two variants), Level Order (two variants) and Clustered Leaves (3 variants). The latter two types are based on some spatial relations between nodes. By comparing the query results of these types to the first type, we are able to judge if we can improve the query performance by using spatial relations.

If there are improvements shown by the latter two types, we also have to prove that these improvements are caused by the reduction of seek time. Next, we evaluate the remaining algorithms if they are capable of performing in dynamic environment. This should leave us with some (if any) algorithms that fulfill and prove the hypothesis.

Some of the graphs that we use in the following sections are the averages of the 12 different trees that were built using each algorithm (three different trees per distribution times four distributions). We use this average, because it gives us only one graph, which is easier to exam. Furthermore, we noticed no important differences in the patterns of the graphs of different trees.

5.1.2 Evaluation

We have seven algorithms that possibly can fulfill the hypothesis (paragraph 3.3). We can immediately discard the static variants of the Clustered Leaves and Level Order algorithm, because they are not dynamic. We can also discard the Simple algorithms, since they do not use spatial
relations between nodes. This leaves us with only three candidates; Dynamic Level Order, Dynamic and Hilbert Clustered Leaves.

5.1.2.1 Query performance

Let us exam the query performance of these three algorithms. We start by reviewing the average query times obtained during querying of the trees, as shown in Figure 5.1.

The graphs clearly indicate that the Dynamic and Hilbert Clustered Leaves algorithms have a better performance on all different sized queries. The Dynamic variant is from 80% to 160% faster than the simple algorithms. It also shows that dynamic variant of Level Order does not perform better than the simple algorithms. The static variants of Clustered Leaves and Level Order perform better than their dynamic variants and the simple algorithms.

Concluding, we can discard one of the candidate algorithms, the dynamic Level Order algorithm, since it does not show improvement in query performance, despite use of spatial relations. In contrast, the Dynamic and Hilbert Clustered Leaves algorithms show significant query performance improvement.

5.1.2.2 Seek time reduction

We now have to determine if the improvement of these two algorithms is caused by seek time reduction. For this purpose, we refer to Figure 5.2, where the average read times per node are charted with respect to the various query window sizes used in the experiments.
Remember that the only part of reading a disk page that can be influenced by software is the seek time. Thus, a reduction of reading time means reduction of seek-time. The graphs clearly show that our two remaining Clustered Leaves algorithms have a lower average read time per node than the simple algorithms and the Dynamic Level Order algorithm.

In addition, we present more evidence that shows that we are reducing the seek time. In Figure 5.3 we see that the distance between read nodes is reduced. This implies that the distance on disk is also reduced. (Note that this analogy is not linear, since the mapping of the file is not equal to our assumption of a linear non-fragmented file. This distance is measured and calculated by our benchmark software, thus does not represent the real distance that the disk head travels).

Thus, we can conclude that the shown query improvements are indeed caused by reduced seek time. Therefore, we still have as candidates the two Clustered Leaves algorithms, Hilbert and Dynamic Clustered Leaves.
Average Distance per Query

Figure 5.3

Average Build Time
Normalised to fastest

Figure 5.4
5.1.2.3 Dynamic capability

This leaves us with the last topic, the capability to perform in a dynamic environment. In order to answer this question, we have to investigate the benchmark results that we obtain during modification and building of trees. In Figure 5.4 we present the total build time per tree type. We did not include the static algorithms since they are not relevant to this discussion.

It is clear that the Simple algorithms and the dynamic Level Order algorithm have smaller and similar build times. The two Clustered Leaves algorithms need around 35% more time for building an initial tree, 55% to 80% more for bulk modification and 12% to 22% more for interlaced modification. (As a reference, we present the absolute building times in Figure 5.10 in section 5.2.2.)

We also look at the average insertion and deletion times per entry. This is presented in Figures 5.5 and 5.6, which show that the insertion and deletion time of an entry increases considerably for the Clustered Leaves algorithms (ranging from 30% to 60% for insertions and 50% to 130% for deletions). However, the times for the interlaced modification did not increase that much (less than 30%) compared to the Simple algorithms. (As a reference, we present the absolute insertion and deletion times in Figures 5.8 and 5.9 in section 5.2.2.)

We now look at what these results mean for the dynamic capability. First we make some assumptions. We assume that the simple algorithms are capable of performing in a dynamic environment, since there is nothing faster. Obviously the two remaining candidate algorithms perform worse, but are their building performances acceptable slower?

The Dynamic Clustered Leaves algorithm is the most likely to be capable enough, since it (most of the time) uses less build time than the Hilbert variant. Thus we shall consider only the dynamic variant.
We assume that the following represents normal use of a tree in a dynamic environment. Initially a tree is built by inserting many entries – this only happens once. Further modification of the tree will consist of inserting and deleting entries. Because data is expensive and hard to obtain, it is likely that modification of the tree by big amounts of data is not very frequent and that smaller modifications are more common. We also assume that insertions are more common than deletions. Thus, an increase of the initial building time is acceptable since it is done only once. Increased bulk insertion and deletion times can also be accepted if they do not happen too often and are not too expensive. Increased time for inserting and deleting small amounts of entries is less acceptable, since these operations are frequent.

In our opinion, the extra time taken by Dynamic Clustered Leaves is acceptable in a dynamic environment, although it depends on the use of the database and the relation between the amount of modification and querying.

Let us now look at the Hilbert variant of the Clustered Leaves algorithm again. We can see in Figure 5.7 that its query performance is slightly better (7%) on average compared to the Dynamic variant. In contrast, it needs around 15% more time for modifying trees. Therefore we find this algorithm less suitable and it does not prove the hypothesis because in our opinion cannot perform well enough in dynamic environment.

5.1.3 Conclusion hypothesis

Concluding, we consider our hypothesis proven by the development of the Dynamic Clustered Leaves algorithm and its benchmark results. In other words, it is indeed possible to develop an algorithm that reduces the query time, by using spatial relations between nodes that reduce the seek time of reading nodes and is capable of performing well in a dynamic environment.
5.2 Additional results

We found more results that we did not use to evaluate our hypothesis, but contain some interesting information.

5.2.1 Query speed

In Figure 5.7 we present a graph which contains the relative average query time for every algorithm. This graph makes it easy to compare the query performance of the Dynamic Clustered Leaves algorithm with the other algorithms. In other words we can determine the relative difference between the algorithms more easily. The data that we used here is the same as in Figure 5.1.

The graph shows that the Dynamic Clustered Leaves algorithm performs around 2 times faster for all query sizes than the Simple algorithms and the Dynamic Level Order algorithm. For queries with sizes of 10%, 5% or 2.5%, it performs even better, around two and a half times faster.

Note that the improvement decreases for smaller queries. This is because small queries tend to request few leaf nodes. The improvement also gets smaller for big queries. This is because we have to read many nodes of the trees and the node buffer gets full. This causes us to reread higher level nodes that were removed. Those are not clustered together thus take more time to be read. In medium sized queries, the higher level nodes stay in memory and thus show more improvement. Another reason why there is less improvement is that in big queries the leaf nodes of different parents have to be read, and these can be stored far from each other.
Dynamic Level Order

Random

Dynamic Level Order

Dynamic Clustered Leaves

Hilbert Clustered Leaves

Average Insert Time per Entry

Figure 5.8

Average Delete Time per Entry

Figure 5.9

Average Build Time

Figure 5.10

File utilization rate

Figure 5.11
We also want to make a note about the Dynamic Level Order (DLO) algorithm. If we compare the query performance of the DLO algorithm to the simple algorithms, we can see that they have a similar performance. If look at the static variant of the Level Order algorithm (SLO), we can see that the principle of storing the nodes in level order can improve the query performance.

Thus, the DLO algorithm cannot maintain the level order in a dynamic environment very well. This is because there are not enough free nodes when requesting new nodes.

Because the static variant does show improvement, we suspect that we can improve the dynamic variant. We did not research this, however. Moreover, we believe that the Clustered Leaves algorithm is more promising, because its static variant shows a better performance than the Level Order algorithm and is thus more deserving of a more extensive investigation.

5.2.2 Construction, insertion and deletion

For completeness we present here the (absolute) average insert and delete times per entry. These graphs are of course analogous to the graphs presented in Figure 5.8 and 5.9. In Figure 5.10 we present the absolute building time per building type.

5.2.3 Disk space

In Figure 5.11 we present the file utilization rate of the algorithms. The Simple algorithms and the Level Order algorithm don’t use sophisticated reorganization algorithms and thus don’t use more file space more than necessary. In contrast, the Clustered Leaves algorithms do use more file space and have a lower file utilization rate. They use 57% to 63% less of the index file, compared to the other algorithms.

If we look at the Static version of the Clustered Leaves algorithm in Figure 5.7, we must conclude that when we shift all nodes to the beginning of the file that the query performance improves considerably. Thus, a low utilization rate means a higher read time per node. In other words it is important to keep the gaps in the file as small as possible. We recommend researching an algorithm that uses the Clustered Leaves principle, but has a better file utilization and is still capable of performing in a dynamic environment. We suggest using dynamic interval management [Kanellakis, 1993], which can reserve intervals of disk space. In other words, sibling nodes can be stored together in one interval, that was reserved for the siblings of the parent node. This should reduce needed disk space and thus give a better query performance. It also probably reduces disk traffic during modification and building. Alas, we had no time to pursue this improvement.
Figure 5.12

Figure 5.13

Figure 5.14
5.2.4 Disk traffic

Here we take a look at the disk traffic during building of trees. The benchmarking of this variable is presented in Figure 5.12.

Clearly visible is the increased number of read and written nodes of the Clustered Leaves algorithms. Especially when we are constructing a tree from scratch.

We can also see that the random algorithm performs more reads and writes during interlaced building. This is caused by the slightly different modification compared to the other algorithms.

The extra time that the Clustered Leaves algorithms use for building and modification can be explained by the increased number of read and written nodes. However, these algorithms have a better read and write time per node, which lowers total time that would have been bigger if other algorithms should read the same amount of nodes. Read and write time per node during building and modification are presented in Figure 5.13 and 5.14.

5.2.5 Node buffering

Another point of interest is the hit rate of the buffering of the R*-tree implementation prevents. In Figure 5.15 the hit rate of the buffer during building of trees is presented. In Figure 5.16 the hit rate during querying.

It is clearly visible that the buffer prevents most of the read operations from disk. During building of trees it prevents between 93% to 98%. During querying it prevents between 83% to 97%. Therefore our storage allocation algorithms only influences a small percentage (from 17% to 3%) of requested nodes during querying. Although this small percentage we manage to obtain a reasonable query performance improvement, see Figure 5.7.
Figure 5.15

Average Building Buffer Hit Rate

Figure 5.16

Average Query Buffer Hit Rate

equal for all algorithms
6 Conclusions

The hypothesis of our research has been proven. In other words, we exploited spatial relations between nodes to develop an R*-tree storage allocation algorithm that improved the query performance by reducing seek time costs.

The proposed Dynamic Clustered Leaves algorithm yields the best query performance among the tested algorithms. Compared to simple algorithms, it queries from 80% to 160% faster, depending on the size of the query. On the other hand, it increases building and modification times from 12% to 55%, depending on the type of modification.

The use of space filling curves in this algorithm gave just a slight improvement of query performance (7%). Moreover it has a slower modification performance (around 15%), thus using space filling curves makes it even less suitable for use in a dynamic environment.

All static versions of the algorithms perform better than their dynamic counterparts. The dynamic variants can be improved to have a more similar performance. The benchmark results comparing the dynamic and static variant of the Cluster Leaves algorithm showed the importance of a high file utilization ratio. Thus, improving this ratio in the dynamic variant will probably result in a better dynamic algorithm.

The dynamic variant of the Level Order algorithm does not work well without reorganization, when we compare it to the static variant.

The Clustered Leaves principle is showing more query performance improvement than the Level Order principle. On the contrary it has a slower building and modification performance.

6.1 Recommendations and future research

Here we present some topics that can be subject for further research. Some of these are natural extensions of the present work, while others cover sideline issues which we did not have opportunity to pursue in depth.

6.1.1 Algorithm improvements

The query performance of the implemented static algorithms gives the indication that we can gain more than we have achieved with our dynamic variants. In particular, we recommend spending some effort on improving the Dynamic Clustered Leaves algorithm and the Dynamic Level Order
algorithm. Since the static variant of the Clustered Leaves algorithm shows more improvement than the Level Order variant, we suggest a more intense effort on the former.

One way of improving the Dynamic Clustered Leaves algorithm consists of trying to increase the file utilization rate by using dynamic interval management. The Dynamic Level Order algorithm can probably improved by introducing small reorganizations.

6.1.2 Implementation of a storage container

A storage container data structure provides an integrated means of querying and retrieving data. This is in contrast with an index, which only points to locations in another data structure – usually a direct access file – where the data is actually stored. Thus, for instance, we could modify an R-tree index so that the leaf nodes contain actual data, rather than pointers. The main problem with this simple approach is that the spatial data items may have different storage requirements that probably would make it impractical to store them in fixed-size blocks. Another concern is that the data items might be referred to by external indices, and thus the container should provide facilities for retrieving data quickly based on a short identifier rather than by contents.

Since the present work showed that it is worthwhile to cluster index entries on disk based on their spatial proximity, it is reasonable to suppose that clustering data items similarly could also prove to be advantageous.

6.1.3 More testing

Due to the limited amount of time of our research, we did not research how our algorithms perform on higher dimensional data. We only investigated two-dimensional data.

We also recommend further researching the static storage allocation algorithms.

We only investigated Gaussian distributed window queries. The influence of the storage algorithms on other types of queries should be researched as well.

6.1.4 Buffer

We want to suggest investigating the buffer algorithm of the R*-tree. Due to the limitation of our research, we did not research different buffer algorithms. However, this is very important since the buffer decides which nodes are kept in memory and which are not. For instance, by exploiting the buffering module of our R*-tree implementation, it should be possible to arrange an optimum node/page layout without resorting to intensive disk I/O.
6.1.5 Code optimization

The current implementation of the R*-tree and storage allocation algorithms is not optimized. Significant time gain can be achieved by optimizing the code. This gain can consist of both reduced CPU utilization and fewer disk read/write operations.
Appendices

A. References


“Well I’ve nothing to show for all these years -
But it still makes me smile
That the ones you always hear the most
Say nothing that’s worthwhile.”

Martin Walkyier


I. Kamel and C. Faloutsos. “Hilbert R-tree: An Improved R-tree using Fractals”. In proceedings of the 20th International Conference on Very Large Data Bases (VLDB), September 1994.


B. Internet Links

**Brazil**

COPPE, UFRJ, Rio de Janeiro  
http://www.coppe.ufrj.br

Database Group, UFRJ, Rio de Janeiro  
http://www.cos.ufrj.br/~bd/database.htm

Systems Engineering and Computer Science Program, UFRJ, Rio de Janeiro  
http://www.cos.ufrj.br/welcome.html

Universidade Federal do Rio de Janeiro (UFRJ)  
http://www.ufrj.br

**The Netherlands**

Database Systems Group, TU Delft  
http://is.twi.tudelft.nl/dbs/dbs_en.htm

Information Systems Department, TU Delft  
http://is.twi.tudelft.nl

Faculty of Information Technology and Systems, TU Delft  
http://www.its.tudelft.nl

Technical University Delft  
http://www.tudelft.nl

**Personal**

Arnaut de Rijk, Delft  
http://rotzorg.org/~arnaut/
C. System Benchmark

System benchmarking is done under Windows 98 by WinTune, version 1.0.42, freely distributed.

Summary
Intel Celeron A (1) 300 MHz
Windows 98 4.10.1998
850±0,013 (0,001%) MIPS (Integer operations)
341±4,7/(3,4%) MFLOPS (Floating point operations)
70±0,056 (0,08%) (Integer application simulation)
72±0,064 (0,089%) (Floating point application simulation)
69±0,023 (0,033%) (MMX application simulation)
530±2,6 (0,48%) RAM MB/s
58±0,7 (1,2%) cached disk MB/s
2,6±0,015 (0,57%) uncached disk MB/s

System Details
BIOS: ?
Bus: PCI,
APM: Version 1.2 Flags 0x3
Possible 'hog' apps running: Find Fast
Other apps running: Test results report; MM_MenuPadClass; WinTune;
Go2Zilla - Files Without Category;
DDHelpWndClass; [hidden window];
LVComSWnd; Magellan MSWHEEL; Logitech E/M Executive; Logitech GetMessage Hook; LogiIcon Window 10; Reminder;
Medidor de energia; Spooler Process;
MS_WebcheckMonitor

CPU Details
CPU load: 2
low MIPS: 600
CPUID: 0x0660 0x183F9FF
MMX Present: True
3DNow Present: False
Streaming SIMD Extensions Present: False
Processor Serial Number Present & Enabled: False
dhrystone time (s): 2,4
whetstone time (s): 0,03
Integer time (s): 11
Floating point time (s): 9,9
MMX time (s): 13

Memory Details
Memory Read Speed (MB/s): 589
Memory Write Speed (MB/s): 561
Memory Copy Speed (MB/s): 441
Installed RAM (MB): 64
Free RAM (MB): 13
Memory used (%): 60
Page File Driver: 32-bit
Total Page File (MB): 127
Free Page File (MB): 102
Read speed, 4 KB blocks, MB/s: 1113
Read speed, 8 KB blocks, MB/s: 1113
Read speed, 16 KB blocks, MB/s: 1095
Read speed, 32 KB blocks, MB/s: 614
Read speed, 64 KB blocks, MB/s: 615
Read speed, 128 KB blocks, MB/s: 387
Read speed, 256 KB blocks, MB/s: 216
Read speed, 512 KB blocks, MB/s: 217
Read speed, 1024 KB blocks, MB/s: 217
Read speed, 2048 KB blocks, MB/s: 217
Write speed, 4 KB blocks, MB/s: 1415
Write speed, 32 KB blocks, MB/s: 564
Write speed, 64 KB blocks, MB/s: 153
Write speed, 256 KB blocks, MB/s: 119
Copy speed, 4 KB blocks, MB/s: 1226
Copy speed, 32 KB blocks, MB/s: 357
Copy speed, 256 KB blocks, MB/s: 91
Copy speed, 2048 KB blocks, MB/s: 87

Disk Details
Disk Last Scanned: 19/08/99 12:58:10
Disk Last Optimized: 00:00:00
Cached Disk, 128 blocks
Open file time (s): 0,00079
Sequential write time (s): 0,008
Sequential read time (s): 0,065
Random write time (s): 0,0013
Random read time (s): 0,0057
Close file time (s): 0,00074
Open file bytes: 1024
Sequential write bytes: 524288
Sequential read bytes: 4194304
Random write bytes: 65536
Random read bytes: 262144
Close file bytes: 1024
Open file MB/s: 1,2
Sequential write MB/s: 62
Sequential read MB/s: 61
Random write MB/s: 49
Random read MB/s: 43
Close file MB/s: 1,3
Uncached Disk, 2048 blocks
Open file time (s): 0,00079
Sequential write time (s): 1,5
Sequential read time (s): 0,93
Random write time (s): 1,2
Random read time (s): 3,3
Close file time (s): 0,0027
Open file bytes: 1024
Sequential write bytes: 8388608
Sequential read bytes: 8388608
Random write bytes: 1048576
Random read bytes: 1048576
Close file bytes: 1024
Open file MB/s: 1,2
Sequential write MB/s: 5,5
Sequential read MB/s: 8,6
Random write MB/s: 0,84
Random read MB/s: 0,3
Close file MB/s: 0,3
D. Interval List

// interval.h & interval.cc
// ========= ===========
// these files include all the required methods and declarations of the
// interval list.
//
// -- every rtree has its own interval list.
// -- the interval list is a list of interval of all the nodes of the rtree
// -- is used for optimizing the allocation of new nodes
// -- the interval list is stored in the head node (after the head info).
// WARNING! If it doesn't fit, it will CRASH on writing/closing the tree.
// Fix of this still has be implemented.
//

class RtreeNode;

typedef unsigned short int Nodenumber;
typedef short int Leveltype;

struct IntervalEntry {
    Nodenumber begin;
    Nodenumber end;
    Leveltype level;
};
E. **Pseudo code of algorithms**

**Random request for new node**

if Free Nodes
   Use random free node
else
   Create new node at the end of file

**Lowest Free request for new node**

if Free Nodes
   Use lowest free node
else
   Create new node at the end of file

**Dynamic Level Order request for new node**

if Free nodes
   if node free in "perfect" interval
      Use highest free node in interval
   else if Free node next to interval of same level as the new node
      Use Position after interval or before
   else
      Use free node as close as possible to "perfect" interval
   else
      Create new node at the end of file

**Clustered Leaves**

Request for new node

if Free node next to other children nodes
   Use node next to other children
else if Free nodes
   Use lowest free node
else
   Create new node at the end of file
Put children together

Put children together (used after request for a new node and after deletion of a node)

if Children are not together
   if only free nodes in between children
      Move children together by filling gaps
   else
      Look for free interval with minimal length of the number of children and move them there
if Hilbert variant
   sort children
F. Research environment

Database Group, UFRJ, Rio de Janeiro

The Database group is part of the System Engineering and Computer Science Program, at COPPE/UFRJ in Rio de Janeiro, Brazil. It develops a large number of research and consulting projects related to traditional, distributed and parallel databases. It also provides courses on these subjects.

The group developed GOA++, an object management system compatible with ODMG’s OQL and OMG’s CORBA. There is a sequential and a parallel version of GOA++ and its antecessor, GOA.

Among the applications areas the group deals with are Geographic Information Systems, High Energy Physics, Oil Prospecting, Satellite Management, Strategic Planning, Telephony, Environment, High Tech Commercial Applications, Web Site Development, Mobile Computing and Project Management.

Database Systems Group, TU Delft

The Database Systems Group is part of the master science courses of the Faculty of Information Technology and Systems at the Technical University of Delft, The Netherlands.

The research of this group is directed to problems in the database area. Some of these problems are related to the following subjects; persistency, performance, parallelism, distribution versioning, support for work flow, support for multi-media, quality of information, costs on development and maintenance and protecting older investments in legacy databases.

The group has two main directions of research and education:

a) Object Management, directed to requirements from engineering environments and enterprise automation to achieve seamless integration of the traditional 3-schema architecture and the applications.

b) Semantic Data modeling, continuing the traditional research line of the section, based on the 3-schema architecture.