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

Department of Computer Science, University of São Paulo at São Carlos, São Carlos, SP - Brazil
{mrvieira,caetano,chino,agma}@icmc.usp.br

Abstract. Metric Access Methods (MAM) are widely employed to speed up the evaluation of similarity queries, such as range and k-nearest neighbor queries. Most of the currently MAM available today achieve reasonable good performance of query evaluation through minimizing the number of disk accesses required to evaluate a query. The classical way of minimizing the number of disk accesses in hierarchical access methods is to keep the height of the structures very short. This class of structures is called height-balanced structures and are widely employed in Database Management Systems (DBMS). Slim-tree and M-tree are examples of MAM that are height-balanced structures which achieve very good performance both in terms of disk accesses and running time mainly because the height of the trees are kept very short. However, the performance of these two structures degrade very easy because the covering radius of nodes and the overlapping between nodes increase in a way that a large number of subtrees have to be analyzed when processing a query. 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, in a controlled way, "relaxing" the height-balancing of the structure. Thus, the height of the tree is higher in denser regions in order to keep a tradeoff between breadth-searching and depth-searching. We also present a new optimization algorithm called `Shrink`, which improves the performance of already built DBM-trees by reorganizing the elements among their nodes. Experiments performed over several synthetic and real-world datasets showed that the DBM-tree is on average 50% faster than traditional MAM. The DBM-tree also reduces the number of distance calculations by up to 72% and disk accesses by up to 54% for answering queries. After performing the `Shrink` algorithm, query performance of the DBM-tree improves up to 30% regarding the number of disk accesses. In addition, the DBM-tree scales up well, exhibiting sub-linear performance when increasing the database size.

Categories and Subject Descriptors: Information Systems [**Miscellaneous**]: Databases

Keywords: indexes, metric access methods, similarity queries

1. INTRODUCTION

In the last decades, the volume of data managed by Database Management Systems (DBMS) is increasing in large proportions. 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 incorporated to DBMS. In order to handle these new complex data types and the continuously increasing volume of data managed by DBMS, new index structures, as well as new query processing techniques, have been proposed.

The main technique employed to speed up query processing in commercial DBMS is building efficient index structures, also know as Access Methods (AM). Commercial DBMS work pretty well on traditional data domains, e.g. numbers and short strings of characters, because on such domains the *total ordering* property holds among elements in the data domain. For example, we can establish

A previous version of this paper was published in the XIX Brazilian Symposium on Databases [Vieira et al. 2004].
This work was supported by the FAPESP (São Paulo State Research Foundation) under grants 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.

total ordering among natural numbers using the "<" relational operator, e.g. $2 < 5 < 7 < 13$. Many AM used by DBMS today, like the B-tree [Bayer and McCreight 1972] and the B$^+$-tree [Comer 1979], can answer interval and equality queries very efficiently because the total ordering property can be explored. In other words, in each step of the search evaluation a large number of elements can be "pruned" from the search space.

Unfortunately, the total ordering property does not hold for most of the complex data domains [Faloutsos 1996], which makes the use of traditional AM to index and query complex data infeasible. Nevertheless similarity queries are more intuitive in these data domains and comparison is performed between pair of elements. For a given reference element, also called *query element*, a similarity query returns all elements that meet a given similarity criteria w.r.t. the query element. Traditional AM rely on the total order relationship only, and are not able to handle these complex data properly or to answer similarity queries over such data. On the other hand Metric Access Methods (MAM) are well-suited to answer similarity queries over complex data types.

MAM such as Slim-tree [Traina Jr. et al. 2000], [Traina Jr. et al. 2002] and M-tree [Ciaccia et al. 1997] were developed to answer similarity queries based on the similarity relationships among pairs of elements. The similarity relationships are usually represented by distance functions computed over pair of elements. The data domain and distance function defines a metric space. 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**: $d(s_1, s_2) = d(s_2, s_1)$, $s_1 \in \mathbb{S}$ and $s_2 \in \mathbb{S}$;
—**non-negativity**: $0 < d(s_1, s_2) < \infty$ if $s_1 \neq s_2$, and $d(s_1, s_1) = 0$; and
—**triangle 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 domains with any L_p distance function, such as Euclidean distance (L_2) and Manhattan distance (L_1), are special cases of metric spaces [Chavez et al. 2001], [Hjaltason and Samet 2003]. The two most common types of similarity queries are:

—**Range query -** RQ: given a query element $s_q \in \mathbb{S}$, a maximum query distance r_q, and a search domain $S \subseteq \mathbb{S}$, $RQ(s_q, r_q)$ returns a set of elements $R \subseteq S$, such that $\forall s_i \in R, d(s_i, s_q) \leq r_q$. An example is: "Select all 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 element $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, $kNNQ(s_q, k)$ returns $R \subseteq S$ containing k elements that have the smallest distance from the query element s_q, according to the distance function d, that is $k = |R|$ and $\forall s_i \in R, s_j \in \{S - R\}, d(s_q, s_i) \leq d(s_q, s_j)$. An example is: "Select the 3 most similar proteins to protein p", 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 overlapping among nodes with elements in high-density regions by relaxing the height of the structure. Thus, the height is higher for subtrees that cover areas with high density of elements in order to keep a "compromise" between the number of disk accesses required to perform a breadth-search in several subtrees and to perform a depth-search in one subtree. We show in our experiments that the query performance of the DBM-tree is better than the "rigidly" balanced trees (Slim-tree and M-tree). In order to further optimize already built DBM-trees we propose an optimization algorithm called `Shrink`, which reorganizes elements among subtrees in order to reduce even more the overlapping among nodes.

Experiments performed over several synthetic and real-world datasets show that the DBM-tree outperforms the traditional MAM Slim-tree and M-tree. The DBM-tree is on average 50% faster than traditional balanced MAM. It also reduces up to 54% the number of disk accesses and up to 72% the number of distance calculations required to answer similarity queries. After optimizing DBM-trees with the `Shrink` optimization algorithm, the total number of disk accesses to answer similarity queries decreased up to 30% compared with non-optimized DBM-trees. We also show in our experiments that the DBM-tree is scalable regarding the database size, exhibiting sub-linear behavior in the total processing time, the number of disk accesses and the number of distance calculations.

The remainder of this paper is organized as follows: Section 2 presents the basic concepts used in this paper; Section 3 summarizes the related work; The DBM-tree organization and algorithms are explained in Section 4; Section 5 describes the experiments performed in order to compare the DBM-tree with the Slim-tree and M-tree; and Section 6 concludes the paper and suggests new directions of research.

2. BACKGROUND

Access Methods (AM) are one of the most used resources for improving search performance in DBMS. AM employ important properties of the data domain to achieve good performance. One of the most used properties in hierarchical access methods is the total ordering property. Using this very useful property it is possible to discard large subsets of data even without comparing them with the query element. For example, consider the case of numeric data where the total ordering property holds; this property allows us to split a dataset in this domain into two subsets: a subset with all numeric elements that are greater than a reference numeric element; and a subset with numeric elements that are smaller or equal than a reference numeric element. Hence, one efficient way to perform several searches in a numeric domain is to maintain the dataset in this domain sorted. Then, whenever there is a need to perform a search using a query number on this domain, comparing the query number with a "particular" number in the dataset enables the search algorithm to discard further comparisons with the part of the data where the number cannot be in. A well-know example of a search algorithm that employes this property is the one used in the B^+-tree [Comer 1979].

One very important class of AM is the hierarchical structures (or tree structures), which enables recursive processes to index and search the data. These structures can be classified as (height-)balanced or unbalanced, where in the first case the heights of all subtrees in the same level of a tree are the same, or at most they differ by a fixed amount. The elements are stored in blocks called nodes. When a search is performed, the query element is compared with one or more elements in the root node to decide which subtree(s) need to be traversed. This process is repeatedly applied to each subtree that can possibly store elements that belong to the answer set.

Note that whenever the total ordering property applies, only a single subtree at each tree level can hold the answer for equality queries. If the partial ordering property applies to a data domain, then it is possible that more than one subtree need to be analyzed in each tree level. As numeric domains possess the total ordering property, trees indexing numbers require access of only one node at 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 of nodes, or overlapping between subtrees, and happens in structures like R-trees [Guttman 1984], [Sellis et al. 1987], [Beckmann et al. 1990].

In general, DBMS store nodes of hierarchical structures in disk using fixed-size pages. Storing nodes in disk is essential to warrant persistent data and to allow handling a very large number of elements. However, as disk accesses are slow it is very important to keep small the number of disk accesses required to process a query. Traditional DBMS build indexes only on data holding the total ordering property. Thus, if a tree grows deeper, more disk accesses are required to traverse the tree. Therefore it is crucial to keep every tree as shallow as possible in order to minimize the number of accesses to disk. It is possible that whenever a tree is allowed to grow unbalanced, the structure can degenerate completely, making it useless. Therefore, only balanced trees have been widely employed in commercial DBMS.

Metric Access Methods (MAM), or just metric trees, divide a particular dataset in one or more regions, and choose representative elements for each region to represent them. Each node in a MAM stores a single representative s_{rep} (there are structures that store more than one representative, e.g. MVP-tree [Bozkaya and Özsoyoglu 1997]), the minimum distance radius $s_{rep}.radius$ from the representative element that covers the entire covering region of the node, a set of elements $\{s_1, s_2, ..., s_c\}$ inside its covering region $s_{rep}.radius$, and the distance from each element covered in its radius to the representative of the node $\{d(s_{rep}, s_1), d(s_{rep}, s_2), ..., d(s_{rep}, s_c)\}$. This node representation is commonly used for leaf nodes, and for index nodes the representative element s_i has also a pointer to the subtree $s_i.subtree$ that is covered by the radius $s_i.radius$. Using this general representation, a hierarchical structure is build to construct a metric tree. Whenever a query is evaluated, the query element is first compared with the representatives of the root node. The triangle inequality is then used to prune subtrees, avoiding distance calculations between the query element and elements or subtrees in the region pruned. Distance calculations between complex elements can have a high computational cost to calculate. Therefore, to a MAM to achieve a good query performance, not only the number of disk accesses has to be minimal, but also the number of distance calculations.

There are two very useful ways to employ the triangle inequality property to avoid calculating distance between elements and the query element. The first one is for pruning the whole subtree $s_i.subtree$ using the distance between its representative s_i and the query element s_q. If the distance between s_q and s_i is greater than the query radius r_q plus the radius of the subtree $s_i.radius$, then we can safely prune *all* elements in the subtree $s_i.subtree$. In other words, if $d(s_q, s_i) > r_q + s_i.radius$, then there is no element in the whole subtree $s_i.subtree$ that qualify to be in the answer set. The second one is avoiding computing the distance calculation $d(s_i, s_q)$, and thus the subtree $s_i.subtree$ as well, between the query element s_q and the representative of a subtree s_i. If the difference between the distance $d(s_{rep}, s_q)$ and $d(s_{rep}, s_i)$ is greater than the query radius r_q plus the radius of the subtree $s_i.radius$, then we do not need to compute $d(s_i, s_q)$ to safe prune the whole subtree $s_i.subtree$. In other words, if $|d(s_{rep}, s_q) - d(s_{rep}, s_i)| > r_q + s_i.radius$, then we do not need to compute $d(s_q, s_i)$ to prune $s_i.subtree$.

The effect of node overlapping is observed in all dynamic MAM that are well suitable to work on disk. When a similarity query is performed, the total number of disk accesses depends both on the height of the tree (depth-search) and on the amount of overlapping among the nodes (breadth-search). In this case, it is not worthwhile reducing the number of levels at the expense of increasing the overlapping among nodes. Indeed, reducing the number of subtrees that cannot be pruned at each node may 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. To the best of the authors' knowledge, no one considered this observation so far for dynamic MAM. The DBM-tree is the first dynamic MAM that "breaks" the widely studied and employed paradigm of building height-balancing structures for persistent data on disk. The proposed DBM-tree considers "relaxing" the height-balancing policy in a "controlled" way in order to build unbalanced subtrees in denser regions of the dataset for a reduced overlapping among subtrees. In our experimental evaluation we show that this "break of paradigm" works very well in **every** measurement observed (disk accesses, distance calculations and running time) for similarity queries using the DBM-tree.

3. RELATED WORK

Several Spatial Access Methods (SAM) have been proposed for multidimensional data so far. Example of such structures are the k-d-tree [Bentley and Friedman 1979], and R-tree structures [Guttman 1984], [Sellis et al. 1987], [Beckmann et al. 1990]. A comprehensive survey showing the evolution of SAM and their main concepts can be found in [Gaede and Günther 1998]. However, the majority of them cannot handle data in metric domains, and suffer from the "curse of dimensionality" [Beyer et al.

1999], being efficient only to index low-dimensional datasets (see [Böhm et al. 2001]).

Metric Access Methods (MAM) seem to be more attractive for a broad class of applications. Two good surveys on MAM can be found in [Chavez et al. 2001] and [Hjaltason and Samet 2003]. The techniques of recursive partitioning the data in metric domains proposed in [Burkhard and Keller 1973] were the starting point for the development of MAM. The first technique divides the dataset choosing one representative (routing element) 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 GH-tree proposed by Uhlmann [Uhlmann 1991] and the VP-tree (Vantage-Point tree) [Yianilos 1993] are examples based on the first technique described in [Burkhard and Keller 1973]. Aiming to reduce the number of distance calculations to answer similarity queries in the VP-tree, Baeza-Yates et al. proposed the FQ-tree [Baeza-Yates et al. 1994] where the same representative is used for every node in the same level. The MVP-tree (Multi-Vantage-Point tree) [Bozkaya and Özsoyoglu 1997], [Bozkaya and Özsoyoglu 1999] 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) [Uhlmann 1991] is another method that recursively partitions the dataset into two groups, selecting two representatives and associating the remaining elements to the nearest representative. The GNAT (Geometric Near-Neighbor Access tree) [Brin 1995] can be viewed as a refinement of the second technique presented in [Burkhard and Keller 1973], where it stores the distances between pairs of representatives and the biggest distance between each stored element to each representative. The GNAT uses these information and the triangle inequality to prune distance calculations.

Unfortunately all of the MAM discussed so far are static in the sense that the tree is built at once using the whole dataset, and new insertions are not allowed afterward. Furthermore, they mostly attempt to reduce the number of distance calculations, paying no attention on disk accesses. The M-tree [Ciaccia et al. 1997] was the first MAM to overcome these deficiencies. It is a height-balanced tree based on the second technique of [Burkhard and Keller 1973], with data elements stored in leaf nodes. The Slim-Tree [Traina Jr. et al. 2000], [Traina Jr. et al. 2002] is an "evolution" of M-Tree, embodying the first published method to reduce the amount of node overlapping, called the *Slim-Down*. The *Slim-Down* process leads to a smaller number of disk accesses to answer similarity queries by reorganizing elements in the leaf level of the tree. Although dynamic, neither the Slim-tree nor the M-tree have element deletion operations described. In [Santos et al. 2001], it is proposed to use multiple representatives, called "omni-foci", to generate a coordinate system for 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".

The dynamic MAM described so far build height-balanced trees. The main reason of building height-balanced trees is to minimize the tree height at the expense of little flexibility in reducing overlap among nodes. Overlapping nodes degenerates query performance mainly because all nodes that has some overlap in the query region cannot be pruned. If the query region lies, even partially, in an overlap region of more than one node, then all these nodes must be further examined recursively, thus increasing the query cost in terms of disk accesses, distance calculations and total running time. In order to maintain the height of the tree minimum in all of the dynamic structures previous described, both the index and leaf nodes have to have a minimum element capacity. Two important observations can be seen when using this approach to keep the height of the tree minimum: **(1)** the covering radius in low dense regions of elements has to be *very large* in order to cover all elements in those regions; **(2)** in *high dense* regions of elements, the node overlapping increases substantially since several nodes, all with the same height, have to be created in order to keep the tree balanced. Because the performance of similarity queries is *highly* correlated to the degree of node overlapping **and** covering radius, we can easily see that the performance of trees that use such approach degrades very fast. To solve all

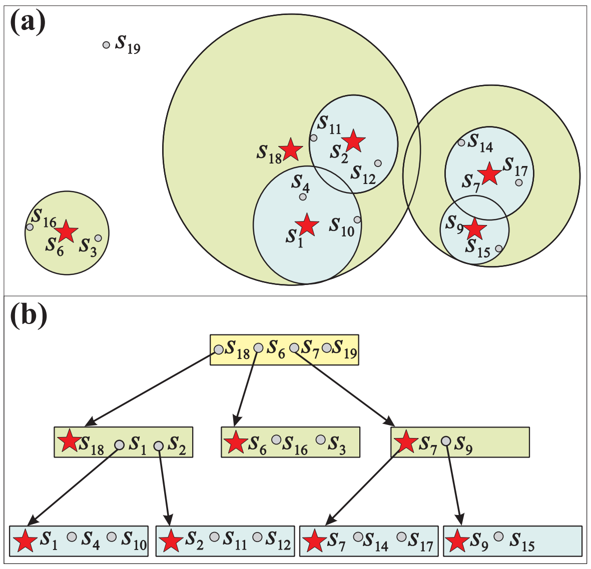

Fig. 1. A schematic view of the DBM-tree: (a) covering radius for subtrees and nodes; and (b) tree node organization. The height of the subtree represented by element s_6 is shallower than the subtrees s_{18} and s_7. elements s_{19}, s_{18} (second level), s_{16} and s_3 are not in the last level of the tree.

of the problems mention before, we take a new direction in our research and *break* the paradigm that "good performance is directed correlated to height-balanced trees".

4. THE DBM-TREE

The DBM-tree is a hierarchical, dynamic structure that grows bottom-up and allows, at any time, new elements to be inserted. Self organizations and node splits can occur in order to accommodate new elements. Basically, when inserting a new element, the tree root is evaluated in order to choose subtrees that can accommodate the new element. There are cases where the new element is inserted in the current node, and not in its subtrees. This process on choosing a node that can accommodate a new element is repeated recursively until a suitable node is found w.r.t. some policy. The DBM-tree has several policies, described in this section, on choosing a suitable node to accommodate a new element, as well as node splitting policies.

The most remarkable difference in the structure, when compared to other structures like the Slim-tree, is that there is no distinction among index and leaf nodes. In this way, elements or subtrees can be stored in *any node* in *any level* of the tree. Similar to other MAM, the DBM-tree uses nodes of fixed size in order to optimize data transfer from/to disk. Its main intent is to organize elements in a hierarchical structure using representatives as centers of minimum bounding regions (or balls) that covers all elements in their subtrees. An element can be assigned to a node if the covering radius of the representative covers it. An example of a DBM-tree indexing elements in a 2-dimensional space is shown in Figure 1. In Figure 1(a), 16 elements are represented with small circles, while the representative for a node with a star. The covering radius of each node or subtree is represented with a large circle covering all elements that belong to the node or subtree. The same tree is shown in Figure 1(b), but this time in a tree representation. Note that element s_{19} in the root node is an element, not a representative, like s_6, s_7 and s_{18}. Storing element s_{19} in the root node prevents that the subtree represented by s_{18} increases its covering radius to cover s_{19}, and thus increasing the overlapping between subtree s_{18} and its siblings s_6 and s_7.

Every node in the DBM-tree has the following structure:

***Node*: $[c_e$, array $[1..c_e]$ of $\{<s_i, d(s_{rep}, s_i), s_i.ptr, s_i.radius>$ or $<s_i, d(s_{rep}, s_i)>\}]$**

where a node can hold up to c entries. Attribute c_e, $c_e \leq c$, stores the total number of entries (elements or subtrees) stored in a node. Entries can be of two types: subtrees, represented by $<s_i, d(s_{rep}, s_i), s_i.ptr, s_i.radius>$, and elements, represented by $<s_i, d(s_{rep}, s_i)>$. For subtree entries, s_i is the representative for the subtree $s_i.ptr$, $s_i.radius$ is the minimum radius that covers *all* entries in subtree $s_i.ptr$, i.e. $\forall s'_i \in s_i.ptr, s_i.radius = max(d(s_i, s'_i) + s'_i.radius)$, and $d(s_{rep}, s_i)$ is the distance between the subtree representative s_i to the node representative s_{rep}. For element entries, only two fields are required: the element itself s_i with identifies $s_i.oid$, and the distance $d(s_{rep}, s_i)$ between s_i to the representative of the node s_{rep} that it belongs to. This distance is stored and used, using the triangle inequality previous described, in order to avoid computing the distance of the element to the query element. All element entries, stored in a particular node, are not covered by any subtree that it belongs to, that is, there is no subtree that can accommodate the element without changing the radius of the subtree. An example of such example is element s_{19} in Figure 1; there is no subtree in the root node that accommodate element s_{19} without changing its radius (subtrees s_{18}, s_6 and s_7).

If an element is chosen to be a representative of its node, then a copy of it is promoted to an immediate level up the tree (unless the node is the root of the tree). This is similar to the promotion process of the B$^+$-tree. For a set of elements in a node, the best *local* choice for a representative is the one that has the minimum covering radius. This simple heuristic is used for both Slim-tree and M-tree. The intuition behind it is that, minimizing the radius for a particular node also leads to less covering area among its siblings. Moreover, this heuristic is very easy to implement since updates only occur in the path of insertion.

The DBM-tree self adjust itself when a new element is inserted in the structure. The process of insertion starts from the root of the tree to one node in the tree (*top-bottom fashion*). This process is recursive in nature and try to find a node that better accommodate the new element. There are two policies on choosing a node, or a subtree, to insert the new element. They are:

—`minDist`: among all the subtrees that cover the new element, choose the one that has the smallest distance between the representative and the new element. If there is no such subtree, then the element is inserted in the current node;

—`minGDist`: among all the subtrees that cover the new element, choose the one that has the smallest distance between the representative and the new element. In case there is no such subtree that satisfy this condition, then it is chosen a subtree that has its representative closer to the new element. Thus, in the second condition, the covering radius needs to increase in order to accommodate the new element.

The main difference between these two policies are that, for the first one the flexibility in creating unbalanced trees is more than the second one. That is, if there is no subtree that can accommodate the new element without changing its covering radius, then it is inserted in the current node, no matter the level of the current node. On the other hand, in the second policy whenever the case mention before fails, then a subtree is chosen if its representative is the closest to the new element.

The algorithm to insert new elements is described in Algorithm 1. The `ChooseSubtree` function returns the most suitable place to insert the new element, based on the policy described above (`minDist` or `minGDist`). 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 element s_{new}. The insertion of s_{new} can occur at any level of the structure, depending on the policy of `ChooseSubtree`. In each node, the `Insert` algorithm calls the `ChooseSubtree` function, which returns the subtree that better qualifies to store the new element. If there is no subtree that qualifies, the new element is inserted in the current node *ptr*.

Algorithm 1 `Insert`

```
Require: $s_{new}$: new element to be inserted,  $ptr$: pointer to the subtree where $s_{new}$ will be inserted
 1: $s_i \leftarrow$ ChooseSubtree($s_{new}, ptr$) {find a suitable subtree that can accommodate $s_{new}$}
 2: if $s_i$ is a valid subtree then
 3:    $promotion \leftarrow$ Insert($s_{new}, s_i.ptr$) {recursively insert $s_{new}$ in $s_i$ subtree}
 4:    if $promotion$ =true then
 5:       Update($ptr$) {update the current node}
 6:       insert the element set not covered for node split in the current node
 7:       for each $s_j$ now covered by the update do
 8:          Demote($s_j$) {demote entries for optimization purposes}
 9:       end for
10:    end if
11: else if $ptr.c_e < c$ then
12:    $ptr.Add(s_{new})$ {insert $s_{new}$ in node $ptr$}
13: else
14:    SplitNode($s_{new}, ptr$) {node $ptr$ does not have enough space}
15:    return $promotion$ {creates a new node, and promotes two representatives for new and old nodes}
16: end if
```

The policy chosen by the `ChooseSubtree` function has a big 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 chosen node does not have enough space to accommodate s_{new}, then all the existing entries together with the new element must be redistribute between one or two nodes, depending on the redistribution policy of the `SplitNode` function. The `SplitNode` function deletes the node *ptr* and remove its representative from its parent node. Its former entries and s_{new} 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* not covered by the new nodes, are promoted and inserted in the parent node. 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:

—`minMax`: this option distributes entries into two nodes, allowing a possibly empty set of entries not covered by these two nodes. Each pair of entries is considered as candidate to be the representatives of each new node. 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 $\boldsymbol{O}(c^3)$, where c is the number of entries to be distribute between the nodes;

—`minSum`: this option is similar to `minMax`, but the two representatives selected is the pair with the smallest *sum* of the two covering radii.

The minimum node occupation can 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 [Ciaccia et al. 1997], 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 regions between the two nodes.

If the minimum occupation is set to a value lower than half of the node capacity, each node is first

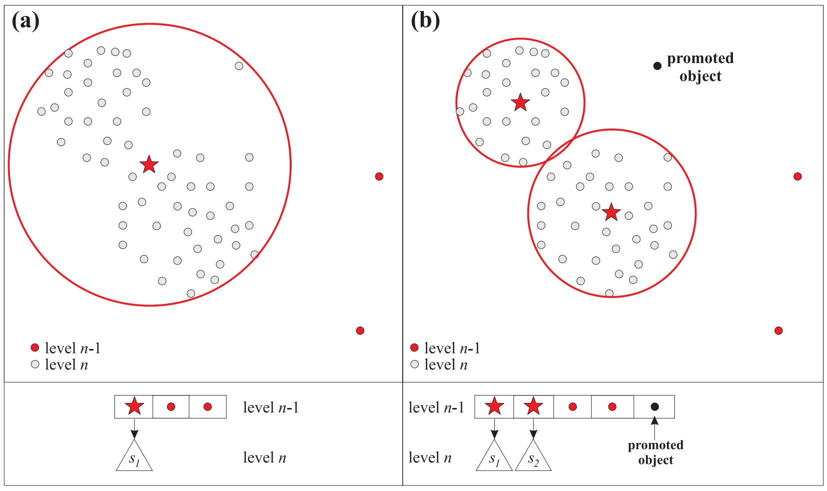

Fig. 2. In order to accommodate two new elements (represented as bold circles), the node in (a) needs to be split into two. Because three elements (represented in bold in (b)) are in an area with low element density, including them in any of the two new nodes it generates to much overlapping between these two nodes, and also triggers more overlapping in other parts of the tree (not shown here).

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 or the update of the representative radii, it can occur that former uncovered single element 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 (`Demote`). Figure 2 illustrates a node configuration (a) before and (b) after a split. It is possible to see that if any of the three elements promoted (bold circles) were included in any of the two new nodes (represented with a large circle for its covering radius and star for its representative), then the overlap would be very large among these two nodes. Moreover, this expansion could trigger much more overlapping in their siblings and parents. Instead, we let these three elements that are in a low density area to be included in a node belonging in a higher level in the tree.

4.1 Similarity Queries on DBM-tree

Here we describe the two main similarity queries implemented in the DBM-tree: Range query (RQ) and k-Nearest Neighbor query ($kNNQ$). Both queries work in a similar way as the ones in the Slim-tree and M-tree. We describe both algorithms here for informative purposes. The only major difference among the algorithms is in the `kNNQ` algorithm for the DBM-tree. This difference is detailed next.

The `RQ` algorithm is described in Algorithm 2. It receives as input parameters a tree node ptr, the query element s_q and radius r_q. This `RQ` algorithm recursively analyze subtrees that cannot be pruned first using the triangle inequality, and then the actual distance between representatives and the query element. The use of the triangle inequality property allows pruning *whole* subtrees or elements that do not intersect the region defined by the query. Entries that cannot be pruned using the triangle inequality property are further analyzed using the real distance between the entry (element or representative of subtree) and the query element. Entries that are subtrees are recursively analyzed, and elements are inserted or not in the answer set R. In the end, the `RQ` algorithm returns all elements that are inside the region defined by the query element s_q and its radius r_q.

Algorithm 2 `RQ`

```
Require: $ptr$ tree to be perform the search, query element $s_q$ and query radius $r_q$
 1: for each $s_i \in ptr$ do
 2:   {use the triangle inequality first}
 3:   if $|d(s_{rep}, s_q) - d(s_{rep}, s_i)| \leq r_q + s_i.radius$ then
 4:     $dist \leftarrow d(s_i, s_q)$ {compute distance}
 5:     if $dist \leq r_q + s_i.radius$ then
 6:       if $s_i$ is a subtree then
 7:         RQ($s_i.ptr, s_q, r_q$) {recursively analyze subtree $s_i.ptr$}
 8:       else
 9:         $R.Add(s_i)$ {add element $s_i$ to the result set $R$}
10:       end if
11:     end if
12:   end if
13: end for
```

The `kNNQ` algorithm requires a dynamic radius r_k to perform the pruning. In the beginning of the process r_k is set to a value that covers all the indexed elements. r_k is adjusted with a smaller new radius after k elements are inserted in R. After this, whenever a swap occurs, r_k is adjusted again with a new smaller radius, e.g. the biggest distance of all elements in R to the query element. A swap happens if an element closer to the query element is found and this distance is smaller than any entry in R. This principle is also applied in both Slim-tree and M-tree. The only main difference between the DBM-tree and these two structures is related to the evaluation order of entries in node *ptr*. In order to fast shrink r_k, entries that are elements in node *ptr* are analyzed first, then its subtrees are further analyzed in a second step. The `kNNQ` algorithm uses this two distinct phases mainly because the rate that r_k adjusts to the optimal value[1] interferes in the performance of the $kNNQ$ algorithm. We do not show the experiments of this operation in this article, but on average the running time is up to 18% faster using the two phases analyzes.

4.2 The `Shrink` Optimization Algorithm

We also propose a post-optimization algorithm for already built DBM-trees called `Shrink`. This algorithm aims at "shrinking" the nodes by exchanging entries between nodes in order to reduce the amount of overlapping between subtrees. Reducing overlapping regions improves the structure query performance, 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. The `Shrink` algorithm can be executed at any time during the evolution of a DBM-tree, for example, after the insertion of many new elements.

The `Shrink` algorithm is described in Algorithm 3. The optimization process is applied in every node of a DBM-tree. The stop condition for this algorithm occurs in two cases: when there is no entry exchanges in a previous iteration; or when the number of exchanges already performed is larger than 3 times the total number of entries in the node. This latter condition assures that no cyclic exchanges can lead to a dead loop. After performing some experiments we observed that increasing this number to a bigger value does not improve too much the structure. For each entry s_i in node *ptr*, the algorithm searches for another entry s_j that covers the *farthest* entry s_i.*farthest* from s_i. If such node exists, then entry s_i.*farthest* is remove from s_i and inserted in s_j. If after some exchanges node s_i become empty, then s_i is removed from the structure. After these exchanges, the information associated to s_i and s_j needs to be updated in *ptr* node. 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.

[1]the optimal r_k is the maximum distance in the final result R

Algorithm 3 `Shrink`

```
Require: $ptr$: root of the tree to be optimized
 1: $exchanges \leftarrow 0$ {variable used to count the total number of exchanges}
 2: $changed \leftarrow$ true {variable used to check if an exchange was performed}
 3: {default value of $MAXEX$ is set to 3}
 4: while $exchanges \leq MAXEX * ptr.c_e$ and $changed$=true do
 5:    $changed \leftarrow$ false
 6:    for each subtree $s_i \in ptr$ do
 7:       let $s_i.farthest$ be the farthest entry in $s_i$ from its representative
 8:       for each entry $s_j \in ptr$ and $s_i \neq s_j$ do
 9:          if $d(s_j, s_i.farthest \leq s_j.radius$ and $s_j.c_e < c$ then
10:             {node $s_j$ covers $s_i.farthest$ and has space to store it}
11:             $s_i.Remove(s_i.farthest)$ {remove entry $s_i.farthest$ from $s_i$}
12:             $s_j.Add(s_i.farthest)$ {insert entry $s_i.farthest$ in $s_j$}
13:             $s_i.UpdateRadius()$ {update radius of subtree $s_i$}
14:             $s_j.UpdateRadius()$ {update radius of subtree $s_j$}
15:             $changed \leftarrow$ true
16:             $exchanges \leftarrow exchanges + 1$
17:          end if
18:       end for
19:       if node $s_i.c_e = 0$ then
20:          Delete($s_i.ptr$) {delete node $s_i.ptr$ from the tree}
21:          $ptr.Remove(s_i)$ {remove entry $s_i$ from $ptr$}
22:       end if
23:    end for
24: end while
25: {part to remove single entries in $ptr.s_i$}
26: for each subtree $s_i \in ptr$ do
27:    if node $s_i.c_e = 1$ then
28:       Delete($s_i.ptr$) {delete node $s_i.ptr$ from the tree}
29:       $ptr.Update(s_i)$ {update entry $s_i$ from subtree to element in $ptr$}
30:    end if
31: end for
```

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

Name	# Objs.	D	*Page* (KBytes)	d	Description
ColorHisto	68,040	32	8	L_2	Color image histograms from the UCI-KDD repository (`kdd.ics.uci.edu`). The metric L_2 returns the distance between two elements in a 32-d Euclidean space
MedHisto	4,247	–	4	L_M	Metric histograms of medical gray-level images. This dataset does not have a fixed number of dimension [Traina et al. 2002]
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 [Ciaccia et al. 1997]
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 all cities in Brazil (`www.ibge.gov.br`)

5. EXPERIMENTAL EVALUATION

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 I presents some illustrative datasets used to evaluate the performance of the DBM-tree. The dataset name is indicated together with its total number of elements (# Objs.), the embedding dimensionality of the dataset (D), the page size in KBytes (*Page*), the metric used (d), and a brief description of each dataset.

Table II. 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 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. All three structures (DBM-tree, Slim-tree and M-tree) 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 the well-known, most efficient and used dynamics MAM so far: Slim-tree and M-tree.

The Slim-tree and the M-tree were set up using their best recommended setups. 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 evaluated three configurations for the DBM-tree, all with minimal occupation set to 30% of node capacity, which are the following:

—*DBM-MM*: `minDist` policy for `ChooseSubtree` and `minMax` for `SplitNode` method;
—*DBM-MS*: `minDist` policy for `ChooseSubtree` and `minSum` for `SplitNode` method;
—*DBM-GMM*: `minGDist` policy for `ChooseSubtree` and `minMax` for `SplitNode` method.

We extracted 500 elements for each dataset to generate query centers. They were randomly chosen from each dataset, where half of them (250) were removed from the dataset before creating the trees, and the other half were duplicated in the query set (this last half set is in the tree and the query set). Hence, half of the queries in the query set are indexed in a DBM-tree and the other half are not. This allows us to evaluate queries that are indexed and not. Each dataset were used to build one tree of each type, and every tree was built inserting one element at a time, calculating the average number of distance calculations, average number of disk accesses and total processing time (in seconds). In the plots of measurements, each point corresponds to performing 500 queries with the same parameters but varying query centers. The value for k varies from 2 to 20 for each measurement, and the radius for $kNNQ$ varies from 0.1% to 10% of the largest distance between pairs of elements in the dataset. We chosen these range of values for both queries because they are the most meaningful values used when performing similarity queries. The *RQ* graphics are in *log* scale for the radius abscissa to emphasize the relevant part of the graph. All measurements were performed after the execution of the `Shrink` algorithm.

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

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 performed on 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 heights in the majority of trees were similar to those of the Slim-trees and M-trees. This is an interesting result and indicates that the balancing is not so important for MAM as it is for the conventional structures.

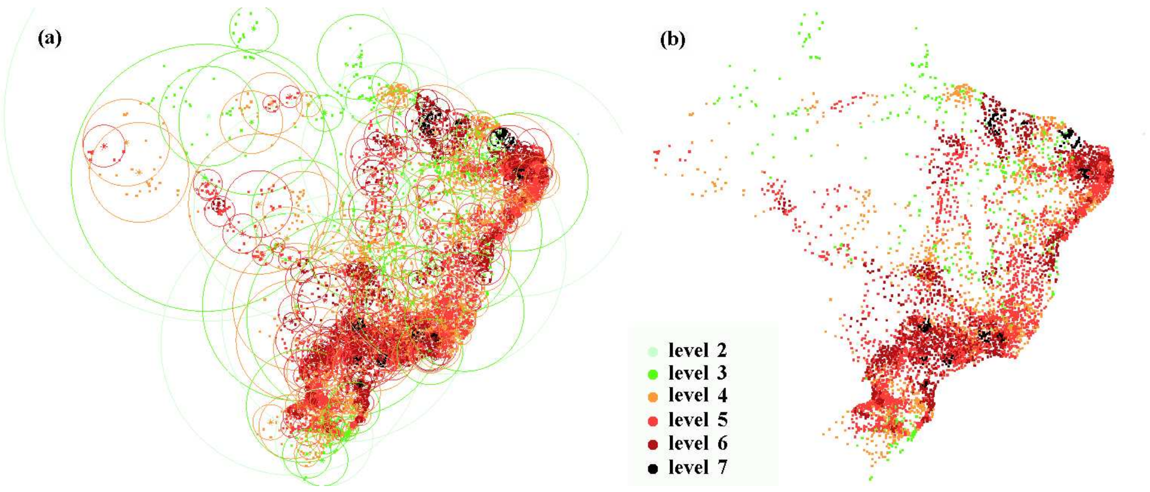

Fig. 3. Visualization of the *DBM-MM* structure for the *Cities* dataset. (a) with the covering radius of the nodes; and (b) only the elements. It is possible to verify that the structure is deeper (darker elements) in high-density regions, and shallower (lighter elements) in low-density regions.

The data distribution across the levels of a DBM-tree is shown in Figure 3 using the *Cities* dataset. This is only possible because this dataset is in a 2-dimensional Euclidean space. Figure 3 shows the elements indexed in a *DBM-MM* with different color representing elements at different levels of the tree. Figure 3(a) shows all elements and the covering radius of each node in the tree, and Figure 3(b) shows only the elements in the tree. The elements with darker colors (red and black) are in a deeper level than those with lighter colors (green). The depth of the tree is larger in higher density regions and that elements are stored in every level of the structure, as expected. It visually shows that the depth of the tree is smaller in low density regions, and that the number of elements at the deepest levels is small, even in the high-density regions.

5.1 Performance Comparison

We have used many synthetic and real datasets to evaluate the performance of 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 fair sample of the behavior of each structure. The main motivation in these experiments is to evaluate the performance of the DBM-tree with its best competitors with respect to the 2 main similarity queries: RQ and $kNNQ$.

Figure 4 shows the measurements to answer RQ and $kNNQ$ on 4 datasets. The graphs on the first row (Figures 4(a-d)) show the average number of distance calculations. It is possible to note in these graphs that every DBM-tree executed on average less distance calculations than the 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*. None of the DBM-trees executed, for every dataset, more distance calculations than the Slim-tree or the M-tree. 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 wrt 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 4(e-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

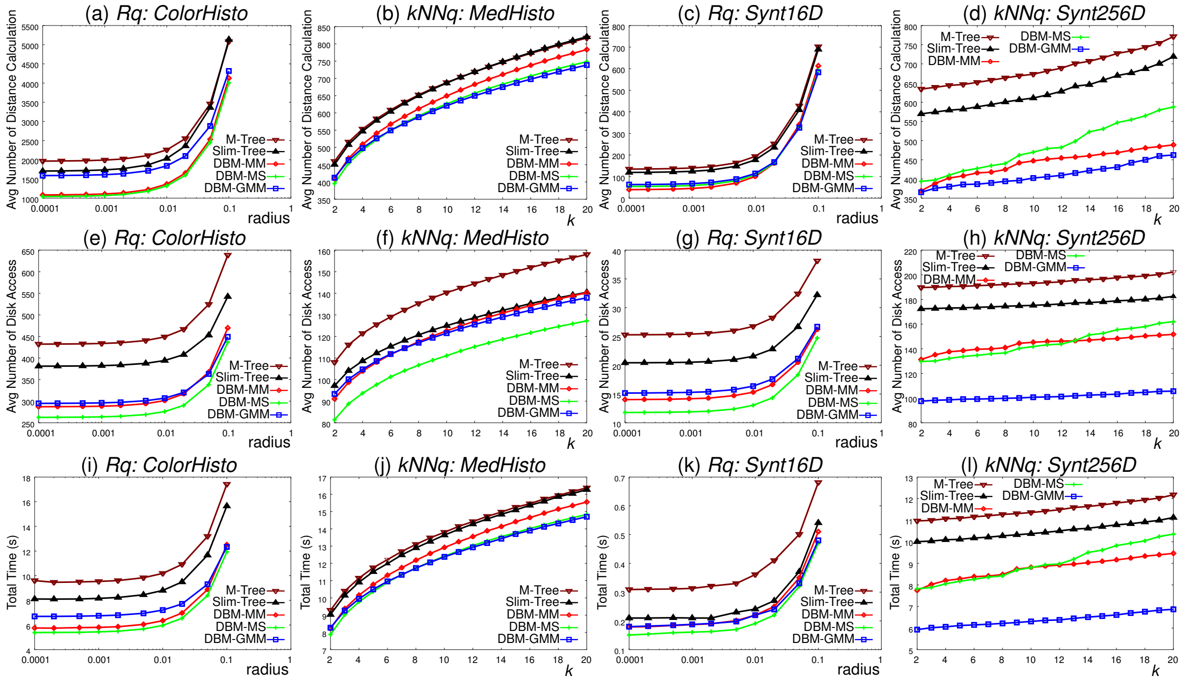

Fig. 4. Comparison of the average number of distance calculations (first row), average number of disk accesses (second row) and total processing time (in *s*) (third row) of DBM-tree, Slim-tree and M-tree, for *RQ* and *kNNQ* queries for the *ColorHisto* ((a), (e) and (i) - *RQ*), *MedHisto* ((b), (f)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

and the M-tree. 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 among 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 4(i-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 reduction 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

Here we test the benefits of running the `Shrink` optimization algorithm for DBM-trees. We only show the results for two datasets, *ColorHisto* and *Synt16D* since the behavior reported here is the same for

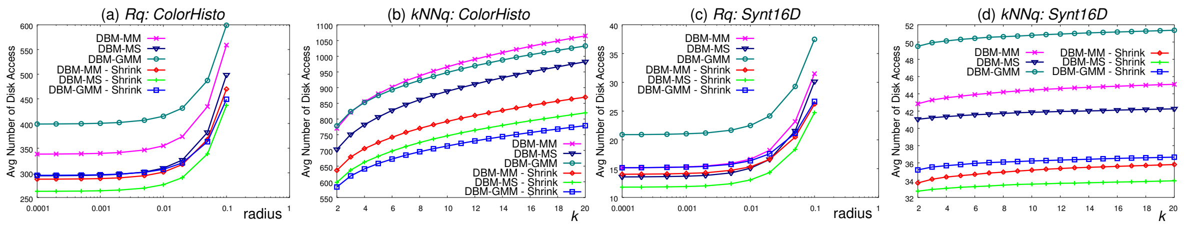

Fig. 5. Average number of disk accesses to perform *RQ* and *kNNQ* queries in the DBM-tree before and after the execution of the `Shrink` algorithm: (a) *RQ* on *ColorHisto*, (b) *kNNQ* on *ColorHisto*, (c) *RQ* on *Synt16D*, (d) *kNNQ* on *Synt16D*.

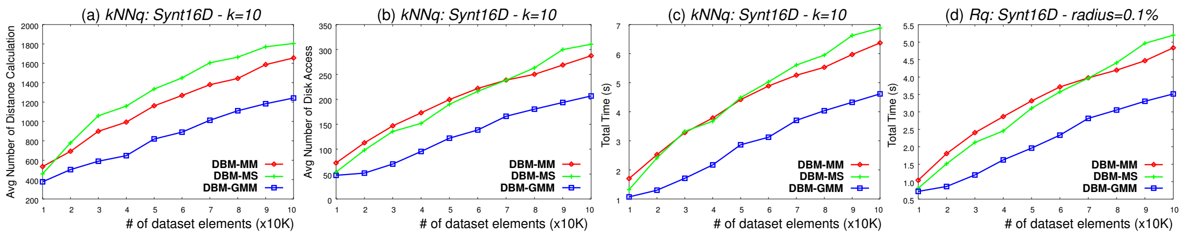

Fig. 6. Scalability of all three DBM-trees regarding the dataset size indexed during *kNNQ* ((a) and (b)) and *RQ* ((c) and (d)), The dataset indexed was the *Synt16D* with 100,000 elements.

other datasets tested. Because the greater benefit of the `Shrink` algorithm is reducing the number of disk accesses for queries, we only report this measurements here. Figures 5(a) for *RQ* and (b) for *kNNQ* represents the results for the *ColorHisto* dataset, and Figures 5(c) for *RQ* and (d) for *kNNQ* for the *Synt16D* dataset.

Figure 5 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*. The performance of the Slim-tree (after ran the Slim-Down) and M-tree are also shown in the figures. 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

In the last set of experiments we evaluated the behavior of the DBM-tree regarding scalability when increasing the number of elements indexed. 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 distinct elements. After inserting each dataset we run sets of queries, executing 500 similarity queries for each point in the graph. The behavior was equivalent for different values of k and radius, thus we present only the results for k=10 and radius=0.1%.

Figure 6 presents the behavior of the three DBM-tree considering: the average number of distance calculations (first row), disk accesses (second row), and total running time (third row) for *kNNQ* (first column) and *RQ* (second column). 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.

6. CONCLUSIONS

This paper presents a new dynamic MAM called *DBM-tree* (*Density-Based Metric tree*), which in a controlled way relaxes the height-balancing requirement of access methods, trading a controlled amount of unbalancing at denser regions of the dataset for a reduced overlapping among 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 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 element-deletion algorithm that can really remove elements from the tree. All existing rigidly balanced MAM such as the Slim-tree and the M-tree, cannot effectively delete elements 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 elements, making it a completely dynamic MAM.

REFERENCES

Baeza-Yates, R. A., Cunto, W., Manber, U., and Wu, S. Proximity matching using fixed-queries trees. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching.* Cancun, Mexico, pp. 198–212, 1994.

Bayer, R. and McCreight, E. M. Organization and maintenance of large ordered indexes. *Acta Informatica* 1 (3): 173–189, 1972.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data Conference.* ACM Press, Atlantic City, USA, pp. 322–331, 1990.

Bentley, J. L. and Friedman, J. H. Data structures for range searching. *ACM Computing Surveys* 11 (4): 397–409, 1979.

Beyer, K. S., Goldstein, J., Ramakrishnan, R., and Shaft, U. When is "nearest neighbor" meaningful? In *Proceedings of the Int'l Conference on Database Theory.* Lecture Notes in Computer Science, vol. 1540. Springer-Verlag, pp. 217–235, 1999.

Böhm, C., Berchtold, S., and Keim, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys* 33 (3): 322–373, 2001.

Bozkaya, T. and Özsoyoglu, M. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data Conference.* Tucson, USA, pp. 357–368, 1997.

Bozkaya, T. and Özsoyoglu, M. Indexing large metric spaces for similarity search queries. *ACM Trans. on Database Systems* 24 (3): 361–404, 1999.

Brin, S. Near neighbor search in large metric spaces. In *Proceedings of the Int'l Conference on Very Large Data Bases.* Zurich, Switzerland, pp. 574–584, 1995.

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

Chavez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. Searching in metric spaces. *ACM Computing Surveys* 33 (3): 273–321, 2001.

Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the Int'l Conference on Very Large Data Bases*. Athens, Greece, pp. 426–435, 1997.

Comer, D. The ubiquitous B-tree. *ACM Computing Surveys* 11 (2): 121–137, 1979.

Faloutsos, C. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Boston, USA, 1996.

Gaede, V. and Günther, O. Multidimensional access methods. *ACM Computing Surveys* 30 (2): 170–231, 1998.

Guttman, A. R-tree : A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data Conference*. Boston, USA, pp. 47–57, 1984.

Hjaltason, G. R. and Samet, H. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems* 28 (4): 517–580, 2003.

Santos, R. F., F., Traina, A. J. M., Traina Jr., C., and Faloutsos, C. Similarity search without tears: The omni family of all-purpose access methods. In *Proceedings of the IEEE Int'l Conference on Data Engineering*. Heidelberg, Germany, pp. 623–630, 2001.

Sellis, T. K., Roussopoulos, N., and Faloutsos, C. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the Int'l Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, Brighton, England, pp. 507–518, 1987.

Traina, A. J. M., Traina Jr., C., Bueno, J. M., and de A. Marques, P. M. The metric histogram: A new and efficient approach for content-based image retrieval. In *IFIP Visual Database Systems*. Brisbane, Australia, pp. 297–311, 2002.

Traina Jr., C., Traina, A. J. M., Faloutsos, C., and Seeger, B. Fast indexing and visualization of metric datasets using Slim-trees. *IEEE Trans. on Knowledge and Data Engineering* 14 (2): 244–260, 2002.

Traina Jr., C., Traina, A. J. M., Seeger, B., and Faloutsos, C. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Proceedings of the Int'l Conference on Extending Database Technology*. Konstanz, Germany, pp. 51–65, 2000.

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

Vieira, M. R., Traina Jr., C., Chino, F. J. T., and Traina, A. J. M. DBM-tree: A dynamic metric access method sensitive to local density data. In *Proceedings of the Brazilian Symposium on Databases*. Brasília, Brazil, pp. 163–177, 2004.

Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*. Austin, USA, pp. 311–321, 1993.