Parallel R-trees

Ibrahim Kamel and *Christos Faloutsos**

Department of CS
University of Maryland
College Park, MD 20742

Abstract

We consider the problem of exploiting parallelism to accelerate the performance of spatial access methods and specifically, R-trees [11]. Our goal is to design a server for spatial data, so that to maximize the throughput of range queries. This can be achieved by (a) maximizing parallelism for large range queries, and (b) by engaging as few disks as possible on point queries [22].

We propose a simple hardware architecture consisting of one processor with several disks attached to it. On this architecture, we propose to distribute the nodes of a traditional R-tree, with cross-disk pointers ('Multiplexed' R-tree). The R-tree code is identical to the one for a single-disk R-tree, with the only addition that we have to decide which disk a newly created R-tree node should be stored in. We propose and examine several criteria to choose 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 lowest similarity. Experimental results show that our scheme consistently outperforms all the other heuristics for node-to-disk assignments, achieving up to 55% gains over the Round Robin one. Experiments also indicate that the multiplexed R-tree with PI heuristic gives better response time than the disk-stripping (="Super-node") approach, and imposes lighter load on the I/O sub-system.

The speed up of our method is close to linear speed up, increasing with the size of the queries.

*Also, a member of UMIACS. This research was sponsored partially by the National Science Foundation under the grants IRI-8719458 and IRI-8958546, by a Department of Commerce Joint Statistical Agreement JSA-91-9, by a donation from EMPRESS Software Inc. and by a donation from Thinking Machines Inc..

1 Introduction

One of the requirements for the database management systems (DBMSs) of the future is the ability to handle spatial data. Spatial data arise in many applications, including: Cartography [26], Computer-Aided Design (CAD) [18], [10], computer vision and robotics [2], traditional databases, where a record with k attributes corresponds to a point in a k-d space, rule indexing in expert database systems [25], temporal databases, where time can be considered as one more dimension [15], scientific databases, with spatial-temporal data, etc.

In the above applications, one of the most typical queries is the *range query*: Given a rectangle, 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.

In this paper we study the problem of improving the search performance using parallelism, and specifically, multiple disk units. There are two main reasons for using multiple disks, as opposed to a single disk:

(a) All of the above applications will be I/O bound. Our measurements on a DECstation 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 per year of satellite data. 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 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 a lot of data should activate as many disks as possible.

The proposed hardware architecture consists of one processor with several disks attached to it. We do not consider multi-processor architectures, because multiple CPUs will probably be an over-kill, increasing the dollar cost and the complexity of the system without relieving the I/O bottleneck. Moreover, a multi-processor loosely-coupled architecture, like GAMMA [5], will have communication costs, which are non-existing in the proposed single-processor architecture. In addition, our architecture is simple: it requires only widely available off-the-shelf components, without the need for synchronized disks, multiple CPU's, or specialized operating system.

On this architecture, we will distribute the nodes of a traditional R-tree. We propose and study several heuristics on how to choose a disk 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 the least similar contents. Experimental results showed that our scheme consistently outperforms other heuristics.

The paper is organized as follows. Section 2 briefly describes the R-tree and its variants. Also, it surveys previous efforts to parallelize other file structures. Section 3 proposes the 'multiplexed' R-tree as a way to store an R-tree on multiple disks. Section 4 examines alternative criteria to choose a disk for a newly create R-tree node. It also introduces the 'proximity' measure and derives the formulas for it. Section 5 presents experimental results and observations. Section 6 gives the conclusions and directions for future research.

2 Survey

Several spatial access methods have been proposed. A recent survey can be found in [21]. These classification includes methods that transform rectangles into points in a higher dimensionality space [12], methods that use linear quadtrees [8] [1] or, equivalently, the z-ordering [17] or other space filling curves [6] [13], and finally, methods based on trees (k-d-trees [4], k-d-B-trees [19], hB-trees [16], cell-trees [9] e.t.c.)

One of the most characteristic approaches in the last class is the the R-tree [11]. Due to space limitations, we omit a detailed description of the method. Figure 1 illustrates data rectangles (in black), organized in an R-tree with fanout 3. Figure 2 shows the file structure for the same R-tree, where nodes correspond to disk pages.

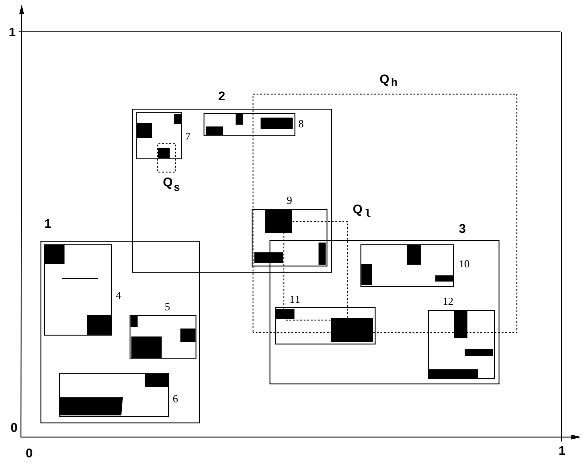

Figure 1: *Data (dark rectangles) organized in an R-tree. Fanout=3 - Dotted rectangles indicate queries*

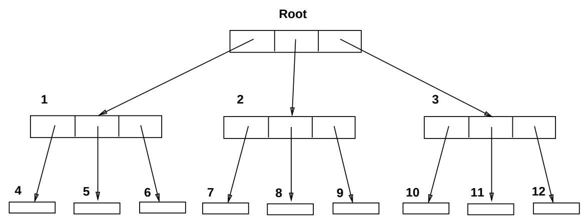

Figure 2: The file structure for the R-tree of the previous figure (fanout=3)

In the rest of this paper, the term 'node' and the term 'page' will be used interchangeably, except when discussing about the super-nodes method, subsection 3.2. Extensions, variations and improvements to the original R-tree structure include the packed R-trees [20], the R^+-tree [23] and the R^*-tree [3].

There is also much work on how to organize traditional file structures on multi-disk or multi-processor machines. For the B-tree, Pramanic and Kim proposed PNB-tree [24] which uses a 'super-node' ('super-page') scheme on synchronized disks. Seeger and Larson [22] 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.

Parallelization of the R-trees is an unexplored topic, to the best of the authors' knowledge. Next, we present some software designs to achieve efficient parallelization.

3 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, to achieve high throughput under concurrent range queries.

The first step is to decide on 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 to do that: (a) d independent R-trees, (b) Disk stripping (or 'super-nodes', or 'super-pages'), and (c) the 'Multiplexed' R-tree, or *MX R-tree* for short, which we describe and propose later. We examine the three approaches qualitatively:

3.1 Independent R-trees

In this scheme we can distribute the data rectangles between the d disks and build a separate R-tree index for each disk. This mainly works 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 using 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 The idea is to divide the space into d partitions, and assign each partition to a separate disk. For example, for the R-tree of Figure 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 (uniform spread, or 'uniSpread' requirement).

3.2 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-stripping [7]. Almost identical performance will be obtained with bit- or byte-level stripping.

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

3.3 Multiplexed ('MX') 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 shows one possible multiplexed R-tree, corresponding to the R-tree of Figure 1. The root node kept in main memory while other nodes are distributed over the disks A, B and C. For the multiplexed R-tree, each

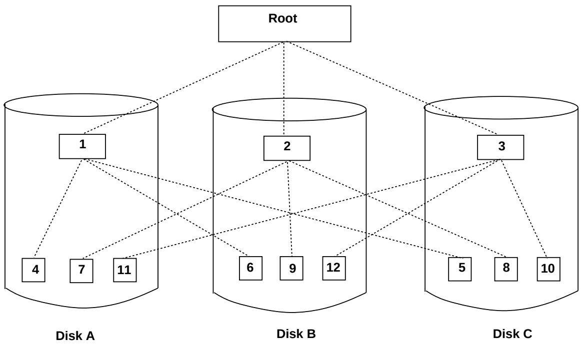

Figure 3: *R-tree stored on three disks*

pointer consists of a disk_id, in addition to the page_id of the traditional R-tree. However, the fanout of the node is not affected, because the disk_id can be encoded within the 4 bytes of the page_id.

Notice that the proposed method fulfills both requirements (minLoad and uniSpread): For example, from Figure 3, we see that the 'small' query Q_s of Figure 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 MX R-tree should outperform 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 to assign nodes to disks.

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

Let $R(q)$ denote the response time for the query q. First, we have to discuss how the search algorithm works. 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 a 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.

The nodes that the disks retrieve are inserted at the end of the processing queue (FIFO). 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 number of disk accesses that the latest disk will require. The 'disk-time' diagram helps visualize this concept better. Figure 4 presents the 'disk-time' diagram for the query Q_l of Figure 1. The horizontal axis is time, divided in slots. The duration of each slot is the time for a disk access (which 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 4 pages in total.

As another example, the 'huge' query Q_h of Figure 1 results in the disk-time diagram of Figure 5, with response time $R(Q_h)$=3, and a load of $L(Q_h)$=7.

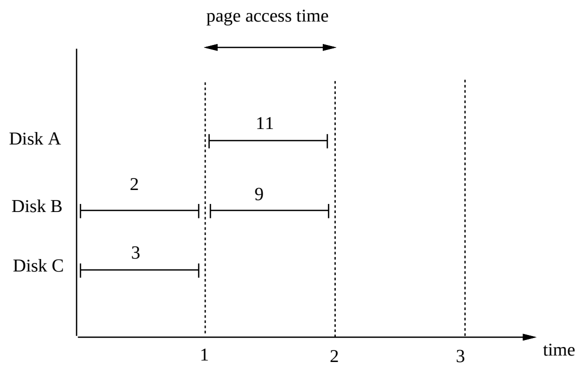

Figure 4: *Disk-Time diagram for the large query Q_l of Figure 1.*

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

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.

4 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, to satisfy the requirement for uniform disk activation ('uniSpread'). As discussed before, the minimum load requirement is fulfilled.

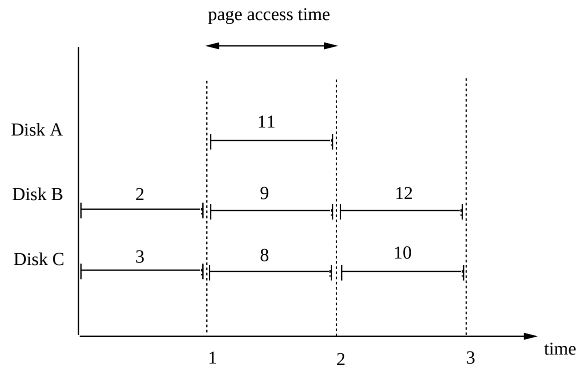

Figure 5: *Disk-Time diagram for the huge query Q_h of Figure 1.*

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 they reside. Ideally, we should consider all the nodes that are on the same level with N_0, before we decide where to store it. However, this will 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} with N_0. Accessing the father node comes at no extra cost, because we have to bring it in 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 (extend of MBR (= minimum bounding rectangle) 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, 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 others, 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 are intersecting, or are close to each other, they should be stored on different disks, to maximize the parallelism.

We can not satisfy all these criteria simultaneously, because some of them may conflict. Next, we describe some heuristics, each trying to satisfy one or more of the above criteria. In Section 5 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 6, 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 6, 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 sum of area.

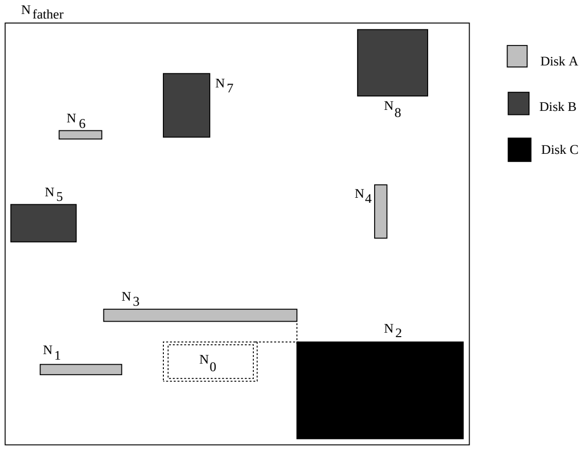

Figure 6: *Node N_0 is to be assigned to one of the three disks.*

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 such a disk, so 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 see soon, it 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.

This heuristic 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 6, PI will assign N_0 to disk B because it contains the most remote rectangles. Intuitively, disk B is the best choice for N_0.

Although favorably prepared, the example of Figure 6 indicates that PI should perform better than the rest of the heuristics. Next we show how to calculate exactly the 'proximity' of two rectangles.

4.1 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 1- and 2- 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,

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

or, formally

$$proximity(R, S) = \frac{\#of\ queries\ retrieving\ both}{total\#\ of\ queries} = \frac{|q|}{|Q|} \quad (1)$$

To avoid complications with infinite numbers, let's assume during this subsection that our address space is discrete, 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 the proximity, given the coordinates of the two rectangles R and S. To simplify the presentation, let's consider the 1-dimensional case first.

1-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 to 2-dimensional points [12] as shown in Figure 7. In the

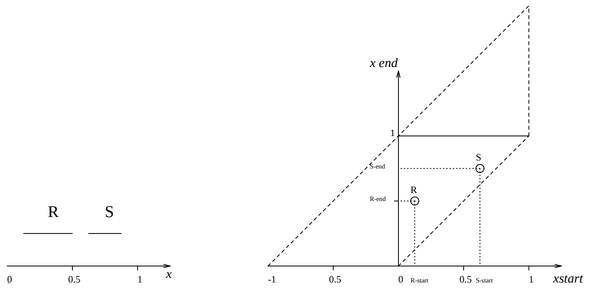

Figure 7: *Mapping line segments to points*

same Figure, the area within the dashed lines is a measure of the number 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 (ie., $r_{start} \leq s_{start}$)

(a) If 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 7; its area is $|Q| = (2 \times 2 - 1 \times 1)/2 = 3/2$. The total number of queries $|q|$ that retrieve both R and S is proportional to the shaded area of Figure 8

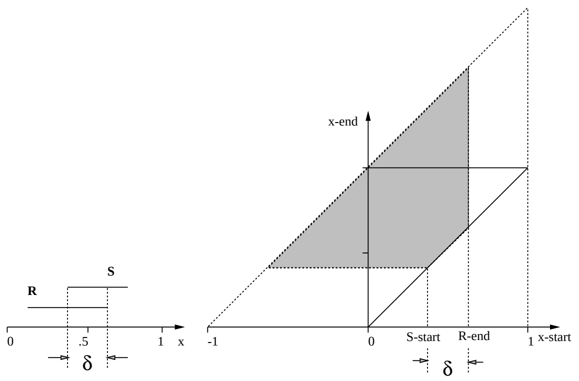

Figure 8: *The shaded area contains all the segments that intersect R and S simultaneously*

$$|q| = ((1 + \delta)^2 - \delta^2)/2 = 1/2 \times (1 + 2 \times \delta) \quad (2)$$

Thus, for intersecting segments R and S we have

$$proximity(R, S) = |q|/|Q| = 1/3 \times (1 + 2 \times \delta) \quad (3)$$

where δ is the length of the intersection

(b) If R and S are disjoint, let Δ be the distance between them (see Figure 9). Then, a query has to

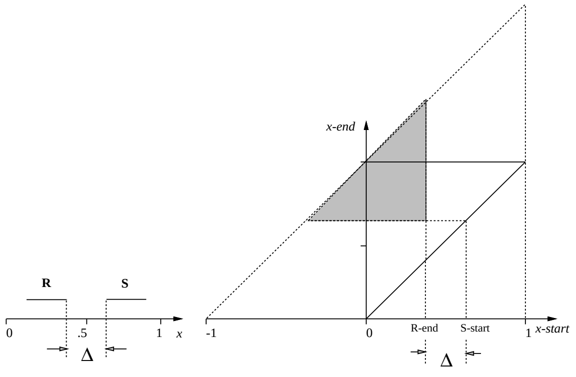

Figure 9: *The shaded area contains all the segments that intersect R and S*

cover the segment (r_{end}, s_{start}), in order to retrieve both segments. The number of such queries is proportional to the shaded area in Figure 9.b; its area is given by

$$q = 1/2 \times (1 - \Delta)^2 \quad (4)$$

and the proximity measure for R and S is

$$proximity(R, S) = |q|/|Q| = 1/3 \times (1 - \Delta)^2 \quad (5)$$

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: The previous formulas can be generalized by assuming uniformity and independence. For a 2-d space, 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:

$$proximity_2(R, S) = proximity(R_x, S_x) \times proximity(R_y, S_y) \quad (6)$$

The generalization for n-dimensions is straightforward:

$$proximity_n(R, S) = \prod_{i=1}^{n} proximity(R_i, S_i) \quad (7)$$

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 and 5.

The proximity *index* measures the similarity of a rectangle R_0 with 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

Symbols	Definitions
a	average area of a data rectangle
c	cover quotient
d	number of disks
$diskOf()$	maps nodes to disks
$L(q)$	total number of pages touched by q ('Load')
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
$r(q)$	relative response time (compared to PI)
s	speed-up

Table 1: Summary of Symbols and Definitions

the set $\mathcal{R}$. The proximity index is the proximity of the most similar rectangle in $\mathcal{R}$. Formally:

$$proximityIndex(R, \mathcal{R}) = \max_{R_i \in \mathcal{R}} proximity_n(R, R_i) \quad (8)$$

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

5 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()' e.t.c.) 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 [11]. The quadratic algorithm resulted in better R-trees, i.e., with smaller father nodes. The exponential algorithm was very slow and it 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 d disk units are identical.
- the page access time is constant.
- 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 a number of disks we have examined (1-25 disks), the delay caused by the CPU is negligible.

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 input parameters, ranging as mentioned: The number of disks d (5-25); the total number of data rectangles N (25,000 to 200,000); the size of queries $q_s \times q_s$ (q_s ranged from 0 (point queries) to 0.25); the page size P (1Kb to 4Kb).

Another important factor, which is derived from N and the average area a of the data rectangles, is the "cover quotient" c (or "density") 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: $c = N \times a$. For the selected values of N and a, the cover quotient ranges from 0.25 - 2.0.

The 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 therefore is the proposed approach. To make the comparison easier, we normalize the response time of the different heuristics to that of the proximity index and plot the ratios of the response times.

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 more detailed study of the RR heuristic vs. the PI one. (c) A comparison of the PI vs. super-node method (d) A study of the speed-up achieved by PI.

5.1 Comparison of the disk assignment heuristics

Figure 10 shows the actual response time for each of the four heuristics (RR, MI, MA, PI), as a function of the query size q_s. The parameters were as follows: N=25,000 c=0.26, d=10, P=4. This behavior is typical for several combinations of the parameter values: P=1,2,4; c=0.5,1,2; d=5,10,20. The main observation is that PI and MI, the two heuristics that take the spatial relationships into account, perform the best. Round

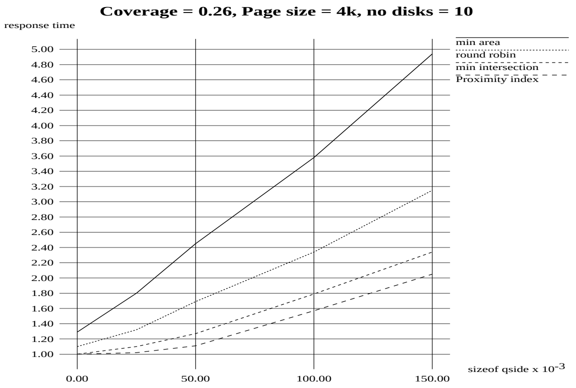

Figure 10: *Comparison among all heuristics (PI,MI,RR and MA)- response time vs query size*

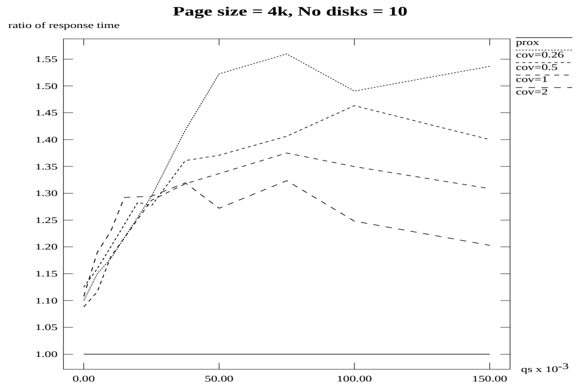

Figure 12: *Relative response time (RR over PI) vs query size.*

Robin is the next best, while the Minimum Area heuristic has 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.

5.2 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.

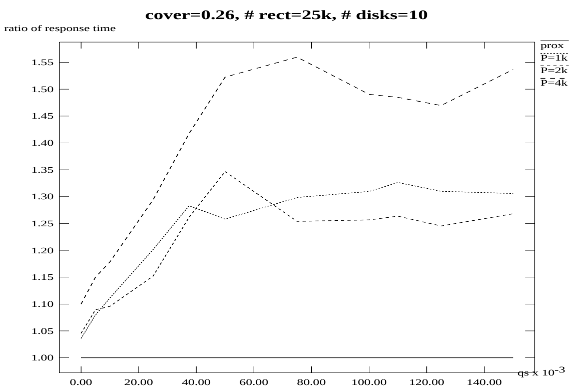

Figure 11: *Relative response time (RR over PI) vs query size.*

Figure 11 plots the response time of RR relative to PI as a function of the query size q_s. The number of disks was d=10, the cover quotient is c=0.26 and the page sizes P varied: 1, 2 and 4Kb. The conclusion is 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 the page size, the heuristic takes more nodes into account, and therefore makes better decisions.

Figure 12 illustrates the effect of the cover quotient on the relative gains of PI over RR. The page size P was fixed at 4Kb; the cover quotient varied (c=0.26, 0.5, 1 and 2). Everything else was the same as in Figure 11. The main observation is that $r(q)$ decreases with the cover quotient c. This is explained as follows: For large c, there are more rectangles in the vicinity of the newly created rectangle, which means that we need to take more sibling nodes into account and use more disk units to get a better results.

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 should re-emphasize that Figures 11 and 12 reveal only a small fraction of the experimental data we gathered. We also performed experiments with several values for the number of disks d, as well as with the linear splitting algorithm for the R-tree. In all our experiments, PI invariantly outperformed RR.

5.3 Comparison with the super-node method

In order to justify our claims about the advantages of the Multiplexed ('MX') R-tree over the super-node method, we compared the two methods with respect to the two requirements, 'uniform spread' and 'minimum load'. The measure for the first 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 13 compares the response time of the Multiplexed R-tree (with PI) against the super-node

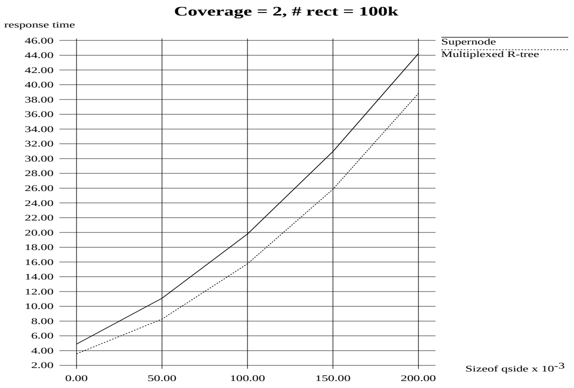

Figure 13: *Response time vs query size for Multiplexed R-tree, with PI, and for super-nodes. (d=5 disks)*

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 d and the query is small. The explanation is that, since d is large, the R-tree with super-nodes has fewer levels than the multiplexed R-tree; in addition, since the query is small, the response time of both trees is the 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 on the effect on the throughput, we plot the 'load' for each method, for 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 d simple pages). Figure 14 shows the results for the same setting as before (Figure 13). The multiplexed R-tree imposes a much lighter load: for small queries, its load is 2-3 times smaller 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, it 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.

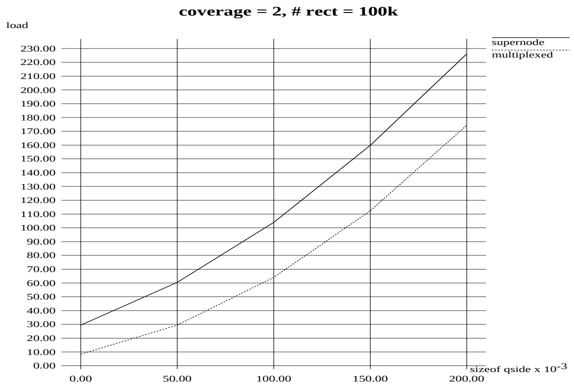

Figure 14: *Total number of pages retrieved (load), vs. query size q_s - d=5.*

5.4 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 d disks. Then: $s = R_1(q)/R_d(q)$ We examined exclusively the multiplexed R-tree method, with the PI heuristic, since it seems to offer the best performance. Due to space limitations, we report here the conclusions only; the details are in a technical report [14]. The major result was that the speed up is high, eg. 84% of the linear speedup, for cover quotient c=2, page size P=4 and for query side size q_s=0.25. Moreover, the speed up increases with the size of the query.

6 Conclusions

We have studied alternative designs for a spatial object server, using R-trees as the underlying file structure. Our goal is to maximize the parallelism for large queries, and on the same time to engage as few disks as possible for small queries. To achieve these goals, we propose

- a hardware architecture with one CPU and multiple disk, which 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 take easily advantage of large buffer pools.
- a software architecture (termed 'Multiplexed' R-tree). It operates exactly like a single-disk R-tree, with the only difference that its nodes are carefully distributed over the 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 d

disks. Specifically, it tries to store a new node on that one disk that contains nodes as dissimilar to the new node as possible.

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

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

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

Future research could focus on the use of the proximity concept to aid the parallelization of other spatial file structures.

Acknowledgements: The authors would like to thank George Panagopoulos, for his help with the typesetting.

References

[1] Walid G. Aref and Hanan Samet. Optimization strategies for spatial query processing. *Proc. of VLDB (Very Large Data Bases)*, pages 81–90, September 1991.

[2] D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, 1982.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD*, pages 322–331, May 1990.

[4] J.L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, September 1975.

[5] D. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proc. 12th International Conference on VLDB*, pages 228–237, Kyoto, Japan, August 1986.

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

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

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

[9] O. Gunther. The cell tree: an index for geometric data. Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, December 1986.

[10] A. Guttman. *New Features for Relational Database Systems to Support CAD Applications.* PhD thesis, University of California, Berkeley, June 1984.

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

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

[13] H.V. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD Conf.*, pages 332–342, May 1990.

[14] Ibrahim Kamel and Christos Faloutsos. Parallel r-trees. *Proc. of ACM SIGMOD Conf.*, pages 195–204, June 1992. Also available as Tech. Report UMIACS TR 92-1, CS-TR-2820.

[15] Curtis P. Kolovson and Michael Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proc. ACM SIGMOD*, pages 138–147, May 1991.

[16] David B. Lomet and Betty Salzberg. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[17] J. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD*, pages 326–336, May 1986.

[18] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: a vlsi layout system. In *21st Design Automation Conference*, pages 152 – 159, Alburquerque, NM, June 1984.

[19] J.T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD*, pages 10–18, 1981.

[20] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *Proc. ACM SIGMOD*, May 1985.

[21] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[22] Bernhard Seeger and Per-Ake Larson. Multi-disk b-trees. *Proc. ACM SIGMOD*, pages 138–147, May 1991.

[23] 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.

[24] S.Pramanik and M.H. Kim. Parallel processing of large node b-trees. *Trans on Computers*, 39(9):1208–1212, 90.

[25] M. Stonebraker, T. Sellis, and E. Hanson. Rule indexing implementations in database systems. In *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April 1986.

[26] M. White. *N-Trees: Large Ordered Indexes for Multi-Dimensional Space.* Application Mathematics Research Staff, Statistical Research Division, U.S. Bureau of the Census, December 1981.