DBM-Tree: A Dynamic Metric Access Method Sensitive to Local Density Data [*]

Marcos R. Vieira, Caetano Traina Jr., Fabio J. T. Chino, Agma J. M. Traina

ICMC – Institute of Mathematics and Computer Sciences
USP – University of São Paulo at São Carlos
Avenida do Trabalhador São-Carlense, 400
CEP 13560-970 – São Carlos – SP – Brazil

{mrvieira, caetano, chino, agma}@icmc.usp.br

Abstract

Metric Access Methods (MAM) are employed to accelerate the processing of similarity queries, such as the range and the k-nearest neighbor queries. Current methods improve the query performance minimizing the number of disk accesses, keeping a constant height of the structures stored on disks (height-balanced trees). The Slim-tree and the M-tree are the most efficient dynamic MAM so far. However, the overlapping between their nodes has a very high influence on their performance. This paper presents a new dynamic MAM called the DBM-tree (Density-Based Metric tree), which can minimize the overlap between high-density nodes by relaxing the height-balancing of the structure. Thus, the height of the tree is larger in denser regions, in order to keep a tradeoff between breadth-searching and depth-searching. Moreover, an optimization algorithm called Shrink is also presented, which improves the performance of an already built DBM-tree by reorganizing the elements among their nodes. Experiments performed over both synthetic and real datasets showed that the DBM-tree is, in average, 50% faster than traditional MAM and reduces the number of distance calculations by up to 72% and disk accesses by up to 54%. After performing the Shrink algorithm, the performance improves up to 30% regarding the number of disk accesses for range and k-nearest neighbor queries. In addition, the DBM-tree scales up well, exhibiting sub-linear performance with growing number of elements in the database.

1. Introduction

The volume of data managed by the Database Management Systems (DBMS) is increasing continually. Moreover, new complex data types, such as multimedia data (image, audio, video and long text), geo-referenced information, time series, fingerprints, genomic data and protein sequences, among others, have been added to DBMS.

The main technique employed to accelerate data retrieval in DBMS is indexing the data using Access Methods (AM). The data domains used by traditional databases, i.e. numbers and short character strings, have the total ordering property. Every AM used in traditional DBMS to answer both equality and relational ordering predicates, such as the B-trees, are based on this property.

Unfortunately, the majority of complex data domains do not have the total ordering property. The lack of this property precludes the use of traditional AM to index complex

[*]This work has been supported by FAPESP (São Paulo State Research Foundation) under grants number 01/11987-3, 01/12536-5 and 02/07318-1, and CNPq (Brazilian National Council for Supporting Research) under grants 52.1685/98-6, 52.1267/96-0 and 860.068/00-7.

data. Nevertheless these data domains allow the definition of similarity relations among pairs of objects. Similarity queries are more natural for these data domains. For a given reference object, also called the query center object, a similarity query returns all objects that meet a given similarity criteria. Traditional AM rely on the total order relationship only, and are not able to handle these complex data properly, neither to answer similarity queries over such data. These restrictions led to the development of a new class of AM, the Metric Access Methods (MAM), which are well-suited to answer similarity queries over complex data types.

The MAM such as the Slim-tree [12, 13] and the M-tree [7] were developed to answer similarity queries based on the similarity relationships among pairs of objects. The similarity relationships are usually represented by distance functions computed over the pairs of objects of the data domain. The data domain and distance function defines a metric space or metric domain.

Formally, a metric space is a pair $< \mathbb{S}, d() >$, where $\mathbb{S}$ is the data domain and $d()$ is a distance function that complies with the following three properties: **symmetry**: $d(s_1, s_2) = d(s_2, s_1)$; **non-negativity**: $0 < d(s_1, s_2) < \infty$ if $s_1 \neq s_2$ and $d(s_1, s_1) = 0$; and **triangular inequality**: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2), \forall s_1, s_2, s_3 \in \mathbb{S}$. Vector datasets with any L_p distance function, such as Euclidean distance (L_2), are special cases of metric spaces. The two main types of similarity queries are:

- **Range query -** Rq: given a query center object $s_q \in \mathbb{S}$ and a maximum query distance r_q, the query $Rq(s_q, r_q)$ retrieves every object s_i such as $d(s_i, s_q) \leq r_q$. An example is: "Select the proteins that are similar to the protein P by up to 5 purine bases", and it is represented as $Rq(P, 5)$;
- k**-Nearest Neighbor query -** $kNNq$: given a query center object $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, the query $kNNq(s_q, k)$ retrieves the k objects that have the smallest distance from the query object s_q, according to the distance function $d()$. An example is: "Select the 3 protein most similar to the protein P", where k=3, and it is represented as $kNNq(P, 3)$.

This paper presents a new dynamic MAM called DBM-tree (*Density-Based Metric tree*), which can minimize the overlap of nodes storing objects of high-density regions relaxing the structure height-balance. Therefore, the height of a DBM-tree is larger in higher-density regions, in order to keep a compromise between the number of disk accesses required to breadth-search various subtrees and to depth-searching one subtree. As the experiments will show, its performance is better to answer similarity queries than the rigidly balanced trees. This article also presents an algorithm to optimize DBM-trees, called *Shrink*, which improves the performance of these structures reorganizing the elements among the tree's nodes.

The experiments performed over synthetic and real datasets show that the DBM-tree outperforms the traditional MAM, such as the Slim-tree and the M-tree. The DBM-tree is, in average, 50% faster than these traditional balanced MAM, reducing up to 54% the number of disk accesses and up to 72% the number of distance calculations required to answer similarity queries. The *Shrink* algorithm, helps to achieve improvements of up to 30% in number of disk accesses to answer range and k-nearest neighbor queries. Moreover, the DBM-tree is scalable, exhibiting sub-linear behavior in the total processing time, the number of disk accesses and the number of distance calculations regarding the number of elements indexed.

The remainder of this paper is structured as follows: Section 2 presents the basic concepts and Section 3 summarizes the related works. The new metric access method DBM-tree is presented in Section 4. Section 5 describes the experiments performed and the results obtained. Finally, Section 6 gives the conclusions of this paper and suggests future works.

2. Basic Concepts

An **Access Method** (AM) is the most used resource of DBMS to improve their performance on retrieval operations. The use of meaningful properties from the objects indexed is fundamental to achieve improvements. Using the properties of the data domain, it is possible to discard large subsets of data without comparing every stored object with the query object. For example, consider the case of numeric data, where the total ordering property holds: this property allows dividing the stored numbers in two sets: those that are greater and those that are smaller than or equal to the query reference number. Hence, the fastest way to perform the search is maintaining the numbers sorted, so that when a search for a given number is required, comparing this number with a stored object enables discarding further comparisons with the part of the data where the number cannot be in.

An important class of AM is the one that forms hierarchical structures (trees), which enables recursive processes to index and search the data. In a tree, the objects are divided in blocks called nodes. When a search is needed, the query object is compared with one or more object in the root node, determining which subtrees need to be traversed, recursively repeating this process for each subtree that is able to store answer objects.

Note that whenever the total ordering property applies, only a subtree at each tree level can hold the answer. If the data domain has only a partial ordering property, then it is possible that more than one subtree need to be analyzed in each level. As numeric domains possess the total ordering property, the trees indexing numbers requires the access of only one node in each level of the structure. On the other hand, trees storing spatial coordinates, which only have a partial ordering property, require searches in more than one subtree in each level of the structure. This effect is known as covering, or overlapping between subtrees, and occurs for example in R-trees [10].

Hierarchical structures can be classified as (height-)balanced or unbalanced. In the balanced structures, the height of every subtree is the same, or at most changes by a fixed amount.

Nodes of AM used in DBMS are stored in disk using registers of fixed size. Storing the nodes in disk is essential to warrant data persistence and to allow handling any number of objects. However, as disk accesses are slow, it is important to keep small the number of disk accesses required to answer queries. Traditional DBMS build indexes only on data holding the total ordering property, so if a tree grows deeper, more disk accesses are required to traverse it. Therefore it is important to keep every tree the shallowest possible. When a tree is allowed to grow unbalanced, it is possible that it degenerates completely, making it useless. Therefore, only balanced trees have been widely used in traditional DBMS.

A metric tree divides a dataset into regions and chooses objects called representatives or centers to represent each region. Each node store the representatives, objects in the covered region, and their distances to the representatives. As the stored objects can be representatives in other nodes, this enables the structure to be organized hierarchically, forming a tree. When a query is performed, the query object is first compared with the representatives of the root node. The triangular inequality is then used to prune subtrees, avoiding distance calculations between the query object and objects or subtrees in the pruned subtrees. Distance calculations between complex objects can have a high computational cost. Therefore, to achieve good performances it is vital to minimize also the number of distance calculations in query operations.

Metric access methods exhibits the node overlapping effect, so the number of disk accesses depends both on the height of the tree and on the amount of overlapping. In this case, it is not worthwhile reducing the number of levels at the expense of increasing the overlapping. Indeed, reducing the number of subtrees that cannot be pruned at each node access can be more important than keep the tree balanced. As more node accesses also requires more distance calculations, increasing the pruning ability of a MAM becomes even more important. However, no published access method took this fact into account so far.

The DBM-tree presented in this paper is a dynamic MAM that relax the usual rule that imposes a rigid height-balancing policy, therefore trading a controlled amount of unbalancing at denser regions of the dataset for a reduced overlap between subtrees. As our experiments show, this tradeoff allows an overall increase in performance when answering similarity queries.

3. Related Works

Plenty of Spatial Access Methods (SAM) were proposed for multidimensional data. An excellent survey showing the evolution of SAM and their main concepts can be found in [9]. However, the majority of them are not able to index data in metric domains, and suffer from the dimensionality curse, that is they are efficient only on low-dimensional datasets.

The techniques of recursive partitioning of data in metric domains proposed by Burkhard and Keller [5] were the start point for the development of MAM. Their first technique divides the dataset choosing one representative for each subset, grouping the remaining elements according to their distances to the representatives. The second technique divides the original set in a fixed number of subsets, selecting one representative for each subset. Each representative and the biggest distance from the representative to all the elements in the subset are stored in the structure to make feasible nearest-neighbor queries.

The MAM proposed by Uhlmann [15] and the VP-tree (Vantage-Point tree) [16] are examples based on the first technique, where the vantage points are the representatives proposed by [5]. Aiming to reduce the number of distance calculations to answer similarity queries in the VP-tree, Baeza-Yates et al. [1] proposed to use the same representative for every node in the same level. The MVP-tree (Multi-Vantage-Point tree) [2, 3] is an extension of the VP-tree, allowing selecting M representatives for each node in the tree. Using many representatives the MVP-tree requires lesser distance calculations to answer similarity queries than the VP-tree. The GH-tree (Generalized Hyper-plane tree) [15] is another method that recursively partitions the dataset in two groups, selecting two representatives and associating the remaining objects to the nearest representative.

The GNAT (Geometric Near-Neighbor Access tree) [4] can be viewed as a refinement of the second technique presented in [5]. It stores the distances between pairs of representatives, and the biggest distance between each stored object to each representative. The tree uses these data to prune distance calculations using the triangular inequality.

All MAM for metric datasets presented so far are static, in the sense that the data structure is built once using the full dataset, and new insertions are not allowed. The M-tree [7] was the first MAM to overcome this deficiency. The M-tree is a height-balanced tree based on the second technique of [5], where the data elements are stored in leaf nodes. The Slim-Tree [12, 13] is an evolution of the M-Tree, embodying the first published technique to reduce the amount of overlap between tree nodes, called the *Slim-Down*. This structure is also a dynamic

MAM, and the *Slim-Down* process leads to a smaller number of disk accesses to answer similarity queries. Although dynamic, neither the Slim-tree nor the M-tree have object deletion operations described. In [8] it was proposed to use multiple representatives, called the "omni-foci", to generate a coordinate system of the objects in the dataset. These coordinates can then be indexed using any SAM, ISAM (Indexed Sequential Access Method), or even sequential scanning, generating a family of MAM called the "Omni-family". Two excellent surveys of MAM can be found in [6] and [11].

All these MAM build balanced trees, aiming to minimize the height of the tree but at the expense of a reduced flexibility to allow minimizing the overlap between nodes storing objects in a high-density region. In this paper we propose a new MAM that minimize the overlapping between nodes storing objects from a high-density region, relaxing the need to keep the tree rigidly balanced, whereas controlling it as a tradeoff between balancing and overlapping. According to the best of the authors' knowledge, no other published work has proposed such tradeoff before.

4. The MAM DBM-tree

The DBM-tree is a dynamic MAM that grows bottom-up. The objects of the dataset are grouped into fixed size disk pages, each page corresponding to a tree node. An object can be stored at any level of the tree. Its main intent is to organize the objects in a hierarchical structure using a representative as the center of each minimum bounding region that covers the objects in a subtree. An object can be assigned to a node if the covering radius of the representative covers it.

Unlike the Slim-tree and M-tree, there is only one type of node in the DBM-tree. There are no distinctions between leaf and index nodes. Each node has a capacity to hold up to C entries. It stores a field C_{eff} counting how many entries s_i are effectively stored in the node and the entries, which can be either a subtree or a single object. A node can have subtree entries, single object entries, or both. Single objects cannot be covered by any of the subtrees stored in the same node. Each node has one of its entries elected to be a representative. The representative of a node is copied to its immediate parent node, unless it is already the root node. Entries storing subtrees have: one representative object s_i that is the representative of the i-th subtree, the distance between the node representative and the representative of the subtree $d(s_{rep}, s_i)$, the link Ptr_i pointing to the node storing that subtree and the covering radius of the subtree R_i. Entries storing single objects have: the single object s_j, the identifier of this object OId_j and the distance between the object representative and the object $d(s_{rep}, s_j)$. This structure can be represented as:

Node [C_{eff}, array [1..C_{eff}] of |< s_i, $d(s_{rep}, s_i)$, *Ptr*$_i$, R_i > or < s_j, *OId*$_j$, $d(s_{rep}, s_j)$>|]

4.1. Building the DBM-tree

The DBM-tree is a dynamic structure, allowing to insert new objects at any time after its creation. When the DBM-tree is asked to insert a new object, it searches the structure for one node qualified to store it. The *Insert* algorithm is shown as Algorithm 1. It starts searching in the root node and proceeds searching recursively for a node that qualifies (that is, the one most appropriated) to store the new object. The insertion of the new object can occur at any level of the structure. In each node, the *Insert* algorithm uses the *ChooseSubtree* algorithm (line 1), which returns the subtree that better qualifies to have the new object stored. If there is no

Algorithm 1 *Insert*

```
Require: $Ptr_t$: pointer to the subtree where the new object $s_n$ will be inserted.
         $s_n$: the object to be inserted.
Ensure: Insert object $s_n$ in the $Ptr_t$ subtree.
 1: $ChooseSubtree(Ptr_t, s_n)$
 2: if There is a subtree that qualifies then
 3:   $Insert(Ptr_i, s_n)$
 4:   if There is a promotion then
 5:     Update the new representatives and their information.
 6:     Insert the object set not covered for node split in the current node.
 7:     for Each entry $s_i$ now covered by the update do
 8:       Demote entry $s_i$.
 9: else if There is space in current node $Ptr_t$ to insert $s_n$ then Insert the new object $s_n$ in node $Ptr_t$.
10:   else $SplitNode(Ptr_t, s_n)$
```

subtree that qualifies, the new object is inserted in the current node (line 9). A qualifying node is one with at least one subtree that covers the new object. The DBM-tree provides two options for the *ChooseSubtree* algorithm:

- **Minimum distance that covers the new object** (*minDist*): among the subtrees that cover the new object, choose the one that has the smallest distance between the representative and the new object. If there is not an entry that qualifies to insert the new object, it is inserted in the current node;
- **Minimum growing distance** (*minGDist*): similar to *minDist* but if there is no subtree that covers the new object, it is chosen the one whose representative is the closest to the new object, increasing the covering radius accordingly. Therefore, the radius of one subtree is increased only when no other subtree covers the new object.

The option chosen by the *ChooseSubtree* algorithm has a high impact in the resultant tree. The *minDist* option tends to build trees with small covering radii, but the trees can grow higher than the trees built with the *minGDist* option. The *minGDist* option tends to produce shallower trees than those produced with the *minDist* option, but with higher overlapping between the subtrees.

If the node chosen by the *Insert* algorithm has no free space to store the new object, then all the existing entries together with the new object taken as a single object must be redistribute between one or two nodes, depending on the redistribution option set in the *SplitNode* algorithm (line 10). The *SplitNode* algorithm deletes the node Ptr_t and remove its representative from its parent node. Its former entries are then redistributed between one or more new nodes, and the representatives of the new nodes together with the set of entries of the former node Ptr_t not covered by the new nodes are promoted and inserted in the parent node (line 6). Notice that the set of entries of the former node that are not covered by any new node can be empty. The DBM-tree has two options to choose the representatives of the new nodes in the *SplitNode* algorithm:

- **Minimum of the largest radii** (*minMax*): this option distributes the entries into at most two nodes, allowing a possibly null set of entries not covered by these two nodes. To select the representatives of each new node, each pair of entries is considered as candidate. For each pair, this algorithm tries to insert each remaining entry into the node having the representative closest to it. The final representatives will be the ones that generated a pair of radii where the largest radius of the pair is the smallest among all possible pairs. The computational complexity of this algorithm is ***O***(C^3), where C is the number of entries to be distribute between the nodes;

- **Minimum radii sum** (*minSum*): this option is similar to the *minMax*, but the two representatives selected is the pair with the smallest sum of the two covering radii.

The minimum node occupation is set when the structure is created, and this value must be between one and half of the node capacity C. If the minimum occupation is set to be half of the node capacity, all the C entries must be distributed between the two new nodes created by the *SplitNode* algorithm. After defining the representative of each new node, the remaining entries are inserted in the node with the closest representative. After distributing every entry, if one of the two nodes stores only its representative, then this node is destroyed and its sole entry is inserted in its parent node. Based on the experiments and in the literature [7], splits leading to an uneven number of entries in the nodes can be better than splits with equal number of entries in each node, because it tends to minimize the overlapping of regions between nodes.

If the minimum occupation is set to a value lower than half of the node capacity, each node is first filled with this minimum number of entries. After this, the remaining entries will be inserted into the node only if its covering radius does not increase the overlapping regions between the two. The rest of entries, that were not inserted into these two nodes, are inserted in the parent node.

Splittings promote the representative to the parent node, which in turn can cause other splittings. After the split propagation (promotion - line 4) or the update of the representative radii (line 5), it can occur that former uncovered single object entries are now covered by the updated subtree. In this case each of these entries is removed from the current node and reinserted into the subtree that covers it (demotion in lines 7 and 8).

4.2. Performing Similarity Queries in the DBM-tree

The DBM-tree can answer the two main types of similarity queries: Range query (Rq) and k-Nearest Neighbor query ($kNNq$). Their algorithms are similar to those of the Slim-tree and the M-tree.

The $Rq()$ algorithm for the DBM-tree is described in Algorithm 2. It receives as input parameters a tree node Ptr_t, the query center s_q and the query radius r_q. All entries in Ptr_t are checked against the search condition (line 2). The triangular inequality allows pruning subtrees and single objects that do not intersect the region defined by the query. Those entries that can not be pruned in this way have their distance to the query object (line 3) calculated. Each entry covered by the query (line 4) is now processed. If it is a subtree, it will be recursively analyzed by the Rq algorithm (line 5). If the entry is an object, then it is added to the answer set (line 6). The end of the process returns the answer set including every object that satisfies the query criteria.

The $kNNq()$ algorithm is similar to the $Rq()$, but it requires a dynamic radius r_k to perform the pruning. In the beginning of the process, this radius is set to a value that covers all the indexed objects. It is adjusted when the answer set is first filled with k objects, or when the answer set is changed thereafter. Another difference is that there is a priority queue to hold not checked entries from the nodes. Entries are checked processing the single objects first and then the subtrees. Among the subtrees, those closer to the query object are checked first. When an object closer than the k already found is located, it substitutes the previous farthest one and the dynamic radius is adjusted (diminished) to ensure tighter pruning.

Algorithm 2 *Rq*()

```
Require: $Ptr_t$ tree to be perform the search, the query object $s_q$ and the query radius $r_q$.
Ensure: Answer set with all objects satisfying the query conditions.
1: for Each $s_i \in Ptr_t$ do
2:   if $|d(s_{rep}, s_q) - d(s_{rep}, s_i)| \leq r_q + R_i$ then
3:     Calculate $dist = d(s_i, s_q)$
4:     if $dist \leq r_q + R_i$ then
5:       if $s_i$ is a subtree then $Rq(Ptr_i, s_q, r_q)$
6:       else Insert $s_i$ in the answer set.
7:     end if
8:   end if
9: end for
```

4.3. The *Shrink* Optimization Algorithm

A special algorithm to optimize loaded DBM-trees was created and called *Shrink*. This algorithm aims at shrinking the nodes by exchanging entries between nodes to reduce the amount of overlapping between subtrees. Reducing overlaps improves the structure, which results in decreasing of the number of distance calculations, total processing time and, mainly, the number of disk accesses required to answer both Rq and $kNNq$ queries. During the exchanging of entries between nodes, some nodes can retain just one entry, so they are promoted and the empty node is deleted from the structure, further improving the performance of the search operations over the tree.

The *Shrink* algorithm can be called at any time during the evolution of a tree, as for example, after the insertion of many new objects. This algorithm is described in Algorithm 3.

The algorithm is applied in every node of a DBM-tree. The input parameter is the Ptr_t to the subtree to be optimized, and the result is the optimized subtree. The stop condition (line 1) for this algorithm occurs in two cases: when there is no entry exchange in the previous iteration or when the number of exchanges already done is larger than 3 times the number of entries in the node. This latter condition assures that no cyclic exchanges can lead to a dead loop. Anyway, it was experimentally verified that a larger number of exchanges does not improve the results. For each entry A in node Ptr_t (line 2), the farthest entry from the node representative is set as i (line 3). Then search another entry in Ptr_t node that can store the entry i (line 5). If such a node exists, remove i from A and reinsert it in this node (line 6). If the exchange makes node A empty, it is deleted, as well as its entry in node Ptr_t (line 9). If this does not generate an empty node, it is only needed to update the reduced covering radius of entry A in node Ptr_t (line 10). After every entry in Ptr_r has been verified, if a node now holds only one entry, this single entry replaces the entry A in node Ptr_t and node A is deleted (line 14).

5. Experimental Evaluation of the DBM-tree

The performance evaluation of the DBM-tree was done with a large assortment of real and synthetic datasets with varying properties that affects the behavior of a MAM. Among these properties are the intrinsic dimensionality of the dataset, the dataset size and the distribution of the data in the metric space. Table 1 presents some illustrative datasets used to evaluate the DBM-tree performance. The dataset name is indicated together with its total number of objects

Algorithm 3 *Shrink*

```
Require: $Ptr_t$ tree to optimize.
Ensure: $Ptr_t$ tree optimized.
1: while The number of exchanges does not exceed 3 times the number of entries in $Ptr_t$ node or no exchanges
   occurred the previous iteration do
2:   for Each subtree entry $A$ in node $Ptr_t$ do
3:     Set entry $i$ from $A$ as the farthest from the $A$ representative.
4:     for Each entry $B$ distinct from $A$ in $Ptr_t$ do
5:       if The entry $i$ of $A$ is covered by node $B$ and this node has enough space to store $i$ then
6:         Remove the entry $i$ from $A$ and reinsert it in $B$.
7:       end if
8:     end for
9:     if node $A$ is empty then delete node $A$ and delete the entry $A$ from $Ptr_t$.
10:    else Update the radius of entry $A$ in $Ptr_t$.
11:  end for
12: end while
13: for Each $A$ subtree in node $Ptr_t$ do
14:   if node $A$ has only one entry then Delete node $A$ and update the entry $A$ in $Ptr_t$.
15: end for
```

(# Objs.), the embedding dimensionality of the dataset (*D*), the page size in KBytes (*Pg*), the metric used ($d()$), and the composition and source description of each dataset.

Table 1: Description of the synthetic and real-world datasets used in the experiments.

Name	# Objs.	*D*	*Pg* (KBytes)	*d*()	Description
ColorHisto	68,040	32	8	L_2	Color image histograms from the KDD repository of the University of California at Irvine (http://kdd.ics.uci.edu). The metric L_2 returns the distance between two objects in a 32-d Euclidean space.
MedHisto	4,247	-	4	L_M	Metric histograms of medical gray-level images. This dataset is adimensional and was generated at GBDI-ICMC-USP. For more details on this dataset and the metric used see [14].
Synt16D	10,000	16	8	L_2	Synthetic vector data with Gaussian distribution with 10 clusters in a 16-d unit hypercube. The process to generate this dataset is described in [7].
Synt256D	20,000	256	32	L_2	Similar to *Synt16D*, but this is a 256-d unit hypercube.
Cities	5,507	2	1	L_2	Geographical coordinates of the Brazilian cities (www.ibge.gov.br).

The computer used for the tests is an Intel Pentium 4 1.6GHz processor with 512 MB of RAM and 40 GB of disk space, running Microsoft Windows 2000. The DBM-tree, the Slim-tree and the M-tree algorithms were implemented using the C++ language into the Arboretum MAM library (www.gbdi.icmc.usp.br/arboretum), all with the same code optimization, to obtain a fairly comparison. The DBM-tree was compared with Slim-tree and M-tree, that are the most known and used dynamics MAM.

The Slim-tree and the M-tree were configured using their best recommended setup. They are: *minDist* for the *ChooseSubtree* algorithm, *minMax* for the split algorithm and the minimal occupation set to 25% of node capacity. The results for the Slim-tree were measured after the execution of the *Slim-Down* optimization algorithm.

We tested the DBM-tree considering three distinct configurations, to evaluate its available options. The tested configurations are the following:

- *DBM-MM*: *minDist* for the *ChooseSubtree* algorithm, *minMax* for the *SplitNode* algorithm and minimal occupation set to 30% of node capacity,
- *DBM-MS*: equal to *DBM-MM*, except to the option *minSum* for the *SplitNode* algorithm and
- *DBM-GMM*: *minGDist* for *ChooseSubtree*, *minMax* for *SplitNode* and minimal occupation set to 50% of node capacity.

All measurements were performed after the execution of the *Shrink* algorithm.

From each dataset it was extracted 500 objects to be used as query centers. They were chosen randomly from the dataset, and half of them (250) were removed from the dataset before creating the trees, and the other half were copied to the query set, but maintained in the set of objects inserted in the trees. Hence, half of the query set belongs to the dataset indexed by the MAM and the other half does not, allowing to evaluate queries with centers indexed or not. Each dataset were used to build one tree of each type, and every tree was built inserting one object at a time, calculating the average number of distance calculations, average number of disk accesses and total processing time (in seconds). In the query measurements, each point corresponds to performing 500 queries with the same parameters but varying query centers. The number k for the $kNNq$ queries varied from 2 to 20 for each measurement, and the radius varied from 0.01% to 10% of the largest distance between pairs of objects in the dataset, because they are the most meaningful range of parameters asked when performing similarity queries. The Rq graphics are in log scale for the radius abscissa, to enphasize the relevant part of the graph.

The building time, maximum height and the distribution of objects in the structure were measured for every tree. The building time of the 5 trees were similar for each dataset. It is interesting to compare the maximum height of the DBM-tree options and the balanced trees, so they are summarized in Table 2.

Table 2: Maximum height of the tree for each dataset tested.

Name	Slim-tree	M-tree	*DBM-MM*	*DBM-MS*	*DBM-GMM*
ColorHisto	4	4	10	10	4
MedHisto	4	4	9	11	5
Synt16D	3	3	7	7	3
Synt256D	4	4	17	17	5
Cities	4	4	7	7	4

The maximum height for the *DBM-MM* and the *DBM-MS* trees were bigger than the balanced trees in every dataset. The biggest difference was in the *Synt256D*, with height of 17 as compared to 4 for the Slim-tree and the M-tree. However, as the other experiments show, this does not increase the number of disk accesses. In fact, those DBM-trees did, in average, less disk accesses than the Slim-tree and M-tree, as is shown in the next subsection.

For the *DBM-GMM* trees, although it does not force the balance, the maximum height in the majority of trees was equal to those of the Slim-tree and M-tree. This is an interesting result and indicates that the balancing is not so important for MAM as it is for the conventional structures.

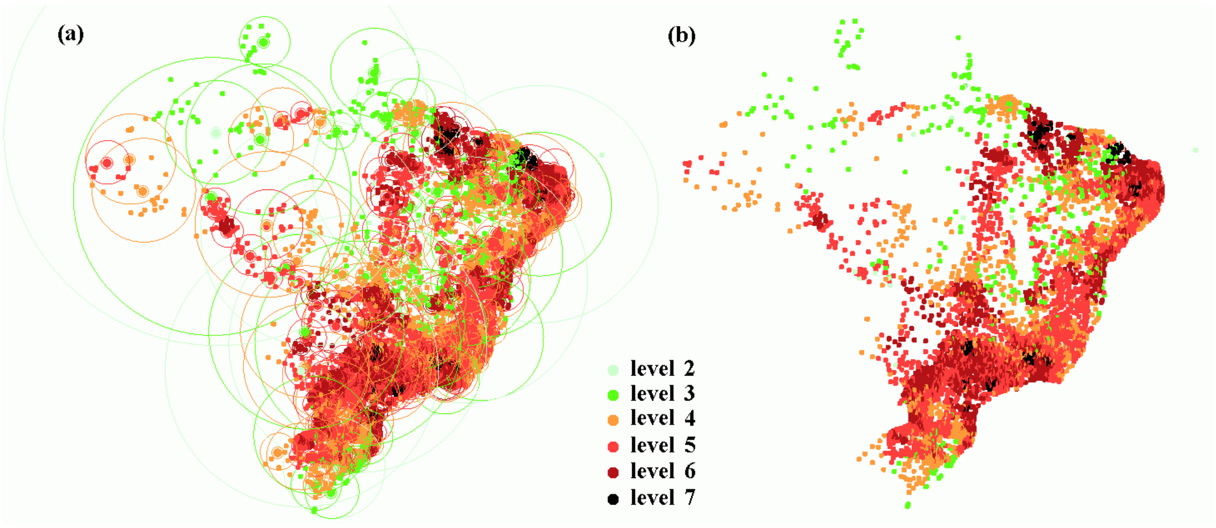

Figure 1: Visualization of the *DBM-MM* structure for the *Cities* dataset. (a) with the covering radius of the nodes; and (b) only the objects. It is possible to verify that the structure is deeper (darker objects) in high-density regions, and shallower (lighter objects) in low-density regions.

The data distribution in the levels of a DBM-tree are shown using the *Cities* dataset. This is possible because this dataset is in a bi-dimensional Euclidean space. Figure 1 shows the objects indexed in the *DBM-MM* with each different color representing objects at different levels. Figure 1(a) shows the objects and the covering radius of each node, and Figure 1(b) shows only the objects. The objects with darker colors are in a deeper level than those with lighter colors. The figure shows that the depth of the tree is larger in higher density regions and that objects are stored in every level of the structure, as is expected. This figure shows visually that the depth of the tree is smaller in low density regions, and that the number of objects at the deepest levels is small, even in the high-density regions.

5.1. Performance Comparison between the DBM-tree and the Slim-tree and M-tree

We have used many synthetic and real datasets to evaluate the DBM-tree. We now present the results obtained when comparing the DBM-tree with the best setup of the Slim-tree and M-tree. Due to space limitations we only present the results from four meaningful datasets (*ColorHisto*, *MedHisto*, *Synt16D* and *Synt256D*), which are high-dimensional and non-dimensional (metric) datasets, and gives a fairly sample of what happened. The main motivation in these experiments is evaluating the DBM-tree performance with its best competitors with respect to the 2 main similarity queries: Rq and $kNNq$.

Figure 2 shows the measurements to answer Rq and $kNNq$ on these 4 datasets. The graphs on the first row (Figures 2(a), (b), (c) and (d)) show the average number of distance calculations. It is possible to note in these graphs that every DBM-tree executed in average a smaller number of distance calculations than Slim-tree and M-tree. Among all, the *DBM-MS* presented the best result for almost every dataset, losing only at the *Synt256D* dataset to *DBM-GMM*. No DBM-tree executed more distance calculations than the Slim-tree or the M-tree, for every dataset. The graphs also show that the DBM-tree reduces the average number of distance calculations up to 67% for Rq (graph (c)) and up to 35% for $kNNq$ (graph (d)), when compared to the Slim-tree, which is the best balanced tree in every dataset with respect to distance calculations. When compared to M-tree, the DBM-tree reduced up to 72% for Rq (graph (c)) and up to 41% for $kNNq$ (graph (d)).

The graphs of the second row (Figures 2(e), (f), (g) and (h)) show the average number of disk accesses for both Rq and $kNNq$ queries. In every measurement the DBM-trees clearly outperformed the Slim-tree and the M-tree, with respect to number of disk accesses. The graphs show that the DBM-tree reduces the average number of disk accesses up to 43% for Rq (graph (g)) and up to 35% for $kNNq$ (graph (h)), when compared to the Slim-tree. It is important to note that the Slim-tree is the MAM that in general requires the lowest number of disk accesses between every previous published MAM. These measurements were taken after the execution of the *Slim-Down* algorithm in the Slim-tree. When compared to the M-tree, the gain is even greater, increasing to up to 54% for Rq (graph (g)) and up to 42% for $kNNq$ (graph (h)).

An important observation is that the immediate result of decreasing overlap between nodes of a tree is the reduced number of distance calculations. However, the number of disk accesses in a MAM is also related to the overlapping between subtrees. An immediate consequence of this fact is that decreasing the overlap reduces both the number of distance calculations and of disk accesses, to answer both types of similarity queries. These two benefits contribute to reduce the total processing time of queries.

The graphs of the third row (Figures 2(i), (j), (k) and (l)) show the total processing time (in seconds). As the three DBM-trees performed lesser distance calculations and disk accesses than both Slim-tree and M-tree, they are naturally faster to answer both Rq and $kNNq$. The importance of comparing query time is that it reflects the total complexity of the algorithms besides the number of distance calculations and the number of disk accesses. The graphs shows that the DBM-tree is up to 44% faster to answer Rq and $kNNq$ (graphs (k) and (l)) than Slim-tree. When compared to the M-tree, the reducion in total query time is greater, going to be up to 50% for Rq and $kNNq$ queries (graphs (k) and (l)).

5.2. Experiments regarding the *Shrink* Algorithm

The tests performed to measure the improvement achieved by the *Shrink* algorithm in the three DBM-trees with all datasets shown in Table 1. As the results of all the datasets were similar, we show in Figure 3 only the results for the number of disk accesses with the *ColorHisto* (Figures 3(a) for Rq and (b) for $kNNq$) and *Synt16D* dataset (Figures 3(c) for Rq and (d) for $kNNq$).

Figure 3 compares the query performance before and after the execution of the *Shrink* algorithm for *DBM-MM*, *DBM-MS* and *DBM-GMM* for both Rq and $kNNq$. Every graph shows that the *Shrink* algorithm improves the final trees. The most expressive result is the *DBM-GMM* indexing the *Synt16D*, which achieved up to 30% lesser disk accesses for $kNNq$ and Rq as compared with the same structure not optimized.

5.3. Scalability of the DBM-tree

This experiment evaluated the behavior of the DBM-tree with respect to the number of elements in the dataset. To do so, we generated 10 datasets similar to the *Synt16D*, each one with 10,000 elements. We inserted all 10 datasets in the same tree, totaling 100,000 elements. After inserting each dataset we run sets of queries, executing 500 similarity queries for each point in the graph, as before. The behavior was equivalent for different values of k and radius, thus we

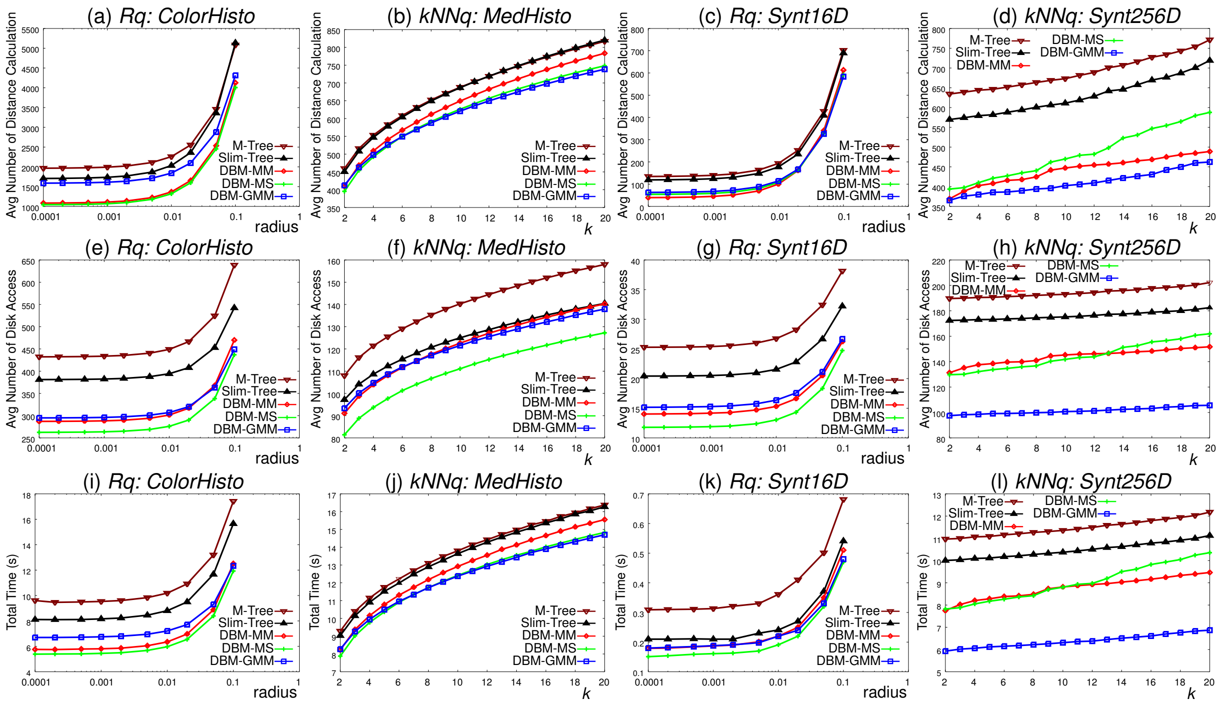

Figure 2: Comparison of the average number of distance calculations (first row), average number of disk accesses (second row) and total processing time (in *s*) (third row) of DBM-tree, Slim-tree and M-tree, for Rq and $kNNq$ queries for the *ColorHisto* ((a), (e) and (i) - Rq), *MedHisto* ((b), (f) and (j) - $kNNq$), *Synt16D* ((c), (g) and (k) - Rq) and *Synt256D* ((d), (h) and (l) - $kNNq$) datasets.

present only the results for k=10 and radius=0.1%. As the total processing time embodies the results of the other measurements, we show only the total processing time for range queries in Figure 4.

This figure presents the behavior of the three DBM-tree considering: the average number of distance calculations (a), the average number of disk accesses (b), the total processing time for $kNNq$ (c) and for Rq (d). As it can be seen, the three DBM-trees exhibits sub-linear behavior when the number of elements indexed grows, what makes the method adequate to index very large datasets, in any of its configurations.

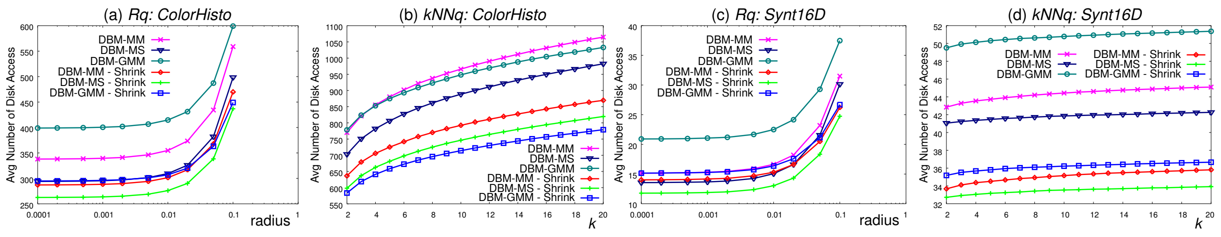

Figure 3: Average number of disk accesses to perform Rq and $kNNq$ queries in the DBM-tree before and after the execution of the *Shrink* algorithm: (a) Rq on *ColorHisto*, (b) $kNNq$ on *ColorHisto*, (c) Rq on *Synt16D*, (d) $kNNq$ on *Synt16D*.

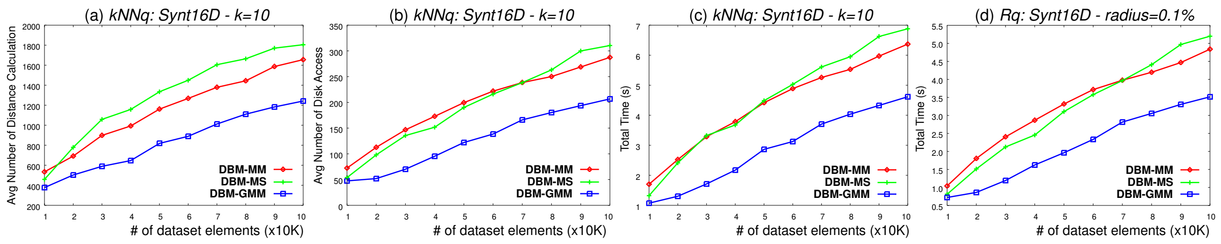

Figure 4: Scalability of DBM-tree regarding the dataset size indexed during $kNNq$, measuring the average number of distance calculations (a), the average number of disk accesses (b) and the total processing time (c). The total processing time for Rq is shown in (d). The dataset indexed was the *Synt16D* with 100,000 objects.

6. Conclusions and Future Works

This paper presents a new dynamic MAM called *DBM-tree* (*Density-Based Metric tree*), which in a controlled way relax the height-balancing requirement of access methods, trading a controlled amount of unbalancing at denser regions of the dataset for a reduced overlap between subtrees. This is the first dynamic MAM that makes possible to reduce the overlap between nodes relaxing the rigid balancing of the structure. The height of the tree is higher in denser regions, in order to keep a tradeoff between breadth-searching and depth-searching. The options to define how to construct a tree and the optimizations possibilities in DBM-tree are larger than in rigid balanced trees, because it is possible to adjust the tree according to the data distributions at different regions of the data space. Therefore, this paper also presented a new optimization algorithm, called *Shrink*, which improves the performance in trees reorganizing the elements among their nodes.

The experiments performed over synthetic and real datasets showed that the *DBM-tree* outperforms the main balanced structures existing so far: the Slim-tree and the M-tree. In average, it is up to 50% faster than the traditional MAM and reduces the number of required distance calculations to up to 72% when answering similarity queries. The DBM-tree spends fewer disk accesses than the the Slim-tree, that until now was the most efficient MAM with respect to the number of disk accesses. The DBM-tree requires up to 54% fewer disk accesses than the balanced trees. After performed the *Shrink* algorithm, its performance achieves improvements up to 30% for range and k-nearest neighbor queries considering disk accesses. It was also shown that the DBM-tree scales up very well with respect to the number of elements indexed, presenting sub-linear behavior, which makes it well-suited to very large datasets.

Among the future works, we intend to develop a bulk-loading algorithm for the DBM-tree. As the construction possibilities of the DBM-tree is larger than those of the balanced structures, a bulk-loading algorithm can employ strategies that can achieve better performance than is possible in other trees. Other future work is to develop an object-deletion algorithm that can really remove objects from the tree. All existing rigidly balanced MAM such as the Slim-tree and the M-tree, cannot effectively delete objects being used as representatives, so they are just marked as removed, without releasing the space occupied, so it remains being used in the comparisons required in the search operations. The organizational structure of the DBM-tree enables the effective deletion of objects, making it a completely dynamic MAM.

References

[1] Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 807 of *Lecture Notes in Computer Science (LNCS)*, pages 198–212, Asilomar, USA, 1994. Springer Verlag.

[2] Tolga Bozkaya and Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 357–368, 1997.

[3] Tolga Bozkaya and Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404, Sep 1999.

[4] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 574–584, Zurich, Switzerland, 1995. Morgan Kaufmann.

[5] Walter A. Burkhard and Robert M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, Apr 1973.

[6] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys (CSUR)*, 33(3):273–321, Sep 2001.

[7] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 426–435, Athens, Greece, 1997. Morgan Kaufmann.

[8] Roberto F. Santos Filho, Agma J. M. Traina, Caetano Traina Jr., and Christos Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *IEEE International Conference on Data Engineering (ICDE)*, pages 623–630, Heidelberg, Germany, 2001.

[9] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, Jun 1998.

[10] A. Guttman. R-tree : A dynamic index structure for spatial searching. In *ACM International Conference on Data Management (SIGMOD)*, pages 47–57, Boston, USA, 1984.

[11] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 28(4):517–580, Dec 2003.

[12] Caetano Traina Jr., Agma J. M. Traina, Christos Faloutsos, and Bernhard Seeger. Fast indexing and visualization of metric datasets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(2):244–260, Mar/Apr 2002.

[13] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *International Conference on Extending Database Technology (EDBT)*, volume 1777 of *Lecture Notes in Computer Science (LNCS)*, pages 51–65, Konstanz, Germany, 2000. Springer.

[14] Agma J. M. Traina, Caetano Traina Jr., Josiane M. Bueno, and Paulo M. de A. Marques. The metric histogram: A new and efficient approach for content-based image retrieval. In *IFIP Working Conference on Visual Database Systems (VDB)*, Brisbane, Australia, 2002.

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

[16] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321, Austin, USA, 1993.