# Bulk Loading the M-tree

**Article** · February 1970

Source: CiteSeer

CITATIONS
73

READS
873

**2 authors:**

Paolo Ciaccia
University of Bologna
**145** PUBLICATIONS **4,004** CITATIONS
SEE PROFILE

Marco Patella
University of Bologna
**72** PUBLICATIONS **3,603** CITATIONS
SEE PROFILE

# Bulk Loading the M-tree *

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

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

**Abstract.** The M-tree is a dynamic paged structure that can be effectively used to index multimedia databases, where objects are represented by means of complex features and similarity queries require the computation of time-consuming distance functions. The initial loading of the M-tree, however, can be very expensive. In this paper we propose a fast (bulk) loading algorithm to speed-up the creation of the tree on a given dataset. Experimental results show that our `BulkLoading` algorithm can significantly improve the index' performance with respect to M-tree insertion methods, and its performance is comparable to that of *static* metric trees.

## 1 Introduction

*Content-based retrieval* of objects is one of the most common operations required by the incoming multimedia (MM) era. Multimedia users often request images, sounds, texts, and videos from large repositories for medical, scientific, legal, and art applications, to name a few. To be efficiently retrieved, such objects are characterized and indexed using relevant *features* (shapes, textures, patterns, and colors for images, loudness and harmonicity for sounds, shots and objects' trajectories for videos, etc.), then *similarity queries* can be issued to retrieve the objects which are most similar to a user-provided query object.

A classical approach to support similarity queries is to use *spatial* access methods (SAMs). The applicability of SAMs, such as R-tree [Guttman, 1984] and its variants, is, however, limited by the fact that, for indexing purposes, objects are to be represented by means of feature values in a *vector space* of dimensionality $Dim$. Furthermore, SAMs have been designed by assuming that comparison of feature values has a negligible CPU cost with respect to the cost of disk I/O, which is not always the case in MM applications.

A more general approach consists in supporting similarity queries over generic *metric spaces*, which include vector spaces. 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 metric – a non-negative and symmetric function which also satisfies the *triangular inequality* ($d(O_x, O_y) \leq d(O_x, O_z) + d(O_z, O_y)\ \forall O_x, O_y, O_z \in \mathcal{D}$ ).[1]

* This work has been partially supported by projects ESPRIT LTR no. 9141, HERMES, and Italian C.N.R. MIDA.

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

The two basic types of similarity queries are the following:

**Range query:** *Given an object $Q \in \mathcal{D}$ and a* maximum search distance $r_Q$*, the query* $\texttt{range}(Q, r_Q)$ *selects all the objects $O_j$ such that $d(O_j, Q) \leq r_Q$.* □

**k nearest neighbors (k-NN) query:** *Given an object $Q \in \mathcal{D}$ and an integer $k \geq 1$, the query* $\texttt{NN}(Q, k)$ *selects the k objects having the shortest distance from $Q$, according to the distance function $d$.* □

The difficult problem of indexing metric spaces has led to the development of so-called *metric trees*. 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. Several metric trees have been developed so far, including the vp-tree [Chiueh, 1994], the GNAT [Brin, 1995], and the mvp-tree [Bozkaya and Özsoyoglu, 1997]. Since all above designs build the index in a top-down way, the tree is not guaranteed to stay balanced in case of random insertions and deletions, thus requiring costly reorganizations to prevent performance degradation. Contrary to SAMs, these metric trees have only insisted on reducing 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 [Ciaccia et al., 1997], thus performance optimization concerns both CPU (distance computations) and I/O costs. The incremental construction of an M-tree could lead, depending on the order of insertion of the objects, to very different performances during the querying phase. In this paper we present an algorithm to optimize the building of an M-tree given a set of data objects. This "loading" of the tree can be done to speed-up the index creation on a static database, or to reorganize the tree on a dynamic database, when allowed by time constraints.

The paper is organized as follows. In Section 2 we review some techniques which are relevant to our work. Section 3 gives an overview of M-tree principles and algorithms. In Sections 4 and 5 we present the basic version of our `BulkLoading` algorithm and discuss on some optimization techniques. Section 6 presents experimental results and Section 7 concludes the paper.

## 2 Related Work

Typical spatial (multidimensional) index structures support random insertion, deletions and updates, but, recently, there has been an increasing interest in *bulk operations* (i.e. a series of operations executed atomically, without interruption by other actions), in particular the index' creation from a given set of records (bulk loading). A great variety of bulk loading techniques has been proposed for R-trees. Most of them [Roussopoulos and Leifker, 1985; Kamel and Faloutsos, 1993; Leutenegger et al., 1997] sort the dataset using one-dimensional criteria, usually based on the clustering properties of space-filling (e.g. Hilbert) curves. Only a few methods [Li and Laurini, 1991; Gavrila, 1994] exploit the "metric" properties of the dataset (i.e. relative distances between objects), and try to achieve a good clustering of the objects by iteratively improving on the choice of clusters' "centers".

A generic algorithm to bulk loading multidimensional index structures (but the method could be easily extended to metric indices) has been recently proposed [van den Bercken et al., 1997]. The index is built bottom-up by using *buffer-trees*. The major drawbacks of this technique are that its performance still depends on the order in which data are inserted and its goal is to reduce only I/O costs, thus ignoring CPU costs.

Another choice for indexing objects in a metric space could be to use a mapping technique like *FastMap* [Faloutsos and Lin, 1995]. This technique maps objects into points of a $Dim$-dimensional space (where $Dim$ is user-defined), such that the distances between objects are preserved as much as possible. The major benefit of this approach is that the $Dim$-dimensional points can be indexed using a SAM, like the R-tree. However, the method suffers from using approximated distances instead of the real ones and from being intrinsically static.

Finally, methods like BIRCH [Zhang et al., 1996] and other data clustering techniques [Jain and Dubes, 1988] address problems very similar to ours, but they cannot be directly applied in our context for several reasons: (a) they are essentially vector-based, (b) they try to reduce only I/O and not CPU costs, and (c) the cluster analysis is performed off-line with expensive, multiple-pass algorithms.

## 3 The M-tree

The M-tree stores the objects into fixed-size nodes, 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. Each entry in a leaf node has the format

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

where $\texttt{oid}(O_j)$ is the identifier of the object which resides on a separate data file, $O_j$ are the feature values of the object, 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 node stores a feature value, $O_r$, also called a *routing object*, and a *covering radius*, $r(O_r) > 0$. The entry of $O_r$ includes a pointer, $\texttt{ptr}(T(O_r))$, to the root of subtree $T(O_r)$ – the *covering tree* of $O_r$ – and $d(O_r, P(O_r))$, the distance from the parent object:[2]

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

The basic property of the covering radius is that each object $O_j$ in the covering tree of $O_r$ satisfies the inequality $d(O_j, O_r) \leq r(O_r)$. 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. $O_r \equiv P(O_j)$. 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. The covering radius, $r(O_j)$, and the distance between the object and its parent object, $d(O_j, P(O_j))$, both stored in each entry of the tree, are used

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

by the search algorithms to "prune" the search space (for more specific details, see [Ciaccia et al., 1997]).

### 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 *promoting* relevant information about the two nodes to the parent node, $N_p$. When the root splits, a new root is created and the M-tree grows by one level. Figure 1 shows this in the case $\mathcal{M} = (\Re^2, L_2)$, i.e. the real plane with the Euclidean distance.

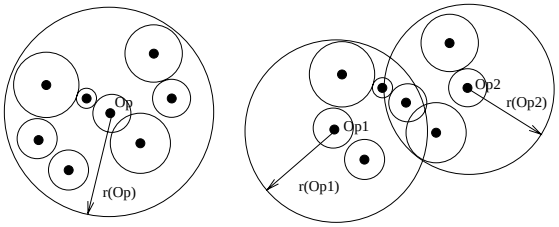


**Fig. 1.** Split of an internal node in $(\Re^2, L_2)$.

In [Ciaccia et al., 1997], we have proposed a suite of heuristic policies for splitting a node. Here, we recall only a few of them:

`mM_RAD` The "minimum Maximum of two 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 maximum of the covering radii, $r(O_{p_1})$ and $r(O_{p_2})$, is minimum.

`RANDOM` The reference object(s) are selected randomly.

`M_LB_DIST` This policy only uses the pre-computed stored distances, sets $O_{p_1} \equiv O_p$, and determines $O_{p_2}$ as the farthest object from $O_p$, that is $d(O_{p_2}, O_p) = \max_j\{d(O_j, O_p)\}$.

## 4 The `BulkLoading` Algorithm (base version)

The bulk loading algorithm we propose performs a clustering of $n$ *data objects* $\mathcal{S} = \{O_1, \ldots, O_n\}$, with constraints on minimum, $u_{min}$, and maximum, $u_{max}$, node utilization, and returns an M-tree $\mathcal{T}$.

Given the set of objects $\mathcal{S}$, we randomly sample $k$ objects $O_{f_1}, \ldots, O_{f_k}$ from $\mathcal{S}$, insert them in the *sample set* $\mathcal{F}$, and then assign each object in $\mathcal{S}$ to its nearest sample, thus producing $k$ sets $\mathcal{F}_1, \ldots, \mathcal{F}_k$. Then, we invoke the bulk loading algorithm on each of these $k$ sets, obtaining $k$ sub-trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$.[3] The effect of these recursive calls is that, in the leaves of each sub-tree, we obtain a partition of the dataset up to the desired level of granularity. Then, we have to

[3] This is similar to the generation of *seeded trees* presented in [Lo and Ravishankar, 1995].

invoke the bulk loading algorithm one more time on the set $\mathcal{F}$, obtaining a super-tree $\mathcal{T}_{sup}$. Finally, we append each sub-tree $\mathcal{T}_i$ to the leaf of $\mathcal{T}_{sup}$ corresponding to the sample object $O_{f_i}$, and obtain the final tree $\mathcal{T}$.

Of course, the partitioning of the dataset highly depends on the choice of the sample objects: taking samples in a sparse region would produce very low sub-trees, since the corresponding sets will have a very low cardinality, while a sample in a dense region would produce a higher sub-tree.

Figure 2 shows a 2-D example and the resulting tree with node capacity $M = 3$ and $u_{max} = 1$: at the first step, the algorithm selects the objects A, B, and C as samples for the sub-trees. Other named objects are samples for other sub-trees (note that, in order to simplify the drawing, in the example we assumed that a sample at a higher level is also a sample for lower levels, e.g. C=C'=C").

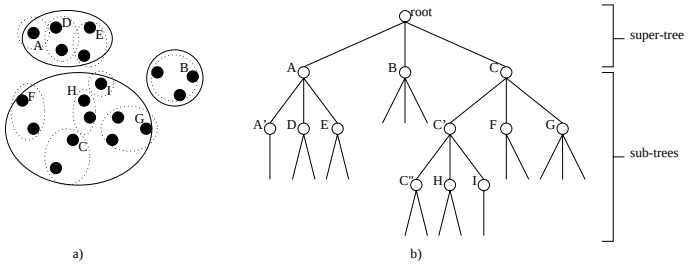


**Fig. 2.** A 2-D example (a) and the resulting tree (b).

If the sub-trees have different heights, or underfilled nodes, when a set $\mathcal{F}_i$ has less than $m \stackrel{\text{def}}{=} u_{min} \cdot M$ objects, the above algorithm can produce a non-balanced tree (both cases are included in the example of Figure 2). To resolve these problems we use two different techniques: (a) reassign the objects in underfilled sets to other sets, and delete the corresponding sample object from $\mathcal{F}$; (b) split the "taller" sub-trees, obtaining a number of "shorter" sub-trees; the roots of the obtained sub-trees will be inserted in the sample set $\mathcal{F}$, replacing the original sample object. Note that the first heuristics leads to a lower number of samples and ensures that each node of the sub-trees has a guaranteed minimum utilization, while the latter technique increases the number of samples by splitting the taller sub-trees. If a single set of objects results at the end of this phase, we repeat the overall process, starting from a new sampling stage.

In the example of Figure 2, supposing $u_{min} = 1/2$, nodes A' and I are deleted and their object reassigned to nodes D and H, respectively. Then, since the sub-tree with minimum height is that having root in B, the sub-trees rooted in A and C are splitted, generating the sub-trees rooted in D, E, C", H, F, and G. Finally, the algorithm builds the super-tree on all these sub-trees (see Figure 3, supposing that the samples for the super-tree would be objects D, B, and C).

The algorithm proposed so far cannot guarantee the minimum utilization $u_{min}$ for the root node of each sub- or super-tree. To obviate this, we check if the root of the sub-tree just created is underfilled and, in this case, split the sub-tree at the root level. The complete algorithm is given below.

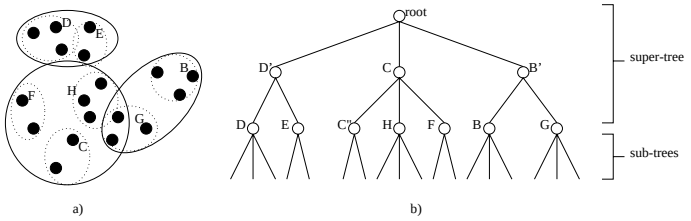


**Fig. 3.** The example of Figure 2 after the redistribution phase.

```
BulkLoading(S: objects_set, M: integer, m: integer)
{ if ||S|| ≤ M, create a new M-tree T, insert all O_i in T and return T;
  else
  { sample k objects O_{f_1}, ..., O_{f_k} from S and insert them in F; // build the sampling set
    for each O_i ∈ S, insert O_i in F_j, where d(O_i, O_{f_j}) ≤ d(O_i, O_{f_p}), ∀O_{f_p} ∈ F;
      // assign each objects to its nearest sample
    for each F_j, if ||F_j|| < m // redistribution phase
    { delete O_{f_j} from F;
      for each O_i ∈ F_j, insert O_i in F_l, where d(O_i, O_{f_l}) ≤ d(O_i, O_{f_p}), ∀O_{f_p} ∈ F; }
    if ||F|| = 1, restart from the sampling phase;
    for each F_j // build the sub-trees
    { let T_j =BulkLoading(F_j, M, m);
      if root(T_j) is underfilled, split T_j into p sub-trees T_j, ..., T_{j+p-1}; }
    let h_min be the minimum height of the sub-trees T_j;
    let T' = ∅ be the sub-trees set;
    for each T_j, if height(T_j)> h_min // split the higher trees
    { delete O_{f_j} from F;
      split T_j into p sub-trees T'_1, ..., T'_p of height h_min;
      insert T'_1, ..., T'_p in T'; // build the set of sub-trees
      insert the root objects of T'_1, ..., T'_p, O'_{f_1}, ..., O'_{f_p}, in F; }
    else insert T_j in T';
    let T_sup =BulkLoading(F, M, m); // build the super-tree
    append each T_j ∈ T' to the leaf of T_sup corresponding to O_{f_j} ∈ F,
      obtaining a new M-tree T;
      // append each sub-tree to the corresponding leaf of the super-tree
    update the radius of the upper regions of T;
    return T; } }
```

## 5 Optimization Techniques

The proposed algorithm, as presented above, has proven itself to be very effective with respect to search costs (as we will see in Section 6), but its major drawback is the still high number of computed distances during the building phase, as compared to other M-tree insertion strategies. This led us to investigate some optimization techniques to reduce the number of distance computations during the construction phase.

### 5.1 Saving some Distance Computations

At each call after the initial one, we have to build a tree $\mathcal{T}_r$ rooted in $O_r$ on a subset $\mathcal{S}_r$ of $\mathcal{S}$. By construction, observe that, for each $O_j \in \mathcal{S}_r$, we have already computed, in the previous step, the distance $d(O_r, O_j)$.

The sampling phase applied to $\mathcal{S}_r$ yields a set of $k$ sample objects $O_{f_1}, \ldots, O_{f_k}$, and we have to find, for each $O_j \in \mathcal{S}_r$, its nearest sample object. Suppose $O_f^*$ is

the nearest sample for object $O_j$ obtained so far, and the distance between $O_j$ and a sample $O_{f_p}$ has to be computed: since the value $\mid d(O_j, O_r) - d(O_r, O_{f_p}) \mid$ is a lower bound on $d(O_j, O_{f_p})$, if $d(O_j, O_f^*) \leq \mid d(O_j, O_r) - d(O_r, O_{f_p}) \mid$, we can safely avoid to compute $d(O_j, O_{f_p})$, because this would surely be greater than $d(O_j, O_f^*)$. The root object, thus, acts as a *vantage point* for the computation of the distance matrix [Shapiro, 1977].

This technique has two major limitations: (a) its application is possible only during the construction of sub-trees, whereas a lot of distance computations are performed after the initial sampling phase (where no distance has been computed yet) and during the construction of the super-tree, and (b) the use of the root object as a vantage point is not very effective, since it likely lies in the "center" of the cluster constituted by the set $\mathcal{S}_r$, while it is suggested [Shapiro, 1977] that vantage points should be multiple and far from the center of the cluster.

### 5.2 Saving more Distance Computations

In the base version of `BulkLoading`, we do not compute the relative distances between sample objects, since it is clear that the nearest sample of such objects will be the sample itself. If we compute such distances, the sample objects can play the role of *multiple vantage points*, like the root object in Section 5.1. In this case, however, the sample objects are not crowded near the center of the cluster, thus leading to a more efficient distance pruning. Experimental results will show if the overhead introduced by the computation of the relative distances between the sample objects can result in a lower number of computed distances. The global algorithm for the distance matrix computation is given below.

```
DistanceMatrix(S: objects_set, F: sample_set, O_r: root_object, D: distance_matrix)
{ for each O_i ∈ S
  { if O_i = O_{f_k} ∈ F, then let D_{j,i} = ∞, ∀j ≠ k; let D_{k,i} = 0
    else
    { D_{1,i} = d(O_{f_1}, O_i) ; // compute the first distance
      j* = 1;
      for each O_{f_j} ∈ F, j > 1;
      { if (| d(O_r, O_{f_j}) − d(O_r, O_i) |< d(O_{f_{j*}}, O_i) and ∀k < j : D_{k,i} < ∞,
           | D_{k,i} − d(O_{f_j}, O_{f_k}) |< d(O_{f_{j*}}, O_i)) then D_{j,i} = d(O_{f_j}, O_i);
        else D_{j,i} = ∞; } // we can avoid this distance computation
      if D_{j,i} < D_{j*,i} then j* = j; } } } // update the nearest sample
```

Figures 4 and 5 show the results for both optimization techniques in terms of percentage of distance computations saving and of total computed distances, respectively, as a function of the space dimensionality (`BulkLoading0` refers to the original algorithm, `BulkLoading1` to the first optimization technique, while `BulkLoading2` uses both optimization techniques). Here and in the experiments of Section 6, the synthetic datasets we use consist of normally-distributed clusters in a $Dim$-D vector space. Unless otherwise stated, the number of indexed objects is $10^4$. The number of clusters is 10, the variance is $\sigma^2 = 0.1$, and clusters' centers are uniformly distributed in the unit hypercube. The used metric is $L_\infty(O_x, O_y) = \max_{j=1}^{Dim}\{\mid O_x[j] - O_y[j] \mid\}$, and the node size is 4 Kbytes.

The second optimization achieves far better results than the first one, reaching a 70% saving for $Dim = 5$. The decreasing trend of distance computations

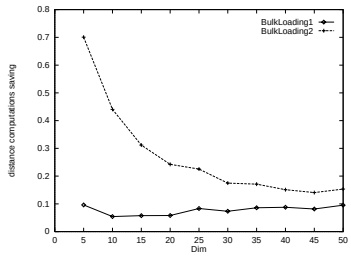


**Fig. 4.** Percentage of distance computations saving during construction phase.

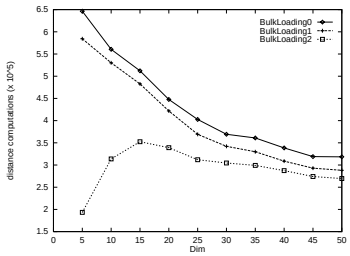


**Fig. 5.** Computed distances during construction phase.

saving is essentially due to two factors: (a) With increasing $Dim$, the number of samples, $k$, decreases, so that the percentage of distances to be computed increases accordingly. (b) Using the $L_\infty$ metric, distances between objects tend to a constant value, for increasing space dimensionality, which reduces the positive effect of having multiple vantage points. The most surprising result displayed by Figure 5 is the initial increasing trend for low dimensionalities for the final optimized algorithm, which shows the effectiveness of the pruning criterion when the distances between data objects have a non-low variance distribution.

## 6 Performance Evaluation

In this section we provide experimental results on the performance of the presented algorithm compared to other M-tree insertion techniques. The split policies used are those briefly described in Section 3.1. For `BulkLoading` we always use $u_{max} = 1$, and both optimization techniques presented in Section 5.

As to the sample size $k$, its value is set to $min(M, n/M)$, which equals $M$ if $n \geq M^2$, thus filling a single root node with the $k$ samples. If $n < M^2$, setting $k = M$ would likely result in underfilled sets. To avoid this, we choose $k = n/M$, thus reducing the probability of this unfavorable event.

### 6.1 The Effect of Minimum Utilization

Figures 6 and 7 show the CPU costs for building the M-tree and for 10-NN queries when the minimum storage utilization threshold, $u_{min}$, varies in the range $0.1 \ldots 0.4$ – distance selectivity is just the ratio of computed distances to the total number of objects. As expected, for increasing values of $u_{min}$ the search costs are decreasing, whereas the building costs have an increasing trend. The explanation for this behavior is that higher storage utilizations lead to a better clustering of the objects within each node. Higher utilization thresholds, however, induce higher building costs since it is more likely that the sets associated with the samples would result in a single set because of redistribution of objects in underfilled sets (in this case the distances computed so far are useless and are discarded). Search costs, though, are quite similar to those obtained with $u_{min} = 0.4$.

I/O costs (not shown here) for the building phase are quite similar for different utilization thresholds, whereas higher $u_{min}$ thresholds lead to lower I/O

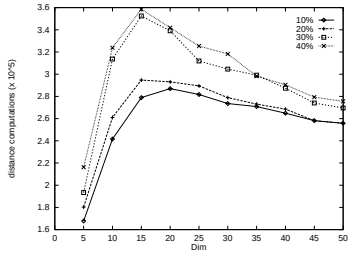


**Fig. 6.** Computed distances for building the tree.

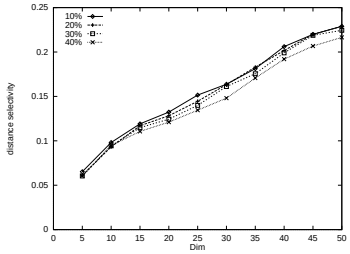


**Fig. 7.** Distance selectivity for 10-NN queries.

search costs. This is explained by the lower number of pages of trees with higher utilization and, as seen before, by the better objects' clustering of such trees, both effects inducing lower I/O costs.

### 6.2 Comparing `BulkLoading` with Standard Insertion Techniques

Now we consider how the dimensionality of the dataset influences the performance of the `BulkLoading` algorithm. As with standard insertion methods, the number of distances computed by the algorithm decreases with growing dimensionalities, whereas I/O costs have an inverse trend, as shown by Figures 8 and 9. The reason for this behavior is that increasing $Dim$ reduces the node capacity, thus reducing the size of the distance matrix to be computed during the clustering phase of the algorithm, and so the number of computed distances, but, of course, leads to larger trees. The I/O costs for the `BulkLoading` algorithm are, not surprisingly, the lowest, since this method makes massively use of internal memory, e.g. to store the distance matrices at different levels of the tree. CPU costs show that the proposed algorithm is very efficient in the index construction, with costs very similar to, if not lower than, the cheapest `RANDOM_2` split policy. Indeed, Figure 9 also shows the effectiveness of the optimization technique to reduce the number of distance computations with low dimensionalities, that can lead to save the 50% of CPU costs for $Dim = 5$. Performance on 10-NN query processing is summarized in Figures 10 and 11, where number of page I/O's and distance selectivities are shown, respectively.

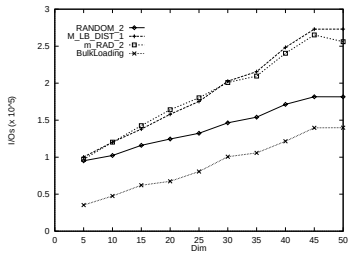


**Fig. 8.** I/O costs for building M-tree.

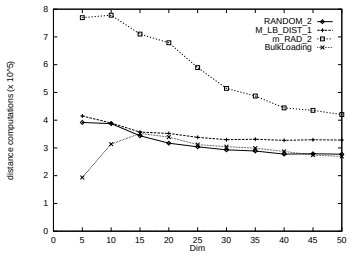


**Fig. 9.** CPU costs for building M-tree.

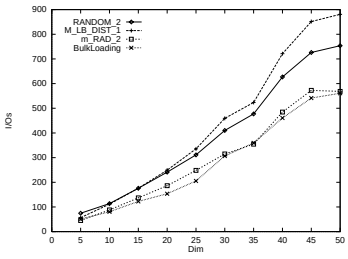


**Fig. 10.** I/O costs for processing 10-NN queries.

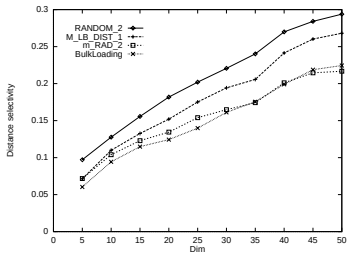


**Fig. 11.** Distance selectivity for 10-NN queries.

Experimental results demonstrate that the good clustering of objects achieved by `BulkLoading` is also reflected by query performance. In fact, the proposed algorithm has CPU and I/O costs very close to, if not better than, those of the `mM_RAD_2` strategy, the "smartest" split policy.

### 6.3 Comparing M-tree and mvp-tree

Finally, we compared the `BulkLoading` algorithm performance with that of mvp-tree [Bozkaya and Özsoyoglu, 1997], a static indexing method for metric spaces, considering only CPU costs, since mvp-tree is a primary memory organization.[4] The mvp-tree partitions the data space using spherical cuts around vantage (reference) points, and extra information for data objects is kept in the leaves in order to effectively prune the search space. An mvp-tree is characterized by 4 parameters: (1) the number of vantage points in every node (we always used 2 vantage points), (2) the number of partitions created by each vantage point ($v$), (3) the maximum capacity of leaf nodes ($f$), and (4) the number of distances to be stored for the data objects at leaves' level ($p$).

In each node of the mvp-tree, the first vantage point divides the space in $v$ parts, while the second vantage point divides each of these parts in $v$ partitions. The fanout of each node, thus, is $v^2$. To partition the space in $v$ spherical cuts, the data objects are ordered with respect to their distances from the vantage point, and then divided in $v$ groups of equal cardinality. The $v^2$ groups of data objects are, then, recursively indexed by the $v^2$ children of the root node. The leaf nodes of the tree store the exact distances between the $f$ data objects and the 2 vantage points of that leaf, as well as the $p$ distances between each data object $x$ and the first $p$ vantage points along the path from the root to the leaf node containing $x$. For our experiments we used the following values: $v = 3$, as suggested in [Bozkaya and Özsoyoglu, 1997], $f = M$, so that leaves' capacity would be the same as in the M-tree, and $p = 0$, since this optimization, that could be easily inserted in the `BulkLoading` algorithm, will cause consistency problems should the index undergo re-organizations during a split phase subsequent to an insertion (this problem is not present for mvp-tree, since this is a static index and does not permit insertions and deletions of the objects in the database).

[4] The authors thank T. Bozkaya and M. Özsoyoglu for providing the code of mvp-tree.

Results in Figures 12 and 13 compare CPU costs to search, respectively, M-trees, using the `BulkLoading` algorithm, and mvp-trees, as a function of the space dimensionality and of the number of indexed objects. Figure 12 shows the search costs for range queries with side $\sqrt[Dim]{0.01}$. The graphs display quite similar results for both the index structures, with M-tree penalized in higher dimensionalities by a 10% overhead in terms of computed distances, which should be a very good tradeoff for the dinamicity of this indexing method. Construction costs, not shown here, highly amerce the M-tree, leading to a 350% overhead of distance computations for $Dim = 15$, even if the gap decreases for increasing dimensionalities, leading to a 250% overhead for $Dim = 50$. This is because the CPU costs for `BulkLoading`, as seen, are decreasing for higher dimensionalities, while the mvp-tree has a constant, slightly increasing, trend. Figure 13 shows the search behavior of both index structures for datasets of increasing size ($10^4 \ldots 10^5$ 5-D objects): performances are very similar, with a difference almost independent of the dataset size. As for construction costs, both indices exhibit a logarithmic trend, typical of tree-like structures. The CPU overhead for M-tree, not shown here (200% for $Dim = 5$), is independent of the number of indexed objects.

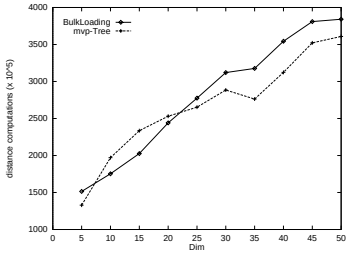


**Fig. 12.** CPU costs for range queries as a function of the space dimensionality.

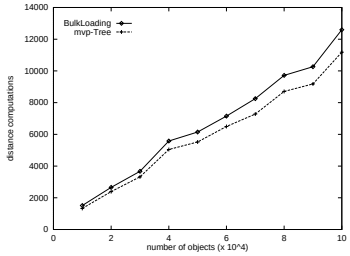


**Fig. 13.** CPU costs for range queries as a function of the number of indexed objects.

Finally, we compared the performance of the `BulkLoading` algorithm with that of mvp-tree on real datasets. The 5 datasets consist of sets of keywords extracted from masterpieces of Italian literature. The metric used was the *edit* (Levenshtein) distance, i.e. the minimal number of changes (insertions, deletions, substitutions of characters) needed to transform a string into another one. Table 1 describes each dataset and shows the average number of distance computations required for a range query with radius $r_Q = 3$, averaged over 1000 queries, for both indexing methods.

| Dataset | Size | `BulkLoading` | mvp-tree |
|---|---|---|---|
| Decamerone | 17396 | 12455.71 | 12537.13 |
| Divina Commedia | 12701 | 9630.857 | 10189.47 |
| Gerusalemme Liberata | 11973 | 9092.99 | 9528.955 |
| Orlando Furioso | 18719 | 13764.49 | 14761.37 |
| Promessi Sposi | 19846 | 13872.94 | 13991.98 |

**Table 1.** Performance for range queries on real datasets

## 7 Conclusions

The M-tree is a new access method able to index dynamic datasets from generic metric spaces. In this paper we have presented and evaluated a bulk loading algorithm for M-tree, comparing it with traditional insertion techniques and with other metric trees. We also developed two different techniques to reduce the number of distance computed by the algorithm during the building phase. Results show that the presented algorithm achieves a good clustering of data objects, leading to very low search costs, and having not very high construction costs, and that its search performance is similar to that of other metric indexing structures, that suffer from being intrinsically static.

## References


Bozkaya, T. and Özsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. SIGMOD'97, pages 357–368, Tucson, AZ.

Brin, S. (1995). Near neighbor search in large metric spaces. VLDB'95, pages 574–584, Zurich, Switzerland.

Chiueh, T. (1994). Content-based image indexing. VLDB'94, pages 582–593, Santiago, Chile.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. VLDB'97, pages 426–435, Athens, Greece.

Faloutsos, C. and Lin, K.-I. (1995). FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. SIGMOD'95, pages 163–174, San Jose, CA.

Gavrila, D.M. (1994). R-tree index optimization. Technical Report CS-TR-3292, University of Maryland, College Park.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. SIGMOD'84, pages 47–57, Boston, MA.

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for Clustering Data*. Prentice-Hall.

Kamel, I. and Faloutsos, C. (1993). On packing R-trees. CIKM'93, pages 490–499, Washington, DC.

Leutenegger, S. T., Lopez, M. A., and Edgington, J. (1997). STR: A simple and efficient algorithm for R-tree packing. ICDE'97, pages 497–506, Birmingham, UK.

Li, K.-J. and Laurini, R. (1991). The spatial locality and a spatial indexing method by dynamic clustering in hypermap systems. SSD'91, pages 207–223, Zurich, Switzerland.

Lo, M.-L. and Ravishankar, C. V. (1995). Generating seeded trees from data sets. SSD'95, pages 328–347, Portland, ME.

Roussopoulos, N. and Leifker, D. (1985). Direct spatial search on pictorial databases using packed R-trees. SIGMOD'85, pages 17–31, Austin, TX.

Shapiro, M. (1977). The choice of reference points in best-match file searching. Comm. of the ACM, 20(5):339–343.

van den Bercken, J., Seeger, B., and Widmayer, P. (1997). A generic approach to bulk loading multidimensional index structures. VLDB'97, pages 406–415, Athens, Greece.

Zhang, T., Ramakrishnan, R., and Livny, M. (1996). BIRCH: An efficient data clustering method for very large darabases. SIGMOD'96, pages 103–114, Montreal, Canada.