M-tree: An Efficient Access Method for Similarity Search in Metric Spaces

Paolo Ciaccia
DEIS - CSITE-CNR
Bologna, Italy
pciaccia@deis.unibo.it

Marco Patella
DEIS - CSITE-CNR
Bologna, Italy
mpatella@deis.unibo.it

Pavel Zezula
CNUCE-CNR
Pisa, Italy
zezula@iei.pi.cnr.it

Abstract

A new access method, called M-tree, is proposed to organize and search large data sets from a generic "metric space", i.e. where object proximity is only defined by a distance function satisfying the positivity, symmetry, and triangle inequality postulates. We detail algorithms for insertion of objects and split management, which keep the M-tree always balanced - several heuristic split alternatives are considered and experimentally evaluated. Algorithms for similarity (range and k-nearest neighbors) queries are also described. Results from extensive experimentation with a prototype system are reported, considering as the performance criteria the number of page I/O's and the number of distance computations. The results demonstrate that the M-tree indeed extends the domain of applicability beyond the traditional vector spaces, performs reasonably well in high-dimensional data spaces, and scales well in case of growing files.

1 Introduction

Recently, the need to manage various types of data stored in large computer repositories has drastically increased and resulted in the development of *multimedia database systems* aiming at a uniform management of voice, video, image, text, and numerical data. Among the many research challenges which the multimedia technology entails – including data placement, presentation, synchronization, etc. – content-based retrieval plays a dominant role. In order to satisfy the information needs of users, it is of vital importance to effectively and efficiently support the retrieval process devised to determine which portions of the database are relevant to users' requests.

In particular, there is an urgent need of indexing techniques able to support execution of *similarity queries*. Since multimedia applications typically require complex *distance functions* to quantify similarities of multi-dimensional features, such as shape, texture, color [FEF+94], image patterns [VM95], sound [WBKW96], text, fuzzy values, set values [HNP95], sequence data [AFS93, FRM94], etc., multi-dimensional (*spatial*) access methods (SAMs), such as R-tree [Gut84] and its variants [SRF87, BKSS90], have been considered to index such data. However, the applicability of SAMs is limited by the following assumptions which such structures rely on:

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

2. the (dis)similarity of any two objects has to be based on a distance function which does not introduce any correlation (or "cross-talk") between feature values [FEF+94]. More precisely, an L_p metric, such as the Euclidean distance, has to be used.

Furthermore, from a performance point of view, SAMs assume that comparison of keys (feature values) is a trivial operation with respect to the cost of accessing a disk page, which is not always the case in multimedia

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

applications. Consequently, no attempt in the design of these structures has been done to reduce the number of distance computations.

A more general approach to the "similarity indexing" problem has gained some popularity in recent years, leading to the development of so-called *metric trees* (see [Uhl91]). Metric trees only consider relative distances of objects (rather than their absolute positions in a multi-dimensional space) to organize and partition the search space, and just require that the function used to measure the distance (dissimilarity) between objects is a *metric* (see Section 2), so that the *triangle inequality* property applies and can be used to prune the search space.

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 required to answer a query, paying no attention to I/O costs.

In this article, we introduce a paged metric tree, called M-tree, which has been explicitly designed to be integrated with other access methods in database systems. We demonstrate such possibility by implementing M-tree in the GiST (Generalized Search Tree) [HNP95] framework, which allows specific access methods to be added to an extensible database system.

The M-tree is a balanced tree, able to deal with dynamic data files, and as such it does not require periodical reorganizations. M-tree can index objects using features compared by distance functions which either do not fit into a vector space or do not use an L_p metric, thus considerably extends the cases for which efficient query processing is possible. Since the design of M-tree is inspired by both principles of metric trees and database access methods, performance optimization concerns both CPU (distance computations) and I/O costs.

After providing some preliminary background in Section 2, Section 3 introduces the basic M-tree principles and algorithms. In Section 4, we discuss the available alternatives for implementing the split strategy used to manage node overflows. Section 5 presents experimental results. Section 6 concludes and suggests topics for future research activity.

2 Preliminaries

Indexing a *metric space* means to provide an efficient support for answering *similarity queries*, i.e. queries whose purpose is to retrieve DB objects which are "similar" to a reference (query) object, and where the (dis)similarity between objects is measured by a specific metric distance function d.

Formally, a *metric space* is a pair, $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is a domain of feature values – the indexing *keys* – and d is a total (distance) function with the following properties:[1]

1. $d(O_x, O_y) = d(O_y, O_x)$ (*symmetry*)
2. $d(O_x, O_y) > 0$ $(O_x \neq O_y)$ and $d(O_x, O_x) = 0$ (*non negativity*)
3. $d(O_x, O_y) \leq d(O_x, O_z) + d(O_z, O_y)$ (*triangle inequality*)

In principle, there are two basic types of similarity queries: the *range query* and the *k nearest neighbors query*.

Definition 2.1 (Range)
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 indexed objects O_j such that $d(O_j, Q) \leq r(Q)$. □

Definition 2.2 (k nearest neighbors (k-NN))
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. □

There have already been some attempts to tackle the difficult metric space indexing problem. The FastMap algorithm [FL95] transforms a matrix of pairwise distances into a set of low-dimensional points, which can then be indexed by a SAM. However, FastMap assumes a static data set and introduces approximation errors in the mapping process. The *Vantage Point* (VP) tree [Chi94] partitions a data set according to distances the objects have with respect to a reference (vantage) point. The median value of such distances is used as a separator to partition objects into two balanced subsets, to which the same procedure can recursively be applied. The MVP-tree [BO97] extends this idea by using *multiple vantage points*, and exploits pre-computed distances to reduce the number of distance computations at query time. The GNAT design [Bri95] applies a different – so-called *generalized hyperplane* [Uhl91] – partitioning style. In the basic case, two reference objects are chosen and each of the remaining objects is assigned to the closest reference object. So obtained subsets can recursively be split again, when necessary.

Since all of the above organizations build trees by means of a top-down recursive process, these trees are not guaranteed to remain balanced in case of insertions and deletions, and require costly reorganizations to prevent performance degradation.

[1] 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 research challenge which has led to the design of M-tree was to combine advantages of balanced and dynamic SAMs, with the capabilities of static metric trees to index objects using features and distance functions which do not fit into a vector space and are only constrained by the metric postulates.

The M-tree partitions objects on the basis of their relative distances, as measured by a specific distance function d, and stores these objects into fixed-size nodes,[2] which correspond to constrained regions of the metric space. The M-tree is fully parametric on the distance function d, so the function implementation is a *black-box* for the M-tree. The theoretical and application background of M-tree is thoroughly described in [ZCR96]. In this article, we mainly concentrate on implementation issues in order to establish some basis for performance evaluation and comparison.

3.1 The Structure of M-tree Nodes

Leaf nodes of any M-tree store all indexed (database) objects, represented by their keys or features, whereas internal nodes store the so-called *routing objects*. A routing object is a database object to which a routing role is assigned by a specific *promotion* algorithm (see Section 4).

For each routing object O_r there is an associated pointer, denoted $ptr(T(O_r))$, which references the root of a sub-tree, $T(O_r)$, called the *covering tree* of O_r. All objects in the covering tree of O_r are within the distance $r(O_r)$ from O_r, $r(O_r) > 0$, which is called the *covering radius* of O_r and forms a part of the O_r entry in a particular M-tree node. Finally, a routing object O_r is associated with a distance to $P(O_r)$, its parent object, that is the routing object which references the node where the O_r entry is stored. Obviously, this distance is not defined for entries in the root of the M-tree. The general information for a routing object entry is summarized in the following table.

O_r	(feature value of the) routing object
$ptr(T(O_r))$	pointer to the root of $T(O_r)$
$r(O_r)$	covering radius of O_r
$d(O_r, P(O_r))$	distance of O_r from its parent

An entry for a database object O_j in a leaf is quite similar to that of a routing object, but no covering radius is needed, and the pointer field stores the actual object identifier (oid), which is used to provide access to the whole object possibly resident on a separate data file.[3] In summary, entries in leaf nodes are structured as follows.

O_j	(feature value of the) DB object
$oid(O_j)$	object identifier
$d(O_j, P(O_j))$	distance of O_j from its parent

3.2 Processing Similarity Queries

Before presenting specific algorithms for building the M-tree, we show how the information stored in nodes is used for processing similarity queries. Although performance of search algorithms is largely influenced by the actual construction of the M-tree , the *correctness* and the logic of search are independent of such aspects.

In both the algorithms we present, the objective is to reduce, besides the number of accessed nodes, also the number of distance computations needed to execute queries. This is particularly relevant when the search turns out to be CPU- rather than I/O-bound, which might be the case for computationally intensive distance functions. For this purpose, all the information concerning (pre-computed) distances stored in the M-tree nodes, i.e. $d(O_i, P(O_i))$ and $r(O_i)$, is used to effectively apply the triangle inequality.

3.2.1 Range Queries

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

```
RS(N: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 { ∀ O_r in N do:
           if |d(O_p,Q) − d(O_r,O_p)| ≤ r(Q) + r(O_r)
           then { Compute d(O_r,Q);
                  if d(O_r,Q) ≤ r(Q) + r(O_r)
                  then RS(*ptr(T(O_r)),Q,r(Q)); }}
  else { ∀ O_j in N do:
           if |d(O_p,Q) − d(O_j,O_p)| ≤ r(Q)
           then { Compute d(O_j,Q);
                  if d(O_j,Q) ≤ 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 computing any new distance at all. The condition applied for pruning is as follows.

Lemma 3.1 *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.*

[2] 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.

[3] Of course, the M-tree can also be used as a primary data organization, where the whole objects are stored in the leaves of the tree.

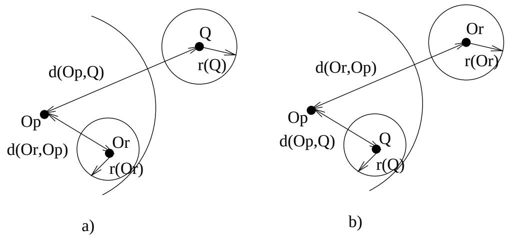

Figure 1: Lemma 3.2 applied to avoid distance computations

In fact, since $d(O_j, Q) \geq d(O_r, Q) - d(O_j, O_r)$ (triangle inequality) and $d(O_j, O_r) \leq r(O_r)$ (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 3.1, the $d(O_r, Q)$ distance has to be computed. This can be avoided by taking advantage of the following result.

Lemma 3.2 *If* $\mid d(O_p, Q) - d(O_r, O_p) \mid > r(Q) + r(O_r)$, *then* $d(O_r, Q) > r(Q) + r(O_r)$.

This is a direct consequence of the triangle inequality, which guarantees that both $d(O_r, Q) \geq d(O_p, Q) - d(O_r, O_p)$ (Figure 1 a) and $d(O_r, Q) \geq d(O_r, O_p) - d(O_p, Q)$ (Figure 1 b) hold. 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 necessary to compute is when dealing with the root node, for which O_p is undefined.

3.2.2 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, contains 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. According to our experimental observations, other criteria do not lead to better performance.

```
ChooseNode(PR:priority_queue):  node
{ let d_min(T(O_r*)) = min{d_min(T(O_r))},
      considering all the entries in PR;
  Remove 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:root_node,Q:query_object,k:integer)
{ PR = [T,_];
  for i = 1 to k do:  NN[i] = [_,∞];
  while PR ≠ ∅ 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 inserts 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 by means of the pre-computed distances from the parent object, is also applied.

```
k-NN_NodeSearch(N:node,Q:query_object,k:integer)
{ let O_p be the parent object of node N;
  if N is not a leaf then
  { ∀ O_r in N do:
    if |d(O_p,Q) - d(O_r,O_p)| ≤ d_k + r(O_r) then
    { Compute d(O_r,Q);
      if d_min(T(O_r)) ≤ 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 /* N is a leaf */
  { ∀ O_j in N do:
    if |d(O_p,Q) - d(O_j,O_p)| ≤ d_k then
    {Compute d(O_j,Q);
     if d(O_j,Q) ≤ 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 Building the M-tree

Algorithms for building the M-tree specify how objects are inserted and deleted, and how node overflows and underflows are managed. Due to space limitations, deletion of objects is not described in this article.

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 used 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. $d(O_r, O_n) \leq r(O_r)$. If multiple sub-trees with this property exist, the one for which object O_n is closest to O_r is chosen. This heuristics 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, the choice is to minimize the *increase* of the covering radius, $d(O_r, O_n) - r(O_r)$. This is tightly related to the heuristic criterion that suggests to minimize the overall "volume" covered by routing objects in the current node.

```
Insert(N:node, entry(O_n):M-tree_entry)
{ let 𝒩 be the set of entries in node N;
  if N is not a leaf then
  { let 𝒩_in = entries such that d(O_r,O_n) ≤ r(O_r);
    if 𝒩_in ≠ ∅
    then let entry(O_r*) ∈ 𝒩_in:d(O_r*,O_n) is minimum;
    else { let entry(O_r*) ∈ 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 /* N is a leaf */
  { if N is not full
    then store entry(O_n) in N
    else Split(N,entry(O_n)); }}
```

The determination of the set $\mathcal{N}_{in}$ – routing objects for which no enlargement of the covering radius is needed – can be optimized by saving distance computations. From Lemma 3.2, by substituting O_n for Q and setting $r(O_n) \equiv r(Q) = 0$, we derive that:

If $|d(O_p, O_n) - d(O_r, O_p)| > r(O_r)$ then $d(O_r, O_n) > (O_r)$

from which it follows that $O_r \notin \mathcal{N}_{in}$. Note that this optimization cannot be applied in the root node.

3.4 Split Management

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 entries among these two nodes, and posting (*promoting*) to the parent node, N_p, two routing objects to reference the two nodes. When the root splits, a new root is created and the M-tree grows by one level up.

```
Split(N:node; E:M-tree_entry)
{ let 𝒩 = entries of node N ∪ {E};
  ifN is not the root then
    let O_p be the parent of N, stored in N_p node;
  Allocate a new node N';
  Promote(𝒩,O_p1,O_p2);
  Partition(𝒩,O_p1,O_p2,𝒩1,𝒩2);
  Store 𝒩1's entries in N and 𝒩2's entries in N';
  if N is the current root
  then { Allocate a new root node, N_p;
         Store entry(O_p1) and entry(O_p2) in N_p; }
  else { Replace entry(O_p) with entry(O_p1) in N_p;
         if node N_p is full
         then Split(N_p, entry(O_p2))
         else store entry(O_p2) 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 divides 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 the `Promote` and `Partition` methods 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 a possibility of implementing alternative split policies, in order to tune performance depending on specific application needs (see Section 4).

Regardless of the specific split policy, the semantics of covering radii has to be preserved. If the split node is a leaf, then the covering radius of a promoted object, say O_{p_1}, is set to

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

whereas if overflow occurs in an internal node

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

which guarantees that $d(O_j, O_{p_1}) \leq r(O_{p_1})$ holds for any object in $T(O_{p_1})$.

4 Split Policies

The "ideal" split policy should promote O_{p_1} and O_{p_2} objects, and partition other objects so 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 covering radii, the distance between chosen reference objects should be maximized.[4]

Besides above requirements, which are quite "standard" also for SAMs [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. The specific algorithms we consider can first be classified according to whether or not they "confirm" the original parent object in its role.

Definition 4.1 *A* confirmed *split policy chooses as one of the promoted objects, say O_{p_1}, the object O_p, i.e. the parent object 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.

`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.

`mM_RAD` This is similar to `m_RAD`, but it minimizes the maximum of the two radii.

`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

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

`RANDOM` This variant selects in a random way the reference object(s). Although it is not 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 resulting maximum of the two covering radii 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.

4.2 Distribution of the Entries

Given a set of entries $\mathcal{N}$ and 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: 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 the way how routing objects are chosen, an unbalanced split policy

[4] 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.

can lead to a better objects' partitioning, due to the additional degree of freedom it obtains. It has to be remarked that, while obtaining a balanced split with SAMs 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".

5 Experimental Results

In this section we provide experimental results on the performance of M-tree in processing similarity queries. Our implementation is based on the GiST C++ package [HNP95], and uses a constant node size of 4 KBytes. Although this can influence results, in that node capacity is inversely related to the dimensionality of the data sets, we did not investigate the effect of changing the node size.

We tested all the split policies described in Section 4, and evaluated them under a variety of experimental settings. To gain the flexibility needed for comparative analysis, most experiments were based on synthetic data sets, and here we only report about them. Data sets were obtained by using the procedure described in [JD88] which generates normally-distributed clusters in a *Dim*-D vector space. In all the experiments the number of clusters is 10, the variance is $\sigma^2 = 0.1$, and clusters' centers are uniformly distributed (Figure 2 shows a 2-D sample). Distance is evaluated using the L_∞ metric, i.e. $L_\infty(O_x, O_y) = \max_{j=1}^{Dim}\{| O_x[j] - O_y[j] |\}$, which leads to hyper-cubic search (and covering) regions. Graphs concerning construction costs are obtained by averaging the costs of building 10 M-trees, and results about performance on query processing are averaged over 100 queries.

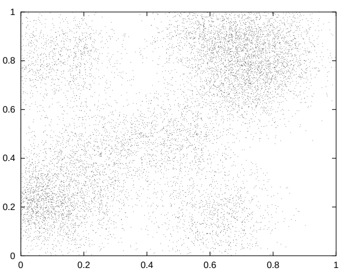

Figure 2: A sample data set used in the experiments

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 also obtained 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 4.1).

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`.

The Effect of Dimensionality

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

Figure 3 shows that all the split policies but `m_RAD_2` and `mM_RAD_2` compute almost the same number of distances for building the tree, and that this number decreases when *Dim* grows. The explanation is that increasing *Dim* reduces the node capacity, which has a beneficial effect on the numbers of distances computed by insertion and split algorithms. The reduction is particularly evident for `m_RAD_2` and `mM_RAD_2`, whose CPU split costs grow as the square of node capacity. I/O costs, shown in Figure 4, have an inverse trend and grow with space dimensionality. This can again be explained by the reduction of node capacity. The fastest split policy is `RANDOM_2` and the slowest one is, not surprisingly, `m_RAD_2`.

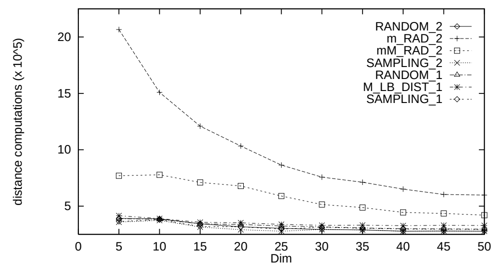

Figure 3: Distance comp. for building M-tree

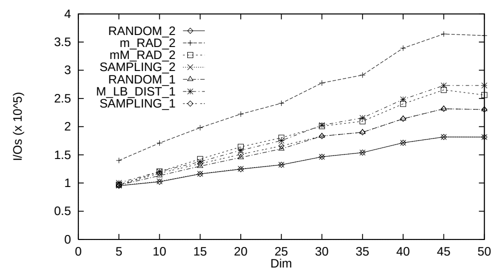

Figure 4: I/O costs for building M-tree

		`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: CPU, I/O, and volume overheads

Figure 5 shows that the "quality" of tree construction, measured by the average covered volume per page, depends on split policy complexity, and that the criterion of the cheap `M_LB_DIST_1` policy is indeed effective enough.

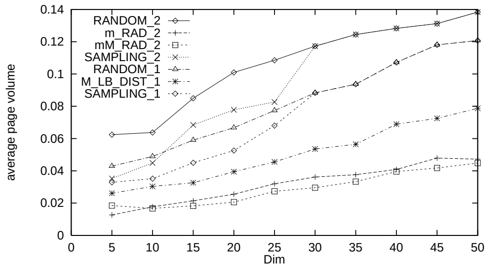

Figure 5: Average covered volume per page

Performance on 10-NN query processing, considering both I/O's and distance selectivities, is shown in Figures 6 and 7, respectively – distance selectivity is the ratio of computed distances to the total number of objects.

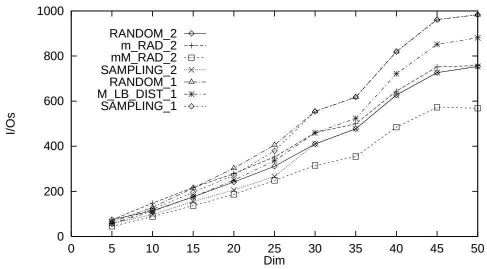

Figure 6: I/O's for processing 10-NN queries

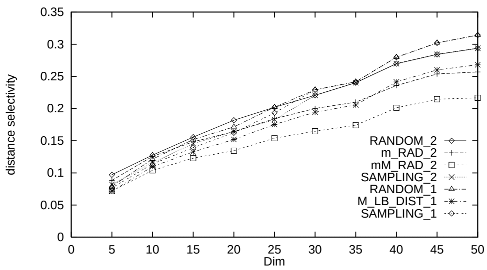

Figure 7: Distance selectivity for 10-NN queries

Some interesting observations can be done about these results. First, policies based on "non-confirmed" promotion, perform better that "confirmed" policies as to I/O costs, especially on high dimensions where they save up to 25% I/O's. This can be attributed to the better object clustering that such policies can obtain. I/O costs increase with the dimensionality mainly because of the reduced page capacity, which leads to larger trees. For the considered range of Dim values, node capacity varies by a factor of 10, which almost coincides with the ratio of I/O costs at $Dim = 50$ to the costs at $Dim = 5$.

As to distance selectivity, differences emerge only with high values of Dim, and favor `mM_RAD_2` and `m_RAD_2`, which exhibit only a moderate performance degradation. Since these two policies have the same complexity, and because of above results, `m_RAD_2` is discarded in subsequent analyses.

Scalability

Another major challenge in the design of M-tree was to ensure scalability of performance 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.

Table 2 shows the average number of distance computations and I/O operations per inserted object, for 2-D data sets whose size varies in the range $10^4 \div 10^5$. Results refer to the `RANDOM_2` policy, but similar trends were also observed for the other policies. The moderate increase of the average number of distance computations depends both on the growing height of the tree and on the higher density of indexed objects within clusters. This is because the number of clusters was kept fixed at 10, regardless of the data set size.

Figures 8 and 9 show that both I/O and CPU (distance computations) 10-NN search costs grow logarithmically with the number of objects, which demonstrates that M-tree scales well in the data set size, and that the dynamic management algorithms do not deteriorate the quality of the search. It has to be emphasized that such a behavior is peculiar to the M-tree, since other known metric trees are intrinsically static.

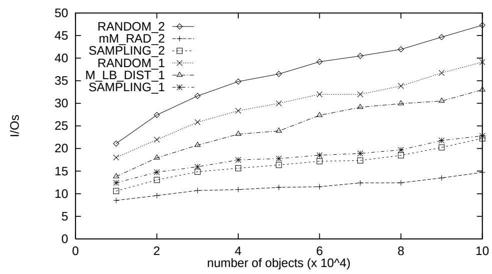

Figure 8: I/O's for processing 10-NN queries

As to the relative behaviors of split policies, figures show that "cheap" policies (e.g. `RANDOM` and

n. of objects ($\times 10^4$)	1	2	3	4	5	6	7	8	9	10
avg. n. dist. comp.	45.0	49.6	53.6	57.5	61.4	65.0	68.7	72.2	73.6	74.7
avg. n. I/O's	8.9	9.3	9.4	9.5	9.6	9.6	9.6	9.6	9.7	9.8

Table 2: Average number of distance computations and I/O's for building the M-tree (`RANDOM_2` split policy)

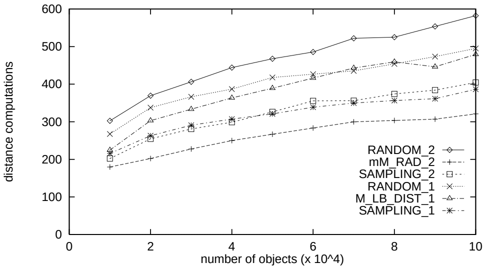

Figure 9: Distance computations for 10-NN queries

`M_LB_DIST_1`) are penalized by the high node capacity ($M = 60$) 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.

5.1 Comparing M-tree and R*-tree

The final set of experiments we present compares M-tree with R*-tree. The R*-tree implementation we use is the one available with the GiST package. We definitely do not deeply investigate merits and drawbacks of the two structures, rather we provide some reference results obtained from an access method which is well-known and largely used in database systems. Furthermore, although M-tree has an intrinsically wider applicability range, we consider important to evaluate its relative performance on "traditional" domains where other access methods could be used as well.

Results in Figures 10 and 11 compare I/O and CPU costs, respectively, to build R*-tree and M-tree, the latter only for the `RANDOM_2`, `M_LB_DIST_1`, and `mM_RAD_2` policies. The trend of the graphs for R*-tree confirms what already observed about the influence of the node capacity (see Figures 3 and 4). Graphs emphasize the different perfomance of M-trees and R*-trees in terms of CPU costs, whereas both structures have similar I/O building costs.

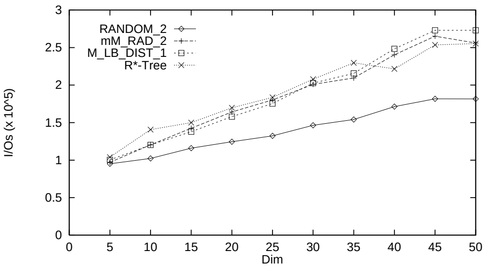

Figure 10: I/O costs for building M-tree and R*-tree

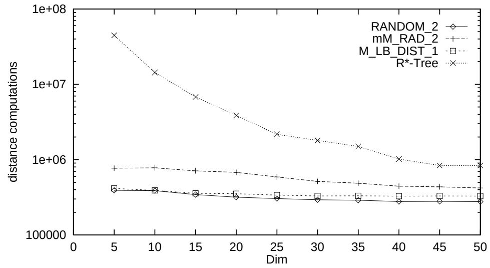

Figure 11: Distance computations for building M-tree and R*-tree

Figures 12 and 13 show the search costs for square 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 13 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 3.2) is not applied. Graphs show that this optimization is highly effective, saving up to 40% distance computations (similar results were obtained for NN-queries). Note that, even without such an optimization, M-tree is almost always more efficient than R*-tree.

From these results, which we remind are far from providing a detailed comparison of M-trees and R*-trees, we can anyway see that M-tree is a competitive access method even for indexing data from vector spaces.

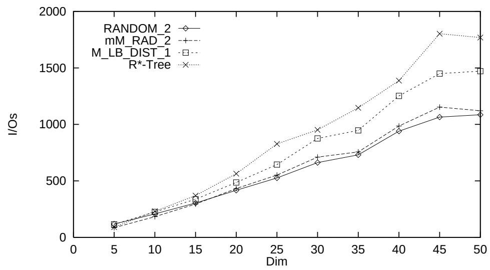

Figure 12: I/O's for processing range queries

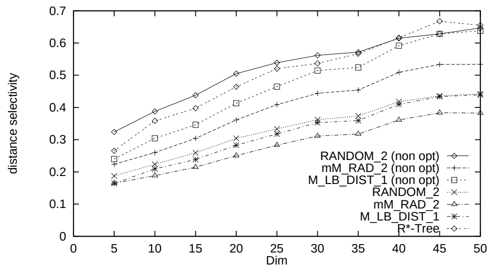

Figure 13: Distance selectivity for range queries

6 Conclusions

The M-tree is an original index/storage structure with the following major innovative properties:

- it is a paged, balanced, and dynamic secondary memory structure able to index data sets from generic metric spaces;
- similarity range and nearest neighbor queries can be performed and results ranked with respect to a given query object;
- query execution is optimized to reduce both the number of page reads and the number of distance computations;
- it is also suitable for high-dimensional vector data.

Experimental results show that M-tree achieves its primary goal, that is, dynamicity and, consequently, scalability with the size of data sets from generic metric spaces. Analysis of many available split policies suggests that the proper choice should reflect the relative weights that CPU (distance computation) and I/O costs may have. This possibility of "tuning" M-tree performance with respect to these two cost factors, which are highly dependent on the specific application domain, has never been considered before in the analysis of other metric trees. Our implementation, based on the GiST package, makes clear that M-tree can effectively extend the set of access methods of a database system.[5]

Current and planned research work includes: support for more complex similarity queries, use of variable size nodes to perform only "good" splits [BKK96], and parallelization of CPU and I/O loads. Real-life applications, such as fingerprint identification and protein matching, are also considered.

Acknowledgements

This work has been funded by the EC ESPRIT LTR Proj. 9141 HERMES. P. Zezula has also been supported by Grants GACR 102/96/0986 and KONTAKT PM96 S028.

References

[AFS93] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. *FODO'93*, pp. 69–84, Chicago, IL, Oct. 1993. Springer LNCS, Vol. 730.

[BKK96] S. Berchtold, D.A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. *22nd VLDB*, pp. 28–39, Mumbai (Bombay), India, Sept. 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. *ACM SIGMOD*, pp. 322–331, Atlantic City, NJ, May 1990.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. *ACM SIGMOD*, pp. 357–368, Tucson, AZ, May 1997.

[Bri95] S. Brin. Near neighbor search in large metric spaces. *21st VLDB*, pp. 574–584, Zurich, Switzerland, Sept. 1995.

[Chi94] T. Chiueh. Content-based image indexing. *20th VLDB*, pp. 582–593, Santiago, Chile, Sept. 1994.

[FEF+94] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *J. of Intell. Inf. Sys.*, 3(3/4):231–262, July 1994.

[FL95] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *ACM SIGMOD*, pp. 163–174, San Jose, CA, June 1995.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *ACM SIGMOD*, pp. 419–429, Minneapolis, MN, May 1994.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, pp. 47–57, Boston, MA, June 1984.

[HNP95] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized search trees for database systems. *21st VLDB*, Zurich, Switzerland, Sept. 1995.

[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. *ACM SIGMOD*, pp. 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. *13th VLDB*, pp. 507–518, Brighton, England, Sept. 1987.

[Uhl91] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Lett.*, 40(4):175–179, Nov. 1991.

[VM95] M. Vassilakopoulos and Y. Manolopoulos. Dynamic inverted quadtree: A structure for pictorial databases. *Inf. Sys.*, 20(6):483–500, Sept. 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. TR 7, HERMES ESPRIT LTR Project, 1996. Available at URL `http://www.ced.tuc.gr/hermes/`.

[5] We are now implementing M-tree in the PostgreSQL system.