Similarity Query Processing Using Disk Arrays*

Apostolos N. Papadopoulos
Department of Informatics
Aristotle University
Thessaloniki 54006, Greece
apapadop@athena.auth.gr

Yannis Manolopoulos
Department of Informatics
Aristotle University
Thessaloniki 54006, Greece
manolopo@athena.auth.gr

Abstract

Similarity queries are fundamental operations that are used extensively in many modern applications, whereas disk arrays are powerful storage media of increasing importance. The basic trade-off in similarity query processing in such a system is that increased parallelism leads to higher resource consumptions and low throughput, whereas low parallelism leads to higher response times. Here, we propose a technique which is based on a careful investigation of the currently available data in order to exploit parallelism up to a point, retaining low response times during query processing. The underlying access method is a variation of the R*-tree, which is distributed among the components of a disk array, whereas the system is simulated using event-driven simulation. The performance results conducted, demonstrate that the proposed approach outperforms by factors a previous branch-and-bound algorithm and a greedy algorithm which maximizes parallelism as much as possible. Moreover, the comparison of the proposed algorithm to a hypothetical (non-existing) optimal one (with respect to the number of disk accesses) shows that the former is on average two times slower than the latter.

1 Introduction

Modern applications are both data and computationally intensive and require the storage and manipulation of voluminous traditional (alphanumeric) and non-traditional data sets (images, text, geometric objects, time-series). Examples of such emerging application domains are: Geographical Information Systems (GIS), Multimedia Information Systems, Picture Archive and Communication Systems (PACS), CAD/CAM applications, Time-Series Analysis applications, Medical Information Systems, On-Line Analytical Processing (OLAP). These applications impose diverse requirements with respect to the information and the operations that need to be supported, and therefore from the database perspective, new techniques and tools need to be developed towards increased processing efficiency.

In many of the aforementioned applications, the data objects are represented as vectors in a high-dimensional feature space. For example, a 256-color image can be represented as a single vector using the 256 values of the color histogram, or a time sequence can be represented as a Fourier vector in a high-dimensional space [8]. The main advantages of the vector representation are:

- it allows efficient indexing of the data by means of multi-dimensional access methods, and
- it supports multi-dimensional queries using filtering, where a set of candidate objects is first determined using the vector representation, and after refinement only the objects that fulfill the query are reported in the output.

The similarity query is one of the most important operations in applications that are based on the vector model. Given a query vector (or query point) P_q, the similarity query asks for objects that are *similar* to P_q, where similarity is defined by means of a distance (dissimilarity) measure. Although efficient solutions have been reported for the similarity query problem in a sequential environment, the problem is relatively untouched for a parallel setting.

In this paper, we study similarity query processing on a RAID (Redundant Array of Inexpensive Disks) level 0 system. More specifically, we examine four algorithmic techniques. The first one is based on a previous branch-and-bound algorithm, whereas the second is based on a greedy philosophy. The third one is based on a careful investigation of the available data, trying to exploit parallelism to a sufficient degree and avoid fetching data that their usefulness probability is low. Comparing the latter technique with the aforementioned algorithms, it is observed that the proposed approach consistently shows the best performance with respect to speed-up, scale-up and query response time in a multiuser environment. Finally, a non-existing optimal algorithm is studied.

The focus is on dynamic environments, where insertions, deletions and updates can be intermixed with read-only operations. We do not consider complete reorganization of the database in order to provide an efficient data partitioning scheme [5]. Complete reorganization is prohibited by the huge volumes of data that modern database systems manipulate. The proposed similarity search algorithm supports all variants of the R-tree family as well as TV-trees [14],

*Work supported by the European Union's TMR program ("Chorochronos" project, contract number ERBFMRX-CT96-0056).

SS-trees [26], X-trees [3] and SR-trees [13], with some modifications. To the best of the authors' knowledge, this is the first work towards extensive performance evaluation of similarity search algorithms on a disk array architecture.

The rest of the article is organized as follows: In the next section we give the appropriate background to keep the paper self-contained. In Section 3 we describe related work and the motivation behind the proposed approach. Also, the similarity search algorithms are presented in detail. In Section 4 the experimental framework is explained, and performance results are given and interpreted. Finally, Section 5 concludes the paper.

2 Background

2.1 The R-tree family

The R-tree [10] is a hierarchical, height balanced data structure designed for use in secondary storage, and it is a generalization of the B^+-tree for multidimensional spaces. The structure handles objects by means of Minimum Bounding Rectangles (MBRs) which is the simplest conservative approximation of an object's shape. Each node of the tree corresponds to one disk page. Internal nodes contain entries of the form *(R,child-ptr)*, where *R* is the MBR that encloses all the MBRs of its descendants and *child-ptr* is the pointer to the specific child node. Leaf nodes contain entries of the form *(R,object-ptr)* where *R* is the MBR of the object and *object-ptr* is the pointer to the objects detailed description.

One of the major factors affecting the overall structure performance is the node split policy. In [10] three split policies have been reported, namely exponential, quadratic and linear. More sophisticated policies reducing the overlap of MBRs have been reported in [22] (the R^+-tree), in [1] (the R^*-tree) and in [12] (the Hilbert R-tree). The R^*-tree uses the concept of forced reinsertion of entries, in addition to a good split policy. Henceforth, we focus on the R^*-tree variant. The only modification applied to the structure is that in each MBR entry, there is an integer number denoting the number of objects that the corresponding branch contains.

2.2 Disk arrays and R^*-tree partitioning

RAID systems have been introduced in [18] as an inexpensive solution to the I/O bottleneck. Using more than one disk devices, leads to increased system throughput, since the workload is balanced among the participating disks and many operations can be processed in parallel. A typical layout of a disk array architecture is illustrated in Figure 1, where three disks (each one with its own controller) are attached to a single processor.

Several RAID levels have been reported in the literature [6] aiming at higher system reliability and data availability. Three of the RAID levels that are widely used in data intensive applications are: RAID level-0 (non-redundant striping), RAID level-1 (mirrored or shadowed disks), and RAID level-5 (block-interleaved distributed parity). For the rest of the paper, we focus on RAID level-0, assuming that the striping unit is a disk block. Introducing redundancy (e.g. my means of mirroring) and studying the behavior of similarity search in other RAID levels, is an issue of further research.

Given a disk array, one faces the problem of partitioning the data and the associated access information, in order to take advantage of the I/O parallelism. The way data

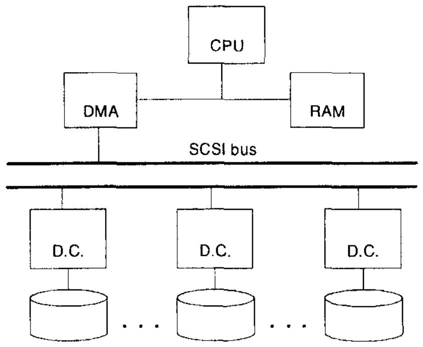


Figure 1: Example of a disk array architecture.

is partitioned reflects the performance of read/write operations. The declustering problem attracted many researchers and a lot of work has been performed towards taking advantage of the I/O parallelism, to support data intensive applications. Techniques for B^+-tree declustering have been reported in [21]. In [11] a disk array variant of the R-tree, the multiplexed (or parallel) R-tree has been proposed in order to process range queries efficiently. The parallel structure behaves just like an ordinary R-tree, and shows considerable improvements in response time and system throughput, since the organization supports intraquery and interquery I/O parallelism. In [27] the authors study effective declustering schemes for the grid file structure.

In a dynamic environment, an insertion of a new element may cause a page to split into two pieces[1]. A choice must be made here in order to place the newly created page to a disk. A plethora of heuristics have been reported in the literature, ranging from the simple round-robin and random assignment to more sophisticated ones. Among the best declustering techniques in the original R-tree, is the one proposed in [11], which is based on the *Proximity Index* (PI) concept between two hyper-rectangles. Upon a split, the MBR of the newly created node is compared with the MBRs of its father node. The new node is assigned to the disk which is "less proximal" with respect to the new MBR. In other words, the selected disk must contain sibling nodes that are far from the new node, avoiding fetching a large number of nodes from the same disk during query execution, and thus preventing I/O bottlenecks. After conducting a thorough experimental study, we have observed that PI shows consistently the best performance in similarity query processing over a parallel R^*-tree, in comparison to all known declustering heuristics: random assignment, data balance, area balance, round-robin, etc. Therefore, we adopt PI as the declustering method of choice.

2.3 Similarity queries

Similarity queries can be categorized in two different query types: the range query, and the k-nearest-neighbor (or k-NN) query. The following definitions explain:

Definition 1
Given a query point P_q, a distance ϵ and a dissimilarity measure $dist(a, b)$ between two objects a and b, the range

[1]Although techniques exist that handle 2-to-3 or generally m-to-(m+1) splits, we focus on the traditional 1-to-2 split. Generalizations are straightforward.

query asks for all objects x_j such that $dist(P_q, x_j) \leq \epsilon$. If the dissimilarity measure is the Euclidean distance, then the answer is composed of all objects that intersect (or are totally enclosed by) the hyper-sphere centered at P_q and having radius ϵ. □

Definition 2
Given a query point P_q and an integer number k, the k-NN query asks for the k closest neighbors to P_q, among all the objects comprising the data set. Again, a dissimilarity measure must be adopted in order to define the proximity between two objects. □

We assume that the dissimilarity measure is the Euclidean distance in the n-d space. With respect to range queries, a lot of research has been performed, since this is one of the most often posed queries in multidimensional applications. In [16] different query models are presented along with an analysis of queries posed uniformly on the address space. In [2, 7], analysis of range queries on R-trees is illustrated using the concept of fractal dimension and in [24] an analysis based on the data set density is presented. In these works, analytical formulae have been reported in order to estimate the performance of a range query. With respect to nearest neighbor queries an algorithm to answer such queries in k-d trees has been reported in [9]. In [26] various algorithms and data structures have been reported to answer efficiently nearest neighbor queries. Finally, some analytical results on nearest neighbor queries can be found in [4, 17].

When the distance ϵ is known in advance (i.e. provided by the user), then answering a similarity query (range query) is quite straightforward in either the sequential or the parallel case. However, the distance ϵ is not always known, and in some cases is difficult to estimate from the user's viewpoint. Instead, the user provides an object O_x and requires the most similar objects to O_x (e.g. the k most similar objects). Evidently, a nearest neighbor query can always be transformed to a series of range queries by using different ϵ values. Following this approach, we may face unnecessary resource consumption. In one extreme, the selected values for ϵ may be too small, leading to inadequate number of answers (less than k). On the other extreme, the ϵ values may be too large, accessing a large number of objects (much more than k). The problem is even harder when I/O parallelism must be exploited, and this is exactly the focus of this paper.

3 Similarity Search Algorithms

An efficient algorithm for similarity search on disk arrays must preserve some fundamental properties:

- parallelism must be exploited as much as possible,
- the number of retrieved nodes must be minimized, and
- the response time of user queries should be reduced as much as possible.

In order to exploit I/O parallelism in similarity search, we have to access several nodes (residing in different disks) in parallel. In the general case, this implies that some of the accessed nodes eventually will be proved irrelevant with respect to the final answer, and therefore they should have never been accessed. Compare the above scheme with a range query. A range query is described by a well-defined region of arbitrary shape (usually hyper-rectangular or hyper-spherical) and all objects intersecting this region are requested. After a node is accessed, we are able to determine which of its children need to be visited by inspecting the corresponding MBRs that are located in the node. Then, the disks that host the relevant children nodes can be activated in parallel. Evidently, the visiting sequence of the relevant nodes is not important, since any such sequence leads to the same answer (assuming only read-only operations). On the other hand, in similarity search, the visiting order is the most important parameter in performance efficiency, since it is responsible for the further pruning of irrelevant nodes. Therefore, we come up with a problem definition, which has as follows:

Problem Definition:
Given a query point P_q in n-d space and an integer number k, **determine** an efficient search of the parallel R*-tree, **in order to** report the k nearest neighbors of P_q, **trying to** (i) maximize parallelism, (ii) access as few nodes as possible, (iii) reduce response time. □

From the above discussion we observe that two fundamental sub-problems must be solved:

- to determine an effective way of pruning irrelevant nodes in every tree level, and
- to use a clever criterion in order to decide which nodes and when are going to be accessed in parallel.

In the remaining of this section, four algorithms are examined in detail, that solve the similarity search problem on a disk array architecture.

3.1 The branch-and-bound algorithm

In this subsection we review the branch-and-bound algorithm reported in [19], for answering nearest neighbor queries in R-trees. It is a modification of the algorithm reported in [9] for k-d trees. In order to find the nearest neighbor of a query point, the algorithm starts from the root of the R-tree and proceeds towards the leaf level. The key idea of the algorithm is that many tree branches can be discarded according to some basic rules. Two basic distances are defined in n-d space, between a point P_q with coordinates $(p_1, p_2, ..., p_n)$ and a rectangle R with (bottom-left and top-right) corners having coordinates $(s_1, s_2, ..., s_n)$ and $(t_1, t_2, ..., t_n)$ respectively. These distances correspond to an optimistic and a pessimistic approach for the nearest object respectively.

Definition 3
The distance $D_{min}(P_q, R)$ between a point P_q and a rectangle R, is defined as follows:

$$D_{min}(P_q, R) = \sqrt{\sum_{j=1}^{n} |p_j - r_j|^2}$$

where:

$$r_j = \begin{cases} s_j, & p_j < s_j \\ t_j, & p_j > t_j \\ p_j, & \text{otherwise} \end{cases}$$

□

Definition 4
The distance $D_{mm}(P_q, R)$ between a point P_q and a rectangle R, is defined as follows:

$$D_{mm}(P_q,R) = \sqrt{\min_{1\le k\le n}\left(|p_k - rm_k|^2 + \sum_{1\le j\le n, j\ne k} |p_j - rM_j|^2\right)}$$

where:

$$rm_k = \begin{cases} s_k, & p_k \le \frac{s_k+t_k}{2} \\ t_k, & \text{otherwise} \end{cases}$$

$$rM_j = \begin{cases} s_j, & p_j \ge \frac{s_j+t_j}{2} \\ t_j, & \text{otherwise} \end{cases}$$

□

Evidently, D_{min} is the optimistic metric, since it is the minimum possible distance that the nearest neighbor of P_q can reside in the corresponding page. On the other hand, D_{mm} is the pessimistic metric, since it guarantees that the nearest neighbor of P_q lies in a distance $\le D_{mm}$. These two metrics are used in three basic rules which are applied for pruning the search in the R-tree:

1. **If** an MBR R has $D_{min}(P_q, R)$ greater than the D_{mm} (P_q, R') of another MBR R', **then** it is discarded because it cannot enclose the nearest neighbor of P_q.
2. **If** an actual distance d from P_q to a given object, is greater than the $D_{mm}(P_q, R)$ of P_q to an MBR R, **then** d is replaced by $D_{mm}(P_q, R)$ because R contains an object which is closer to P_q.
3. **If** d is the current minimum distance, **then** all MBRs R_j with $D_{min}(P_q, R_j) > d$ are discarded, because they cannot enclose the nearest neighbor of P_q.

Upon visiting an internal node of the tree, Rules 1 and 2 are used in order to discard irrelevant branches. Then, a branch is selected according to a priority order. Roussopoulos et al. suggest that when the overlap is small, the D_{min} order should be used since it would discard more candidate branches. This is also verified in the experimental results of their work. Therefore, the branch corresponding to the minimum D_{min} among all node entries is chosen. Upon returning from the processing of the subtree, Rule 3 is applied in order to discard other candidates (if there are any). In order to process general k-NN queries, an ordered sequence of the current k most promising answers has to be maintained, and the pruning of the MBRs has to be performed with respect to the furthest distance. Thus, an MBR is discarded if its D_{min} from the query point is greater than the actual distance from the query point to its k-th nearest neighbor. Henceforth, this algorithm will be referred to as **Branch and Bound Similarity Search (BBSS)**.

3.2 Full-parallel similarity search

Observing how the sequential algorithm works, we see that a careful refinement of the candidate nodes is performed, trying to avoid node accesses that will not contribute to the final answer. We continue with an important definition regarding the maximum possible distance D_{max} between a point and a hyper-rectangle.

Definition 5
The distance D_{max} between a query point P_q and an MBR R, is the distance from P_q to the furthest vertex of R and equals:

$$D_{max}(P_q,R) = \sqrt{\sum_{j=1}^{n} |p_j - r_j|^2}$$

where:

$$r_j = \begin{cases} t_j, & p_j \le \frac{s_j+t_j}{2} \\ s_j, & \text{otherwise} \end{cases}$$

□

To distinguish between the three distances (D_{min}, D_{mm} and D_{max}) an example is illustrated in Figure 2, showing a point, two rectangles and the corresponding distances.

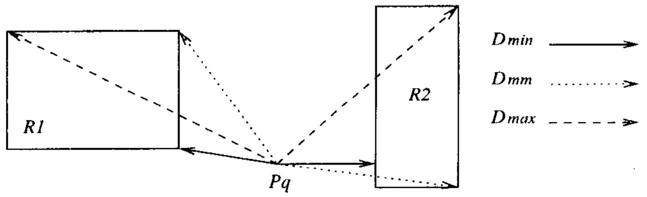


Figure 2: D_{min}, D_{mm} and D_{max} between a point P and two rectangles R_1 and R_2.

The first node that is inspected by the algorithm is, evidently, the root of the parallel R*-tree. Note that at this stage (and until the first k objects are visited) there is no available information concerning the upper bound for the distance to the k-th nearest neighbor. Let in the current node N reside m MBRs, pointing to m children nodes. The question is which of the m branches can be discarded (if any), and how can we obtain the needed information to perform the pruning. In order to proceed we need to calculate a threshold distance. The following lemma explains:

Lemma 1
Assume we have m MBRs $R_1, ..., R_m$ where MBR R_j contains $O(R_j)$ objects. Given a query point P_q, the k nearest neighbors with respect to P_q are requested. Assume further that all m MBRs are sorted in increasing order with respect to the D_{max} distance from the query point P_q. Then, all k best answers are contained in the circle (sphere, hypersphere) with center P_q and radius $r = D_{max}(P_q, R_x)$ where x is determined from the following inequality:

$$\sum_{j=1}^{x-1} O(R_j) \le k \le \sum_{j=1}^{x} O(R_j) \qquad (1)$$

Proof (omitted) □

Using the above lemma we can always determine a threshold distance D_{th}. Having D_{th}, some of the m entries may be rejected immediately. An example is illustrated in Figure 3. The threshold distance in the example equals: $D_{th} = D_{max}(P_q, R_1)$. As we observe, MBR R_5 is rejected since the dotted circle is guaranteed to contain all the relevant answers, and R_5 does not intersect the circle. However, there are some MBRs like R_2, R_3 and R_5, which are intersected by the circle. Therefore, the set of candidate MBRs is composed of R_1, R_2, R_3 and R_4. The problem arising is which of these candidates will be searched in the next step and which will be saved for future reference.

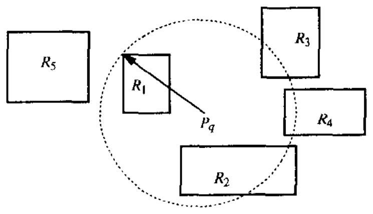


Figure 3: Illustration of pruning and candidate selection.

Assume that m_1 out of m entries have been pruned (like R_5 in the example). Now, we have m_2=m-m_1 entries that need further inspection. The most straightforward approach is to assume that all these m_2 entries will eventually contribute to the final answer and therefore have to be searched. This technique is the main idea of the **F**ull **P**arallel **S**imilarity **S**earch algorithm (**FPSS**), which is very optimistic with respect to the usefulness of a node.

3.3 Candidate reduction search

We propose to apply a heuristic, in order to (possibly) reduce the number of candidate MBRs. By observing Figure 3, it seems that MBR R_2 has better chances to contain relevant objects than MBRs R_3 and R_4. Therefore, candidates R_3 and R_4 are saved for future reference, whereas R_1 and R_2 will be searched. The criterion for candidate reduction has as follows:

Candidate Reduction Criterion
Given a query point P_q, a threshold distance D_{th} and a set of MBRs $\mathcal{R} = \{R_1, ..., R_m\}$ then for an MBR R_x:

(i) if $D_{th} < D_{min}(P_q, R_x)$, then R_x is rejected.

(ii) if $D_{th} \geq D_{mm}(P_q, R_x)$, then R_x is set active.

(iii) if $D_{th} \geq D_{min}(P_q, R_x)$ and $D_{th} < D_{mm}(P_q, R_x)$, then R_x is saved for possible future reference. □

The activation list contains the addresses to all pages that are going to be requested from the disks in the current step. Each entry contains a pointer to its son. This means that we can fetch the pages pointed by R_1 and R_2 from the disk array (if these nodes reside on different disks this can be done in parallel). As soon as the first k objects are retrieved, we have a more precise knowledge regarding the distance D_k from the query point P_q to its k-th nearest neighbor. Every time the distance D_k is updated due to access of data objects, the structure maintaining the remaining candidate MBRs is searched and new MBRs become active. The algorithm that is obtained from the application of the heuristic is called **C**andidate **R**eduction **S**imilarity **S**earch (**CRSS**).

Evidently, in order for the **CRSS** method to work, some auxiliary data structures need to be maintained. Based on the previous discussion we can identify three auxiliary structures:

- a structure to maintain the pointers to the nodes that are going to be fetched in the next step (activation structure),
- a structure to hold the newly fetched nodes in order to process them further (fetch structure), and
- a structure to store the candidate MBRs that have neither been searched nor have they been rejected yet (candidate structure).

The first two structures can be simple arrays or linked lists and no special treatment is required. As soon as the currently relevant pointers (page addresses) have been collected in the activation structure, requests are sent to the corresponding disks in order to access the required pages. When the disks have processed the requests, the pages are collected in the fetch structure where further processing (pruning, candidate reduction, etc.) is performed. The auxiliary structure to store the candidate MBRs must however be a stack, with its entries organized in a convenient way that helps processing. The cooperation of all three structures is explained in the following illustrative example.

Example
An R*-tree is illustrated in Figure 4, where all tree nodes are assumed to hold three occupied entries. Nodes are numbered from N_1 to N_{13}. Let us trace the execution **CRSS** algorithm for a simple query requiring the k = 4 nearest neighbors of a query point. The algorithm begins with the root (node $N1$) where the MBRs R_1, R_2 and R_3 reside. Assume that R_1 and R_2 qualify for immediate activation (according to the candidate reduction criterion), whereas R_3 is considered as a possible candidate MBR. No MBR has been rejected at this point. The pointers to N_2 and N_3 are maintained in the activation structure and MBR R_3 is pushed into the candidate stack. Note that the candidates are pushed in decreasing order with respect to the D_{min} from the query point. After the stack is updated, we are ready to fetch N_2 and N_3 from the disks. Assume that these two nodes reside in different disks and therefore the requests can be serviced in parallel. The situation is depicted in Figure 5(a).

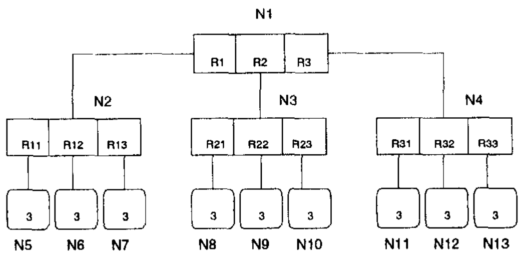


Figure 4: Example of an R*-tree with 13 nodes and 3 entries per node.

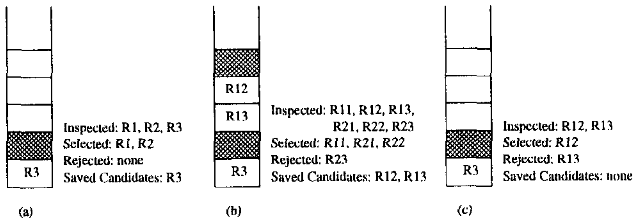


Figure 5: Illustration of the first three stages of the **CRSS** algorithm. Shaded boxes indicate guards.

In the next step, entries R_{11} through R_{23} are inspected. Assume that we have concluded that entry R_{23} is rejected, R_{11}, R_{21} and R_{22} will be activated, and finally R_{12} and R_{13}

will be saved in the stack. The situation is illustrated in Figure 5(b).

The following stage involves the access of the data pages N_5, N_8 and N_9. This is the first time during the execution of the algorithm that real data objects contribute to the formulation of the upper bound to the k-th best distance (where $k = 4$). Therefore, the best four out of nine objects, contained in the three data pages, are selected and the distance D_{th} is updated accordingly. In the sequel, we pop from the stack the first candidate run that is composed of the MBRs R_{12} and R_{13}. After comparing $D_{min}(P_q, R_{12})$ and $D_{min}(P_q, R_{13})$ with D_{th}, we conclude that R_{12} is intersected by the query sphere, whereas R_{13} can be safely rejected. The current situation is depicted in Figure 5(c).

In the next step, node N_6 is accessed, the distance D_{th} is updated and the next candidate run is popped from the stack. This run contains only R_3. Comparing $D_{min}(P_q, R_3)$ with D_{th}, we find that there is no intersection with the query sphere and therefore R_3 is rejected from further consideration. Now the algorithm has been terminated, the best k matches have been determined and $D_{th}=D_k$. □

Let us explain the use of the stack, and the reason why it is the appropriate structure in our case. As we descent the tree from root to leaves, the granularity of MBRs increases, since the empty space is reduced. Therefore, the information obtained from the MBRs near the leaf level is more precise than the information obtained from MBRs near the root. It is not wise to start the inspection of a new branch in a higher level of the R*-tree, if there are still candidate branches to be inspected in a lower level. The structure that captures this concept is the stack. Therefore, candidate MBRs that belong to a higher level are pushed in the stack before candidates of a lower level. Moreover, organizing the candidates in the stack by means of candidate runs, helps in pruning. The candidates in each run are pushed in decreasing order with respect to the D_{min} distance from the query point. When a candidate run is inspected and a candidate is found that does not intersect any more the query sphere, we know that all the remaining candidates in the current run should be rejected from further consideration. A guard entry is used to separate two different candidate runs. This technique saves computational power during candidate elimination and leads to more efficient processing.

In Figure 6 the **CRSS** algorithm is sketched. There are four basic operating modes that the algorithm can be at some given time:

- The algorithm operates in **ADAPTIVE** mode from the time the root is examined until the leaf-level is reached for the first time. During this period, the upper bound of the threshold distance D_{th} is adapted from one tree level to the next.
- Every time the leaf-level is reached, the algorithm goes into **UPDATE** mode. This means that the array holding the current best k distances is (possibly) updated, since more data objects have been accessed.
- In any other case, the algorithm operates in **NORMAL** mode. This mode includes the cases where the algorithm operates in an intermediate level of the tree, but after the first time the leaf-level has been reached.
- Finally, the **TERMINATE** mode signals that there are no more candidate nodes to be searched, and therefore the k best distances have been determined.

```
Input: P /* query point */
       k /* number of nearest neighbors */
       T /* the parallel R*-tree */
Output: the k nearest-neighbors of P
BEGIN
  0. Initialize: D_threshold <- infinite,
                 AL <- empty, CS <- empty, FL <- empty
  1. Read Root(T);
  2. IF (leaf-level reached) mode <- update;
  3. Process (FL);
  4. if (mode IS NOT terminate)
       send requests to disks; Update(FL); GOTO 2;
     else STOP;
END

/* Routine to process a set of new MBRs */
Process (FL)
BEGIN
  if (mode IS adaptive)
  {
    Find new value for D_threshold; Apply candidate reduction;
    Formulate new candidate run; Push run in CS;
    Update(AL);
  }
  else
  if (mode IS normal)
  {
    Eliminate non-relevant MBRs;
    if (FL IS empty)
      Get_Candidate_Run(CS);
    Update(AL);
  }
  else
  if (mode IS update)
  {
    Calculate new set of nearest-neighbors;
    Get_Candidate_Run(CS);
    Update(AL);
  }
END

/* Routine to obtain the next candidate run */
Get_Candidate_Run(CS)
BEGIN
  if (CS IS empty)
    mode <- terminate;
  else
  {
    Pop next candidate run from CS;
    Eliminate non-relevant MBRs;
    Apply candidate reduction;
    mode <- normal;
  }
  return;
END
```

Figure 6: The most important code fragments of the **CRSS** algorithm.

It is observed that **FPSS** and **BBSS** are special cases of the **CRSS** algorithm. **FPSS** does not use a candidate stack and activates all MBRs that intersect the current query sphere, maximizing intra-query parallelism, whereas **BBSS** activates the MBRs one at a time, limiting the degree of intra-query parallelism. Let us elaborate more in code fragments **A** and **B** shown in Figure 6. In **A**, the candidate reduction criterion is applied. Among the fetched MBRs, some of them are discarded immediately, and some will be saved in the candidate stack for future reference. The restriction applied here is that the number of activated MBRs should be $\geq l$ and $\leq u$, where l is the number of MBRs which guarantee the containment of at least k points in the activated MBRs, and u equals the number of disks in the system ($NumOfDisks$). This restriction is used in order to bound the number of fetched nodes in the next step. A similar policy is used in the **B** code fragment, where the *candidate reduction criterion* is again applied. When there is a need to pop the next candidate run from the stack, we never allow the activation of more than u=*NumOfDisks* elements. Using this technique, there is a balance between parallelism exploitation and similarity search refinement. In order for the u MBRs to reside in different disks, the declustering scheme must be as close to optimal as possible.

Theorem 1
Given a query point P_q and a number k, algorithm **CRSS** reports the best k nearest neighbors of P_q.

Proof (sketch)
Basically, the algorithm can be considered as a repetition of three fundamental operations: (i) candidate elimination, (ii) generation of new candidates and (iii) retrieval of new data. Since the threshold distance D_{th} guarantees the inclusion of the best answers (Lemma 1) and only irrelevant MBRs are eliminated (candidate reduction criterion), it is impossible that a best match will be missed. Moreover, the algorithm reports exactly k answers, unless the total number of objects in the database is less than k, in which case reports all the objects. □

3.4 Optimal similarity search

Designing an algorithm for similarity search we need a criterion in order to characterize the algorithm as efficient or inefficient. The ideal would be to design an optimal algorithm, guaranteeing the best possible performance. In the context of similarity search, two levels of optimality are identified: **weak** and **strict** which are defined as follows.

Definition 6
A similarity search algorithm is called **weak-optimal**, if for every k-NN query the only nodes that are accessed are those that are intersected by the sphere having center the query point and radius the distance to the k-th nearest neighbor. □

Definition 7
A similarity search algorithm is called **strict-optimal**, if it is weak-optimal, and in addition for every k-NN query the only objects that are inspected lie in the sphere with center the query point and radius the distance to the k-th nearest neighbor. □

It is evident that in order for an algorithm to be either weak-optimal or strict-optimal, the distance from the query point to the k-th nearest neighbor must be known in advance. Moreover, in strict optimality the algorithm must also process only the objects that are enclosed by the sphere with center P_q and radius D_k. This implies a special organization of the data objects and it is almost impossible to achieve strict optimality in similarity search. Also, although weak optimality still imposes a strong assumption, we assume the existence of a hypothetical algorithm **W**eak **OPT**imal **S**imilarity **S**earch (**WOPTSS**), and we include it in our experimental evaluation. The performance of **WOPTSS** method serves as a lower bound for the performance of any similarity search algorithm. The following theorem illustrates that the three aforementioned algorithms are not optimal:

Theorem 2
The similarity search algorithms **BBSS**, **FPSS** *and* **CRSS** *operating over an* R*-tree, *are neither strict-optimal nor weak-optimal.*

Proof (sketch)
We can find a counterexample for all algorithms with respect to certain query points and R*-tree layouts, showing that *neither the minimum number of nodes are visited, nor* the minimum number of objects are inspected. □

The number of accessed nodes is a good metric for the performance of a similarity search algorithm in the sequential case. However, in the parallel case the situation is more complicated. When processing similarity queries on a disk array, one wants high parallelism exploitation in addition to small number of accesses. A more concrete measure of efficiency in this case is the mean response time of a similarity query in a multi-user environment. Evidently, one can use the response time of a single query but this does not reflect reality. To see why, assume that an algorithm A accesses half of the pages than algorithm B. On a disk array, the I/O subsystem is capable of servicing several requests in parallel. Therefore, we may notice no difference in the response time of a single query for both algorithms, whereas in a multi-user environment the performance of algorithm B is more likely to degrade rapidly in comparison to the performance of A, due to heavy workloads.

4 Performance Evaluation

4.1 Preliminaries

The algorithms **BBSS**, **FPSS**, **CRSS** and **WOPTSS** are implemented on top of a parallel R*-tree structure which is distributed among the components of a disk array. The behavior of the system is studied using event-driven simulation. The algorithms and the simulator have been coded in C/C++ under UNIX, and the experiments have been performed on a SUN Sparcstation4 running Solaris 2.4. The datasets used are illustrated in Appendix I. An R*-tree for a particular data set is constructed incrementally (i.e. by inserting the objects one-by-one). The disks are assumed to communicate with the processor by means of a common I/O bus. The network queue model of the system that is used for the simulation is presented in Figure 7. Each disk has

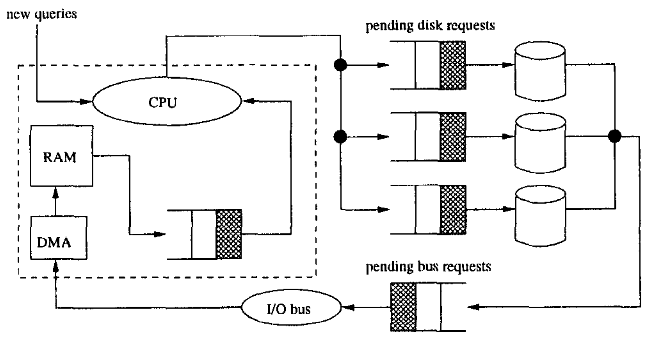


Figure 7: The simulation model for the system under consideration.

its own queue where pending requests reside. The service policy for each queue in the system is FCFS (First-Come

First-Served). The bus is also modeled as a queue, with constant service time (the time it takes to transmit a page from the disk controller through the I/O bus). Queues are also present in the processor in order to handle pending requests. However, we assume that when a new query request arrives, it enters the system immediately without waiting.

Query arrivals follow a Poisson distribution with mean λ arrivals per second. Therefore, the query interarrival time interval is a random variable following an exponential distribution. The service time for the bus is constant, whereas the service time of a disk access is calculated taking into consideration the most important disk characteristics (seek time, rotational latency, transfer time and controller overhead). Moreover, we do not assume that the disks are synchronized, and therefore each disk can move its heads independently from the others. The parameters that are used in the experimental evaluation are presented in Table 1.

Parameter	Description	Assigned Value
S_{node}	Node capacity	4 KB
n	Space dimensionality	2 to 30
N	Number of objects	>10,000
k	Number of nearest neighbors	1 to 700
d	Number of disks	1 to 40
λ	Query arrivals per second	$\leq$ 30
B	I/O bus bandwidth	20 MB/sec
CPU_{speed}	CPU execution speed	100 MIPS
$Q_{startup}$	Query startup time	0.001 sec

Table 1: Description of query processing parameters.

In order to model each disk device, the two-phase non-linear model is used which is described in detail in [15, 20]. If d_{seek} denotes the seek distance that the head needs to travel, the seek time T_{seek} as a function of d_{seek} is expressed by the following equation:

$$T_{seek} = \begin{cases} 0, & d_{seek} = 0 \quad \text{(no seek)} \\ c_1 + c_2 \cdot \sqrt{d_{seek}}, & 0 < d_{seek} \leq sdt \quad \text{(short seek)} \\ c_3 + c_4 \cdot d_{seek}, & d_{seek} > sdt \quad \text{(long seek)} \end{cases}$$

where c_1, c_2, c_3 and c_4 are constants (in msec) specific to the disk drive used and sdt is a seek distance threshold, which differentiates the acceleration phase and the steady-speed phase of the disk arm movement. The characteristics of the disk drive that is used in the conducted simulation experiments are illustrated in Table 2.

Parameter	Description	Assigned Value
Cyl	Number of cylinders	1449
T_{rev}	Disk revolution time	0.0149 sec
R_{trans}	Disk transfer rate	5 MB/sec
T_{seek}	Disk seek time	variable
O_{ctrl}	Disk controller overhead	0.0011 sec
c_1	Short-seek constant 1	3.45 msec
c_2	Short-seek constant 2	0.597 msec
c_3	Long-seek constant 1	10.8 msec
c_4	Long-seek constant 2	0.012 msec
sdt	Seek distance threshold	616

Table 2: Description of disk characteristics (model HP-C220A) [20].

During R*-tree creation, each newly generated node (after a split operation) is assigned a cylinder value with respect to the uniform distribution. Evidently this is not the best possible allocation strategy, since it does not respect locality. Placing pages that are referenced together on the same cylinder reduces the disk service times and this effect is orthogonal with respect to the similarity search algorithms, with the difference that response times are reduced. Initially, all disk arms are positioned in cylinder zero. The simulator executes 100 queries in total, and the response time per query is obtained by calculating the average.

With respect to CPU execution costs, it is assumed that computation time is dominated by the scanning and sorting of each requested set of MBRs. Assume that N MBRs have been fetched from the disks. The scanning of these MBRs costs $O(N)$ time. After scanning, some of them are rejected so that M MBRs remain. In order to sort M elements, the computational effort is $O(M \cdot logM)$ comparisons (assuming heapsort or mergesort). Each main memory word has four bytes and also each number is modeled as four bytes of main memory. Fetching a number from main memory requires one CPU instruction. Therefore, in order to compare two numbers, three CPU instructions are required (two for fetching the operants and one for the comparison). Thus, the computation cost for scanning equals $2 \cdot N$ CPU instructions and the computation time for sorting is equivalent to executing $3 \cdot M \cdot logM$ CPU instructions, resulting in a total of $2 \cdot N + 3 \cdot M \cdot logM$ CPU instructions. Since the MIPS rate for the CPU is a known parameter, the computation time is easily calculated. Although this cost model is simple, it reflects the CPU overhead to a sufficient degree.

4.2 Performance results

Evidently, it is very difficult to provide experimental results by modifying all parameter values. Therefore, we choose to illustrate representative results that shed light in the following issues:

Effectiveness: how many nodes an algorithm visits in order to produce the final answer in comparison to the **WOPTSS** method,

Speed-up and scale-up: how the performance of the methods is affected by increasing the number of disks in the disk array, and/or increasing the size of the database,

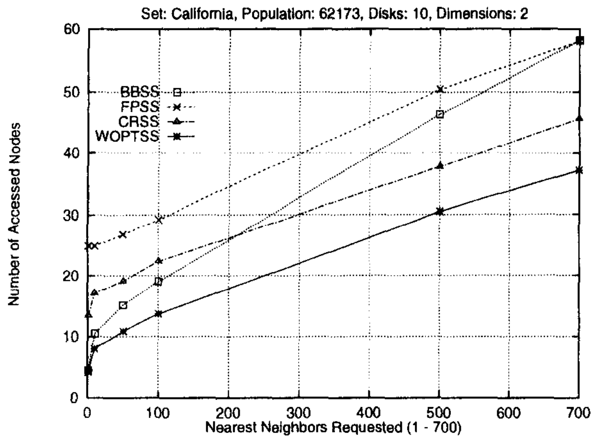
Query size and dimensionality: how the algorithms perform by increasing query size and/or space dimensionality,

Workload: what is the behavior of the methods when concurrent queries are serviced by the system.

By inspecting Figures 8 - 12 and Tables 3, 4 some very interesting observations can be stated. As expected, **WOPTSS** shows the best performance in all experiments contacted. With respect to effectiveness (see Figures 8, 9), **BBSS** fetches the smaller number of nodes up to a point. After this point, **CRSS** is more effective, and the performance of **BBSS** deteriorates by increasing the number of nearest neighbors.

In order to explain this behavior of **BBSS** a small example is given in Figure 13, assuming that $k = 12$.

Since the algorithm chooses to visit the MBR with the smallest D_{min} distance, MBR R_1 will be visited first. If 12 data objects lie in the subtree of R_1, all of them will be visited, despite the fact that some of them will not contribute to the final answer. Evidently, in the branch of R_2 lie some objects that are closer to the query point. Therefore, if R_1 and R_2 were visited in a BFS (Breadth First Search) manner, the total number of disk accesses could have been reduced



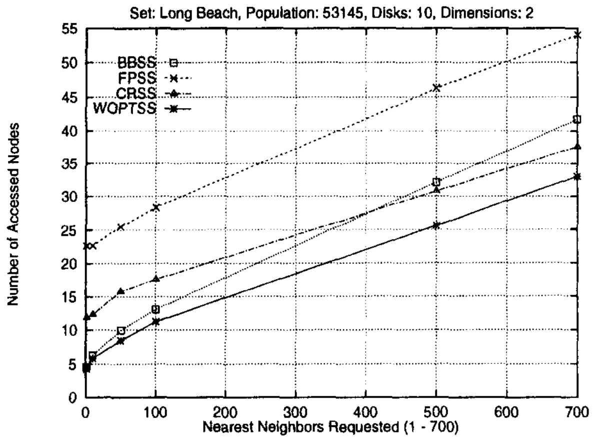
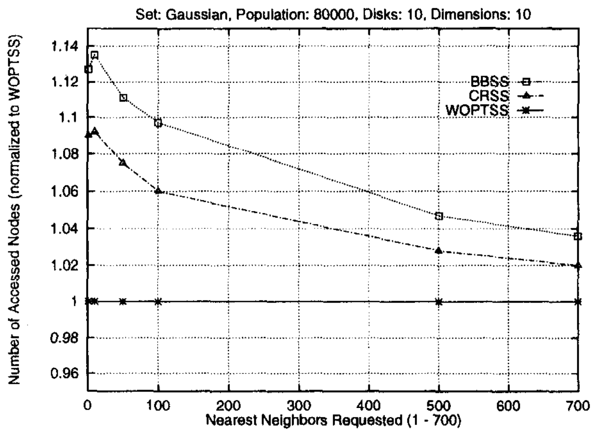


Figure 8: Number of visited nodes vs. query size for 2-d data sets.



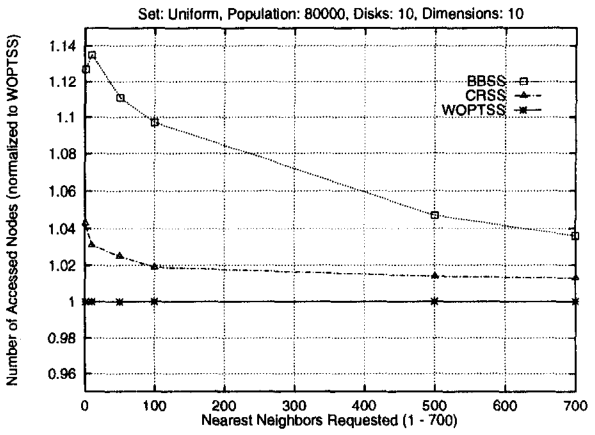
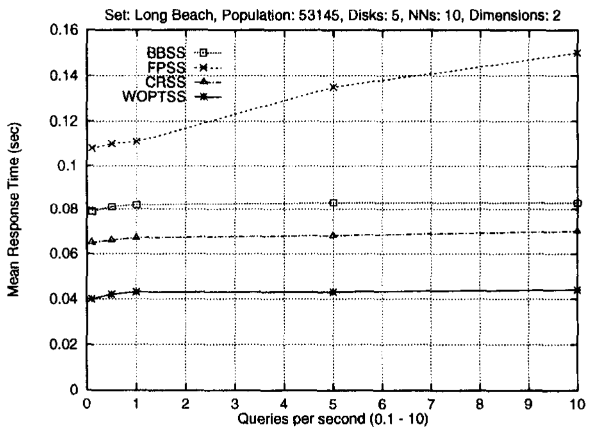


Figure 9: Number of visited nodes (normalized to **WOPTSS**) vs. query size for synthetic data in 10-d space.



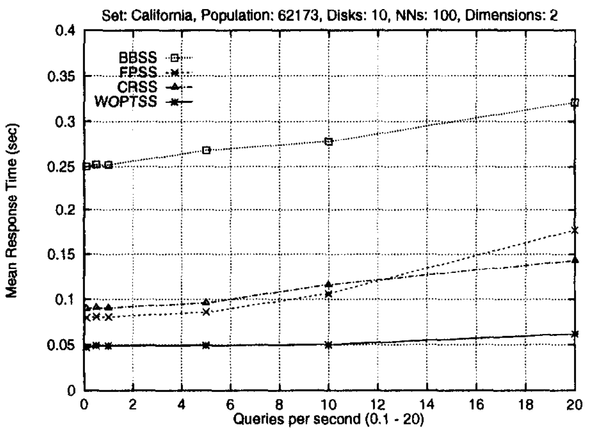
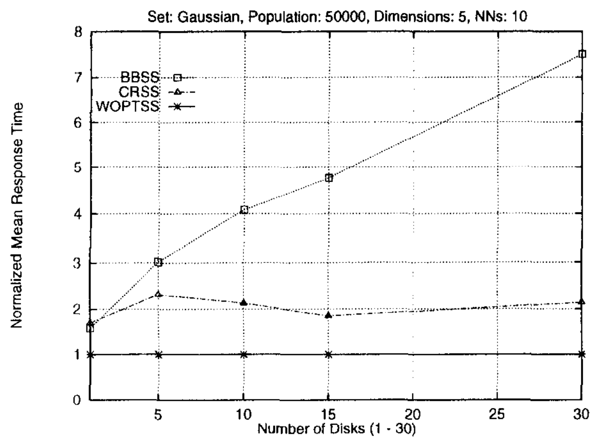


Figure 10: Response time (sec) vs. query arrival rate (λ).



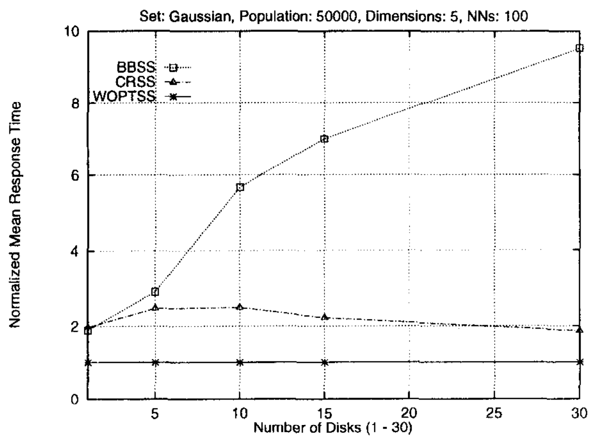
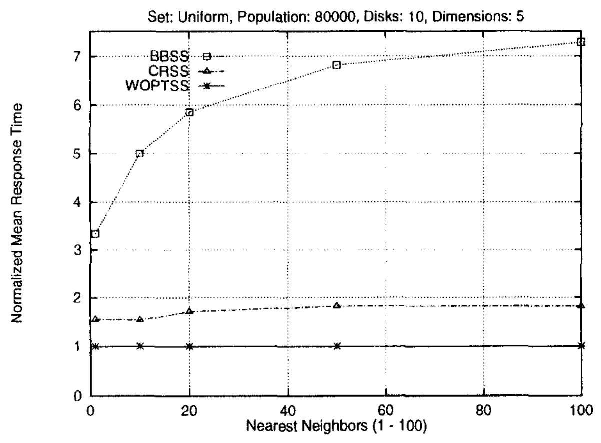


Figure 11: Response time (normalized to WOPTSS) vs. number of disks (λ=5 queries/sec, dimensions=5)



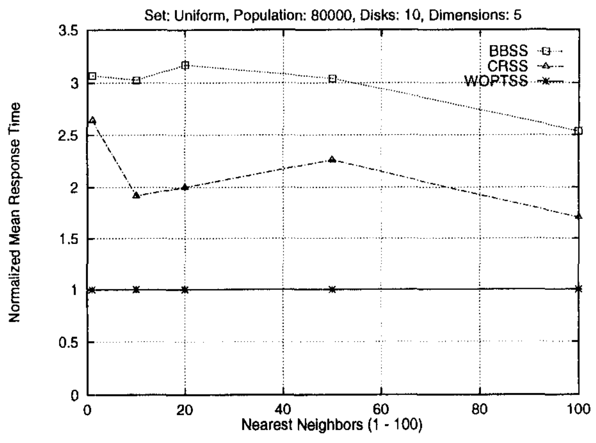


Figure 12: Response time (normalized to **WOPTSS**) vs. number of nearest neighbors (Left: λ=1 queries/sec, Right: λ=20 queries/sec).

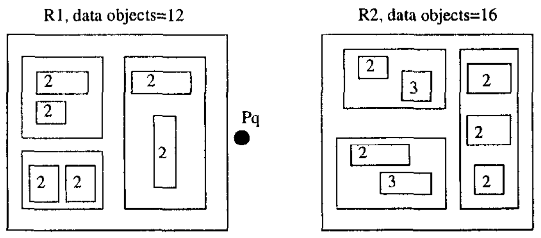


Figure 13: **BBSS** will visit all nodes associated with the branch of R_1, leading to unnecessary accesses.

considerably. The drawback of **BBSS** affects its performance even more, by increasing the number of dimensions, as shown in Figure 9. By increasing the space dimensionality, the overlap of the MBRs increases also, and therefore the pruning of branches becomes a difficult task. Moreover, several MBRs may have zero value for the D_{min} distance, resulting in a difficulty to select the appropriate next branch to follow. The superiority of **CRSS** lies in the fact that it uses a successful combination of BFS and DFS (Depth First Search) of the parallel R*-tree. On the other hand, **BBSS** is DFS-based, whereas **FPSS** is BFS-based. Algorithm **FPSS** fails to control the number of fetched nodes and this results in a large number of disk accesses. The good performance of **CRSS** is retained in all data sets used and all examined dimensionalities.

In Figure 10, we illustrate the response time per query versus the query arrival rate. **FPSS** is very sensitive in workload increase, since there is no control on the number of fetched nodes. Its performance is the worst in comparison to the other methods. However, for small workloads and large number of disks **FPSS** is marginally better than **CRSS**. This is illustrated in Figure 10 (right graph). This happens because the large number of disks compensates the increased demand for disk accesses.

Figure 11 demonstrates response time versus number of disks. It is evident that the speed-up of **CRSS** is better than that of **BBSS**. In fact **CRSS** is between 2 to 4 times faster than **BBSS**. Algorithm **FPSS** is not considered any more, since its performance is very sensitive on the workload and the number of disks in the system.

The performance of the methods with respect to the number of nearest neighbors is illustrated in Figure 12. Again, it is observed that **CRSS** shows the best performance, outperforming **BBSS** by factors (3 to 4 times faster). Finally, Tables 3 and 4 present the scalability of the algorithms with respect to population growth and query size growth. **CRSS** is more stable than **BBSS** and on average is 4 times faster.

Population	**Disks**	**BBSS**	**CRSS**	**WOPTSS**
10,000	5	0.76	0.47	0.23
20,000	10	0.74	0.28	0.15
40,000	20	1.07	0.29	0.15
80,000	40	1.59	0.33	0.16

Table 3: Scalability with respect to population growth: Response time (sec) vs. population and number of disks. (set: gaussian, dimensions: 5, NNs: 20, λ=5 queries/sec).

k	**Disks**	**BBSS**	**CRSS**	**WOPTSS**
10	5	2.48	1.30	0.48
20	10	2.14	0.32	0.19
40	20	2.37	0.55	0.28
80	40	2.95	0.40	0.21

Table 4: Scalability with respect to query size growth: Response time (sec) vs. number of nearest neighbors and number of disks. (set: gaussian, dimensions: 5, population: 80,000, λ=5 queries/sec).

The general conclusion derived is that **CRSS** is on average 2 times slower than **WOPTSS** and outperforms by factors both **BBSS** and **FPSS**. Thus, **CRSS** succeeds in:

- fetching a small number of nodes, and
- exploiting parallelism to a sufficient degree.

For these reasons, the use of **CRSS** is recommended as a simple and efficient similarity search algorithm in a system based on disk arrays. Table 5 contains a qualitative comparison of the algorithms, summarizing the performance evaluation results.

Characteristic	BBSS	FPSS	CRSS	WOPTSS
number of disk accesses	√		√	√
mean response time			√	√
speed-up		√	√	√
scalability			√	√
intraquery parallelism		√	√	√
interquery parallelism	√	limited	√	√

Table 5: Qualitative comparison of algorithms (the symbol √ means good performance).

5 Concluding Remarks

The problem of exploiting I/O parallelism in database systems is a major research direction. In this paper, we have investigated similarity search techniques for disk arrays. The fundamental properties that such an algorithm should preserve are: parallelism must be exploited as much as possible, the total resource consumption should be minimized, and the response time of user queries should be reduced as much as possible.

Three possible similarity search techniques are presented and studied in detail with respect to the above issues. Moreover, an optimal approach (**WOPTSS**) is defined, which assumes that the distance D_k from the query point to the k-th nearest neighbor is known in advance, and therefore only the relevant nodes are inspected. Unfortunately, this algorithm is hypothetical, since the distance D_k is generally not known. However, useful lower bounds are derived by studying the behavior of the optimal method. All methods are studied under extensive experimentation through simulation. Among the studied algorithms, the proposed one (**CRSS**) which is based on a careful inspection of the R*-tree nodes, and leads to an effective candidate reduction, shows the best performance. However, the performance difference between **CRSS** and **WOPTSS** suggests that further research is required in order to reach the lower bound as much as possible. Future research may include:

- the derivation and exploitation of analytical results in similarity search for disk arrays, estimating the response time of a query,
- the study of similarity search on shadowed disks,
- the impact of increasing the number of processors (e.g. in a shared-memory multiprocessor architecture), and
- the application of the algorithm on other access methods for similarity search, like SS-tree, SR-tree, TV-tree and X-tree.

Appendix I - Description of Data Sets

The data sets that are used in order to perform the performance comparison of the algorithms include real-life as well as synthetic ones.

Figure 14 presents the real-life data sets that are selected from the Sequoia 2000 (California places) [23] and the TIGER project (Long Beach) [25]. The CP data set is composed of 62,173 2-d points representing locations of various places in California state. The LB data set consists of 53,145 2-d points representing road segment intersections in Long Beach county.

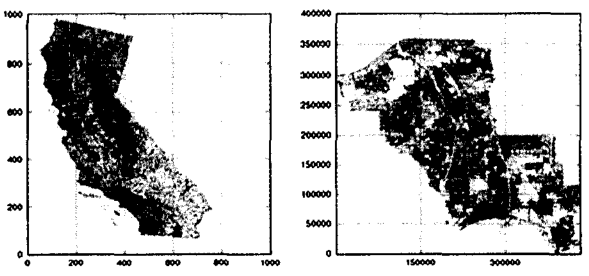


Figure 14: Left: California Places (CP), 62,173 objects. Right: Long Beach (LB), 53,145 objects.

Figure 15 presents two of the synthetic data sets that have been used. The SG set is composed of a number of points generated with respect to the Gaussian (normal) distribution. The SU set consists of a number of points obeying the uniform distribution. The population and the dimensionality of the synthetic data sets were varying during the experiments. In the figure, their 2-d counterparts are illustrated, containing 10,000 points each.

References

[1] N. Beckmann, H.P. Kriegel and B. Seeger: "The R*-tree: an Efficient and Robust Method for Points and

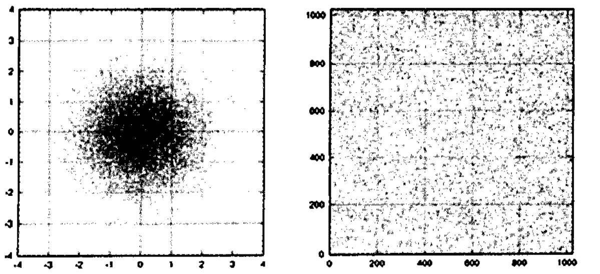


Figure 15: Left: Synthetic Gaussian (SG), 10,000 objects. Right: Synthetic Uniform (SU), 10,000 objects.

Rectangles", *Proceedings of the 1990 ACM SIGMOD Conference*, pp.322-331, Atlantic City, NJ, 1990.

[2] A. Belussi and C. Faloutsos: "Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension", *Proceedings of the 21th VLDB Conference*, pp.299-310, Zurich, Switzerland, 1995.

[3] S. Berchtold, D. Keim and H.-P. Kriegel: "The X-tree: An Index Structure for High-Dimensional Data", *Proceedings of the 1996 VLDB Conference*, Bombay, India, 1996.

[4] S. Berchtold, C. Boehm, D.A. Keim and H.-P. Kriegel: "A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space", *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*, Tucson, AZ, 1997.

[5] S. Berchtold, C. Boehm, B. Braunmueller, D. A. Keim and H.-P. Kriegel: "Fast Parallel Similarity Search in Multimedia Databases", *Proceedings of the 1997 ACM SIGMOD Conference*, pp.1-12, Tucson, AZ, 1997.

[6] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz and D.A. Patterson: "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, vol.26, no.2, pp.145-185, 1994.

[7] C. Faloutsos and I. Kamel: "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension", *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '94)*, pp.4-13, Minneapolis, MN, 1994.

[8] C. Faloutsos, M. Ranganathan and Y. Manolopoulos: "Fast Subsequence Matching in Time-Series Databases", *Proceedings of the 1994 ACM SIGMOD Conference*, pp.419-429, Minneapolis, 1994.

[9] J.H. Friedman, J.L. Bentley and R.A. Finkel: "An Algorithm for Finding the Best Matches in Logarithmic Expected Time", *ACM Transactions on Mathematical Software*, vol.3, pp.209-226, 1977.

[10] A. Guttman: "R-trees: a Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 ACM SIGMOD Conference*, pp.47-57, Boston, MA, 1984.

[11] I. Kamel and C. Faloutsos: "Parallel R-trees", *Proceedings of the 1992 ACM SIGMOD Conference*, pp.195-204, 1992.

[12] I. Kamel and C. Faloutsos: "Hilbert R-tree: an Improved R-tree Using Fractals", *Proceedings of the 20th VLDB Conference*, pp.500-509, Santiago, Chile, 1994.

[13] N. Katayama and S. Satoh: "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries", *Proceedings of the 1997 ACM SIGMOD Conference*, pp.369-380, Tucson, AZ, 1997.

[14] K. Lin, H.V. Jagadish and C. Faloutsos: "The TV-tree: An Index Structure for High Dimensional Data", *The VLDB Journal*, vol.3, pp.517-542, 1995.

[15] Y. Manolopoulos: "Probability Distributions for Seek Time Evaluation, *Information Sciences*, vol.60, no.1-2, pp.29-40, 1992.

[16] B.U. Pagel, H.W. Six, H. Toben and P. Widmayer: "Towards an Analysis of Range Query Performance in Spatial Data Structures", *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '93)*, pp.214-221, Washington DC, 1993.

[17] A.N. Papadopoulos and Y. Manolopoulos: "Performance of Nearest Neighbor Queries in R-trees", *Proceedings of the 6th International Conference on Database Theory (ICDT 97)*, pp.394-408, Delphi, Greece, January 1997.

[18] D.A. Patterson, G. Gibson and R.H. Katz: "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the 1988 ACM SIGMOD Conference*, pp.109-116, Chicago, IL, 1988.

[19] N. Roussopoulos, S. Kelley and F. Vincent: "Nearest Neighbor Queries", *Proceedings of the 1995 ACM SIGMOD Conference*, pp.71-79, San Jose, CA, 1995.

[20] C. Ruemmler and J. Wlkes: "An Introduction to Disk Drive Modeling", *IEEE Computer*, vol.27, no.3, 1994.

[21] B. Seeger and P.A. Larson: "Multi-Disk B-trees", *Proceedings of the 1992 ACM SIGMOD Conference*, pp.436-445, Denver, Colorado, 1991.

[22] T. Sellis, N. Roussopoulos and C. Faloutsos: "The R^+-tree: a Dynamic Index for Multidimensional Objects", *Proceedings of the 13th VLDB Conference*, pp.507-518, Brighton, UK, 1987.

[23] M. Stonebraker, J. Frew, K. Gardels and J. Meredith: "The Sequoia 2000 Storage Benchmark", *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 2-11, Washington, DC, 1993.

[24] Y. Theodoridis and T. Sellis: "A Model for the Prediction of R-tree Performance", *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '96)*, Montreal, Canada, 1996.

[25] TIGER/Line Files, 1994 Technical Documentation / prepared by the Bureau of the Census, Washington, DC, 1994.

[26] D. White and R. Jain: "Similarity Indexing with the SS-tree", *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, LO, 1996.

[27] Y. Zhou, S. Shekhar and M. Coyle: "Disk Allocation Methods for Parallelizing grid files", *Proceedings of the 10th International Conference on Data Engineering*, pp.243-252, Houston, TX, 1994.