Declustering spatial databases on a multi-computer architecture

Nikos Koudas[1] and *Christos Faloutsos*[2*] and *Ibrahim Kamel*[3]

[1] Computer Systems Research Institute
University of Toronto
[2] AT&T Bell Laboratories
Murray Hill, NJ
[3] Matsushita Information
Technology Laboratory

Abstract. We present a technique to decluster a spatial access method on a shared-nothing multi-computer architecture [DGS+90]. We propose a software architecture with the R-tree as the underlying spatial access method, with its non-leaf levels on the 'master-server' and its leaf nodes distributed across the servers. The major contribution of our work is the study of the optimal capacity of leaf nodes, or 'chunk size' (or 'striping unit'): we express the response time on range queries as a function of the 'chunk size', and we show how to optimize it.
We implemented our method on a network of workstations, using a real dataset, and we compared the experimental and the theoretical results. The conclusion is that our formula for the response time is *very* accurate (the maximum relative error was 29%; the typical error was in the vicinity of 10-15%). We illustrate one of the possible ways to exploit such an accurate formula, by examining several 'what-if' scenarios. One major, practical conclusion is that a chunk size of 1 page gives either optimal or close to optimal results, for a wide range of the parameters.

Keywords: Parallel data bases, spatial access methods, shared nothing architecture.

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 [Whi81], Computer-Aided Design (CAD) [OHM+84], [Gut84a], computer vision and robotics [BB82], traditional databases, where a record with k attributes corresponds to a point in a k-d space, rule indexing in expert database systems [SSH86], temporal databases [SS88], where time can be treated as one more dimension [KS91], scientific databases, with spatial-temporal data, etc.

In several of these applications the volume of data is huge, necessitating the use of multiple units. 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 of the U.S. Bureau of

* On leave from the University of Maryland, College Park. His research was partially funded by the Institute for Systems Research (ISR), and by the National Science Foundation under Grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc.

Census is 19 Gigabytes. Historic and temporal databases tend to archive all the changes and grow quickly in size.

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.

We study the use of parallelism in order to improve the response time of spatial queries. We plan to use R-trees [Gut84b] as our underlying data structure, because they guarantee good space utilization, they treat geometric objects as a whole and they give good response time.

We envision a shared-nothing architecture, with several workstations connected to a LAN. The challenge is to organize the data on the available units, in order to minimize the response time for range queries.

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 our architecture and describes its parameters and components. Section 4 presents the analysis for computing the optimal chunk size. Section 5 presents experimental results and validates the formulas derived from our analysis. Section 6 lists the conclusions and highlights directions for future work.

2 Survey

Several spatial access methods have been proposed. A recent survey and classification can be found in [Sam90]. This classification includes (a) methods that transform rectangles into points in a higher dimensionality space [HN83], subsequently using a *point access method*, like a grid file [NHS84] (b) methods that use linear quadtrees [Gar82] or, equivalently, the z-ordering [Ore86] or other space filling curves [FR89] [Jag90], and finally, (c) methods based on trees (k-d-trees [Ben75], k-d-B-trees [Rob81], cell-trees [Gun86], the BANG file [Fre87], hB-trees [LS90], n-d generalizations of B-trees [Fre95] e.t.c.)

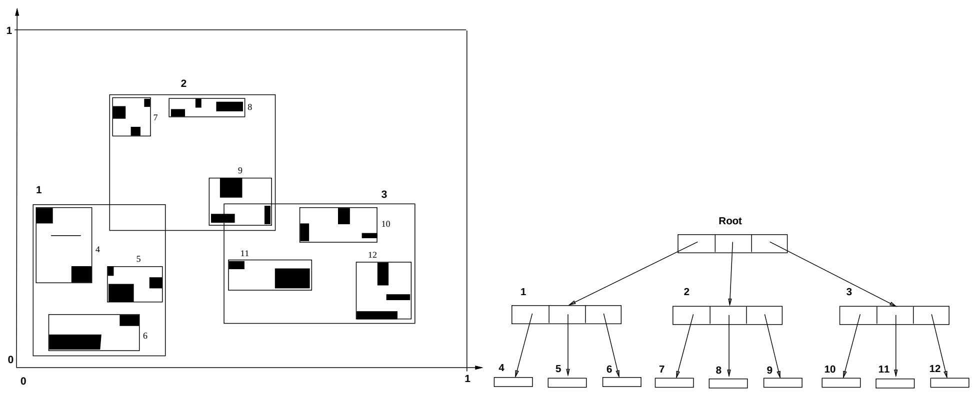


Fig. 1. (a) Data (dark rectangles) organized in an R-tree with fanout=3 (b) the resulting R-tree, on disk.

Among the above three approaches, we focus on the R-tree family, for the following reasons:

- the R-trees do not cut data rectangles into pieces (unlike the linear-quadtree approach). Cutting data into pieces results in an artificially increased database size (linear on the number of *pieces*); moreover, it requires a duplicate-elimination step, because a query may retrieve the same object-id several times (once for each piece of the qualifying object)
- the R-trees operate on the native address space, which is of lower dimensionality; in contrast, transforming rectangles into points in a higher-dimensionality space invites the 'dimensionality curse' problems early on.
- R-trees seem more robust for higher dimensionalities [FBF+94]. Scientific, medical and statistical databases may involve several dimensions, eg., (x, y, z, time, pressure, wind velocity, temperature), or (gender, age, cholesterol-level, blood-pressure, etc.).

Thus, we mainly focus on the R-tree [Gut84b] and its variants. The R-tree is an extension of the B-tree for multidimensional objects. 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 main innovation in the R-tree is that parent nodes are allowed to overlap. This way, the R-tree can guarantee good space utilization and remain balanced. Figure 2 illustrates data rectangles, (in black), organized in an R-tree with fanout 3 (a), while (b) shows the file structure for the same R-tree, where nodes correspond to disk pages.

The R-tree inspired much subsequent work, whose main focus was to improve the search time. A packing technique is proposed in [RL85] to minimize the overlap between different nodes in the R-tree for static data. An improved packing technique based on the Hilbert Curve is proposed in [KF93]; it is extended for dynamic environments in [KF94]. The R^+-tree [SRF87] avoids the overlap between non-leaf nodes of the tree, by clipping data rectangles that cross node boundaries. Beckmann et. al. proposed the R^*-tree [BKSS90], which seems to have very good performance. The main idea is the concept of *forced re-insert*, which tries to defer the splits, to achieve better utilization: When a node overflows, some of its children are carefully chosen and they are deleted and re-inserted, usually resulting in a better structured R-tree.

There is also much work on how to organize spatial access methods on multi-disk or multi-processor machines. The majority of them examine the parallelization of grid-based structures. Typical representatives include the 'disk modulo allocation' method and its variants [DS82], [WYD87], methods using minimum spanning trees [FLC86], the field-wise exclusive-OR ('FX') method [KP88], methods using error correcting codes [FM89], methods based on the Hilbert curve [FB93], and methods using lattices [CR93]. The objective in all these methods is to maximize the parallelism for partial match or range queries. An adaptive algorithm to achieve dynamic re-declustering, to 'cool-off' hot spots is presented in [WZS91].

However, the above methods try to do the best possible declustering without taking into account the communication cost. One of the exceptions is the work by Ghandeharizadeh et. al. [GDQ92], which considers a grid file with a certain profile of range queries against it; a major contribution of this work is a formula to estimate the optimal number of activated processors for a given query.

Here, we focus on the parallelization of R-trees. The difference between the R-tree and the grid file is that the latter will suffer if the attributes are correlated. Moreover, the grid file is mainly designed for point data; if the data are rectangles, the R-tree is better equipped to handle them. Little work has been done on the parallelization of R-trees: In [KF92] we studied the multi-disk architecture, with no communication cost, and we proposed the so-called 'proximity index' to measure the dis-similarity of two rectangles, in order to decide which unit to assign each rectangle to. DeWitt et al. [DKL+94] discuss a client-server architecture for Geographical Information Systems, with a rich data model and a storage manager that uses a variation of R-trees.

In this paper, we present a software design to achieve efficient parallelization of R-trees on a multi-computer ('shared-nothing') architecture, using commodity hardware and interconnect, as well as fast processors. Unlike previous work, our major focus is to optimize the *unit of declustering* itself, which will implicitly optimize the number of activated processors. We derive closed-form formulas and we validate our results experimentally, on a network of SUN workstations, operating on real data.

3 Proposed Method and System Architecture

An overview of the hardware architecture is in figure 2. It consists of a number of workstations connected together with a LAN (e.g., Ethernet) that does not support *multi-casting* (ability to send a message at all or a subset of nodes at the cost of one message). The problem is defined as follows:

Given a set of n-dimensional rectangles
Organize them on the available machines
to minimize the response time for range queries.

Target applications are, for example, GIS applications (*'retrieve the elevation data for a given region'*); statistical queries (eg., *'find the average salary in the state of Michigan, for a census database'*); 'data mining' applications [AS94] and decision support systems (eg., *'find correlations/rules among demographic data and symptoms, for a collection of patient records'*).

Given a spatial database, our first reaction would be to do declustering over the available machines. Thus, similar (=nearby) data rectangles should not be stored on the same machine. This approach is only partially correct, because it neglects the communication cost. In this case, it is not optimal to decluster at the data-rectangle level. The reason is the following: a small query, who retrieves, say, 5 data rectangles, will have to engage 5 machines (sending at least 5 messages on the network). For such a small query, it would be best if those 5 data rectangles were stored on the same machine, which hopefully could be identified easily and could get activated with 1 only message.

Thus, we still want to do declustering, but not on the rectangle level. Therefore we propose (a) to group similar data rectangles in 'chunks' (also termed *'striping units'*, eg., in the RAID literature) and (b) to decluster the *chunks*. We have the following three design decisions:

1. what is the optimal *chunk* size, i.e., the unit of the amount of data that we should place in every server such that the search time is minimized.
2. data placement: Once the chunk size has been determined, what is the best algorithm to distribute the chunks among several servers, to optimize the response time and the parallelism.
3. How to do the book-keeping, to record the server-id for each chunk.

We discuss briefly the last two issues in the next two subsections. The first issue, the optimal chunk size selection, is the main focus of this work, and is discussed in detail in section 4.

3.1 Book-keeping

As mentioned before, we have to do some book-keeping, to record which chunks reside on which machine. This is necessary, because we need to know which machine should be activated, as well as where to put new data rectangles, if insertions (and, eventually, splits) are allowed.

We propose to dedicate one of the machines to the book-keeping. This machine will be referred to as the '*master server*' (≡ 'host', in the terminology of database machines). The rest of the machines are plain *servers*: they receive a (range) query and some chunk requests from the master server, they retrieve the appropriate chunks from their local disks and ship the (qualifying) data rectangles to the master server.

A natural way for the master server to do the book-keeping is through a spatial access method. For the reasons explained in section 2, we propose to use an R-tree (any R-tree variant would work). Figure 3 gives an example of an R-tree, distributed across the master server and N=3 servers. Chunks are stored on the servers and they are represented by their MBRs. Chunks correspond to the leaves of the R-tree. We have decided to store *all* the non-leaf nodes in the master server, for the following reasons:

- the non-leaf portion of the R-tree in the master server will be small (for an R-tree with a fanout of f=100, the non-leaf nodes will take roughly 1/100 of the total space of the SAM). Thus, the non-leaf portion of the R-tree might even fit in the main memory of the master server, and definitely on its disk.
- It does not pay off to decluster the non-leaf levels of the R-tree across the servers, because this will introduce inter-server pointers, higher communication cost, and longer delays, as each node access may have to go through a potentially slow/congested network.

The proposed method has only the following structural differences from a traditional, centralized R-tree;

- the leaf nodes (=chunks) span C pages, where C is not necessarily 1 page.
- the pointers from the master server to the chunks (denoted by dashed lines in Figure 3) consist of a chunk-identifier and a machine-identifier.

The query-resolution algorithm for the proposed system is as follows:

- A query starts at the master server, who examines the (locally stored) non-leaf nodes of the R-tree. Note that this search is done in the normal way that R-trees are searched, by comparing MBR's. This search will result in some chunk-ids and the corresponding machine-ids
- The master server activates the servers identified in the previous step, by sending a message. The message contains the MBR of the query and the chunk id's to be retrieved from the specific server. Notice that the activation can be done either by broadcasting, or by 'selective activation'. Both our implementation and our analysis can handle either case, with small modifications; however, we focus on the 'selective activation', because it scales-up better. This is the typical approach that commercial parallel DBMSs follow (eg., the parallel version of IBM's DB2 [BFG+95]).

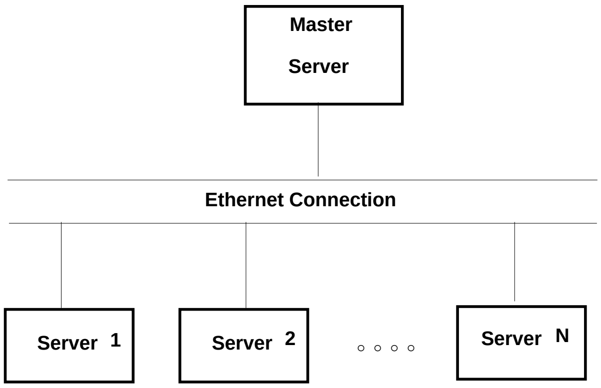


Fig. 2.
The proposed architecture

– Each activated server fetches the relevant chunks from its local disk, processes them (to discard data rectangles that do not intersect the query rectangle), and ships the qualifying data back to the master server.

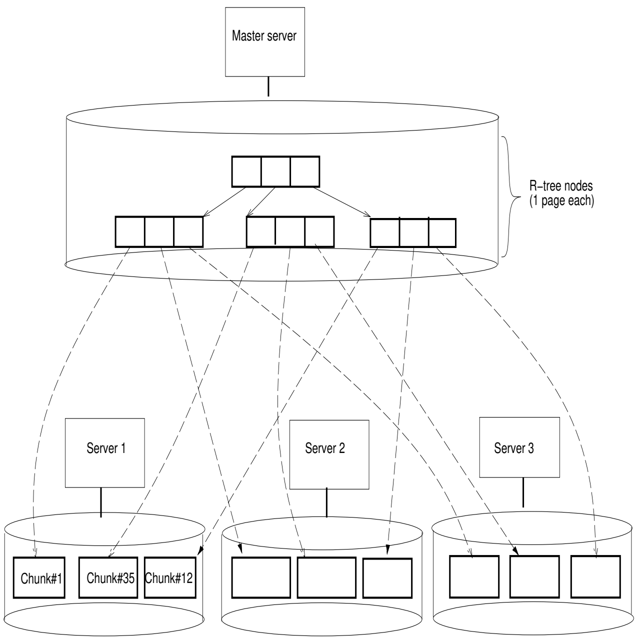


Fig. 3.
Parallel Decomposition for 3 servers

3.2 Data placement

The goal is that 'similar' chunks should be declustered: that is, if two chunks have MBR's that are nearby in space and therefore likely to qualify under the same range query, these chunks should not be stored on the same server. We distinguish two cases: (a) a static database (no insertions/deletions/updates) and (b) a dynamic one.

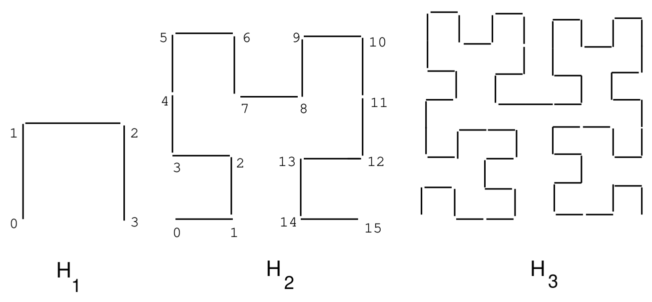


Fig. 4. *Hilbert Curves of order 1,2 and 3*

For a static database, following [FB93], we propose to use Hilbert Curves to achieve good declustering. A Hilbert curve is a type of space filling curve that visits all points in a k-dimensional grid exactly once and never crosses itself. Thus, it can be used to linearize the points of a grid. In [FR89], it was shown experimentally that the Hilbert curve achieves better clustering than several other space-filling curves. We can achieve good declustering by using a variation of the Hilbert-based declustering method [FB93], as applied in the so-called 'Hilbert-packed R-trees' [KF93]). For a static collection of data rectangles, the method works briefly as follows: we sort the data rectangles on the Hilbert values of their centers and pack them into R-tree leaf-nodes ($\equiv$ chunks, each of size C pages); we scan the resulting list of chunks, assigning chunks to servers in a round-robin fashion. Thanks to the good clustering properties of the Hilbert curve, successive chunks will be similar; thanks to the round-robin assignment, they will be assigned to different servers. Thus, similar chunks will be de-clustered.

For a dynamic database, the quality of declustering depends on the insertion and deletion routines. Insertions and deletions in our proposed structure can be handled by modifying the corresponding R-tree routines. The only thing that needs to be added is that, during a split, we have to decide where to put the newly created chunk. There are several criteria, ranging from a random assignment, to elaborate methods such as the so-called *proximity index* [KF92]: This method tries to put a new chunk on a server that has the most un-similar chunks.

To keep the discussion simple and the emphasis on the optimal chunk-size selection, we restrict ourselves to the *static case*, ignoring insertions and splits.

4 Optimal chunk-size selection

In this section we describe our approach to optimize the chunk size. Figure 5 illustrates the problems of a poor choice for the chunk size: a tiny chunk size will result in activating several servers even for small queries (see Q_small in Figure 5(b)), resulting in high message traffic; a huge chunk size will limit parallelism unnecessarily, even for large queries (see Q_large in Figure 5(c)).

The goal of this whole section is to find the optimal chunk-size C_{opt}, so that to minimize the response time for a range query. Thus, the major part of the effort is to express the response time as a function of C and apply some optimization technique (something like setting the first derivative to zero).

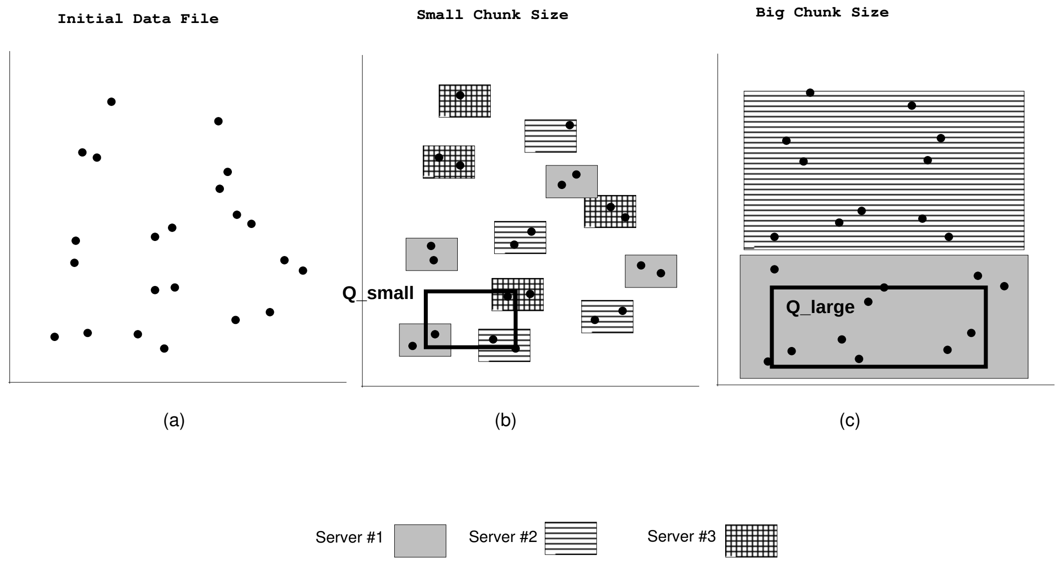


Fig. 5. *Illustration of problems with too small and too large chunk sizes: A small chunk size will force small queries (Q_small) to execute on many servers, resulting in high message traffic. A large chunk size will force large queries (Q_large) to execute in one only server, resulting in little parallelism.*

Symbols	Explanation	Values
N	Number of servers in the architecture	2-9
PS	Average packet size (in pages)	1 page
CC_{idle}	Time to send a message on an idle Network	10ms
CC	Time to send a message and wake up cost under load	-
q_x, q_y	query sides	-
D	Total #of records in the data base	39717
P_{rec}	Records per page	50 rec
$T_{local-per-page}$	Disk plus CPU time to fetch and process 1 page	10 ms
Q	Average #of qualifying pages for query q_x, q_x	-
C	chunk size in Pages	-
C_{opt}	optimal chunk size in Pages	-

Table 1. Symbols, definitions and typical values

As a first step, we shall express the response time as a function of the number K of servers that get activated (also called 'promising servers' from now on). Then, we express K as a function of C (and the volume of qualifying data Q), and finally we optimize the resulting (piece-wise continuous) function. Each subsection is devoted to each of the three above steps.

4.1 Response time as a function of number of activated servers K

Recall that, as soon as a range query arrives to the master server, it consults the R-tree index and determines which K servers to activate; it sends a message to each of them, and it waits for results from all of the K servers.

The method can be applied to any n-dimensional address space. Without loss of generality, we use examples from 2-d space, to simplify the presentation and the drawing of the examples. For further clarity, we assume that the address space has been normalized to the unit square. Let Q be the expected number of data pages that a query of size $q_x \times q_y$ will retrieve. There are various ways to estimate Q, with varying degrees of accuracy: Using the uniformity assumption [KF93], we have $Q = \frac{q_x \times q_y \times D}{P_{rec}}$ where D is the total number of records in the database and P_{rec} is the number of records per page. More accurate estimates require formulas that use the fractal dimension of the specific dataset [FK94]. However, as the experiments section shows, even the uniformity assumption gives satisfactory results because the fractal dimension of the specific data set was $\approx$1.7 [FK94], rather close to the value of 2, that corresponds to the uniformity assumption.

Let $T_{local-per-page}$ be the time that each server takes to process one disk page locally, including the disk access time and the CPU processing time.

The response time or round trip time $RT(q_x, q_y)$ for a query of size $q_x \times q_y$ can be expressed as a sum of (at most) three terms. The first is the local processing time in each server. This time includes CPU time and disk access time and is referred as T_{LOCAL}. The second is the time required to send the query to each server that is involved in the execution, $T_{COMMUNICATION}$. The last is the time required to ship all the qualifying data to the master server, T_{RESULT}. Notice that the time to traverse the R-tree on the master server is ignored because it is constant, regardless of the chunk size, and thus does not participate in the optimization. Moreover, the R-tree of the master server will most probably be small, typically fitting in core, and thus requiring only some small CPU time to be traversed.

Thus the response time can be expressed as:

$$RT(q_x \times q_y) = T_{COMMUNICATION} + T_{LOCAL} + T_{RESULT} \quad (1)$$

To keep the analysis simple (but still accurate, as the experiments show), we make the optimistic assumption of *perfect load balance*: Thus, we assume that all the servers will have the same local processing cost, and will ship back the same amount of qualifying data. In figures 6(a,b) we show how each server spends its time: Gray, black and striped boxes indicate the time receive a message, to do local processing and to ship back the results, respectively. Lack of a box indicates an idle period for this server.

Notice that messages on the network *may not* overlap. The dashed lines in Figure 6 help illustrate exactly this fact. Also, for the moment, we assume that there are no other users on the network; we shall see shortly how to take them into account. Figure 6 illustrates the two possible scenarios, depending on which is the dominating delay (communication time vs. local processing).

Domination of communication time: Figure 6(a) illustrates this situation. The first server receives the message, collects its results, and has to wait, because the network is still busy with the messages from the master server to the rest of the K servers. In this case, T_{LOCAL} will not appear in the expression for $RT(q_x, q_y)$ because this term is completely overlapped by the communication time. Therefore, we have:

$$T_{COMMUNICATION} = K \times CC \quad (2)$$

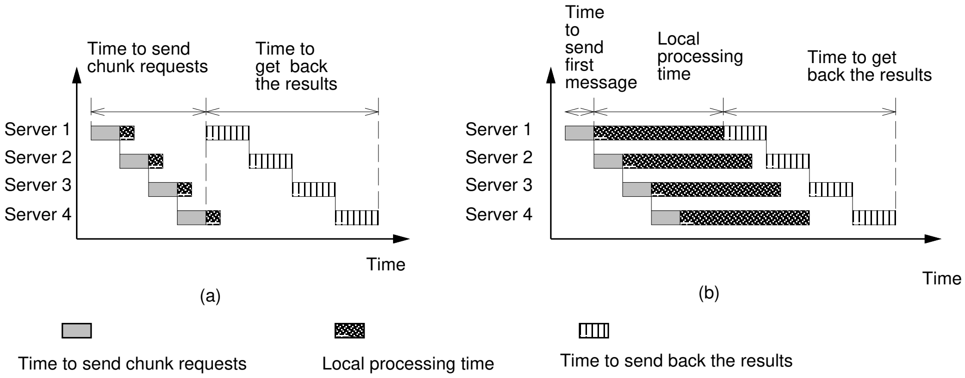


Fig. 6. *Timings when the dominating factor is (a) the communication and (b) the local processing.*

where K is the number of activated (ie., 'promising') servers and CC is the time to send a message (of size PS) from the master server to one of the servers;

$$T_{RESULT} = CC \times \frac{Q}{PS} \tag{3}$$

which is the total time to ship the results back to the master site (in packets of size PS). Thus, $RT(q_x, q_y)$ becomes:

$$\boxed{RT(q_x, q_y) = K \times CC + CC \times \frac{Q}{PS}, \quad \text{when} \quad (K-1) \times CC \geq Q \times \frac{T_{local-per-page}}{K}} \tag{4}$$

Domination of local processing time: Figure 6(b) illustrates this setting. Since the local processing time is long, the first server is still retrieving data, even after all K servers have been activated. Thus, the response time $RT(q_x, q_y)$ can be estimated as the time to send the first message, the time for the first server to finish its local processing, and the time to ship all the results back. Thus,

$$T_{COMMUNICATION} = CC \tag{5}$$

(for the first message only);

$$T_{LOCAL} = \frac{Q \times T_{local-per-page}}{K} \tag{6}$$

which is the local processing time for the first (as well as every other) server activated, and

$$T_{RESULT} = CC \times \frac{Q}{PS} \tag{7}$$

as before. Thus, we finally have:

$$\boxed{RT(q_x, q_y) = CC + \frac{Q \times T_{local-per-page}}{K} + \frac{Q \times CC}{PS}, \quad \text{when} \quad (K-1) \times CC \leq Q \times \frac{T_{local-per-page}}{K}} \tag{8}$$

where $\frac{Q \times T_{local-per-page}}{K}$ represents the local processing time in each node: Recall that it includes the disk access time (to fetch the page), plus the CPU time, to check whether the data rectangles indeed intersect the query rectangle.

It is important to highlight how overhead from other users using the network can be represented in the formulas above. As analyzed in [BMK88], the time to send a packet grows linearly with the number α of active nodes transmitting, that is:

$$CC = \alpha \times CC_{idle} \tag{9}$$

where α is the number of active (transmitting) nodes simultaneously with the node in question and CC_{idle} is the time to transmit a message on an idle network. Thus, α expresses the overhead due to other use of the network. Ideally, the time to send a small message on an Ethernet under very light load is on the average CC_{idle}=10ms.

4.2 *K* as a function of *C*

The next step is to find the connection between the number of activated servers (K) and the chunk size (C). For a given chunk size C, the average number K of servers that will be activated can be estimated by the formula:

$$K = (1 - (1 - \frac{1}{N})^{\frac{Q}{C}}) \times N \tag{10}$$

This formula assumes that the qualifying chunks are randomly and independently distributed among the servers (eg., through good hashing function). The justification of the formula is as follows:

- Q/C estimates the number of chunks that the query will retrieve.
- $1/N$ is the probability that a given server will contain a specific chunk of interest.
- $1 - 1/N$ is the probability that a given server will not contain a specific chunk.
- $(1 - 1/N)^{Q/C}$ is the probability that a given server will not contain any of the Q/C qualifying chunks.
- $1 - (1 - 1/N)^{Q/C}$ is the probability that a given server will contain at least one of the requested chunks.

4.3 Final optimization

Combining the results of the previous steps, we can express the response time as a function of C. From equations (4,8) (repeated here for convenience), we have:

$$RT(q_x, q_y) = K \times CC + CC \times \frac{Q}{PS}, \text{when} \quad (K-1) \times CC \geq Q \times \frac{T_{local-per-page}}{K} \quad (4)$$

$$RT(q_x, q_y) = CC + \frac{Q \times T_{local-per-page}}{K} + \frac{Q \times CC}{PS}, \quad \text{when} \quad (K-1) \times CC \leq Q \times \frac{T_{local-per-page}}{K} \quad (8)$$

Our goal is to find the optimal value K_{opt} of K (and, eventually, C_{opt}) that minimizes $RT(q_x, q_y)$. Notice that Eq. 4 increases with K (being linear on K), while Eq. 8 decreases with K (since K is in the denominator). Moreover, the combination of Eq.(4,8) forms a piece-wise continuous function, whose minimum is achieved at the point of discontinuity, that is, when

$$(K-1) \times CC = Q \times \frac{T_{local-per-page}}{K} \tag{11}$$

Solving for K, we have

$$K^2 - K - \frac{Q \times T_{local-per-page}}{CC} = 0 \tag{12}$$

and, after discarding the negative root (which has no physical meaning):

$$K_{opt} = \frac{1 + \sqrt{1 + 4 \times \frac{Q \times T_{local-per-page}}{CC}}}{2} \tag{13}$$

The above can be solved for C_{opt} using (10), giving

$$\boxed{C_{opt} = \frac{-Q}{N \times \log(1 - K_{opt}/N)} \quad \text{when} \quad K_{opt} \leq N} \tag{14}$$

In the formula above the computation for the chunk size holds as long as, $K_{opt} \leq N$. When the number of servers available (N) is less than K_{opt} then the optimal value of chunk size tends to become 0, from equation 14. Intuitively this is expected, since in this case the best we can hope is that all units participate in the query execution, and this is guaranteed by making the chunk size as small as possible. However, choosing the chunk size to be less than 1 page will increase the I/O cost: several pages will be retrieved in order to access only a tiny portion of their contents. This observation is confirmed in our experiments (see figure 7). Thus:

$$\boxed{C_{opt} = 1 \ \text{ page } \quad \text{when} \quad K_{opt} \geq N} \tag{15}$$

Summarizing, the goal of this section was to provide a formula for the optimal value of the chunk size C_{opt}; this is achieved with Eqs. (14, 15), where K_{opt} is given by Eq. 13. Notice that Eqs. (13,14,15) hold for *any* dimensionality of the address space.

5 Experimental Results

In this section we present experimental results with the proposed architecture. All the experiments were conducted on a set of rectangles, representing the MBRs (minimum bounding rectangles) of road-segments from the Montgomery County, MD. The data set consisted of 39,717 line segments; the address space was normalized to the unit square. In all cases we built a Hilbert-packed R-tree and declustered it as explained in subsection 3.2. The above R-tree package stored the tree in main-memory; thus, we had to simulate each disk access with a 10msec delay. A disk page was taken to be 1Kbyte. The CPU time to process a page was timed and found two orders of magnitude smaller than the disk access time; thus, we ignored it in our formulas ($T_{local-per-page}$= disk-access-time = 10msec).

The implementation of the communication software was done in the 'C' language under UNIX, using TCP sockets. For the experiments, we used several Sun SPARC 4/50 (IPX) workstations, with 16MB of main memory and sparc processors at 40MHz, connected via Ethernet at 10Mbits/sec. To avoid variations in the message traffic due to other users, we ran the experiments at night, where the load was light and the communication cost CC per message was fairly stable.

The queries were squares ($q_x = q_y$) of varying side q; they were randomly distributed in the address space. For each query side q, we report the averages over 100 queries.

We performed the following 3 sets of experiments:

– In the first set we illustrate the accuracy of the formula for the response time.
– In the second set, we experimented with various chunk sizes to show that indeed our choice of chunk size minimizes the response time.
– The third set of graphs uses the formulas to plot analytical results in 'what-if' scenarios.

Each set of experiments is discussed in the corresponding subsection next.

5.1 Accuracy of the formula

Our very first step was to estimate the communication cost CC (ie., startup costs and transmission of a message) in our environment. Since our network was not isolated, random overhead was imposed to our system, from other users of the network.

We performed several measurements, to obtain estimates at different times during the night, when the network traffic was low. All our experiments were also done at night. To obtain one measurement, we set up the communication interface between two processes and exchanged 100 messages of size PS=1 page, at random times. The averages times are presented in table 2. The estimate of CC=30ms was very consistent and is the one we use from now on.

Measurement	CC (msec)
Measurement 1	32.502194
Measurement 2	25.664210
Measurement 3	29.409790
Measurement 4	22.373228
Measurement 5	35.966356
Measurement 6	33.273418
Average:	29.86486

Table 2. *Experiments to measure the communication cost CC for a single message*

$CC/T_{local-per-page}$=3, N=3	Response Time (msec)		
Query Side	*Experimental*	*Theoretical*	*error %*
0.1	232	192	17
0.2	806	569	29
0.3	1577	1230	22

Table 3. *Theoretical and Experimental values for response time (in ms) for N=3 servers*

Given an accurate estimate for CC, we can use it in Eq's (4) and (8) to make predictions about the response time. Tables 3, 4 and 5 compare theoretical

$CC/T_{local-per-page}$=3, N=5	Response Time (msec)		
Query Side	*Experimental*	*Theoretical*	*error %*
0.1	228	208	9
0.2	786	625	20
0.3	1461	1229	16

Table 4. *Theoretical and experimental values for response time, for N=5 servers*

$CC/T_{local-per-page}$=3, N=7	Response Time (msec)		
Query Side	*Experimental*	*Theoretical*	*error %*
0.1	290	288	1
0.2	730	672	20
0.3	1289	1289	1

Table 5. *Theoretical and Experimental values for response time (in ms) for N=7 servers*

and experimental values for the response time (in milli-seconds), for N = 3, 5, 7 servers, respectively, and for chunk size C=1 page. We also present the percentage of error of each experiment performed, rounded up to the nearest integer.

The main observation is that our formula for the response time is accurate within 29% or better for all the experiments we performed.

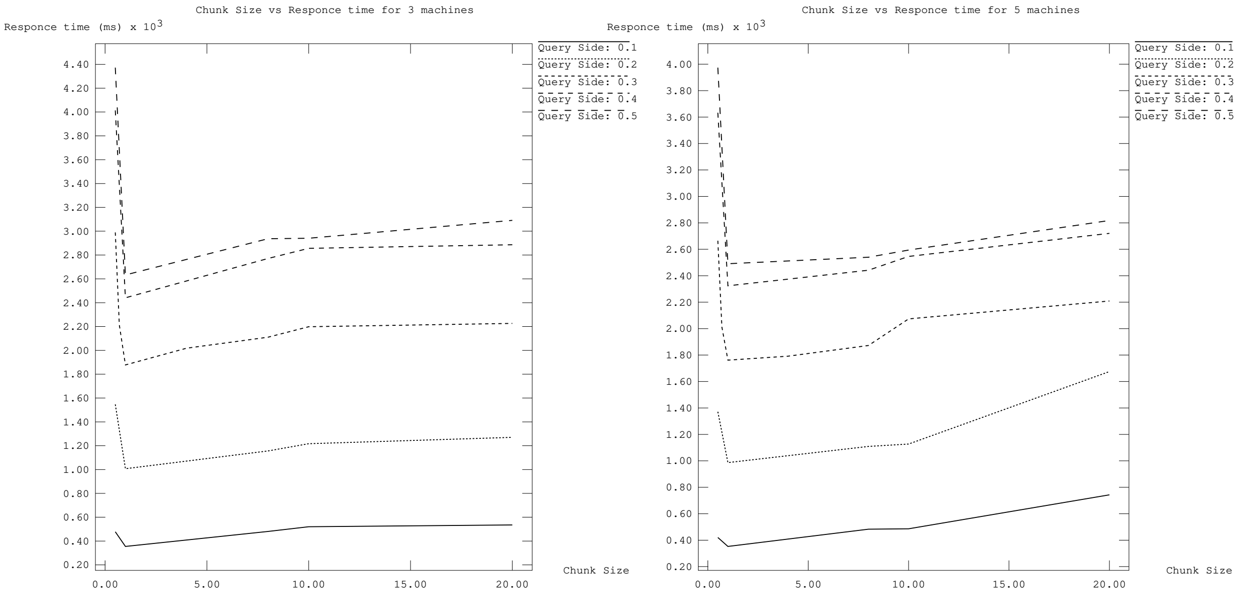
5.2 Optimal chunk-size selection

In this set of experiments we vary the chunk size, and we plot the response time for a given setting (number of servers N, query side q). The goal is to find the chunk size that minimizes the response time.

Figures 7(a,b) present our results for N=3 and 5 servers, respectively. The various lines correspond to different query sides q. In all these plots, the response time is minimized when the chunk size is one page (C_{opt}=1).

This result agrees with the outcome of our formula for the estimation of the optimal chunk size (Eq. 14, 15): Substituting the parameter values of the experiment of Figure 7a into (Eq. 14), for query sides $q_x = q_y$=0.1, we obtain C_{opt}=0.94; using the parameter values of figure 7b, we obtain C_{opt}=1.3. For the other values of query sides, the optimal value of processors K_{opt} returned by (Eq. 13) is greater than the number of servers ($K_{opt} > N$); thus, (Eq. 15) gives C_{opt} = 1 page, in agreement with the experiments.

Also, notice that $C < 1$ gives poor results, justifying (Eq. 15). The reasons have been explained in section 4: With a smaller-than-page chunk size, each server will have to retrieve several disk pages, only to discard most of their contents (since it will only need a small 'chunk' from each page); thus, the I/O subsystems will be unnecessarily overloaded, doing useless work.



(a) N=3 servers (b) N=5 servers

Fig. 7. Response time (in ms) vs chunk size (in pages) for (a) N=3 servers (b) N=5 servers. Notice the spike for $C < 1$ page.

5.3 Extrapolation for other scenarios

Having obtained confidence in our analysis, in this subsection we study its behavior through arithmetic examples. We choose some realistic settings, and do extrapolations and 'what-if' scenarios. Our goals are to study

1. large databases
2. networks of different speeds, in anticipation of fast networks (eg., fiber optics), as well as slower networks (eg., heavily loaded ones).

For the database size, we use two sizes:

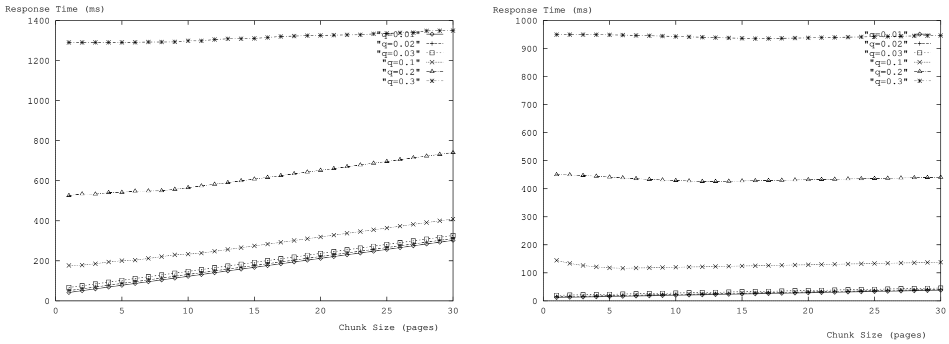
- a '*medium db*', with D=1Mb
- a '*large db*', with D=1GigaByte

From our analysis (eq. 13, 14), it follows that $CC/T_{local-per-page}$ affects the choice of chunk size. Intuitively, this is expected: When $CC/T_{local-per-page} \ll 1$, the local processing time will be the bottleneck; thus, increasing the use of the network and decreasing the local processing per server is the proper action. As a consequence, more servers should participate in query execution, and therefore a small chunk size is preferable. In the reverse situation, when $CC/T_{local-per-page} \gg 1$, the communication is the bottleneck. The network should be off-loaded, which can be achieved by engaging fewer servers; thus, a larger chunk size is favorable now.

We experiment with three settings:

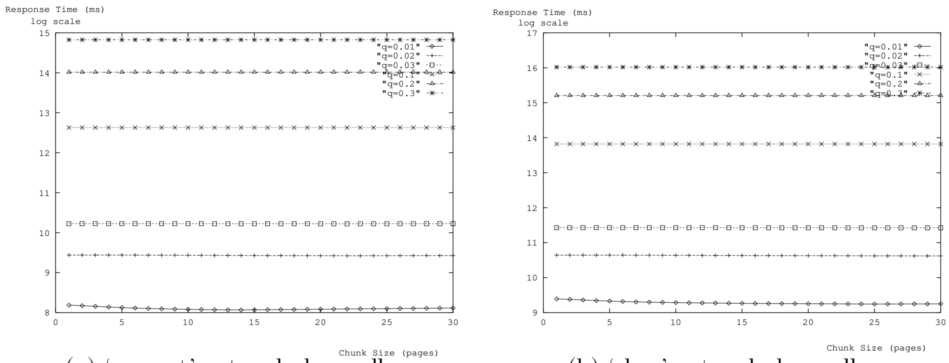
- '*current network*', where $CC/T_{local-per-page}$ = 30msec/10msec =3
- '*fast network*', where $CC/T_{local-per-page} = 0.1$
- '*slow network*', with $CC/T_{local-per-page} = 10$

We present the results in Figures (8,9, 10). For each combination of network speed and database size, we plot the response time as a function of the chunk size, for the following query sides q=q_x = q_y=0.01, 0.02, 0.03, 0.1, 0.2 and 0.3.



(a) 'current' network; medium db (b) 'slow' network; medium db

Fig. 8. Analytical results for a D=1Mb ('medium') database. Response time (in ms) vs chunk size (in pages) for: N = 5 servers and query sides q= 0.01, 0.02 ,0.03, 0.1,0.2,0.3 (bottom to top). (a) current network ($CC/T_{local-per-page}$ = 3) (b) slow network ($CC/T_{local-per-page}$ = 10)



(a) 'current' network; large db (b) 'slow' network; large db

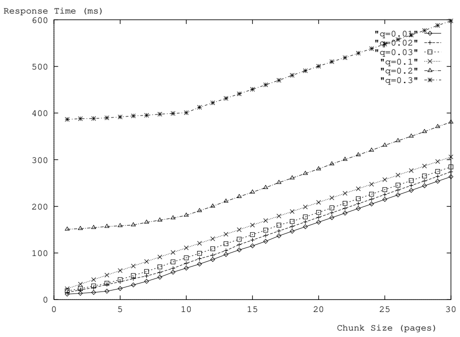
Fig. 9. Analytical results for a 'large' database (D=1Gb). Logarithm of response time (in ms) vs chunk size (in pages); N = 20 servers, query sides q=0.01, 0.02, 0.03, 0.1,0.2,0.3 (bottom to top). (a) current network ($CC/T_{local-per-page}$ = 3) (b) slow network ($CC/T_{local-per-page}$ = 10)

Figure 8 shows the results for a database of size 1MB, for the 'current' network ($CC/T_{local-per-page}$ = 3) and for a 'slow' network ($CC/T_{local-per-page}$ = 10). For the first case, the response time is minimized for C_{opt} = 1 page: all the curves increase with C, because messages are relatively cheap, and therefore it pays off to distribute the work among many servers (ie., small chunk size).

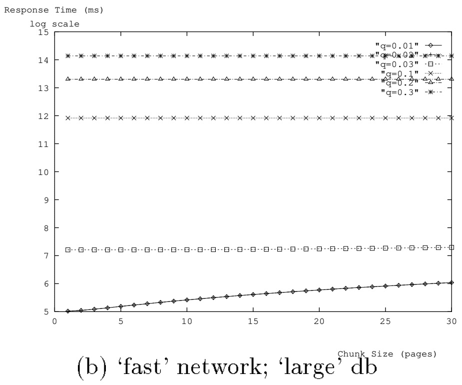
For the 'slow' network (Figure 8(b)), messages are more expensive and therefore larger chunk sizes are more favorable. However, even then, C = 1 gives a response time that is close to the minimum. The largest deviation from the minimum response time is 19.4%, which occurs for q=0.1 for the query sizes in the Figure. Notice that this performance penalty diminishes for larger queries (top two curves, q=0.2, 0.3). The reason is that, for larger queries, the response time becomes insensitive to the chunk size. The explanation is the following: Unless the chunk size has a huge value, a large query will activate all the N servers anyway, each of which will do roughly $1/N$ of the total local processing, which will be the bottleneck.

Notice also that, for small queries, the optimal chunk size is C=1. The reason is that small queries retrieve a small amount of data; a large chunk size will be an over-kill, because all the qualifying data will fit in a chunk, with room to spare (ie., wasted I/O time to fetch extraneous data from the disk)

Figures 9a,b show the same plots for a 1GB database, for the 'current' network ($CC/T_{local-per-page}$ = 3) and for the 'slow' network ($CC/T_{local-per-page}$ = 10), respectively. Notice that the vertical axis is in logarithmic scale, because a linear scale would visually collapse the curves of the small queries. The response time is rather insensitive to the chunk size: the plots are almost straight lines, indicating that C_{opt} = 1 is a good choice (as good as anything else). Only for very small queries (eg., q=0.01) the curves bend a little; even then, the choice of C=1 page gives response time that is within 13% of the minimum.



(a) 'fast' network; 'medium' db



(b) 'fast' network; 'large' db

Fig. 10. Analytical results for a 'fast' network ($CC/T_{local-per-page}$ = 0.1). Response time (in ms) vs chunk size(in pages); N=20 servers, query sides q=0.01, 0.02, 0.03, 0.1, 0.2, 0.3 (bottom to top). (a) 'medium' db (D=1MB) (b) 'large' db (D=1GB) - notice the logarithmic y-axis.

Figure 10 gives the plots for the 'fast' network ($CC/T_{local-per-page}$ = 0.1), plotting the response time RT as a function of the chunk size C for several values of the query sides q. If the network is fast, we want to maximize parallelism. This is achieved with the smallest allowable chunk size (C_{opt}=1). This is very pronounced for the 'medium' database (Figure 10(a)), where the response time increases with the chunk size, for all the query sizes. For the 'large' database (notice the logarithmic y-axis in Figure 10(b)), we see again that C=1 gives the minimum for small queries; for large queries, the response time is insensitive to the chunk size, as discussed before. Thus, the overall conclusion is that C = 1 is a 'safe' choice for a wide range of parameters: it will either give the optimal response time, or very close to it.

6 Conclusions

In this paper we have studied a method to decluster a spatial access method (and specifically an R-tree) on a shared-nothing multi-computer architecture. The nodes are connected through an off-the-shelf LAN. The major contributions of this work are

1. The derivation of formulas for the optimal chunk size C_{opt} ($\equiv$ striping unit), for a given query size (Eqs 14, 15).
2. The observation that $C_{opt} = 1$ page is a 'safe' choice, for a wide range of the problem parameters (query size, network/disk speed, database size etc). This choice either gives the minimum response time, or close to the (rather flat) minimum of the response time.

Additional, smaller, contributions include

1. the software architecture, with the R-tree at the master-server, and the leaf nodes ('chunks') at the rest of the servers. We made and justified several design decisions (eg., 'selective activation' vs. broadcasting; no pointers of the R-tree across servers, etc)
2. the derivation of simple, but accurate formulas that estimate the response time for a given query size (Eq 4,8).

We implemented the proposed method and we ran several experiments on a network of SUN workstations, operating on real data (road segments from the Montgomery county of Maryland, U.S.A). The experiments showed that

- the formulas (Eq 4,8) for the response time are accurate within 29% or better
- the formulas (Eq 14, 15) for the optimal chunk size agree very well with the experimental results

Having an accurate formula for the response time is a strong tool. One of its uses is for extrapolation and 'what-if' scenarios: We can obtain estimates about the performance of our system when the network is faster or slower (= loaded), when the disks (or, in general, the I/O units) are slower (e.g., juke-boxes of optical disks), or when the disks are faster (e.g., thanks to large buffer pools at each server, and high buffer-hit ratios), etc. All we have to do to simulate these cases is to adjust the appropriate values for the parameters (CC for the communication cost, $T_{local-per-page}$ for the I/O and CPU cost per page, etc.). A second use of the formula could be the analytical study of the throughput in case of multiple, concurrent queries: We believe that the effects of the additional queries can be modeled by appropriately 'inflating' the service time (CC) of the network, as well as of the disks and CPUs of the servers ($T_{local-per-page}$).

Using our formulas, we studied several 'what-if' scenarios. A useful, practical conclusion from this exercise is that using 1-page 'chunks' typically leads to optimal performance, or very close to it. This is especially true for fast networks, as well as for large queries. This conclusion is important from a practical point of view, because it provides a simple, intuitive rule for the optimal choice of the chunk size.

Future work includes the design of parallel R-tree algorithms for other types of queries, such as spatial joins [BKS93], and experimentation with other types of interconnects, such as ATM switches.

References

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. *Proc. of VLDB Conf.*, pages 487–499, September 1994.

[BB82] D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, 1982.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, September 1975.

[BFG+95] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. DB2 Parallel Edition. *IBM Systems Journal*, 32(2):292–322, 1995.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *Proc. of ACM SIGMOD*, pages 237–246, May 1993.

[BKSS90] 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.

[BMK88] David Boggs, Jeffrey C. Mogul, and Christopher A. Kent. Measured capacity of an ethernet: Myths and reality. *WRL Research Report 88/4*, 1988.

[CR93] Ling Tony Chen and Doron Rotem. Declustering objects for visualization. *Proc. VLDB Conf.*, August 1993. to appear.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DKL+94] David. J DeWitt, Navin Kabra, Jun Luo, Jignesh Patel, and Jie-Bing Yu. The client/server paradise. *Proceedings of the VLDB, 1994 Santiago, Chile*, September 1994.

[DS82] H.C. Du and J.S. Sobolewski. Disk allocation for cartesian product files on multiple disk systems. *ACM Trans. Database Systems (TODS)*, 7(1):82–101, March 1982.

[FB93] Christos Faloutsos and Pravin Bhagwat. Declustering using fractals. In *2nd Int. Conference on Parallel and Distributed Information Systems (PDIS)*, pages 18–25, San Diego, CA, January 1993.

[FBF+94] Christos Faloutsos, Ron Barber, Myron Flickner, J. Hafner, Wayne Niblack, Dragutin Petkovic, and William Equitz. Efficient and effective querying by image content. *J. of Intelligent Information Systems*, 3(3/4):231–262, July 1994.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. *Proc. ACM SIGACT-SIGMOD-SIGART PODS*, pages 4–13, May 1994. Also available as CS-TR-3198, UMIACS-TR-93-130.

[FLC86] M.F. Fang, R.C.T. Lee, and C.C. Chang. The idea of de-clustering and its applications. In *Proc. 12th International Conference on VLDB*, pages 181–188, Kyoto, Japan, August 1986.

[FM89] C. Faloutsos and D. Metaxas. Declustering using error correcting codes. *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 253–258, March 1989. Also available as UMIACS-TR-88-91 and CS-TR-2157.

[FR89] 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.

[Fre87] Michael Freeston. The bang file: a new kind of grid file. *Proc. of ACM SIGMOD*, pages 260–269, May 1987.

[Fre95] Michael Freeston. A general solution of the n-dimensional b-tree problem. *Proc. of ACM-SIGMOD*, pages 80–91, May 1995.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Comm. of ACM (CACM)*, 25(12):905–910, December 1982.

[GDQ92] Shahram Ghandeharizadeh, David J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. *SIGMOD Conf.*, June 1992.

[Gun86] O. Gunther. The cell tree: an index for geometric data. Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, December 1986.

[Gut84a] A. Guttman. *New Features for Relational Database Systems to Support CAD Applications.* PhD thesis, University of California, Berkeley, June 1984.

[Gut84b] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[HN83] 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.

[Jag90] H.V. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD Conf.*, pages 332–342, May 1990.

[KF92] 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.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. *Second Int. Conf. on Information and Knowledge Management (CIKM)*, November 1993.

[KF94] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: an improved r-tree using fractals. In *Proc. of VLDB Conference,*, pages 500–509, Santiago, Chile, September 1994.

[KP88] M.H. Kim and S. Pramanik. Optimal file distribution for partial match retrieval. *Proc. ACM SIGMOD Conf.*, pages 173–182, June 1988.

[KS91] Curtis P. Kolovson and Michael Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proc. ACM SIGMOD*, pages 138–147, May 1991.

[LS90] 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.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[OHM+84] 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.

[Ore86] J. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD*, pages 326–336, May 1986.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *Proc. ACM SIGMOD*, May 1985.

[Rob81] J.T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD*, pages 10–18, 1981.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[SRF87] 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.

[SS88] R. Stam and Richard Snodgrass. A bibliography on temporal databases. *IEEE Bulletin on Data Engineering*, 11(4), December 1988.

[SSH86] 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.

[Whi81] 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.

[WYD87] J.-H. Wang, T.-S. Yuen, and D.H.-C. Du. On multiple random accesses and physical data placement in dynamic files. *IEEE Trans. on Software Engineering*, SE-13(8):977–987, August 1987.

[WZS91] Gerhard Weikum, Peter Zabback, and Peter Scheuermann. Dynamic file allocation in disk arrays. *Proc. ACM SIGMOD*, pages 406–415, May 1991.

Table of Contents