Vector Approximation based Indexing for High-Dimensional Multimedia Databases

I. Daoudi, S.E. Ouatik, A. El Kharraz, K. Idrissi, and D. Aboutajdine

***Abstract*—With the proliferation of multimedia data, there is an increasing need to support the indexing and searching of high-dimensional data. In this paper, we propose an efficient indexing method for high-dimensional multimedia databases using the *filtering approach*, known also as *vector approximation approach* which supports the nearest neighbor search efficiently. Our technique called *RA^+-Blocks (Region Approximation Blocks)* divides a high-dimensional feature vector space into compact and disjoined regions. Each region will be approximated by two bit-strings according to the RA-Blocks technique. *RA^+-Blocks* improves the division strategy of data space compared to the RA-Blocks. From our experiment using high-dimensional feature vectors, we show that RA^+-Blocks achieves better performance on the nearest neighbor search than VA-File and RA-Blocks on both uniform and real data.**

***Index Terms*—High-Dimensional space, indexing method, multimedia database, nearest neighbor (NN) search.**

I. Introduction

One of the fundamental problems, in multimedia databases domain, resides in the similarity search, i.e. the need to retrieve a small set of objects which are similar or closest to a given query object. Generally, the similarity is not measured on the multimedia objects directly, but on their traditional primitives (histograms of colours, signatures sound…). These primitives appear as vectors of numeric values known as *feature vectors* and constitute the indexes of these objects. This way, similarity searching in multimedia database becomes a K-nearest neighbor (K-NN) search in a high-dimensional vector space. It consists to find in a space of great dimension the nearest K vectors, in term of distance, to a query vector. For applications where the vectors have low or medium dimensionalities (e.g., less than ten), the conventional indexing methods, such as R–Tree [1], R^*-Tree[2], SR-Tree [3], K-D-B-Tree [4], and X-Tree [5], can be adequately used to solve the problem of K-NN search. However, there is no effective solution to this problem for the applications with vectors having high dimensionalities, e.g., dimension over 100. In fact, for a high dimensionality, the performance of the conventional methods degenerates to being worse than the brute-force sequential scan comparing the query object to each data object [7]-[8]. Therefore, the main issue is to overcome the *dimensionality curse* [9]-[10] i.e. a phenomenon that the performance of indexing methods degrades drastically as soon as dimension becomes large. Several new methods based on the *filtering approach* known also as *vector-approximation* are proposed as a palliative solution for this problem (VA-File [8], VA^+-File [11], LPC-File [10], A-Tree [12], GC-Tree [13], RA-Blocks [14]...). These methods divide the data space into rectangular cells, allocate a unique bit-string of each cell, and approximate the data vector that falls into a cell by this bit-string. To search the K-NN of a query vector, they sequentially read the relatively smaller approximations file instead of the data file and try to filter the original vectors so that only a small fraction of them can be read. Then a limited access to the database of the real vectors is done just for those which have been selected at the first stage.

In this paper, we propose an efficient indexing method for high-dimensional spaces. Being based on the vector approximation approach, it adopts a better strategy of partitioning than that of K-D-B-Tree used by RA-Blocks. Thus, it significantly reduces the number of regions to cross, at the phase of filtering, by eliminating the trivial regions (empty). In addition, all the regions generated by our method are dense i.e. they contain a number of vectors close to their capacity (capacity is the maximum number of objects which can be accommodated in a disk page). Hence, a CPU time reduction related to the computation of the distances between the regions and the query vector is assured.

This paper is organized as follows. Section II presents the major high-dimensional indexing methods in particular those based on the filtering approach. Section III describes the overall structure of our technique and presents its partitioning strategy ant its nearest neighbor search algorithm. Section IV describes our experimental evaluations and shows performance results. We conclude with summary and further work.

Manuscript received June 13, 2007.

I. Daoudi is with the GSCM, University Mohammed V, Faculty of Science, Rabat, Morocco; (e-mail: imanedaoudi1@ yahoo.fr).

S. E. Ouatik, is with the laboratory of Computer science, Statistics, and quality (LISQ), Faculty of Science Dhar Mahraz., Fez, Morocco (phone: +212 63 49 70 68; fax: +212 35 73 30 59; e-mail: s_ouatik@yahoo.com/ souatik@fsdmfes.ac.ma).

A. Elkharraz is with National School of Management, Tangier, Morocco (e-mail: kharraz_2003@yahoo.fr).

K. Idrissi is with the LIRIS laboratory, INSA, Lyon, France; (email: idrissi@insa-lyon.fr)

D. Aboutajdine is with the GSCM, University Mohammed V, Faculty of Science, Rabat, Morocco; (e-mail: aboutaj@ fsr.ac.ma).

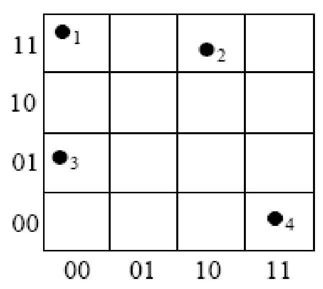
II. RELATED WORK

The conventional indexing methods to supporting similarity search in high-dimensional vector space can be broadly classified into two categories. The first approach uses data-partitioning methods, witch divide the data space according to their distribution. Many index tree schemes have been proposed. They include the R-Tree [1], R^+-Tree [15], R^*-Tree [2], X-Tree [5], M-Tree [16], SS-Tree [17], and SR-Tree [3]. Neighbor vectors are covered by MBRs (Minimum Bounding Rectangles) or MBSs (Minimum Bounding Spheres), which are organized in a hierarchical tree structure. These methods can yield possible overlapping regions. The second approach use space-partitioning methods that divide the data space along predefined hyper planes regardless of data distribution like Grid-File [18], K-D-B-Tree [4], LSD-Tree [19], and LSD^h-Tree [20]. The resulting regions are mutually disjoint, with their union being the complete space. Although such access methods generally work well for low-dimensional spaces, their performance is known to degrade as the number of dimensions increases.

New techniques based on the approximation vector approach appeared to provide a solution to the dimensionality curse. VA-file (Vector Approximation File) [8]-[21] is the first method based on this approach. The basic idea of the VA-File is to keep two files; one contains the exact representation (original vectors), the other, relatively smaller, has geometrical approximation for each vector. When searching vectors, the entire approximations file is scanned to select candidate vectors. Upper and lower bounds of the distance to the query (d_{max} and d_{min}) are computed for each vector (see figure 1). These bounds frequently suffice to filter most of the vectors. Those candidates are then verified by visiting the original vectors file. We note that the sequential scan is done on the smaller approximations file; it is very fast and allows reading just a few vectors from real database. This process decreases the number of I/O operations and the CPU cost compared with the sequential method that analyzes the totality of the database.

Like the VA-File, the LPC-File (Local Polar Coordinate File) [10] is based on the approximate approach. Thus, the vector space is partitioned into rectangular cells which are used to generate bit-encoded approximations for each vector. Cha [10] noticed that the performances of the VA-File can be improved only by increasing the number of bits for approximations.

However, the performance of the VA-File converges to that of the sequential scan or degenerates to being worse than the number of bits used for approximations increases.



Vectors in database

●1	0.1 0.9
●2	0.6 0.8
●3	0.1 0.4
●4	0.9 0.1

Approximations

●1	0011
●2	1011
●3	0001
●4	1100

Fig.1 Construction of VA-File

Cha proposes, to overcome this problem, by adding polar coordinate information of the vector to the approximation. It increases the filtering rate of the filter-based approach.

Berchtold, et al. also used the idea of vector quantization, and proposed the IQ-Tree (Independent Quantization Tree) [6], a multilevel indexing structure specially designed to perform fast nearest neighbor searches over high-dimensional data sets [6]. The IQ-tree is derived from the X-tree [5], and uses the partitioning strategy proposed with the X-tree bulk-loading algorithm. The structure has three levels: Directory Pages, Vector Approximation Pages and Vector Pages. For each entry of the directory, there is a page containing Vector Approximations (VAs) in the second level, and multiple pages containing vectors in the third level (represented all together by a circle). The number of vector pages associated with a directory entry is variable, and depends on the capacity of the associated VA page. Each directory entry contains the coordinates of the minimum bounding rectangles that encloses all the vectors corresponding to that entry. The VAs contained in a page of the second level are computed with respect to the MBR of the corresponding directory entry, and each page can use a different number of bits per dimension for the approximations. Given a list of pages to be read, the nearest neighbor search strategy for the IQ-Tree involves ordering the list of pages by their position in the file, and then performing a single read request for any contiguous (or even nearly contiguous) sequence of pages in the list. This optimization reduces the number of (expensive) positioning at the cost of a few extra (cheap) page transfers.

In the IQ-tree context, it is not known initially exactly which pages hold the K nearest neighbors, but Berchtold et al. propose a probabilistic model for estimating the probability that each page will eventually have to be read in order to satisfy the query. The model specifies the set of pages that should be read in order to minimize the expected overall I/O cost.

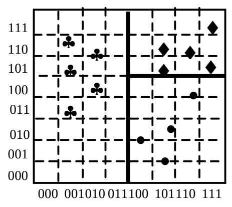
Sakurai et al. also make use of the idea of quantization in proposing the A-tree (Approximation Tree) [12]. The A-tree structure and construction algorithm are derived primarily from the SR-tree [3]. The A-tree design introduces the concept of virtual bounding rectangle (VBR), which a compressed approximation is using only a few bits per dimension of a minimum bounding rectangle (MBR). In a more general form of quantization, VBRs approximate MBRs in a manner analogous to how VAs approximates vectors. That is, the VBR defines a hyper-rectangle guaranteed to fully enclose the corresponding MBR. The use of VBRs, which are smaller than MBRs, increases the fanout of the nodes, which in turn reduces the height of the index tree and speeds up the search. An A-tree is a hierarchical index, with three levels: Internal Nodes, Leaf Nodes, and Vector Pages. Each of these levels has its own type of node. Internal nodes and leaf nodes together form a hierarchical index. Vector nodes contain a cluster of neighboring vectors. Both internal and leaf nodes contain a header and multiple entries. The headers include an MBR that encloses all the vectors of the subtree rooted at the node. The entries of each internal node contain a VBR that bounds the MBR of the corresponding child node. The entries of a leaf node are VAs of the vectors in the associated vector pages. Each node entry includes the centroïd of the vectors within the associated MBR. The centroïd is used only by the insertion algorithm and not by the search algorithm.

We note that the performance of the VA-File decreases when the capacity of database becomes very large. Thereafter, the approximations file cannot be placed entirely in the memory. Thus, the RA-Blocks technique (Region Approximated Blocks) [14] is proposed. It divides the vectors space into regions containing each one a set of cells, in order to approximate each region by two string-bits. This reduces the computation time of the VA-File and also optimizes the page replacement to further reduce the number of I/O access as well. Moreover, the bounds d_{min} and d_{max} are calculated for each region and not for each vector in order to reduce the CPU cost. Despite these advantages, RA-Blocks presents some limitations. Indeed, the strategy of partitioning, used according to the algorithm K-D-B-Tree [4], generates empty regions together with regions containing few vectors compared to their capacity. This involves a significant number of regions to be treated during the filtering step (approximations file relatively large). Also, the CPU time due to the calculation of the d_{min} and d_{max} of these regions increases.

III. RA^{+}-Blocks (Region Approximation Blocks)

Our method is inspired of the RA-Blocks method for the quantification of the data space; it improves the partitioning strategy of the data space compared to the RA-Blocks method.

For the quantification step, each dimension d_i is split into 2^{bi} intervals where each interval is encoded with bi bits. Then, data space is subdivided into compact and disjoined regions having practically the same capacity. Each region will be approximated by two bit -string corresponding to the



	Approximation
♦	100 101
	111 111 end
♣	000 000 begin
	011 111 end
●	100 000 begin
	111 100 end

Fig. 2. Partitioning of the data space and approximation of the regions

beginning (left low cell) and the ending (right high cell) position of this region (see figure 2). Our method is divided in tow phases: the first one focus on data indexing, while the second aims at interrogation of the database.

A. *Data indexing*

This phase consists in splitting the data space in regions containing each one a number of vectors lower or equal to the capacity, then to approximate them by two bit-strings. These approximations will constitute the index file.

1) Subdivision of the data space: After having subdivided the data space into hyper-rectangular cells, this stage consists of partitioning space in disjoined regions having practically the same number of vectors. Almost the whole of the obtained regions contain a number of vectors great than the half of the capacity (capacity/2). Our partitioning method is inspired by the K-D-B-Tree algorithm. It uses the same strategy for searching the side to be divided (dimension of subdivision), and the calculation of the subdivision value according to this dimension.

The Dimension of subdivision and the Value of subdivision are defined as follows:

- Dimension of subdivision: the subdivision is carried out according to the dimension having the maximum of data dissemination, i.e. the dimension that corresponds to the greatest difference between the vectors components.
- Value of subdivision: it is the value of quantification nearest to the median value according to the dimension of subdivision.

First, the split algorithm defines the dimension of subdivision, and then determines its value. The subdivision of the mother region is carried out according to the hyper plane passing by the value of subdivision, it, thus, results in two regions containing practically the same number of vectors. Each one of the obtained regions will be approximated by two bit-strings corresponding to the left low and right high cells, forming the approximations file. Figure 2 describes region approximations in a two dimensional vector space of 14 vectors. The capacity of regions is chosen to be 5, b_i is fixed equal to 3. It results in 3 regions; each one coded by two bit-strings. Let us note that our splitting algorithm is based on the elimination of the internal nodes partitioning (downward subdivision) of the K-D-B-Tree structure. The main drawback of this structure is the fact that it produces important reorganizations of the totality of the tree in order to preserve

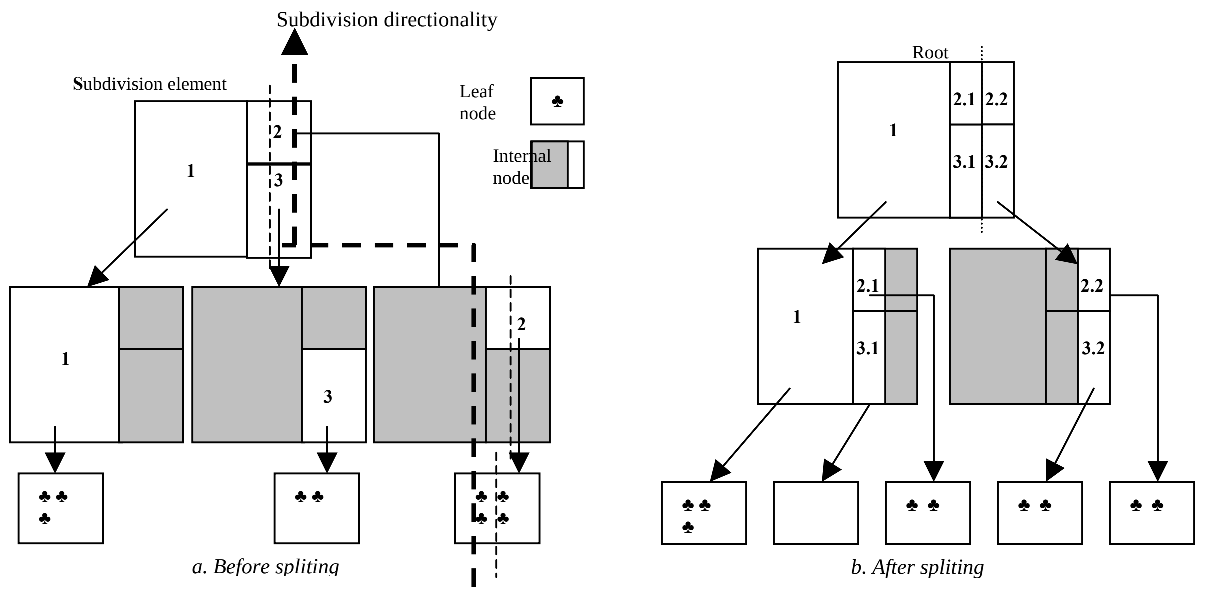


Fig. 3. Example of regions splitting according to K-D-B-Tree

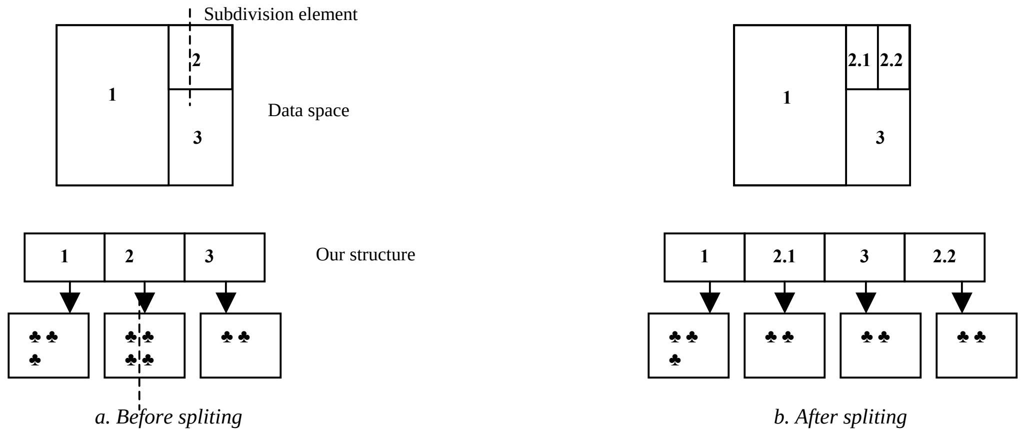


Fig. 4. Example of regions splitting according to our method

the properties of the K-D-B-Tree structure. These structuring often cause the creation of a large number of empty leafs or almost empty, which cannot guarantee an optimal rate of allocated space use, and thereafter increases the search time of K nearest neighbor. Hence, the basic idea of our method is to partition only the overflowed point pages setting a difference to the K-D-B-Tree method which divides a priori all region pages having an intersection with the hyper plane of subdivision. In our case, only the nonempty point pages are generated by the splitting algorithm. This way, we can eliminate the internal nodes partitioning by transforming the tree structure into a list of regions. Thereafter, each overflowed region will be split and replaced by two new regions (left region and right region) having each one a size less than a half size of the capacity (size <capacity/2).

It allows eliminating the empty regions. Furthermore, all the generated regions are dense (i.e. number of vectors>capacity/2). On the one hand, this guaranteed, in the majority of the cases, the existence of the K-NN in the same disk page, and on the other hand, reduced the total number of the obtained regions and thereafter reduced the I/O time. Figures 3 and 4 present an example of a 2d-space splitting using the K-D-B-Tree method and our method. The regions capacity is fixed at 3. According to the K-D-B-Tree (fig. 3), all the regions having an intersection with the subdivision element are split into two sub regions, including those which are not overflowed (region 3). Thereafter, resulting point pages are not very dense and sometimes empty (e.g. regions 3.1 and 3.2). However, according to our method (fig. 4), the subdivision element partition only overflowed regions (e.g. region 2). Therefore, the number of obtained points pages (4) is lower than that of figure 3 (5).

The construction of the regions is carried out by successive insertions of the data vectors. The construction algorithm is described in figure 5.

```
Input :
data : set of real vectors in the database.
Output :
ListReg : list of regions containing the splitting result.
Function RegionsConstruction(vector data[ ], integer N)
{
// data : the vectors in database scanned per block
// N : the number of  vectors per block
//Pos : integer
  //ListReg : this list of regions is initialised by one region which is the root region
//Region R_left, R_right : regions resulting from the splitting an old region
1. For i=1 to N do{
1.1. Pos= GetIdRegion(data[i]) // return index of region in which must be inserted the vector data[i]
1.2. Let Reg be the region having as index pos
1.3. Insert(data[i], Reg) // allow to insert data[i] in Reg
1.4. If (overflow(Reg)) {
        // number of vectors in the region Reg is greater than capacity, then this region is overflow.
    1.4.1. split(Reg, R_left, R_right) // Reg is split into 2 sub regions
    1.4.2. If (nbredata(R_left)!=0) and (nbredata(R_right)!=0) {
// both the number of vectors in R_left and R_right are not equal to 0
        a. Replace(ListReg, Reg, R_left) // replace the region Reg by R_left in ListReg
        b. InsertToEnd(listReg, R_right) // insert R_right into the end of ListReg.
}// End If
} // End If
}// End For
}// End
```

Fig. 5. Regions construction algorithm

2) Index structure: The index structure is similar to RA-Blocks. Thus, we record data in two levels. In the first level, we use a small file containing the region approximations coded in bits. This approximations file is very small, hence, in most cases, it can be kept in the main memory and we don't have to access the disk in the first phase. In the second level, we record in a file the real vectors.

To insert a new vector into the database, we find the region in which the data should be placed and we insert this vector. If the disk page recording this region is full, i.e. the number of its vectors exceeds the capacity; we split it into two regions having practically the same number of vectors based on the split algorithm. In this case, the approximations of these regions are calculated and inserted in the approximations file, and then the first region is removed.

Deleting a vector consists in finding the region that contains this vector. If the number of vectors remaining after the suppression is lower than half capacity (capacity/2), then a reorganization of the region is carried out.

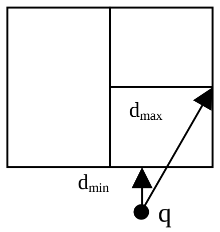


Fig. 6. Upper bound d_{max} and lower bound d_{min} using Euclidean

B K-nearest neighbor algorithm

Our K-nearest neighbor queries algorithm is inspired of the VA-NOA search algorithm in the VA-File [8]. The research algorithm for K-NN is divided into two phases. In the first stage (filtering step), the approximations file is scanned sequentially, and the candidates regions are selected. During the second phase (access to original vectors), we determine the K-NN by calculating the real distance from vectors of candidates regions to the query vector.

1) Filtering phase: To search the K-NN, the approximations file is entirely scanned, and each region's upper bound dmax and lower bound dmin compared to the query vector are calculated (see Figure. 6). We initialize the list of candidates LC by k/2 regions having the smallest dmax. This list is sorted out by the ascending distance dmax. The approximations file is scanned: if an approximation is found such as its dmin is less than the biggest upper bound dmax of the list, the corresponding region is inserted in LC by keeping the sorted list.

2) Access to real vectors: We initialize the list of the results LR by K first vectors of the LC regions. This list is then sorted according to the real distance to the query vector. Then, we scan the retained regions in LC according to an ascending order of their dmin and we calculate the real distance between its vectors and the query vector. We suppose that Dm is the current maximum real distance. If a region is found, and its dmin is less than Dm, the vectors of this region are examined: if the real distance which separates a vector from this region and the query vector is less than Dm, then the vector is inserted in LR and we remove the last LR vector. However, if a region of LC has a dmin higher than Dm then we should stop.

IV. EXPÉRIMENTAL EVALUATIONS

In order to demonstrate the applicability and the advantages of our technique, the experimental evaluations of our method compared to RA-Blocks, and VA-File are presented in this section.

All the experiments are conducted on a Microsoft Windows XP machine with 3.2 GHZ CPU, 1 Go RAM, 160 GB local disk. The following two data sets were used for the performance test: (1) random data set, (2) real data set.

Let d, be the dimensionality of the vectors, and S, the number of vectors in the database (database size).

The random data set is a random data which consists of S = 18000 vectors distributed uniformly in range [0 0.01392], with d = 100. The real data set consists of the histograms of colors extracted from a real image database, with d=256 and S=18000.

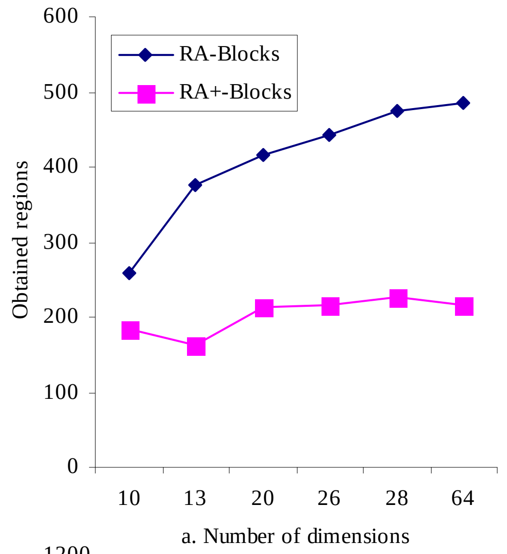
Three experimental evaluations were conducted. In the first one, the index structures size of both the RA-Blocks and RA^+-Blocks methods with various dimensions and with different database size are compared. In the second experiment, we compare the filling rate for both the RA-Blocks and the RA^+-Blocks index. Finally, the third experiment, evaluate the CPU time of RA^+-Blocks, RA-Blocks, and VA-File.

A. *Index structure size study*

Figure 7 shows the results of our experiments and illustrates the K-NN search performance of both RA-Blocks and RA^+-Blocks methods. These experiments are conduct on both real and random image database; we show only the results for real data. In (fig.7a), the dimensionality of data space is ranged from 10 to 100 and the database size is fixed to 10000 descriptors. While in the (fig.7b), the dimensionality of vectors is fixed to 12 and the size database ranged from 2000 to 18000 vectors. Similarly to the experiments reported in [22], the K-NN search performance of RA-Blocks and RA^+-Blocks index was measured in term of number of the obtained regions with two index structures by fixing : K-NN=5, number bit per dimension=8, disk page size=16 KB, and capacity=7.

As we can see from figure 7, the number of the obtained regions using K-D-B-Tree strategy for space partitioning adopted by RA-Blocks is higher than the one obtained using RA^+-Blocks. Note that the number of the obtained regions with RA-Blocks index grows with the dimensionality and with the database size. This will increase the size of the RA-Bocks' index structure, and will also require more space allocation for its regions. Thus, the storage utilization for the RA-Blocks is lower than that one of the RA^+-Blocks index. Consequently, the performance of the RA-Blocks is lower than that of the RA^+-blocks

To explain the reasons behind this degeneration of RA-Blocks index, consider an RA-Blocks index when it has only one region page R. Let assume that R is completely full and the split of a point page P has just begun. Splitting the point page P will trigger the split of the parent page R. Then, all nodes (MBRs) in R that intersect with the dividing hyperplane selected during the split of P will also be partitioned, and the splits will propagate into the corresponding child nodes of R. Since the node (MBRs) that full extension along the split i axis intersect with the dividing plane, a large number of regions will be created in the structure. This will increase the size of the structure and decrease the average utilization of its regions compared to RA^+-Blocks index which divide only full regions.



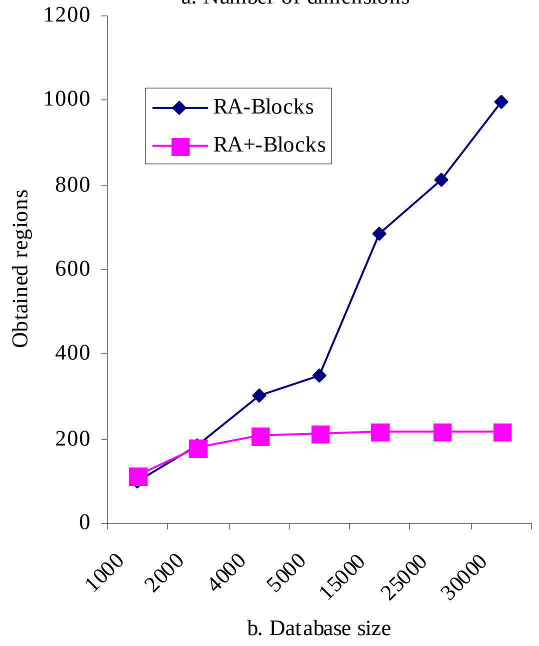


Fig.7. Number of the regions obtained with both RA-Blocks and RA^+-Blocks index according to the number of dimensions and the database size

Moreover, due to the splits of many subregions in R, the resulting regions R' and R'' instantly become occupied to nearly full capacity. Fairly soon, both R' and R'' will be split again, most likely along the dimension i+1. As before, one can expect that many nodes (MBRs) in R' and R'' will have full extension along the split dimension. Thus, the splits of R' and R'' would also create a large number of new regions, further increasing the size of structure. Figure 8 shows an example of forced splitting of regions in a K-D-B-Tree index used by RA-Blocks with d=2.

Finally, when the dimensionality or the database size increases, the RA-Blocks index requires an important reorganization of the totality of the index which often causes a forced splitting of the empty regions, and thus creates a large number of empty point pages.

In RA$^+$-Blocks approach, the partitioning strategy eliminates the forced splitting. When a region page R is completely full, the split of a point page P will trigger the split of only the region in the parent page R that contains the page point P. Thus, the split propagation is stopped at this level. From a certain number of dimensions or number of vectors, the split of regions cannot take place due to their low bounds, so the vectors are inserted without splitting. Consequently, the obtained regions with RA$^+$-Blocks are few and mostly dense compared to those obtained with RA-Blocks.

As we can see in figure 7. when dimension (Fig 7.a) and database size (Fig 7.b) increase, the number of the obtained regions varies slowly then remains practically constant. This will increase the filling rate of the RA$^+$-blocks, and reduce the number of calculation of upper and lower bound, and thus the CPU time.

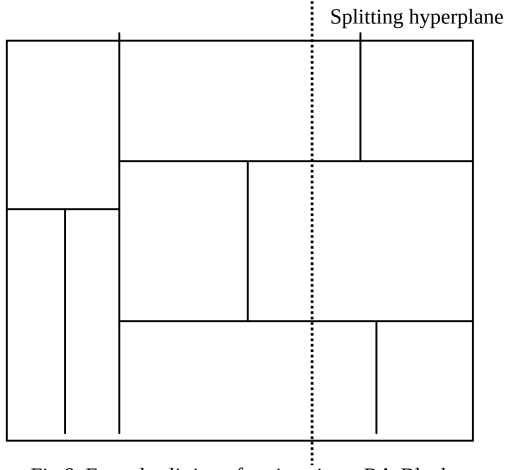


Fig.8. Forced splitting of regions in an RA-Blocks

B. Filling rate comparison

These experiments present a comparison between the RA$^+$-Blocks and RA-Blocks, in term of the filling rate for both random and real image databases. The filling rate is the ratio between the number of the vectors in the database and the total capacity of the index structure (capacity of the index structure is the total capacity of all the obtained regions). This ratio is a good performance estimator because the multidimensional index structure's performance depends largely on their ability to support the "scalability problems" associated with large amounts of data. A high value (near 100%) of this parameter guarantees an optimal use of disk pages, and decreases the number of I/O.

As can be seen from figure 9, the filling rate of RA-Blocks is less than the one in the RA$^+$-Blocks for both random and real data. Thus, the RA$^+$-Blocks index generates a low number of dense regions, while the RA-Blocks index generates a great number of sparse regions, which decrease storage utilization of the RA-Block index. The low storage utilization problem in the RA-Blocks index is caused by its imperfect splitting strategy described previously; it deteriorates directly the retrieval performance.

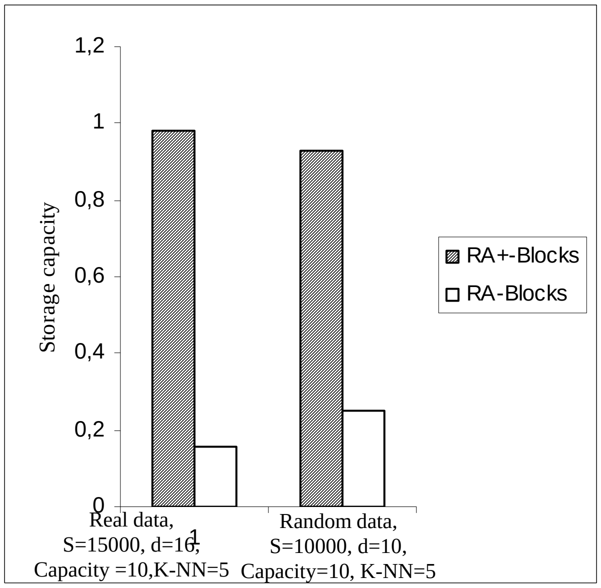


Fig. 9. Storage capacity comparison between the RA-Blocks and

CPU time study.

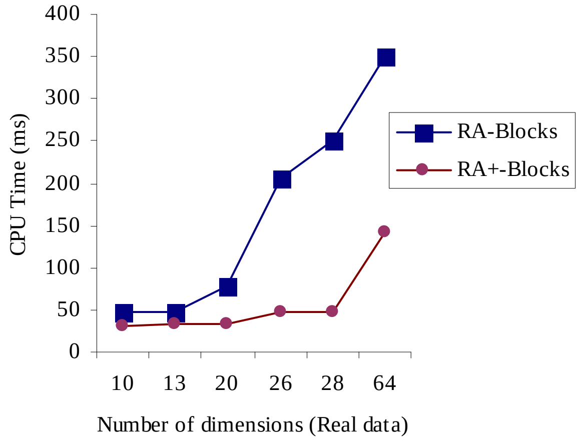
In figures 10 and 11, we compared the K-NN search performance for RA-Blocks, VA-File, and RA$^+$-Blocks on both random and real image database with various dimensions and various number of vectors.

In the figure 10, S=10000 and the dimensionality varies from 10 to 100 for both the random and real data. In the figure 11, the number of vectors varies from 2000 to 18000, while the dimensionality is fixed to 12.

From these experiments, we note that the CPU time of the K-NN search of the RA-Blocks method is higher than that of the RA$^+$-blocks method. As the dimensionality or the number of vectors increases, the RA$^+$-Blocks becomes better and better, compared to the RA-Blocks. Our method reduces the search time to be satisfactory. This can be explained by the optimized strategy of subdivision used in the RA$^+$-Blocks, that allows us to reduce the index structure size compared to the RA-Blocks by eliminating the forced splitting used in K-D-B-tree method adopted by RA-Blocks. The subdivision strategy of RA$^+$-Blocks reduces the number of the candidates regions to be selected in the filtering phase and thus the number of calculation of upper and lower bounds. The K-NN search time of the RA$^+$-Blocks is reduced compared to the RA-Blocks method.

In the figure 12, we also compare the CPU time of RA$^+$-Blocks, RA-Blocks, and VA-File using the parameters:

d=60, K-NN=5, capacity=5, S=10000.



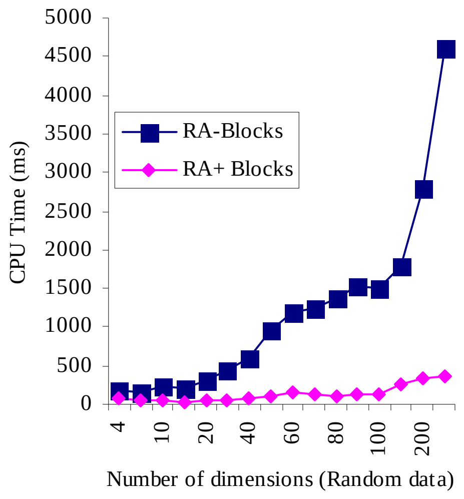
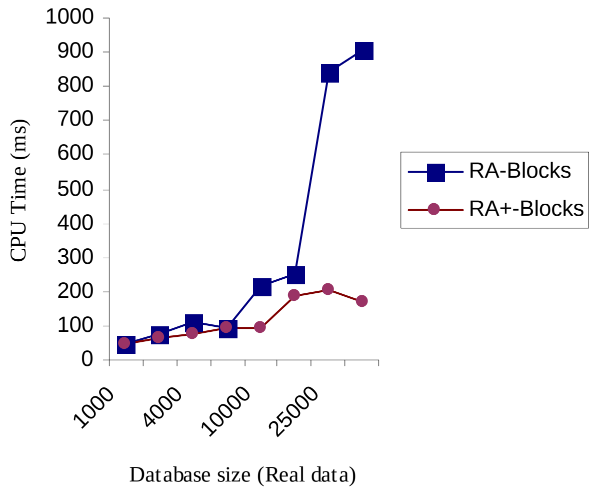


Fig.10. CPU time according to dimensionality



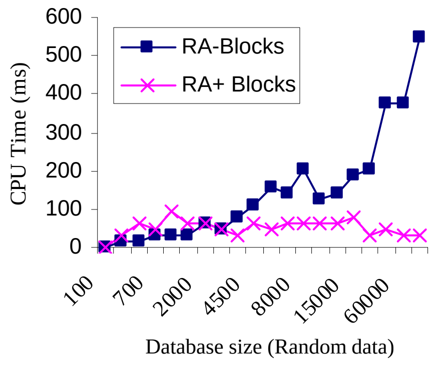


Fig.11. CPU time according to the database size

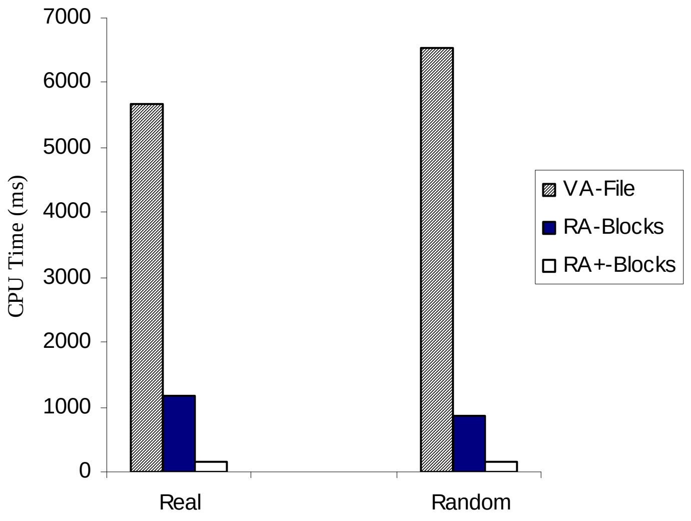


Fig.12. CPU time, d=60, K-NN=5, capacity=5, S=10000

We note that our method outperforms also the VA-File. In fact, like RA-Blocks, the RA$^+$-Blocks method is based on regions filtering approach to calculate the K-NN of the query request. Then, the computation time in the filtering phase and the access phase has been reduced, compared to the VA-file method. This, because in one hand, the lower and upper bounds in the filtering phase are calculated for each region and not for each point; In the other hand, the neighboring vectors in RA$^+$-Blocks structure are more likely to be in the same physical page or consecutives pages. Therefore, the CPU time is reduced compared to the VA-File.

Finally, we illustrate the performance of RA$^+$-Blocks on both real and random image database. RA$^+$-Blocks outperforms VA-File and RA-Blocks in high dimensional data space and in large image databases. This is one of the best solution to solve the problem of the “dimensionality curse” and to index a very large image databases, currently employed in different area such as geographic information systems (GIS), medical imaging systems, documents systems, military, biomedicine…

V.CONCLUSION

In this paper, we propose a new indexing method based on the filtering approach, to enhance the nearest neighbor search in multimedia databases. The RA$^+$-Blocks outperforms other current methods especially with large amount of data or with very high dimensionality. With RA$^+$-Blocks method, the partitioning strategy of data space reduces the K-NN search time by eliminating the empty and not very dense regions. This strategy of subdivision guarantees that the entire index structure has almost 100% storage utilization, and reduces the size of the RA$^+$-Blocks index structure. In addition, RA$^+$-Blocks index generated a very small approximations file which can be kept in the main memory in most cases; then the system doesn't have to access the disk. Another advantage of this technique is the optimal use of the disk pages since the close points are often stored either in the same, or on neighboring disk pages.

Our method outperforms both RA-Blocks and VA-File especially in large multimedia databases and even for a very high dimensionality.

Among further works, we propose to introduce, “relevance feedback” to interactively refine the search or to fit the user's special preferences.

REFERENCES

[1] *A. Guttman. R-trees: A dynamic index structure for spatial searching. In Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pages 47-57, Boston, MA, June 1984.*

[2] *N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. ’’The R*-tree: An efficient and robust access method for points and rectangles’’ In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pages 322-331, Atlantic City, NJ, 23-25 May 1990.*

[3] *Katayama, N. & Satoh, S. The SR-Tree: An index structure for high-dimensional nearest neighbour queries. In proceedings of the ACM SIGMOD Intl. Conf. on Management of Data, Tucson, Arizona, USA pages 369-380. 1997.*

[4] *J.T. Robinson, The K-D-B-Tree: A search structure for Large Multidimensional Dynamic Indexes. In Proceedings of the ACM SIGMOD, 1981.*

[5] *S. Berchtold, D. Keim, H.-P. Kriegel, The X-tree: An index structure for high-dimensional data. In Proc. Of the Int. Conference on Very Large Databases, pages 28-39, 1996.*

[6] *S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In Proc. of the 16th ICDE, pages 577-588, 2000.*

[7] *P. Indyk and R. Motwani, “Approximate nearest neighbours: Toward removing the curse of dimensionality”, in Proc. ACM symp. Theory of computing, 1998*

[8] *R. Weber, H.-J. Schek, S. Blott, A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Space. Proceedings of the 24th VLDB Conference, USA, 1998.*

[9] *M. Verleysen, “Learning high-dimensional data”, In V. T Piuri, M. Gory, S. A. &Goras, L., editors, Limitations and future trends in neural computation, pp 141-162, IOS Press, 2003.*

[10] *G.–H. Cha, X. Zhu, D. Petrovic, "An Efficient Indexing Method for Nearest neighbor searches in High-Dimensional Image Databases,” IEEE transactions On Mutimedia, Vol 4, N°1, pp 76-87, March 2002.*

[11] *H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. “Vector Approximation based indexing for non-uniform hgh dimensional data sets. In Proceedings of the 9th ACM International Conference on Information and Knowledge Management, McLen, VA, USA, pp 202-209, 2000.*

[12] *Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, “The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation”. Proceedings of the 26th VLDB Conference, Egypt, 2000.*

[13] *Guang –Ho Cha, Chin-Wan Chung. ’’The GC-Tree: A High-Dimensional Index Structure for Similarity Search in Image Databases’’ IEEE Transactions On Mutimedia Vol 4, N°2 June 2002.*

[14] *T. Chen, M. Nakazato, T. S. Huang, Speeding up the Similarity Search in Multimedia Database. In Proceedings of IEEE ICME, 2002.*

[15] *T.K. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: A dynamic index for multidimensional object. In Proceeddings of the 13h VLDB International Conference, pages 507-578, Brighton, England, September 1987.*

[16] *Ciaccia and al. Indexing metric space with M-tree. In SEBD’97, pp 67-86, 1997.*

[17] *D. A. White, R. Jain, Similarity indexing with the SS-tree. In Proceedings of the 12th International Conference on Data Engineering, New Orleans, louisiana, USA, pages 516-523, 1996.*

[18] *J. Nievergelt, H. Hinterberger, K. Sevcik, “The grid file: An adaptable symmetric multikey files structure.” ACM Transactions on Database Systems, 9(1):38-71, Mar. 1984.*

[19] *A. Herich, H.-W. Six, and P. Widmayer. The LSD-Tree: Spatial access to multidimensional point and non point object. In Apers, P.M.G. and Wiederhold, G. editors. Proceedings of the 15th International Conference on Very Large Databases, Amsterdam, The Netherlands, pp 45-53: Morgan Kaufmann, 1989.*

[20] *C. Harris, M. Stephens, A combined corner and edge detector. In Alvey Vision Conference, pages 147-151, 1998.*

[21] *D. R. Heisterkamp and J. Peng, “Kernel Vector Approximation Files for Relevance Feedback Retrieval in Large Image Databases,” Multimedia Tools and Applications, vol 25., N° 2, pp. 175-189, June, 2005.*

[22] *Hung-Yi , Lin and P. Huang, "Perfect KDB-Tree Structure for Indexing Multidimensional Data", hired International Conference on Information technology and applications (ICITA 2005), 4-7 July 2005, Sydney, Australia.*