DBM-Tree: Trading Height-Balancing for Performance in Metric Access Methods*

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

ICMC – Institute of Mathematics and Computer Sciences
USP – University of Sao Paulo at Sao Carlos
Avenida do Trabalhador Sao-Carlense, 400
CEP 13560-970 – Sao 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, such as the Slim-tree and the M-tree, improve the query performance minimizing the number of disk accesses, keeping a constant height of the structures stored on disks (height-balanced trees). 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. An underpinning for cost estimation on tree structures is their height, so we show a non-height dependable cost model that can be applied for DBM-tree. 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 world 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 66%. After performing the* Shrink *algorithm, the performance improves up to 40% regarding the number of disk accesses for range and k-nearest neighbor queries. In addition, the* DBM-tree *scales up well, exhibiting linear performance with growing number of elements in the database.*

Keywords: Metric Access Method, Metric Tree, Indexing, Similarity Queries.

*This work has been supported by **FAPESP** (São Paulo State Research Foundation) under grants 01/11987-3, 01/12536-5 and 02/07318-1 and by **CNPq** (Brazilian National Council for Supporting Research) under grants 52.1685/98-6, 860.068/00-7, 50.0780/2003-0 and 35.0852/94-4.

1. Introduction

The volume of data managed by the Database Management Systems (DBMS) is continually increasing. 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 $\neq$) and relational ordering predicates ($\leq$, $<$, $\geq$ and $>$), 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 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 ordering 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.

A MAM such as the Slim-tree [15, 14] and the M-tree

[9] were developed to answer similarity queries based on the similarity relationships among pairs of objects. The similarity (or dissimilarity) 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:

1. **symmetry**: $d(s_1, s_2) = d(s_2, s_1)$;
2. **non-negativity**: $0 < d(s_1, s_2) < \infty$ if $s_1 \neq s_2$ and $d(s_1, s_1) = 0$; and
3. **triangular inequality**: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$,

$\forall s_1, s_2, s_3 \in \mathbb{S}$. A metric dataset $S \subset \mathbb{S}$ is a set of objects $s_i \in \mathbb{S}$ currently stored in a database. Vectorial data with a 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 $s_i \in S$, such that $d(s_i, s_q) \leq r_q$. An example is: "Select the proteins that are similar to the protein P by up to 5 purine bases", which 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 in S 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, which 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 in high-density regions relaxing the height-balance of the structure. 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, the DBM-tree presents better performance 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 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 66% 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 40% in number of disk accesses to answer range and k-nearest neighbor queries. Moreover, the DBM-tree is scalable, exhibiting linear behavior in the total processing time, the number of disk accesses and the number of distance calculations regarding the number of indexed elements.

A preliminary version of this paper was presented at SBBD 2004 [20]. Here, we show a new split algorithm for the DBM-tree. Additionally, this paper shows an accurate cost function for the DBM-tree using only information easily derivable from the tree, thus providing a cost function that does not depend upon a constant tree-height. A cost function is fundamental to enable the DBM-tree to be employed in real DBMS. Every tree-based AM used in existing DBMS uses the height of the tree as the main parameter to optimize a query plan. As the DBM-tree does not have a reference height, every existing theory about query plan optimizations are knocked out when using a DBM-tree. Therefore, the cost function presented in this paper is a fundamental requirement to enable using DBM-trees in a real DBMS.

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

Access Methods (AM) are used by DBMS to improve performance on retrieval operations. The use of meaningful properties from the objects indexed is fundamental to achieve the improvements. Using 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 larger 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. Thus, when a search for a given number is required, comparing this number with a stored one enables discarding further comparisons with the part of the data where the number cannot be in.

An important class of AM are the 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 objects in the root

node, determining which subtrees need to be traversed, recursively repeating this process for each subtree that is able to store answers.

Notice 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 have only 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 [12].

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.

The nodes of an AM used in a DBMS are stored in disk using fixed size registers. 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 the number of disk accesses required to answer queries small.

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 stores the representatives, the 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, resulting in 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 performance in metric access methods, 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. A comprehensive survey showing the evolution of SAM and their main concepts can be found in [11]. However, the majority of them cannot index data in metric domains, and suffer from the dimensionality curse, being efficient to index only low-dimensional datasets.

An unbalanced R-tree called CUR-tree (Cost-Based Unbalanced R-tree) was proposed in [16] to optimize query executions. It uses promotion and demotion to move data objects and subtrees around the tree taking into account a given query distribution and a cost model for their execution. The tree is shallower where the most frequent queries are posed, but it needs to be reorganized every time a query is executed. This technique works only in SAM, making it infeasible to MAM.

Considering cost models, a great deal of work were also published regarding SAM [17]. However they rely on data distribution in the space and other spatial properties, what turns them infeasible for MAM.

The techniques of recursive partitioning of data in metric domains proposed by Burkhard and Keller [5] were the starting point for the development of MAM. The 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 elements in the subset are stored in the structure to improve nearest-neighbor queries.

The MAM proposed by Uhlmann [19] and the VP-tree (Vantage-Point tree) [21] 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 to select 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) [19] 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 discussed so far are static, in the sense that the data structure is built at once using the full dataset, and new insertions are not allowed afterward. Furthermore, they only attempt to reduce the number of distance calculations, paying no attention on disk accesses. The M-tree [9] was the first MAM to overcome this deficiency. The M-tree is a height-balanced tree based on the second technique of [5], with the data elements stored in leaf nodes.

A cost model based only in the distance distributions of the dataset and information of the M-tree nodes is provided in [8].

The Slim-Tree [15] is an evolution from the M-Tree, embodying the first published method to reduce the amount of node overlapping, called the *Slim-Down*.

The use of multiple representatives called, "omni-foci", was proposed in [10] to generate a coordinate system of the objects in the dataset. The coordinates can be indexed using any SAM, ISAM (Indexed Sequential Access Method), or even sequential scanning, generating a family of MAM called the "Omni-family". Two good surveys on MAM can be found in [7] and [13].

The MAM described so far build height-balanced trees aiming to minimize the tree height at the expense of little flexibility to reduce node overlap. The DBM-tree proposed in this paper is the first MAM which keep a tradeoff between breadth-searching and depth-searching to allows trading height-balancing with overlap reduction, to achieve better overall search performance.

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 object as the center of each minimum bounding region that covers the objects in a subtree. An object can be stored in a node if the covering radius of the representative covers it.

Unlike the Slim-tree and the 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, and it stores a field C_{eff} to count how many entries s_i are effectively stored in that node. An entry can be either a single object or a subtree. 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. If a subtree is elected, the representative is the center of the root node of the subtree. 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:

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

In this structure, the entry s_i whose $d(s_{rep}, s_i) = 0$ holds the representative object s_{rep}.

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. A qualifying node is one with at least one subtree that covers the new object. The $Insert()$ algorithm is shown as Algorithm 1. It starts searching in the root node and proceeds searching recursively for a node that qualifies 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 subtree that qualifies, the new object is inserted in the current node (line 9). The DBM-tree provides two policies 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.

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** $SplitNode(Ptr_t, s_n)$

The policy chosen by the $ChooseSubtree()$ algorithm has a high impact on the resulting tree. The *minDist* policy tends to build trees with smaller covering radii, but the trees can grow higher than the trees built with the *minGDist* policy. The *minGDist* policy tends to produce shallower trees than those produced with the *minDist* policy, but with higher overlap 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 redistributed 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 two 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 three 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 option tries to insert each remaining entry into the node having the representative closest to it. The chosen representatives will be those generating the pair of radii whose largest radius is the smallest among all possible pairs. The computational complexity of the algorithm executing this option is $\boldsymbol{O}(C^3)$, 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;

- **2-Clusters**: this option tries to build at most two groups. These groups were built choosing objects that minimizes the distances inside each group, organizing them as a minimal spanning tree. This option is detailed as Algorithm 2. The first step of this algorithm is the creation of C groups, each one of only one entry. The second step is joining each group with its nearest group. This step finishes when only 2 groups remain (line 2). The next step checks if there is a group with only one object then it will be inserted in the upper level (line 4). A representative object is chosen (line 5) for each remaining group, and nodes are created to store their objects (line 6). The representatives and all their information are promoted to the next upper level. Figure 1 illustrates this approach applied to a bi-dimensional vector space. The node to be split is presented in Figure 1(a). After building the C groups (Figure 1(b)), the groups are joined to form 2 groups (Figure 1(c)). Figure 1(d) presets the two resulting nodes after the split by the **2-Clusters** approach.

The minimum node occupation is set when the structure is created, and this value must be between one object and at most half of the node capacity C. If the $ChooseSubTree$ policy is set to *minGDist* then 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 the representative, then this node is destroyed and its sole entry is inserted in its parent node as a single object. Based on the experiments and in the literature [9], splits leading to an unequal number of entries in the nodes can be better than splits with equal number of entries in each node, because it tends to minimize overlap between nodes.

If the $ChooseSubTree$ policy is set to *minDist* and the minimum occupation is set to a value lower than half of the node capacity, then 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 the entries, that were not inserted into the two nodes, are inserted in the parent node.

Algorithm 2 2-$Clusters()$

Require: C entries to be redistributed in nodes.
Ensure: A representative set (*RepSet*) and a entry set to be inserted in the upper level (*PromoSet*).
1: Build C groups.
2: Try to join, one by one, the C groups, until only 2 groups remain.
3: **for** each group that have unique entries. **do**
4: Insert the unique entries in *PromoSet*.
5: **end for**
6: Choose each representative object for each group.
7: Create the nodes for the remaining groups.
8: Insert in *RepsSet* the generated representatives.

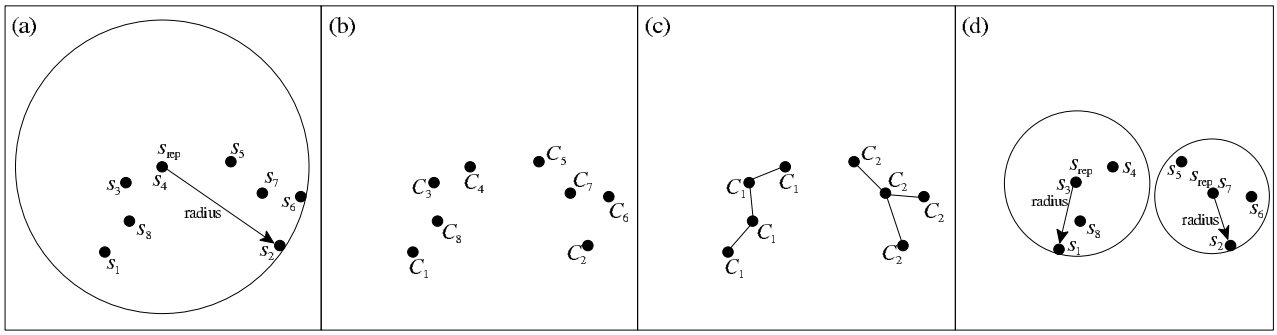

Figure 1. Exemplifying a node split using the **2-**$Clusters()$ algorithm: (a) before the split, (b) forming C groups with unique nodes, (c) 2 final groups, and (d) the final nodes created with the chosen representatives.

Splittings promote the representative to the parent node, which in turn can cause other splittings. After the split propagation in Algorithm 1 (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. 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 as Algorithm 3. 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 pertain to the region defined by the query. The entries that cannot 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.

Algorithm 3 $Rq()$

Require: Ptr_t tree to be perform the search, the query object s_q and the query radius r_q.
Ensure: Answer set $AnswerSet$ with all objects satisfying the query conditions.

```
1: for Each s_i ∈ Ptr_t do
2:     if |d(s_rep, s_q) − d(s_rep, s_i)| ≤ r_q + R_i then
3:         Calculate dist = d(s_i, s_q)
4:         if dist ≤ r_q + R_i then
5:             if s_i is a subtree then Rq(Ptr_i, s_q, r_q)
6:             else AnswerSet.Add(s_i).
7:         end if
8:     end if
9: end for
```

The $kNNq()$ algorithm, described as Algorithm 4, is similar to $Rq()$, 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 (line 1). It is adjusted when the answer set is first filled with k objects, or when the answer set is changed thereafter (line 12). Another difference is that there is a priority queue to hold the not yet checked subtrees from the nodes. Entries are checked processing the single objects first (line 4 to 12) and then the subtrees (line 13 to 18). Among the subtrees, those closer to the query object that intersect the query region are checked first (line 3). When an object closer than the k already found is located (line 8), it substitutes the previous farthest one (line 11) and the dynamic radius is adjusted (diminished) to tight further pruning (line 12).

4.3. The $Shrink()$ optimization Algorithm

A special algorithm to optimize loaded DBM-trees was created, called $Shrink()$. This algorithm aims at shrinking the nodes by exchanging entries between nodes to reduce the amount of overlapping between subtrees. Reducing overlap improves the structure, which results in a decreased number of distance calculations, total processing time and 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.

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 as Algorithm 5.

The algorithm is applied in every node of a DBM-tree. The input parameter is the pointer Ptr_t to the subtree to be optimized, and the result is the optimized subtree. The stop condition (line 1) holds 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

Algorithm 4 $kNNq()()$

Require: root node Ptr_{root}, the query object s_q and number of objects k.
Ensure: Answer set with all objects satisfying the query conditions.

```
 1: r_k = ∞
 2: PriorityQueue.Add(Ptr_root, 0)
 3: while ((Node = PriorityQueue.First()) <= r_k) do
 4:     for each s_i ∈ Node do
 5:         if s_i is a single object then
 6:             if |d(s_rep, s_q) − d(s_rep, s_i)| ≤ r_k then
 7:                 Calculate dist = d(s_i, s_q)
 8:                 if dist ≤ r_k then
 9:                     AnswerSet.Add(s_i).
10:                     if AnswerSet.Elements() ≥ k then
11:                         AnswerSet.Cut(k).
12:                         r_k = AnswerSet.MaxDistance().
13:                     end if
14:                 end if
15:             end if
16:         end if
17:     end for
18:     for each s_i ∈ Node do
19:         if s_i is a subtree then
20:             if |d(s_rep, s_q) − d(s_rep, s_i)| ≤ r_k + R_i then
21:                 Calculate dist = d(s_i, s_q)
22:                 if dist ≤ r_k + R_i then
23:                     PriorityQueue.Add(s_i, dist).
24:                 end if
25:             end if
26:         end if
27:     end for
28: end while
```

Algorithm 5 $Shrink()$

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 s_a in node Ptr_t do
 3:         Set entry i from s_a as the farthest from the s_a representative.
 4:         for Each entry s_b distinct from s_a in Ptr_t do
 5:             if The entry i of s_a is covered by node s_b and this node has enough space to store i then
 6:                 Remove the entry i from s_a and reinsert it in s_b.
 7:             end if
 8:         end for
 9:         if node s_a is empty then delete node s_a and delete the entry s_a from Ptr_t.
10:         else Update the radius of entry s_a in Ptr_t.
11:     end for
12: end while
13: for Each s_a subtree in node Ptr_t do
14:     Shrink(s_a).
15:     if node s_a has only one entry then Delete node s_a and update the entry s_a in Ptr_t.
16: end for
```

assures that no cyclic exchanges can lead to a dead loop. It was experimentally verified that a larger number of exchanges does not improve the results. For each entry s_a in node Ptr_t (line 2), the farthest entry from the node representative is set as i (line 3). Then search another entry s_b in Ptr_t that can store the entry i (line 5). If such a node exists, remove i from s_a and reinsert it in node s_b (line 6). If the exchange makes node s_a empty, it is deleted, as well as its entry in node Ptr_t (line 7). If this does not generate an empty node, it is only needed to update the reduced covering radius of entry s_a in node Ptr_t (line 8). This process is recursively applied over all nodes of the tree (line 9 and 10). After every entry in Ptr_t has been verified, the nodes holding only one entry are deleted and its single entry replaces the node in Ptr_t (line 11).

4.4. A Cost Model for DBM-tree

Cost models for search operations in trees usually rely on the tree height. Such cost models does not apply for the DBM-tree. However, an AM requires a cost model in order to be used in a DBMS. Therefore, we developed a cost model for the DBM-tree, based on statistics of each tree node. The proposed approach does not rely on the data distribution, but rather on the distance distribution among objects. The cost model developed assumes that the expected probability $P()$ of a node Ptr_t to be accessed is equal to the probability of the node radius R_{Ptr_t} plus the query radius r_q be greater or equal to the distance of the node representative s_{rep} of Ptr_t to the query object s_q. The probability of Ptr_t to be accessed can therefore be expressed as:

$$P(Ptr_t) = P(R_{Ptr_t} + r_q \geq d(s_{rep}, s_q)) \qquad (1)$$

We assume that every object has a distribution of distances to the other objects in the dataset, in average, similar to the distribution of the other objects. Thus, Formula (1) can be approximated by a normalized histogram $Hist()$ of the distance distribution instead of computing the distance of the query object to the node representative. Therefore

$$P(Ptr_t) \approx Hist(R_{Ptr_t} + r_q) \qquad (2)$$

where $Hist()$ is an equi-width histogram of the distances among pairs of objects of the dataset.

The histogram can be computed calculating the average number of distances falling at the range defined at each histogram bin, for every object in the dataset, or only for a small unbiased sample of the dataset. Thereafter, to calculate the expected number of disk accesses (DA) for any Rq, it is sufficient to sum the above probabilities over all N nodes of a DBM-tree, as:

$$DA(Rq(s_q, r_q)) = \sum_{i=1}^{N} Hist(R_{Ptr_i} + r_q) \qquad (3)$$

The cost to keep the histogram is low and requires a small amount of main memory to maintain the histogram. Moreover, if it is calculated over a fixed size sample of the database, it is linear on the database size, making it scalable for the database size.

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 embedded 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 with its total number of objects (# Objs.), the embedding dimensionality of the dataset (*E*), the page size in KBytes (*Pg*), and the composition and source description of each dataset. The multidimensional datasets uses the Euclidean distance L_2, and the *MedHisto* dataset uses the metric-histogram M_{histo} distance [18].

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 $ChooseSubtre()$ 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 four 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 using the option *minSum* for the $SplitNode()$ algorithm;
- *DBM-GMM*: *minGDist* for $ChooseSubtree()$, *minMax* for $SplitNode()$;
- *DBM-2CL*: *minGDist* for $ChooseSubtree()$, *2-Clusters* for $SplitNode()$.

All measurements were performed after the execution of the $Shrink()$ algorithm.

The computer used for the experiments was an Intel Pentium III 800MHz processor with 512 MB of RAM and 80 GB of disk space, running the Linux operating system. The DBM-tree, the Slim-tree and the M-tree MAM 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 fair comparison.

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. 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 indexed dataset by the MAM and the other half does not, allowing to evaluate queries with centers indexed or not. However, as the query centers are in fact objects of the original dataset, the set of queries closely follows the queries expected to be posed by a real application. Each dataset was used to build one tree of each type, creating a total of thirty trees. Each tree was built inserting one object at a time, counting the average number of distance calculations, the average number of disk accesses and measuring the total building time (in seconds). In the graphs showing results from the query evaluations, 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 radii asked when performing similarity queries. The Rq graphics are in log scale for the radius abscissa, to emphasize the most relevant part of the graph.

5.1. Evaluating the tree building process

The building time and the maximum height were measured for every tree. The building time of the 6 trees were similar for each dataset. It is interesting to compare the maximum height of the various DBM-tree options and the balanced trees, so they are summarized in Table 2.

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 *ColorHisto*, with achieved a height of 10 levels as compared to only 4 levels for the Slim-tree and the M-tree. However, as the other experiments show, this higher height does not increases 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.

It is worth to note that, although the *DBM-GMM* trees do not force the height-balance, the maximum height in these trees were equal or very close to those of the Slim-tree and the M-tree. This fact is an interesting result that corroborates our claim that the height-balance is not as important for MAM as it is for the overlap-free structures.

The data distribution in the levels of a DBM-tree is shown using the *Cities* dataset. This visualization was generated using the *MAMView* system [6]. The *MAMView* system is a tool to visualize similarity queries and MAM behavior, making it possible to explore metric trees. This is possible because this dataset is in a bi-dimensional Euclidean space. Figure 2 shows the indexed objects in the *DBM-MM* with each color representing objects at different levels. Darker colors indicate objects in deeper levels. Figure 3(a) shows the objects and the covering radius of each node, and Figure 2(b) shows only the objects. 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. It also shows that the number of objects in the deepest levels is small, even in high-density regions.

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

Name	# Objs.	*E*	*Pg*	Description
Cities	5,507	2	1	Geographical coordinates of the Brazilian cities (www.ibge.gov.br).
ColorHisto	68,040	32	8	Color image histograms from the KDD repository of the University of California at Irvine (http://kdd.ics.uci.edu). The metric returns the distance between two objects in a 32-d Euclidean space.
MedHisto	4,247	-	4	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 [18].
Synt16D	10,000	16	8	Synthetic clustered datasets consisting of 16-dimensional vectors normally-distributed (with σ=0.1) in 10 clusters over the unit hypercube. The process to generate this dataset is described in [9].
Synt256D	20,000	256	128	Similar to *Synt16D*, but it is with 20 clusters (with σ=0.001) in a 256-*d* hypercube.

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

Name	*Cities*	*ColorHisto*	*MedHisto*	*Synt16D*	*Synt256D*
M-tree	4	4	4	3	3
Slim-tree	4	4	4	3	3
DBM-MM	7	10	9	6	7
DBM-MS	7	10	11	6	6
DBM-GMM	4	4	5	3	3
DBM-2CL	4	4	5	3	4

5.2. Performance of query execution

We present the results obtained comparing the DBM-tree with the best setup of the Slim-tree and the M-tree. In this paper we present the results from four meaningful datasets (*ColorHisto*, *MedHisto*, *Synt16D* and *Synt256D*), which are or high-dimensional or non-dimensional (metric) datasets, and gives a fair 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 query types: range Rq and k-nearest neighbors $kNNq$.

Figure 4 shows the measurements to answer Rq and $kNNq$ on these 4 datasets. The graphs on the first column (Figures 4(a), (d), (g) and (j)) show the average number of distance calculations. It is possible to note in the 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. No DBM-tree executed more distance calculations than the Slim-tree or the M-tree, for any dataset. The graphs also show that the DBM-tree reduces the average number of distance calculations up to 67% for Rq (graph (g)) and up to 37% for $kNNq$ (graph (j)), when compared to the Slim-tree. When compared to the M-tree, the DBM-tree reduced up to 72% for Rq (graph (g)) and up to 41% for $kNNq$ (graph (j)).

The graphs of the second column (Figures 4(b), (e), (h) and (k)) 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 the number of disk accesses. The graphs show that the DBM-tree reduces the average number of disk accesses up to 43% for Rq (graph (h)) and up to 53% for $kNNq$ (graph (k)), 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 of the Slim-tree. When compared to the M-tree, the gain is even larger, increasing to up to 54% for Rq (graph (h)) and up to 66% for $kNNq$ (graph (k)).

The results are better when the dimensionality and the number of clusters of the datasets increase (as shown for the *Synt16D* and *Synt256D* datasets). The main reason is that traditional MAM produces high overlapping areas with these datasets due both to the high dimension and the need to fit the objects in the inter-cluster regions together with the objects in the clusters. The DBM-tree achieves a very good performance in high dimensional datasets and in datasets with non-uniform distribution (a common situation in real world datasets).

An important observation is that the immediate result of reducing the overlap between nodes is a 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 sums up to reduce the total processing time of queries.

The graphs of the third column (Figures 4(c), (f), (i) and (l)) show the total processing time (in seconds). As the four 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

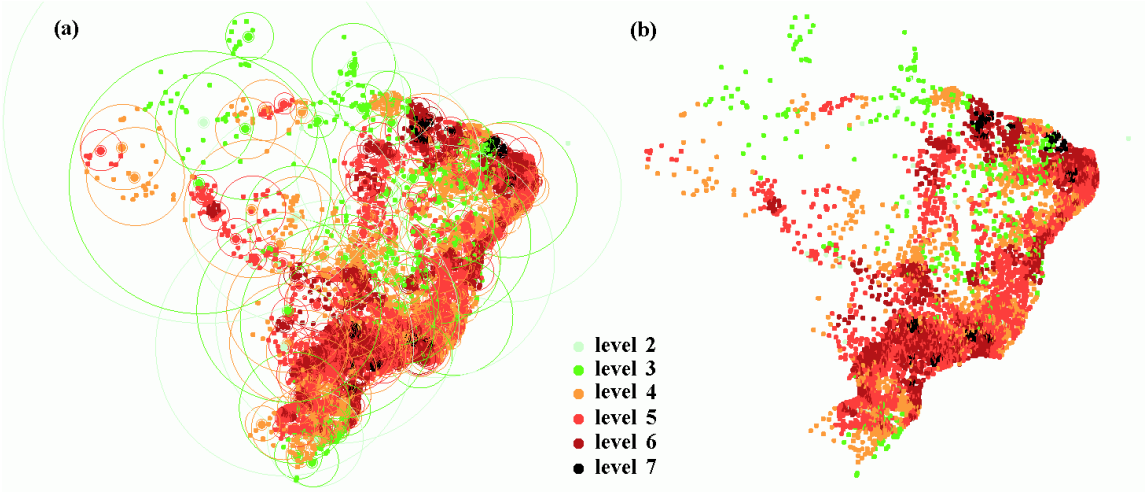

Figure 2. Visualization of the *DBM-MM* structure for the *Cities* dataset. (a) with the covering radius of the nodes; and (b) only the objects. It is possible to verify that the structure is deeper (darker objects) in high-density regions, and shallower (lighter objects) in low-density regions.

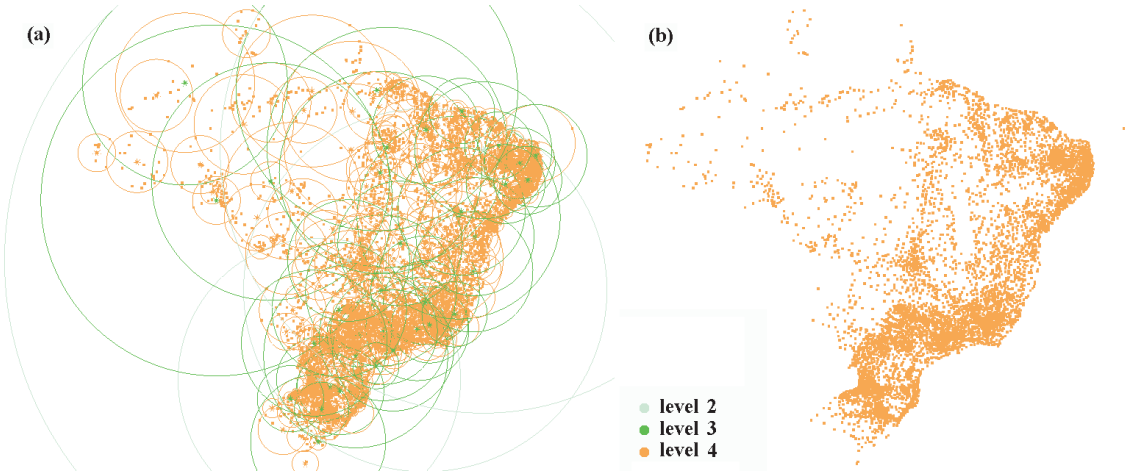

Figure 3. Visualization of the *Slim-tree* structure for the *Cities* dataset. (a) with the covering radius of the nodes; and (b) only the objects. It is possible to verify that the structure has the same level in high-density regions and in low-density regions (level 4).

of distance calculations and the number of disk accesses. The graphs show that the DBM-tree is up to 44% faster to answer Rq and $kNNq$ (graphs (i) and (l)) than Slim-tree. When compared to the M-tree, the reducion in total query time is even larger, with the DBM-tree being up to 50% faster for Rq and $kNNq$ queries (graphs (i) and (l)).

5.3. Experiments regarding the $Shrink()$ Algorithm

The experiments to evaluate the improvement achieved by the $Shrink()$ algorithm were performed on the four DBM-trees over all datasets shown in Table 1. As the results of all the datasets were similar, in Figure 5 we show only the results for the number of disk accesses with the *ColorHisto* (Figures 5(a) for Rq and (b) for $kNNq$) and *Synt256D* dataset (Figures 5(c) for Rq and (d) for $kNNq$).

Figure 5 compares the query performance before and after the execution of the $Shrink()$ algorithm for *DBM-MM*, *DBM-MS*, *DBM-GMM* and *DBM-2CL* for both Rq and $kNNq$. Every graph shows that the $Shrink()$ algorithm improves the final trees. The most expressive result occurs in the *DBM-GMM* indexing the *Synt256D*, which achieved up to 40% lesser disk accesses for $kNNq$ and Rq as compared with the same structure not optimized.

5.4. Cost of Disk Accesses in the DBM-tree

This experiment evaluates the cost model to estimate the number of disk accesses of query operations. Only 10% of the dataset objects were employed to build the histograms $Hist$, as a larger number of objects slightly improves the estimation.

Figure 6 shows the predicted values obtained from the formula 3, and the real measurements obtained executing the query on the tree. Here we show only experiments

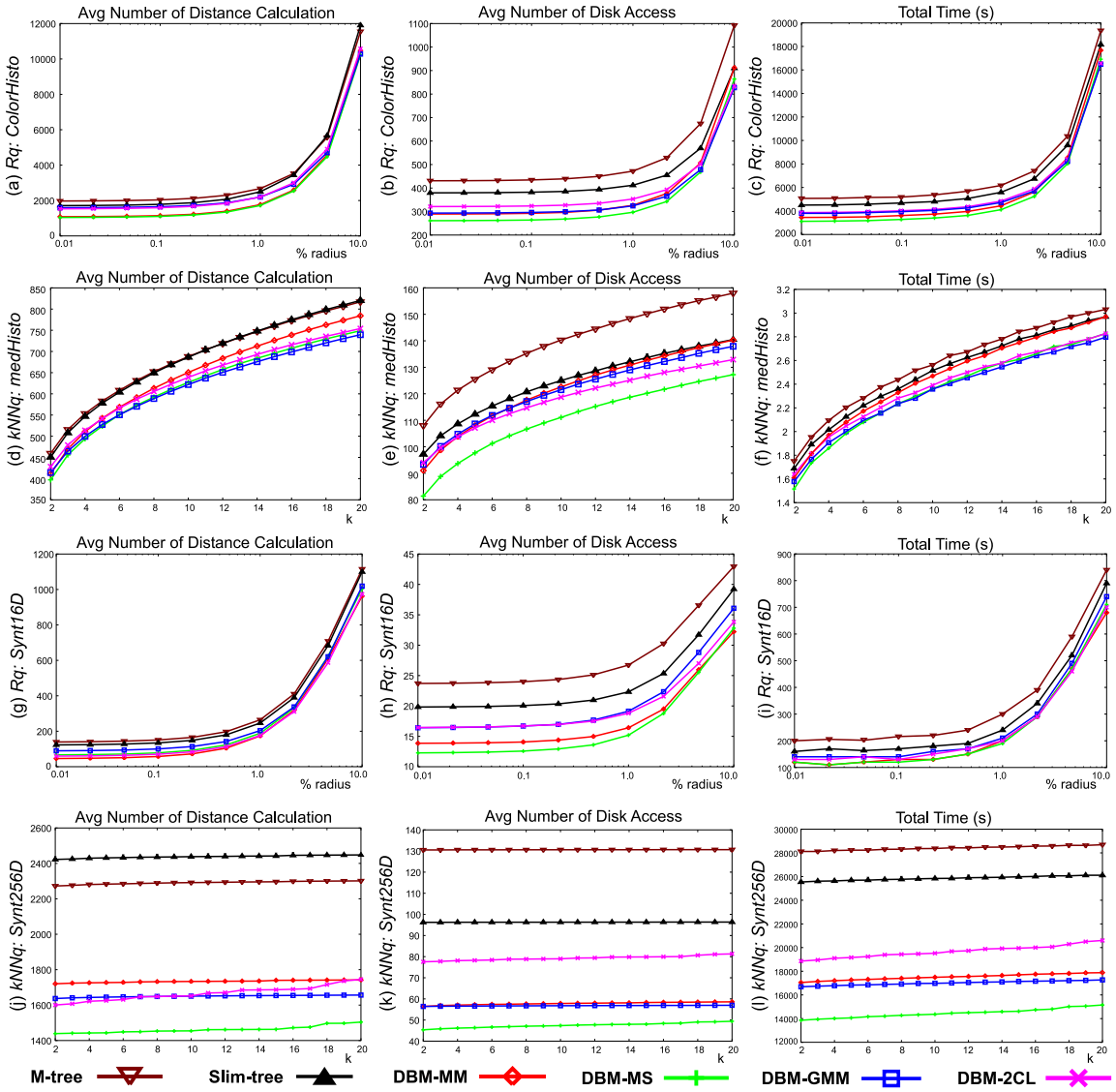

Figure 4. Comparison of the average number of distance calculations (first column), average number of disk accesses (second column) and total processing time in seconds (third column) of DBM-tree, Slim-tree and M-tree, for Rq and $kNNq$ queries for the *ColorHisto* ((a), (b) and (c) - Rq), *MedHisto* ((d), (e) and (f) - $kNNq$), *Synt16D* ((g), (h) and (i) - Rq) and *Synt256D* ((j), (k) and (l) - $kNNq$) datasets.

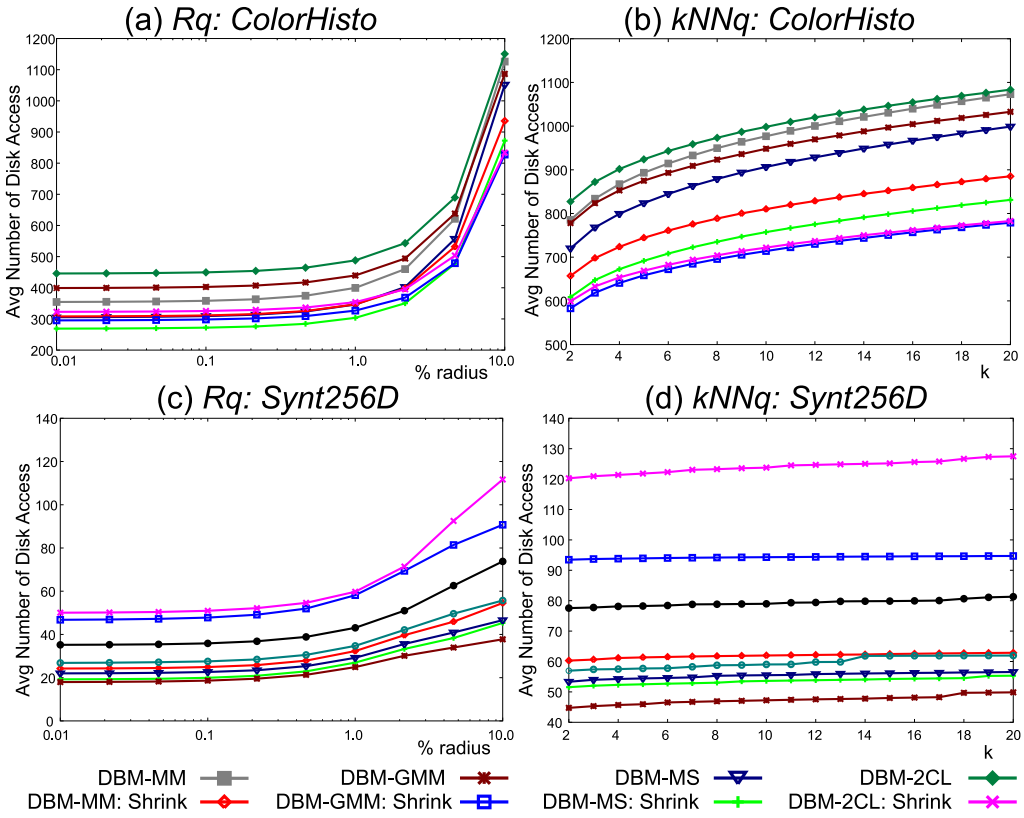

Figure 5. Average number of disk accesses to perform Rq and $kNNq$ queries in the DBM-tree before and after the execution of the $Shrink()$ algorithm: (a) Rq on *ColorHisto*, (b) $kNNq$ on *ColorHisto*, (c) Rq on *Synt256D*, (d) $kNNq$ on *Synt256D*.

for the *DBM-MM* on *MedHisto* Figure 6(a), *DBM-MS* on *Synt16D* Figure 6(b), *DBM-GMM* on *ColorHisto* Figure 6(c) and *DBM-2CL* on *Synt256D* dataset Figure 6(d), as the others are similar. The real measurements are the average of 500 queries as before, and the error bars indicate the standard deviation of each measure. It can be seen that the proposed formula is very accurate, showing errors within 1% of the real measurement for the *DBM-GMM*, and within 20% for the *DBM-MS*. The estimations is always within the range of the standard deviation.

5.5. Scalability of the DBM-tree

This experiment evaluated the behavior of the DBM-tree with respect to the number of elements stored in the dataset. For the experiment, we generated 20 datasets similar to the *Synt16D*, each one with 50,000 elements. We inserted all 20 datasets in the same tree, totaling 1,000,000 elements. After inserting each dataset we run the $Shrink()$ algorithm and asked the same sets of 500 similarity queries for each point in the graph, as before. The behavior was equivalent for different values of k and radius, thus we present only the results for k=15 and radius=0.1%.

Figure 7 presents the behavior of the four DBM-tree considering the average number of distance calculations for $kNNq$ (a) and for Rq (b), the average number of disk accesses for $kNNq$ (c) and for Rq (d), and the total processing time for $kNNq$ (e) and for Rq (f). As it can be seen, the DBM-trees exhibit linear behavior as the number of indexed elements, what makes the method adequate to index very large datasets, in any of its configurations.

6. Conclusions and Future Works

This paper presents a new dynamic MAM called *DBM-tree* (*Density-Based Metric tree*) that, 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 optimization 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

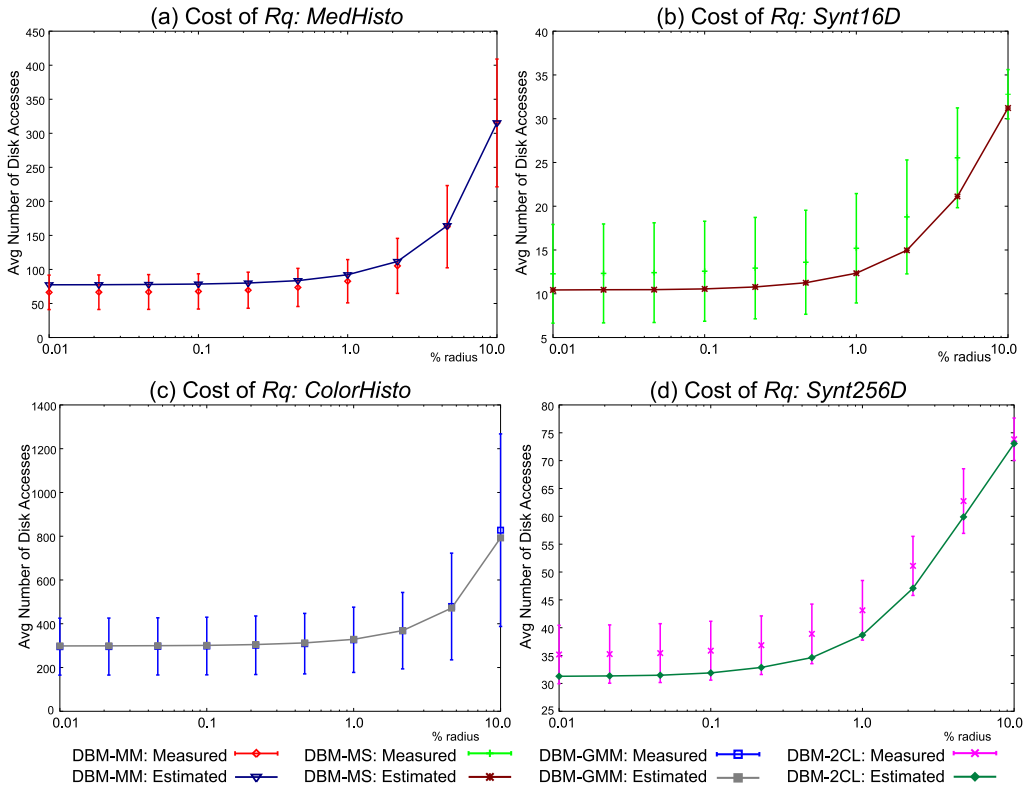

Figure 6. Comparation of the real and the estimated number of disk accesses for Rq in the (a) *MedHisto* dataset using a *DBM-MM* tree, (b) *Synt16D* using a *DBM-MS*, (c) *ColorHisto* using a *DBM-GMM* and (d) *Synt256D* using a *DBM-2CL*.

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 in 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 disk access. The DBM-tree requires up to 66% fewer disk accesses than the balanced trees. After applying the $Shrink()$ algorithm, the performance achieves improvements up to 40% 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 indexed elements, presenting 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. Moreover, they remain 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 *LNCS*, pages 198–212, Asilomar, USA, 1994. Springer Verlag.

[2] Tolga Bozkaya and Meral zsoyoglu. 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 zsoyoglu. 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 Confer-*

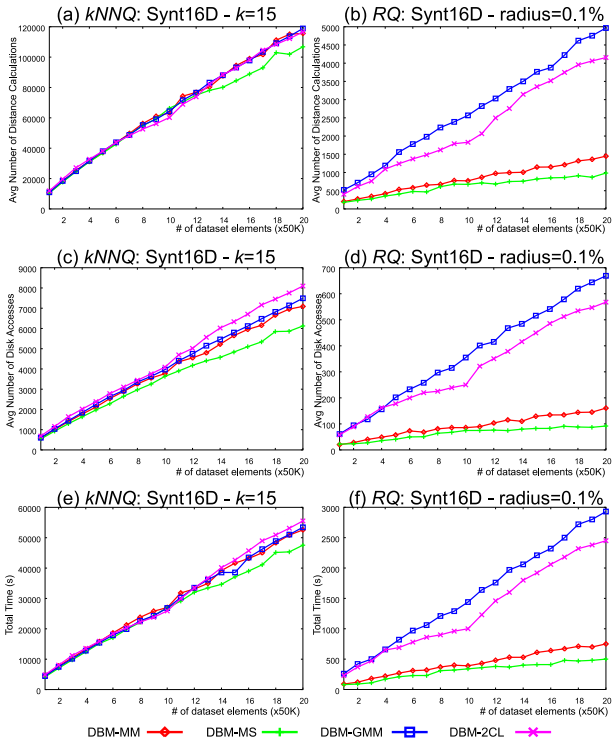

Figure 7. Scalability of DBM-tree regarding the dataset size executing $kNNq$ queries ((a), (c) and (e)) and Rq queries ((b), (d) and (f)), measuring the average number of distance calculations ((a) and (b)), the average number of disk accesses ((c) and (d)) and the total processing time ((e) and (f)). The indexed dataset was the *Synt16D* with 1,000,000 objects.

ence on Very Large Data Bases (VLDB), pages 574–584, Zurich, Switzerland, 1995. Morgan Kaufmann.

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

[6] Fabio J. T. Chino, Marcos R. Vieira, Agma J. M. Traina, and Caetano Traina Jr. Mamview: A visual tool for exploring and understanding metric access methods. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC)*, page 6p, Santa Fe, New Mexico, USA, 2005. ACM Press.

[7] Edgar Chvez, Gonzalo Navarro, Ricardo Baeza-Yates, and Jos Luis Marroqun. Searching in metric spaces. *ACM Computing Surveys (CSUR)*, 33(3):273–321, sep 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 59–68, 1998.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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, 2002.

[15] 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 *LNCS*, pages 51–65, Konstanz, Germany, 2000. Springer.

[16] K. A. Ross, I. Sitzmann, and P. J. Stuckey. Cost-based unbalanced R-trees. In *IEEE International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 203–212, 2001.

[17] Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(1):19–32, 2000.

[18] 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 *Sixth IFIP Working Conference on Visual Database Systems (VDB)*, Brisbane, Australia, 2002.

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

[20] Marcos R. Vieira, Caetano Traina Jr., Fabio J. T. Chino, and Agma J. M. Traina. DBM-tree: A dynamic metric access method sensitive to local density data. In *XIX Brazilian Symposium on Databases (SBBD)*, pages 163–177, Brasília, Brazil, 2004.

[21] 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.