Abstract

Title of Dissertation: High Performance Spatial Indexing for Parallel I/O and Centralized Architectures

Ibrahim M. Kamel, Doctor of Philosophy, 1994

Dissertation directed by: Associate Professor Christos Faloutsos
Department of Computer Science

Recently, spatial databases have attracted increasing interest in the database field. Because of the volume of the data with which they deal with, the performance of spatial database systems' is important. The R-tree is an efficient spatial access method. It is a generalization of the B-tree in multidimensional space. This thesis investigates how to improve the performance of R-trees. We consider both parallel I/O and centralized architectures.

For a parallel I/O environment we propose an R-tree design for a server with one CPU and multiple disks. On this architecture, the nodes of the R-tree are distributed between the different disks with cross-disk pointers (*'Multiplexed R-tree'*). When a new node is created we have to decide on which disk it will be stored. We propose and examine several criteria for choosing a disk for a new node. The most successful one, termed *'Proximity Index'* or PI, estimates the

similarity of the new node to other R-tree nodes already on a disk and chooses the disk with the least degree of similarity.

For a centralized environment, we propose a new packing technique for R-trees for static databases. We use space-filling curves, and specifically the Hilbert curve, to achieve better ordering of rectangles and eventually to achieve better packing. For dynamic databases we introduce the *Hilbert R-tree*, in which every node has a well defined set of sibling nodes; we can thus use the concept of local rotation [47]. By adjusting the split policy, the *Hilbert R-tree* can achieve a degree of space utilization as high as is desired.

High Performance Spatial Indexing for Parallel I/O and Centralized Architectures

by

Ibrahim M. Kamel

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1994

Advisory Committee:

Associate Professor Christos Faloutsos, Chairman/Advisor
Assistant Professor Michael Franklin
Professor Alan Hevner
Professor Nick Roussopoulos
Professor Hanan Samet
Professor Ben Shneiderman

Dedication

To my parents and my wife

Acknowledgements

I would like to express special thanks to my advisor, Professor Christos Faloutsos, who has provided me much advice, encouragement, and criticism in the course of my study and research at the University of Maryland. He read draft papers patiently and promptly, and always gave me valuable comments, which greatly helped me clarify my ideas and formulate the problems in a correct and simple way. I am very grateful to him. We had many thought-provoking discussions. It has been a great privilege to work with him.

I wish to express my gratitude to Professor Nick Roussopoulos who served in my dissertation committee and helped me a lot in various ways. Thanks are due to the other members of my dissertation committee, Professors Michael Franklin, Allan Hevner, Hanan Samet, Ben Shneiderman.

I would like to express my thanks and gratitude to professor Ken Salem and Timos Sellis for their help, support, and encouragement during my entire course of study.

I would like to thank my friend Walid Aref for the fruitful discussions and valuable advice. Special thanks are due to the colleges Mohammed Abedel-Mottaleb, Chung-Min Chen, Alex Delis, Nick Koudas, David Lin, Ibrahim Matta, George Panagopoulos, Kyuseok Shim, and Konstantinos Stathatos. I would also remember Nancy Lindley, Helen Papadopoulos and Sue Elliott who have helped in numerous administrative affairs.

During my stay at IBM Almaden Research Center in San Jose, California, I benefited from discussions with many members of the K55 department. Special thanks are due to Manish Arya and Bill Cody.

Special thanks to Professor Ramez Elmasri and Vram Kourmajian for valuable discussions fruitful cooperations.

Finally, back home, I would like to thank all my professors at Alexandria University in the Department of Computer Science and the Department of Environmental Studies who provided a lot of support and instruction during my undergraduate and the initial years of my graduate studies; my parents and parents in-law who have provided endless support during my stay in the United States; and my wife and children for their great sacrifice and patience during my course of study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Databases of the near future will be required to support non-traditional data types, such as spatial objects [71]. Multimedia databases [54], Geographical Information Systems (GIS) [66], and medical databases [4] are examples of databases receiving increasing attention. Handling spatial and multidimensional objects is a common requirement among these databases. For example, in multimedia databases we should be able to store images [5], voice [56], video [62] etc. In GIS, maps contain multidimensional points, lines, and polygons all of which are new data types. Another example of such non-traditional data types can be found in medical databases which contain 3-dimensional brain scans (e.g. PET and MRI studies); in these databases we want to ask a query such as "display the PET studies of 40-year old females that show high physiological activity inside the hippocampus" where high activity corresponds to high glucose consumption. Temporal databases fit easily in the framework, since time can be considered as one more dimension [48, 50]. Multidimensional objects appear even in traditional databases, where a record with k attributes corresponds to a point in the k-d space.

In the above applications, one of the most typical queries is the *range query*:

Given a rectangle in k-d space, retrieve all the elements that intersect it. A special case of the range query is the *point query* or *stabbing query*, where the query rectangle degenerates to a point. Spatial join is an important query which is also expensive to compute. It is used to combine spatial objects of two sets according to some spatial properties. For example, consider two spatial relations that define the borders of lakes and counties. The query "give me a list of counties and all the lakes in them" is an example of a spatial join query. Other queries of interest include the nearest neighbor queries [6]. The query "find the nearest lake to Prince Georges county" is an example of a nearest neighbor query.

Several spatial access methods approximate objects with, for example, their minimum bounding rectangle (MBR), (or circle, or ellipse, etc.). Range queries are also approximated by their MBR's, requiring a post-processing step to discard the false alarms. We focus on the first step, that is, on how to organize efficiently a large set of multidimensional rectangles for range queries. This is one of the major goals of this thesis.

The second goal is to examine issues of declustering and data partitioning. All the above applications have a common property in that they deal with huge amounts of data. With the increase in the volume of data, the response time of the range query increases. Also, the data itself eventually will not fit on one disk. One way to relieve these problems is to distribute the data carefully on more than one unit so that the data can be retrieved and searched in parallel (e.g. [43, 69]).

The remainder of the thesis is organized as follows: Chapter 2 presents some of the related work on spatial indexing. In Chapter 3, we present the Multiplexed I/O R-tree. In Chapter 4, we present two new R-tree designs based on the Hilbert curve for a centralized environment. Chapter 5 gives some concluding remarks and directions for future research.

Chapter 2

Survey

In this chapter we present a classification of older spatial access methods, a survey of declustering methods, and a survey of analysis of R-trees. A recent survey can be found in [67]. Several spatial access methods have been proposed. For the purpose of this dissertation, we provide the following mean of classifying the structures.

1) Methods that are designed for storing multidimensional points only. These methods are called Point Access Methods (PAM) – e.g., Grid files [35], LSD tree [34], buddy tree [68], and DOT [20]. One way to use PAM for storing non-point objects is to transform the objects to points in higher-dimensional space [35]. For example, rectangles in two-dimensional space can be transformed to points in four-dimensional space by using the x,y coordinates of two opposite corners. Other transformations are also possible, such as using the coordinate of the center and the extent values along the x and y axes. This technique however, has drawbacks, for example, the mapping from the original space into the point space may result in a skewed point distribution and may thus adversely affect the search performance.

2) Methods for spatial objects 'Spatial Access Method' or (SAM) : Other in-

dexes are designed to store points as well as non-point objects – e.g., Quadtree [28] [3], R-tree [32, 7, 44, 41, 16], Z-order [60], R^+-tree [70], and Cell tree [31].

In this thesis, we concentrate on R-tree like-structures.

2.1 Point Access Methods (PAMS)

PAM are designed to handle multidimensional points. Non-point objects can be transformed to points in higher dimensional space before being stored. Several PAM have been proposed. We can divide them into hierarchical structures such as the K-D-B tree [63] and non-hierarchical structures such as Grid files [55] and their variants.

2.1.1 Hierarchical Structures

The k-d tree [8] is a generalization of the binary search tree for multi-dimensional points. At each level a different attribute (or key) value is tested to determine the direction in which a branch is to be made. The k-d tree is a main memory-based structure; it was the inspiration of several disk-based data structures such as the K-D-B tree and the LSD tree.

Henrich *et al.* [34] proposed the Local Split Decision (LSD) tree. Its directory structure is similar to that of the k-d tree [8]. It partitions the data space into pairwise disjoint cells. The cutting boundaries may occur at arbitrary positions. Henrich also introduced an algorithm for paging a multi-dimensional binary tree. The LSD tree can store only multi-dimensional points. K-dimensional intervals are transformed into points in a 2k-dimensional space.

The K-D-B tree of Robinson [63] is one of the first multidimensional indexes proposed for secondary storage. It combines the properties of the k-d tree [8] and the B^+-tree. Each time an overflow occurs, the search space is partitioned into two disjoint rectangular subspaces along one axis. Like the B-tree, the K-D-B tree is a balanced tree; that is, all paths to leaves of the tree are equal in length. All data is stored in leaf nodes. The internal nodes containing only entries which direct the search. When a non-leaf node is split, the split may propagate downwards; the structure thus does not guarantee minimum space utilization. To avoid this problem several variants have been proposed, including the Buddy tree [68] and the hB-tree [52].

Seeger and Kriegel [68] proposed the Buddy tree, which is similar to the K-D-B tree [63]. They avoided some of the drawbacks of the K-D-B tree, such as the downward split, by using a partitioning schema similar to the buddy system [47]. The buddy tree stores the MBR of the data in each node in order to better prune the search space.

Lomet and Salzberg [52] suggested a variant of the K-D-B tree called the hB-tree, which exhibts the following distinctions. Index nodes are organized as a k-d tree to improve the intra-node search response. When a node overflows, it is not necessarily split into two rectangular k-dimensional regions (bricks), but rather divides into "holey" bricks, or bricks from which smaller bricks have been removed. Because of this, hB-trees can avoid the downward split propagation that occurs in K-D-B tree. The hB-tree guarantees at least 33% node utilization.

The BANG file [24] of Freeston is a grid file type (Grid files are explained in the next section), but its directory is organized as a tree structure (as opposed to a multi-dimensional array as in Grid files). As in the B-tree, the updates and splits propagate upwards through the tree, thus balancing the tree.

2.1.2 Non-hierarchical Structures

Nievergelt's Grid file [55] is a non-hierarchical index structure for data characterized by several keys or attributes. The records can be represented as points in a multi-dimensional space formed by the Cartesian product of the domains of the keys. The space is divided into a grid; each grid cell is stored in a disk page and contains b records (points) at most. A multi-dimensional array ('directory') is used to map grid cells to the corresponding pages on the disk. The directory may reside on the disk. A set of one-dimensional arrays called *linear scales* are used to store the partition points along each attribute. They enable access to the appropriate grid cells by aiding the computation of cell addresses as determined by the value of the relevant attributes. The $linearscales$ are kept in main memory. When a page overflows, the corresponding grid cell has to split; the directory may grow. Similarly, when deletions occur, grid cells can be merged. The grid file guarantees that any record can be retrieved (exact match query) with two disk accesses, one for the directory and one for the data.

Tamminen's EXCELL [72] is similar to the grid file. It is based on a regular decomposition of the space, and it requires a grid directory; however, all grid cells are of the same size. The main difference between the grid file and EXCELL is that when a data page overflows, the grid file splits only the corresponding directory cell. In contrast, the EXCELL method splits all directory cells and results in a doubling of the size of the grid directory. As a result, the sizes of the direcory cells are the same. In contrast, the directory cells of the grid file are not necessarily of the same size. Because the directory cells are all of the same size, EXCELL does not require a set of linear scales to access the grid directory, as does the grid file.

For a data set with correlated attributes, the index size increases and becomes sparse and thus the search performance degrades. Hinrichs and Nievergelt [35] suggested using the grid file after a rotation of the axes. The rotation is necessary in order to avoid non-uniform distribution of points, which would lead to poor grid file performance. Faloutsos and Rego [19] proposed dividing the address space into triangular cells (as opposed to rectangular one as in the grid files) in order to better handle the correlated data and non-point geometric objects.

The standard grid files achieve about 70% storage utilization. Hutflesz *et al.* [38] proposed the 'Twin Grid File' which achieves roughly 90% storage utilization. The basic idea is to use two grid files instead of one as in the standard Grid file. A new point is inserted in either file in such a way as to avoid node splits as much as possible. They showed experimentally that the storage gain is obtained at no extra cost and that range queries can be answered in twin grid files at least as fast as in the standard grid file.

2.2 Spatial Access Methods

In this section we present spatial access methods that are designed to handle point as well as non-point spatial objects.

2.2.1 Quadtree-based Methods

The quadtree is a hierarchical data structure based on a recursive decomposition of the space [23]. Quadtrees are used for points, as in the point quadtree [23], the MX quadtree, and the PR quadtree [57, 65]; for rectangles, as in the MX-CIF [45, 1]; and for lines, as in the PMR quadtree. The decomposition may be regular (e.g. the PR quadtree) or irregular (e.g. point quadtree). "Irregular" decomposition means that the decomposition is driven by the data: Splits occur at each data point, which is represented as a node in the tree. In "regular" decomposition, the space is decomposed into quadrants of the same size. Orenstein [57] proposed a k-d trie which is similar to the PR quadtree but uses binary trees instead of quadtrees. Octrees [37, 39] are the extension of quadtrees in three-dimensional space. A detailed survey of the quadtree and its variants can be found in [67].

Gargantini [28] proposed a disk-resident quadtree called the linear quadtree. Spatial objects are divided into quadtree blocks, whose z-order (Morton key) is used as the primary key for a B^+-tree [1] organization. Equivalently, Orenstein [60, 58] proposed the Z-order which divide the spatial object into rectangular blocks and store them in any PAM. In order to avoid an excessive number of elements, Orenstein also studied the trade-off between the number of elements that cover the spatial object (amount of redundancy introduced) and the amount of extra space they cover [59].

The Z-order is a member of a family of curves called 'space-filling curves'. One of their characteristics is to pass by every point in the space exactly once. Other space-filling curves such as the Hilbert and Gray codes can be used to linearize the multi-dimensional space and to store the data in a PAM [21]. In [21, 40] they experimentally showed that the Hilbert curve achieves the best clustering among other methods.

2.2.2 R-tree-based Methods

One of the most characteristic approaches in spatial access methods is the R-tree proposed originally by Guttman [32]. It is an extension of the B-tree for multi-dimensional objects. The R-tree is a balanced structure, and it maintains at least 50% space utilization. A geometric object is represented by its minimum bounding rectangle (MBR). Non-leaf nodes contain entries of the form (ptr,R) where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form (obj-id, R), where obj-id is a pointer to the object description, and R is the MBR of the object. The R-tree allows father nodes to overlap. In this way, the R-tree can guarantee good space utilization and remain balanced. Figure 2.1 illustrates data rectangles (in black) organized in an R-tree with fanout value

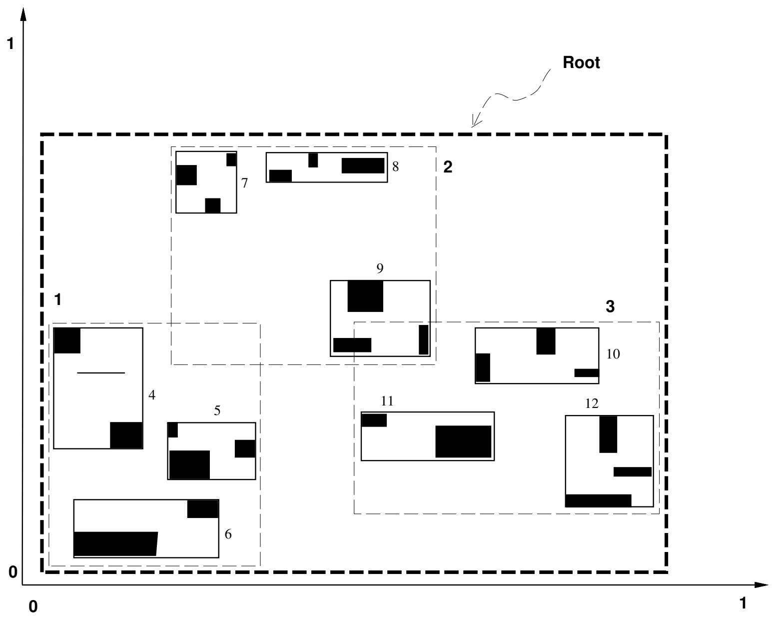


Figure 2.1: Data (dark rectangles) organized in an R-tree. Fanout=3.

of three. Figure 2.2 shows the file structure for the same R-tree, where nodes

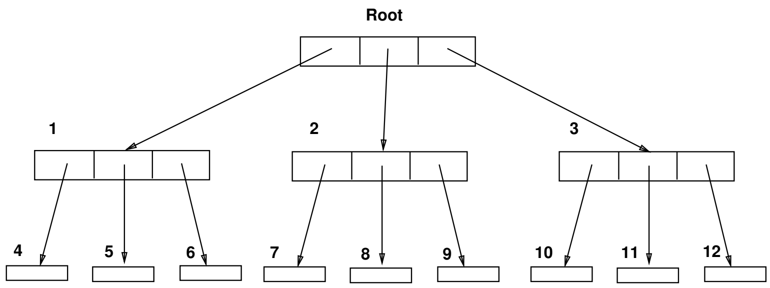


Figure 2.2: The file structure for the R-tree in Figure 2.1 (fanout=3).

correspond to disk pages. On the other hands, excessive overlaps of the father nodes penalizes the search performance. The worst case for the search is to retrieve the whole tree, but this rarely happens with practical datasets.

The R-tree is a dynamic structure in the sense that insertions and deletions may be intermixed with queries; the tree grows and shrinks accordingly. When a node overflows as a result of an insertion, a split occurs to create two nodes, each of which is half full. The split may propagate up the tree until the root is split, in which case the tree grows by one level. Guttman originally proposed three splitting algorithms, the *linear split*, *quadratic split*, and the *exponential split*. Their names reflect their complexity; among the three, the quadratic split is the one that achieves the best trade-off between splitting time and search performance.

The R-tree inspired much subsequent work, the main focus of which was to improve the search time. A packing technique proposed by Roussopoulos [64] minimizes the overlap between different nodes in the R-tree for static data. That is their R-tree does not support insertion nor deletion; once the R-tree is built, it is breezed. The idea is to sort the data on the either x or y coordinate of one of the corners of the rectangles. The sorted list of rectangles is scanned; successive rectangles are assigned to the same R-tree leaf node until the node is full; a new leaf node is then created and the scanning of the sorted list continues. Thus, the nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. The utilization is thus $\approx 100\%$. Their experimental results on point data showed that their packed R-tree performs much better than does the linear split R-tree for point queries. Sellis *et al.* [70] proposed the R^+-tree that avoids the overlap between non-leaf nodes of the tree by clipping data rectangles that cross node boundaries. In this model there is therefore only one path to the data in a given region as opposed to the multiple paths of Guttman's R-tree. The trade-off is that for a specific data object there might be more than one entry in the R^+-tree, and there can thus be more levels in the search path than in that of an equivalent R-tree. Also, a non-leaf node split in the R^+-tree might cause a downward split propagation. When splits propagate downwards, there is no way to guarantee a minimum number of entries per node. Beckman *et al.* proposed the R^*-tree [7]. Their experiment showed that it gives better performance than other R-tree variants. The main idea in their proposal is the concept of *forced re-insert*, which is analogous to the deferred-splitting in B-trees. When a node overflows, some of its children are deleted and re-inserted, usually resulting in a better-structured R-tree. Beckman *et al.* also introduced a new splitting and a new insertion algorithm. These algorithms take into consideration not only the area as in Guttman's R-tree, but also the perimeter and the overlap of the directory rectangles.

Gunther's Cell tree [31] is an extension of the BSP tree [26, 25] for secondary storage. It divides the search space into disjoint polyhedron cells. The data are organized in a hierarchical structure. The interior nodes correspond to nested hierarchy of convex polyhedra. Jagadish [41] suggested the use of polygonal bounding instead of rectangular bounding of the spatial objects. He showed that the benefit, in terms of better selectivity due to improved bounding, is significant for the first few dimensions; however, the incremental benefit of an added dimension goes down as more dimensions are added.

R-trees can also be used for spatial join queries. Brinkhoff *et al.* [11, 10] studied spatial join processing when two R-trees are available. Their primary idea is the use of several (typically three) filter steps. In the first step, the spatial join is performed on the minimum bounding rectangles (MBR's) of the objects. In the second step, they use a geometric filter that better approximates the object. In the last step, the join predicate is checked for all remaining candidates using the exact match geometry. They used several algorithms for each filter step. For the last step, they decomposed the polygonal objects into sets of trapezoids. Each object is organized in a memory resident tree. They have shown experimentally that using their approach improves the total execution time of the spatial join by a factor of more than three over the straightforward approach.

For the case in which we have one dataset with an R-tree index and another dataset without such an index, Lo and Ravishankar [51] suggested to build an R-tree like structure called a *seeded tree* for the second data set at the join time. Using some parameters from the first R-tree, they build the *seeded tree* in such a way to minimize the join cost.

2.3 Distributed Spatial Indexing

Much work has been done on methods for organizing traditional file structures on multi-disk or multi-processor machines. For the B-tree, Pramanic and Kim proposed the PNB-tree [61], which uses a 'super-node' ('super-page') scheme on synchronized disks. Seeger and Larson [69] proposed an algorithm to distribute the nodes of the B-tree on different disks. Their algorithm takes into account not only the response time of the individual query but also the throughput of the system.

A large number of methods have been proposed to decluster the Cartesian product files (i.e. Grid files). These methods can be grouped into two classes: In the first class, the methods are designed for partial match queries. Methods in this class include the Disk Modulo family [13], the field-wise exclusive OR (FX) method [46], methods using error correcting codes (ECC) [17], and methods using minimum spanning trees [22]. In the second class, the methods are designed for range queries: e.g., HCAM [15] and MAGIC [29]. In HCAM [15] the Hilbert curve is used to impose linear order on the buckets in a multi-dimensional space, and then to traverse this sorted list of buckets, assigning each bucket to the disk in round-robin fashion. In MAGIC [29], it is assumed that the access pattern is known, and the size of the bucket is calculated in order to balance the loads at the units. Also, the number of processors activated per query is restricted in order to minimize the overhead imposed by parallelism.

Chapter 3

Multiplexed I/O R-trees

3.1 Introduction

In this chapter we study the problem of improving the search performance using parallel I/O architectures such as multiple disk units. There are two main reasons for using multiple disks as opposed to a single disk:

(a) Spatial database applications are mostly I/O bound. Our measurements on a DEC station 5000 showed that the CPU time to process an R-tree page, once brought in core, is 0.12 msec. This is 156 times smaller than the average disk access time (20 msec). Therefore it is important to parallelize the I/O operation.

(b) The second reason for using multiple disk units is that several of the above applications involve huge amounts of data, which do not fit in one disk. For example, NASA expects 1 Terabyte ($=10^{12}$) of data per day; this corresponds to 10^{16} bytes of satellite data per year. Geographic databases can be large; for example, the TIGER database mentioned above is 19 Gigabytes. Historic and temporal databases tend to archive all the changes and tend to grow quickly in size.

The target system is intended to operate as a server, responding to range queries of concurrent users. Our goal is to maximize the throughput, which translates into the following two requirements:

'minLoad' Queries should touch as few nodes as possible, imposing a light load on the I/O sub-system. As a corollary, queries with small search regions should activate as few disks as possible.

'uniSpread' Nodes that qualify under the same query should be distributed over the disks as uniformly as possible. As a corollary, queries that retrieve much data should activate as many disks as possible.

The proposed hardware architecture consists of one processor with several disks attached to it. Multi-processor architectures are still under study [49]

On this architecture, we will distribute the nodes of a traditional R-tree. We propose and study several heuristics in order to determine how to choose a disk on which to place a newly created R-tree node. The most successful heuristic, based on the 'proximity index', estimates the similarity of the new node with the other R-tree nodes already on a disk, and chooses the disk with content having the least degree of similarity. Experimental results have shown that our scheme consistently outperforms other heuristics.

The rest of this chapter is organized as follows. Section 3.2 proposes the 'multiplexed' R-tree as a way to store an R-tree on multiple disks. Section 3.3 examines alternative criteria for choosing a disk for a newly created R-tree node. It also introduces the 'proximity' measure and derives the formulas for it. Section 3.4 presents experimental results and observations. Section 3.5 gives some concluding remarks.

3.2 Alternative Designs

The underlying file structure is the R-tree. Given that, our goal is to design a server for spatial objects on a parallel architecture in order to achieve high throughput under concurrent range queries.

The first step is to select the hardware architecture. For the reasons mentioned in the introduction, we propose a single processor with multiple disks attached to it. The next step is to decide how to distribute an R-tree over multiple disks. There are three major approaches: (a) d independent R-trees, (b) Disk stripping (or 'super-nodes', or 'super-pages'), and (c) the 'Multiplexed' R-tree, or *MUX R-tree* for short, which we describe and propose later. We examine the three approaches qualitatively:

3.2.1 Independent R-trees

In this scheme we can distribute the data rectangles among the d disks and build a separate R-tree index for each disk. This works primarily for unsynchronized disks. The performance will depend on how we distribute the rectangles over the different disks. There are two major approaches:

Data Distribution. The data rectangles are assigned to the different disks in a round robin fashion, or through the use of a hashing function. The data load (number of rectangles per disk) will be balanced. However, this approach violates the minimum load ('minLoad') requirement: even small queries will activate all the disks.

Space Partitioning. In this method the space is divided into d partitions, and each partition is assigned to a separate disk. For example, for the R-tree of Figure 3.1, we could assign nodes 1, 2, and 3 to disks A, B, and C, respectively. The children of each node follow their parent on the same disk. This approach will activate few disks on small queries, but it will fail to engage all disks on large queries, thus violating the uniform spread ('uniSpread') requirement.

3.2.2 Super-nodes

In this scheme we have only one large R-tree, with each node (='super-node') consisting of d pages; the i-th page is stored on the i-th disk ($i = 1, \ldots, d$). To retrieve a node from the R-tree, we read in parallel all d pages that constitute this node. In other words, we 'stripe' the super-node on the d disks, using page-striping [27]. Almost identical performance will be obtained with bit- or byte-level striping.

This scheme can work both with synchronized and unsynchronized disks. However, this scheme violates the 'minimum load' requirement: regardless of the size of the query, all the d disks become activated.

3.2.3 Multiplexed (MUX) R-tree

In this scheme we use a single R-tree, with each node spanning one disk page. Nodes are distributed over the d disks, with pointers across disks. For example, Figure 3.2 shows one possible multiplexed R-tree, corresponding to the R-tree of Figure 3.1. The root node is kept in main memory while other nodes are distributed over the disks A, B, and C. For the multiplexed R-tree, each pointer contains a disk_id in addition to the page_id of the traditional R-tree. However,

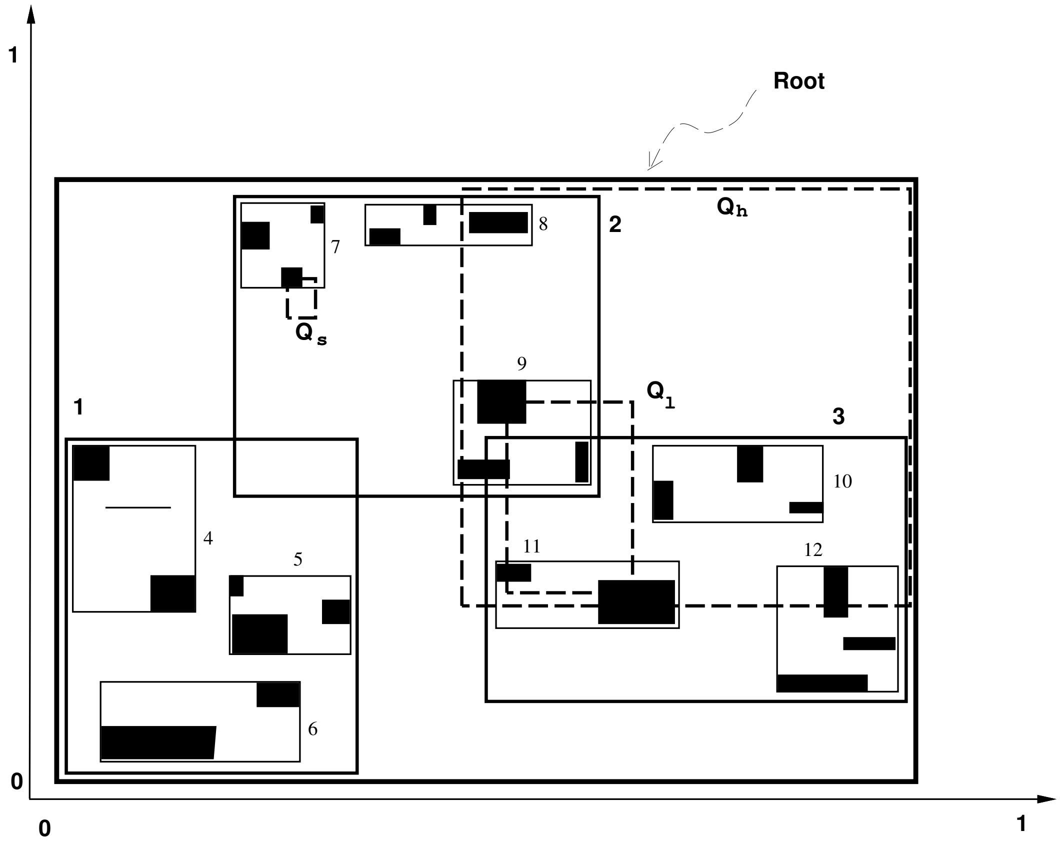


Figure 3.1: Data (dark rectangles) organized in an R-tree. Fanout=3. Dotted rectangles indicate queries.

the fanout of the node is not affected because the disk_id can be encoded within the four bytes of the page_id.

Notice that the proposed method fulfills both requirements (minLoad and uniSpread): For example, from Figure 3.2, we see that the 'small' query Q_s of Figure 3.1 will activate only one disk per level (disk B, for node 2, and disk A, for node 7), fulfilling the minimum load requirement. The large query Q_l will activate almost all the disks in every level (disks B and C at level 2, and then all three disks at the leaf level), fulfilling the uniform spread requirement.

Thus, with a careful node-to-disk assignment, the MUX R-tree should out-

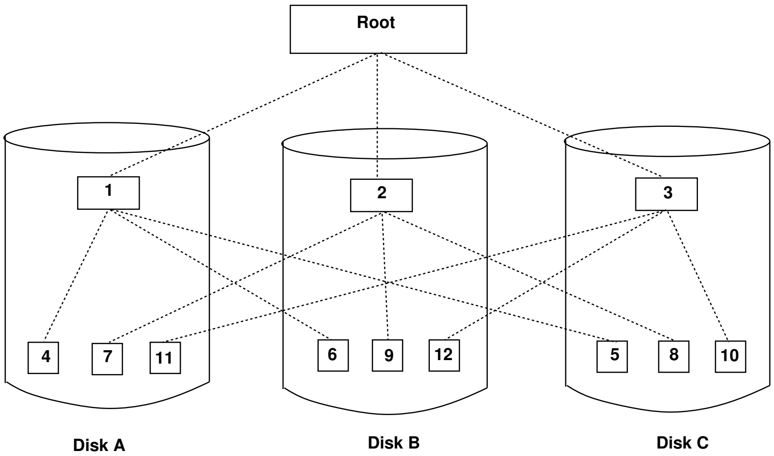


Figure 3.2: An R-tree stored on three disks.

perform both the methods that use super-nodes as well as the ones that use d independent R-trees. Our goal now is to find a good heuristic for assigning nodes to disks.

By its construction, the multiplexed R-tree fulfills the minimum load requirement. To meet the uniform spread requirement, we must find a good heuristic for assigning nodes to disks. In order to measure the quality of such heuristics, we shall use the response time as a criterion; response time is calculated as follows.

Let $R(q)$ denote the response time for the query q. We must first discuss how the search algorithm operates. Given a range query q, the search algorithm needs a queue of nodes, which is manipulated as follows:

Algorithm 1: Range Search

S1. Insert the root node of the R-tree in the processing queue.

S2. While (more nodes in queue)

- Pick the next node n from the processing queue.
- Process node n by checking for intersections with the query rectangle.

If this is a leaf node, print the results; otherwise, send a list of requests to some or all of the d disks, in parallel and insert their node-id's into the FIFO queue

Since the CPU is much faster than the disk, we assume that the CPU time is negligible (=0) compared to the time required by a disk to retrieve a page. Thus, the measure for the response time is the time (in terms of number of disk accesses) required by the latest disk to finish servicing the query. The 'disk-time' diagram helps visualize this concept better. Figure 3.3 presents the 'disk-time' diagram for the query Q_l of Figure 3.1. The horizontal axis is time, which is divided into slots. The duration of each slot is the time for a disk access and is considered constant. The diagram indicates when each disk is busy, as well as the page it is seeking, during each time slot. Thus, the response time for Q_l is 2, while its load $L(Q_l)$= 4, because Q_l retrieved four pages total.

As another example, the 'huge' query Q_h of Figure 3.1 results in the disk-time diagram of Figure 3.4, with response time $R(Q_h)$=3, and a load of $L(Q_h)$=7.

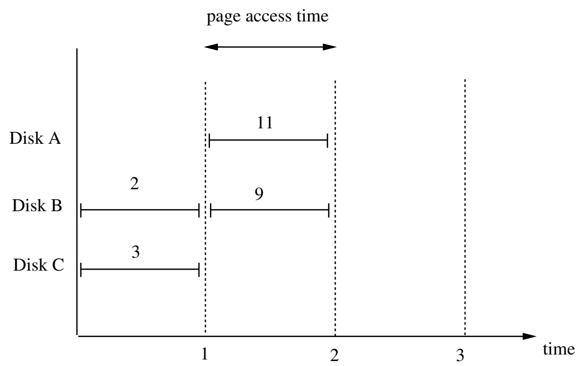


Figure 3.3: Disk-Time diagram for the large query Q_l.

Given the above examples, we have the following definition for the response time:

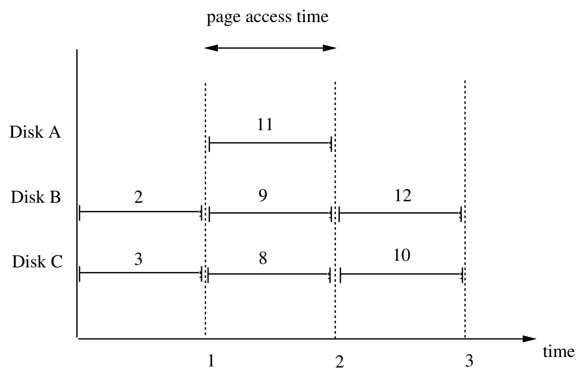


Figure 3.4: Disk-Time diagram for the huge query Q_h.

Definition 1 (Response Time) . The response time $R(q)$ for a query q is the response time of the latest disk in the disk-time diagram.

3.3 Disk Assignment Algorithms

The problem we examine in this section is how to assign nodes to disks within the Multiplexed R-tree framework. The goal is to minimize the response time and to satisfy the requirement for uniform disk activation ('uniSpread'). As discussed before, the minimum load requirement is fulfilled.

When a node (page) in the R-tree overflows, it is split into two nodes. One of these nodes, say, N_0, has to be assigned to another disk. If we carefully select this new disk we can improve the search time. Let $diskOf()$ be the function that maps nodes to the disks in which they reside. Ideally, we should consider all the nodes that are on the same level with N_0, before we decide where to store N_0. Such consideration, however, would require too many disk accesses. Thus, we consider only the *sibling* nodes $N_1, \ldots, N_k$, that is, the nodes that have the same father N_{father} as N_0. Accessing the father node comes at no extra cost, because we have to bring it into main memory anyway to insert N_0. Notice that we do not need to access the sibling nodes $N_1, \ldots, N_k$ because all the information we need about them (extent of MBR and disk of residence) are recorded in the father node.

Thus, the problem can be informally abstracted as follows:

Problem 1: Disk assignment

Given a node (= rectangle) N_0, a set of nodes $N_1, \ldots, N_k$ and the assignment of nodes to disks (diskOf() function)

Assign N_0 to a disk in such a way as to maximize the response time on range queries.

There are several criteria that we have considered:

Data balance: Ideally, all disks should have the same number of R-tree nodes. If a disk has many more pages than do other disks, it is more likely to become a 'hot spot' during query processing.

Area balance: Since we are storing not only points but also rectangles, the area of the pages stored on a disk is another factor. A disk that covers a larger area than the rest is again more likely to become a hot spot.

Proximity: Another factor that affects the search time is the spatial relation between the nodes that are stored on the same disk. If two nodes intersect, or are close to each other, they should be stored on different disks to maximize parallelism.

We can not satisfy all these criteria simultaneously because some of them may conflict. We now describe some heuristics, each of which attempts to satisfy one or more of the above criteria. In Section 3.4 we compare these heuristics experimentally.

Round Robin ('RR'). When a new page is created by splitting, this criterion assigns it to a disk in a round robin fashion. Without deletions, this scheme achieves perfect data balance. For example, in Figure 3.5, RR will assign N_0 to the least populated disk, that is, disk C.

Minimum Area ('MA'). This heuristic tries to balance the area of the disks: When a new node is created, the heuristic assigns it to the disk that has the smallest area covered. For example, in Figure 3.5, MA would assign N_0 to disk A, because the light gray rectangles N_1, N_3, N_4 and N_6 of disk A have the smallest combined area.

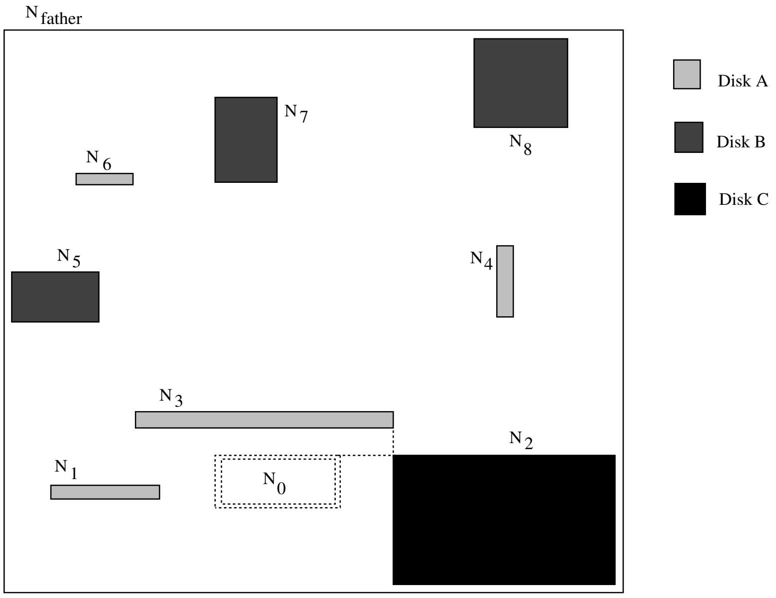


Figure 3.5: Node N_0 is to be assigned to one of the three disks.

Minimum Intersection ('MI'). This heuristic tries to minimize the overlap of nodes that belong to the same disk. Thus, it assigns a new node to a disk, such that the new node intersects as little as possible with the other nodes on that disk. Ties are broken using one of the above criteria.

Proximity Index ('PI'). This heuristic is based on the *proximity measure*, which we describe in detail in the next subsection. Intuitively, this measure compares two rectangles and assesses the probability that they will be retrieved by the same query. As we shall soon see, this procedure is related to the Manhattan (or city-block or L_1) distance. Rectangles with high proximity (i.e., intersecting, or close to each other) should be assigned to different disks. The *proximity index* of a new node N_0 and a disk D (which contains the sibling nodes $N_1, \ldots, N_k$) is the proximity of the most 'proximal' node to N_0. A metric for the proximity index (as a function of the proximity measure) is explained in the next section.

The algorithm works as follows: It calculates the proximity index between the new node N_0 and each of the available disks. Then it assigns node N_0 to the disk with the lowest proximity index, i.e., to the disk with the least similar nodes with respect to N_0. Ties are resolved using the number of nodes (data balance): N_0 is assigned to the disk with the fewest nodes. For the setting of Figure 3.5, PI will assign N_0 to disk B because it contains the most remote rectangles (least Proximity Index). Intuitively, disk B is the best choice for N_0.

Although favorably prepared, the example of Figure 3.5 indicates that PI should perform better than the rest of the heuristics. Next we show how to calculate exactly the 'proximity measure' of two rectangles.

3.3.1 Proximity Measure

Whenever a new R-tree node N_0 is created, it should be placed on the disk that contains nodes (= rectangles) that are as dissimilar to N_0 as possible. Here we try to quantify the notion of similarity between two rectangles. The proposed measure can be trivially generalized to hold for hyper-rectangles of any dimensionality. For clarity, we examine one- and two- dimensional spaces first.

Intuitively, two rectangles are similar if they qualify often under the same query. Thus, a measure of similarity of two rectangles R and S is the proportion of queries that retrieve both rectangles. Thus,

$$proximity(R, S) = \text{Prob } \{ \text{ a query retrieves both } R \text{ and } S \text{ } \}$$

or, formally

$$proximity(R, S) = \frac{\#of\ queries\ retrieving\ both}{total\#\ of\ queries} = \frac{|q|}{|Q|} \tag{3.1}$$

To avoid complications with infinite numbers, let us assume during this subsection that our address space is discretized, with very fine granularity. (The case of a continuous address space will be the limit for infinitely fine granularity).

Based on the above definition, we can derive the formulas for proximity, given the coordinates of the two rectangles R and S. To simplify the presentation, let us consider the one-dimensional case first.

One-d Case

Without loss of generality, we can normalize our coordinates, and assume that all our data segments lie within the unit line segment [0,1]. Consider two line segments R and S where R=(r_{start}, r_{end}) and S=(s_{start}, s_{end}).

If we represent each segment X as the point $(x_{start},\ x_{end})$, the segments R and S are transformed into two-dimensional points [35] as shown in Figure 3.6. In the same Figure, the area within the dashed lines is a measure of the number

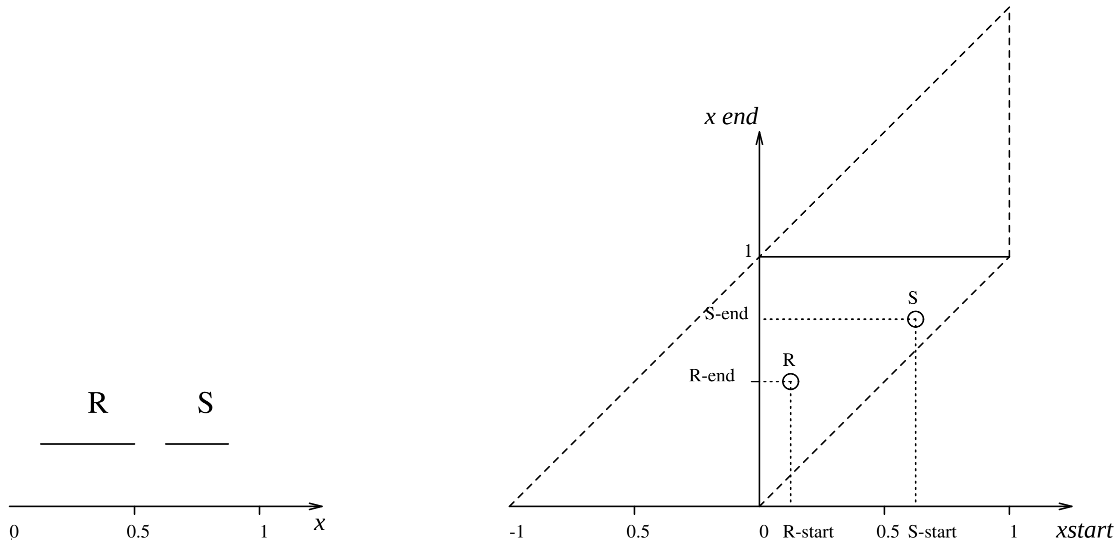


Figure 3.6: Mapping line segments to points.

of all the possible query segments, ie, queries whose size is ≤ 1 and who intersect the unit segment. There are two cases to consider, depending on whether R and S intersect or not. Without loss of generality, we assume that R starts before S (i.e., $r_{start} \leq s_{start}$).

(a) R and S intersect. Let 'I' denote their intersection, and let δ be its length. Every query that intersects 'I' will retrieve both segments R and S. The total number $|Q|$ of possible queries is proportional to the trapezoidal area within the dashed lines in Figure 3.6; its area is

$$|Q| = \frac{(2 \times 2 - 1 \times 1)}{2} = \frac{3}{2} \tag{3.2}$$

The total number of queries $|q|$ that retrieve both R and S is proportional to the shaded area of Figure 3.7

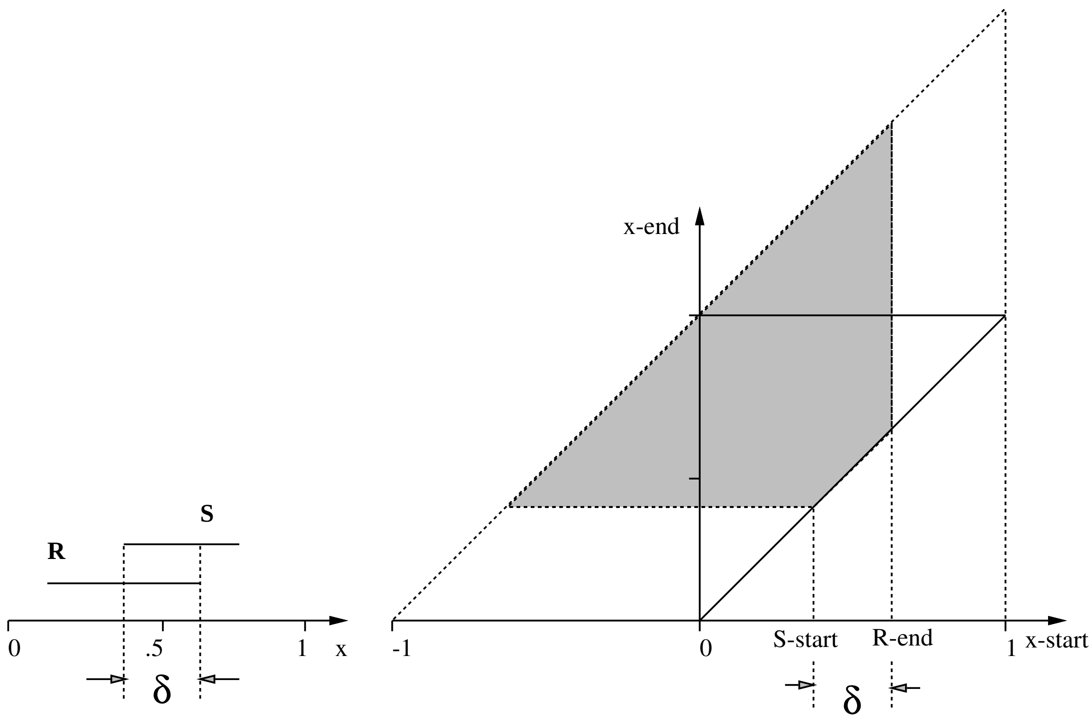


Figure 3.7: Intersecting line segments; the shaded area contains all the segments that intersect R and S.

$$|q| = ((1+\delta)^2 - \delta^2)/2 = \frac{1}{2} \times (1 + 2 \times \delta) \tag{3.3}$$

Thus, for intersecting segments R and S we have

$$proximity(R, S) = \frac{|q|}{|Q|} = \frac{1}{3} \times (1 + 2 \times \delta) \tag{3.4}$$

where δ is the length of the intersection

(b) R and S are disjoint, with distance Δ between them (see Figure 3.8). In this case, a query has to cover the segment (r_{end}, s_{start}), in order to retrieve both segments. The number of such queries is proportional to the shaded

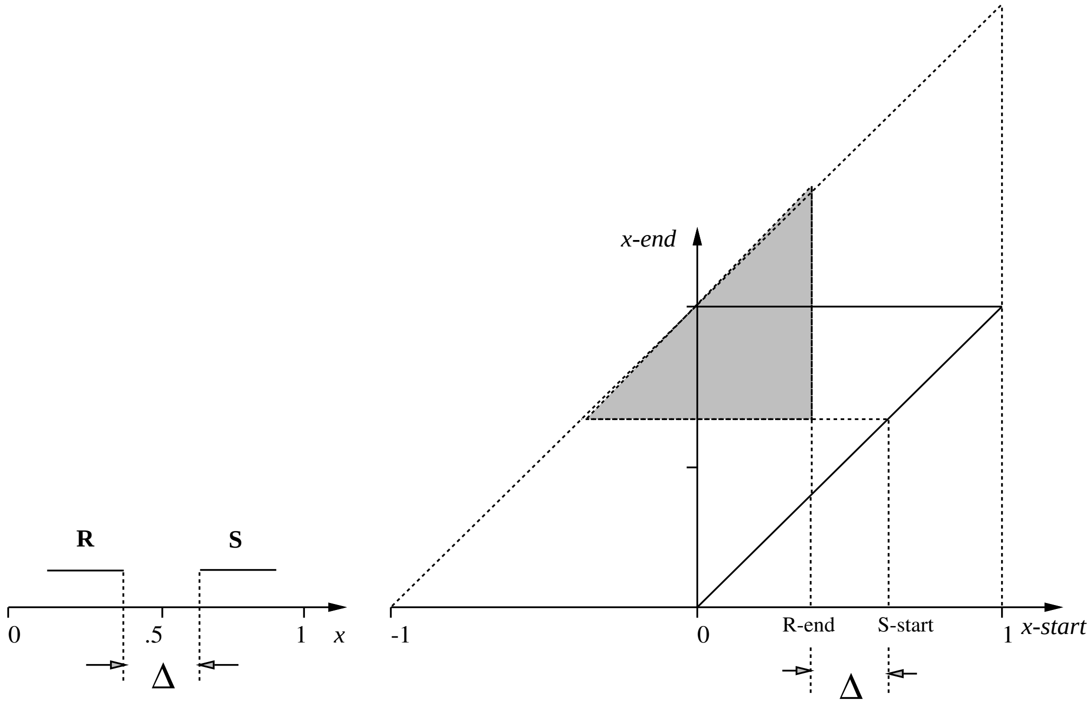


Figure 3.8: Disjoint line segments; the shaded area contains all the segments that intersect R and S.

area in Figure 3.8.b; its area is given by

$$q = \frac{1}{2} \times (1 - \Delta)^2 \tag{3.5}$$

and the proximity measure for R and S is

$$proximity(R, S) = \frac{|q|}{|Q|} = \frac{1}{3} \times (1 - \Delta)^2 \tag{3.6}$$

Notice that the two formulas agree when R and S just touch: in this case, $\delta = \Delta = 0$ and the proximity is $1/3$.

N-d Case

For the 2-d case, the previous formulas can be generalized by assuming uniformity and independence: Let R and S be two data rectangles, with R_x, R_y

denoting the x and y projections of R. A query X will retrieve both R and S if and only if (a) its x-projection X_x retrieves both R_x and S_x *and* (b) its y-projection X_y retrieves both R_y and S_y.

Since the x and y sizes of the query rectangles are independent, the fraction of queries that meet both of the above criteria is the product of the fractions for each individual axis; i.e., the proximity measure $proximity_2()$ in two dimensions is given by:

$$proximity_2(R,S) = proximity(R_x, S_x) \times proximity(R_y, S_y) \qquad (3.7)$$

The generalization for n-dimensions is straightforward:

$$proximity_n(R,S) = \prod_{i=1}^{n} proximity(R_i, S_i) \qquad (3.8)$$

where R_i and S_i are the projections on the i-th axis, and the $proximity()$ function for segments is given by Eqs. 3.4 and 3.6.

The proximity *index* measures the similarity of a rectangle R_0 to a set of rectangles $\mathcal{R} = \{R_1, \ldots, R_k\}$. We need this concept to assess the similarity of a new rectangle R_0 and a disk D containing the rectangles of the set $\mathcal{R}$. The proximity index is the proximity of the most similar rectangle in $\mathcal{R}$. Formally:

$$proximityIndex(R_0, \mathcal{R}) = \max_{R_i \in \mathcal{R}} proximity_n(R_0, R_i) \qquad (3.9)$$

where $R_i \in \mathcal{R}$, and n is the dimensionality of the address space.

In Figure 3.9, we calculate the proximity measure between N_0 and each of its siblings, namely N_5, N_7, and N_8 and then pick the largest proximity measure as a value for the proximity index between N_0 and disk A (= 0.15 in this example).

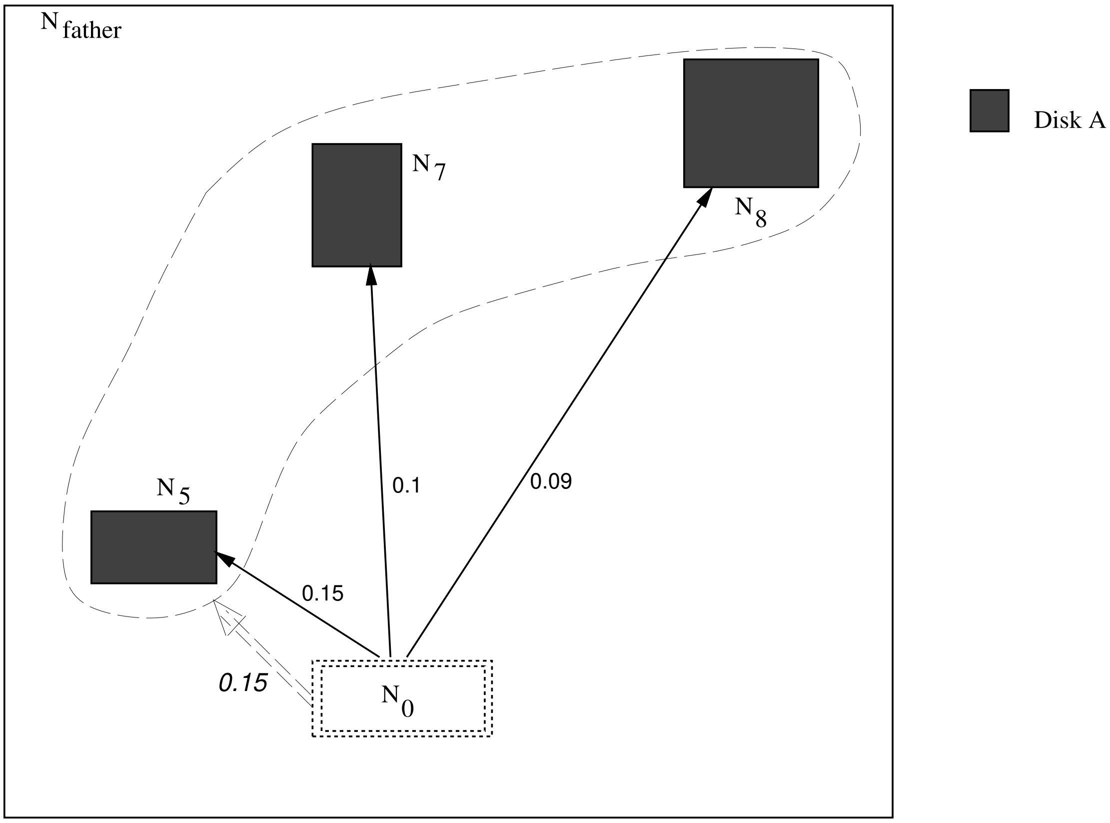
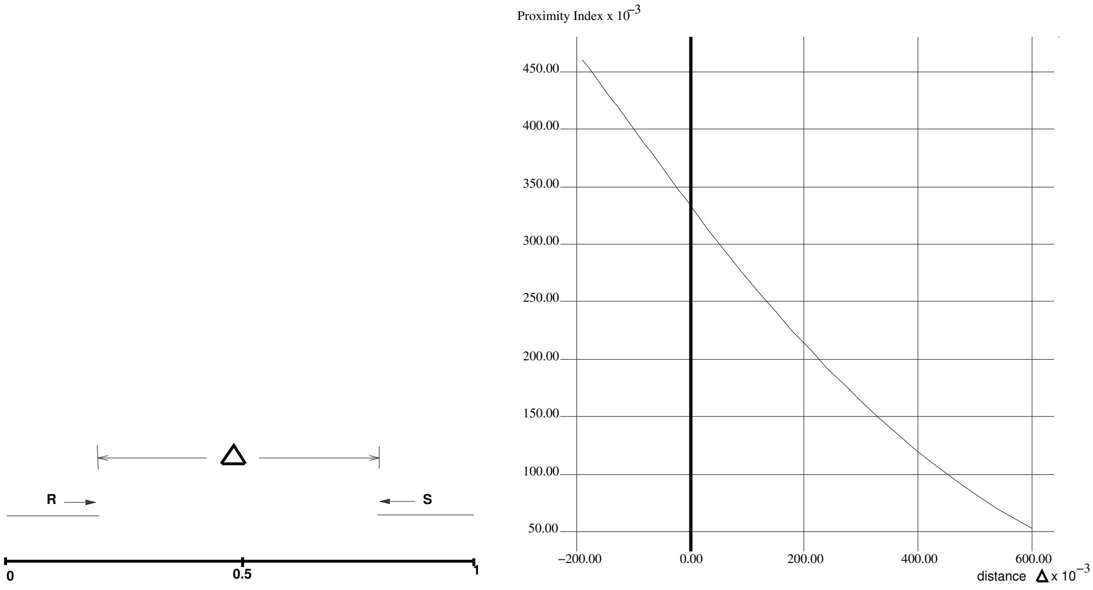


Figure 3.9: The proximity index between the node N_0 and the set $\{N_5, N_7, N_8\}$. The numbers next to the solid arrows show the proximity measure.

3.3.2 Observations

As we can see, the proximity measure can take any real value in the range $[0, 1]$ with its value increasing with the similarity between the two rectangles. Next, we present some arithmetic examples to illustrate that the proximity measure behaves as intuitively expected. As before, a one-dimensional segment X is represented by its starting and ending coordinate $[x_{start}, x_{end}]$. In the one-dimensional case we have

- $proximity([0,0],[1,1])$=0, which says that the two extreme points have the minimum possible proximity.



(a) (b)

Figure 3.10: (a) Two equal line segments R, S (length(R) = 0.2) moving toward each other (b) Proximity index(R, S) values as a function of their distance Δ.

- $proximity([x, x+\delta], [x, x+\delta]) = (1+2\delta)/3$, that is, the proximity of a segment with itself increases with the size of the segment, reaching the maximum value of 1 when the segment covers the whole space

- the proximity measure is 1/3 for two segments that touch in a point; it is larger for two intersecting segments, depending on the length of the intersection; it is <1/3 for non-intersecting segments, depending on their distance.

Figure 3.10 shows how the proximity index value changes in one-dimensional space. R and S are two line segments (one-dimensional rectangles) of length 0.2 each. At the beginning, they are placed at the two opposite ends of the space. Figure 3.10(b) shows the proximity index values between R and S as they move toward each others. When R and S are separate from each other (no intersection), the separation is measured by the distance between the edges of R and S and is represented by positive values. On the other hand, when R and S intersect, the length of the intersection represented as a negative distance in Figure 3.10(b). Note that the proximity index decreases quadratically with the increase in the distance between the non-intersecting segments while it increases linearly with increasing the overlap region for the intersecting segments.

In the two-dimensional case, the proximity measure is better than the (inverse of) the Manhattan distance:

- For overlapping rectangles, the Manhattan distance is zero, regardless of the area of overlap. On the contrary, the proximity measure takes into account not only the area, but the perimeter of the intersection as well.

- For disjoint rectangles, the Manhattan distance ignores the relative position of the rectangles. For example, in Figure 3.11, the rectangles R and T have the same Manhattan distance from the rectangle S. Intuitively, R is 'more similar' to S than T is to S. The proximity measure reflects this fact: $proximity_2(R, S)= 0.126 > 0.09 = proximity_2(T, S)$.

In conclusion, the behavior of the proximity measure completely agrees with our intuition: It related to the inverse Manhattan distance of two objects; in addition, it takes into account the relative position of the objects, and it handles overlapping objects correctly.

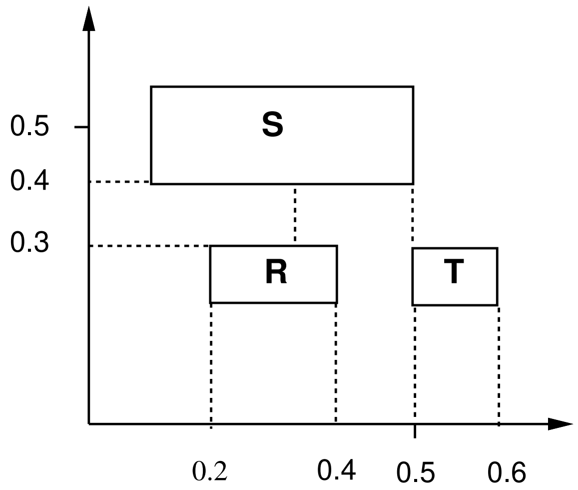


Figure 3.11: Example illustrating the accuracy of the proximity index over the Manhattan distance.

3.4 Experimental Results

To assess the merit of the proximity index heuristic over the other heuristics, we ran simulation experiments on two-dimensional rectangles. We augmented the original R-tree code with some routines to handle the multiple disks (e.g., ‘choose_disk()’, ‘proximity()’, etc.) The code is written in C under Ultrix and the simulation experiments ran on a DECstation 5000. We used both the linear and the quadratic splitting algorithm of Guttman [32]. The quadratic algorithm resulted in better R-trees, i.e., R-trees with smaller father nodes. The exponential algorithm was very slow and was not used. Unless otherwise stated, all the results we present are based on R-trees that used the quadratic split algorithm.

In our experiments we assume that

- all $\mathcal{D}$ disk units are identical.
- the page access time is constant.

Symbols	Definitions
a	average area of a data rectangle
τ	data density ('cover quotient')
$\mathcal{D}$	number of disks
$diskOf()$	maps nodes to disks
$L(q)$	'Load': total number of pages touched by query q
N	number of data rectangles
p	size of a disk page in Kbytes
$proximity_n()$	proximity of two n-d rectangles
q_s	side of a query rectangle
$R(q)$	response time for query q (in disk accesses)
$r(q)$	relative response time (compared to PI)
s	speed-up

Table 3.1: Summary of symbols and definitions.

- the first two levels of the Multiplexed R-tree (the root and its children) fit in main memory. The required space is of the order of 100 Kb, which is a modest requirement even for personal computers.

- the CPU time is negligible. As discussed before, the CPU is two orders of magnitude faster than the disk. Thus, for the number of disks we have examined (1-25 disks), the delay caused by the CPU is negligible. In the following experiments, we use the number of disk accesses as a measuring unit for the query response time.

Without loss of generality, the address space was normalized to the unit square. There are several factors that affect the search time. We used real data as well as synthetic data. The reason for using synthetic data is that we have better control over the several parameters that characterize the data set. One real data set comes from the TIGER files (Bureau of Census). It consists of 39,717 line segments, representing the roads of Montgomery County in Maryland. Using the minimum bounding rectangles of the segments, we obtained 39,717 rectangles, with data density $\tau = 0.35$. We refer to this dataset as the *'MGCounty'* dataset. Another dataset, which came from NASA, consists of 11,284 observation points from the International Ultraviolet Explorer (IUE) satellite. We refer to this dataset as the *'IUE'* dataset. It is important to note that these data sets are non-uniform and highly skewed. We studied the following input parameters:

The number of disks $\mathcal{D}$: It ranged from 5-25.

The total number of data rectangles N: It ranged from 11,000 to 200,000 rectangles.

The size of queries $q_s \times q_s$: The query side q_s ranged from 0 (point queries) to 0.25.

The page size p: It ranged from 1Kb to 4Kb.

Another important factor, which is derived from N and the average area a of the data rectangles, is the "data density" τ (or "cover quotient") of the data rectangles. This is the sum of the areas of the data rectangles in the unit square, or equivalently, the average number of rectangles that cover a randomly selected point. Mathematically: $\tau = N \times a$. For the selected values of N and a, the data density ranges from 0.25 to 2.0.

The synthetic data rectangles were generated as follows: Their centers were uniformly distributed in the unit square; their x and y sizes were uniformly distributed in the range [0,max], where $max = 0.006$

The query rectangles were squares with side q_s. Their centers are uniformly distributed in the unit square. For every experiment, 100 randomly generated queries were asked and the results were averaged. Data or query rectangles that were not completely inside the unit square were clipped. The proximity index heuristic performed very well in our experiments and is therefore the proposed approach.

In the following subsections, we present: (a) A comparison among the node-to-disk assignment heuristics (MI, MA, RR and PI); recall that they are all within the Multiplexed R-tree framework. (b) A comparison of the proximity index versus Round Robin. (c) A comparison of the MUX R-tree + PI versus the super-node method. (d) A study of the speed-up achieved by PI.

3.4.1 Comparison of the Disk Assignment Heuristics

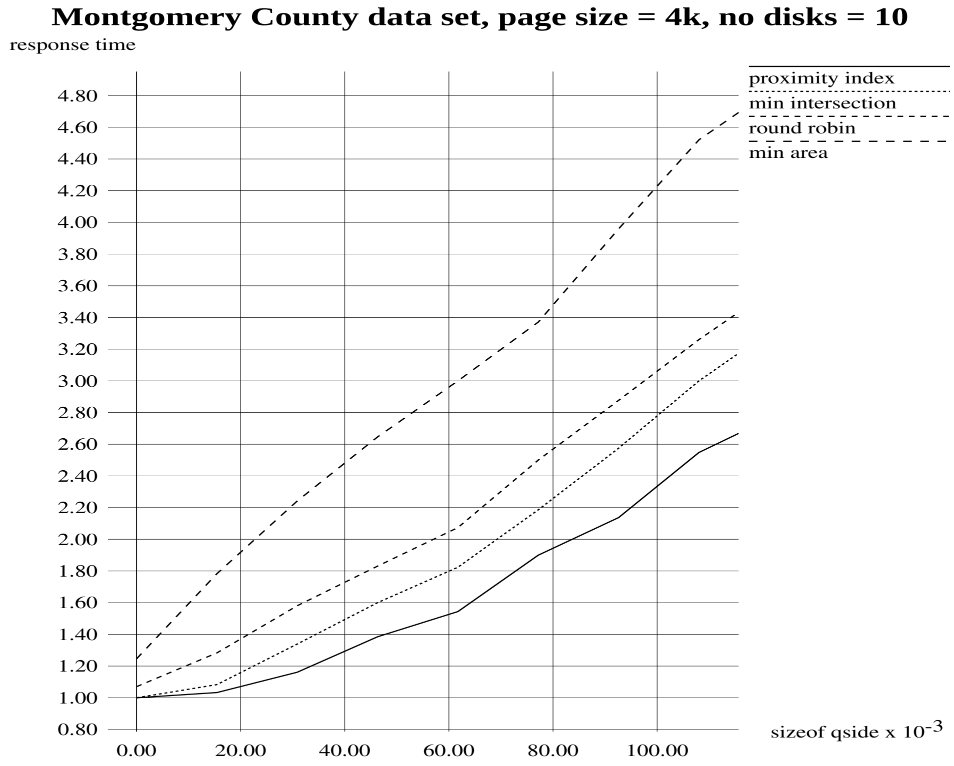


Figure 3.12: Comparison among all heuristics (PI,MI,RR and MA) – Real Data – TIGER file.

In this section we compare the real response time for each of the four heuristics (RR, MI, MA, PI), as a function of the relative query size q_s. Figures 3.12- 3.14 plots the response time (in terms of the number of disk accesses) as a function of the size of the query side. In Figure 3.12 we used the *MGCounty* dataset that represents the roads of Montgomery County, Maryland. Figure 3.13 shows the same experiment carried over the *IUE* dataset. In addition to the real datasets we used the synthetic dataset; Figure 3.14 consists only of rectangles with the following parameters : N=25,000 τ=0.26, $\mathcal{D}$=10, p=4.

Figures 3.12 - 3.14 show that the proximity index (PI) heuristic performs better than other heuristics. This behavior is typical for other real datasets

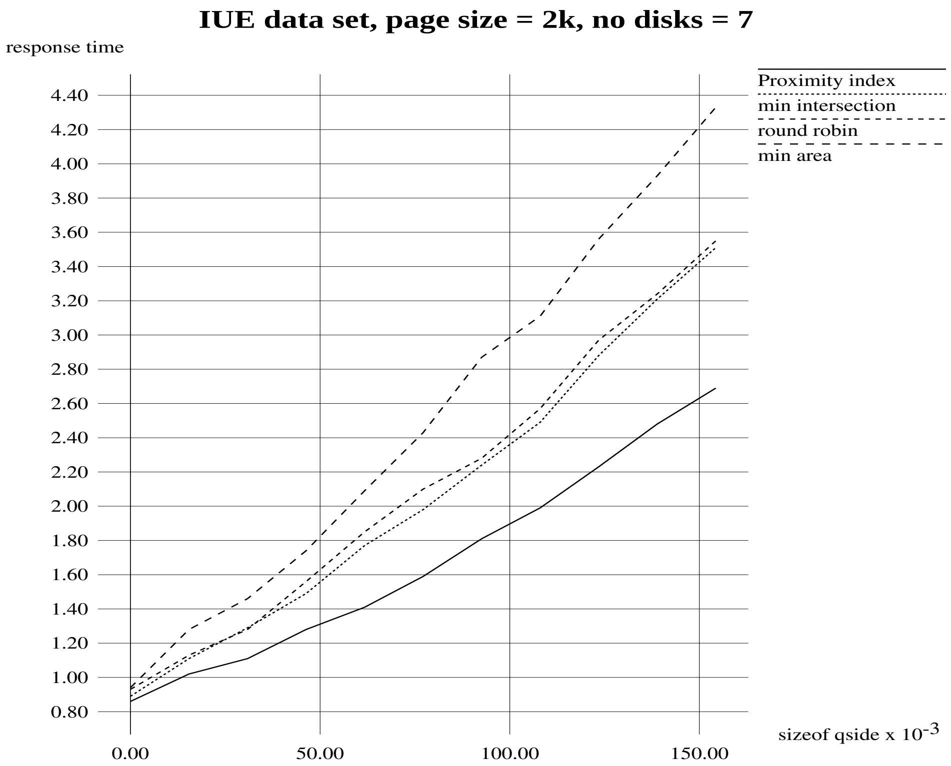


Figure 3.13: Comparison among all heuristics (PI,MI,RR and MA) – Real Data – IUE data set.

which we used and was consistent over several combinations of parameter values (for synthetic data) : p=1,2,4 Kb; τ=0.5,1,2; $\mathcal{D}$=5,10,20. The main observation is that PI and MI, the two heuristics which take the spatial relationships into account, perform the best. Round Robin is the next best, while the Minimum Area heuristic demonstrates the worst performance.

Comparing the MI and PI heuristics, we see that MI performs as well as the proximity index heuristic for small queries; for larger queries, the proximity index wins. The reason is that MI may assign the same disk to two non-intersecting rectangles that are very close to each other.

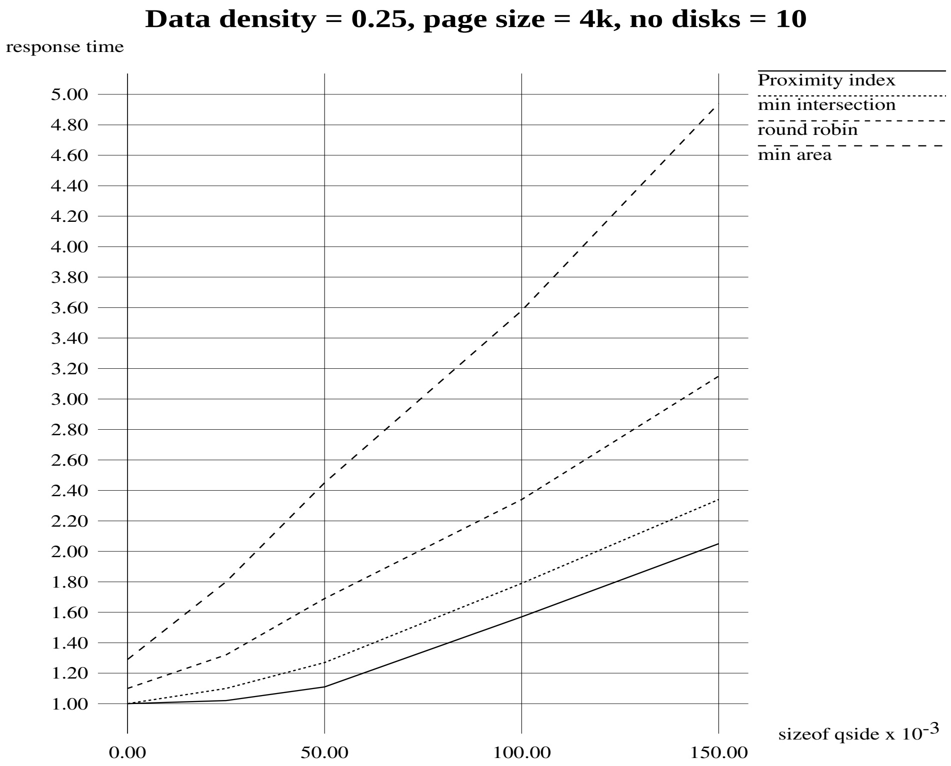


Figure 3.14: Comparison among all heuristics (PI,MI,RR and MA) – Rectangles Only.

3.4.2 Proximity Index Versus Round Robin

Here we study the savings that the proposed heuristic PI can achieve over the RR. The reason we have chosen RR is because it is the simplest heuristic to design and implement. We show that the extra effort to design the PI heuristic pays off consistently.

To make the comparison easier, in this subsection we normalize the response time of the different heuristics to that of the proximity index and plot the ratios of the response times. Figure 3.15 plots the response time of RR relative to PI as a function of the query size q_s. The number of disks is $\mathcal{D}$=10, the data density is τ=0.26 and the page size p varied, with values of 1, 2 and 4Kb. We conclude

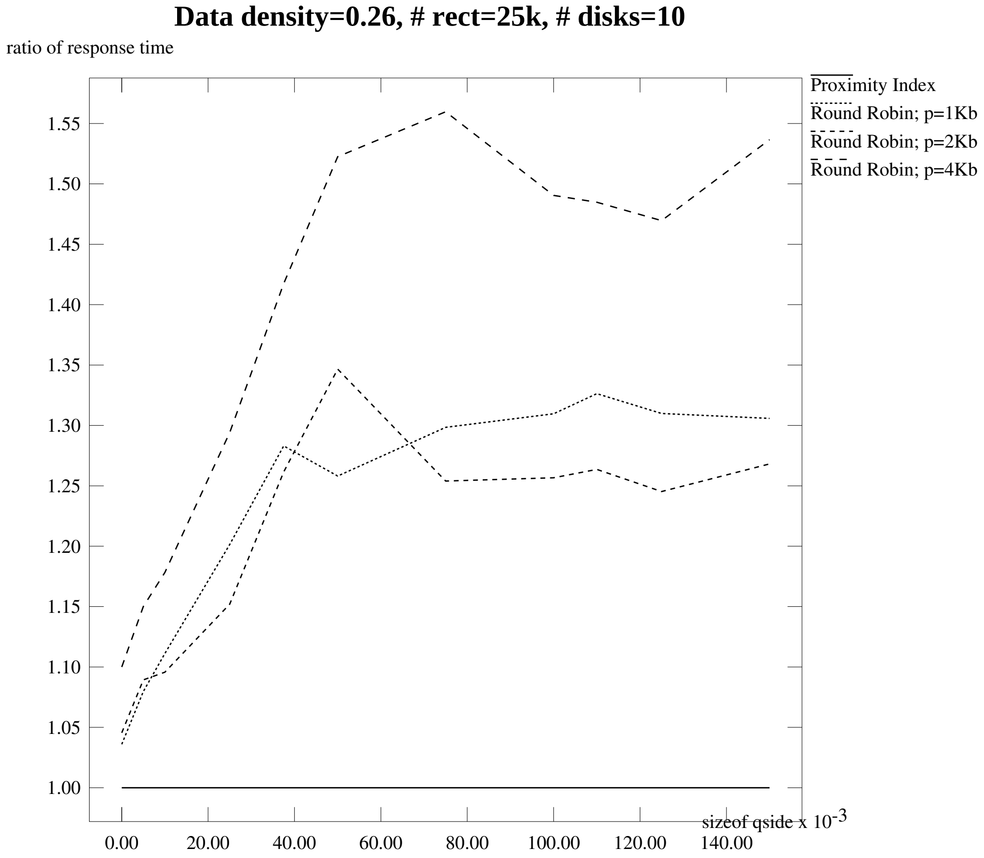


Figure 3.15: Relative response time (RR over PI) vs. query size for different page sizes.

that the gains of PI increase with increasing page size. This is because the PI heuristic considers only sibling nodes (nodes under the same father); with a larger page size, the heuristic takes more nodes into account and therefore makes better decisions.

Figure 3.16 illustrates the effect of the data density on the relative gains of PI over RR. The page size p was fixed at 4Kb; the data density varied (τ=0.26, 0.5, 1 and 2). Everything else was the same as in Figure 3.15. The main observation is that $R(q)$ decreases with the data density τ. This is explained

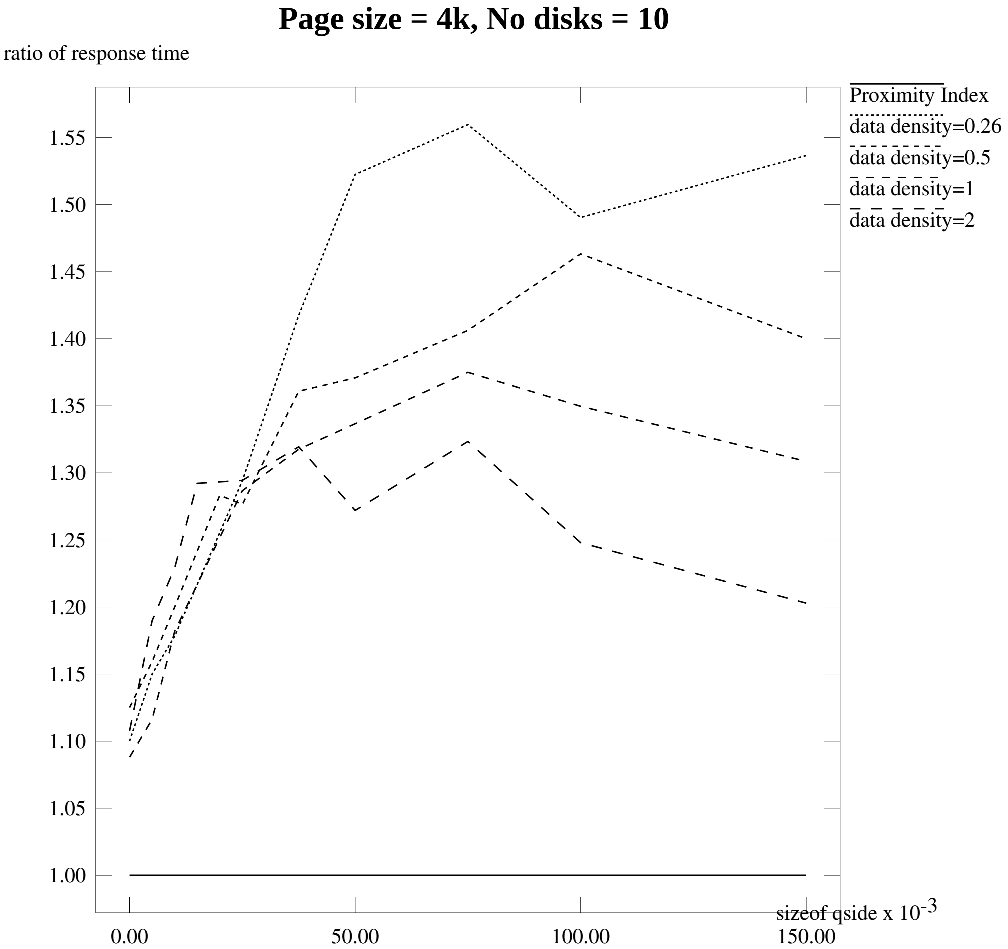


Figure 3.16: Relative response time (RR over PI) vs query size for different data densities.

as follows: For large τ, there are more rectangles in the vicinity of the newly created rectangle. With constant number of disk units, the probability that two rectangles retrieved by constant query size from the same disk increases. To improve the performance in this case, we need to take more sibling nodes into account in the disk assignment algorithm (by increasing the node size) and use more disk units.

An observation common to both Figures is that $r(q)$ peaks for medium size queries. For small queries, the number of nodes to be retrieved is small, leaving little room for improvement. For huge queries, almost all the nodes need to be retrieved, in which case the data balance of RR achieves good results.

We ran experiments with the linear split algorithm of the R-tree. The PI heuristic outperformed the RR consistently, with smaller relative gains, however. The peak gain was ≈20-30%, instead of the 30-60% that we achieved in Figures 3.15 and 3.16. This difference occurs because the proximity index anticipates that most of the nodes that are close to the new node will be under the same father. Linear splitting, however, does not pack nodes together as well as does quadratic splitting. As a result, in the linear splitting R-tree, many nodes that are close to the new node will not be considered by the PI algorithm because they are not siblings.

3.4.3 Comparison with the Super-node Method

In order to justify our claims about the advantages of the Multiplexed ('MUX') R-tree over the super-node method, we compared the two methods with respect to the two requirements, 'uniSpread' and 'minLoad'. The measure for the first requirement is the response time $R(q)$; the measure for the second is the load $L(q)$. We present graphs with respect to both measures.

Figure 3.17 compares the response time of the Multiplexed R-tree (with PI) against the super-node method. Notice that the difference in performance increases with the query size q_s. In general, the Multiplexed R-tree outperforms the super-node scheme for large queries. The only situation where the super-node scheme performs slightly better is when there are many disks $\mathcal{D}$ and the query is small. This phenomenon occurs because, since $\mathcal{D}$ is large, the R-tree

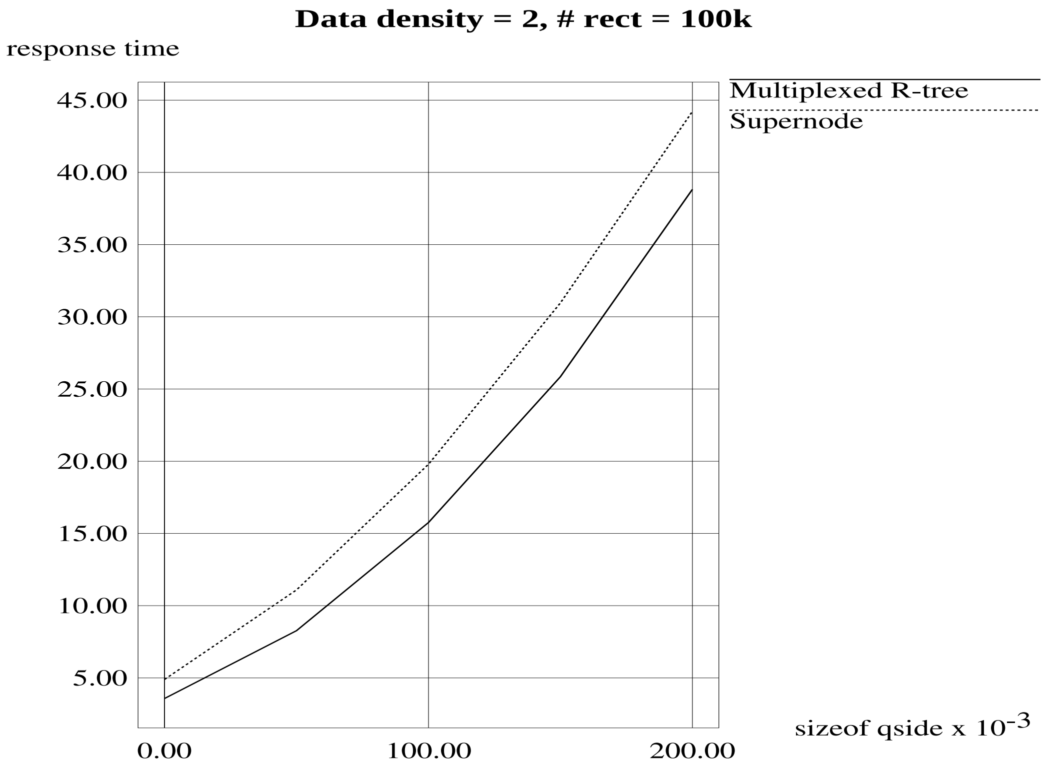


Figure 3.17: Response time vs. query size for Multiplexed R-tree with PI, and for super-nodes ($\mathcal{D}$=5 disks).

with super-nodes has fewer levels than does the Multiplexed R-tree; in addition, since the query is small, the response time of both trees is bounded by the height of the respective tree. However, this is exactly the situation where the super-node method violates the 'minimum load' requirement, imposing a large load on the I/O sub-system and paying penalties in throughput. In order to gain insight into the effect on the throughput, we plot the 'load' for each method with various parameter values. Recall that the load $L(q)$ for a query q is the total number of pages touched (1 super-page counts as $\mathcal{D}$ simple pages). Figure 3.18 shows the results for the same setting as before (Figure 3.17). The Multiplexed R-tree imposes a much lighter load: for small queries, its load is two to three times smaller

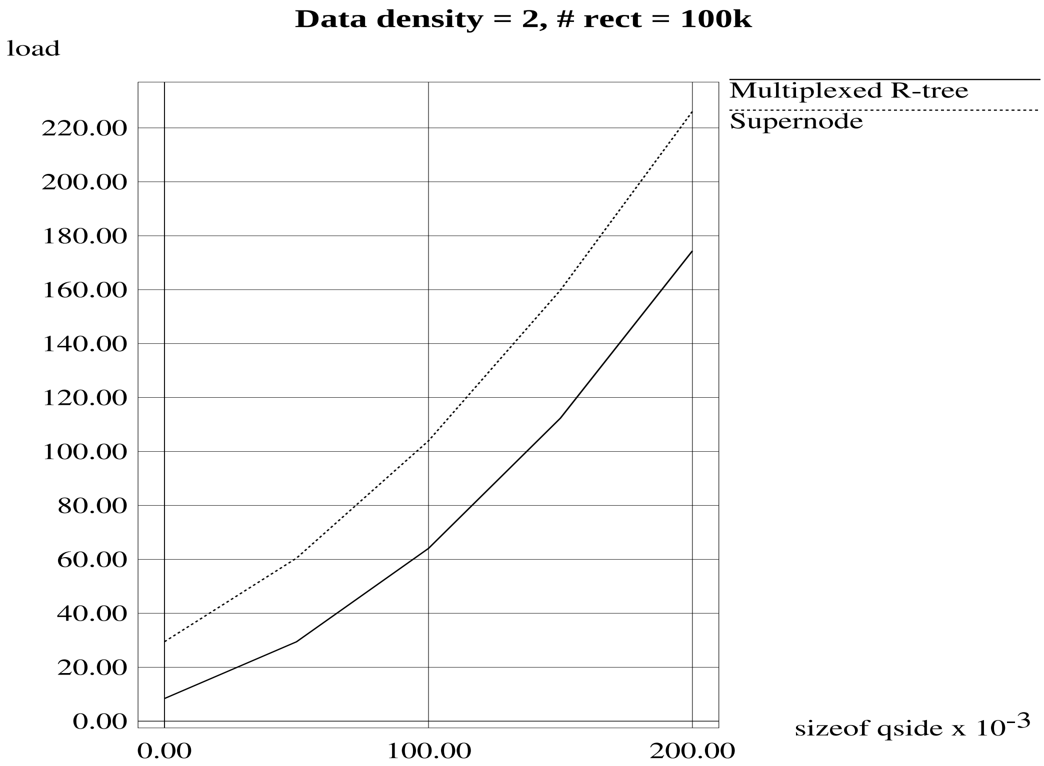


Figure 3.18: Total number of pages retrieved (load), vs. query size q_s – $\mathcal{D}$=5.

than the load of the super-node method. Interestingly, the absolute difference increases with the query size.

The conclusion of the comparisons is that the proposed method has better response time than the super-node method, at least for large queries. In addition, the proposed method will lead to higher throughput because it tends to impose lighter loads on the disks. Both results agree with our intuition and indicate that the proposed method will offer a higher throughput for a spatial object server.

3.4.4 Speed-up

The standard measure of the efficiency of a parallel system is the speed-up s, which is defined as follows: Let $R_d(q)$ be the response time for the query q on a system with $\mathcal{D}$ disks. Then:

$$s = \frac{R_1(q)}{R_d(q)} \tag{3.10}$$

In this subsection we examine exclusively the Multiplexed R-tree method with the PI heuristic because it seems to offer the best performance. Figure 3.19 shows the speed-up for data density τ=2, page size p=4Kb and for query side size q_s ranging from 0 to 0.25. The speed-up is high, e.g., 84% of the linear speed-up (for q_s=0.25). It achieves even higher values for smaller $\mathcal{D}$. Moreover, the speed-up increases with the size of the query, apparently because larger queries can take better advantage of more disks.

Conversely, small queries reach a plateau in their speed-up curve. The smaller the query size, the sooner the speed-up reaches the plateau. Figure 3.20 provides the explanation pictorially. It shows the actual response times versus the number of disks $\mathcal{D}$ for the very same setting. Notice that all curves approach the optimal bound, namely, the number of levels of the tree that are *not* in core. A small query will reach this bound quickly for a small $\mathcal{D}$. Thus, the flattening of the speed-up curves means that the respective queries enjoy minimal response time.

Finally, in Figure 3.21 we show how speed-up is affected by data density. The query size is fixed at q_s=0.25 and everything else remain the same. Increasing τ yields higher speed-ups exactly because the query retrieves more nodes and is therefore more amenable to parallelism.

3.5 Discussion

Using R-trees as the underlying file structure, we have studied alternative designs for a spatial object server, We focused on rectangular range queries. Our goal is

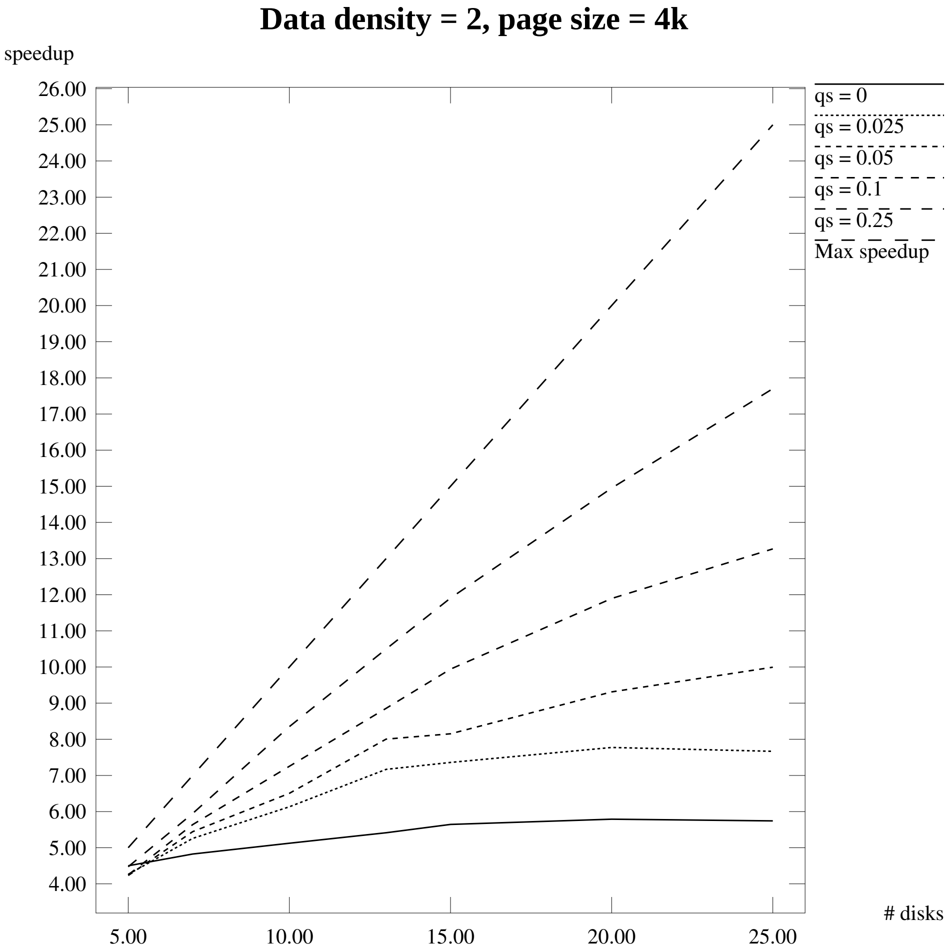


Figure 3.19: Speed-up of the Multiplexed R-tree vs. number of disks with data density = 2.

to maximize the parallelism for large queries, while at the same time engaging as few disks as possible for small queries. To achieve these goals, we propose

- a hardware architecture with one CPU and multiple disks; this architecture is simple, effective and inexpensive. It has no communication costs; it requires inexpensive, general-purpose components; it can easily be expanded (by simply adding more disks); and it can easily take advantage of large buffer pools.

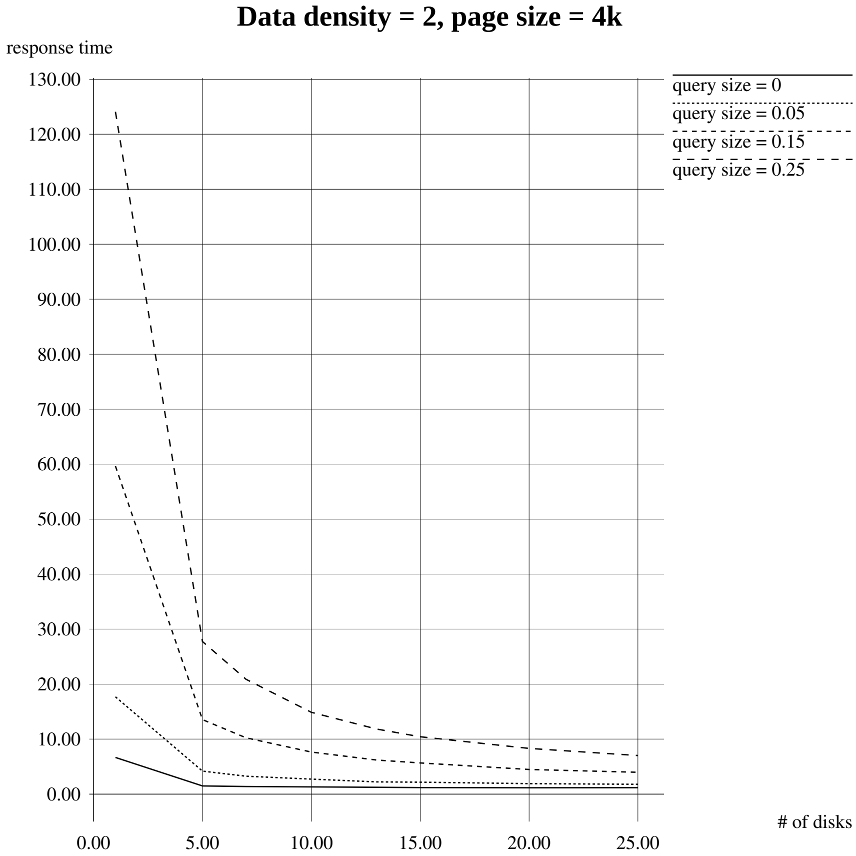


Figure 3.20: Response time of the Multiplexed R-tree vs. number of disks for different query sizes.

- a software architecture (termed 'Multiplexed' R-tree). It operates exactly like a single-disk R-tree, the only difference being that its nodes are carefully distributed over the $\mathcal{D}$ disks. Intuitively, this approach should be better than the super-node approach and the "independent R-trees" approach with respect to throughput.

- the "proximity index" (PI) criterion, which decides how to distribute the nodes of the R-tree on the $\mathcal{D}$ disks. Specifically, it tries to store a new node on that one disk that contains nodes as dissimilar to the new node

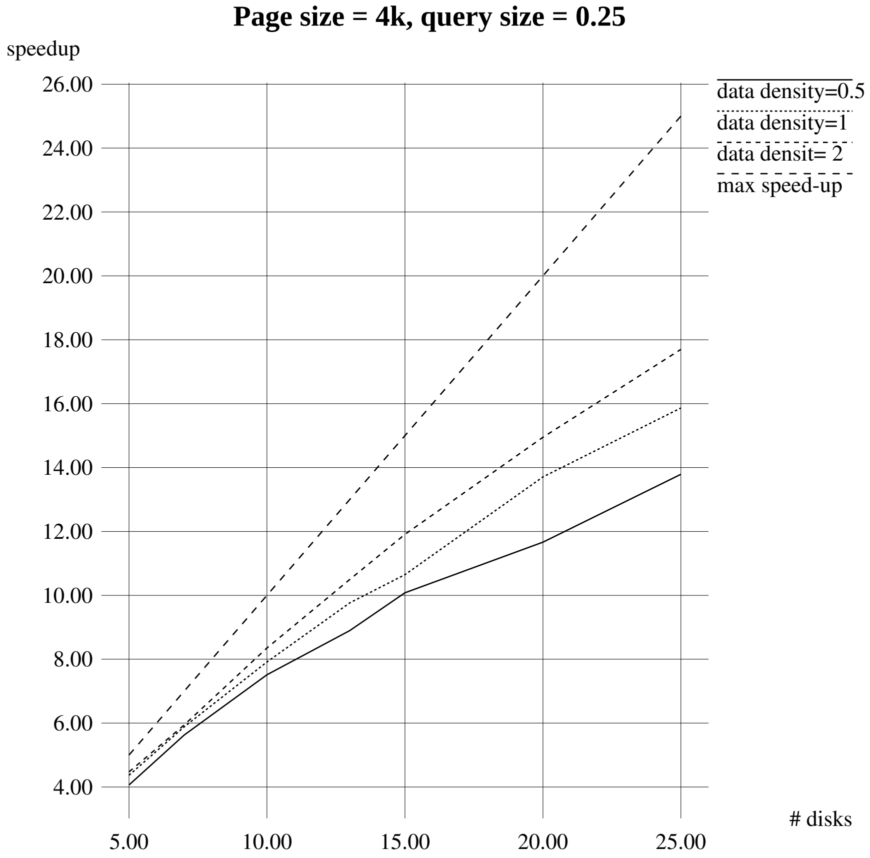


Figure 3.21: Speed-up of the Multiplexed R-tree vs. number of disks with query size = 0.25.

as possible.

Extensive simulation experiments show that the PI criterion consistently outperforms other criteria (round robin and minimum area), and that it performs approximately as well or better than the minimum intersection criterion.

A comparison with the super-node (= disk striping) approach shows that the proposed method offers a better response time for large queries and that it imposes a lighter load, leading to higher throughput.

With respect to speed-up, the proposed method can achieve near-linear speed-up for large queries. Thus, the multiplexed R-tree with the PI heuristic seems to be the best method for implementing a spatial object server.

Chapter 4

Hilbert R-trees

4.1 Introduction

In this chapter, we introduce two new spatial indexes based on space-filling curves. The first index is suitable for the static database in which updates are very rare or in which there are no updates at all. The nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the utilization is $\approx 100\%$; we call this structure a *Static Hilbert R-tree*. The second index supports insertions and deletions and is suitable for a dynamic environment; we call it a *Dynamic Hilbert R-tree*.

For the static environment, we design and study several heuristics for building the R-tree bottom-up. Most of these heuristics are based on space-filling curves, and specifically on the Hilbert curve. The difficult step is to sort the rectangles in some meaningful way; once this is done, we scan them, assigning each group of C rectangles to a leaf page of the R-tree (where C stands for the capacity of the disk page). We report experiments from two-dimensional data, although our method can handle higher dimensionalities. The experimental results show that the most effective of our heuristics is the one that sorts the data rectangles according to the Hilbert value of their centers ('2D-c' heuristic). This heuristic consistently outperforms all the known R-tree variants, namely, the quadratic-split R-tree and the R^*-tree, as well as the method proposed by Roussopoulos and Leifker [64], which is the only method of R-tree packing known up to now.

For the dynamic case, we propose an efficient indexing method for spatial data; this method is called the Dynamic Hilbert R-tree or simply "Hilbert R-trees." Our proposed indexing scheme combines the best characteristics of the R-tree and the B^*-tree. The Hilbert R-tree uses a simple insertion and splitting algorithms similar to those used in the B^*-tree. A new data rectangle is inserted into the tree according to its place on the Hilbert curve that passes through all the data in the space. Unlike other dynamic R-tree variants which have about 70% space utilization [7], the Hilbert R-tree can demonstrate much higher space utilization. When a node overflows, it refrains from splitting. If the left sibling is not full, the overflowing node pushes some of its entries to it. If the left sibling is full, the two nodes are split into three nodes. This idea is known as *local rotation* [47]. Moreover, the overflowing node can refrain from splitting unless s of the sibling nodes are full. Thus, by varying the parameter 's' (splitting policy), the Hilbert R-tree trade off insertion cost for search speed and higher utilization.

The rest of this chapter is organized as follows. Section 4.2 describes our proposed heuristics for building the static Hilbert R-tree. Section 4.3 describes the Dynamic Hilbert R-tree. In section 4.4, we introduce the analytical formula for computing the average response time for a given R-tree instance, given some information about the minimum bounding rectangles of its nodes. Section 4.5 presents our experimental results, which verify the validity of the analytical formula and compare the proposed methods (namely the Static and Dynamic

Hilbert R-trees) with other R-tree variants. Section 4.6 gives the conclusions and directions for future research.

4.2 Static Version – Ordering Rectangles

Method name	Description
2D-c	*sorts on the 2d-Hilbert value of the centers* (c_x, c_y)
4D-xy	*sorts on the 4-d Hilbert value of the two corners, i.e.,* $(low_x, low_y, high_x, high_y)$
4D-cd	*sorts on 4-d Hilbert value of the center and diameters, i.e.,* (c_x, c_y, d_x, d_y)
2Dz-c	*sorts on the z-value of the center* (c_x, c_y)
lowx packed R-tree [64]	sorts on the x coordinate of the lower left corner
linear-split R-tree [32]	Guttman's R-tree with linear split
quadratic-split R-tree [32]	Guttman's R-tree with quadratic split
R^*-tree [7]	R-tree variant, better packing, forced reinsert

Table 4.1: List of methods - the proposed ones are in *italics*.

We assume that the data are static, or that the frequency of modification is low. Our goal is to design a simple heuristic for constructing an R-tree with 100% space utilization, which, at the same time, will have as good response time as possible. For a static environment, Roussopoulos and Leifker [64] proposed a method for building a packed R-tree that achieves (almost) 100% space utilization. The idea is to sort the data on the x or y coordinate of one of the corners of the rectangles. The sorted list of rectangles is scanned; successive rectangles are assigned to the same R-tree leaf node until that node is full; a new leaf node is then created and the scanning of the sorted list continues. Thus, the nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the utilization is $\approx 100\%$. Higher levels of the tree are created in a similar way. Their experimental results on point data showed that their packed R-tree performs much better than does the linear split R-tree for point queries. In our experiments (Section 4.5), their packed R-tree outperformed the quadratic split R-tree and the R^*-tree as well for point queries on point data. However, the method does not perform that well for region queries and/or rectangular data.

We shall refer to the Roussopoulos and Leifker's method as the *lowx packed R-tree*. In our implementation of their method, we sort the rectangles according to the x value of the lower left corner ('*lowx*'). Sorting on any of the other three values gives similar results; thus our implementation does not impose an unfair disadvantage to the *lowx* packed R-tree. The fact that the *lowx* packed R-tree performs worse than do dynamic designs (e.g. R^*-tree) compels us to compare our new packing methods with both Static and Dynamic designs, including the *lowx* packed R-tree, the Guttman R-tree, and the R^*-tree. Figures 4.1 and 4.2 highlight the problem of the *lowx* packed R-tree. Figure 4.2 shows the leaf nodes of the R-tree that the *lowx* packing method will create for the points of Figure 4.1. The fact that the resulting father nodes cover little area explains why the *lowx* packed R-tree achieves excellent performance for point queries; the fact that the fathers have large perimeters (in conjunction with the ramification of Eq. 4.3 which is given in Section 4.4), explains the degradation of performance for region queries. Intuitively, the packing algorithm should ideally assign nearby points to the same leaf node. Ignorance of the y-coordinate by the *lowx* packed R-tree tends to violate this empirical rule.

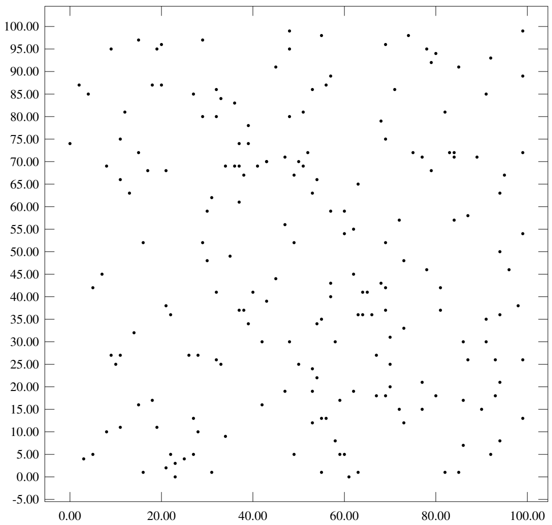


Figure 4.1: 200 points uniformly distributed.

In order to cluster the data better than can be done by the *lowx* packed R-trees, we propose the use of space-filling curves and specifically, the Hilbert curve.

A space-filling curve visits all the points in a k-dimensional grid exactly once and never crosses itself. The Z-order (or Morton key order, or bit-interleaving, or Peano curve), the Hilbert curve, and the Gray-code curve [14] are examples of space-filling curves. In [21], it was shown experimentally that the Hilbert curve achieves the best clustering of the above three methods.

We now provide a brief introduction to the Hilbert curve. The basic Hilbert curve on a 2x2 grid, denoted by H_1, is shown in Figure 4.3. To derive a curve of order i, each vertex of the basic curve is replaced by the curve of order $i-1$,

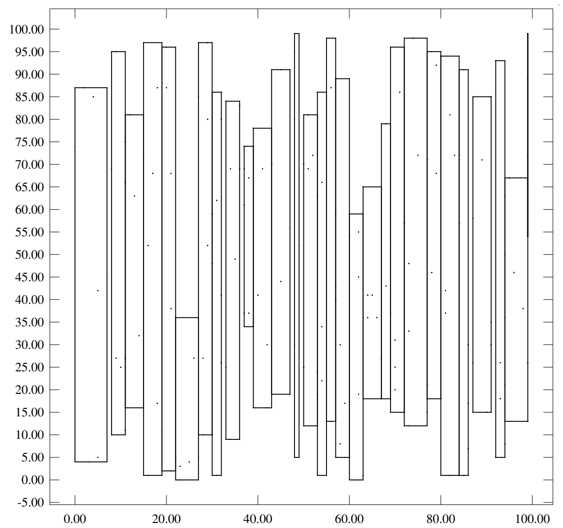


Figure 4.2: MBR of nodes generated by the 'lowx packed R-tree' algorithm.

which may be appropriately rotated and/or reflected. Figure 4.3 also shows the Hilbert curves of order two and three. When the order of the curve tends to infinity, like other space-filling curve, the resulting curve is a *fractal*, with a fractal dimension of two [53]. The Hilbert curve can be generalized for higher dimensionalities. Algorithms for drawing the two-dimensional curve of a given order can be found in [30], [40]. An algorithm for higher dimensionalities is given in [9].

The path of a space-filling curve imposes a linear ordering on the grid points; this path may be calculated by starting at one end of the curve and following the path to the other end. The actual coordinate values of each point can be calculated. However, for the Hilbert curve this is much harder than, for example, for the Z-order curve. Figure 4.3 shows one such ordering for a 4×4 grid (see curve H_2). For example, the point (0,0) on the H_2 curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2.

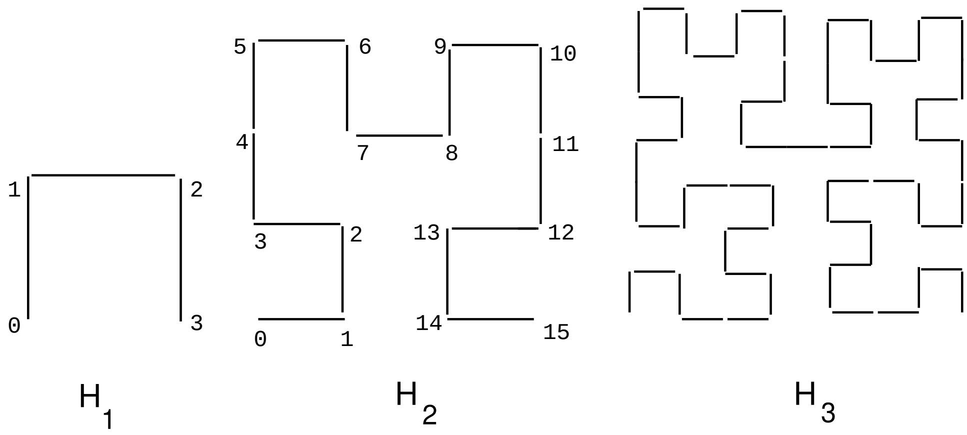


Figure 4.3: Hilbert curves of order 1, 2 and 3.

Having discussed this preliminary material, we are in a position now to describe the proposed methods. Exploiting the good clustering that the Hilbert curve can achieve, we impose a linear ordering on the data rectangles and then traverse the sorted list, assigning each set of C rectangles to a node in the R-tree. The final result is that the set of data rectangles on the same node will be close to each other in the linear ordering, and most likely in the native space; thus the resulting R-tree nodes will have smaller areas. Figure 4.3 illustrates the intuitive reasons why our Hilbert-based methods will result in good performance. The data is composed of points (the same points as given in Figures 4.1 and 4.2). We see that, by grouping the points according to their Hilbert values, the MBRs of the resulting R-tree leaf nodes tend to be small *square*-like rectangles. This indicates that the nodes will likely have small area and small perimeters. Small area values result in good performance for point queries; small area *and* small perimeter values lead to good performance for larger queries. Eq. 4.3 confirms the above claims.

We studied several methods for sorting the data rectangles. All of them use

Algorithm Hilbert-Pack:

(packs rectangles into an R-tree)

Step 1. Calculate the Hilbert value for each data rectangle

Step 2. Sort data rectangles on ascending Hilbert values

Step 3. */* Create leaf nodes (level l=0) */*

While (there are more rectangles)

generate a new R-tree node

assign the next C rectangles to this node

Step 4. */* Create nodes at higher level $(l+1)$ */*

While (there are > 1 nodes at level l)

sort nodes at level $l \geq 0$ on ascending

creation time

repeat Step 3

Figure 4.4: Pseudo-code of the packing algorithm.

the same algorithm (see Figure 4.4) to build the R-tree. The only point at which the proposed Hilbert-based methods distinguish themselves from each other is in the way they compute the Hilbert value of a rectangle. We examine the following alternatives:

4d Hilbert through corners ('4D-xy'): Each data rectangle is mapped to a point in four-dimensional space formed by the lower left corner and the upper right corner, namely (lowx, lowy, highx, highy). The Hilbert value of this four-dimensional point is the Hilbert value of the rectangle.

4-d Hilbert through center and diameter('4D-cd'): Each data rectangle is mapped to the four-dimensional point (c_x, c_y, d_x, d_y) where c_x, c_y are the coordinates of the center of the rectangle and d_x, d_y the 'diameters' or sides of the rectangle. As in 4D-xy, the Hilbert value of this four-dimensional point is the Hilbert value of the rectangle.

2-d Hilbert through Centers Only ('2D-c'): Each data rectangle is represented by its *center only*; the Hilbert value of the center is the Hilbert value of the rectangle.

For the sake of comparison, we also examined a method that uses the Peano curve, or 'z-ordering', despite the fact that the z-ordering achieves inferior clustering compared to the Hilbert curve [21]. The z-value of a point is computed by bit-interleaving the binary representation of its x and y coordinates. For example, in Figure 4.5, the point (0,0) has a z-value of 0, while the point (1,3) has a z-value of 7.

Z-order through Centers only ('2Dz-c'): The value of the rectangle is the z-value of its center.

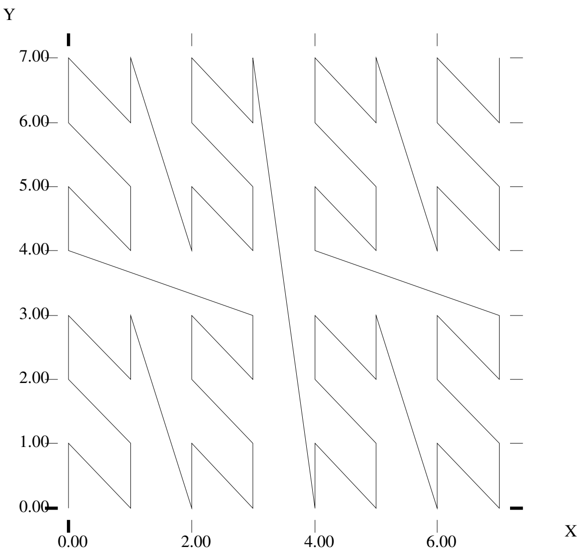


Figure 4.5: Peano (or z-order) curve of order 3.

Table 4.1 gives a list of the methods we compared, along with a brief description of each. The new methods are in italics; R-tree methods for static environments are above the double horizontal line; the other methods can be applied for dynamic environments as well.

4.3 Design of (Dynamic) Hilbert R-trees

In this section we introduce the 'Dynamic Hilbert R-tree' (or simply 'Hilbert R-tree') and discuss algorithms for searching, insertion, deletion, and overflow handling. The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. We propose the use of space-filling curves, and specifically the Hilbert curve, to impose a linear ordering on the data rectangles.

The Hilbert value of a rectangle needs to be defined. A good choice is the following:

Definition 2 : *The Hilbert value of a rectangle is defined as the Hilbert value of its center.*

4.3.1 Description

The goal is to create a tree structure that can

- behave like an R-tree on search.
- support local rotation on insertion, using the Hilbert value of the inserted data rectangle as the primary key.

These goals can be achieved as follows: for every node n of our tree, we store (a) its MBR, and (b) the *Largest Hilbert Value (LHV)* of the *data* rectangles that belong to the subtree with root n.

Specifically, the Hilbert R-tree has the following structure. A leaf node contains at most C_l entries each of the form

$$(R,\ obj\_id)$$

where C_l is the capacity of the leaf, R is the MBR of the real object $(x_{low}\ , x_{high}\ , y_{low}\ , y_{high}\ )$, and $obj - id$ is a pointer to the object description record. The main difference between the Hilbert R-tree and the R*-tree is that nonleaf nodes also contain information about the LHVs. Thus, a non-leaf node in the Hilbert R-tree contains at most C_n entries of the form

$$(R,\ ptr,\ LHV)$$

where C_n is the capacity of a non-leaf node, R is the MBR that encloses all the children of that node, ptr is a pointer to the child node, and LHV is the largest Hilbert value among the *data* rectangles enclosed by R. Notice that since the non-leaf node picks one of the Hilbert values of the children to be the value of its own LHV, we *never* calculate or use the Hilbert values of the MBR of non-leaf nodes. Figure 4.6 illustrates some rectangles organized in a Hilbert R-tree. The Hilbert values of the centers are the numbers near the 'x' symbols (shown only for the parent node 'II'). The LHV's are in [brackets]. Figure 4.7 shows how the tree of Figure 4.6 is stored on the disk; the contents of the parent node 'II' are shown in more detail. Every data rectangle in node 'I' has a Hilbert value $\leq$33; everything in node 'II' has a Hilbert value greater than 33 and $\leq$107, etc.

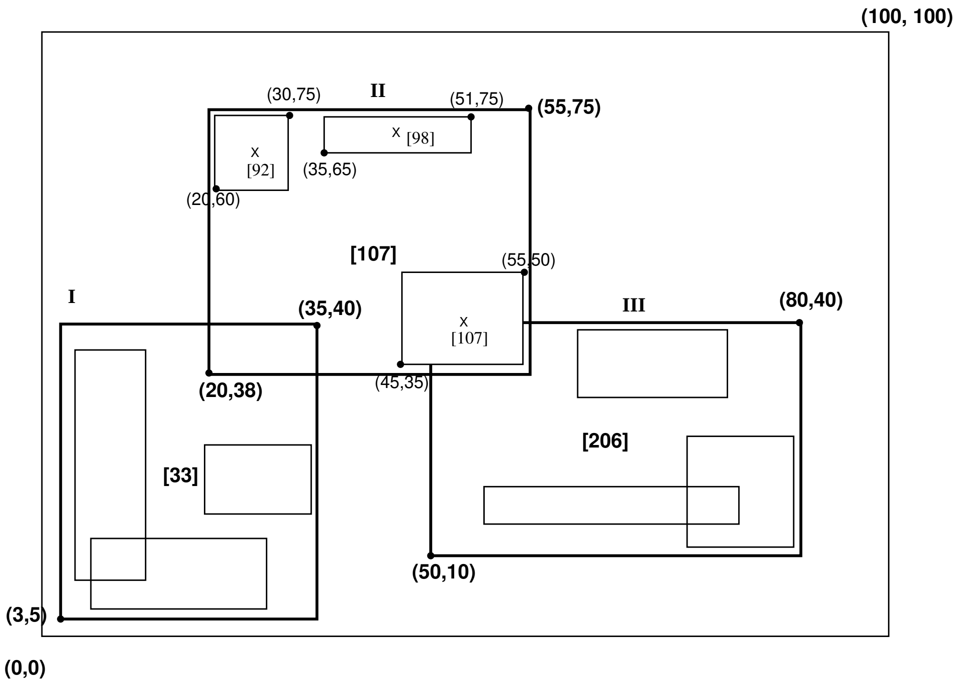


Figure 4.6: Data rectangles organized in a Hilbert R-tree (Hilbert values and LHV's are in brackets).

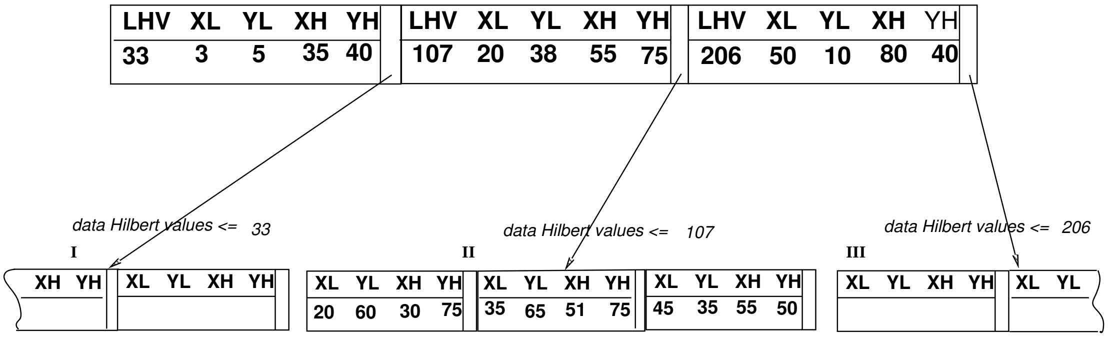


Figure 4.7: The file structure for the Hilbert R-tree.

Before we continue, we list some definitions. A plain R-tree splits a node on overflow, creating two nodes from the original one. We call this policy a *1-to-2* splitting policy. We propose to defer the split, waiting until two nodes split into three. Note that this is similar to the $B^* - tree$ split policy. We refer to this method as the 2-to-3 splitting policy. In general, we can have an s-to-(s+1) splitting policy; we refer to s as the *order of the splitting policy.* To implement the order-s splitting policy, the overflowing node tries to push some of its entries to one of its $s-1$ siblings; if all of them are full, then we have an s-to-(s+1) split. We refer to the $s-1$ siblings as the *cooperating siblings* of a given node.

Next, we describe in detail the algorithms for searching, insertion, and overflow handling.

4.3.2 Searching

The searching algorithm is similar to the one used in other R-tree variants. Starting from the root, it descends the tree and examines all nodes that intersect the query rectangle. At the leaf level, it reports all entries that intersect the query window w as qualified data items.

Algorithm Search(node Root, rect w):

S1. *Search nonleaf nodes:*

Invoke Search for every entry whose MBR intersects the query window w.

S2. *Search leaf nodes:*

Report all entries that intersect the query window w as candidates.

4.3.3 Insertion

To insert a new rectangle r in the Hilbert R-tree, the Hilbert value h of the center of the new rectangle is used as a key. At each level we choose the node with the minimum LHV of all its siblings. When a leaf node is reached, the rectangle r is inserted in its correct order according to h. After a new rectangle is inserted in a leaf node N, **AdjustTree** is called to fix the MBR and LHV values in the upper-level nodes.

Algorithm Insert(node Root, rect r):

/* Inserts a new rectangle r in the Hilbert R-tree. h is the Hilbert value of the rectangle. */

I1. *Find the appropriate leaf node:*

Invoke **ChooseLeaf(r, h)** to select a leaf node L in which to place r.

I2. *Insert r in a leaf node L:*

If L has an empty slot, insert r in L in the appropriate place according to the Hilbert order and return.

If L is full, invoke **HandleOverflow(L,r)**, which will return new leaf if split was inevitable.

I3. *Propagate changes upward:*

Form a set $\mathcal{S}$ that contains L, its cooperating siblings and the new leaf (if any).

Invoke **AdjustTree($\mathcal{S}$).**

I4. *Grow tree taller:*

If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Algorithm ChooseLeaf(rect r, int h):

/* Returns the leaf node in which to place a new rectangle r. */

C1. *Initialize:*

Set N to be the root node.

C2. *Leaf check:*

If N is a leaf, return N.

C3. *Choose subtree:*

If N is a non-leaf node, choose the entry (R, ptr, LHV) with the minimum LHV value greater than h.

C4. *Descend until a leaf is reached:*

Set N to the node pointed by ptr and repeat from C2.

Algorithm AdjustTree(set $\mathcal{S}$):

/* $\mathcal{S}$ is a set of nodes that contains the node being updated, its cooperating siblings (if overflow has occurred) and the newly created node NN

(if split has occurred).

The routine ascends from the leaf level towards the root, adjusting MBR and LHV of nodes that cover the nodes in $\mathcal{S}$.

It propagates splits (if any). */

A1. If root level is reached, stop.

A2. *Propagate node split upward*

Let N_p be the parent node of N.

If N has been split, let NN be the new node.

Insert NN in N_p in the correct order according to its Hilbert value if there is room. Otherwise, invoke **HandleOverflow**(N_p, **NN**).

If N_p is split, let PP be the new node.

A3. *Adjust the MBR's and LHV's in the parent level:*

let $\mathcal{P}$ be the set of parent nodes for the nodes in $\mathcal{S}$.

Adjust the corresponding MBR's and LHV's of the nodes in $\mathcal{P}$ appropriately.

A4. *Move up to next level:*

Let $\mathcal{S}$ become the set of parent nodes $\mathcal{P}$, with

NN = PP, if N_p was split.

repeat from A1.

4.3.4 Deletion

In the Hilbert R-tree we do NOT need to re-insert orphaned nodes whenever a father node underflows. Instead, we borrow keys from the siblings or we merge an underflowing node with its siblings. We are able to do so because the nodes have a clear ordering (according to Largest Hilbert Value, LHV); in contrast, in R-trees there is no such concept concerning sibling nodes. Notice that for deletions we need s cooperating siblings, while for insertion we need $s-1$ siblings.

Algorithm Delete(r):

D1. *Find the host leaf:*

Perform an exact match search to find the leaf node L
  that contains r.

D2. *Delete* r :

Remove r from node L.

D3. If L underflows

borrow some entries from s cooperating siblings.

if all the siblings are ready to underflow,
  merge $s+1$ to s nodes,
  adjust the resulting nodes.

D4. *Adjust MBR and LHV in parent levels.*

form a set $\mathcal{S}$ that contains L and its cooperating
  siblings (if underflow has occurred).

invoke **AdjustTree**($\mathcal{S}$).

4.3.5 Overflow Handling

The overflow handling algorithm in the Hilbert R-tree treats the overflowing nodes either by moving some of the entries to one of the $s-1$ cooperating siblings or by splitting s nodes into $s+1$ nodes.

Algorithm HandleOverflow(node N, rect r):

/* return the new node if a split occurred. */

H1. Let $\mathcal{E}$ be a set that contains all the entries from N
    and its $s-1$ cooperating siblings.

H2. Add r to $\mathcal{E}$.

H3. If at least one of the $s-1$ cooperating siblings is not full,
    distribute $\mathcal{E}$ evenly among the s nodes according to
    Hilbert values.

H4. If all the s cooperating siblings are full,
    create a new node NN and
    distribute $\mathcal{E}$ evenly among the $s+1$ nodes according
    to Hilbert values.
    return NN.

4.4 Analytical Formula for the Response Time

In this section we introduce an analytical formula for evaluating the average response time for a query of size $q_x \times q_y$ as a function of the geometric characteristics of a given instance of an R-tree. This means that once we have built the R-tree we can estimate the average response time of the query $q_x \times q_y$ *with-*

Symbols	Definitions
p	page size, in bytes
C	page capacity
	(max. number of rectangles per page)
$P(q_x, q_y)$	avg. pages retrieved by a $q_x \times q_y$ query
N_d	number of data rectangles
$\mathcal{N}$	number of tree nodes
τ	density of data
n_i	node i in the R-tree
$n_{i,x}$	length of node i in x direction
$n_{i,y}$	length of node i in y direction
L_x	sum of x-sides of all nodes in the tree
L_y	sum of y-sides of all nodes in the tree
$TotalArea$	sum of areas of all nodes in the tree
q_x	length of the query in x direction
q_y	length of the query in y direction

Table 4.2: Summary of symbols and definitions.

out generating random queries and then computing the average and variance of their response times. In this discussion we assume that queries are rectangles uniformly distributed over the unit square address space. Without loss of generality we consider a two-dimensional space. The same idea can be generalized to higher dimensions.

The response time of a range query is primarily affected by the time required to retrieve the nodes touched by the query plus the time required by the CPU to process the nodes. Since the CPU is much faster than the disk, we assume that the CPU time is negligible (=0) compared to the time required by a disk to retrieve a page. Thus, the measure for the response time is approximated by the number of nodes (pages) that will be retrieved by the range query.

The next lemma forms the basis for the analysis:

Lemma 1 *If the node n_i of the R-tree has an MBR of $n_{i,x} \times n_{i,y}$, then the probability $DA(n_{i,x}, n_{i,y})$ that this node will contribute one disk access to a point query is*

$$\begin{aligned} DA(n_{i,x}, n_{i,y}) &= \text{Prob(point query retrieves node } n_i) \\ &= n_{i,x} * n_{i,y} \end{aligned} \tag{4.1}$$

$DA()$ is the expected number of disk accesses that the specific node will contribute in an arbitrary point query. Notice that the level *of the node in the R-tree is* immaterial.

Proof: Since we assume that the (point) queries are uniformly distributed in the address space and the address space is the unit square. The probability that a random point fall within the rectangle $(n_{i,x}, n_{i,y})$ is the area of the rectangle $n_{i,x} \times n_{i,y}$.

The next two lemmas calculate the expected number of disk accesses for point and rectangular queries respectively.

Lemma 2 (Point query) *For a point query, the expected number of disk accesses $P(0,0)$ is given by*

$$P(0,0) \; = \sum_{i=1}^{\mathcal{N}} n_{i,x} * n_{i,y} \tag{4.2}$$

Proof: Every node n_i in the R-tree is represented in the native space by its minimum bounding rectangle (MBR) of size say, $n_{i,x}$, $n_{i,y}$ in the x, y direction respectively. Given Lemma 1, each node of the R-tree contributes $DA()$ disk accesses; to calculate the average number of disk accesses resulting from *all* the nodes of the R-tree, we have to sum Eq. 4.1 over all the nodes.

Similar analysis was done independently in **??**.

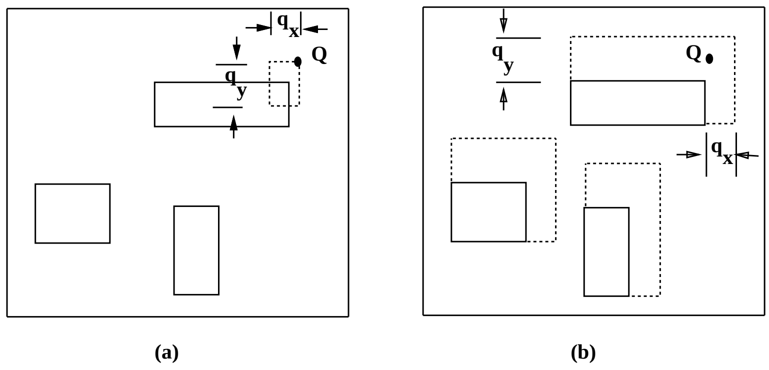


Figure 4.8: (a) Original nodes along with rectangular query $q_x \times q_y$; (b) Extended nodes with point query Q.

Lemma 3 (Rectangular query) *For a rectangular query $q_x \times q_y$, the expected number of disk accesses $P(q_x, q_y)$ is given by*

$$P(q_x, q_y) \; = \sum_{i=1}^{\mathcal{N}} n_{i,x} * n_{i,y} + q_y \times \sum_{i=1}^{\mathcal{N}} n_{i,x}$$

$$+q_x * \sum_{i=1}^{\mathcal{N}} n_{i,y} + \mathcal{N} * q_x * q_y \tag{4.3}$$

Proof: A rectangular query of size $q_x \times q_y$ is equivalent to a point query, if we 'inflate' the nodes of the R-tree by q_x and q_y in the x- and y-directions respectively (equivalently, the node can be inflated by $q_x/2$ along the x direction from the two ends and by $q_y/2$ along the y direction from the two ends). Thus, the node n_i with size $n_{i,x} \times n_{i,y}$ behaves like a node of size $(n_{i,x} + q_x) \times (n_{i,y} + q_y)$. Figure 4.8 illustrates the idea: Figure 4.8(a) shows a range query $q_x \times q_y$ with the upper-left corner at Q; this query is equivalent to a point query anchored at Q as long as the data rectangles are 'inflated' as shown by the dotted lines in Figure 4.8(b). Applying (Eq. 4.2) on Figure 4.8(b) we obtain:

$$P(q_x, q_y) \ = \sum_{i=1}^{\mathcal{N}} (n_{i,x} + q_x) * (n_{i,y} + q_y) \tag{4.4}$$

which after trivial mathematical manipulations gives (Eq. 4.3).

Notice that Lemma 3 gives

$$\begin{aligned} P(q_x, q_y) \ &= \\ &TotalArea + q_x * L_y + q_y * L_x + \mathcal{N} * q_x * q_y \end{aligned} \tag{4.5}$$

where $TotalArea = P(0,0)$ is the sum of all the areas of the nodes of the tree, and L_x, L_y are respectively the sums of x and y extents of all nodes in the R-tree.

There are several comments and observations with respect to the above formulas:

- The formula is *independent* of the details of the R-tree creation/insertion/split algorithms; it holds for packed R-trees, for R^*-trees, etc.

- Notice that Eq. 4.3 for range queries reduces to Eq. 4.2 for point queries if $q_x = q_y = 0$, as expected.

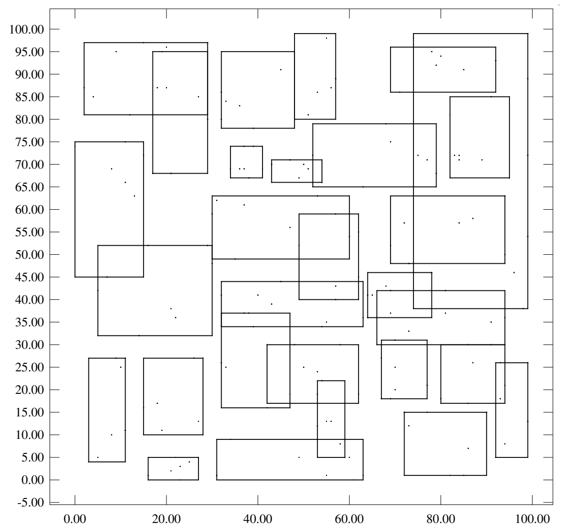


Figure 4.9: MBR of nodes generated by 2D-c (2-d Hilbert through centers) for 200 random points.

- The last equation illustrates the importance of minimizing the perimeter of the R-tree nodes, in addition to the area. The larger the queries, the more important the perimeter becomes. This explains why the *lowx* packed R-tree performs well for point queries ($q_x = q_y = 0$), but not so well for larger queries. The nodes produced by the *lowx* packed R-tree (Figure 4.2) have small areas but large perimeters. Figure 4.9 shows the leaf nodes produced by the '2D-c' Hilbert packing method for the set of points given in Figure 4.1. Notice that the resulting nodes have smaller perimeters.

- Eq. 4.3 has theoretical as well as practical value: From a practical point of view, it can assist with the cost estimation and query optimization for spatial queries [3]: Maintaining only a few numbers about the R-tree (total area, total perimeter), a query optimizer can make a good estimate of the cost of a range query. Moreover, researchers working on R-trees can use

Eq 4.3 to avoid issuing queries in their simulation studies. This eliminates one randomness factor (the query), leaving the generation and insertion order of the data as random variables.

4.5 Experimental Results

To assess the merit of our proposed Hilbert R-trees, we implemented both the static and the dynamic Hilbert R-trees and ran experiments on a two dimensional space. The methods were implemented in C under UNIX. We compared our methods against the quadratic-split R-tree, the R^*-tree, and the *lowx* R-tree. Since the CPU time required to process the node is negligible, we based our comparison on the number of nodes (=pages) retrieved by range queries.

Without loss of generality, the address space was normalized to the unit square. There are several factors that affect the search time; we studied the following ones:

Data items: points and/or rectangles and/or line segments (represented by their MBRs)

File size: ranged from 10,000 to 100,000 records.

Query area $Q_{area} = q_x \times q_y$: ranged from 0 to 0.3 of the area of the address space.

Recall that the 'data density' τ (or 'cover quotient') of the data rectangles is the sum of the areas of the data rectangles in the unit square, or equivalently, the average number of rectangles that cover a randomly selected point. Mathematically: $\tau = N \times a$. For the selected values of N and a, the data density ranges from 0.25 to 2.0.

To compare the performance of our proposed structures we used five data files that contained different types of data: points, rectangles, lines, or mixed. Specifically, we used:

A) Real Data: we used real data from the TIGER system of the U.S. Bureau of Census. These were the same files that we used before. We repeat their description for convenience. An important observation is that the data in the TIGER datasets follow a highly *skewed* distribution.

'MGCounty' : This file consists of 39,717 line segments representing the roads of Montgomery County in Maryland. Using the minimum bounding rectangles of the segments, we obtained 39,717 rectangles, with data density $\tau = 0.35$. We refer to this dataset as the '*MG-County*' dataset.

'LBeach' : This file consists of 53,145 line segments representing the roads of Long Beach, California. The data density of the MBRs that cover these line segments is $\tau = 0.15$. We refer to this dataset as the '*LBeach*' dataset.

B) Synthetic Data: The reason for using synthetic data is that we can control the parameters (data density, number of rectangles, ratio of points to rectangles, etc.).

'Points' : This file contains 75,000 uniformly distributed points.

'Rects' : This file contains 100,000 rectangles, no points. The centers of the rectangles are uniformly distributed in the unit square. The data density is $\tau = 1.0$

'Mix' : This file contains a mix of points and rectangles; specifically 50,000 points and 10,000 rectangles; the data density is $\tau = 0.029$.

The query rectangles were squares with side q_s; their centers were uniformly distributed in the unit square. For each experiment, 200 randomly generated queries were asked and the results were averaged. The standard deviation was very small and is not even plotted in our graphs. In the following subsections we present two groups of experiments to evaluate our methods in a static and in a dynamic environment respectively.

4.5.1 Static Hilbert R-trees

Here we evaluate the performance of our Hilbert R-tree for a static environment. In the following subsections we present experiments (a) verifying (Eq. 4.3) for the response time; (b) comparing the response time of the best of our methods (2D-c) with the response time of older R-tree variants (dynamic or static); and (c) comparing all proposed packing schemes against each other in order to pinpoint the best.

Verifying the formula for the response time

In the previous section we introduced a probabilistic model for the R-tree under rectangular range queries. Equation 4.3 gives an estimate for the number of pages retrieved by a query of size $q_x \times q_y$. In this section we introduce experimental results to show how far our estimate is from the experimental values.

Table 4.3 shows the number of pages retrieved as a function of the query size (area). We carried out many experiments to compare the formula with the simulation results. For each query size, 50 random queries are generated.

query area Q_{area}	Exper. nodes/query avg.(pages/query)	std. dev.	Theor. (Eq. 4.3) pages/query
0.00000	3.88	0.86	3.75
0.00001	4.06	1.00	4.12
0.00027	5.84	1.24	5.95
0.00333	9.00	1.35	9.01
0.01333	16.94	1.91	17.67
0.08333	63.18	4.14	64.07
0.11111	208.20	7.41	209.45

Table 4.3: Verifying (Eq. 4.3); theoretical vs. experimental response time (pages/query).

The average and standard deviation are calculated, and compared with the one derived analytically. In Table 4.3, we use the area Q_{area} as the measure of the size of queries. Column 2 shows the response time in terms of the number of disk accesses measured experimentally for the different query sizes in Column 1. The standard deviation in the experimental response time (due to the randomness in the query) is shown in Column 3. Column 4 shows the response time (in terms of the number of page accesses) as estimated by (Eq. 4.3). The data file contains 75k points. Notice that the formula matches the experimental results extremely well. The difference between the estimated number of pages retrieved by a query and the experimental value is less than one standard deviation. For this experiment, the R-tree was built using the Hilbert 2D-c packing heuristic. We also experimented with the following R-tree structures: R^*-tree, $lowx$ packed R-tree, quadratic split R-tree, 4D-cd packed R-tree, and 4D-xy packed R-tree. We obtained similar results in all cases which we do not show, because they provide no additional information. These results are typical for several combinations of parameter values ($p = (1Kb - 4Kb), \tau = (0.25 - 1)$) and several datasets (e.g. datasets consisting of both points and rectangles, etc.). In all the results we present throughout the rest of Section 4.5.1, we used (Eq. 4.3) to calculate the number of pages retrieved by a query.

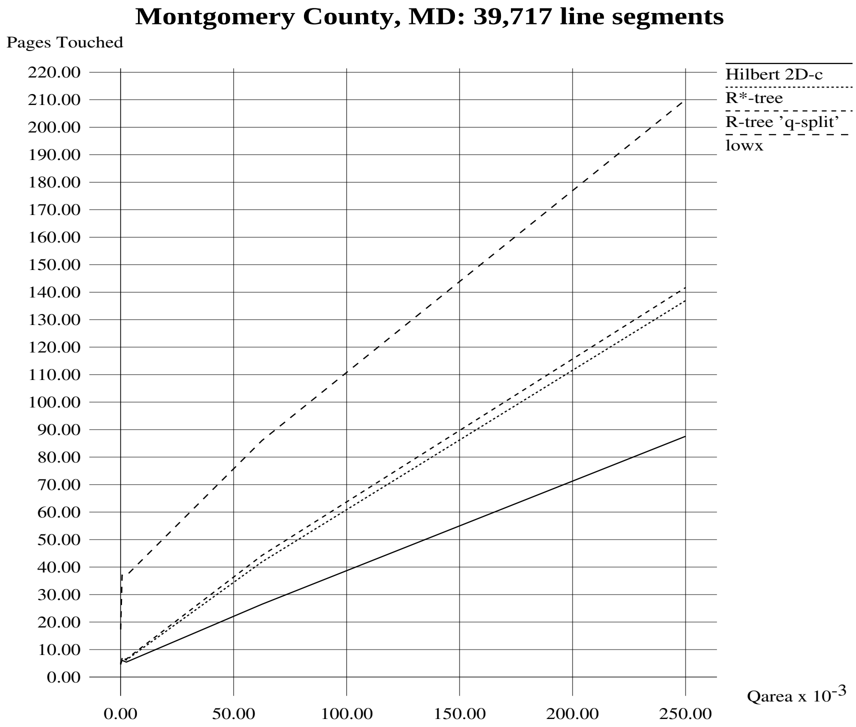


Figure 4.10: Hilbert 2D-c packed R-tree vs. other R-tree variants; 'MGCounty' dataset – real data.

Comparison of 2D-c Hilbert packed R-tree vs. older R-tree variants

In this section we introduce experimental results to compare the performance of the Hilbert 2D-c packed R-tree versus other R-tree variants. In our experiments we focused on the rectangular range queries. The page size p = 1Kb. Fig-

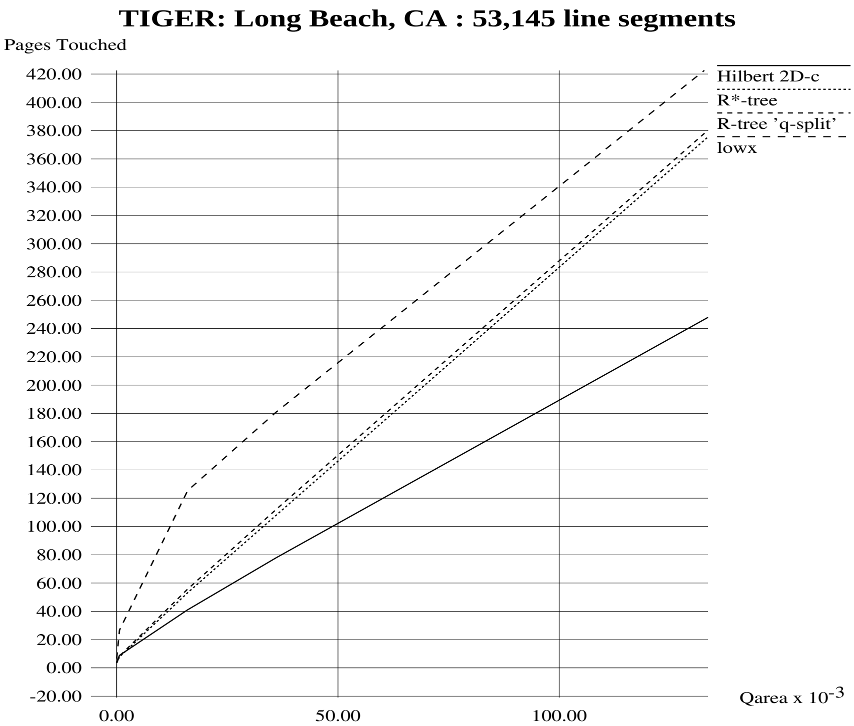


Figure 4.11: Hilbert 2D-c packed R-tree vs. other R-tree variants; 'LBeach dataset' – real data.

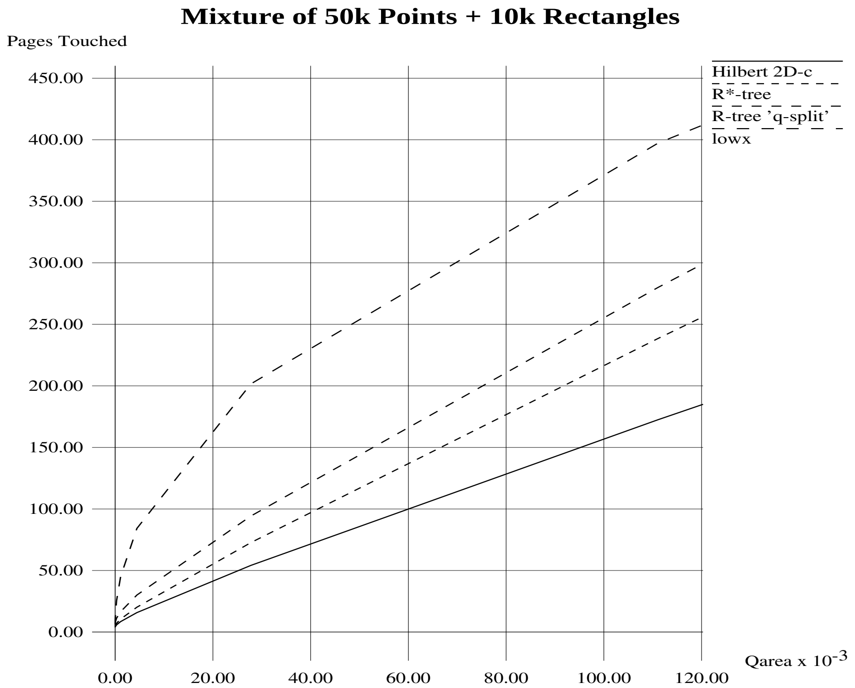


Figure 4.12: Hilbert 2D-c packed R-tree vs. other R-tree variants; 'Mix' dataset – synthetic data.

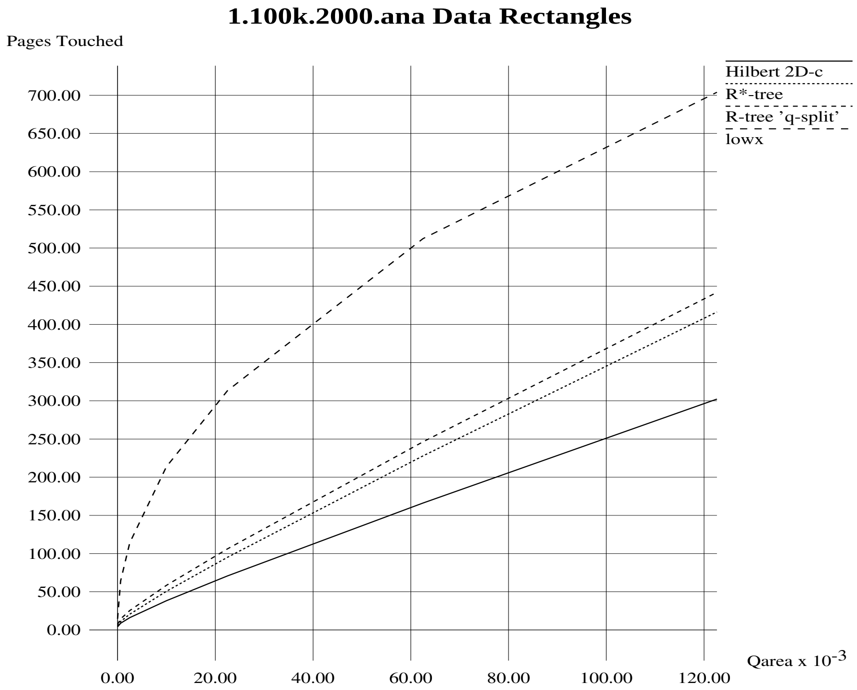


Figure 4.13: Hilbert 2D-c packed R-tree vs. other R-tree variants; 'Rects' dataset – synthetic data.

ures 4.10- 4.13 plot the average number of pages retrieved as a function of Qarea (= query area = $q_x \times q_y$). We show that the 2D-c packing method gives better response times than do older R-tree variants. The fact that the *lowx* packed R-tree performs worse than do dynamic designs (such as R^*-tree) compelled us to compare our new packing methods with both static and dynamic designs, namely the *lowx* packed R-tree, the Guttman R-tree with quadratic split, and the R^*-tree. In our plots we omit the results of the linear-split R-tree, because the quadratic-split R-tree consistently outperformed it. The exponential-split R-tree was very slow in building the tree, and it was not used. For the R^*-tree, the percentage of nodes to be deleted in case of node overflow in the forced reinsert algorithm is set to the recommended value of 30% [7]. To avoid cluttering the plots, we only plot the best of our proposed algorithms, namely the one using the '2D-c' heuristic. The detailed results for the other Hilbert-based packing algorithms are presented in the next subsection.

Figures 4.10- 4.13 plot the number of pages retrieved by a range query (from Eq. 4.3) as a function of the area of the query. In each graph we show four curves for the following R-tree variants: the Hilbert 2D-c packed R-tree ("Hilbert 2D-c"), the R^*-tree, the quadratic split R-tree of Guttman ("R-tree 'q-split"'), and the lowx packed R-tree ("lowx"). Figures 4.10 and 4.11 show the results for the TIGER data sets, which represent the roads of Montgomery County of Maryland and the roads of Long Beach of California respectively. Figure 4.12 shows the results for 'Mix' data set. The fourth set (Figure 4.13) is the 'Rects' data set.

A common observation is that, for point queries, all methods perform almost the same, with small differences. However, for slightly larger queries, the proposed 2D-c Hilbert packed R-tree is the clear winner. The performance gap increases with the area of the queries.

The second important observation is that the performance gap seems to increase with the *skewness* of the data distribution: for the TIGER data sets, the proposed method achieves up to 36% improvement over the next best method (R^*-tree), and up to 58% improvement over the *lowx* packed R-tree. One might expect that the Hilbert R-tree would perform better because of its high space utilization (almost 100%). But since the performance of the static *lowx* packed R-tree (100% space utilization) is worse than the performance of the dynamic designs (e.g., quadratic split R-tree and the R^*-tree), we ascribe the good performance of our proposed methods not only to the higher space utilization but also to the good clustering property of the Hilbert curve.

Moreover, the difference between the R^*-tree and the quadratic split R-tree is even smaller when real data are used. The R^*-tree performs better than the quadratic split R-tree for the following reasons. First, the R^*-tree algorithms take into account the area and perimeter of the resulting nodes, while, the quadratic split R-tree tries to minimize the area only. Note that these empirical results conform with our analysis (Equation 4.3), which shows that the response time of the rectangular queries depends on the area and the perimeter of the R-tree node. Second, the R^*-tree employs the concept of "forced reinsert" when a node overflows; this factor helps in reorganizing the tree occasionally.

Comparison of Hilbert-based packing schemes

Here we compare all the packing heuristics that we have introduced in this paper, namely 2D-c , 4D-xy, 4D-cd and the only heuristic that uses the z-ordering, 2Dz-c. Table 4.1 contains a list of these methods, along with a brief description.

query area Q_{area}	Hilbert 2D-c	Hilbert 4D-cd	Hilbert 4D-xy
0.000000	3.74	5.10	7.04
0.000278	5.60	7.28	9.26
0.001111	8.22	10.24	12.04
0.004444	15.20	17.84	20.32
0.111111	169.76	177.06	180.54

Table 4.4: Comparison (disk accesses/query) of different schemes which use the Hilbert order.

Table 4.4 gives the response time versus the query area for all of these heuristics that use the *Hilbert* order. The (synthetic) data file consists of 50K points and 10K rectangles. The page size p = 1Kb. The differences between the alternative methods are small. However, from Table 4.4 we see that (2D-c) does better, especially for large queries. The next best method is the (4D-cd), which uses a 4-d Hilbert curve on the parameter space (center-x, center-y, diameter-x, diameter-y). The last contender is the 4D-xy.

For the same setting, Table 4.5 compares the 2D-c heuristic, which sorts the data according to the *2d Hilbert-value* of the centers of the data rectangles and the 2Dz-c heuristic, which sorts according to the two-dimensional *z-value* of the center. Table 4.5 shows that the 2D-c which uses the Hilbert order, always performs better than the 2Dz-c which uses the z-order. In our experiments, we only compared the clustering property of the Hilbert and the z-order curves. For the comparison to be fair, other properties need to be compared; such a comparison would include the cost of calculating the code, the cost of reversing the code and

query area Q_{area}	Hilbert 2D-c	Z-order 2Dz-c
0.000000	3.74	5.98
0.000278	5.60	8.64
0.001111	8.22	11.48
0.004444	15.20	20.28
0.111111	169.76	183.56

Table 4.5: Schema that uses Hilbert order vs. one that uses z-order (disk accesses/query).

other properties which are important for image processing algorithms (such as admissibility [12]), which are out of the scope of this thesis. In our application, the cost of calculating the Hilbert value is small. Also, we only need to compute the Hilbert value ONCE on insertion; and NEVER on search.

The relative ranking of the methods was the same for every dataset we tried; we omit the results because they provide no new information.

4.5.2 Dynamic Hilbert R-trees

Here we evaluate the performance of the proposed Hilbert R-tree in a dynamic environment. We compare the Hilbert R-tree to the original R-tree (quadratic split) and the R^*-tree. Next we present experiments that (a) compare our method against other R-tree variants, (b) show the effects of the different split policies on the performance of the proposed method, and (c) evaluate the insertion cost.

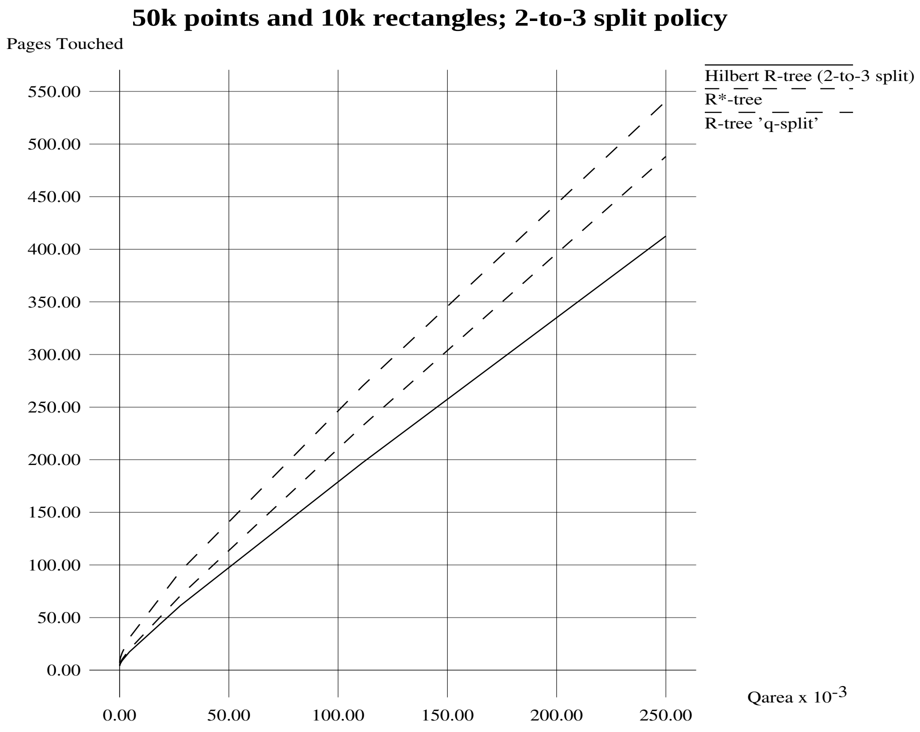


Figure 4.14: Dynamic Hilbert R-tree (2-to-3 split) vs. other R-tree variants; 'Mix' dataset.

Comparison of the Hilbert R-tree vs. other R-tree variants

In this section we compare our Hilbert R-tree '2-to-3' split with the R^*-tree and the quadratic split R-tree. We present experiments with all five datasets, namely: 'Mix', 'Rects', 'Points', 'MGCounty', and 'LBeach' (see Figures 4.14 - 4.18, respectively). In all these experiments, we used the '2-to-3' split policy for the Hilbert R-tree. In each experiment we plot the average number of page accesses per query as a function of the area of the query rectangle. For each query size we ask 50 random queries and calculate the average number of nodes touched per query.

In all the experiments, the Hilbert R-tree gives the best performance and is

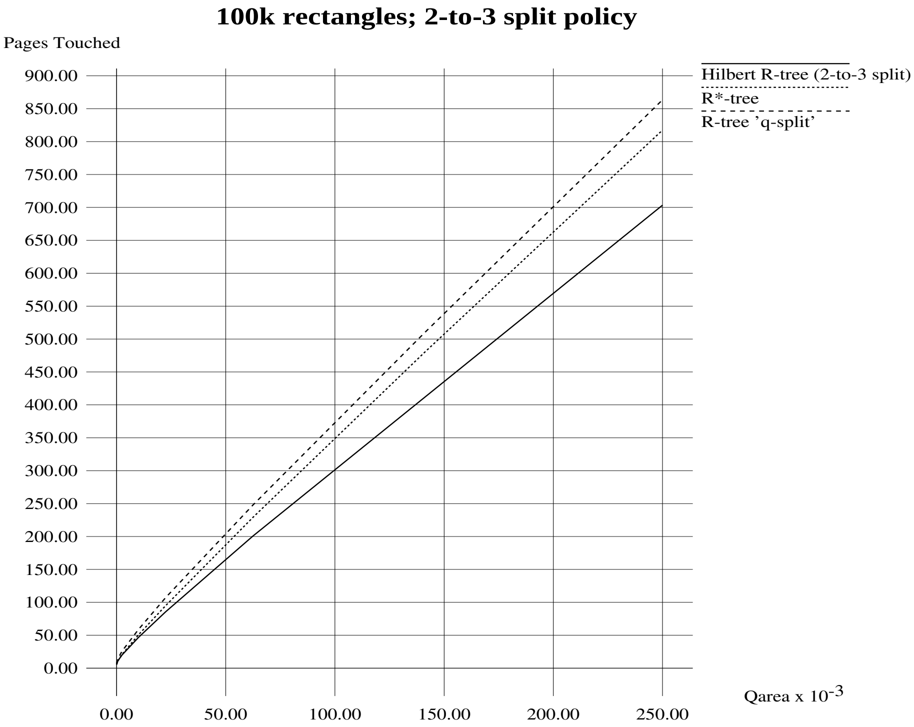


Figure 4.15: Dynamic Hilbert R-tree (2-to-3 split) vs. other R-tree variants; 'Rects' dataset.

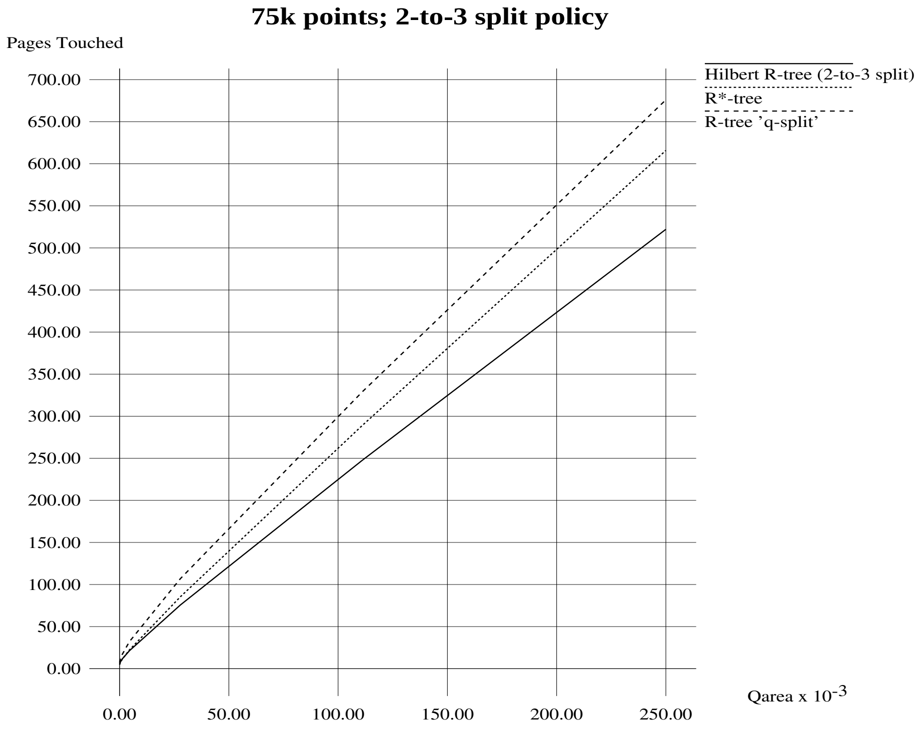


Figure 4.16: Dynamic Hilbert R-tree (2-to-3 split) vs. other R-tree variants; 'Points' dataset.

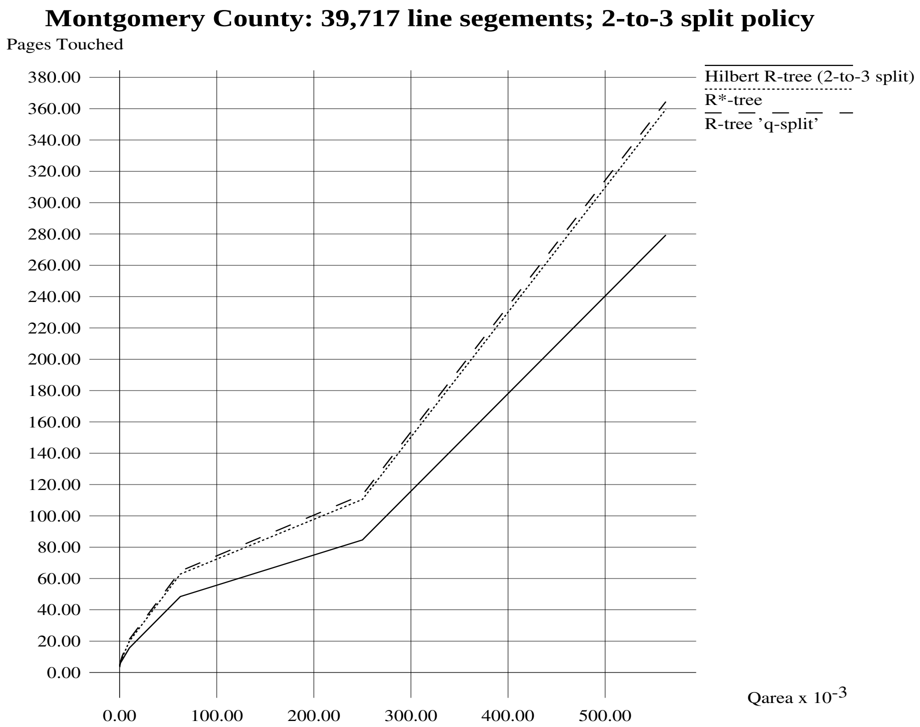


Figure 4.17: Dynamic Hilbert R-tree (2-to-3 split) vs. other R-tree variants; 'MGCounty' dataset.

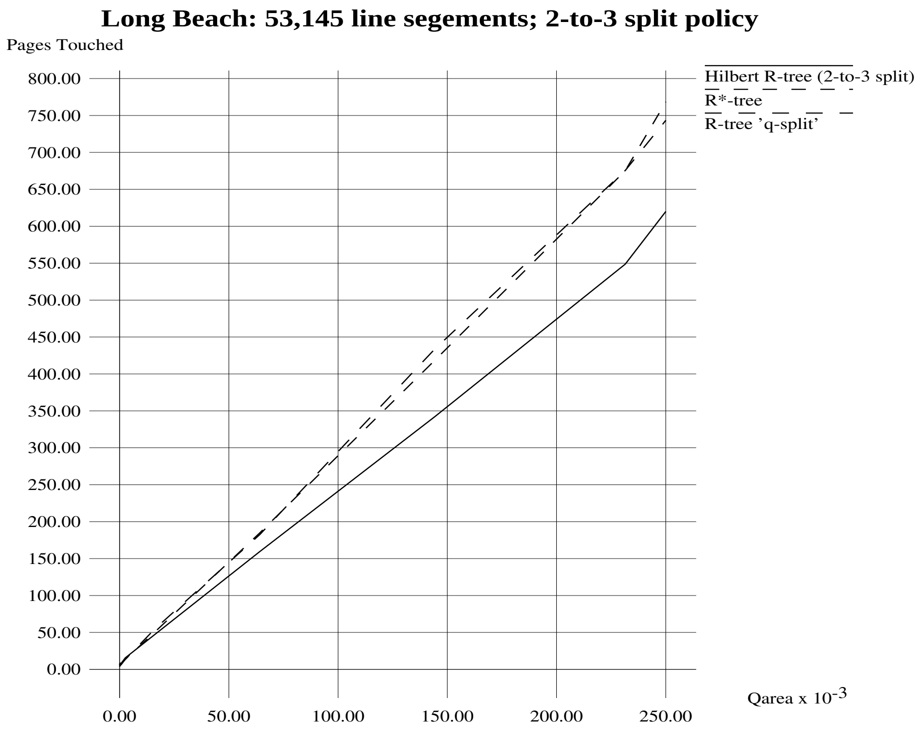


Figure 4.18: Dynamic Hilbert R-tree (2-to-3 split) vs. other R-tree variants; 'LBeach' dataset.

followed by the R^*-tree. The good performance of the Hilbert R-tree '2-to-3' split is due to the good clustering property of the Hilbert curve and the higher space utilization ($\approx 83\%$) achieved by the '2-to-3' split policy. The good space utilization is not a sufficient condition for the good performance simply because the *lowx* packed R-tree, which has 100% space utilization, performs worse than the R^*-tree and the quadratic split R-tree. Note also that the performance of the three R-tree variants is comparable for point queries (Qarea = 0). In the light of Equation 4.3, we can make the following observation: The total areas of the nodes of the three R-tree structures, namely the Hilbert R-tree, the R^*-tree, and the quadratic split R-tree, are approximately equal; this is why all three R-tree structures give similar response times when Qarea=0. The three structures differ, however, in total perimeter. The Hilbert R-tree gives the smallest total perimeter. Note that the perimeter term appears and becomes dominant for non-point queries (Qarea > 0). Also, the perimeter term is the one that gives the edge to the R^*-tree over the quadratic split R-tree. As we mentioned earlier, the split algorithm for the R^*-tree minimizes the areas and the perimeters of the resulting nodes, while the quadratic split R-tree minimizes the area only.

In all the given experiments, the Hilbert R-tree is the clear winner, achieving up to 28% savings in response time over the next best contender (the R^*-tree). This maximum gain is achieved for the 'MGCounty' dataset (Figure 4.17). It is interesting to note that the performance gap is larger for the real data, whose main difference from the synthetic one is that it is skewed, as opposed to uniform. Thus, we can conjecture that the skewness of the data favors the Hilbert R-tree.

The Effect of the Split Policy on Performance

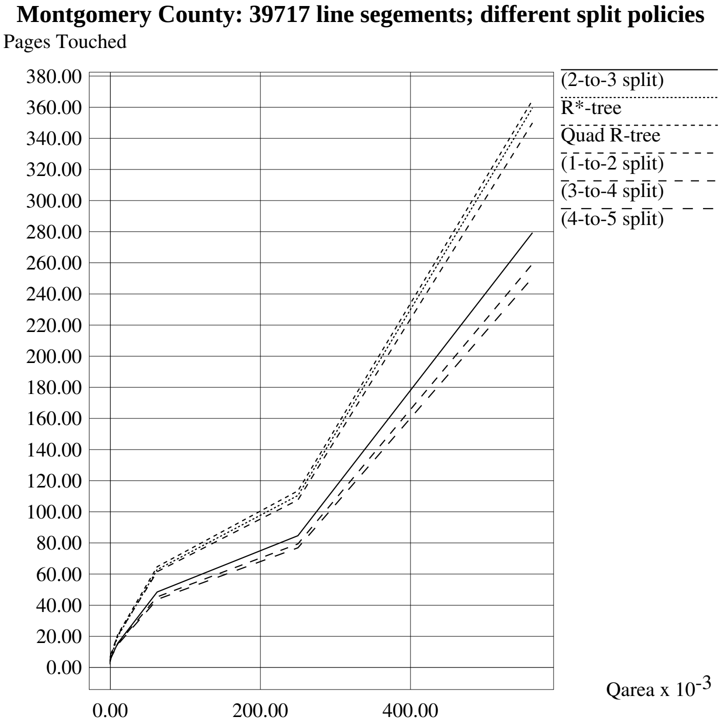


Figure 4.19: The effect of the split policy on the query retrieval time of the Dynamic Hilbert R-tree.

This section shows how the increase in the split policy affects the performance of the Hilbert R-tree. Intuitively, with the good clustering property of the Hilbert R-tree, we expect the total area and perimeter of the resulting R-tree nodes to decrease with increasing space utilization. Consequently, the number of nodes retrieved by a query is expected to decrease. One would, of course, expect the insertion cost to increase with increasing the split policy (see Section 4.5.2). Figure 4.19 shows the response time as a function of the query size for the 1-to-2, 2-to-3, 3-to-4 and 4-to-5 split policies. The corresponding space utilizations were 65.5%, 82.2%, 89.1%, and 92.3% respectively. For comparison, we also plot the response times (in terms of the number of disk accesses) of the R^*-tree. As expected, the response time for the range queries improves with the average node utilization. However, there seems to be a point of diminishing returns as s increases. For this reason, we recommend the '2-to-3' splitting policy, which strikes a balance between insertion speed (which deteriorates with s) and search speed, which improves with s.

It is interesting to note that even with the simple 1-to-2 splitting policy (i.e., no rotation), the Hilbert R-tree performs better than the quadratic split R-tree and at least as well as the R^*-tree. With the recommended 2-to-3 splitting policy, the Hilbert R-tree clearly does better than the R^*-tree and the quadratic split R-tree.

Insertion Cost

The higher space utilization in the Hilbert R-tree comes at the expense of higher insertion cost. As we employ a higher split policy, the number of cooperating siblings need to be inspected at overflow increases. We show that the '2-to-3' policy is a good compromise between the performance and the insertion cost. In this section we present experimental results which compare the insertion cost of the Hilbert R-tree '2-to-3' split with the insertion cost in the R^*-tree. Also, we show the effect of the split policy on the insertion cost. The cost is measured by the number of disk accesses per insertion.

Table 4.6 shows the insertion cost of the Hilbert R-tree and of the R^*-tree for the five different datasets. The main observation here is that there is no clear winner in the insertion cost. Although the R^*-tree does not employ local rotation as does the Hilbert R-tree, it has insertion cost comparable to that of

dataset	(disk accesses)/insertion	
	Hilbert R-tree (2-to-3 split)	$R^* - tree$
MGCounty	3.55	3.10
LBeach	3.56	4.01
Points	3.66	4.06
Rects	3.95	4.07
Mix	3.47	3.39

Table 4.6: Comparison of insertion cost between the Hilbert R-tree with '2-to-3' split and the R^*-tree; disk accesses per insertion (average over all datasets).

split policy	(disk accesses)/insertion
1-to-2	3.23
2-to-3	3.55
3-to-4	4.09
4-to-5	4.72

Table 4.7: The effect of the split policy on the insertion cost of the Hilbert R-tree '2-to-3' split; MGCounty dataset.

the Hilbert R-tree. This is because the R^*-tree employs the 'forced reinsert' technique. When a new data rectangle is inserted into the R^*-tree, the first overflow on each level will be treated by deleting 30% of the entries of the overflowing node and by reinserting them in the tree. Note that more than one overflows might take place as a result of one insertion. The number of times the forced reinsert is performed is unpredictable, and it even depends on the insertion order of the data rectangles. This means that the insertion cost in the R^*-tree might differ for the same data set if the insertion order is changed. This also explains the significant difference in the insertion cost for 'MGCounty' and 'LBeach' in the R^*-tree although both datasets represent roads in two counties and both have the same insertion cost under the Hilbert R-tree. In contrast, the insertion cost in the Hilbert R-tree is less dependent on the insertion order and depends rather on the split policy. Since the R^*-tree reinserts 30% of the overflowed node, we expect that the gap between the insertion cost of the R^*-tree and that of the Hilbert R-tree would increase with increasing node size.

Table 4.7 shows the effect of increasing the split policy in the Hilbert R-tree on the insertion cost for the $MGCounty$ dataset. As expected, the insertion cost increases monotonically with the order s of the split policy. This is simply because the number of cooperating siblings $s - 1$ that will be retrieved when an overflow occurs increases with increasing split policy.

4.6 Discussion

In this chapter we designed and implemented a new R-tree variant which outperform all previous R-tree methods in rectangular query retrieval. The major idea is to introduce a method for achieving 'good' ordering among rectangles. We introduced two variants of the Hilbert R-tree for static and dynamic environments.

For static databases, our algorithms exploit the good clustering properties of the Hilbert curve. We proposed several schemes for sorting the data rectangles before grouping them into R-tree nodes. We performed experiments using these methods and the most promising competitors; our conclusion is that the proposed algorithms result in better R-trees. Specifically, the most successful variation (2D-c = 2-d Hilbert curve through centers) consistently outperforms the best dynamic methods, namely, the R^*-trees and the quadratic split R-trees, as well as the only previously known static method (*lowx* packed R-tree). More importantly, the performance gap seems to be wider for real, skewed data distributions. We also showed that the insertion cost is not penalized as one might expect.

For the dynamic environment we introduced the Dynamic Hilbert R-tree. By simply defining an ordering, the R-tree structure is amenable to local rotation; this fact allows the utilization to approach the 100% mark as closely as we want. Better packing results in a shallower tree and a higher fanout. If the ordering happens to be 'good', that is, happens to group similar rectangles together, then the R-tree will also have nodes with small MBRs, and eventually, fast response times.

With this considerations in view, we designed in detail and implemented the Hilbert R-tree, a dynamic tree structure that is capable of handling insertions and deletions. Experiments on real and synthetic data showed that the proposed Hilbert R-tree with the '2-to-3' splitting policy consistently outperforms all other

R-tree methods in rectangular query retrieval with up to 28% savings over the best competitor (the R^*-tree).

Moreover we provided an analytical formula (Eq. 4.3) to estimate the response time of an already built R-tree. From a practical point of view, it can help a query optimizer [33, 2] give a good estimate for the cost of an R-tree index. Moreover, it makes the simulation analysis of R-trees easier and more reliable, eliminating the need to ask queries.

Chapter 5

Conclusions – Future Work

In this dissertation we have studied how to improve the performance of spatial indexing methods and specifically of R-trees under both parallel I/O and centralized environments.

For a parallel I/O environment we proposed a parallel I/O R-tree design for a server with one CPU and multiple disks. On this architecture, the nodes of the R-tree are distributed between the different disks with cross-disk pointers (*'Multiplexed R-tree'*). When a new node is created, we have to decide on which disk it will be stored. We proposed and examined several criteria for choosing a disk for a new node. The most successful one, termed *'Proximity Index'* or PI, estimates the similarity of the new node with the other R-tree nodes already on a disk and chooses the disk with the least degree of similarity. Our experiments on real data showed that our PI scheme consistently outperforms all other heuristics for node-to-disk assignments, with 55% gains over the Round Robin one.

For a centralized environment, we proposed a new packing technique for R-trees for static databases. We used space-filling curves and specifically the Hilbert curve to achieve better ordering of rectangles and eventually better packing. Our method achieves better performance than other packing algorithms and all other R-tree variations. For dynamic databases we introduced the *Hilbert R-tree*, in which every node has a well-defined set of sibling nodes; we can thus implement the concept of local rotation. By adjusting the split policy, the *Hilbert R-tree* can achieve as high a degree of utilization as desired. In contrast, the R-tree/R*-tree has no control over utilization, typically achieving only 50% to 70%.

Future research directions include the following:

- Extension of Parallel I/O R-tree structures on shared-nothing multicomputers: This architecture consists of several computers (e.g. , workstations), each one with its own memory and I/O system. The main advantage of shared-nothing multicomputers is that they can be scaled up to hundreds and probably thousands of computers that do not interfere with one another. This environment differs from the multi-disk environment in two aspects: 1) the number and volume of messages becomes an issue, and 2) the setup time (= time to initiate a query on a processor) can not be neglected. Of course, the total setup time increases with the number of processors involved in the query. For a system consisting of thousands of processors, the setup time for executing a query in parallel will constitute a substantial amount of the query execution time.

- Finally, there are many interesting problem to be studied in *multimedia indexing*. One of the promising areas of research is the indexing of objects to answer "similarity" queries. For example, in multimedia databases with audio (voice, music), video, etc., users might want to retrieve similar objects, such as music scores or video clips. One way to handle the problem is to map the objects into some feature space as multi-dimensional points [18, 42], and subsequently to organize them in a SAM.

Bibliography

[1] D. Abel and J. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision Graph. Image Process.*, 24(1):1–13, October 1983.

[2] W. Aref. *Query processing and optimization in spatial databases.* PhD thesis, Computer Vision Lab., Center for Automation Research, University of Maryland at College Park, August 1993. Also available as Technical Report CAR-TR-676, CS-TR-3097.

[3] W. Aref and H. Samet. Optimization strategies for spatial query processing. In *Proc. of VLDB Conf.*, pages 81–90, Barcelona, Spain, September 1991.

[4] M. Arya, W. Cody, C. Faloutsos, J. Richardson, and A. Toga. QBISM: a prototype 3-D medical image database system. *IEEE Data Engineering Bulletin*, 16(1):38–42, March 1993.

[5] M. Arya, W. Cody, and I. Kamel. Integrating visualization and database systems: A statement of direction. *Workshop on Database Issues for Data Visualization, Visualization'93*, November 1993.

[6] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[7] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[8] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of ACM*, 18(9):509–517, September 1975.

[9] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Trans. on Information Theory*, IT-15(6):658–664, November 1969.

[10] T. Brinkhoff, R. Schneider H. Kriegel, and B. Seeger. Multi-step processing of spatial joins. In *In Proc. of ACM SIGMOD*, Minneapolis, MN, May 1994.

[11] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of ACM SIGMOD*, pages 237–246, Washington, D.C., May 1993.

[12] M. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of ACM*, 39(2), April 1992.

[13] H. Du and J. Sobolewski. Disk allocation for cartesian product files on multiple disk systems. *ACM Trans. Database Systems (TODS)*, 7(1):82–101, March 1982.

[14] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. on Software Engineering*, 14(10):1381–1393, October 1988. early version available as UMIACS-TR-87-4, also CS-TR-1796.

[15] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *2nd Int. Conference on Parallel and Distributed Information Systems (PDIS)*, pages 18–25, San Diego, CA, January 1993.

[16] C. Faloutsos and I. Kamel. High performance R-trees. *IEEE Data Engineering Bulletin*, 16(3):28–33, September 1993.

[17] C. Faloutsos and D. Metaxas. Declustering using error correcting codes. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 253–258, Philadelphia, PA, March 1989. Also available as UMIACS-TR-88-91 and CS-TR-2157.

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of ACM SIGMOD*, pages 419–429, Minneapolis, MN, May 1994.

[19] C. Faloutsos and W. Rego. Tri-cell: a data structure for spatial objects. *Information Systems*, 14(2):131–139, 1989. early version available as UMIACS-TR-87-15, CS-TR-1829.

[20] C. Faloutsos and Y. Rong. DOT: a spatial access method using fractals. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 152–159, Kobe, Japan, April 1991. early version available as UMIACS-TR-89-31, CS-TR-2214.

[21] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, Philadelphia, PA, March 1989. also available as UMIACS-TR-89-47 and CS-TR-2242.

[22] M. Fang, R. Lee, and C. Chang. The idea of de-clustering and its applications. In *Proc. of VLDB Conf.*, pages 181–188, Kyoto, Japan, August 1986.

[23] R. Finkel and J. Bentley. Quad Trees: a data structure for retrieval on composite keys. *ACTA Informatica*, 4(1):1–9, 1974.

[24] M. Freeston. The BANG file: a new kind of Grid file. In *Proc. of ACM SIGMOD*, pages 260–269, San Francisco, CA, May 1987.

[25] H. Fuchs, G. D. Abram, and E. D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics*, 17(3), July 1983.

[26] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a prior tree structures. *Computer Graphics*, 14(3), July 1980.

[27] H. Garcia-Molina and K. Salem. The impact of disk striping on reliability. *IEEE Database Engineering*, 11(1):26–39, March 1988.

[28] I. Gargantini. An effective way to represent quadtrees. *Comm. of ACM*, 25(12):905–910, December 1982.

[29] S. Ghandeharizadeh, D. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. In *Proc. of SIGMOD Conf.*, pages 29–38, San Diego, CA, June 1992.

[30] J. Griffiths. An algorithm for displaying a class of space-filling curves. *Software-Practice and Experience*, 16(5):403–411, May 1986.

[31] O. Gunther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 598–605, Los Angeles, CA, February 1989.

[32] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pages 47–57, Boston, MA, June 1984.

[33] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. of ACM SIGMOD Conf.*, pages 377–388, Portland, OR, May 1989.

[34] A. Henrich, H. Six, and P. Widmayer. The LSD tree: spatial access to multidimentional point and non point objects. In *Proc. of VLDB Conf.*, pages 45–53, Amsterdam, Netherlands, August 1989.

[35] K. Hinrichs and J. Nievergelt. The Grid file: a data structure to support proximity queries on spatial objects. In M. Nagl and J. Perl, editors, *Proc. of the WG'83 (Intern. Workshop on Graph Theoretic Concepts in Computer Science)*, pages 100–113, Linz, Austria, 1983.

[36] E. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proc. of ACM SIGMOD Conf.*, pages 205–214, San Diego, CA, June 1992.

[37] G. Hunter. Efficient computation and data structures for graphics. In *Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University*, Princeton, NJ, 1978.

[38] A. Hutflesz, H. Six, and P. Widmayer. Twin grid files: space optimizing access schemes. In *Proc. of ACM SIGMOD*, pages 183–190, Chicago, IL, June 1988.

[39] C. Jackins and S. Tanimoto. Quad-trees, oct-trees, and k-trees – a generalized approach to recursive decomposition of Euclidean space. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 5(5):533–539, September 1983.

[40] H. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. of ACM SIGMOD Conf.*, pages 332–342, Atlantic City, NJ, May 1990.

[41] H. Jagadish. Spatial search with polyhedra. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 311–319, Los Angeles, CA, February 1990.

[42] H. Jagadish. A retrieval technique for similar shapes. In *Proc. of ACM SIGMOD Conf.*, pages 208–217, Denver, CO, May 1991.

[43] I. Kamel and C. Faloutsos. Parallel R-Trees. In *Proc. of ACM SIGMOD Conf.*, pages 195–204, San Diego, CA, June 1992. Also available as Tech. Report UMIACS TR 92-1, CS-TR-2820.

[44] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. of VLDB Conf.*, Santiago, Chile, September 1994. Also available as Tech. Report UMIACS TR 93-12.1, CS-TR-3032.1.

[45] G. Kedem. The quad-CIF tree: a data structure for hierarical on-line algorithms. In *Proc. of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982.

[46] M. Kim and S. Pramanik. Optimal file distribution for partial match retrieval. In *Proc. ACM SIGMOD Conf.*, pages 173–182, Chicago, IL, June 1988.

[47] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass, 1973.

[48] C. Kolovson and M. Stonebraker. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. In *Proc. of ACM SIGMOD*, pages 138–147, Denver, CO, May 1991.

[49] N. Koudas, C. Faloutsos, and I. Kamel. Declustering r-trees on multicomputer architectures. Technical Report CS-TR-3276, Univ. of Maryland, May 1994.

[50] V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed. The Time Index+: An incremental access structure for temporal databases. *Proc. of 3rd International Conference on Information and Knowledge Management(CIKM-94)*, December 1994.

[51] M. Lo and C. Ravishankar. Spatial joins using seeded trees. In *Proc. of ACM SIGMOD*, pages 209–220, Minneapolis, MN, May 1994.

[52] D. Lomet and B. Salzberg. The hB-Tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[53] B. Mandelbrot. *Fractal Geometry of Nature.* W.H. Freeman, NY, 1977.

[54] A. Narasimhalu and S. Christodoulakis. Multimedia information systems: the unfolding of a reality. *IEEE Computer*, 24(10):6–8, October 1991.

[55] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[56] E. Oomoto and K. Tanaka. OVID: Design and implementation of a video-object database system. *IEEE Trans. on Knowledge and Data Engineering*, 5(4):629–643, August 1993.

[57] J. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[58] J. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. of ACM SIGMOD*, pages 326–336, Washington D.C., May 1986.

[59] J. Orenstein. Redundancy in spatial databases. In *Proc. of ACM SIGMOD Conf.*, pages 294–304, Portland, OR, May 1989.

[60] J. Orenstein and T. Merrett. A class of data structures for associative searching. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 181–190, Waterloo, Ontario, Canada, April 1984.

[61] S. Pramanik and M. Kim. Parallel processing of large node B-trees. *IEEE Trans on Computers*, 39(9):1208–1212, September 1990.

[62] P. Rangan and H. Vin. Efficient storage techniques for digital continous multimedia. *IEEE Trans. on Knowledge and Data Engineering*, 5(4):565–573, August 1993.

[63] J. Robinson. The k-d-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. of ACM SIGMOD*, pages 10–18, Ann Arbor, MI, April 1981.

[64] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proc. of ACM SIGMOD*, pages 17–31, Austin, TX, May 1985.

[65] H. Samet. The quadtree and related hierarchical data structures. *ACM Computer Surveys*, 16(2):187–260, June 1984.

[66] H. Samet. *Applications of Spatial Data Structures Computer Graphics, Image Processing and GIS.* Addison-Wesley, Reading, MA, 1990.

[67] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[68] B. Seeger and H. Kriegel. The Buddy-Tree: an efficient and robust access method for spatial data base systems. In *Proc. of VLDB Conf.*, pages 590–601, Brisbane, Australia, August 1990.

[69] B. Seeger and P. Larson. Multi-disk B-trees. In *Proc. of ACM SIGMOD*, pages 436–445, Denver, CO, May 1991.

[70] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on VLDB*, pages 507–518, England, September 1987. also available as SRC-TR-87-32, UMIACS-TR-87-3, CS-TR-1795.

[71] A. Silberschatz, M. Stonebraker, and J. Ullman. Database systems: Achievements and opportunities. *Comm. of ACM (CACM)*, 34(10):110–120, October 1991.

[72] M. Tamminen. The EXCELL method for efficient geometric access to data. In *Proc. of Design Automation Conference*, pages 345–351, Las Vegas, NV, June 1982.