# Bulk Loading the M-Tree to Enhance Query Performance

**Conference Paper** *in* Lecture Notes in Computer Science · July 2004

DOI: 10.1007/978-3-540-27811-5_18 · Source: DBLP



**2 authors**, including:

Alan P. Sexton
University of Birmingham
**49** PUBLICATIONS **337** CITATIONS

# Bulk Loading the M-tree to Enhance Query Performance

Alan P. Sexton and Richard Swinbank

School of Computer Science, University of Birmingham
Edgbaston, Birmingham, B15 2TT, UK
{A.P.Sexton, R.J.Swinbank}@cs.bham.ac.uk

**Abstract.** The M-tree is a paged, dynamically balanced metric access method that responds gracefully to the insertion of new objects. Like many spatial access methods, the M-tree's performance is largely dependent on the degree of overlap between spatial regions represented by nodes in the tree, and minimisation of overlap is key to many of the design features of the M-tree and related structures. We present a novel approach to overlap minimisation using a new bulk loading algorithm, resulting in a query cost saving of between 25% and 40% for non-uniform data.

The structural basis of the new algorithm suggests a way to modify the M-tree to produce a variant which we call the SM-tree. The SM-tree has the same query performance after bulk loading as the M-tree, but further supports efficient object deletion while maintaining the usual balance and occupancy constraints.

## 1 Introduction

The expansion of database systems to include non-alphanumeric datatypes has led to a need for index structures by which to query them. Tree-based index structures for traditional datatypes rely heavily on these datatypes' strict linear ordering; this is unsurprising as it is this same property that we use naturally in discussing ordered data, using notions such as 'before', 'after' and 'between'.

Newer datatypes such as images and sounds possess no such natural linear ordering, and in consequence we do not attempt to exploit one, but rather evaluate data in terms of their relative *similarities*: one image is 'like' another image but is 'not like' another different image. This has led to the notion of *similarity searching* for such datatypes, and the definition of queries like *k Nearest Neighbour*: find the $k$ objects in the database at the shortest distance from a query object, where *distance* is some measure of the dissimilarity between objects.

Spatial access methods such as the R-tree [3] can support similarity queries of this nature by abstracting objects as points in a multidimensional vector space and calculating distance using a Euclidean metric. A more general approach is to abstract objects as points in a metric space, in which a distance function is known, but absolute positions of objects in a Cartesian space need not be. This

renders consideration of dimensionality unnecessary, and so provides a single method applicable to all (and unclear or unknown) dimensionalities.

The first paged, dynamically balanced metric tree was the M-tree [2]. The M-tree preserves the notion of objects' *closeness* more perfectly than in earlier structures (e.g. [6]) by associating a *covering radius* with pointers above the leaf level in the tree, indicating the furthest distance from the pointer at which an object in its subtree might be found. This, in combination with the triangle inequality property of the metric space, permits branches to be pruned from the tree when executing a query.

For a query result to be found in a branch rooted on a pointer with reference value $O_n$, that result must be within a distance $r(Q)$ (the *search radius*) of the query object $Q$. By definition, all objects in the branch rooted on $O_n$ are also within $r(O_n)$ (the covering radius) of $O_n$, so for a result to be found in the branch rooted on $O_n$, the regions defined by $r(Q)$ around $Q$ and $r(O_n)$ around $O_n$ must intersect. This is a statement of the triangle inequality: For an object answering query $Q$ to be found in the subtree rooted on $O_n$, it must be true that

$$d(Q, O_n) \leqslant r(Q) + r(O_n)$$

so when $O_n$ is encountered in descending the tree, $d(Q, O_n)$ can be calculated in order to decide whether to prune or to descend the branch rooted on $O_n$. Figures 1 and 2 illustrate the two possibilities.

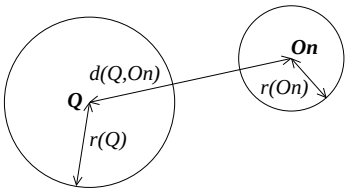


**Fig. 1.** The branch rooted on $O_n$ can be pruned from the search.

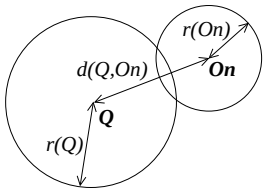


**Fig. 2.** The branch rooted on $O_n$ cannot be pruned.

An alternative to loading a tree by a series of object insertions was presented in [1]. This *bulk loading* algorithm selects a number of "seed" objects around which other objects are recursively clustered to build an unbalanced tree which must later be re-balanced. This provides construction cost saving of up to 50%, with query costs very similar to those of insertion-built trees.

Like many spatial access methods, the M-tree's performance is largely dependent on the degree of overlap between spatial regions represented by nodes in the tree. Minimisation of overlap is key to many of the design features of the M-tree and related structures; some approaches are discussed later. In this paper we present an alternative approach: that of pre-clustering an existing data set in a way that reflects the performance requirements of the M-tree before bulk loading the data into a tree. Unlike conventional bulk loading algorithms, the objective here is not efficient *loading* of the tree, but rather a subsequent tree organisation that maximises query performance. The structural basis of the new algorithm suggests a way to modify the M-tree to produce a variant which we

call the SM-tree. The SM-tree has the same query performance after bulk loading as the M-tree, and supports efficient object deletion within the constraints of node balance and occupancy.

## 2 Insertion into the M-tree

Insertion of an object $O_i$ into an M-tree proceeds as follows. From the root node, an entry pointing to a child node is selected as the most appropriate parent for $O_i$. The child node is retrieved from disk and the process is repeated recursively until the entry reaches the leaf level in the tree.

A number of suggestions have been made as to how the 'best' subtree should be chosen for descent. The original implementation of the M-tree selects, if possible, a subtree for which zero expansion of covering radius is necessary, or, if not possible, the subtree for which the required expansion is least. The Slim-tree [4] further offers a randomly selected subtree or a choice based on the available physical (disk) space in the subtree. In all of these variations, in the event that the covering radius of the selected node entry $O_n$ must be expanded to accommodate the entry, it is expanded to $d(O_n, O_i)$ as $O_i$ passes $O_n$ on its way to the leaf level. This is the smallest possible expansion that maintains a correct covering radius, and thus increases overlap the least.

Having reached a leaf, $O_i$ is inserted if it fits, otherwise the leaf node is split into two with leaf entries being partitioned into two groups according to some strategy, referred to as the *splitting policy*. Pointers to the two leaves are then promoted to the level above, replacing the pointer to the original child. On promotion, the covering radius of each promoted node entry $O_p$ is set to:

$$r(O_p) = \max_{O_l \in \mathcal{L}} \{d(O_p, O_l)\}$$

where $\mathcal{L}$ is the set of entries in the leaf. If there is insufficient space in the node to which entries are promoted, it too splits and promotes entries. When promoting internally, the covering radius of each promoted entry is set to:

$$r(O_p) = \max_{O_n \in \mathcal{N}} \{d(O_p, O_n) + r(O_n)\}$$

where $\mathcal{N}$ is the set of entries in the node. This applies the limiting case of the triangle inequality property of the metric space as an upper bound on $\max_{O_l \in \mathcal{L}} \{d(O_p, O_l)\}$ where $\mathcal{L}$ is the set of all leaf node entries in the subtree rooted on $O_p$. Use of this upper bound avoids the requirement for an exhaustive search of the subtree on every promotion.

The splitting policy offers a second opportunity to minimise overlap, albeit within the constraint of balanced node occupancy. The M-tree offers a number of heuristically-developed alternatives, the best of which selects the partitioning for which the larger of the two covering radii is minimal. The Slim-tree constructs a minimal spanning tree across the entries to be partitioned, and selects the longest edge of this as being the point at which the set of entries most closely resembles two clusters, and is thus likely to suffer least from overlap between the resulting node pair.

## 3 A Clustering Technique for Bulk Loading

As we have seen, the process of insertion attempts to locate new objects in nodes that contain other objects that are in close spatial proximity. After an M-tree has been constructed by the insertion of a set of objects, we may therefore observe that the set has been partitioned into a number of clusters of objects, each of which is contained in a leaf node of the tree. The degree of overlap between the regions represented by these leaf nodes is an indicator of the quality of the clustering obtained by M-tree insertion.

This observation suggests an alternative approach: cluster a pre-existing data set directly into a set of leaf nodes to minimise overlap globally across the data set, then construct an M-tree upwards from the leaf level. When building an M-tree from the bottom up, we set *all* covering radii at the limit of the triangle inequality, in a similar way to the case of entry promotion after node splitting, to avoid the requirement for an exhaustive search of each node entry's subtree. Our optimisation is to provide better organisation of the data than that achieved by the M-tree, replacing the M-tree's optimisation of limiting covering radii expansion at insertion, and offering significant performance benefits.

### 3.1 Desirable Features

In this section we discuss clusters in the context of M-tree nodes: a cluster is a collection of points that may be accomodated within a single disk page. A cluster's location is considered to be that of its medoid, as its centroid cannot be calculated exactly in a purely metric, non-Cartesian space. The cluster *radius* is the distance between the medoid and its furthest neighbour in the cluster. The cluster *boundary* falls at this distance from the medoid.

An unusual consideration of clustering algorithms for bulk-loading is the overriding importance of cluster size, even at the expense of cluster quality. All clusters produced for bulk loading must be sufficiently small to fit into a disk page while also being sufficiently large to exceed its underflow limit, typically 50%.

Another consideration is that of outlying points that do not clearly belong to any cluster. Figure 3(a) shows a group of points that we perceive as being in four clusters, in addition to another four points that do not clearly belong anywhere. The page underflow limit is unlikely to permit each outlier to occupy a node alone, although merging each outlier into its nearest cluster has the effect of making all clusters' M-tree representations mutually overlap, as in Fig. 3(b). In this situation, a better effect on the global cluster population might be achieved by retaining the four good clusters and making one (extremely!) bad cluster, as in Fig. 3(c). In this case, rather than each cluster overlapping three others, as in Fig. 3(b), each good cluster solely overlaps the single bad cluster.

### 3.2 The Clustering Algorithm

We present below a clustering algorithm that achieves the two objectives described above: guaranteed cluster size of between $n$ and $2n$ points for disk pages

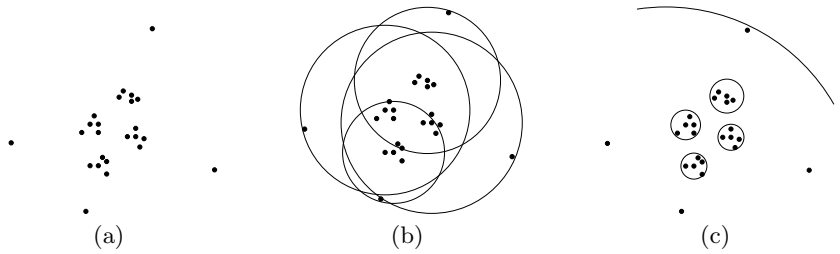


**Fig. 3.** The effect of outlying points on cluster overlap. The boundary of the bad cluster in (c) is shown only in part.

that can contain a maximum of $2n$, and a tendency to find good clusters first, leaving outlying points to be grouped into poorer clusters later.

We make use of the following definitions:

- A cluster medoid is any point $m$ in the cluster for which there exists no other point $p$ in the cluster that, if nominated as a medoid, would give the cluster a radius less than that given by $m$.
- A cluster may have multiple medoids. We impose an arbitrary linear order on points in the metric space and define the primary medoid of a cluster to be the least medoid of the cluster in this order.
- The distance between two clusters is that between their primary medoids.
- Given a cluster $c$, a nearest neighbour of $c$ in a set of clusters $\mathcal{C}$ is a cluster $c'$ such that there is no other cluster in $\mathcal{C}$ whose distance to $c$ is less than that between $c$ and $c'$. ($c$ may have multiple nearest neighbours in $\mathcal{C}$.)
- A *closest pair* of clusters in $\mathcal{C}$ is a pair of clusters $c_1, c_2 \in \mathcal{C}$ such that $d(c_1, c_2) \leqslant d(c_i, c_j)$ for all $c_i, c_j \in \mathcal{C}$.

```
Function Cluster (
   CMAX: maximum acceptable cardinality of a cluster,
   C_in: a set of at least CMAX/2 points
)
Returns a set of clusters, each of cardinality in [CMAX/2, CMAX].

Let C_out = {} ;
Let C = {} ;
For each p ∈ C_in
   Add {p} to C ;

While |C| > 1
   Let c_1, c_2 be a closest pair of clusters in C such that |c_1| ⩾ |c_2| ;
   If |c_1 ∪ c_2| ⩽ CMAX
      Remove c_1 and c_2 from C ;
      Add c_1 ∪ c_2 to C ;
   Else
      Remove c_1 from C ;
      Add c_1 to C_out ;

Let c be the last remaining element of C ;
```

```
If |C_out| > 0
   Let c' be a nearest neighbour of c in C_out ;
   Remove c' from C_out ;
Else
   Let c' = {};
If |c ∪ c'| ≤ CMAX
   Add c ∪ c' to C_out ;
Else
   Split c ∪ c' into c_1 and c_2 using the insertion splitting policy;
   Add c_1 and c_2 to C_out ;
Return C_out ;
```

The first phase of the clustering algorithm converts the input set of points into a set of singleton clusters, while the work of clustering occurs almost entirely within the loop structure of the second phase. In the final phase, if the last remaining element of $\mathcal{C}$ contains fewer than $\frac{CMAX}{2}$ points, its points and those of its nearest neighbour are redistributed to ensure that no cluster breaks the minimum size bound.

### 3.3 Bulk Loading

**Preliminaries.** The bulk load algorithm makes use of the clustering algorithm and definitions given above, and the following definitions:

- An M-tree leaf node entry is a pair $(p, d)$ where $p$ is a point and $d$ is the distance from the point $p$ to its parent, *i.e.* a distinguished point of the current node.
- An M-tree internal node entry is a tuple $(p, d, r, a)$ where $p$ is a point, $d$ a parent distance (as in the case of a leaf node entry), $r$ is the covering radius of the subtree, *i.e* a value that is at least as large as the maximum distance from $p$ to any point in the subtree rooted at this entry, and $a$ is the disk address of the child page identified by this internal node entry.

Furthermore, in the bulk load algorithm we call subroutines for writing leaf and internal node pages to disk. These are as follows:

```
Function OutputLeafPage(
   C_in: a set of points of cardinality no
      greater that will fit into a disk page
)
Returns a tuple (m, r, a) where:
   m is the primary medoid of C_in,
   r is called the covering radius,
   a is the disk address of the page output.

Let m = primary medoid of C_in;
Let r=0;
```

```
Let C = {};
For each p ∈ C_in
   Add (p, d(m,p)) to C ;
   Let r = Max(r, d(m,p));
Allocate new disk page a ;
Output C as a leaf page to disk address a ;
Return (m, r, a) ;


Function OutputInternalPage (
   C_mra: a set of (m, r, a) tuples as returned from OutputLeafPage
)
Returns (M, R, A) where:
   M is the primary medoid of the set of
      points C_in = {m|∃(m, r, a) ∈ C_mra},
   R is called the covering radius,
   A is the disk address of the page output.

Let C_in = {m|∃(m, r, a) ∈ C_mra} ;
Let M = primary medoid of C_in ;
Let R=0;
Let C = {};
For each (m, r, a) ∈ C_mra
   Add (m, d(M,m), r, a) to C ;
   Let R = Max(R, d(M,m) + r);
Allocate new disk page A ;
Output C as an internal page to disk address A ;
Return (M, R, A) ;
```

**The Bulk Loading Algorithm.** This bulk load algorithm uses the clustering algorithm to obtain a set of clusters suitable for writing to disk as one complete cluster per page. As each level of the tree is fully written to disk, entries for the level above are accumulated. The algorithm terminates when the set of entries to be written into the next level fit into a single page; the root of the M-tree.

```
Function BulkLoad(
   C_in: a set of points in a metric space,
   CMAX: maximum number of node entries that will fit into a disk page
)
Returns disk address of root page of constructed M-tree

If C_in ⩽ CMAX
   Let (m, r, a) = OutputLeafPage(C_in);
   Return a ;

Let C_out = Cluster(C_in, CMAX) ;
Let C = {};
For each c ∈ C_out
```

```
   Add OutputLeafPage(c) to C ;

While |C| > CMAX
   Let C_in = {m|(m,r,a) ∈ C} ;
   Let C_out = Cluster(C_in, CMAX) ;
   Let C_mra = {};
   For each c ∈ C_out
      Let s = {(m,r,a)|(m,r,a) ∈ C ∧ m ∈ c} ;
      Add s to C_mra ;
   Let C = {} ;
   For each s ∈ C_mra
      Add OutputInternalPage(s) to C ;

Let (m,r,a) = OutputInternalPage(C);
Return a ;
```

We present `BulkLoad` here as using `Cluster` as a subroutine for reasons of clarity. An optimisation for implementation is to interleave the two to save distance computations used in clustering. This avoids re-computation of parent distances and covering radii during `BulkLoad`.

The clustering algorithm produces clusters containing between $\frac{CMAX}{2}$ and $CMAX$ points and (experimentally) on average $0.8 * CMAX$ points. This produces trees with an average page occupancy of 80%. Note that in the above algorithm we refer to cluster size merely in terms of numbers of points, however a change to reflect physical (on-disk) cluster size would permit consideration of variable-sized tree entries.

## 4 Experimental Evaluation

### 4.1 Details of Implementation

For experimental evaluation, a series of M-trees were constructed by bulk loading and by serial insertion of a set of 25 000 objects in 2, 4, 6, 8, 10, 15 and 20 dimensions. Data objects were implemented as points in a 20-dimensional vector space, enabling dimensionality of experiments to be varied simply by adjusting the metric function to consider a fewer or greater number of dimensions, while maintaining a constant object size. Trees built by insertion used the original M-tree's *MinMax* split policy and an underflow limit of 50%. All trees were built on 4kB pages and used the $d_2$ (Euclidean) metric for $n$ dimensions:

$$d_2(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

for $x = (x_1, x_2, ..., x_n), y = (y_1, y_2, ..., y_n)$.

Experiments were performed using an artifically clustered data set, produced by distributing randomly-generated points around other, also randomly-generated, seed points. A trigonometric function based distribution was chosen to produce a higher point density closer to seed points, although each vector component was produced independently, resulting in regions of higher point density parallel to coordinate axes (see Fig. 4; dimensions 1 and 2 of the 20-dimension data set).

Each tree was required to process a series of 1, 10 and 50 nearest-neighbour queries. Query performance is measured in terms of page-hits (IOs) assuming an initially empty, infinite buffer pool, and in all cases is averaged over 100 queries.

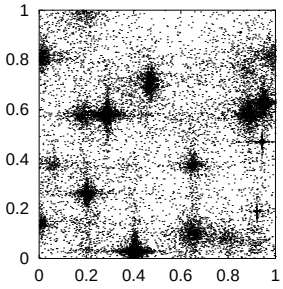


**Fig. 4.** Experimental data distribution

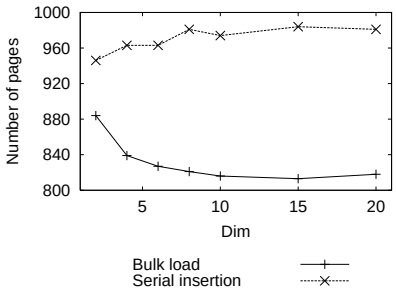


**Fig. 5.** Comparative tree sizes

### 4.2 Results and Discussion

Figure 5 indicates the comparative sizes of the trees built by bulk load and serial insertion. As remarked previously, the bulk load clustering algorithm produces page occupancies of around 80%, compared to 70% produced by the M-tree's insert algorithm. The higher occupancy of bulk loaded pages is reflected in the requirement for fewer pages to contain the full tree.

Figures 6 and 7 show very similar pictures of query performance in the two types of tree for 1-NN and 50-NN queries. In both cases the bulk loaded tree outperforms the insertion-built tree, making a query cost saving of between 25% and 40% in terms of absolute number of page hits. Even when normalised to take account of the more heavily occupied bulk loaded tree, we find that (for example) in 20 dimensions a 50-NN query is answered by reading fewer than 15% of the bulk-loaded tree's pages compared to 20% in the insertion-built M-tree.

These performance figures are readily understood in terms of reduced overlap by considering the leaf radii distributions of bulk loaded and insertion-built M-trees. Figure 8 shows the distribution of leaf node covering radii in M-trees in 2 dimensions. As one might expect from the clustering algorithm, explicitly outputting good clusters first leads to a higher number of low-radius clusters and a distribution which appears to tail off sooner. Not shown in the figure for reasons of scale however is the intermittent *longer* tail-off of the bulk-loaded

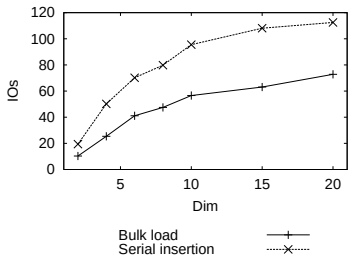


**Fig. 6.** 1-NN query

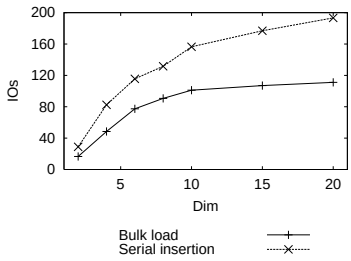


**Fig. 7.** 50-NN query

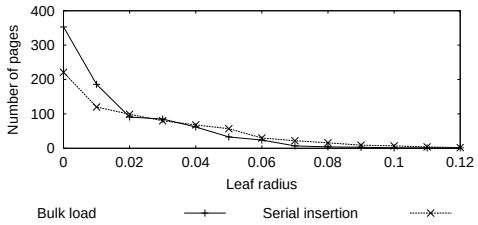


**Fig. 8.** Leaf radii distribution in 2 dimensions

tree's radius distribution. This is a direct consequence of allowing outliers to form unusually bad clusters.

## 5 A Symmetric M-tree (SM-tree)

### 5.1 Asymmetry in the M-tree

Although the M-tree grows gracefully under Insert, there has, to date, been no algorithm published for the complementary Delete operation. The authors of [4] explicitly state in their discussion of the Slim-tree that neither their structure nor the original M-tree yet support Delete. We find that implementation of the delete algorithm is non-trivial as a direct consequence of an aspect of asymmetry introduced by the M-tree's overlap minimising optimisation at insertion.

A critical observation, with respect to the Delete problem, is that in a bulk loaded tree, the covering radius of *any* node entry in the tree is dependent solely on the distance from its immediate children, and the covering radii of those children. In an insertion-built M-tree this is only true in node entries newly-promoted from a lower-level node split. Ordinarily, when an object for insertion $O_i$ passes a node entry and expands its covering radius to $d(O_n, O_i)$, the new covering radius depends on the entire contents of the node entry's subtree, and can only be specified exactly as:

$$r(O_n) = \max_{O_l \in \mathcal{L}} \{d(O_n, O_l)\}$$

where $\mathcal{L}$ is the set of *all* leaf entries in the subtree rooted on $O_n$.

This effect is illustrated in Fig. 9, which shows three levels of an M-tree branch. Figure 9(a) shows leaf entries **B** and **C**, under a subtree pointer with

reference value **B**. This subtree's covering radius is currently contained within that of its own parent, **A**. In Fig. 9(b), insertion of point **D** causes a slight expansion of the radius around **A**, but expands the child's radius *beyond* that of its parent. Thus the correct covering radius around **A** is no longer calculable from the distance to, and the radii of, **A**'s immediate children. The decision to

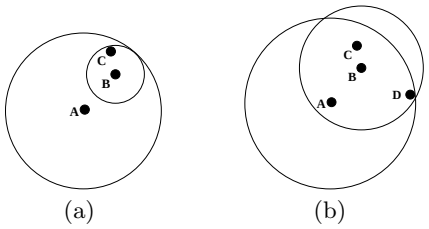


**Fig. 9.** The effect of the M-tree's Insert algorithm on covering radii

expand the covering radius only as far as is immediately necessary and no further therefore introduces asymmetry between the Insert and (unimplemented) Delete operations: Insert adds an object and may expand covering radii, but conversely Delete cannot contract a node entry's covering radius without reference to *all* objects at the leaf level in its subtree, thus requiring an expensive subtree walk for correct implementation.

### 5.2 Maintaining Symmetry After Bulk Loading

Given that, after bulk loading with our algorithm, the covering radii of all node entries are dependent solely on their immediate children, we may observe that the tree is symmetric with respect to Insert and Delete. Maintenance of this symmetry permits support of object deletion, but requires modification of the tree's insert algorithm to fully expand covering radii to the limit of the triangle inequality on further insertion. We refer to trees whose symmetry is maintained in this way as symmetric M-trees (SM-trees). Full algorithms and analysis of the SM-tree are presented in [5] but are omitted here for lack of space.

The principal modification of the insertion algorithm is this: in cases when insertion does not induce a node split, any expansion of node radius must propagate back up the tree from the leaf level towards the root. This is achieved by returning, from a call to Insert, the resulting radius of the node into which the insertion was made. At the leaf level this is simply the M-tree leaf radius:

$$\max_{O_l \in \mathcal{L}} \{d(O_p, O_l)\}$$

where $\mathcal{L}$ is the set of entries in the leaf, while at higher levels the node radius is:

$$\max_{O_n \in \mathcal{N}} \{d(O_p, O_n) + r(O_n)\}$$

where $\mathcal{N}$ is the set of entries in the node. This is exactly the node radius propagated from an internal node split in the M-tree, however in the SM-tree it is maintained as an invariant. Node entry and radius propagation at a node split

is managed in exactly the same way as in the M-tree. The disk I/O cost of the modified Insert algorithm is therefore still $O(h)$, where $h$ is the height of the tree.

The choice of subtree for insertion is made by finding the node entry closest to the entry being inserted $O_i$ (i.e. the entry $O_n \in \mathcal{N}$ for which $d(O_n, O_i)$ is a minimum), rather than by attempting to limit the expansion of existing covering radii, because it is no longer possible while descending the tree to make any assertions about the effect of that choice on the radius of the selected subtree.

The choice made in the original M-tree was based on the heuristic that we wish to minimise the overall volume covered by a node $N$. In the SM-tree, unlike the original M-tree, all node entry covering radii entirely contain their subtrees, suggesting that subtrees should be centred as tightly as possible on their root (within the constraint of minimising overlap between sibling nodes) to minimise the volume covered by $N$.

### 5.3 Deletion from the SM-tree

After bulk loading, maintenance of insert/delete symmetry in the SM-tree using the modified insert algorithm permits support of the delete operation. Covering radii may now be returned by an implementation of the delete operation in the same way that they are by the modified insert algorithm, and permit node entry covering radii to contract as objects are deleted. Furthermore, as a node entry's covering radius is no longer directly dependent on the distance between itself and leaf-level entries in its subtree, node entries can be distributed between other nodes, permitting underflown internal nodes to be merged with other nodes at the same level.

The deletion algorithm proceeds in a similar manner to an exact match query, exploiting the triangle inequality for tree pruning, followed by the actions required to delete an object if it is found, and handle underflow if it occurs. In the simple (non-underflowing) case, calls to Delete, like those to Insert, return the (possibly contracted) covering radius of the subtree.

When a node underflows, the full set of entries from the underflown node is returned from Delete, and must be written into alternative nodes. Although not explored here, this suggests the possibility of a *merging policy* analogous to the splitting policy used at node overflow. Our implementation of underflow handling is relatively simple, merging returned entries with those contained in the child of the nearest neighbour of the underflown node's parent. If the merged collection fits into a single node, that node remains, otherwise the insertion splitting policy is used to redistribute the entries appropriately. As in the B-tree, the Delete algorithm's complexity is $O(h)$, where $h$ is the height of the tree.

### 5.4 Evaluation of the SM-tree

The performance of a bulk-loaded SM-tree matches that of a bulk-loaded M-tree, indeed they are structurally identical, while unlike the M-tree, the SM-tree is

also able to support object deletion. As might be expected however, insertion-built SM-trees suffer rather from losing the optimisation of limited expansion of covering radii. In [5] we show results that suggest (for insertion-built trees only) an average query cost increase of 15% for the SM-tree compared to the M-tree. In order to take advantage of the fully dynamic behaviour offered by the SM-tree, we suggest bulk loading a tree first wherever possible.

## 6 Conclusions and Further Work

In this paper we presented a clustering algorithm designed specifically to meet the requirements of a bulk load algorithm for the M-tree. Unlike conventional bulk loading algorithms our objective was not to efficiently load the tree, but rather to maximise its subsequent query performance. We have shown that the performance of bulk loaded trees exceeds that of insertion-built M-trees by between 25% and 40% for non-uniform data. We further observe that bulk loaded trees have a particular property with respect to the symmetry of the insert and delete operations, which, if maintained, permits support of both.

The clustering algorithm seeks out concentrations of points in the data set while protecting clusters from the effect of outlying points. One direction for future work is the extension of the algorithm to handle near-uniform data with the same degree of success. Another avenue is suggested by the observation that our approach to bulk loading is simply an alternative optimisation to improve the tree's query performance: we are currently searching for a way to avoid the need for such opportunistic optimisations altogether, and rely solely on a tree's intrinsic properties to provide efficient search.

## References


1. P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)*, pages 15–26, Perth, Australia, 1998.
2. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB Conference*, pages 426–435, Athens, Greece, 1997.
3. A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57, June 1984.
4. C. Traina Jr., A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260, March/April 2002.
5. Alan P. Sexton and Richard Swinbank. Symmetric M-tree. Technical Report CSR-04-2, University of Birmingham, UK, 2004. Available at URL `www.cs.bham.ac.uk/~rjs/research`.
6. J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.