Indexing Metric Spaces with M-Tree.

Paolo Ciaccia
University of Bologna
148 PUBLICATIONS **4,204** CITATIONS
SEE PROFILE

Marco Patella
University of Bologna
72 PUBLICATIONS **3,743** CITATIONS
SEE PROFILE

Fausto Rabitti
Italian National Research Council
141 PUBLICATIONS **2,396** CITATIONS
SEE PROFILE

Pavel Zezula
Masaryk University
267 PUBLICATIONS **6,158** CITATIONS
SEE PROFILE

Indexing Metric Spaces with M-tree *

P. Ciaccia[1], M. Patella[1], F. Rabitti[2], P. Zezula[2,3]

[1] DEIS - CSITE-CNR, Univ. Bologna - Italy, {pciaccia,mpatella}@deis.unibo.it
[2] CNUCE-CNR, Pisa - Italy, f.rabitti@cnuce.cnr.it - zezula@iei.pi.cnr.it
[3] CVIS, Technical University, Brno - Czech Republic, zezula@cis.vutbr.cz

Abstract. M-tree is a dynamic access method suitable to index generic "metric spaces", where the function used to compute the distance between any two objects satisfies the positivity, symmetry, and triangle inequality postulates. The M-tree design fulfills typical requirements of multimedia applications, where objects are indexed using complex features, and similarity queries can require application of time-consuming distance functions. In this paper we describe the basic search and management algorithms of M-tree, introduce several heuristic *split policies*, and experimentally evaluate them, considering both I/O and CPU costs. Results also show that M-tree performs better than R*-tree on high-dimensional vector spaces.

1 Introduction

Among the many research challenges which the incoming multimedia (MM) era entails – including data placement, presentation, synchronization, etc. – *content-based retrieval* plays a dominant role [GR95]. Fast retrieval from large MM repositories of images, sounds, texts, and videos, which better satisfy the information needs of users is becoming a compelling requisite for medical, scientific, legal, and art applications, to say a few. To this end, MM objects are first characterized by means of relevant *features* (shapes, textures, patterns, and colors for images [FEF$^+$94, VM95], loudness and harmonicity for sounds [WBKW96], shots and objects' trajectories for videos, etc.), then *similarity queries* – exact-match queries are not appropriate for content-based retrieval – are issued to retrieve the objects which are most similar to a query object. This process thus requires a *distance function* to measure the similarity of feature values.

The intrinsic multi-dimensional nature of features has suggested using *spatial* access methods (SAMs), such as R-tree [Gut84] and its variants [SRF87, BKSS90], to support similarity queries. However, applicability of SAMs is limited by two major factors:

1. for indexing purposes, objects are to be represented by means of feature values in a *vector space* of dimensionality *Dim*;

* This work has been partially supported by ESPRIT LTR project no. 9141, HERMES (Foundations of **H**igh P**er**formance **M**ultimedia Information Manag**e**ment **S**ystems). P. Zezula has also been supported by Grants GACR No. 102/96/0986 and KONTAKT No. PM96 S028.

2. the distance function has to be an L_p norm, such as the Euclidean (L_2) or Manhattan (L_1) distance.[1]

The second restriction rules out the possibility of using distance functions which account for correlation (or "cross-talk") between feature values, such as the Mahalanobis distance used to compare images from their color histograms [FEF$^+$94].[2] From a performance point of view, it has to be remarked that SAMs have been designed by assuming that comparison of feature values has a negligible CPU cost with respect to disk I/O. Since this is not always the case in MM applications, SAMs may turn to be CPU-bound when indexing high-dimensional spaces.

A more recent and general approach to the "similarity indexing" problem has led to the development of so-called *metric trees* [Uhl91]. Metric trees only consider relative distances of objects to organize and partition the search space, and just require the distance function to be a *metric*, which allows the *triangle inequality* property to be applied (see Section 2). Although the effectiveness of metric trees has been clearly demonstrated [Chi94, Bri95, BO97], current designs suffer from being intrinsically *static*, which limits their applicability in dynamic database environments. Contrary to SAMs, known metric trees have only tried to reduce the number of distance computations, paying no attention to I/O costs.

The M-tree is a balanced paged metric tree, which has been explicitly designed to act as a dynamic database access method [ZCR96]. Since the design of M-tree aims to combine advantages of metric trees and database access methods, performance optimization concerns both CPU (distance computations) and I/O costs. Besides dynamicity, M-tree differs from previous metric trees in that it does not dictate a specific algorithm to organize the indexed objects, rather only general principles are specified and details are encapsulated in the implementation of maintenance methods. This approach makes the M-tree a flexible structure, able to balance between construction and search costs.

In this article, after presenting the basic M-tree principles and algorithms (Section 3), in Section 4 we discuss alternative algorithms for implementing the split method, which manages node overflow. Section 5 presents experimental results obtained from the prototype M-tree we have built within the GiST framework [HNP95]. It is shown that good performance levels are attained in the processing of similarity queries, and that M-tree outperforms R*-tree in high-dimensional vector spaces.

2 Preliminaries

Indexing a *metric space* aims to efficiently support similarity queries, whose purpose is to retrieve DB objects which are most similar to a query object, and where the (dis)similarity of any two objects is measured by a specific metric d. Formally, a *metric space* is a pair, $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is a domain of feature

[1] The L_p distance is defined as $L_p(x,y) = \left(\sum_{j=1}^{Dim} \mid x[j] - y[j] \mid^p\right)^{1/p}$.

[2] The Mahalanobis distance is $\sqrt{(x-y)^T C^{-1} (x-y)}$, where C is a covariance matrix.

values – the indexing *keys* – and d is a total (distance) function with the following properties:[3]

(i) $d(O_x, O_y) = d(O_y, O_x)$ (*symmetry*)
(ii) $0 < d(O_x, O_y) < \infty \ (O_x \neq O_y)$, and $d(O_x, O_x) = 0$ (*non negativity*)
(iii) $d(O_x, O_y) \leq d(O_x, O_z) + d(O_z, O_y)$ (*triangle inequality*)

Example 1. Consider a 2-D shape, $O_x = \{O_{x,i}\}$, represented as a set of relevant points (e.g. high curvature points) [HKR93]. The *Hausdorff* metric computes the distance of two sets of points, O_x and O_y, as:

$$d_H(O_x, O_y) = \max\{h(O_x, O_y), h(O_y, O_x)\}$$

where $h(O_x, O_y) = \max_i \min_j L_p(O_{x,i}, O_{y,j})$ is the maximum L_p distance from a point of O_x to any point of O_y.

Example 2. The Levenshtein (*edit*) distance of two strings, $d_L(O_x, O_y)$, is a metric which counts the minimal number of symbols that have to be inserted, deleted, or substituted, to transform O_x into O_y (e.g. $d_L(\texttt{head}, \texttt{tail}) = 4$).

The two most important types of similarity queries are the *range query*, where a minimum similarity (maximum distance) value is specified, and the *k nearest neighbors query*, where the cardinality of the result set is an input parameter.
Range query: *Given a query object* $Q \in \mathcal{D}$ *and a* maximum search distance $r(Q)$*, the range query* $\texttt{range}(Q, r(Q))$ *selects all the indexed objects* O_j *such that* $d(O_j, Q) \leq r(Q)$. □
k nearest neighbors (k-NN) query: *Given a query object* $Q \in \mathcal{D}$ *and an integer* $k \geq 1$*, the k-NN query* $\texttt{NN}(Q, k)$ *selects the* k *indexed objects which have the shortest distance from* Q*, according to the distance function* d. □

There have already been some attempts to tackle the difficult metric space indexing problem. The *Vantage Point* (VP) tree [Chi94] partitions a data set according to the distances the objects have with respect to a reference (vantage) point. The median value of such distances is then used as a separator to partition objects into two balanced subsets, to which the same procedure can be recursively applied. The MVP-tree [BO97] uses *multiple vantage points*. In this way, given, say, 2 vantage points, the metric space gets partitioned into 2^2 regions. The MVP-tree also makes use of pre-computed distances to speed up the search and to reduce the number of distance computations at query execution time. The GNAT design [Bri95] applies a different – so-called *generalized hyperplane* [Uhl91] – partitioning style. According to the basic formulation of this strategy, two reference objects are chosen, and the data set is split by assigning each object to the closest reference object. Again, this procedure is recursively applied up to the desired granularity level.

Since all above designs build the index in a top-down way, the tree is not guaranteed to stay balanced in case of insertions and deletions, thus requiring costly reorganizations to prevent performance degradation.

[3] In order to simplify the presentation, we sometimes refer to O_j as an object, rather than as the feature value of the object itself.

3 The M-tree

The M-tree organizes the objects into fixed-size nodes,[4] which correspond to regions of the metric space. Nodes of the M-tree can store up to M entries – this is the *capacity* of the nodes. For each indexed DB object, one entry with format

$$\texttt{entry}(O_j) \texttt{ = [} O_j\texttt{, oid}(O_j)\texttt{, } d(O_j, P(O_j)) \texttt{]}$$

is stored in a leaf node. In $\texttt{entry}(O_j)$, $\texttt{oid}(O_j)$ is the identifier of the object which resides on a separate data file,[5] O_j are the feature values of the object (i.e., $O_j \in \mathcal{D}$), and $d(O_j, P(O_j))$ is the distance between O_j and $P(O_j)$, the *parent* object of O_j (see below).

An entry in an internal (non-leaf) node stores a feature value, O_r, also called a *routing object*, and a *covering radius*, $r(O_r) > 0$. The entry for routing object O_r includes a pointer, $\texttt{ptr}(T(O_r))$, to the root of sub-tree $T(O_r)$ – the *covering tree* of O_r – and $d(O_r, P(O_r))$, the distance from the parent object:[6]

$$\texttt{entry}(O_r) \texttt{ = [} O_r\texttt{, ptr}(T(O_r))\texttt{, } r(O_r)\texttt{, } d(O_r, P(O_r)) \texttt{]}$$

The semantics of the covering radius is captured by the following

Property 1 *The covering radius of a routing object, O_r, satisfies the inequality $d(O_j, O_r) \le r(O_r)$, for each object O_j stored in the covering tree of O_r.*

A routing object thus defines a region in the metric space $\mathcal{M}$, centered on O_r and with radius $r(O_r)$, and O_r is the parent of each object O_j stored in the node referenced by $\texttt{ptr}(T(O_r))$, i.e. $O_r \equiv P(O_j)$ (see Figure 1). This implies that the M-tree organizes the metric space into a set of, possibly overlapping, regions, to which the same principle is recursively applied.

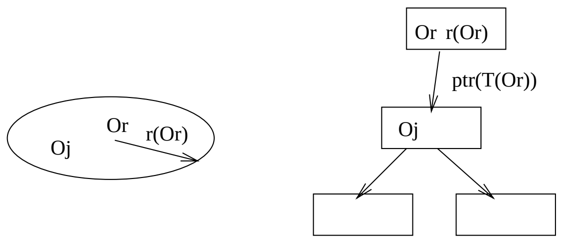

Fig. 1. A routing object, O_r, has a covering radius, $r(O_r)$, and references a covering tree, $T(O_r)$.

[4] Nothing would prevent using variable-size nodes, as it is done in the X-tree [BKK96]. For simplicity, however, we do not consider this possibility here.

[5] Of course, the M-tree can also be used as a primary organization, with the whole objects stored in the leaf nodes.

[6] This is undefined for entries in the root node.

3.1 How M-tree Grows

As any other dynamic balanced tree, M-tree grows in a bottom-up fashion. The overflow of a node N is managed by allocating a new node, N', at the same level of N, partitioning the $M+1$ entries among the two nodes, and then posting (*promoting*) relevant information to the parent node, N_p. When the root splits, a new root is created and the M-tree grows by one level. The `Split` method is concisely described as follows:

```
Split($N$: M-tree_node; $E$: M-tree_entry)
{   let $\mathcal{N}$ be the set of entries of node $N$, including the new entry $E$;
    if $N$ is not the current M-tree root
    then let $O_p$ be the parent object of $N$, stored in the $N_p$ node;
    Allocate a new node $N'$;
    Promote($\mathcal{N}$,$O_{p_1}$,$O_{p_2}$);
    Partition($\mathcal{N}$,$O_{p_1}$,$O_{p_2}$,$\mathcal{N}_1$,$\mathcal{N}_2$);
    Store in node $N$ entries in $\mathcal{N}_1$ and in node $N'$ entries in $\mathcal{N}_2$;
    if $N$ is the current root
    then { Allocate a new root node, $N_p$;
           Store entry($O_{p_1}$) and entry($O_{p_2}$) in $N_p$; }
    else { Replace entry($O_p$) with entry($O_{p_1}$) in $N_p$;
           if node $N_p$ is full
           then Split($N_p$, entry($O_{p_2}$))
           else store entry($O_{p_2}$) in $N_p$; }}
```

The `Promote` method chooses, according to some specific criterion, two routing objects, O_{p_1} and O_{p_2}, to be inserted into the parent node, N_p. The `Partition` method partitions the $(M+1)$ entries of the overflown node (the $\mathcal{N}$ set) into two disjoint subsets, $\mathcal{N}_1$ and $\mathcal{N}_2$, which are then stored in nodes N and N', respectively. A specific implementation of `Promote` and `Partition` defines what we call a *split policy* . Unlike other (static) metric tree designs, each relying on a specific criterion to organize objects, M-tree offers the possibility of implementing alternative split policies, which can be tuned depending on application needs (see Section 4).

Every split policy has to respect the semantics of covering radii, as specified by Property 1. If the split node, N, is a leaf, this is guaranteed by setting

$$r(O_{p_1}) = max\{d(O_j, O_{p_1}) | O_j \in \mathcal{N}_1\}$$

In general, the covering radius of a routing object pointing to a leaf equals the maximum distance between the routing object itself and the objects in the leaf.

When the split involves an internal node, N, each entry O_j in $\mathcal{N}_1$ has a non-null covering radius, $r(O_j)$. By setting

$$r(O_{p_1}) = \max\{d(O_j, O_{p_1}) + r(O_j) | O_j \in \mathcal{N}_1\}$$

it is guaranteed, by the triangle inequality property, that no object in $T(O_{p_1})$ can be distant from O_{p_1} more than $r(O_{p_1})$. Figure 2 shows this in the case $\mathcal{M} = (\Re^2, L_2)$, i.e. the real plane with the Euclidean distance.

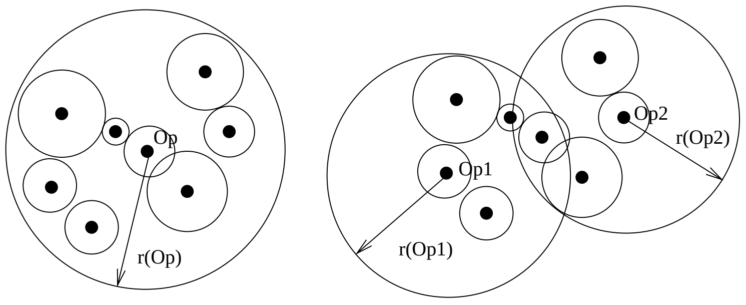

Fig. 2. Split of an internal node on the $(\Re^2, L_2)$ metric space.

3.2 Processing Similarity Queries

The M-tree algorithms for processing similarity queries aim to reduce the number of accessed nodes as well as the number of distance computations. This is particularly relevant when the search turns to be CPU- rather than I/O-bound, which might be the case for computationally intensive distance functions. For this purpose, all the (pre-computed) distances stored in the M-tree nodes, i.e. $d(O_r, P(O_r))$ and $r(O_r)$, are used.

Range Queries The query `range`$(Q, r(Q))$ selects all the DB objects such that $d(O_j, Q) \leq r(Q)$. Algorithm `RangeSearch` starts from the root node and recursively traverses all the paths which cannot be excluded from leading to qualifying objects.

```
RangeSearch($N$: M-tree_node, $Q$: query_object, $r(Q)$: search_radius)
{   let $O_p$ be the parent object of node $N$;
    if $N$ is not a leaf
    then { for each entry($O_r$) in $N$ do:
           if $| d(O_p, Q) - d(O_r, O_p) | \leq r(Q) + r(O_r)$
           then { Compute $d(O_r, Q)$;
                  if $d(O_r, Q) \leq r(Q) + r(O_r)$
                  then RangeSearch(*ptr($T(O_r)$),$Q$,$r(Q)$); }}
    else { for each entry($O_j$) in $N$ do:
           if $| d(O_p, Q) - d(O_j, O_p) | \leq r(Q)$
           then { Compute $d(O_j, Q)$;
                  if $d(O_j, Q) \leq r(Q)$
                  then add oid($O_j$) to the result; }}}
```

Since, when accessing node N, the distance between Q and O_p, the parent object of N, has already been computed, it is possible to prune a sub-tree without having to compute any new distance at all. The condition applied for pruning is as follows.

Lemma 1. *If $d(O_r, Q) > r(Q) + r(O_r)$, then, for each object O_j in $T(O_r)$, it is $d(O_j, Q) > r(Q)$. Thus, $T(O_r)$ can be safely pruned from the search.*

In fact, since $d(O_j, Q) \geq d(O_r, Q) - d(O_j, O_r)$ (by the triangle inequality) and $d(O_j, O_r) \leq r(O_r)$ (by def. of covering radius), it is $d(O_j, Q) \geq d(O_r, Q) - r(O_r)$. Since, by hypothesis, it is $d(O_r, Q) - r(O_r) > r(Q)$, the result follows.

In order to apply Lemma 1, the $d(O_r, Q)$ distance has to computed. This can be avoided by taking advantage of the following result.

Lemma 2. *If* $| d(O_p, Q) - d(O_r, O_p) | > r(Q) + r(O_r)$, *then* $d(O_r, Q) > r(Q) + r(O_r)$.

This is a direct consequence of the triangle inequality, which guarantees that both $d(O_r, Q) \geq d(O_p, Q) - d(O_r, O_p)$ and $d(O_r, Q) \geq d(O_r, O_p) - d(O_p, Q)$ hold (see Figure 3). The same optimization principle is applied to leaf nodes as well. Experimental results (see Section 5) show that this technique can save up to 40% distance computations. The only case where distances are necessarily to be computed is when dealing with the root node, for which O_p is undefined.

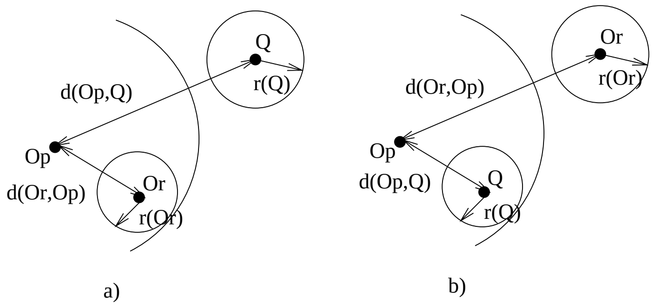

Fig. 3. The figure shows how Lemma 2 is used to avoid computing distances. Case a): $d(O_r, Q) \geq d(O_p, Q) - d(O_r, O_p) > r(Q) + r(O_r)$; Case b): $d(O_r, Q) \geq d(O_r, O_p) - d(O_p, Q) > r(Q) + r(O_r)$.

Nearest Neighbor Queries The `k-NN_Search` algorithm retrieves the k nearest neighbors of a query object Q – it is assumed that at least k objects are indexed by the M-tree. We use a branch-and-bound technique, quite similar to the one designed for R-trees [RKV95], which utilizes two global structures: a priority queue, `PR`, and a k-elements array, `NN`, which, at the end of execution, will contain the result.

`PR` is a queue of pointers to *active sub-trees*, i.e. sub-trees where qualifying objects can be found. With the pointer to (the root of) sub-tree $T(O_r)$, a lower bound, $d_{min}(T(O_r))$, on the distance of any object in $T(O_r)$ from Q is also kept. The lower bound is

$$d_{min}(T(O_r)) = \max\{d(O_r, Q) - r(O_r), 0\}$$

since no object in $T(O_r)$ can have a distance from Q less than $d(O_r, Q) - r(O_r)$. These bounds are used by the `ChooseNode` function to extract from `PR` the next node to be examined. Since the pruning criterion of `k-NN_Search` is dynamic –

the search radius is the distance between Q and its *current* k-th nearest neighbor – the order in which nodes are visited can affect performance. The heuristic criterion implemented by the ChooseNode function is to select the node for which the d_{min} lower bound is minimum. Other criteria do not lead to a better performance, according to our experimental observations.

```
ChooseNode(PR: priority_queue): M-tree_node
{   let $d_{min}(T(O_r^*))$ = $\min\{d_{min}(T(O_r))\}$, considering all the entries in PR;
    Remove the entry [ptr($T(O_r^*)$),$d_{min}(T(O_r^*))$] from PR;
    return *ptr($T(O_r^*)$); }
```

At the end of execution, the i-th entry of the NN array will have value NN[i] = [oid(O_j),$d(O_j, Q)$], with O_j being the i-th nearest neighbor of Q. The distance value in the i-th entry is denoted as d_i, so that d_k is the largest distance value in NN. Clearly, d_k plays the role of a *dynamic search radius*, since any sub-tree for which $d_{min}(T(O_r)) > d_k$ can be safely pruned.

Entries of the NN array are initially set to NN[i] = [_,∞] (i= $1, \ldots, k$), i.e. oid's are undefined and $d_i = \infty$. As the search starts and (internal) nodes are accessed, the idea is to compute, for each sub-tree $T(O_r)$, an upper bound, $d_{max}(T(O_r))$, on the distance of any object in $T(O_r)$ from Q. The upper bound is set to

$$d_{max}(T(O_r)) = d(O_r, Q) + r(O_r)$$

Consider the simplest case $k = 1$, two sub-trees, $T(O_{r_1})$ and $T(O_{r_2})$, and assume that $d_{max}(T(O_{r_1})) = 5$ and $d_{min}(T(O_{r_2})) = 7$. Since $d_{max}(T(O_{r_1}))$ guarantees that an object whose distance from Q is at most 5 exists in $T(O_{r_1})$, $T(O_{r_2})$ can be pruned from the search. The d_{max} bounds are inserted in appropriate positions in the NN array, just leaving the oid field undefined. The k-NN_Search algorithm is given below.

```
k-NN_Search($T$: M-tree_root_node, $Q$: query_object, $k$: integer)
{   PR = [$T$,_];
    for i = 1 to $k$ do: NN[i] = [_,$\infty$];
    while PR $\neq \emptyset$ do:
    { Next_Node = ChooseNode(PR);
      k-NN_NodeSearch(Next_Node,$Q$,$k$); }}
```

The k-NN_NodeSearch method implements most of the search logic. On an internal node, it first determines active sub-trees and insert them into the PR queue. Then, if needed, it calls the NN_Update function (not specified here) to perform an ordered insertion in the NN array and receives back a (possibly new) value of d_k. This is then used to remove from PR all sub-trees for which the d_{min} lower bound exceeds d_k. Similar actions are performed in leaf nodes. In both cases, the optimization to reduce the number of distance computations, which uses the pre-computed distances from the parent object, is applied.

```
k-NN_NodeSearch($N$: M-tree_node, $Q$: query_object, $k$: integer)
{   let $O_p$ be the parent object of node $N$;
    if $N$ is not a leaf
    then { for each entry($O_r$) in $N$ do:
           if $| d(O_p, Q) - d(O_r, O_p) | \leq d_k + r(O_r)$
```

```
        then { Compute $d(O_r, Q)$;
               if $d_{min}(T(O_r)) \leq d_k$
               then { add [ptr($T(O_r)$),$d_{min}(T(O_r))$] to PR;
                      if $d_{max}(T(O_r)) < d_k$
                      then { $d_k$ = NN_Update([_,$d_{max}(T(O_r))$]);
                             Remove from PR all entries
                                for which $d_{min}(T(O_r)) > d_k$; }}}}
else { for each entry($O_j$) in $N$ do:
       if $| d(O_p, Q) - d(O_j, O_p) | \leq d_k$
       then { Compute $d(O_j, Q)$;
              if $d(O_j, Q) \leq d_k$
              then { $d_k$ = NN_Update([oid($O_j$),$d(O_j, Q)$]);
                     Remove from PR all entries
                        for which $d_{min}(T(O_r)) > d_k$; }}}}
```

3.3 Inserting Objects

The Insert algorithm recursively descends the M-tree to locate the most suitable leaf node for accommodating a new object, O_n, possibly triggering a split if the leaf is full. The basic rationale to determine the "most suitable" leaf node is to descend, at each level of the tree, along a sub-tree, $T(O_r)$, for which no enlargement of the covering radius is needed, i.e. $d(O_r, O_n) \leq r(O_r)$. In case multiple sub-trees with this property exist, the one for which object O_n is closest to O_r is chosen. This heuristic criterion tries to obtain well-clustered sub-trees, which has a beneficial effect on performance.

If no routing object for which $d(O_r, O_n) \leq r(O_r)$ exists – thus a covering radius has to be enlarged – our choice is to minimize the *increase* of the covering radius, that is, $d(O_r, O_n) - r(O_r)$. This is tightly related to the heuristic criterion that suggests to minimize the overall "volume" covered by routing objects in the current node. Algorithm Insert summarizes above arguments.

```
Insert($N$: M-tree_node, entry($O_n$): M-tree_entry)
{   let $\mathcal{N}$ be the set of entries in node $N$;
    if $N$ is not a leaf
    then { let $\mathcal{N}_{in}$ = set of entries such that $d(O_r, O_n) \leq r(O_r)$;
           if $\mathcal{N}_{in} \neq \emptyset$
           then let entry($O_r^*$) $\in \mathcal{N}_{in}$: $d(O_r^*, O_n)$ is minimum;
           else { let entry($O_r^*$) $\in \mathcal{N}$: $d(O_r^*, O_n) - r(O_r^*)$ is minimum;
                  let $r(O_r^*)$ = $d(O_r^*, O_n)$; }
           Insert(*ptr($T(O_r^*)$),entry($O_n$)); }
    else { if $N$ is not full
           then store entry($O_n$) in $N$
           else Split($N$,entry($O_n$)); }}
```

4 Split Policies

The "ideal" split policy should select the two objects to be promoted, O_{p_1} and O_{p_2}, and partition entries in such a way that the two so-obtained regions would have minimum "volume" and minimum "overlap". Both criteria aim to improve the effectiveness of search algorithms, since having small (low volume) regions leads to well-clustered trees and reduces the amount of indexed *dead space* – space where no object is present – and having small (possibly null) overlap between regions reduces the number of paths to be traversed for answering a query.

The minimum-volume criterion leads to devise split policies which try to minimize the values of the covering radii, whereas the minimum-overlap requirement suggests that, for fixed values of the covering radii, the distance between the two chosen reference objects should be maximized.[7]

Besides above requirements, which are quite "standard" also for spatial access methods [BKSS90], the possible high CPU cost of computing distances should also be taken into account. This suggests that even naïve policies (e.g. a random choice of routing objects), which however execute few distance computations, are worth considering.

4.1 Choosing the Routing Objects

The `Promote` method determines, given a set of entries, $\mathcal{N}$, two objects to be promoted and stored into the parent node (see Section 3.1). The specific algorithms we consider can first be classified according to whether or not they "confirm" the parent object in its role.

Definition 3. A *confirmed* split policy chooses one of the two objects to be promoted, say O_{p_1}, to be the parent object itself, O_p, of the split node.

In other terms, a confirmed split policy just "extracts" a region, centered on the second routing object, O_{p_2}, from the region which will still remain centered on O_p. In general, this simplifies split execution and reduces the number of distance computations.

The alternatives we describe for implementing `Promote` are only a selected subset of the whole set we have experimentally evaluated. Other policies are described in [CPRZ97].

`m_RAD` The "minimum (sum of) RADii" algorithm is the most complex in terms of distance computations. It considers all possible pairs of objects and, after partitioning the set of entries, promotes the pair of objects for which the sum of covering radii, $r(O_{p_1}) + r(O_{p_2})$, is minimum.

`RANDOM` This variant simply selects in a random way the reference object(s). Although this does not appear to be a "smart" strategy, it is fast and its performance can be used as a reference for other policies.

`SAMPLING` This is the `RANDOM` policy, but iterated over a sample of objects of size $s > 1$. For each of the $s(s-1)/2$ pairs of objects in the sample, entries are distributed and potential covering radii established. The pair for which the

[7] Note that, without a detailed knowledge of the distance function, it is impossible to quantify the exact amount of overlap of two non-disjoint regions in a metric space.

resulting sum of covering radii, $r(O_{p_1})+r(O_{p_2})$, is minimum is then selected. In case of confirmed promotion, only s different distributions are tried. The sample size in our experiments was set to 1/10-th of node capacity.

`M_LB_DIST` The acronym stands for "Maximum Lower Bound on DISTance". This policy differs from previous ones in that it only uses the pre-computed stored distances. In the confirmed version, where $O_{p_1} \equiv O_p$, the algorithm determines O_{p_2} as the farthest object from O_p, that is

$$d(O_{p_2}, O_p) = \max_j \{d(O_j, O_p)\}$$

When $O_{p_1} \neq O_p$, the two promoted objects are chosen so that

$$\begin{aligned} d(O_{p_1}, O_p) &= \min_j \{d(O_j, O_p)\} \qquad\qquad (O_{p_1} \neq O_p) \\ d(O_{p_2}, O_p) &= \max_j \{d(O_j, O_p)\} \end{aligned}$$

The distance between the two routing objects is then guaranteed to be at least $d(O_{p_2}, O_p) - d(O_{p_1}, O_p)$, and no other choice can lead to a higher bound.

4.2 Distribution of the Entries

Given a set of entries $\mathcal{N}$ and the two routing objects O_{p_1} and O_{p_2}, the problem is how to efficiently partition $\mathcal{N}$ into two subsets, $\mathcal{N}_1$ and $\mathcal{N}_2$. For this purpose we consider two basic strategies. The first one is based on the idea of the *generalized hyperplane decomposition* [Uhl91] and leads to unbalanced splits, whereas the second obtains a balanced distribution. They can be shortly described as follows.

`Generalized Hyperplane:` Assign each object $O_j \in \mathcal{N}$ to the nearest routing object, that is, if $d(O_j, O_{p_1}) \leq d(O_j, O_{p_2})$ then assign O_j to $\mathcal{N}_1$, else assign O_j to $\mathcal{N}_2$.

`Balanced:` Compute $d(O_j, O_{p_1})$ and $d(O_j, O_{p_2})$ for all $O_j \in \mathcal{N}$. Repeat until $\mathcal{N}$ is empty:
- Assign to $\mathcal{N}_1$ the nearest neighbor of O_{p_1} in $\mathcal{N}$ and remove it from $\mathcal{N}$;
- Assign to $\mathcal{N}_2$ the nearest neighbor of O_{p_2} in $\mathcal{N}$ and remove it from $\mathcal{N}$.

Depending on data distribution and on how the routing objects are chosen, an unbalanced split policy can lead to a better objects' partitioning, because of the additional degree of freedom one obtains. In particular, it has to be remarked that, while obtaining a balanced split with spatial access methods forces the enlargement of regions along only the necessary dimensions, in a metric space the consequent increase of the covering radius would propagate to *all* the "dimensions".

An intermediate behavior can be obtained by combining the two above algorithms and setting a minimum threshold on node utilization. If at least $m \leq M/2$ entries per node are required, the `Balanced` distribution can be applied to the first $2m$ objects, after which `Generalized Hyperplane` could be used. In the following, however, we do not investigate the effects of this variant.

5 Performance Evaluation

In this section we provide experimental results on the performance of M-tree on synthetic data sets, as obtained from the procedure described in [JD88, appendix H] which generates normally-distributed clusters in a *Dim*-D vector space. Unless otherwise stated, the number of clusters is 10, the variance is $\sigma^2 = 0.1$, and clusters' centers are uniformly distributed, with Figure 4 showing a 2-D sample. Our implementation is based on the GiST C++ package [HNP95], and uses a constant node size of 4 KBytes. This implies that the node capacity, M, is inversely related to the dimensionality of the indexed data set. Finally, distance is evaluated using the L_∞ metric, i.e. $L_\infty(O_x, O_y) = \max_{j=1}^{Dim}\{\mid O_x[j] - O_y[j] \mid\}$, which leads to hyper-cubic search (and covering) regions.

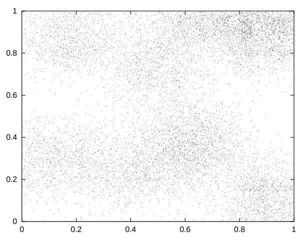

Fig. 4. A sample data set used in the experiments.

5.1 Balanced vs. Unbalanced Split Policies

We first compare the performance of the `Balanced` and `Generalized Hyperplane` implementations of the `Partition` method (see Section 4.2). Table 1 shows the overhead of a balanced policy with respect to the corresponding unbalanced one to process range queries with side $\sqrt[Dim]{0.04}$ ($Dim = 2, 10$) on 10^4 objects. Similar results were obtained also for larger dimensionalities and for all the policies not shown here. In the table, as well as in all other figures, a "confirmed" split policy is identified by the suffix `1`, whereas `2` designates a "non-confirmed" policy (see Definition 3).

		`RANDOM_1`	`SAMPLING_1`	`M_LB_DIST_1`	`RANDOM_2`	`m_RAD_2`
$Dim = 2$	volume ovh.	4.60	4.38	3.90	4.07	1.69
	dist. ovh, I/O ovh.	2.27, 2.11	1.97, 1.76	1.93, 1.57	2.09, 1.96	1.40, 1.30
$Dim = 10$	volume ovh.	1.63	1.31	1.49	2.05	2.40
	dist. ovh, I/O ovh.	1.58, 1.18	1.38, 0.92	1.34, 0.91	1.55, 1.39	1.69, 1.12

Table 1. Balanced vs. unbalanced split policies.

The first value in each entry pair refers to distance computations (CPU cost) and the second value to page reads (I/O costs). The most important observation is that `Balanced` leads to a considerable CPU overhead and also increases I/O costs. This depends on the total volume covered by an M-tree – the sum of the volumes covered by all its routing objects – as shown by the "volume overhead" lines in the table. For instance, on 2-D data sets, using `Balanced` rather than `Generalized Hyperplane` with the `RANDOM_1` policy leads to an M-tree for which the covered volume is 4.60 times larger. Because of these results, in the following all the split policies are based on `Generalized Hyperplane`.

5.2 Building and Searching: The Effect of Dimensionality

We now consider how the dimensionality of the data set influences the performance of M-tree. The number of indexed objects is 10^4 in all the graphs.

In general, as Figures 5 and 6 show, the number of distance computations decreases with growing dimensionalities, whereas I/O costs have an inverse trend. The explanation is that increasing *Dim* reduces the node capacity, and so the number of distances computed by the insertion and split algorithms, but leads to larger trees (see also Figure 12). The reduction of distance computations is particularly evident for `m_RAD_2`, whose CPU split costs grow with the square of node capacity. In general, the fastest split policy is `RANDOM_2`, while the slowest is `m_RAD_2`.

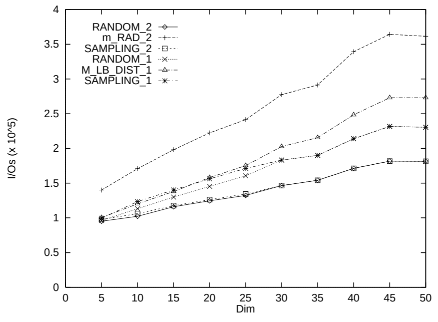

Fig. 5. I/O costs for building M-tree.

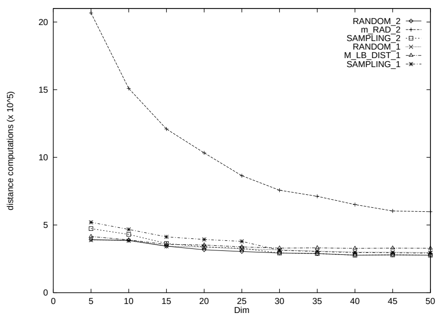

Fig. 6. Distance computations for building M-tree.

Performance on 10-NN query processing is summarized in Figures 7 and 8, where number of page I/O's and distance selectivities are shown, respectively – distance selectivity is just the ratio of computed distances to the total number of objects.

It can be observed that policies based on "non-confirmed" promotion, such as `m_RAD_2` and `RANDOM_2`, perform better that "confirmed" policies as to I/O costs, especially on high-dimensional data where they can save up to 25% I/O's. This can be attributed to the better object clustering that such policies can obtain. As to distance selectivity, differences emerge only with high values of *Dim*, and favor `m_RAD_2` and `M_LB_DIST_1`.

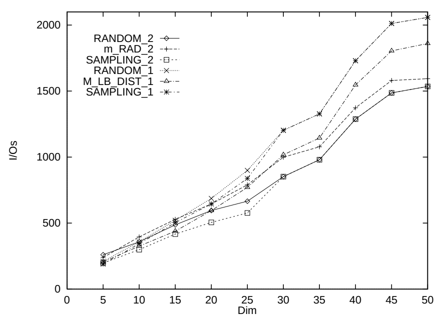

Fig. 7. I/O costs for processing 10-NN queries.

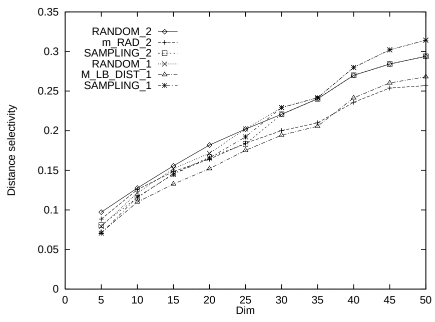

Fig. 8. Distance selectivity for 10-NN queries.

A surprising observation concerns the rather comparable performances of RANDOM_2 and m_RAD_2 policies, the latter being supposed to lead to much better results. After a careful analysis, we observed that m_RAD_2 often leads to very unbalanced entry distributions and to highly unbalanced covering radii, which has the negative effect that one of the two radii originating from a split can have a high value, thus negatively affecting the search performance.

In order to exert a better control on the values of covering radii, we experimented with alternative objective functions. To this end, we introduced the mS_RAD_2 policy, which minimizes the *squared* sum of radii $(r(O_{p_1})^2 + r(O_{p_2})^2)$, and mM_RAD_2, which minimizes the *maximum* of $r(O_{p_1})$ and $r(O_{p_2})$. Figures 9 and 10 indeed show that I/O and CPU costs for processing 10-NN queries when using mM_RAD_2 are consistently better than those obtained from m_RAD_2 and mS_RAD_2. Since these three split policies exhibit similar building costs, in subsequent analyses we only retain mM_RAD_2, and discard the other two policies.

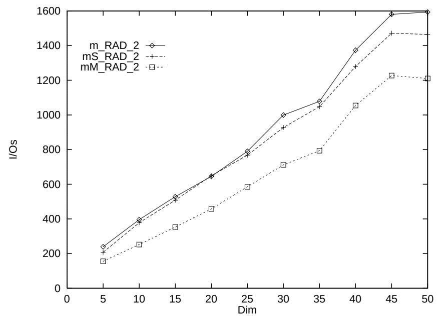

Fig. 9. I/O costs for processing 10-NN queries.

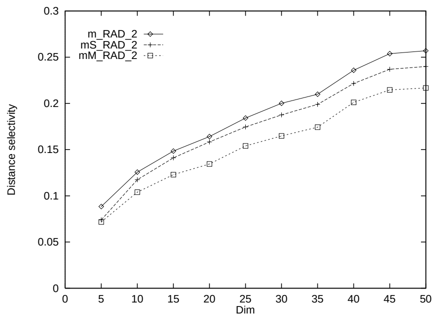

Fig. 10. Distance selectivity for 10-NN queries.

As to the "quality" of tree construction, measured by the total covered volume, Figures 11 shows that the best results are obtained from "non-confirmed" policies. It is interesting to observe that minimal volume is obtained from mM_RAD_2, which also enjoys the best search performance, but this policy leads to trees which are also the largest ones (only `M_LB_DIST_1` generates more nodes), as Figure 12 demonstrates.

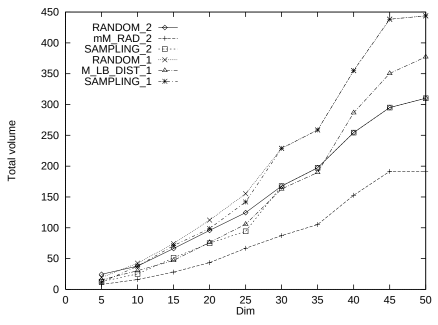

Fig. 11. Total covered volume.

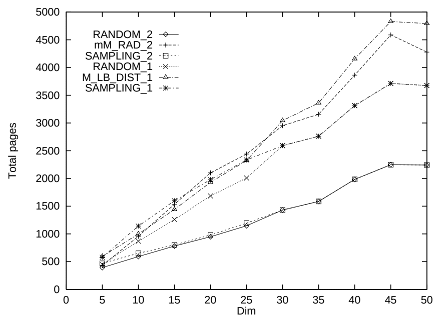

Fig. 12. Total number of pages.

5.3 The Influence of Data Set Size

A major challenge in the design of M-tree was to ensure scalability with respect to the size of the indexed data set. This addresses both aspects of efficiently building the tree and of performing well on similarity queries.

In Figures 13 and 14 we present the average number of I/O operations and distance computations per inserted object, for 2-D data sets whose size varies in the range $10^4 \div 10^5$. Both figures show a logarithmic trend, which is typical of tree-like indices and is mainly due to the increasing height of the tree.

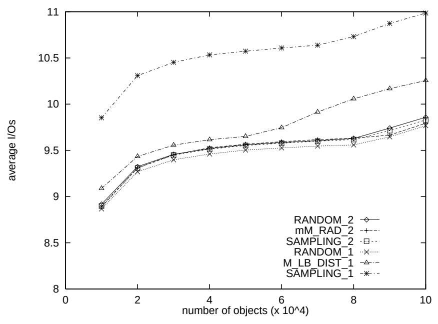

Fig. 13. Average I/O costs for building the M-tree as a function of the data set size.

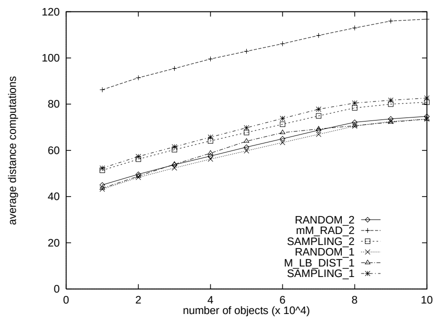

Fig. 14. Average number of distance computations for building the M-tree as a function of the data set size.

Similarly, Figures 15 and 16 show that both I/O and CPU 10-NN search costs grow logarithmically with the number of indexed objects, proving that M-tree scales well in the data set size, and that the dynamic management algorithms do not deteriorate the quality of the search.

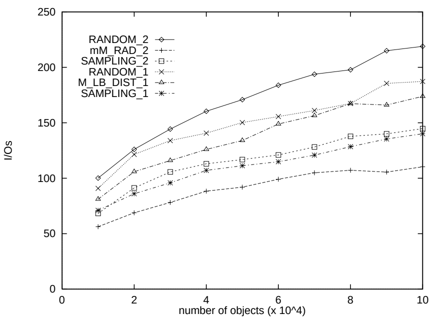

Fig. 15. I/O costs for processing 10-NN queries as a function of the data set size.

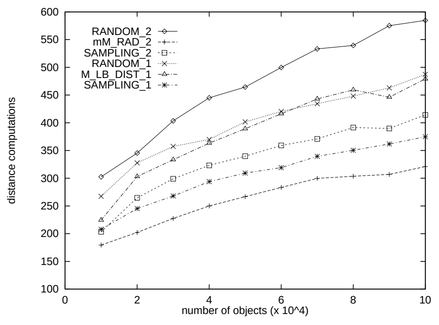

Fig. 16. Number of distance computations for 10-NN queries as a function of the data set size.

As to the relative behaviors of split policies, figures show that "cheap" policies (e.g. `RANDOM` and `M_LB_DIST_1`) are penalized by the high node capacity ($M = 120$) which arises when indexing 2-D points. Indeed, the higher M is, the more effective "complex" split policies are. This is because the number of alternatives for objects' promotion grows as M^2, thus for high values of M the probability that cheap policies perform a good choice considerably decreases.

A related aspect concerns the performance of "confirmed" random policies (`RANDOM_1` and `SAMPLING_1`) with respect to the corresponding "non-confirmed" versions (`RANDOM_2` and `SAMPLING_2`). Even if non-confirmed policies consider a larger number of alternatives[8] this is not enough to outperform confirmed policies. The explanation is as follows. When testing only a few alternatives, there is a low chance to select "good" *new* routing objects (see above). Confirmed policies just reduce the chance of such bad choices, since it can be argued that confirming the parent object in its role is a not too bad alternative.

5.4 Comparing M-tree with R*-tree

The final set of experiments we present compares M-tree with R*-tree [BKSS90], as implemented in the GiST package.[9] Although M-tree has a wider applicability range than R*-tree, we believe it is also important to contrast its performance on "traditional" domains where spatial access methods can be used as well. For the sake of clarity, in the following we will only consider three split policies for the M-tree: `RANDOM_2`, `M_LB_DIST_1`, and `mM_RAD_2`.

Results in Figures 17 and 18 compare I/O and CPU costs, respectively, to build R*-trees and M-trees. The trend of the graphs for R*-trees confirms what already observed about the influence of node capacity (see Figures 5 and 6).

[8] If the sample size is $s = M/10 = 12$, `SAMPLING_2` evaluates $s(s-1)/2 = 66$ alternatives out of 7140, whereas `SAMPLING_1` considers only $s = 12$ cases.

[9] In an earlier version of the paper, we also considered performance of R-trees. This has been dropped here, since we discovered that the GiST implementation of R-tree includes some bugs, which makes results quite unreliable.

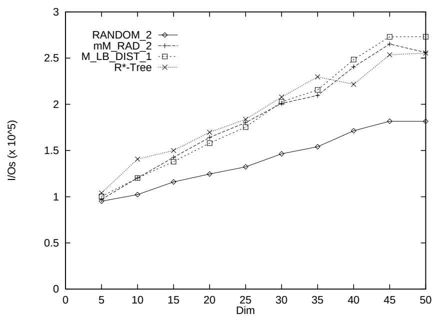

Fig. 17. I/O costs for building M- and R*-trees.

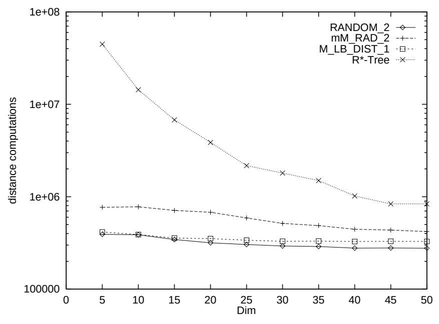

Fig. 18. Distance computations for building M- and R*-trees.

Figures 19 and 20 show the search costs for range queries with side $\sqrt[Dim]{0.01}$. It can be observed that I/O costs for R*-tree are higher than those of all M-tree variants. In order to present a fair comparison of CPU costs, Figure 20 also shows, for each M-tree split policy, a graph (labelled `(non opt)`) where the optimization for reducing the number of distance computations (see Lemma 2) is not applied. Graphs show that this optimization is highly effective, saving up to 40% distance computations (similar results were also obtained for NN-queries). Note that, even without such an optimization, M-tree is almost always more efficient than R*-tree.

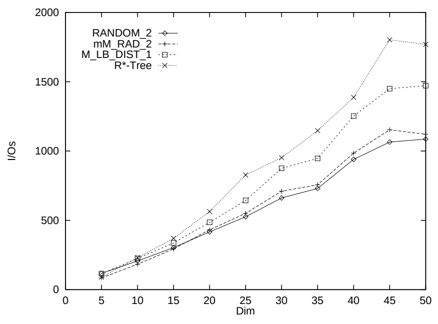

Fig. 19. I/O costs for processing range queries.

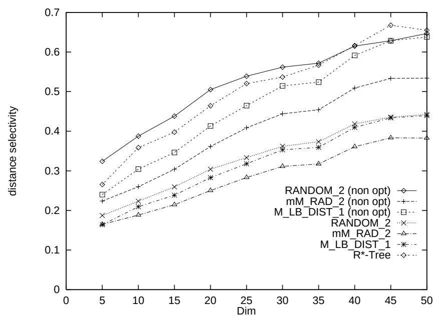

Fig. 20. Distance selectivity for range queries.

Figures 21 and 22 show, respectively, normalized I/O and CPU costs per retrieved object for processing range queries when the volume varies in the range $10^{-10} \div 10^{-1}$ on the 20-D vector space. In order to fairly compare the access methods, we did not use any optimization in the M-tree search algorithm for reducing the number of distances to be computed. Both figures show that the overall behavior of the M-tree is better than that of the R*-tree, regardless of the query volume and of the split policy.

Finally, we consider the effect of changing data set distribution. To this end, we varied the number of clusters in the range $1 \div 10,000$, while fixing the data

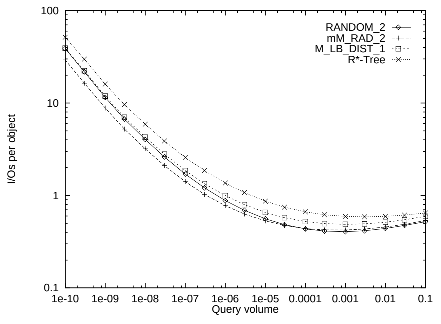

Fig. 21. I/O costs per retrieved object, as a function of the query volume.

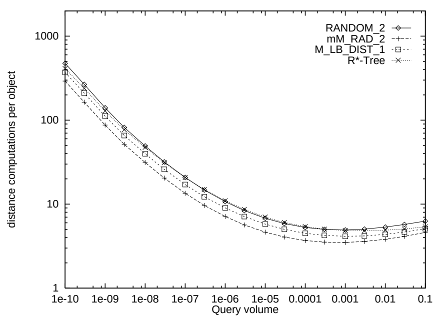

Fig. 22. Distance computations per retrieved object, as a function of the query volume.

set size to 10, 000 20-D points. The number of points in each cluster is, therefore, 10, 000 divided by the number of clusters: when the number of clusters is 1, points are all cluttered in a single sphere, while, when the number of clusters is 10, 000, points are uniformly distributed in the unit hypercube. Again, to present a fair comparison, we used for M-tree the search algorithm without any optimization.

Figures 23 and 24 show the number of I/O operations and the distance selectivity, respectively, for processing range queries with side $\sqrt[20]{0.01}$. It can be seen that M-tree performs considerably better when the distribution of the data set is less uniform, and that only with a high number of (scarcely populated) clusters R*-tree outperforms M-tree as to CPU costs.

Note that the behavior for number of clusters = 1 is essentially due to the fact that the query retrieves almost all the indexed objects, thus visiting all the nodes of the tree.

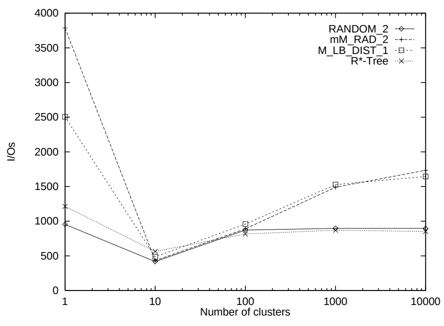

Fig. 23. Number of I/O's for processing range queries, as a function of the data distribution.

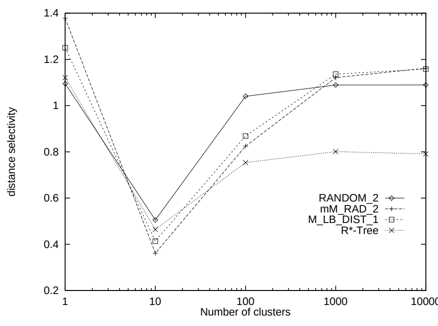

Fig. 24. Distance selectivity for range queries, as a function of the data distribution.

6 Conclusions

The M-tree is a new access method able to index dynamic data sets from generic metric spaces. In this paper we have analyzed several heuristic split policies and have experimentally evaluated them on high-dimensional vector spaces. Results show that a split policy can be chosen so as to determine a good tradeoff between the costs of building and the costs of searching the M-tree, as well as to balance CPU (distance computations) and I/O costs. We can summarize our observations as follows:

- Best search performances are obtained with the `mM_RAD_2` policy, which minimizes the maximum value of the two covering radii resulting from a split.
- A "non-confirmed" split policy performs better than the corresponding "confirmed" version when node capacity is low (i.e. space dimensionality is high).
- M-tree outperforms R*-tree on high-dimensional vector spaces, especially when data distribution is not uniform.
- The optimization used by M-tree search algorithms for reducing the number of distances to be computed is highly effective, saving up to 40% computations.

Our current work is still trying to gain more insights on the behaviors on split policies, in order to come out with a set of "design guidelines" to assist M-tree users in choosing the most appropriate algorithm for the application at hand. For instance, we would like to be able to determine the "right" sample size for the `SAMPLING` policy as a function of the indexed data set. Finally, we are applying M-tree to some difficult real-world problems, such as fingerprint identification and protein matching.

References

[AFS93] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organizations and Algorithms, (FODO'93)*, pages 69–84, Chicago, IL, October 1993. Springer-Verlag, LNCS, Vol. 730.

[BKK96] S. Berchtold, D.A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd VLDB International Conference*, pages 28–39, Mumbai (Bombay), India, September 1996.

[BKSS90] 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, May 1990.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st VLDB International Conference*, pages 574–584, Zurich, Switzerland, September 1995.

[Chi94] T. Chiueh. Content-based image indexing. In *Proceedings of the 20th VLDB International Conference*, pages 582–593, Santiago, Chile, September 1994.

[CPRZ97] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula. Performance of M-tree, an access method for similarity search in metric spaces. Technical Report 13, HERMES ESPRIT LTR Project, 1997. Available at URL `http://www.ced.tuc.gr/hermes/`.

[FEF$^+$94] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, July 1994.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 419–429, Minneapolis, MN, May 1994.

[GR95] V.N. Gudivada and V.V. Raghavan. Content-based image retrieval systems. *IEEE Computer*, 28(9):18–22, September 1995. Guest Editors' Introduction.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[HKR93] D.P. Huttenlocker, G.A. Klanderman, and W.J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

[HNP95] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st VLDB International Conference*, Zurich, Switzerland, September 1995.

[HP94] J.M. Hellerstein and A. Pfeffer. The RD-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison, October 1994.

[JD88] A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, CA, May 1995.

[SRF87] T.K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB International Conference*, pages 507–518, Brighton, England, September 1987.

[Uhl91] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[VM95] M. Vassilakopoulos and Y. Manolopoulos. Dynamic inverted quadtree: A structure for pictorial databases. *Information Systems*, 20(6):483–500, September 1995.

[WBKW96] E. Wold, T. Blum, D. Keislar, and J. Wheaton. Content-based classification, search, and retrieval of audio. *IEEE Multimedia*, 3(3):27–36, 1996.

[ZCR96] P. Zezula, P. Ciaccia, and F. Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. Technical Report 7, HERMES ESPRIT LTR Project, 1996. Available at URL `http://www.ced.tuc.gr/hermes/`.