INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Multiple Query Points Parallel Search Algorithm (Comb Algorithm) for MultiMedia Database Systems.

Laurian Staicu

Major Report
in The Department of Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Masters in Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2001

ABSTRACT

Multiple Query Points Parallel Search Algorithm (Comb Algorithm)
for MultiMedia Database Systems

In this project, we introduce and present a new search method for fast nearest-neighbor search in high-dimensional feature space, which is called **Comb algorithm**. Most similarity search techniques map the data objects into high-dimensional feature space. The similarity search corresponds to a nearest-neighbor search in the feature space. Fagin and Threshold algorithms are two known methods that perform for nearest-neighbor search with one query point. On the other hand, the method we present works on parallel systems that are identical. We provide an alternative solution with several query points searching in parallel identical systems in as many copies as query points are defined. The algorithm is a trade-off between space storage (multiple copies of the multidimensional system), computation resources, and query execution time.

List of Figures

Fig.17 Parallel retrieval of objects in nearest neighbor query.
System S, S1, S2 are working in parallel.

Fig.18 System configuration for Comb Algorihm. S(1) and S(2) are copies of S.
No of resources k=2; No of steps = st.

Fig.18a The query points corresponding to subsystem S1.

Fig.19 Query points in the Comb algorithm.

Fig.20 Buffer of objects retrieved from all the identical systems. *A* cannot retrieved by S(1).

Fig.21 Random access using B+ tree.

Fig.22 Dynamic comb algorithm – finding the query point to relocate.

Fig.23 Reallocation of the query points.

Fig.24 Distributions of Fagin, Threshold ,and Minstep.

Fig.25 System configuration for the Comb algorithm experiment.

Fig.26 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 60 %; Comb_ no_steps =100;
Seq_ no_steps = 100; No_Objects_display =100;

Fig.27 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 60 %; Comb_ no_steps = 20;
Seq_ no_steps = 100; No_Objects_display =100;

Fig.28 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 80 %; Comb_ no_steps = 100;
Seq_ no_steps = 500; No_Objects_display =500;

Fig.29 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 80 %; Comb_ no_steps = 100;
Seq_ no_steps = 500; No_Objects_display =100;

Fig.30 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 80 %; Comb_ no_steps = 50;
Seq_ no_steps = 500; No_Objects_display =100;

Fig.31 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 30 %; Comb_ no_steps = 50;
Seq_ no_steps = 500; No_Objects_display =100.

Fig.32 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 30, 60, 100 %; Comb_ no_steps = 50;
Seq_ no_steps = 100; No_Objects_display =100;

Fig.33 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system =7, 11; ε = 100 %; Comb_ no_steps = 50;
Seq_ no_steps = 100; No_Objects_display =100;

Contents

1. Introduction

1.1 Overview of the Multimedia Databases.

Multimedia data is represented by digital images, audio, video, graphics, and animation objects. The acquisition, generation, storage, and processing of multimedia data in computers and transmission over networks have grown tremendously.

This fast growth has occurred due to three main factors. First, the recent technological advances have spread the use of personal computers with increased computational power. Moreover, we now have more affordable high-resolution devices to capture and display multimedia data (monitors, printers, digital cameras, scanners, etc) and high-density storage devices. Second, high-speed data communication networks have been developed; the WWW has proliferated, and software to manipulate data is now available. Finally, the third factor is the increasing use of multimedia data in many existing applications and also new ones under development.

This fast development is expected to continue at an even faster pace in the coming years. Multimedia data can provide more effective dissemination of information in science, engineering, medicine, biology and social sciences. It also facilitates the development of new paradigms in distance learning and interactive personal and group entertainment.

Databases have been developed to gather and manage huge amounts of data in different applications. Databases provide security, availability, consistency, concurrency, and integrity of data. From a user point of view, they provide three main functionalities. These are the easy manipulation, query, and retrieval of relevant information from huge amounts of stored data. The retrieval is done by abstracting the details of storage access. Until recently, most data handled by computer applications were textual data. Therefore, the traditional databases have been designed and optimized to manage them [1].

Multimedia Database Systems (MMDBS) must deal with the increased usage of huge amounts of multimedia data in several and diverse applications. These applications include: digital libraries, manufacturing and retailing, art and entertainment, journalism, and so forth. Some inherent functions of multimedia data have direct and indirect impacts on the design and development of a multimedia database [2].

MMDBS need to have all the functionalities of traditional databases. In addition, they must have new and enhanced functionalities and features. Broadly, MMDBS are required to provide unified frameworks for storing, processing, retrieving, transmitting, and presenting a variety of media data types in a wide variety of formats. At the same time, they must adhere to numerous constraints that are in traditional databases.

Therefore, a Multimedia Database System is a system that can store and retrieve multimedia objects, such as gray-scale medical images in 2-d or 3–d (e.g., MRI brain

scans), one –dimensional time series, two-dimensional color images, digitized voice or music, traditional data types, video clips, like 'prod-id', 'date', 'title', and any other user-defined data types.

This project is focusing on the design of a fast searching algorithm by content. A typical query by content would be for example 'in a collection of stamps, find all the stamps with the image of a car'.

Some specific applications include the following [3]:

- Image databases are very much used to support queries on shape, color, and texture.

- Scientific databases with collections of sensor data. In this case, the objects are time series, or more general, vector fields, that is, tuples of the form, e.g., <x,y,z,t,pressure,temperature,...>. For example, in the weather data, geological, environmental, astrophysics databases, etc., we want to ask queries of the form, 'find past days in which the air temperature and wind patterns are similar to today's pattern' to help in the prediction of the weather.

- Marketing, financial, and production time series (for example, sales patterns, stock prices, etc). In these types of databases, the typical queries would be 'find companies whose stock prices move likewise' or 'find cases in the past that resemble last year's sales pattern of our products'.

- Medical databases that store 1-d objects (e.g. ECGs), 2-d images (e.g., X-rays) and 3-d images (e.g., MRI brain scans). Ability to retrieve quickly past cases with similar symptoms would help us to determine a diagnosis; moreover, these can be also used for medical teaching and research purposes.

- Multimedia databases with audio (music, voice), video etc. Users might want to retrieve for example, similar video clips or music scores.

- Photograph and text archives, digital libraries with ASCII text, bitmaps, gray-scale and color images.

- Electronic encyclopedias, electronic books, and office automation.

- DNA databases where there is a large collection of long strings (hundred or thousand characters long) from a four-letter alphabet (B,E,C,D); a new string has to be matched against the old strings, to find the best candidates. The distance function is the editing distance (smallest number of insertions, deletions and substitutions that are needed to transform the first string to the second).

A Multimedia Database System needs to manage several different types of information pertaining to the actual multimedia data. These are broadly classified as follow:

1) **Media Data.** This is the actual data. For example, this refers to images, audio, and video that are captured, digitized, processed, compressed and stored.
2) **Media format data.** This contains information pertaining to the format of the media data after it goes through the acquisition, processing and encoding phases. For example, this contains information such as the sampling rate, resolution, frame rate, encoding scheme, etc.
3) **Media keyword data.** This contains the keyword descriptions, usually related to the generation of the media data. For example, for a video, this might include the date, time, and place of recording, the person who recorded, the scene that is recorded, etc. This is also referred to as content descriptive data.
4) **Media feature data.** This contains the features derived from media data. A feature characterizes the media data. For example, this could contain information about the distribution of the colors, the kinds of textures and the different shapes present in an image. This is also referred to as content descriptive data.

The last three types are called 'meta' data [4]. This is because they constitute the information describing several different aspects of the media data. These are derived from the original data as presented in Fig. 1.

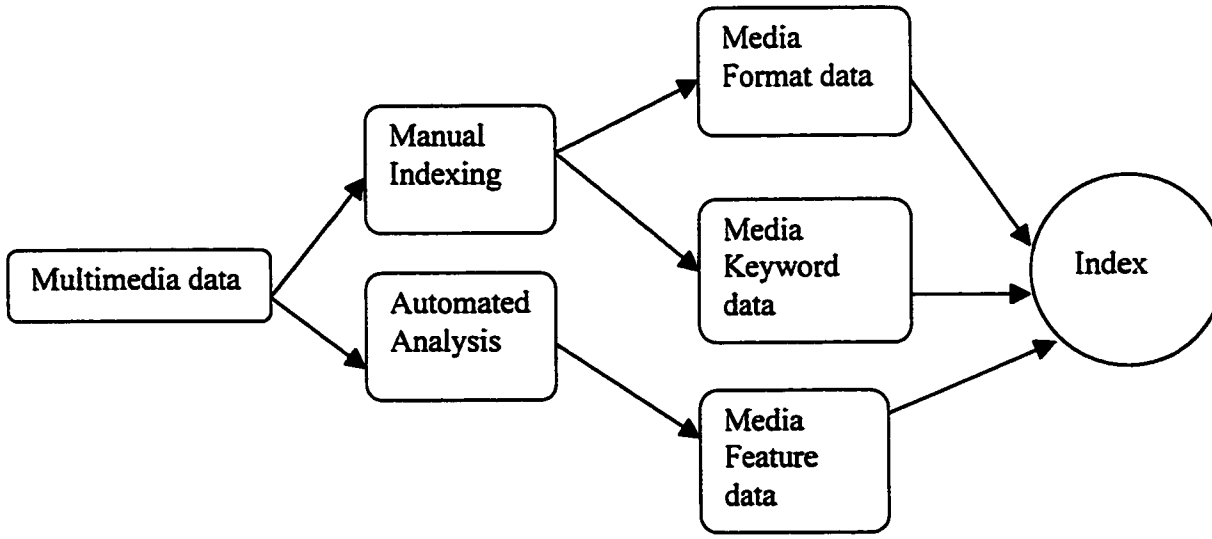


Fig.1 Metadata generation process

The media keyword data and media feature data are used as indices in the search process. The media format data is used in the presentation of retrieval results. Multimedia databases require several functionalities that are not present in traditional databases. These are presented in the boxes in Fig.2.

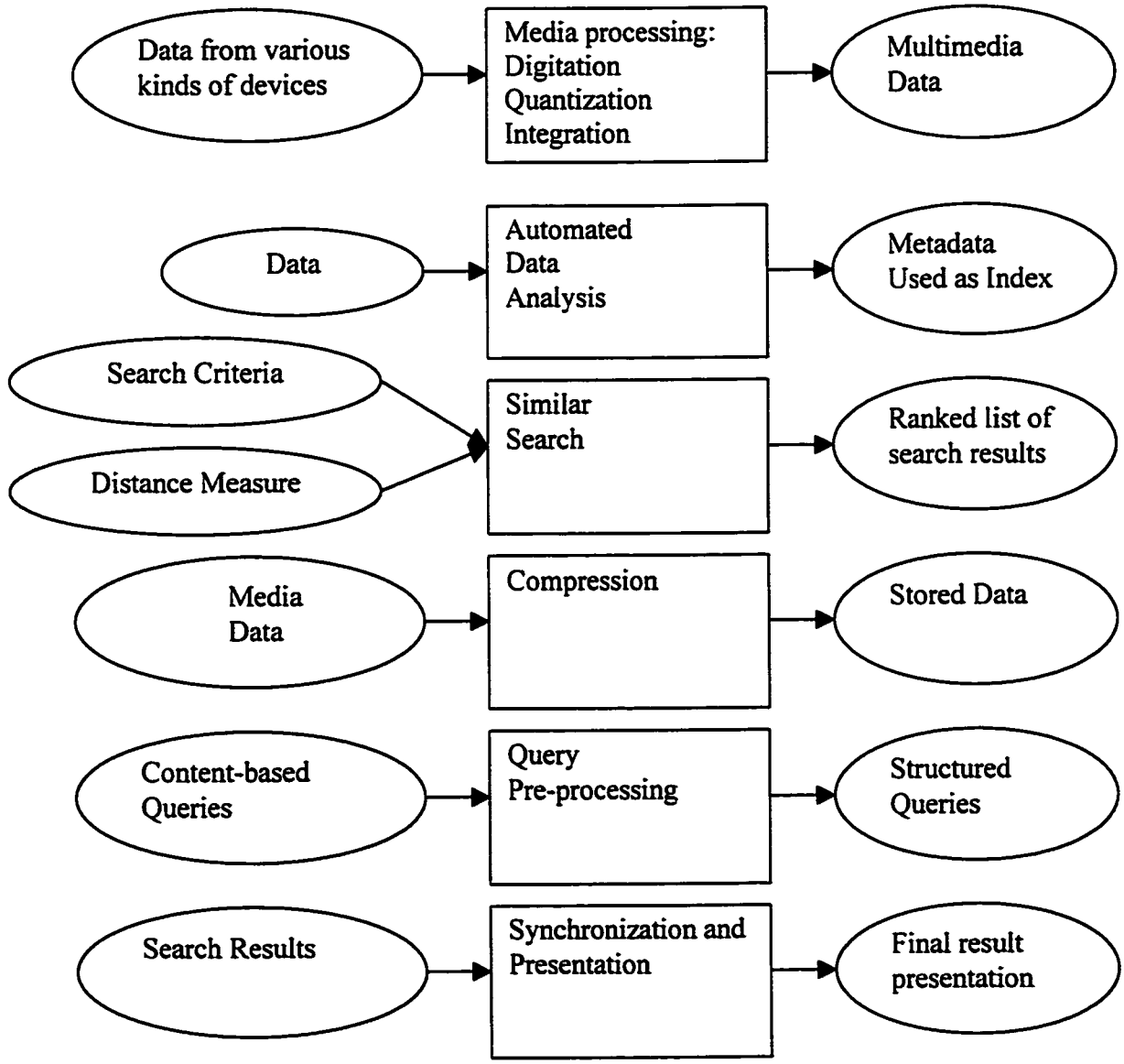


Fig. 2 Salient functions of MMDBS not in traditional databases.

The major activities in managing the data in multimedia databases are the following:

1) **Data acquisition**: In addition to conventional means, data can be input to the database from newer kinds of devices such as scanners for image data; microphones, synthesizer, musical instruments for audio data; video cameras, VCRs and frame grabbers for video data.

2) **Data formats**: These are scores of file formats. Examples include GIF, TIFF, JPEG, etc. for images; au, wav, midi, etc. for audio; and MPEG, etc. for video.

3) **Data storage**: The data for images, audio and video are huge in size and are usually stored in compressed form. Various forms of stripping and other storage schemes are used for efficient access to data.

4) **Index organization**: The index organization requires multi-dimensional structures such as R-trees, hB-trees, Grid files, etc.

5) **Query**: Keyword-based queries are inadequate for multimedia data. Novel schemes like query-by-example and query-by-content are required.

6) **Search and retrieval**: The search is more likely to be a similarly search. The query result is a ranked list of data items similar to the query rather than exact matches. Relevance feedback from the user to the search engine. based on retrieval results, is required.

7) **Transmission**: There are more stringent real-time, Quality of Service (QoS) and synchronization requirements on the transmission due to the time-dependent nature of audio and video for the retrieved data to be meaningful.

8) **Presentation**: Newer devices need be integrated into the system. For example, speakers for audio, high resolution monitors for images and video. The presentation should handle ranked results and different media.

Therefore, in a collection of multimedia objects, we can find queries of special interest. The most frequent types of queries are the following [5]:

1) **Range query**: For example, "find all lakes in Canada" or "find all cities within 50 kilometers of Toronto". In this case, the user specifies a region (the region covered by Canada or a circle around Toronto) and asks for all the objects that cross this region. The query point is a special case of the range query, when the query region collapses to a point.

Typically, the range query requests all the special objects that intersect a region; similarly, it could request the spatial objects that are completely contained, or that contain the query region.
In this project, we mainly focus on the "intersection" variation; the remaining two can usually be answered by slightly modifying the algorithm for the "intersection" version.

2) A second type of query would be the **nearest neighbor query**, a slight generalization of the nearest neighbor query for secondary keys. For example, "find the 5 nearest grocery stores to our house." Again, the user specifies a point or a region, and the system will return with k closest objects. The distance is typically the Euclidean (L2 norm), or some other distance function (e.g., city-block distance L1, or the Loo norm).

3) **Spatial joins, or overlays**. For example, in CAD design, "find the pairs of elements that are closer than ε" (and thus create electromagnetic interference to each other). Or, given a collection of rivers and a collection of cities, "find all the cities that are within 15km of a river."

Therefore, records with k numerical attributes can be visualized as k-dimensional points. Spatial access methods are designed to handle multidimensional points, lines, rectangles, and other geometric bodies. There are two proposed methods:
1) Methods that use space-filling curves (also known as z-ordering or linear quad-trees);
2) Methods that use treelike structures: R-tree and its variant.

In this project, we will focus only on the R-tree method. Next, we will present the R-tree structure.

1.2 R-Tree

Guttman proposed the R-tree [6]. The R-tree can be seen as an extension of the B-tree for multidimensional objects. A spatial object is represented by a minimum-bounding rectangle (MBR).

In the R-tree, we can distinguish two types of nodes: leaf nodes and non-leaf nodes. 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. On the other hand, 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.

In the R-tree, the parent nodes are allowed to overlap, and this can be considered as the main innovation of this kind of three. In fact, the R-tree can assure good space utilization and remain balanced as the same time. Fig. 3 illustrates data rectangles (solid boundaries) organized in an R-tree with fan-out=3. Fig. 4 shows the file structure for the same R-tree, where nodes correspond to disk pages.

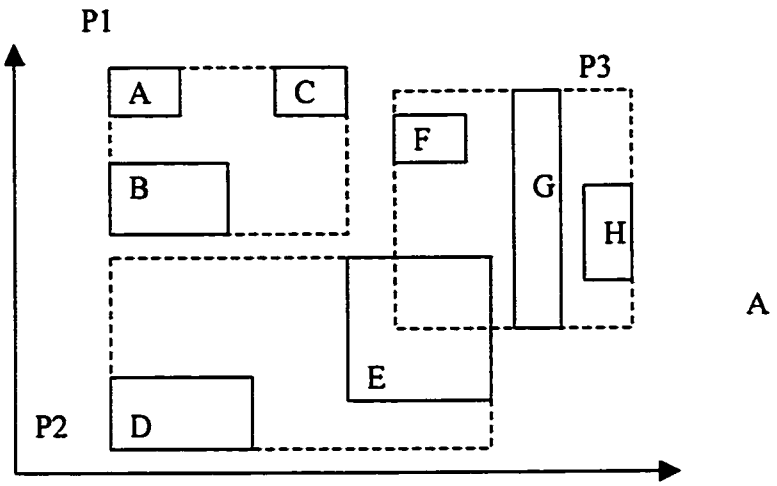


Fig. 3 Data (solid-line rectangles) organized in an R-tree with fan-out = 3

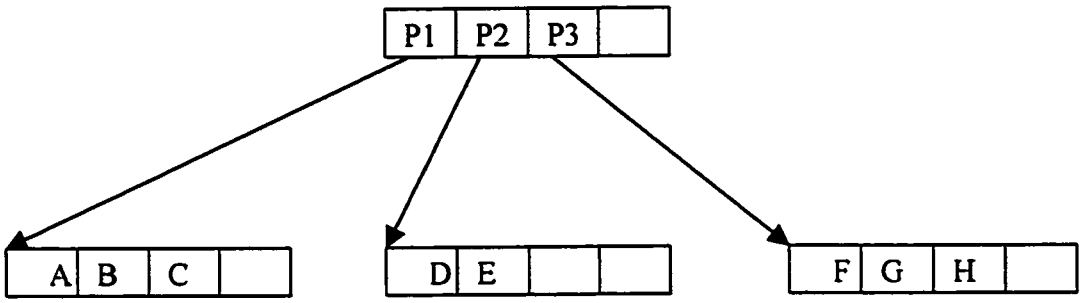


Fig. 4 The resulting R-tree on disk

The main focus of the R-tree is to improve the search time. Guttman [6] proposed a packing technique that minimizes the overlap between different nodes in the R-tree for static data. This packing technique consists in ordering the data in ascending x-low value and scanning the list, filling each leaf node to capacity. On the other hand, based on the Hilbert curve, another packing technique is proposed. This is much more improved, and in this case, the idea is to sort the data rectangles on the Hilbert value of their centers.
Trying to minimize the dead space that an MBR may cover, a more general minimum bounding shapes is considered. Gunther [7] proposed the cell trees, which introduce diagonal cuts in arbitrary orientation. There have been suggested minimum bounding shapes that are concave or even have holes (e.g., in the hB-tree).

One of the most important ideas in R-tree research is the idea of deferred splitting: Beckmann et al. proposed the R*-tree [8], which was reported to outperform Guttman's R-trees [6] by approximately 30%. The main idea is the concept of forced reinsert, which tries to defer the splits to attain better utilization. When a node overflows, some of its children are carefully chosen. After that, they are deleted and reinserted, usually resulting in a better-structured R-tree. This idea of deferred splitting was also exploited in the Hilbert R-tree; there, the Hilbert curve is used to impose a linear ordering on rectangles, thus defining who the sibling of a given rectangle is, and subsequently applying the 2 to3 (or s-to-(s+1)) splitting policy of the B*-tree. Both methods attain higher space utilization as well as better response time (since the tree is shorter and more compact) than Guttman's R-tree [6].

The analysis of the R-tree performance has attracted lot of interest: Faloutsos et al.[9] provide formulas, which assume that the spatial objects are uniformly distributed in the address space. Faloutsos and Kamel [10] relaxed the uniformity assumption; there it was shown that the fractal dimension is a very good measure of the nonuniformity, and that it leads to accurate formulas to estimate the average number of disk access of the resulting R-tree. In addition, the fractal dimension helps to estimate the selectivity of spatial joins.

Algorithms

Insertion. When a new rectangle is inserted, we traverse the tree to find the most suitable leaf node; we extend its MBR if necessary, and store the new rectangle there. If the leaf node overflows, we split it.

Split. Regarding the performance of the R-tree, the split is one of the most important operations. Guttman [6] suggested several heuristics to divide the contents of an overflowing node into sets and store each set in a different node. As mentioned in the R*-tree [8] and in the Hilbert R-tree [12], deferred splitting will improve the performance. As in B-trees, a split may propagate upwards.

Range queries. In this case, the tree is traversed (comparing the query MBR with the MBRs in the current node); accordingly, nonpromising and potentially large branches of the tree can be pruned early.

Nearest Neighbors. The algorithm follows a "branch and bound" technique similar to nearest-neighbor searching in clustered files. Given the query point Q, we examine the MBRs of the highest-level parents. We proceed in the most promising parent, estimate the best-case and worst-case distance from its contents, and using these estimates, we prune out nonpromissing branches of the tree. Roussopoulos et al. [11] give the detailed algorithm for the R-tree.

Spatial Joins. Given two R-trees, the algorithm builds a list of pairs of MBRs that intersect. Then, it examines each pair in more detail, until we reach the leaf level.

By considering all these, we can draw the conclusion that R-trees [6] are one of the most promising spatial access methods. Among its variations, the R*-trees [8] and the Hilbert R-trees [12] seem to achieve the best response time and space utilization, in exchange for more elaborate splitting algorithms.

Further on, we should mention the "dimensionality curse". Unfortunately, all the spatial access method which are design to handle multidimensional objects will suffer for high dimensionalities *n*: for the R-tree, as the dimensionality *n* grows, each MBR will require more space; thus the fan-out of each R-tree page will decrease. This will result in a taller and slower R-tree. However, R-trees have been successfully used for 20-30 dimensions. Most research in exploring the multi-dimensional spaces is concentrated on low dimensional data-structures, such as R-tree. These structures can be extended to higher dimensions, but this results in performance degradation. The performance degrades because as the dimension increases, the querying cost often increases exponentially. The index structures deployed become less effective as a pre-filter for selections and join operations.

1.3 Nearest Neighbor Queries

As previously mentioned, a very common type of query is to find the *k* nearest neighbor objects to a given point in space. Processing such queries requires significantly different search algorithms than those for location or range queries.

Roussopoulos [11] proposed an efficient branch-and-bound R-tree traversal algorithm to find the nearest neighbor object to a point, and then generalized it to find the k nearest neighbors. We can explain this by first introducing two metric definitions: Minimum Distance (MINDIST) and Minimax Distance (MINMAXDIST).

Minimum Distance (MINDIST). The first metric we introduce is a variation of the classic Euclidean distance applied to a point and a rectangle (MBR). If the point is inside the rectangle, the distance between the rectangle and the point is zero. On the other hand, if the point is outside the rectangle, we use the square of the Euclidean distance between the point and the nearest edge of the rectangle. We use the square of the Euclidean distance because it involves fewer and less costly computations. In order to avoid any misunderstanding, whenever we refer to distance, we will be using the square of the distance, and the construction of our metrics will reflect this.

Minimax Distance (MINMAXDIST). In order to avoid visiting unnecessary MBRs, we should have an upper bound of the NN distance to any object inside an MBR. This will allow us to prune MBRs that have MINDIST higher than this upper bound. The following distance construction (called MINMAXDIST) is being introduced to compute the minimum value of all the maximum distances between the query point and points on the each of the n axes respectively. The MINMAXDIST guarantees there is an object within the MBR at a distance less than or equal to MINMAXDIST.

In Fig. 5 we illustrate MINDIST and MINMAXDIST in 2-space.

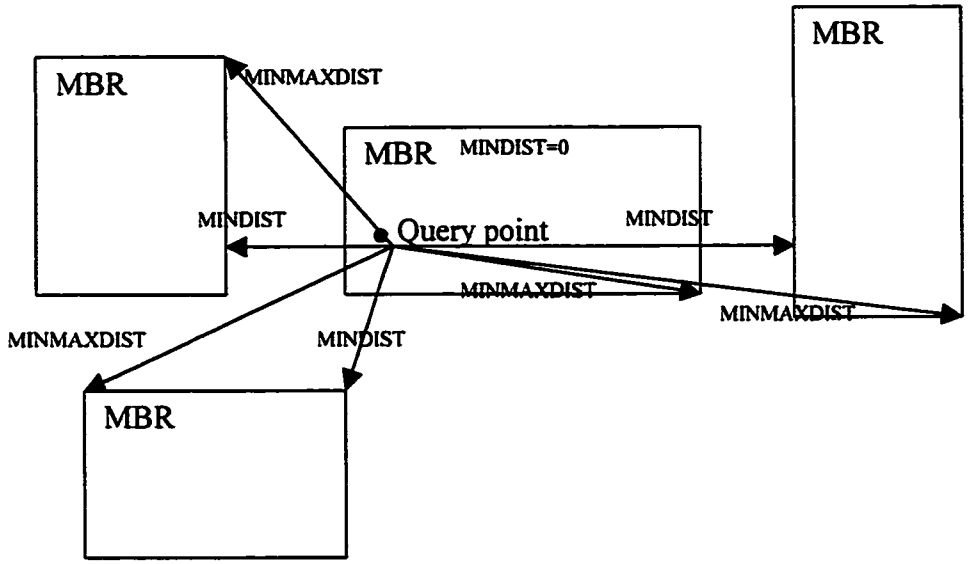


Fig. 5 MINDIST and MINMAXDIST in 2-Space

Further on, we will present the algorithm for the Nearest Neighbor Algorithm for R-trees. More specifically, we will present the branch-and-bound R-tree traversal algorithm to find the k-NN objects to a given query point. Firstly, we will discuss the benefit of using the MINDIST and MINMAXDIST metrics to order and prune the search tree. Secondly, we will present the algorithm for finding 1-NN, and finally, generalize the algorithm for finding the k-NN.

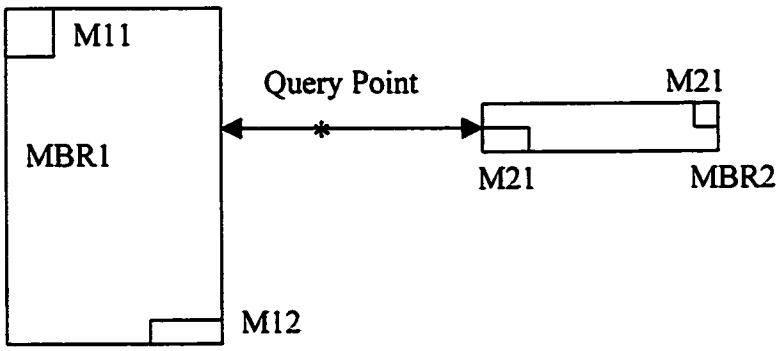
MINDIST and MINMAXDIST for ordering and pruning the search.

Branch-and-bound algorithms have been studied and extensively used in the area of artificial intelligence and operations research. In fact, if the ordering and pruning heuristics are chosen well, they can significantly reduce the number of nodes visited in a large search space.

Search Ordering. The heuristics we use in this algorithm are based on orderings of the MINDIST and MINMAXDIST metrics. While the MINMAXDIST metric is the pessimistic (though not worst case) choice, the MINDIST ordering is the optimistic one. In fact, since MINDIST estimates the distance from the query point to any enclosed MBR or data object as the minimum distance from the point to the MBR itself, it is the most optimistic choice possible. On the other hand, MINMAXDIST produces the most pessimistic ordering that need ever been considered due to the properties of MBR and the construction of it.

By applying a depth first traversal to find the NN to a query point in an R-tree, the optimal MBR visit ordering depends not only on the distance from the query point to each of the MBRs along the path(s) from the root to the leaf node(s), but also on the size and layout of the MBRs (or in the leaf node case, objects) within each MBR. In particular, one can construct example in which the MINDIST metric ordering produces tree traversals that are more costly (in terms of nodes visited) than the MINMAXDIST metric.

This is shown in Fig. 6. MINDIST metric ordering will lead the search to MBR1 which would require the opening of M11 and M12. If on the other hand, MINMAXDIST metric ordering is used, visiting MBR2 results in a smaller estimate of the actual distance to the NN (which will be found to be M21) which will then eliminate the need to examine M11 and M12. The MINDIST ordering optimistically assumes that the NN to P in MBR is going to be close to MINDIST(M,P), which is not always the case. Likewise, counterexamples could be constructed for any predefined ordering.



1. MINDIST ordering: if we visit MBR1 first, we have to visit M11, M12, MBR2 and M21 before finding the NN.
2. MINMAXDIST ordering: if we visit MBR2 first, and then M21, when we eventually visit MBR1, we can prune M11 and M12.

Fig. 6 MINDIST is not always the better ordering

As previously mentioned, the MINDIST metric produces most optimistic ordering, but that is not always the best choice. Many other orderings are possible by choosing metrics that compute the distance from the query point to faces or vertices of the MBR which are further away. The most important feature of MINMAXDIST(P,M) is that it computes the smallest distance between point P and MBR M that guarantees the finding of an object in M at a Euclidean distance less than or equal to MINMAXDIST(P,M).

Search Pruning. There are three main strategies to prune MNRs during the search:

1) An MBR M with MINDIST(P,M) greater than the MINMAXDIST(P,M') of another MBR M' is discarded because it cannot contain the NN.

2) An actual distance from P to a given object O which is greater than MINMAXDIST(P,M) for an MBR M can be discarded because M contains a object O', which is nearer to P.

3) Every MBR M with MINDIST(P,M) greater than the actual distance from P to a given object O is discarded because it cannot enclose on object nearer than O.

Even tough we specified only the use of MINMAXDIST in pruning strategy no. 1, in practice, there are cases where it is more recommended to apply MINDIST (strategy no. 3). For example, when there is no dead space (or at least very little) in the nodes of the R-tree, MINDIST is a much better estimate of ||(P,N)||, the actual distance to the NN than is MINMAXDIST, at all levels in the tree. So, it will prune more candidate MBRs than will MINMAXDIST.

Nearest Neighbor Search Algorithm

The nearest neighbor search algorithm presented here implements an ordered depth first traversal. This starts with the R-tree root node and proceeds down the tree. Initially, our guess for the nearest neighbor distance (call it Nearest) is infinity. During the descending phase, at each newly visited nonleaf node, the algorithm computes the ordering metric bounds (e.g. MINDIST) for all its MBRs and sorts them (associated with their corresponding node) into an Active Branch List (ABL). Then, we apply two pruning strategies 1 and 2 to the ABL to remove the unnecessary branches. The algorithm iterates on this ABL until the ABL is empty: For each iteration, the algorithm selects the next branch in the list and applies itself to the node corresponding to the MBR of the branch. At a leaf node (DB objects level), the algorithm calls a type specific distance function for each object and selects the smaller distance between current value of the Nearest and each computed value and updates Nearest appropriately. After that, we take this new estimate of the NN and apply pruning strategy 3 to remove all branches with MINDIST(P,M) >Nearest for all MBRs M in the ABL.

Generalization: Finding the k Nearest Neighbors

This algorithm that we presented above can be generalized to answer queries of the type: find The *k* Nearest Neighbors to a given Query Point, where k is greater than zero.

There are only two differences:
There is a need of a sorted buffer of at most *k* current nearest neighbors, and
The MBRs pruning is done according to the distance of the furthest nearest neighbor in this buffer.

2. Fagin's Algorithm

Ronald Fagin [13] introduced an algorithm that has a direct applicability for the Multimedia Middleware System. Such a system may often be "middleware" due to the many varieties of data that a multimedia database system must handle. In other words, the system is "on top of" various subsystems, and integrates results from the subsystems. A good example of such a middleware system would be the Garlic [18] system of the IBM Almaden Research Center. In fact, Garlic [18] is integrating data that resides in different database systems as well as a variety of non-database data servers. A single Garlic query can access data in a number of different subsystems. An example of a nontraditional subsystem that Garlic accesses is QBIC [19]. QBIC can search for images by different visual characteristics (e.g., color, texture, etc).

Some of the problems associated with middleware systems include dirty data (caused by multiple sources having conflicting information), schema integration, security concerns, etc.

The database systems were previously required to store only small character strings, such as the entries in a tuple in a traditional relational database. In this case, the data was entirely homogeneous. However, now we want the database systems to be able to deal not only with character strings (both small and large), but also with a heterogeneous variety of multimedia data (such as images, video, and audio). What is more, the data that we want to access and combine may reside in a variety of data repositories, and therefore, we may want our database system to serve as middleware that will access such data.

A very significant difference between multimedia data and small character strings is that multimedia data may have attributes that are inherently fuzzy. For instance, we do not have the case of a given image which is simply "blue" or "not blue". Instead, there is a degree of blueness, which ranges between 0 (not at all blue) and 1 (totally blue).

One way to deal with this kind fuzzy data is to use an aggregation function t. If $x_1,\ldots,x_m$ (each in the interval [0,1]) are the grades of object R under the m attributes, then $t(x_1,\ldots,x_m)$ is the overall grade of object R. These aggregation functions are useful in other contexts as well.

Two popular choices for the standard aggregation functions are min and average (or the sum, in contexts where we do not care if the resulting overall grade no longer lies in the interval [0,1]). When the choice is min, we have the following situation: under the standard rules of fuzzy logic, if object R has grade x_1 under attribute A_1 and x_2 under attribute A_2, then the grade under the fuzzy conjunction $A_1 \wedge A_2$ is min (x_1, x_2).

We can define an aggregation function t as monotone if $t(x_1,\ldots, x_m) \leq t(x'_1,\ldots, x'_m)$ whenever $x_i \leq x'_i$ for every i. The monotonicity is a reasonable property to require from an aggregation function: if for every attribute, the grade of object R' is at least as high as that of object R, then we would expect the overall grade of R' to be at least as high as that of R.

Let us give few definitions:

If x is an object and Q is a query (called atomic query), let us denote by $\mu_Q(x)$ the grade of x under the query Q. This is possible by considering the standard rules of fuzzy logic, as defined by Zadeh [14]. A graded set is consisting of all pairs $(x; \mu_{Ai}(x))$, where *x* is a retrieved object and $\mu_{Ai}(x)$ is the grade of *x* under query Ai. Now the monotonic property of query Ft(A,B) is as follow:

If $\mu_A(x) \leq \mu_A(x')$, and $\mu_B(x) \leq \mu_B(x')$, then $\mu_{Ft(A,B)}(x) \leq \mu_{Ft(A,B)}(x')$,
where $\mu_{Ft(A,B)}(x) = t(\mu_A(x), \mu_B(x))$, where the **aggregation function t** is called **triangular norm** and satisfies the following properties :

a) Conservation: $t(0,0) = 0$; $t(x,1) = t(1,x) = x$.
b) Monotonicity: $t(x_1,x_2) \leq t(x_1',x_2')$ if $x_1 \leq x_1'$ and $x_2 \leq x_2'$.
c) Commutativity: $t(x_1,x_2) = t(x_2,x_1)$.
d) Associativity: $t(t(x_1,x_2),x_3) = t(x_1,t(x_2,x_3))$.
e) Strictness: $t(x_1,x_2) = 1$ iff $x_i = 1$ for every i.
f) Monotonicity: $t(x_1,x_2) \leq t(x_1',x_2')$ if $x_i \leq x_i'$ for every i.

The above properties can be upgraded to any number of retrieved objects and atomic query (A and B in the above example).

In other words a graded set consists of retrieved objects which have scores assigned to them depending on how well they satisfy an atomic query.

Let us consider the query: color = 'blue'. We can assume that the subsystem will output the graded set consisting of all objects, one by one, along with their grades under the subquery (query that refers to a subsystem), in sorted order based on grade, until Garlic tells the subsystem to stop. Later, Garlic [18] could tell the subsystem to resume outputting the graded set where it left off. Alternatively, Garlic could ask the subsystem to sort let's say the top 12 objects along with their grades, then, request the next 12, etc. This type of access can be referred as "sorted access". On the other hand, Garlic can interact with the subsystem in another way. More specifically, Garlic could ask the subsystem the grade (with respect to a query) of any given object. This can be referred as "random access".

Considering all these limited ways of access to the subsystems, we can state that the issues of efficient query evaluation in a middleware system are very different from those in a traditional database system. In fact, it is not even clear what "efficient" means in a middleware system.

Following, we will present the cost of an algorithm. This cost represents the amount of information that an algorithm obtains from the database.

The sorted access cost is the total number of objects obtained from the database under sorted access. For example, if we have two lists (corresponding in the case of conjunction to a query with two conjuncts), and some algorithm requests altogether, the top 100 objects from the first list and the top 20 objects from the second list, then, the sorted access cost for this algorithm is 120.
The random access cost is the total number of objects obtained from the database under random access. The middleware cost is taken to be c1*S + c2*R, where S is the sorted access cost, R is the random access cost, and c1 and c2 are positive constants. Since it ignores the costs inside of a "black box" like QBIC [19], the middleware cost is not a measure of total system cost. There are some situations (for example, in the case of a query optimizer), where there is a need of a more comprehensive cost measure. Finding such a cost measure is an interesting open problem.

The middleware cost is taken for convenience to be simply the sum of the sorted access cost and the random access cost, S + R. Both "formulas" of middleware cost (S + R and c1*S + c2*R) are within constant multiples of each other, and therefore, the same results hold in the "big O" notation.

Algorithms for Query Evaluation

Following, we will present an algorithm for evaluating monotone queries [13]. This algorithm is optimally efficient up to a constant factor, under some particular assumptions. Most probably, the most important queries are the queries that are conjunctions of atomic queries.

Let us presume for now that the conjunctions are being evaluated by the standard min rule. An example of a conjunction of atomic queries is the query (Artist='Beatles') $\wedge$ (AlbumColor='red').

In this example, the first conjunct Artist='Beatles' is a traditional database query, and the second conjunct AlbumColor='red' would be addressed to a subsystem such as QBIC. Consequently, in answering this query, two different subsystems (in this case, perhaps a relational database management system to deal with the first conjunct, along with QBIC to deal with the second conjunct) would be involved.

In this situation, in order to answer the query, Garlic has to gather the information from both subsystems. Under the assumption that there are not many objects that satisfy the first conjunct Artist='Beatles', a good way to evaluate this query would be to first determine all objects that satisfy the first conjunct (call this set of objects S), and then to obtain grades from QBIC (using random access) for the second conjunct for all objects in S. We can therefore obtain a grade for all objects for the full query. If the artist is not the Beatles, then the grade for the object is 0 (since the minimum of 0 and any grade is 0). If the artist is the Beatles, then the grade for the object is the grade obtained from QBIC in evaluating the second conjunct (since the minimum of 1 and any grade g is g).

At this point, we should note that the result of the full query is a graded set, where:
- the only objects whose grade is nonzero have the artist as the Beatles, and
- among objects where the artist is the Beatles, those whose album cover are closest to red have the highest grades.

Further on, let us consider a more difficult example of a conjunction of atomic queries, where more than one conjunct is "nontraditional". An example of this would be the query (Color='red') $\wedge$ (Shape='round').
In this case, we can assume that one subsystem deals with colors, and a totally different subsystem deals with shapes. Let A1 represent the subquery Color='red', and let A2 represent the subquery Shape='round'. The grade of an object x under the query above is the minimum of the grade of x under the subquery A1 from one subsystem and the grade of x under the subquery A2 from the second subsystem.

Once more, Garlic must combine the results from two different subsystems. Let us assume that we are interested in obtaining the top k answers (such as $k = 10$). This means that we want to obtain k objects with the highest grades on this query (along with their grades). If there are ties, then we want to arbitrarily obtain k objects and their grades such that for each y among these k objects and each z not among these k objects, $\mu_Q(y) \geq \mu_Q(z)$ for this query Q.

Following, we will present an obvious naive algorithm [13]:

1. Have the subsystem dealing with color to output explicitly the graded set consisting of all pairs $(x; \mu_{A1}(x))$ for every object x.

2. Have the subsystem dealing with shape to output explicitly the graded set consisting of all pairs $(x; \mu_{A2}(x))$ for every object x.

3. Use this information to compute for every object x:

$$\mu_{A1 \wedge A2}(x) = \min\{\mu_{A1}(x), \mu_{A2}(x)\}$$

For the k objects x with the top grades $\mu_{A1 \wedge A2}(x)$, output the object along with its grade.
For this algorithm, the middleware cost is linear in the database size (the number of objects).

Let us generalize beyond the query (Color='red') $\wedge$ (Shape='round'), which is the conjunction of two atomic queries, and consider conjunctions $A1 \wedge \ldots \wedge Am$ of m atomic queries. An important case arises when these conjuncts are independent (as they are at least intuitively in the above query). We shall be somewhat informal here. The next theorem [17] shows that we can do substantially better than the naive algorithm.

Theorem A: There is an algorithm for finding the top k answers to each monotone query $F_t(A1,\ldots,Am)$, where $A1,\ldots,Am$ are independent, with middleware cost $O(N^{(m-1)/m} * k^{1/m})$ with arbitrarily high probability, where N is the database size. [17]

When the aggregation function is monotone, this theorem applies in particular to the conjunction $A1 \wedge \ldots \wedge Am$ of atomic queries. This includes any aggregation function obtained by iterating triangular norms (such as min), and in fact almost any reasonable choice for evaluating the conjunction. In the case $m = 2$, which corresponds to the conjunction of two atomic queries, the cost is of the order of the square root of the size of the database. By "with arbitrarily high probability", we mean that for every $\varepsilon > 0$, there is a constant c such that for every N, the probability that the middleware cost is more than $c * N^{(m-1)/m} * k^{1/m}$ is less than ε. If A is an algorithm for finding the top k answers to a strict query $F_t(A1, \ldots, Am)$, where $A1,\ldots,Am$ are independent, then for every $\varepsilon > 0$, there is a constant c' such that for every N, the probability that the middleware cost is less than $c' * N^{(m-1)/m} * k^{1/m}$ is less than ε. As a result, we have the following theorem, where as usual Θ means that is a matching upper and lower bound (up to a constant factor).

Theorem B: The middleware cost for finding the top k answers to a monotone, strict query $F_t(A1, \ldots, Am)$, where $A1,\ldots,Am$ are independent, is $\Theta(N^{(m-1)/m} * k^{1/m})$, with arbitrarily high probability, where N is the database size. [17]

The theorem B [17] tells us that we have matching upper and lower bounds for many natural notions of conjunction, such as all triangular norms.

Let us now present an algorithm that meets the conditions of Theorem A. We call this algorithm Ao or **Fagin's algorithm [13]**. This algorithm returns the top k answers for a monotone query $F_t(A1,\ldots,Am)$, which we denote by Q. We assume that there are at least k objects so that "the top k answers" makes sense. Moreover, let us assume that a subsystem i evaluates the subquery Ai. We will present the algorithm informally.

The **Fagin's algorithm** consists of three phases: sorted access, random access, and computation [13].

1. For each i, give subsystem i the query Ai under sorted access. The subsystem I begins to output one by one in sorted order based on grade, the graded set consisting of all pairs $(x; \mu_{Ai}(x))$, where x is an object and $\mu_{Ai}(x)$ is the grade of x under query Ai. Wait until the intersection of the m lists is of size at least k. In other words, wait until there is a set L of at least k objects such that each subsystem has output all of the members of L.

2. For each object x that has been seen, do random access to each subsystem j to find $\mu_{Aj}(x)$.

3. Compute the grade $\mu_Q(x) = t(\mu_{A1}(x),\ldots,\mu_{Am}(x))$ for each object x that has been seen. Let Y be a set containing the k objects that have been seen with highest grades (ties are broken arbitrarily). The output is then the graded set $\{(x, \mu_Q(x)) \mid x \in Y\}$.

Note that the algorithm has the nice feature that after finding the top k answers, in order to find the next k best answers we can "continue where we left off".

Let us now prove the accuracy of the algorithm. Let y be an object that is not seen when the algorithm is running, that is, which is not output by any of the subsystems during sorted access. For each x in L (where as above, L is a set of at least k objects that has been output by all of the subsystems), and for each subsystem i, we know that $\mu i(y) \leq \mu i(x)$: this is because x was output under sorted access by subsystem i while y was not. So by monotonicity of t, we know that $\mu_Q(y) = t(\mu_{A1}(y),\ldots,\mu_{Am}(y)) \leq t(\mu_{A1}(x),\ldots,\mu_{Am}(x)) = \mu_Q(x)$. So there are at least k objects in the output with grades at least as high as that of y.

Since algorithm Ao fulfills Theorem A, it follows from Theorem B that algorithm Ao is optimal (up to a constant factor). In spite of this optimality, there are different improvements that can be made to algorithm Ao (in particular, in the case when t is min, the standard aggregation function in fuzzy logic for the conjunction).

If the aggregation function t is not strict, then Ao is not necessarily optimal. An interesting example arises when t is max, which corresponds to the standard fuzzy disjunction A1∨...∨Am. In this case, there is a simple algorithm whose middleware cost is only mk, independent of the size N of the database. Another example of a nonstrict aggregation function is the median. It turns out that for the median over three attributes, just as in the case of max, there is an algorithm with a middleware cost that beats the lower bound of Theorem B.

3. Threshold Algorithm

R. Fagin, A. Lotem, and M. Naor introduced the Threshold algorithm in the paper "Optimal Aggregation Algorithms for Middleware"[15].

The concept of a query is different in a multimedia database system than in a traditional database system. Given a query in a traditional database system (such as a relational database system), there is an unordered set of answers. On the other hand, in a multimedia database system, the answer to a query can be thought of as a sorted list, with the answers sorted by grade. We shall identify a query with a choice of the aggregation function t. The user is typically interested in finding the *top k answers*, where k is a given parameter (such as $k = 1$, $k = 10$, or $k = 100$). This means that we want to obtain k objects (which we may refer to as the "top k objects") with the highest grades on this query, along with their grades (ties are broken arbitrarily). We will consider k a constant value, and also, we will consider algorithms for obtaining the top k answers.

Other applications. Besides multimedia databases where we use an aggregation function to combine grades, and where we want to find the top k answers, there are other applications. A significant example would be information retrieval, where the objects R are documents, the m attributes are search terms $s_1,\ldots,s_m$, and the grade x_i measures the relevance of document R for search term s_i, for $1 \leq i \leq m$. We will take the choice of the aggregation function t to be the sum. This sum is the total relevance score of document R when the query consists of the search terms $s_1,\ldots,s_m$ is taken to be $t(x_1,\ldots,x_m) = x_1 + \ldots + x_m$.

Another application can be found in the paper written by Aksoy and Franklin about scheduling large-scale on-demand data broadcast[16]. In this particular case, each object is a page and there are two fields. The first field represents the amount of time waited by the earliest user requesting a page, and the second field represents the number of users requesting a page. They make use of the product function t with $t(x_1, x_2) = x_1x_2$, and they wish to broadcast next the page with the top score.

The model. To describe the model, we will assume that each database consists of a finite set of objects. Let us consider N to represent the number of objects. Associated with each object R are *m fields* $x_1,\ldots,x_m$, where $x_i \in [0,1]$ for each i. We may refer to x_i as the ith field of R. The database is consider to consist of m sorted lists $L_1,\ldots,L_m$, each of length N (there is one entry in each list for each of the N objects). Also, we may refer to L_i as *list i*. Each entry of L_i is of the form (R, x_i), where x_i is the *ith* field of R. Each list L_i is sorted in descending order by the x_i value. Since this view is all that is relevant, we take this simple view of a database as far as our algorithms are concerned. We will not take into consideration the computational issues. For instance, in practice it might be expensive to compute the field values, but we ignore this issue here and consider the field values as being given.

We will consider two modes of access to data: sorted access and random access. In the case of a sorted (or sequential) access, the middleware system obtains the grade of an object in one of the sorted lists by proceeding through the list sequentially from the top. Consequently, if object R has the lth highest grade in the ith list, then l sorted accesses to the ith list are required to see this grade under sorted access. Then, in the case of a random access, the middleware system requests the grade of object R in the ith list, and obtains it in one random access. If there are s sorted accesses and r random accesses, then the middleware cost is taken to be $sc_S + rc_R$, for some positive constants c_S and c_R.

Algorithms. There is an obvious naive algorithm for obtaining the top k answers. It looks at every entry in each of the m sorted lists, computes (using t) the overall grade of every object, and returns the top k answers. The naive algorithm has linear middleware cost (linear in the database size), and therefore it is not efficient for a large database.

Another algorithm introduced is the "Threshold Algorithm". We will show that compared with the Fagin's Algorithm, the Threshold Algorithm is optimal in a much stronger sense. We now define this concept of optimality [15].

Instance optimality. Let **A** be a class of algorithms, and **D** be a class of legal inputs to the algorithms. We are considering a particular nonnegative cost measure *cost(A; D)* of running algorithm *A* over input *D*. This cost could be the running time of algorithm *A* on input *D*, the middleware cost incurred by running algorithm *A* over database *D*. We shall mention examples later where *cost(A; D)* has an interpretation other than being the amount of a resource consumed by running the algorithm *A* on input *D*.

We say that an algorithm $B \in \mathbf{A}$ is *instance optimal over* **A** and **D** if $B \in \mathbf{A}$ and if for every $A \in \mathbf{A}$ and every $D \in \mathbf{D}$ we have

$$cost(B, D) = O(cost(A, D)) \quad (1)$$

The above equation states that there are constants c and c' such that $cost(B,D) \leq c^{*}cost(A,D)+c'$ for every choice of *A* and *D*. We refer to c as the *optimality ratio*. This is similar to the competitive ratio in competitive analysis (we will discuss the competitive analysis later on). We call this "optimal" to emphasize that *B* is the best algorithm in **A**.

Aside the worst case or the average case, instance optimality corresponds to optimality in every instance. There are many algorithms that are optimal in a worst-case sense, but they are not instance optimal. An example of this is the binary search: in the worst case, binary search is guaranteed to require no more than log *N* probes, for *N* data items. Nevertheless, for each instance, a positive answer can be obtained in one probe, and a negative answer in two probes.

We will consider a nondeterministic algorithm as being correct if no branch does make any mistake. Then, we will consider the middleware cost of a nondeterministic algorithm to be the minimal cost over all branches where it stops with the top *k* answers. Also, we take the middleware cost of a probabilistic algorithm to be the expected cost (over all probabilistic choices by the algorithm). We say that a deterministic algorithm *B* is instance optimal over **A** and **D**, when we are comparing *B* with the best nondeterministic algorithm, even if **A** contains only deterministic algorithms. The reason for this is because for each $D \in \mathbf{D}$, there is always a deterministic algorithm that makes the same choices on *D* as the nondeterministic algorithm.

We can see the cost of the best nondeterministic algorithm that returns the top *k* answers over a given database as the cost of the shortest proof for that database where these are really the top *k* answers. Accordingly, instance optimality is quite strong. In other words, the cost of an instance optimal algorithm is in fact, the cost of the shortest proof.

Correspondingly, we can view **A** as if it contains also probabilistic algorithms that never make a mistake. For convenience, in our proofs we shall always assume that **A** contains only deterministic algorithms, since the results carry over automatically to nondeterministic algorithms and to probabilistic algorithms that never make a mistake.

Fagin's algorithm is optimal in a high-probability sense (in a way that involves both high probabilities and worst cases under certain assumptions). In comparison, Threshold algorithm is optimal in a much stronger sense. This is instance optimal for several natural choices of **A** and **D**. Specifically, instance optimality holds when **A** is considered to be the class of algorithms that would normally be implemented in practice (since the only algorithms that are excluded are those that make very lucky guesses), and when **D** is considered to be the class of all databases. Instance optimality of Threshold algorithm holds in this case for all monotone aggregation functions. In comparison, high-probability optimality of FA holds only under the assumption of "strictness" (we will define strictness later; this means in fact that the aggregation function is representing some notion of conjunction).

The definition we have given for instance optimality is formally the same definition used in *competitive analysis.*

In competitive analysis, usually, we have the following:
(a) **A** is considered to be the class of offline algorithms that solve a particular problem,
(b) *cost(A; D)* is considered to be a number that represents performance (where bigger numbers correspond to worse performance),
(c) *B* is a particular online algorithm. In this case, the online algorithm *B* is considered to be *competitive*. A competitive online algorithm may perform poorly in some instances, but only on instances where every offline algorithm would also perform poorly.

Another example where we encounter the framework of instance optimality (again without the assumption that $B \in \mathbf{A}$), is in the context of *approximation algorithms*. In this case,
(a) **A** is considered to contain algorithms that exactly solve a particular problem (in cases of interest, these algorithms are not polynomial-time algorithms),
(b) *cost(A; D)* is considered to be the resulting answer when algorithm *A* is applied to input *D*,
(c) *B* is a particular polynomial-time algorithm.

Following, we will present the **Threshold algorithm** [15].

1. Do sorted access in parallel to each of the *m* sorted lists *Li*. When an object *R* is seen under sorted access in some list, do random access to the other lists to find the grade x_i of object *R* in every list *Li*. Then, compute the grade $t(R) = t(x_1,\ldots,x_m)$ of object R. If this grade is one of the k highest we have seen, then remember object *R* and its grade t(*R*) (ties are broken arbitrarily, so that only k objects and their grades need to be remembered at any time).

2. For each list *Li*, let $\underline{x}_i$ be the grade of the last object seen under sorted access. Define the threshold value τ to be $t(\underline{x}_1,\ldots,\underline{x}_m)$. Stop as soon as at least *k* objects have been seen whose grade is at least equal to τ.

3. Let **Y** be a set containing the k objects that have been seen with the highest grades. The output is then the graded set $\{(R, t(R))| R \in \mathbf{Y}\}$.

We will now demonstrate that the Threshold algorithm is correct for each monotone aggregation function t.

Theorem A: *If the aggregation function t is monotone, then the Threshold algorithm correctly finds the top k answers*[15].

Proof: Let **Y** be as in Part 3 of the Threshold algorithm. We need to only show that every member of **Y** has at least as high a grade as every object z not in **Y**. By definition of **Y**, this is the case for each object z that has been seen in running the Threshold algorithm. So, assume that z was not seen. Assume that the fields of z are $x_1, \ldots, x_m$. Therefore, $x_i \leq \underline{x}_i$, for every i. Therefore, $t(z) = t(x_1, \ldots, x_m) \leq t(\underline{x}_1, \ldots, \underline{x}_m) = \tau$, where the inequality follows by monotonicity of t. But, by definition of **Y**, for every y in **Y** we have $t(y) \geq \tau$. As a result, for every y in **Y** we have $t(y) \geq \tau \geq t(z)$, as desired.

Next, we will show that the stopping rule for the Threshold algorithm always occurs at least as early as the stopping rule for Fagin's algorithm (that is, with no more sorted accesses than Fagin's algorithm).

Let us consider the Fagin's algorithm. If **R** is an object that has appeared under sorted access in every list, then by monotonicity, the grade of **R** is at least equal to the threshold value. Therefore, when there are at least k objects, each of which has appeared under sorted access in every list (the stopping rule for Fagin's algorithm), there are at least k objects whose grade is at least equal to the threshold value (the stopping rule for the Threshold algorithm).

This suggests that for every database, the sorted access cost for the Threshold algorithm is at most the cost of Fagin's algorithm. However, since the Threshold algorithm may do more random accesses than Fagin's algorithm, this does not imply that the middleware cost for the Threshold algorithm is always at most the cost of Fagin's algorithm. On the other hand, since the middleware cost of the Threshold algorithm is at most the sorted access cost times a constant (independent of the database size), it does imply that the middleware cost of the Threshold algorithm is at most a constant times that of Fagin's algorithm. We will show that under natural assumptions, the Threshold algorithm is instance optimal.

We will consider the intuition behind the Threshold algorithm. We will first discuss the case where $k = 1$, that is, where the user is trying to determine the top answer. Let us assume that we are at a stage in the algorithm where we have not yet seen any object whose (overall) grade is at least as big as the threshold value τ. At this point, the intuition is that we do not know the top answer, since the next object we see under sorted access could have overall grade τ, and hence bigger than the grade of any object seen so far.

In addition, once we see an object whose grade is at least τ, then it is safe to stop, as we see from the proof of Theorem A. Therefore, intuitively, the stopping rule of the Threshold algorithm states: "Stop as soon as you know you have seen the top answer." Similarly, for general k, the stopping rule of the Threshold algorithm states: "Stop as soon as you know you have seen the top k answers." Moreover, "Do sorted access (and the corresponding random access) until you know you have seen the top k answers".

More generally, we can view the Threshold algorithm as saying: "Gather what information you need to allow you to know the top k answers, and then stop".

These "programs" can be viewed as being very high-level, "knowledge-based programs". In fact, the Threshold algorithm can be viewed as being "designed" by thinking in terms of these knowledge-based programs.. When we consider the case where random accesses are expensive relative to sorted accesses, but are not forbidden, we need an additional design principle to decide how to gather the information, in order to design an optimal algorithm.

The next simple theorem (theorem B) gives a useful property of the Threshold algorithm that distinguishes the Threshold algorithm from Fagin's algorithm.

Theorem B: *The threshold algorithm requires only bounded buffers whose size is independent of the size of the database* [15].

Proof. Aside of little bit of bookkeeping, all that the Threshold algorithm must remember is the current k top objects, their grades, and the pointers to the last objects seen in sorted order in each list.

In comparison, Fagin's algorithm must remember every object it has seen in sorted order in every list, in order to check for matching objects in the various lists. Therefore, Fagin's algorithm requires buffers that grow arbitrarily large as the database grows.

The bounded buffers have the following disadvantage: in order to find the grade of the object in the other lists, for every time an object is found under sorted access, the Threshold algorithm may do m-1 random accesses (where m is the number of lists). This is in spite of the fact that this object may have already been seen under sorted or random access in one of the other lists.

4. Comb Algorithm

In this project, we propose a different approach for queering a multimedia database. The **Comb algorithm** will be introduced.

4.1 Introduction and presentation of the systems.

Before presenting any new method for queering a multimedia database, we will analyze what are the recent capabilities and constraints of the present systems. As we mentioned in the previous descriptions of the multimedia data, the systems that can hold and handle this kind of data for queering by content have to be multidimensional (for example, the R-tree). The R-tree will consist of organizing the storage of the objects represented as feature vectors, and it will manage the insertion, query, and update of an object into the database.

Therefore, the objects represented by a vector of features can be visualized as points in the feature hyperspace (**F**). That means that each feature stands for one dimension of **F**.

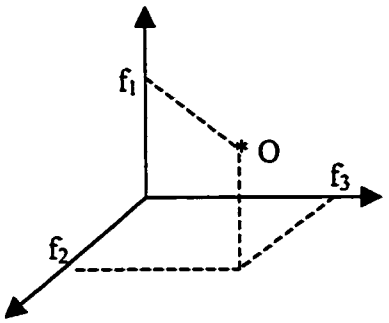


Fig.7 Representation of the object O in the three-dimensional feature space. The features of object O are (f_1, f_2, f_3).

Given a query point and a query by content, our system should be capable of retrieving the near-by objects that are similar to our given query object with respect to the features that characterize each object of the database. The similarity between two objects can be stated as the Euclidian distance between two points in the hyper dimensional space **F**, where each point stands for an object definition. We say that object O1 is more similar to object O than object O2 if the Euclidian distances in the hyperspace F of the objects (O, O1, O2) representation is as follow: distance D1 is smaller than D2 (where **D1** is the Euclidian distance between O and O1, and **D2** is the Euclidian distance between O and O2).

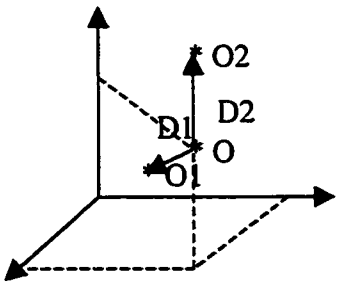


Fig.8 Similarity between objects. O1 is more similar to O than O2 is to O($D1 \leq D2$).

Content base queries are represented by similarity retrievals in the feature hyperspace. For example, given a multimedia database of landscapes, our query by content requires the retrieval of all the images that include a "sunset by the beach with birds". Each photo has to be represented by a feature vector. The dimensions in this vector are computed through a particular method that will define the picture in a specific way for a set of queries.

Considering that the answer of a query by content is subjective to the transformation of the object into a vector of features, finding one object from a content base query wouldn't be enough to give an accurate answer. Therefore, the query for similar images is performed such as the answer is a set of objects and in this set, a subset will be chosen by human analysis.

The retrieval of similar objects to an object O is performed step-by-step as follow:

In the first step, we query the nearest similar object to O, and we retrieve the nearest object in a hyperspace F with respect to the object O representation in F, Q. The result is O1, which is placed in a set S, and the vector in F corresponding to O1, Q1.

To retrieve the next nearest object, we query the nearest object to O disregarding all the objects that are in set S.

We will repeat this second step until the number of objects that set S contains is equal to the number of objects that were initially proposed, or we will perform a stopping condition on the elements of the set S at each retrieval (Fagin and Threshold algorithms are using such stopping conditions for k nearest retrieval method).

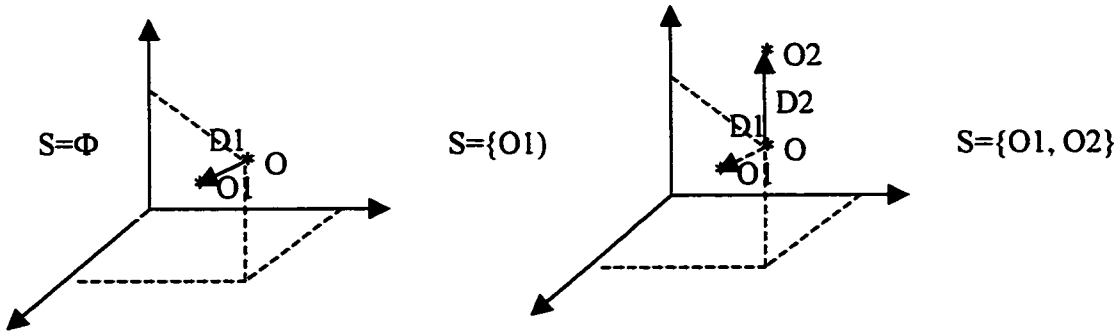


Fig.9 The retrieval of similar objects. S is the set of retrieved objects.

We can notice that each retrieval can be considered as a step in a query process. For example, by using the R-tree structure and performing a nearest-neighbor query as presented in the introduction section, the nearest object to a query Q implies the time to search the R-tree structure and to retrieve from the secondary storage the object itself. These two tasks are the basic components of a step retrieval, which we name step entity.

A major concern in the similarity search queries is the number of steps that the query has to perform for retrieving the k nearest neighbors to a query point Q.

The dimension of the feature vector is relevant for the accuracy of the object representation in the hyperspace F. The larger the vector is defined, the more precise the definition of the object is, and the more diverse the query possibilities are. To capture the complexity of an object into a vector through transformation of the object's features into a vector called feature vector, it is recommended to use a minimum optimal vector size. That means that the specific queries cannot be performed on a set of objects if these feature requirements are not met. Consequently, given a set of objects and a set of queries, there is a minimum vector size that will include the transformed features of the objects, and will perform the specified set of queries. A simple example is in Fig. 10.

(a) (b)

P = (p1, p2, ... , pn)

Fig.10 (a) The object O; which is a picture in this example.
(b) The vector P that represents the picture O. Can be the color histogram.

In our project, the method used for indexing multimedia objects is the R-tree. As we mentioned in the R-tree section, the size of the feature vector that can be used with R-tree method and has reasonable performance is 20 to 30 dimensions. The query time is increasing exponentially with respect to the size of the feature vector. More precisely, if the feature vector increases considerable, given the application requirements, the query time increases exponentially with the complexity. Query performance will be so bad that for some applications using sequential scanning of the entire database, would be less expensive (in terms of query time) than using the R-tree method. Consequently, there are two options:

1) Perform sequential scanning, which for huge databases it is almost impossible to use; the time cost is unrealistic.
2) Divide the feature vector into several smaller vectors; for example:
   O(x1, x2, x3, x4, x5) = {O1(x1, x2); O2(x3, x4, x5)}; and use the R-tree indexing method to include each smaller vector into a separate R-tree structure such that instead of having a single R-tree, we will have several R-trees defining the set of objects of the database. The resulting R-trees can be considered as subsystems that together comprise the same information as the original system. For example, the system S, that has defined an R-tree R, is equivalent to a set of subsystems S1,..., Sn with the respective R-trees R1,..., Rn.
   In Fig 11 we provide a example of a system S which is equivalent to S1,S2, and S3.

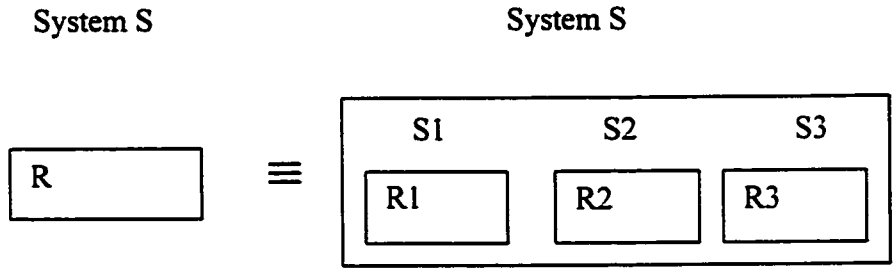


Fig.11 Systems equivalence.
(R, R1, R2, R3 are R-tree indexing structures for system S, subsystem S1, S2, S3)

The traditional method for searching a multimedia object for a query by content is to transform the objects into features vectors and to store them into an R-tree index.

In the case of dealing with multiple R-trees that refer to the same set of objects, we need to find another approach and solution. Fagin's method that we mentioned in the previous sections, is the first solution dealing with this new imposed indexing strategy. As presented, Fagin's algorithm performs the search for multiple systems. Moreover, this has a deterministic approach for the retrieval of the *k* nearest neighbors.

Fagin's algorithm principle is to search for k common objects in each subsystem, and then, stop the query. An analysis of the k objects is performed and if the results are unsatisfying, further nearest neighbor retrievals are performed to meet the correctness and the quality of query. The quality of query can be defined as the best object or set of objects that represent strong similarity to the query object. In Fig.12 we give an example of Fagin's algorithm for a system composed of three subsystems S1, S2, and S3. The method has to retrieve the first nearest neighbor.

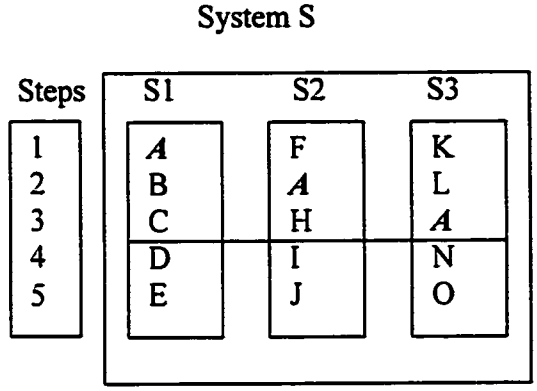


Fig.12 Fagin's algorithm for system S composed of subsystems S1, S2, and S3. K=1, FA returns object *A* after 3 steps.

Fagin's order of complexity with respect to the number of steps retrieving the k first nearest objects from several subsystems is increasing with the number of subsystems that constitute the original system. The cost is arbitrarily high, and it can be as high as reading a number of steps equal to the number of objects of the multimedia database located in each subsystem. Of course, the cost for entirely reading a subsystem is equal to the cost of reading all the subsystems because the searches are performed in parallel on each subsystem. For example, at step number 1, which can be considered time entity one, the retrieval is performed in each system in parallel, and the result is a set of objects representing the nearest neighbor points in each subsystem with respect to the query point. For step number 2, we perform the nearest neighbor query again, and we retrieve a second set of objects. From every subsystem, we retrieve one and only one object as in step no.1. Further on, we continue the retrieval until the stopping conditions are met. As defined before, the stopping conditions involve computations on the feature vector of the objects.

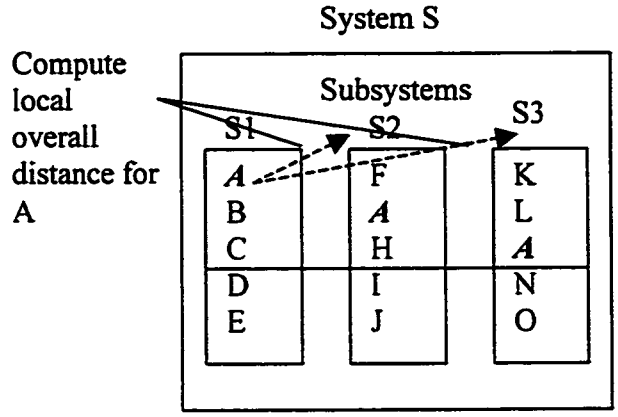
In this project, we adapted **Fagin's algorithm** by considering the distance between objects as being Euclidian. Therefore, after obtaining the k common elements, we compute the overall Euclidian distance from our retrieved points to the query point. This involves the following: for every object retrieved from a subsystem, we have to do a random access to all the other subsystems. The reason of this is to compute the local overall distance from that point to the query point with respect to those subsystems, in order to be able to evaluate the total overall distance. Once the overall distances of the retrieved points from the set are computed, we can order these objects in descending order and select the first k objects.

Query point:
$Q(q_{x1}, q_{x2}, q_{x3}, q_{y1}, q_{y2}, q_{y3}, q_{z1}, q_{z2}, q_{z3})$

Object A:
$A(a_{x1}, a_{x2}, a_{x3}, a_{y1}, a_{y2}, a_{y3}, a_{z1}, a_{z2}, a_{z3})$

System S:
S(x1, x2, x3, y1, y2, y3, z1, z2, z3)

Subsystems:
S1(x1, x2, x3)
S2(y1, y2, y3)
S3(z1, z2, z3)



Local overall distances for object A:
For subsystem S1: $DS1(A)=\text{sqrt}((a_{x1}-q_{x1})^2+(a_{x2}-q_{x2})^2+(a_{x3}-q_{x3})^2)$
For subsystem S2: $DS2(A)=\text{sqrt}((a_{y1}-q_{y1})^2+(a_{y2}-q_{y2})^2+(a_{y3}-q_{y3})^2)$
For subsystem S1: $DS3(A)=\text{sqrt}((a_{z1}-q_{z1})^2+(a_{z2}-q_{z2})^2+(a_{z3}-q_{z3})^2)$

Overall distance $D(A) = \text{sqrt}((DS1(A))^2 + (DS3(A))^2 + (DS3(A))^2)$

Fig.13 Set of retrieved objects {A, B, C, F, H, K, L}, for k=1. Compute the local overall distance for example of object A in system S1 and S3 (for all the objects in the set of retrieved objects).

In conclusion, Fagin's method using several subsystems is useful when the number of subsystems is limited; therefore, the complexity of the feature vector is reasonable.

An experiment is explained and conducted in the next section. This experiment describes a system comprising 11 subsystems and the multimedia feature data that is uniformly distributed in the respective space domain. The test performance is presented and explained.

Another algorithm that copes with multiple subsystems is a derivative of the Fagin's method, and it is called **Threshold algorithm** (this was previously presented in section 3).

The **Threshold algorithm** can be deterministic or heuristic. As a deterministic algorithm, Threshold method is similar to Fagin's method. This means that the method is applied to a system S composed of several subsystems S1, S2,...,Sn as presented for Fagin's algorithm. The index structures corresponding to the subsystems are similar to the one introduced for the Fagin's algorithm. Therefore, for Threshold algorithm, we use R-trees as indexing structures, and the query is performed step by step in parallel in the subsystems.

This means that at every time step, the retrieval of one object from every subsystem is performed with respect to a nearest neighbor query.

The difference between the two algorithms Threshold and Fagin is the stopping condition. We will explain how the Threshold works for the settings of our project. We use R-tree structures in subsystems for the retrieval of objects in the nearest neighbors query. The Threshold algorithm performs the following tasks:

The algorithm retrieves in the first step one nearest object from each subsystems, includes the retrieved objects in a set S, and then, it computes the overall distances between the query and the retrieved objects. In order to compute the overall distance between an object and query point, we need to do random access from a given object to all the other subsystems. From the set S, we select the object that has the minimum overall distance, and we call it Dmin.

e.g. System S contains S1, S2, S3. For all the retrieved objects (A) compute the following:
Local overall distances for object A to Q (query point).
For subsystem S1: $DS1(A)=\text{sqrt}((a_{x1}-q_{x1})^2+(a_{x2}-q_{x2})^2+(a_{x3}-q_{x3})^2)$
For subsystem S2: $DS2(A)=\text{sqrt}((a_{y1}-q_{y1})^2+(a_{y2}-q_{y2})^2+(a_{y3}-q_{y3})^2)$
For subsystem S3: $DS3(A)=\text{sqrt}((a_{z1}-q_{z1})^2+(a_{z2}-q_{z2})^2+(a_{z3}-q_{z3})^2)$
Overall distance to Q $D(A) = \text{sqrt}((DS1(A))^2 + (DS3(A))^2 + (DS3(A))^2$
Dmin = min ({Q D(A)}) ; minimum overall distance of all the objects retrieved.

We repeat step first step until the stopping condition presented is met.

Stopping condition: For each subsystem *i*, we can assign a set of objects that where retrieved from that subsystem, which we call subsets "Si". For each subset, we compute the local overall distance from the objects in the subset to the query points with respect to the subsystem that corresponds to the subset. Then, we select an object from that subset that has the minimum local overall distance. We do this for all the subsystems of the multimedia database. Therefore, we will obtain as many selected values with the above property as the number of subsystems. This set of values will help us to compute quite easily a value "t", which is called the Threshold value. The Threshold value in this project is defined as the square root of the sum of the square of the selected values from the subsystems. Consequently, the stopping condition is met if at a certain step the minimum overall distance is less or equal than the Threshold value, "t". In Fig. 14 the threshold boundary is represented for a system S composed of two subsystems S1 and S2. In Fig. 15 we give an example of the threshold calculation.

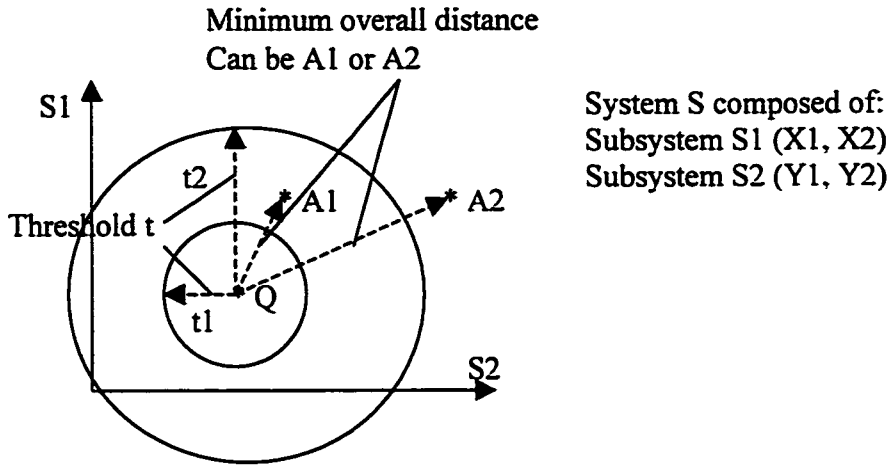
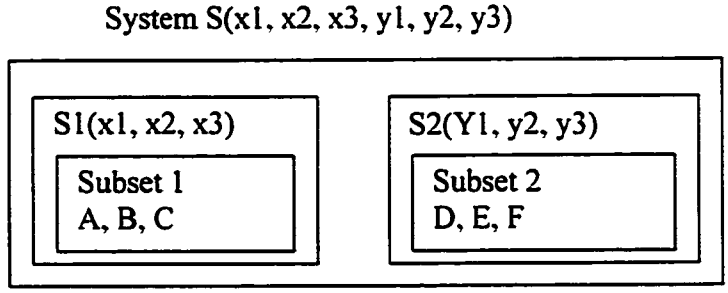


Fig. 14 Threshold algorithm. If minimum overall is A1 and t = t1, then algorithm continues. If t = t2, then algorithm stops. If minimum overall is A2, then algorithm continues for both threshold values t1 and t2.



For subsystem S1 the minimum local overall retrieved is for object B:
$DS1(B) = sqrt((b_{x1}-q_{x1})^2+(b_{x2}-q_{x2})^2+(b_{x3}-q_{x3})^2)$

For subsystem S2 the minimum local overall retrieved is for object F:
$DS2(F) = sqrt((f_{y1}-q_{y1})^2+(f_{y2}-q_{y2})^2+(f_{y3}-q_{y3})^2)$

Threshold $t = sqrt((DS1(B))^2+(DS1(B))^2)$

Fig.15 Threshold calculation.

The Threshold algorithm can be easily modified in order to obtain k first objects retrieved instead of one. This can be done in the following way: instead of memorizing into a buffer one value of the object that has the minimum overall distance, we record in this buffer the first k objects with the minimum overall distances.

The stopping condition will be performed such that the threshold value "t" is greater or equal than any of the k values selected in the buffer. The algorithm can be transformed into a heuristic one if the Threshold value is adjusted by a constant ε that will determine a

premature algorithm stopping. This means that the k values will be with a certain probability found among the retrieved objects.

In the section "Experiments", we present a model of a system composed of 11 subsystems and with the multimedia data features uniformly distributed. Moreover, we compare this model with the Fagin's algorithm.

We have presented two of the well-known algorithms (Fagin and Threshold) that can perform searches in a multimedia system composed of several subsystems. In short, these two algorithms have many common features as:

- They use the given set of subsystems;
- They use a unique query point;
- The cost of nearest neighbors query is directly proportional with the number of steps used in a deterministic or heuristic approach (in the Threshold case);

The difference between the two methods is the number of steps that each algorithm has to perform in order to successfully access the k first objects in the nearest neighbors query. Threshold is performing better than Fagin's algorithm, given a multimedia database and a query point for the comparison of the two methods. The two methods are compared in a realistic experiment in the Experiments section (section 5).

4.2 Presentation of the Comb Algorithm.

All the experiments that we conducted and all the modeling of new possible algorithms were performed with the idea of improving the two well-known methods.

The idea of the Comb algorithm is to multiply the multimedia system and to assign a query point different for each copy of the system. The Comb algorithm is heuristic. We will notice a great improvement for the query by content compared to the search using one system that we call **sequential search**. Sequential search is use in Fagin's and Threshold algorithm.

In other words, the Comb system and algorithm will be able to retrieve more accurate objects with respect to the overall distance from the query point to those objects in a given number of steps (considering the resources available) than any other systems implemented so far.

Fagin and Threshold methods are using one system, which is the original one, and one query point, which is the given one. The search is performed as the nearest neighbor query. The nearest objects to the query point are retrieved in a given subsystem one by one so that we can assure that between two consequent retrievals of objects with the overall distance with respect to the query point (D1 and D2), there is no object in the multidimensional space of that subsystem having the distance D that falls between D1 and D2.

Before giving the definition of the comb algorithm, let us take a look at following options. Let us consider a system that has an R-tree indexing structure. If the feature vector is too wide, then, the time constraint to perform a query on this database in not insured. One possibility to solve this problem is to perform a sequential reading of all the objects of the database, which let's say is accomplished in time "T". If we have "n" computing resources available, we can divide this database among the "n" resources and perform the sequential reading of all the objects. The reading time will be roughly T/n. In this case, we notice a real improvement from reading with only one computational system. This improvement is "n" times faster (Fig. 16).

However, even though the time is considerably improved, the time to read one subsystem is still very big. What we have to remember is that by distributing on parallel systems our database (parallel computing resources), we can improve on the search time. Our Comb algorithm is based on this feature.

Also, it is to be noticed that Fagin and Threshold are using only one query point. In the case of multiple systems working together, if the query point will be the same for all systems, then, the results will be as following:

a) If the systems are identical copies of the original one, then, at each step of the query process, the systems will behave identically. This means that the objects retrieved at a certain step by one system are exactly the same objects retrieved by any other copy of the system, which is obvious. Therefore, if we want to distribute the objects on parallel systems using copies of the system, we will not use the same query point for all the copies. The distribution of the query points is heuristic and, in the Comb algorithm, it is chosen in a way that is covering the vicinity of the query point. The placement of the query points in the parallel and identical systems can be static or dynamic. Based on this observation, we developed a query method, which we will call Comb algorithm. Fig. 17 is presenting three systems S, S1 and S2 working in parallel. S1 and S2 are copies of S. The query points are different for the three systems.

b) If our database is distributed among several computational systems, then, the query point can be the same or can be different for any other system that comprises a disjunctive set of objects of the database.

If the query point is the same, the objects retrieved at a certain step of the query process in a system are different from any other objects retrieved by the other systems at that step. The explanation is that the sets of objects that belong to the subsystems are disjunctive sets. This means that the intersection of these sets is null; there is no common element for these sets. In this case, we haven't studied the behavior of such system; neither the case of a different query point allocated to each system.

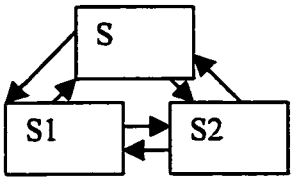


Fig.16 Instead of having S system we have S, S1, S2 computational systems. They all have the same capacity and performance (computing power).

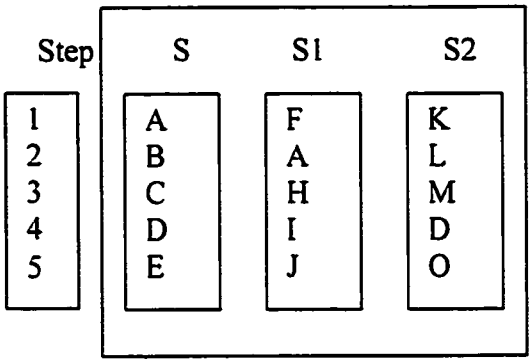


Fig.17 Parallel retrieval of objects in nearest neighbor query. Systems S, S1, S2 are working in parallel.

Based on the observation on point a), we have deducted through trials and errors a heuristic distribution of the query points. Given a set of multimedia objects, we use the R-tree indexing method to create a multimedia searchable database distributed over several subsystems as in the Fagin's algorithm. This distribution is necessary because of the curse of dimensionality that the R-tree indexing and all the other multidimensional indexing methods are suffering.

The resulting subsystems that hold and manage data are S1,..., Sn. Given a query point Q and a number of steps "st", we can define all the requirements such that the Comb algorithm will work. If *k* computational and storage resources are available, we make identical copies of the original system into the k resources. Therefore {S(m)| S(m)= {Sm1,..., Smn}; for any m less or equal than *k*} is the new system that we call **Comb system.** In Fig. 18 we provide an example of a Comb system made of the original system S and two copies S(1) and S(2).

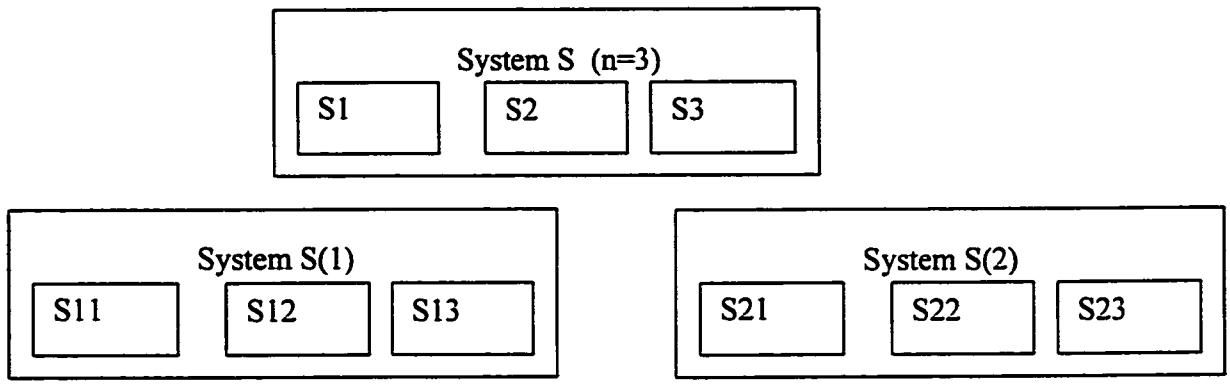


Fig.18 System configuration for Comb Algorihm. S(1) and S(2) are copies of S. No of resources k=2; No of steps = st.

The **Comb query points** corresponding to each system will be computed as follow:

a) The query point for the original system is:

Q = (q1,...,qr), where "r" is the total size of the feature vector of an object. and it's equal to the sum of the dimensions of the subsystems.

b) The query points for the identical system m (m∈{1,...,k} we have a total of k copies):

Q(m) = Q(1 + C(m) * ε), where C(m) =((-1)^m)*floor(m/2) for any m∈{1,...,k}; ε is a vicinity relative distance on a certain feature dimension around the query point Q. Therefore $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_r)$.

For easy comprehension, we will provide the example from Fig. 18:

In system S we have the subsystems: S1(x1, x2, x3), S2(y1, y2,y3), S3(z1, z2, z3)
In system S (1) we have the subsystems: S11(x1, x2, x3), S12(y1, y2,y3), S13(z1, z2, z3)
In system S (2) we have the subsystems: S21(x1, x2, x3), S22(y1, y2,y3), S23(z1, z2, z3)
$Q = (q_{x1}, q_{x2}, q_{x3}, q_{y1}, q_{y2}, q_{y3}, q_{z1}, q_{z2}, q_{z3})$, r = 9.
$\varepsilon = (\varepsilon_{x1}, \varepsilon_{x2}, \varepsilon_{x3}, \varepsilon_{y1}, \varepsilon_{y2}, \varepsilon_{y3}, \varepsilon_{z1}, \varepsilon_{z2}, \varepsilon_{z3})$ is a constant.
C (1) = 0, C (2) = 1, C (3) = -1. Therefore Q (1) = Q+ε and Q (2) = Q-ε.

We will now represent Q, Q (1) and Q (2) in the subsystem S1 (same representation is for S2 and S3) in Fig. 18 a.
$Q(1) = (q_{x1}+\varepsilon_{x1}, q_{x2}+\varepsilon_{x2}, q_{x3}+\varepsilon_{x3})$;
$Q(2) = (q_{x1}-\varepsilon_{x1}, q_{x2}-\varepsilon_{x2}, q_{x3}-\varepsilon_{x3})$;

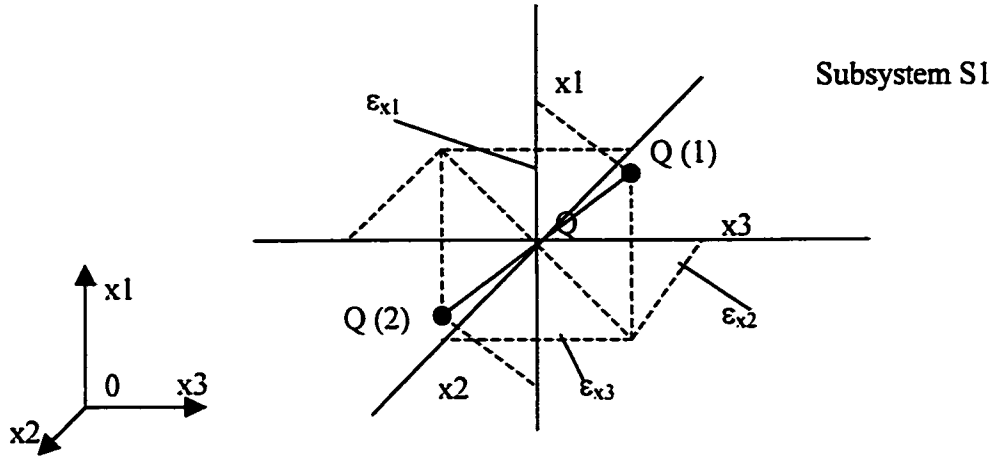


Fig. 18a. The query points corresponding to subsystem S1

We will now represent Q, Q (1) and Q (2) projected on the subsystems in Fig 19.

We define the projection of Q(1) on S1 with respect to Q the distance between Q and Q(1) in system S1. We perform the same projections for all the query points and on all the subsystems.

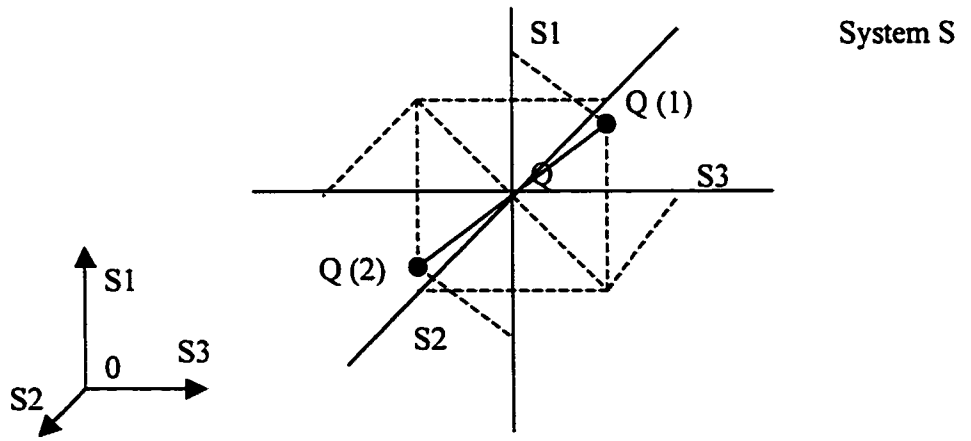


Fig.19 Query points in the Comb algorithm.

Static Comb Algorithm:
Considering the **Comb system**, which includes the original system and the copies of the original system and the **Comb query points** corresponding to each copy, we will perform a nearest neighbor query in a given number of steps. The search is to be performed in each identical system and in parallel. This type of query is called Static Comb Algorithm.

We will see in the Experiments section (section 6) what important advantages we have from using this algorithm in comparison to a sequential retrieval of nearest neighbors in a pre-imposed number of steps (Sequential retrieval was defined previously – it is the type of retrieval used in Fagin's and Threshold Algorithm).

While the Comb system composed of multiple identical systems is retrieving objects, the retrieved objects are recorded in a common buffer zone for all the identical systems. If one object has been retrieved by an identical system no other identical system could retrieve the same object. The Comb system will be designed in such a way that for every retrieval, the buffered common list of the retrieved objects is checked during the nearest neighbor query retrieval. If an identical system retrieves an object already in the buffer list of the retrieved objects then the identical system will perform another nearest neighbor search until a valid object is retrieved. A counterexample is presented in Fig. 20. The experiment shows that the collisions between identical systems for the retrieval of the same objects are few. This is because the query points are apart from each other. The more objects the database has the less collisions occurs.

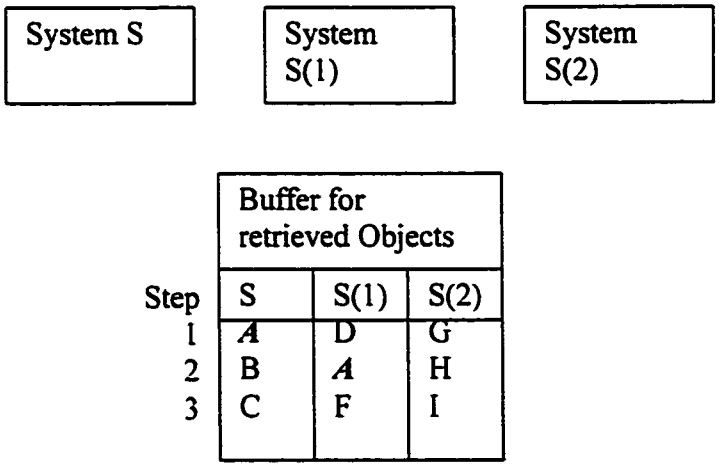
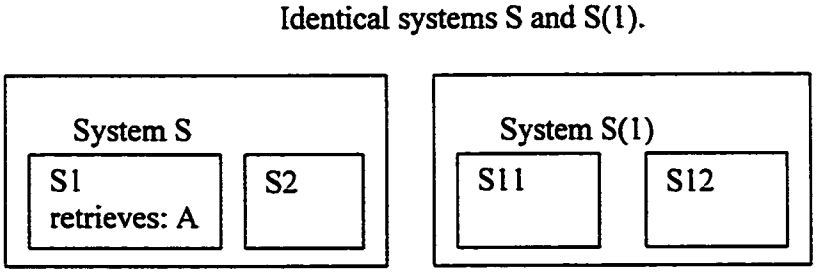


Fig.20 Buffer of objects retrieved from all the identical systems.
A cannot retrieved by S(1).

After finishing the preset number of steps, all the retrieved objects are evaluated using the random access procedure to subsystems, which can be done in parallel when querying the nearest neighbor on the identical systems.

For this purpose, we can use a B+-tree indexing that will provide us with all the unknown feature dimensions of an object retrieved in a nearest neighbor query that are not provided by a specific subsystem of an identical system (Fig. 21).



Local overall distances from object A to Q (query point) in subsystem S1:
For subsystem S1: $DS1(A)=sqrt((a_{x1}-q_{x1})^2+(a_{x2}-q_{x2})^2)$

Do random access and retrieve from subsystem S2:
$A(a_{y1}, a_{y2})$ retrieved from a B+ tree index.
For subsystem S2: $DS2(A)=sqrt((a_{y1}-q_{y1})^2+(a_{y2}-q_{y2})^2)$
Overall distance to Q $D(A) = sqrt((DS1(A))^2 + (DS3(A))^2$

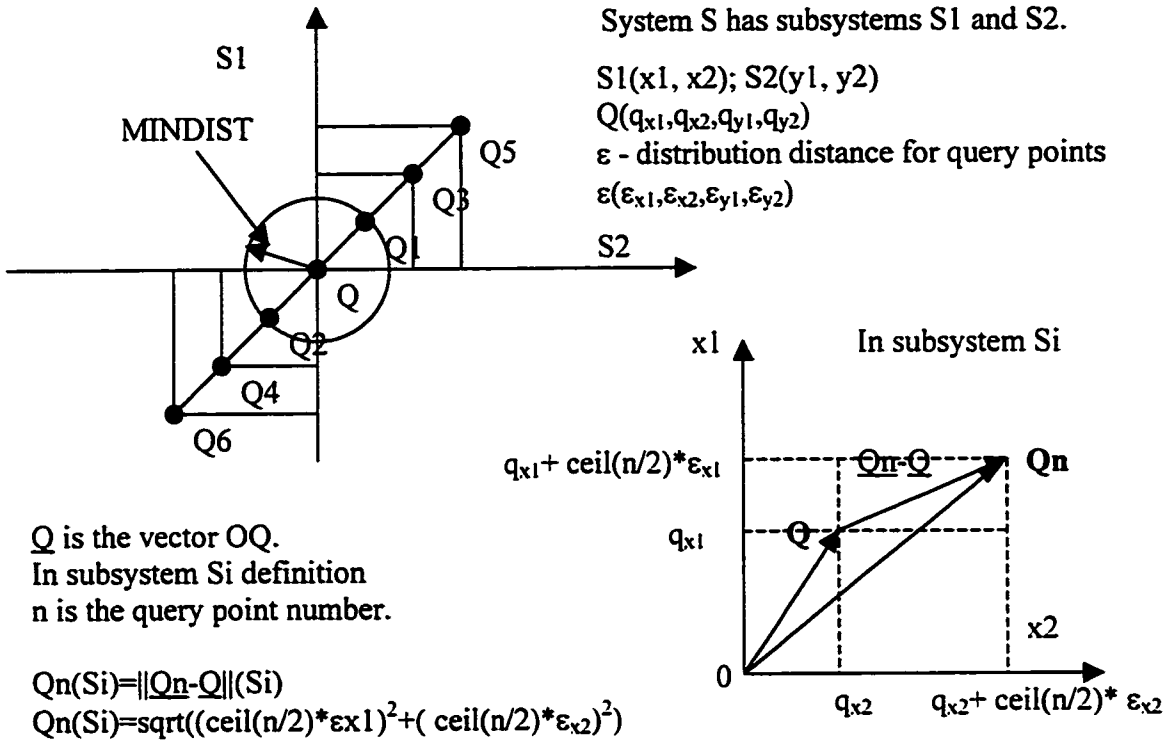
Fig.21 Random access using B+ tree.

In conclusion, the **Static Comb algorithm** is performing the search in a predefined number of steps in a parallel multi-identical system with heuristic positioning of the query points for each copy of the original system. The Comb algorithm is also overlapping the random access retrieval and computation time with the nearest neighbor query steps. In the Experiments section 5, we will see how this algorithm performs, and what improvements it brings to any algorithm using the original system with one query point.

Another version of this algorithm is the **Dynamic Comb algorithm**. The Dynamic Comb algorithm has the same system structure as the static one, which is represented by identical copies of the given system as many as the number of query points. The difference between the Static and the Dynamic Comb algorithms is the positioning of the query points during the nearest neighbor query process. The query points are initially defined in a similar way as in the Static Comb algorithm. During the query process, if some conditions based on the retrieved objects are met, the query points start traveling or relocating to a new position near to the original query point. The query points that have the potential to travel are the furthest from the original query point. More precisely, at every step of the query, the retrieved objects will be placed in a common set of retrieved objects. In this set, we will compute for every object the overall distance between the original query point and the object. Therefore, this set will be ordered in an ascending order with respect to the above-mentioned overall distance.

We will select the element that has the minimum computed distance (MINDIST).

We will perform the following checking and operation on every step. If all the local overall distances (corresponding to every the subsystems that comprise the original system) between the second to last furthest query point with respect to the original query point and the original query point Q are greater or equal than MINDIST, then, we relocate the furthest query point to a new position. Let us give an example in Fig. 22.



Q is the vector OQ.
In subsystem Si definition
n is the query point number.

$Qn(Si)=||\underline{Qn}-\underline{Q}||(Si)$
$Qn(Si)=sqrt((ceil(n/2)*\varepsilon x1)^2+(ceil(n/2)*\varepsilon_{x2})^2)$

If MINDIST ≤ Q3(S1) AND MINDIST ≤ Q3(S2) then relocate Q5.

Fig.22 Dynamic Comb algorithm – finding the query point to be relocated.

The relocation of a query point can be done using different scenarios. After many experiments we have found that moving the query points towards the initial query point increases the efficiency of retrieval. The last query points will be relocated such way that their distribution will be near to uniform without relocating the rest of the query points. We don't relocate all of them because those one that are within the MINDIST hyper-circle are still retrieving useful neighbors. Therefore we shall bring a query point in the middle of two successive query points that have not received a query point between them in the previous relocation. The first two query points are Q and Q1. It continues in ascending order on the positive axis. Next, it would be Q1 and Q3. When it is reaching a query point that is a neighbor to a relocate one, we will reassign the indices of the positive query points in ascending order from the nearest to Q to the furthest. Then it will start from the beginning with Q and Q1.

We have mentioned only the positive axis because the query points are symmetrically distributed with respect to Q. The relocation that we do for the positive query points, we do it for the query points that have negative values (considering $\varepsilon > 0$ stands for each element of the vector > 0). At every step we search for all the possible way for relocating the query points. This is possible because the relocation is computed in the main memory. Let us see how it occurs. An example for query points relocation is given in Fig. 23.

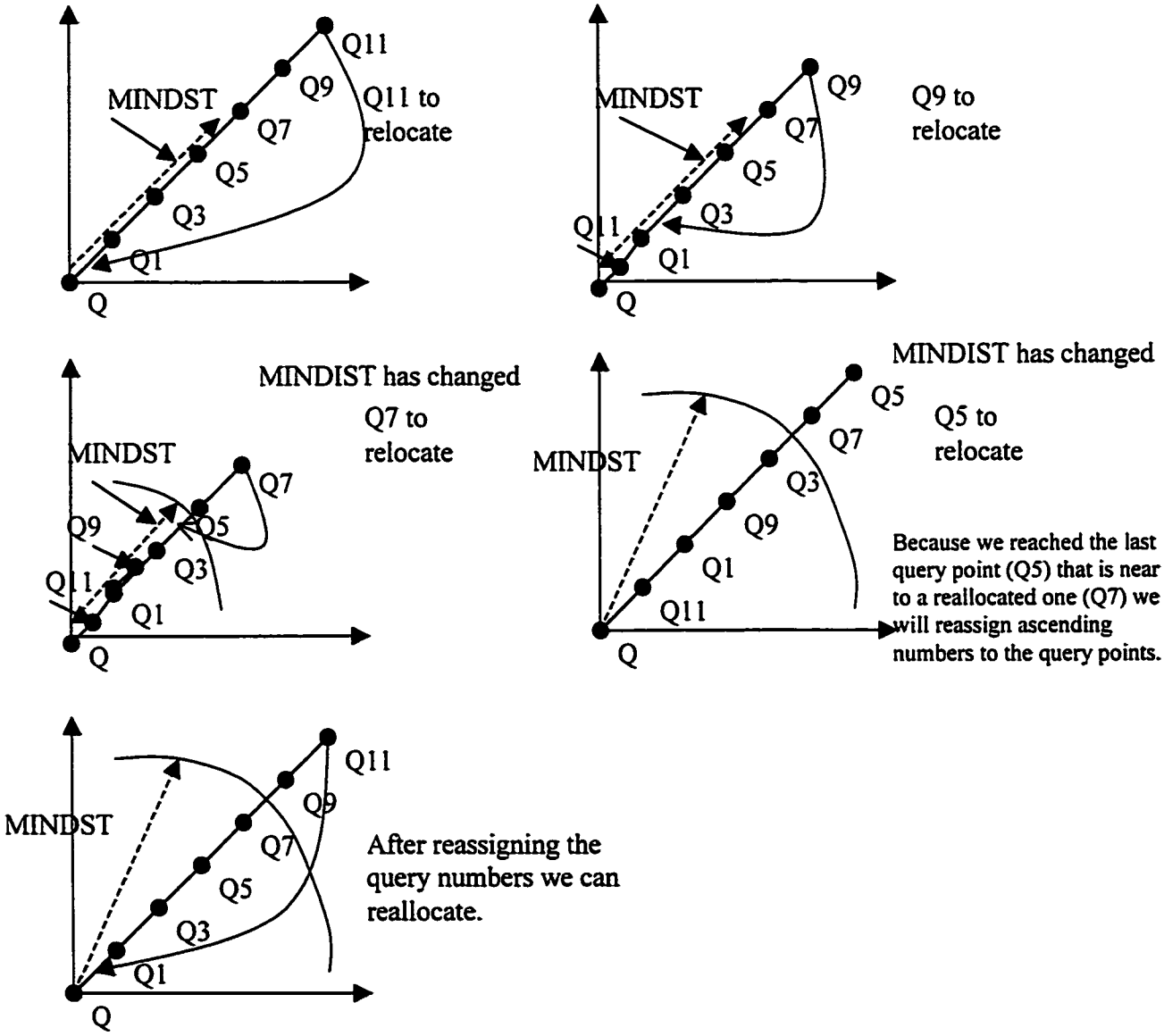


Fig.23 Reallocation of the query points.

We have now an algorithm that retrieves nearest neighbors from a distributed system using a parallel algorithm. In the next section we have some experiments comparing Fagin, Threshold, Comb Static and Dynamic algorithm.

5. Experiments

In this section we perform our experiments on some synthetic data sets. The multimedia data objects have the feature vectors distributed uniformly in the feature data space. The dimensions of the feature vector as random variables are independent. In other words, if any object O is represented by the vector F (x1, x2, x3, x4), then x1, x2, x3, x4 are distributed uniformly. Moreover, x1, x2, x3, x4 as random variables are independent. There is no correlation between x1, x2, x3, and x4.

We will start the experiment with a system simulation of 11 subsystems. We will try this system with Fagin and Threshold algorithm. Next, we will run the Comb algorithm, static and dynamic, versus sequential retrieval of nearest neighbors using the original system, which is composed of 4 subsystems (we will call **sequential retrieval** the retrieval mode used in Fagin's and Threshold algorithm). In this case, the objects' features will be also as random variables uniformly distributed and independent.

The following experiment is using, as we mentioned, a system made of 11 subsystems and a database of 2000 multimedia objects. The query used in this experiment is the retrieval of the first nearest neighbor using Fagin's and Threshold algorithm. We record the number of steps to retrieve the nearest neighbor for different query points for Fagin and Threshold algorithm. Also, for each query point we recorded the step when the nearest neighbor was found using the sequential retrieval. We will call it Minstep. We run the experiment 500 times in order to draw the distribution of the number of step. There will be 3 distributions Fagin, Threshold and Minstep. In the following figure we have the 3 distributions:

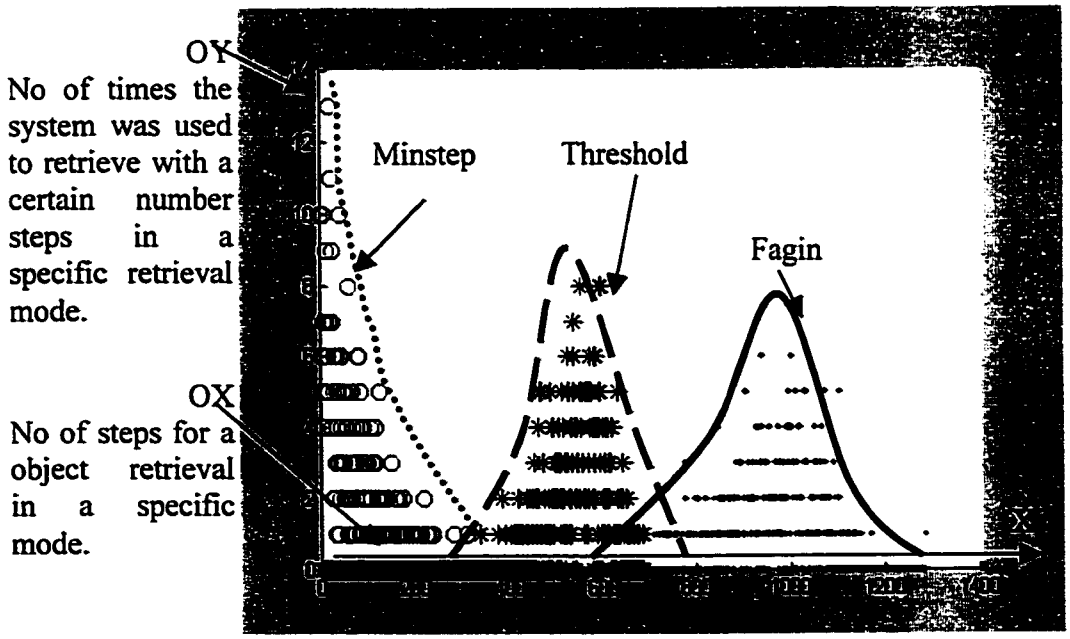


Fig. 24 Distributions of Fagin, Threshold, and Minstep.

The entire experiment is repeated 500 times using the same distribution of the data. The only parameter that changes is the query point, which is chosen randomly. A Minstep point (o) represents the number of times (OY) the system has retrieved objects in a Minstep retrieval mode in a specific number of steps (OX). We can notice that the Minstep has an exponential distribution. A Fagin point (.) represents the number of times (OY) the system has retrieved objects in a Fagin retrieval mode in a specific number of steps (OX). A Threshold point (*) represents the number of times (OY) the system has retrieved objects in a Threshold retrieval mode in a specific number of steps (OX). Threshold and Fagin have a normal distribution.

Using Fagin's algorithm, the average number of steps is 1000. This means that in order to get the first nearest neighbor, we have to retrieve on an average half of the database. Using Threshold algorithm, the average number of steps is 550. In this case, in order to retrieve the nearest object, a quarter of the database has to be read on average. We can also notice that Minstep has an average of 75 steps for retrieving the nearest object. Therefore, we can conclude that Fagin and Threshold have very high costs in terms of number of steps (query time). Extrapolating from one nearest retrieved to *k*, the number of steps distribution will remain the same. As a result, we need a mechanism to retrieve faster the nearest objects. This algorithm should be as fast as Minstep or even faster. One way of doing it is by using the Comb algorithm.

Next, we will present the Comb algorithm for a system that comprises 4 subsystems (Fig. 25). The database holds 12,000 multimedia objects with features uniformly distributed. The query point is the same for all the experiments. Therefore, the graph for sequential retrieval using the original system will be the same for all experiments.

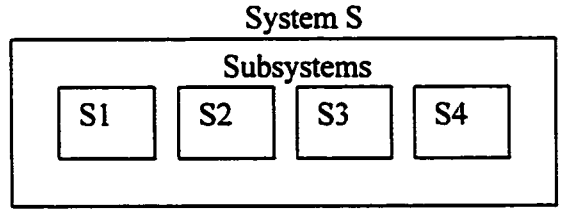


Fig. 25 System configuration for the Comb algorithm experiment.

We will try different scenarios of the Comb algorithm, static and dynamic. We will do this by changing the parameter of the Comb system, which are:

1) No_system - number of identical system (copies of the original system).
2) Distance ε - distribution of the query points (percentage of the average distance between the objects represented in the feature hyperspace). Q(n) = Q * (1 + ε*C(n))-see the definition of ε in the Comb algorithm section.
3) Comb_ no_steps - number of steps to run the Comb algorithm.
4) Seq_ no_steps - number of steps to run the sequential retrieval.
5) No_Objects_display – number of nearest objects display on the graphs.

We will first present a relevant example of the Comb static and dynamic algorithm in comparison with sequential retrieval of the nearest neighbors on one original system. The parameters introduced above will be as follow:

1) No_system = 5;
2) ε = 60 %;
3) Comb_ no_steps =100;
4) Seq_ no_steps = 100;
5) No_Objects_display =100;

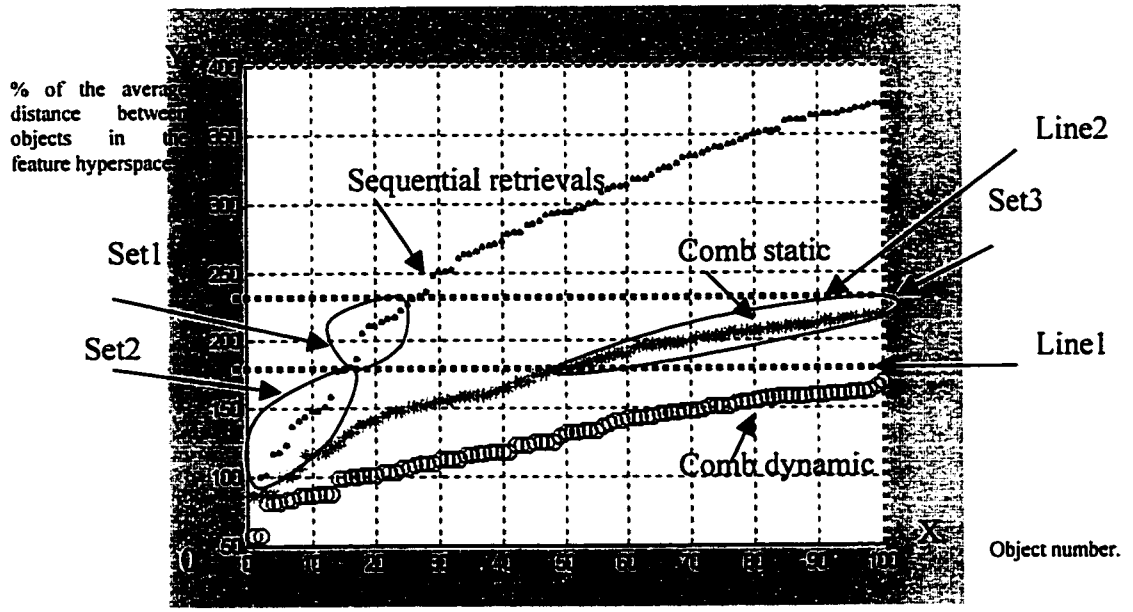


Fig. 26 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 60 %; Comb_ no_steps =100;
Seq_ no_steps = 100; No_Objects_display =100;

In the above figure we have displayed 100 objects retrieved with the Comb algorithm static and dynamic, and the sequential retrieval (100 displayed in each retrieval mode). On the OX axis we have displayed the object number and on the OY axis we have the distance from a retrieved object to the query point Q. This distance is displayed as the distance relative to the average distance between objects in the hyperspace of features in percentage.

We run the three modes for retrieving the objects. All three modes are displaying the same number of objects, which are 100.

In the dynamic Comb case all the objects are below Line1. The only objects retrieved by the sequential mode that are matching the dynamic Comb objects are in Set2. They are

matching in terms of being as good as the dynamic Comb objects with respect to the distance to the query point. The number of points in Set2 are relatively small compare to the number of dynamic objects = 100. Therefore, the objects retrieved by the dynamic algorithm are closer than the objects retrieved by the sequential algorithm. Similarly, the number of objects retrieved by the sequential algorithm that are matching the static Comb algorithm are the number of points in Set1 and Set2. We can also notice that the static Comb algorithm is performing better than the sequential algorithm.

Interesting to notice is the fact that the dynamic Comb algorithm is performing better than the static Comb algorithm. In fact, only the total number of objects displayed by the static Comb algorithm less Set3 are performing as good as the dynamic Comb algorithm. Therefore, for this setting of parameters, the dynamic Comb performs better than the static one. For these experiment settings, we can conclude that the dynamic Comb algorithm is performing much better than the sequential retrieval that is used in Fagin or Threshold algorithms. So, we have a trade off between more computing power and time retrieval.

A second experiment is conducted by setting up the parameters in the following way:

1) No_system = 5;
2) $\varepsilon = 60\ \%$;
3) Comb_ no_steps =20;
4) Seq_ no_steps = 100;
5) No_Objects_display =100;

In this experiment, we try to compare the 3 modes of retrieval when the overall number of steps is the same. In other words, Seq_ no_steps = Comb_ no_steps* No_system. We want to see if the sequential mode is more powerful than the Comb modes.

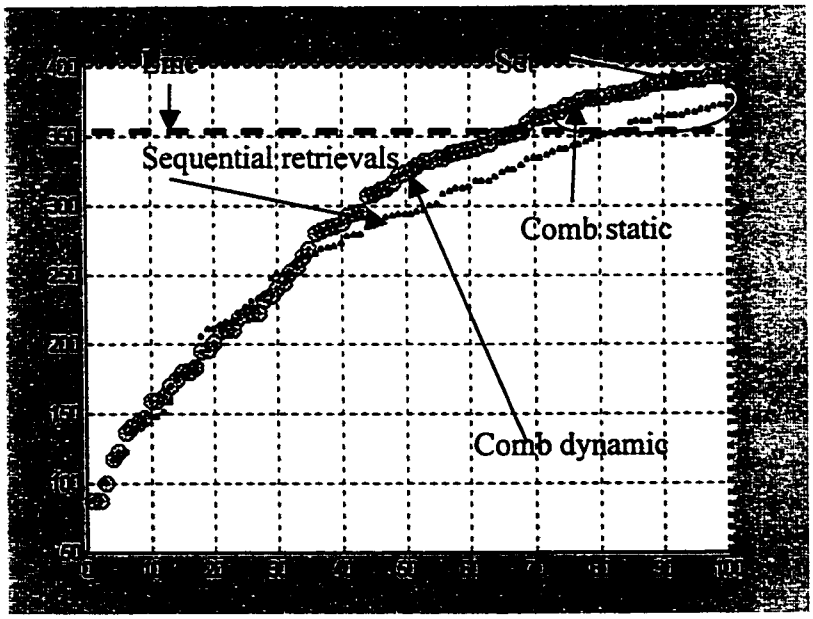


Fig. 27 Comb Algorithm (static and dynamic) and sequential retrieval

We can notice that the objets retrieved by the static Comb algorithm are identical to the objects retrieved by the dynamic Comb algorithm. This is because the number of steps set for the dynamic Comb algorithm is too small, and the relocation of query points didn't start. Also, we can observe that only the objects in set Set are not matching the objects retrieved in sequential mode. But, most of the objects retrieved in the Comb mode are matching the best nearest hits in sequential algorithm. The above experiment is relevant for knowing what are the differences in power computation when using the two modes, sequential and Comb. This shows that the modes are almost equivalent. Therefore, by distributing the application we do not lose computation power.

We repeat the experiment only this time; we have more steps to go (500). We will display 500 objects for each retrieval mode. The parameters' setting is:

1) No_system = 5;
2) $\varepsilon = 80\ \%$;
3) Comb_ no_steps =100;
4) Seq_ no_steps = 500;
5) No_Objects_display =500;

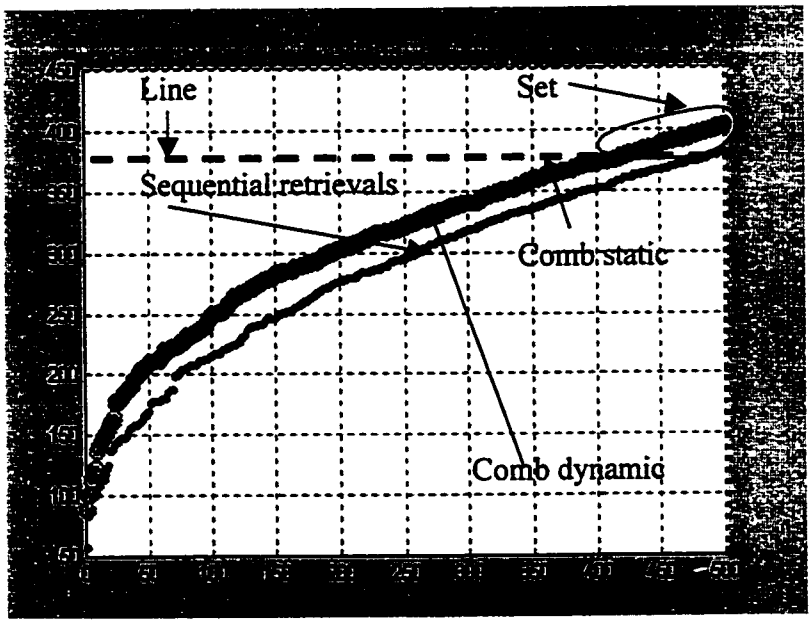


Fig. 28 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; $\varepsilon = 80\ \%$; Comb_ no_steps = 100;
Seq_ no_steps = 500; No_Objects_display =500;

This has the same behavior as the previous experiment. Only the elements from set Set are not matching the objects retrieved sequentially. In the next graph we have the same experiment, but we displayed only the first 100 nearest objects (Fig. 29).

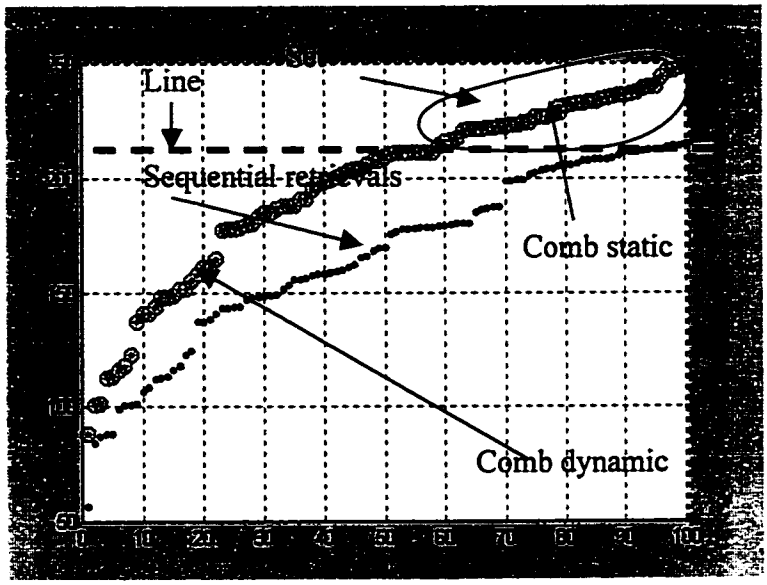


Fig. 29 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 5; ε = 80 %; Comb_ no_steps = 100;
Seq_ no_steps = 500; No_Objects_display =100;

This graph is a detail of the previous experiment. We can notice that most of the Comb objects are matching the sequential objects (except set Set).

In the next experiment, we will see how the three modes of retrieval behave when we have 11 identical systems and an equivalent number of query points for Comb retrieval modes.

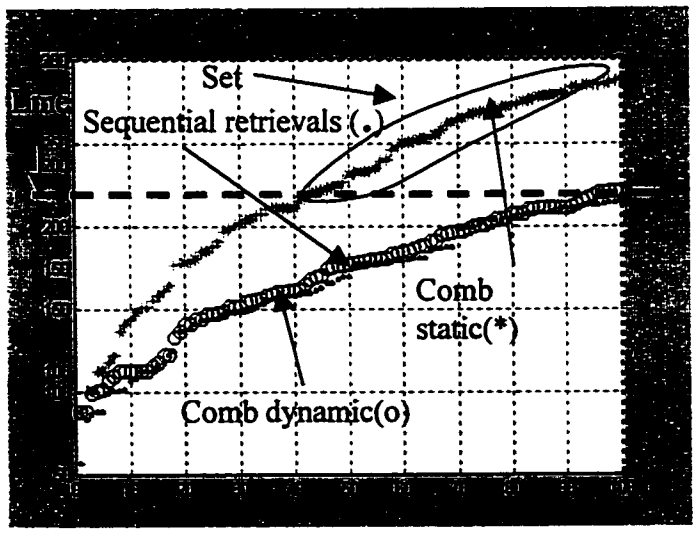


Fig. 30 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 80 %; Comb_ no_steps = 50;
Seq_ no_steps = 500; No_Objects_display =100;

A remarkable fact is that sequential and dynamic Comb modes behave identically. This is due to the fact that we retrieve many objects in sequential mode and we display only a fraction of them. On the other hand, in the dynamic Comb mode we use many identical systems and the relocation of query points works very well. We can also notice that the set of non-matching static Comb objects Set is large because ε is too large. Therefore, the use of static is not efficient. A conclusion for this experiment would be that if the number of sequential retrieved objects is very big, then, the sequential mode could match the dynamic Comb algorithm for the same global number of steps.

The following experiment is almost the same as the one before except that the distance ε is now 30%.

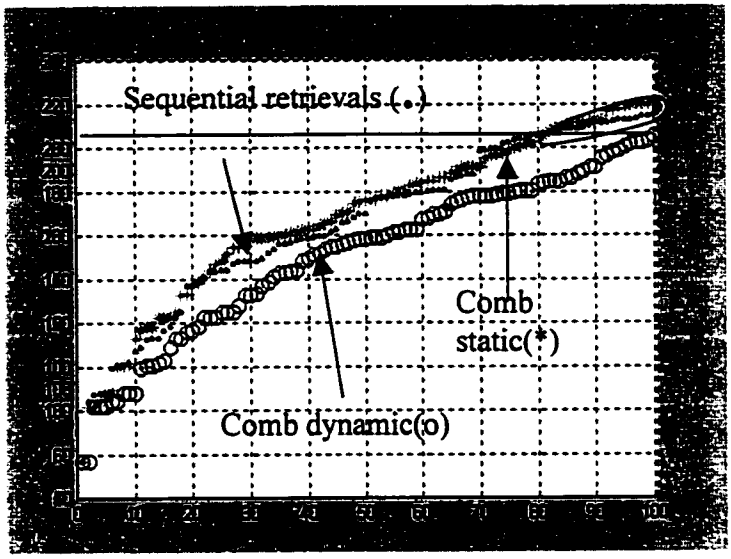


Fig. 31 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 30 %; Comb_ no_steps = 50;
Seq_ no_steps = 500; No_Objects_display =100.

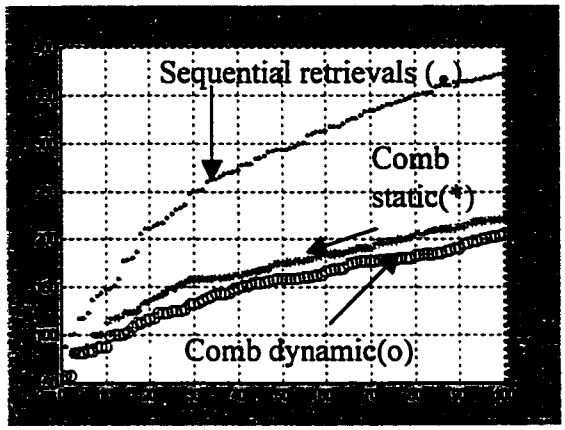
In this experiment the three retrieval modes are almost equivalent. The dynamic Comb retrieval algorithm is slightly better than the other 2. The reallocation of query points works properly for the dynamic Comb mode.

In the next 3 experiments we will see how ε is influencing the retrieval results. We have chosen the following 3 values: 30, 60, and 100%. The system setting is the following:

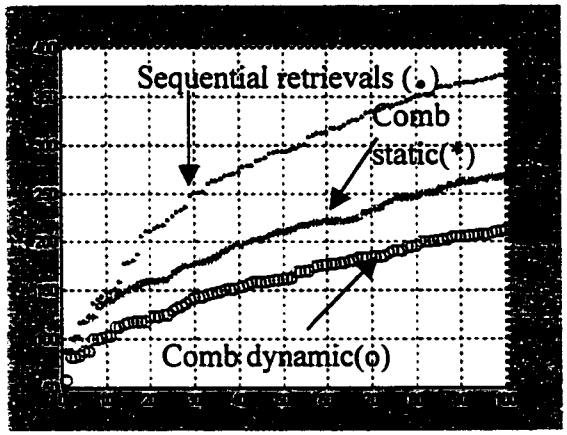
1) No_system = 11;
2) Comb_ no_steps =50;
3) Seq_ no_steps = 100;
4) No_Objects_display =100;

We will notice that the only set of objects that is very different from one experiment to another is the set of objects retrieved in the static Comb mode.

ε = 30%;



ε = 60%;



ε =100%;

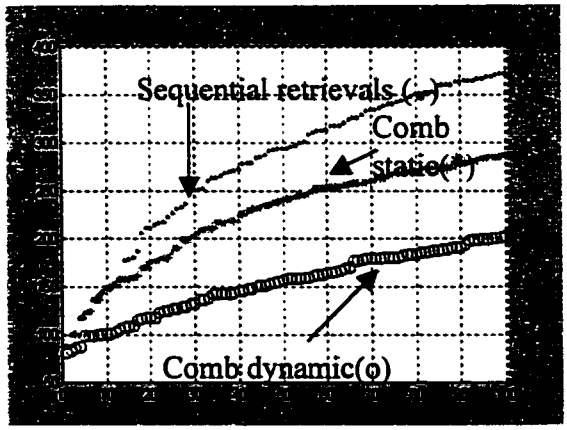
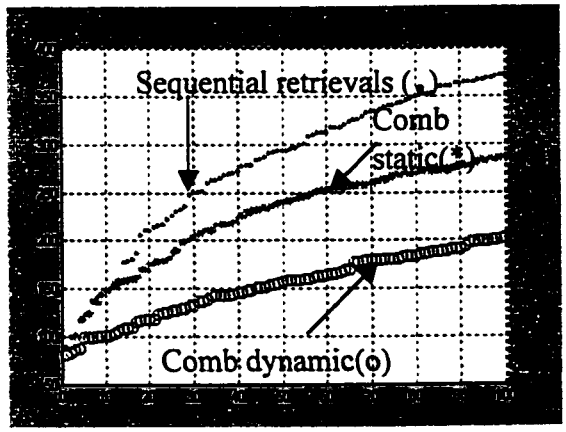


Fig.32 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system = 11; ε = 30, 60, 100 %; Comb_ no_steps = 50;
Seq_ no_steps = 100; No_Objects_display =100;

We notice that the sequential retrieved objects remain unchanged. For the dynamic Comb algorithm the set of retrieved objects changes slightly. Instead, for the static comb algorithm the set of retrieved depends on ε. We can conclude that the dynamic Comb algorithm is adapting and changing the position of the query points such way that it is retrieving a set of objects near to optimal from a distributed parallel system.

In the last experiment we will see how the number of identical systems is influencing the quality of retrieved objects. Therefore, we have in one experiment 11 identical systems and in another one 7 identical systems.

No_system = 11;



No_system = 7;

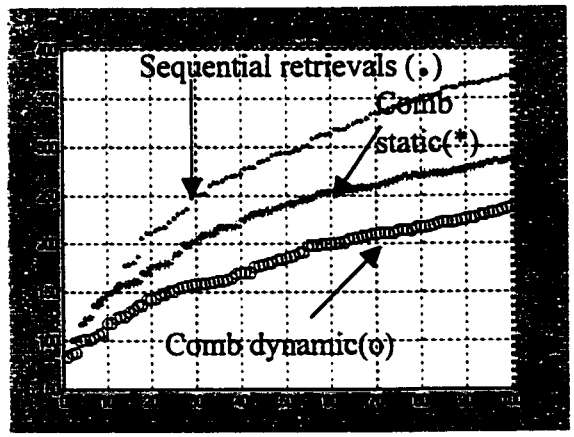


Fig. 33 Comb Algorithm (static and dynamic) and sequential retrieval.
No_system =7, 11; ε = 100 %; Comb_ no_steps = 50;
Seq_ no_steps = 100; No_Objects_display =100;

We notice a better quality for the Comb dynamic objects retrieved when using 11 identical systems compared to 7 identical systems. Obviously, the more systems we have to query in parallel, the better answers we get.

We have to find an optimal number of identical systems. As we can see in the experiments, the difference between the set is not big. Therefore, is it not advantageous to add 4 extra identical systems just to have a little improvement on retrieved set of objects. This trade-off between computing resource availability and query quality depends on the application and requirements. One good thing is that we can improve on the query time as much as we want of course by providing additional computing resources.

6. Conclusion

The Comb algorithm is a solution for increasing the search quality of multimedia objects distributed over a cluster of indexing structures. By search quality we mean better and faster. Fagin and Threshold algorithms have limitations with respect to the speed of query execution. Therefore, Comb and Fagin's-Threshold algorithm can complement each other. Fagin's and Threshold algorithms have accuracy in finding the query answers and Comb algorithm has speed. Future work will be the integration of the two concepts: precise and fast. Precision as we mentioned in the introduction of this report is not well defined for multimedia objects. Once we have defined the criteria for precision, then, we can apply the Fagin-Threshold method.

One way of using the Comb algorithm as a stand alone algorithm is to use it interactively while retrieving objects. In other words, the user programs the system to retrieve a set of objects. The system will provide the user with a subset of significant objects from the retrieved set. If the user is not happy with the results, the system retrieves another set of objects and provides the user with another set of significant objects. The retrieval process continues until the user accepts an answer. In this case, Fagin and Threshold are not needed anymore for accuracy. The user decides what is good and accurate.

In conclusion, the Comb algorithm, which implies also Comb system, helps to improve on time query versus computing resources. If we need fast query time, we have to improve on the indexing system.

7. References

[1] Chang, S-K, "Image Information Systems," Proc. IEEE, Vol. 73, No. 4, April 1995, pp.754-764

[2] Subrahmanian, V.S. and Jajodia, S. (Eds.), "Multimedia Database Systems: Issues and Research Directions", Springer, 96.

[3] Steinmetz, R. and Nahrstedt, K., " Multimedia: Computing, Communications, and Applications", Prentice Hall, 96.

[4] Kemp, Z., "Multimedia and Spatial Information Systems", IEEE Multimedia, Vol.2, No. 4, 1995.

[5] Thuraisingham, B., Nwosu, K. and Berra P.B., "Multimedia Database Management Systems-Research issues and future directions", Kluwer Academic, 97.

[6] Guttman, A.: " R-Tree: A Dynamic Index Structure for Spacial Searching, " Proc. ACM SIGMOD, pp. 47-57,1984.

[7] Oliver Günther: "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases". ICDE pp. 598-605, 1989.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles." Proc. ACM SIGMOD,: 322-331, 1990.

[9] Christos Faloutsos, Timos K. Sellis, Nick Roussopoulos: "Analysis of Object Oriented Spatial Access Methods." SIGMOD Conference 1987: 426-439

[10] Christos Faloutsos, Ibrahim Kamel: "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension." PODS 1994: 4-13

[11] Nick Roussopoulos, Stephen Kelley, Frédéic Vincent: "Nearest Neighbor Queries." SIGMOD Conference pp. 71-79, 1995.

[12] Ibrahim Kamel, Christos Faloutsos: "Hilbert R-tree: An Improved R-tree using Fractals." VLDB pp. 500-509, 1994.

[13] Ronald Fagin: Fuzzy Queries in Multimedia Database Systems. PODS pp.1-10,1998.

[14] Lotfi A. Zadeh: "Fuzzy Sets. Information and Control" pp. 338-353 ,1965.

[15] Ronald Fagin, Amnon Lotem, Moni Naor: "Optimal Aggregation Algorithms for Middleware". PODS 2001.

[16] Demet Aksoy, Michael J. Franklin: "A scheduling approach for large-scale on-demand data broadcast", EEE/ACM Transactions on Networking, pp. 846-860, 1999.

[17] R.Fagin: " Combining Fuzzy Information from Multiple Systems", J. Computer and System Sciences. (Special issue for selected papers from 1996 Symposium on Principles of Database Systems.) Preliminary version appeared in Proc. Fifteenth ACM Symp. On the Principles of Database Systems, Montreal, pp. 216-226, 1996.

[18] W. F. Cody, L.M. Maas, W. Niblack, M. Arya, M. J. Carey, R. Fagin, M. Flickner, D.S. Lee, D. Petrovici, P.M. Schwartz, J. Thomas, M. Tork Roth, J.H. Williams, and E.L. Wimmers: "Querying Multimedia Data from multiple Repositories by Content: The Garlic Project", IFIP 2.6 3rd Working Conference on Visual Database Systems (VDB-3), 1995.

[19] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovici, and P.Yanker: "The QBIC Project: Querying Images by Content Using Color, Texture and Shape", SPIE Conference on Storage and Retrieval for Image and Video Databases, volume 1908, pp.173-187, 1993.