SBBD 2007
XXII Simpósio Brasileiro de Banco de Dados

Bulk-loading Dynamic Metric Access Methods *

Thiago Galbiatti Vespa[1], **Caetano Traina Jr**[1], **Agma Juci Machado Traina**[1]

[1]ICMC - Institute of Mathematics and Computer Sciences
USP - University of São Paulo
Avenida do Trabalhador Sãocarlense, 400
Postal Code: 13566-590 - São Carlos, SP - Brazil

{thiago,caetano,agma}@icmc.usp.br

Abstract. *The main contribution of this paper is a bulk-loading algorithm for multi-way dynamic metric access methods based on the covering radius of a representative, like the Slim-tree. The proposed algorithm is sample-based, and it builds a height-balanced tree in a top-down fashion, using the metric domain's distance function and a bound limit to group and determine the number of elements in each partition of the dataset at each step of the algorithm. Experiments performed to drill its performance shows that our bulk-loading method is up to 6 times faster to build a tree than the sequential insertion method regarding construction time, and that it improves the search performance too.*

1. Introduction

To efficiently retrieve data, a Database Management System (DBMS) indexes data using access methods (AM) as the most important resource to quickly retrieve data stored on discs. In the early days of DBMS development, these methods were designed to work only with numbers and small strings of characters, which we say correspond to simple domains. The total-ordering property holds for these kinds of data, which allow determining what are the elements that precede the others from any pair of elements. Therefore data from simple domains can be compared using the relational operators ($<$, $\leq$, $>$ and $\geq$) as well as the ubiquitously applicable equality operators ($=$ and $\neq$).

Nowadays, the types of data stored and handled by DBMS have grown, including more complex data, which do not present the total ordering property. Examples of complex data domains are images, audio, video, geo-referenced information, DNA sequences, large text, time series, etc. Over these types, the traditional indexing methods are not useful and other properties of the data domains must be employed to allow creating indexing structures able to efficiently retrieve them. Similarity queries are well suited to retrieve data of complex types, and are being increasingly employed. They require defining a similarity function, also called a metric, which can measure how much similar two elements are. The set of complex elements of a given domain and a metric are said to define a metric space.

Formally, a metric space is a pair $\langle \mathbb{S}, d() \rangle$, where $\mathbb{S}$ is the data domain and $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ is a distance function that complies with the following three properties:

1. **symmetry:** $d(s_1, s_2) = d(s_2, s_1)$;

*This work has been supported by **CNPq** (Brazilian National Council for Scientific and Technological Development) and **FAPESP** (State of São Paulo Research Foundation) .

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:** $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2), \forall s_1, s_2, s_3 \in \mathbb{S}$

The distance function $d()$ measures the similarity between two elements. Multi-dimensional datasets and a L_p distance function, such as the Euclidean distance (L_2), are special cases of metric spaces.

The two main types of similarity queries over a data set $S \subset \mathbb{S}$ are:

- **Range query -** Rq**:** given a query center $s_q \in \mathbb{S}$ and a maximum query distance ξ, the query $\hat{\sigma}_{(R_q(s_q,\xi))}S$ retrieves every element $s_i \in S$ such that $d(s_i, s_q) \leq \xi$.
- k**-Nearest Neighbor query -** $kNNq$**:** given a query center $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, the query $\hat{\sigma}_{(kNNq(s_q,k))}S$ retrieves the k elements $s_i \in S$ nearest from the query center s_q.

To speed up retrieving complex data, a new class of AM was developed: the Metric Access Methods (MAM), which are well suited to answer similarity queries. They are classified as either static or dynamic methods. The static ones create the data structure at once, based on the full set of elements to be indexed, and further insertions or deletions are not allowed. In contrast, the dynamic methods can be created incrementally, successively inserting the elements one by one. Nonetheless, creating a dynamic data structure at once (even allowing further insertions and deletions) is an interesting operation, as it provides a way to create the initial structure faster than the sequential insertion of the same elements. This operation is called "bulk loading", and this paper proposes a new, improved method to implement it over dynamic Metric Access Methods, using the Slim-tree [Traina Jr. et al. 2002] as a case study.

As it will be shown, the proposed algorithm can improve the data structure creation as well as it can create structures that allow better query performance. It constructs the structure in a top-down fashion based on the sampling approach, and creates height-balanced trees, with reduced node overlapping.

The remainder of this paper is structured as follows: Section 2 presents the basic and fundamental concepts and summarizes the related works. The proposed algorithm is presented in Section 3, Section 4 describes the experiments performed, which show that the results outperform the traditional sequential insertion method. Finally, Section 5 gives the conclusion of this paper and suggestions for future works.

2. Fundamental Concepts

In this section, we present an overview of the main access methods types: the traditional ones based on the total ordering property, the spatial and the metric access ones. We highlight how they handle the insertion operations, focusing on bulk loading and the three existing techniques: sort, buffer and sample based.

2.1. Traditional access methods

An AM is a resource used by a DBMS to improve the performance of retrieval operations. Numbers, dates and small character strings hold the total ordering property. Therefore, maintaining the stored elements sorted allows each comparison operation to divide the remaining search space into two sets: those greater and those smaller than a comparison element. Hence, successive comparisons with well-chosen elements allows recursively

pruning the subsets, leading AM to perform search operations with complexity $O(\log(N))$ over the number N of indexed elements, regarding the number of comparisons, the number of disk access (for data stored in disks) and the execution time, when DBMS performs a retrieval operation. The B-trees [Comer 1979] and its derivatives B*-tree and the B$^+$-tree [Johnson and Shasha 1993] are the most common AM for data holding the total ordering property.

Total ordering-based hierarchical AM presents an interesting property: point queries, that is, queries aimed to find out if an element s_q is indexed, require traversing the tree comparing s_q with elements of only one node at each level of the tree, as two sibling nodes are disjoint. Therefore, the cost of a point query over a height-balanced tree of height H is $\Theta(H) = \Theta(\log(N))$. As inserting a point in such a tree has a computational effort equivalent to a point query, creating a tree form scratch sequentially inserting N elements has a computational cost $\Theta(N \cdot \log(N))$. However, if all elements are available at the creation time, the complexity of a bulk-load operation is $O(N \cdot \log(N))$ only at the worst case, dropping to $\Theta(N)$ when the elements are previously sorted [Aggarwal and Vitter 1988]. Therefore, bulk-load is preferable over sequential insertion at several situations, as for example when an index must be recreated over a relation in a DBMS that was maintained synchronized with a previous index recently dropped.

2.2. Multidimensional and metric access methods

The basic techniques employed to build traditional AM are also the basis for the spatial and metric access methods, whereas other properties besides total ordering are in place. However, in multidimensional and metric AM, the regions delimited by sibling nodes can overlap each other, so it is not possible to guarantee that the search path of a point query will be restricted to only one deep-traversal. Therefore, the computational complexity to execute a point query using those AM is $O(N)$ at the worst case, leading to a complexity to create an structure by sequential insertion of $O(N^2)$. Spatial properties are employed to help answering queries over multidimensional datasets, which are embodied by the Multidimensional Access Methods (MdAM). The MdAM are dedicated to answer queries where the notion of a dimensional space exists, answering basically three kinds of queries: topological (queries independent of scale or rotation of the space, such as intersections, continence or adjacency), cardinal (queries that depends on relative positioning, such as those based on angles as at left, northeast, above, etc.) and distance-based (in which a distance function, in general the Euclidean one, is required). One important characteristic of multidimensional domains is their dimensionality, and there are several MdAM created to handle low dimensional (2 or 3 dimensions) medium (between 4 ant 20) and high dimensional domains. The MdAM can be classified also as able to index only points in the multidimensional space, which are called Punctual Access Methods (PAM), or able to index regions of the space, which are called Spatial Access Methods (SAM). A good overview of MdAM was presented in [Gaede and Günther 1998].

However, many data types, such as images, audio segments, time series, long texts, genetic sequences and so on, do not present neither the total ordering property nor the notion of dimensions. As a result, we cannot use any topological, cardinal or relational operators. There is no sense in saying that an image precedes another one, or that a genetic sequence is at the north of another one (unless other attributes attached to

the data element provides additional information). For these types of data, similarity is the most sought information. The Metric Access Methods (MAM) were developed to help answering similarity queries over those kinds of complex data empowered by a metric. A good overview of MAM was presented in [Hjaltason and Samet 2003].

Most of the existing dynamic MAM divide a dataset into regions and chooses an element, called a representative or a center and a maximum covering distance to represent each region. A metric tree node stores the representatives, the covering distance, elements in the covered region, and their distances to the representatives. This enables the structure to be organized hierarchically, forming a tree. When a query is performed, the query center is first compared with the representatives of the root node. The triangular inequality is employed to prune sub-trees, avoiding distance calculations between the query center and the elements in sub-trees that cannot be part of the answer. However, as in the MdAM, more than one node can cover a given region, so it is possible that more than one path need to be traversed to answer a point query. Distance calculations between complex elements can have a high computational cost. Therefore, to achieve good performance, besides the number of disk accesses, a MAM needs to minimize also the number of distance calculations in query operations. The most known and used dynamics MAM in literature are the M-Tree [Ciaccia et al. 1997], the Slim-Tree [Traina Jr. et al. 2000, Traina Jr. et al. 2002], the DBM-Tree [Vieira et al. 2006], and the Omni-family [Traina Jr. et al. 2007]

2.3. Bulk loading techniques

To construct an AM, there are at least two options: sequential insertion and bulk loading. The sequential insertion is the basic operation, available for any dynamic AM, and is a required operation to enable it to be used in a DBMS, as it is required to be possible to insert the elements one at a time in the database. The bulk-load operation is not mandatory, but it can be faster when an index is created over an already populated database or when an index needs to be recreated.

The bulk-load operation is well understood for traditional AM, and widely used in commercial SGBD. In this case, we can sort the data and build the three in a bottom-up fashion [Bercken and Seeger 2001]. This type of operation is known as sort-based bulk-load. Other existing techniques are the buffer-based and the sample-based ones. In a simple way, the buffer-based technique tries to keep the efficiency of the algorithms selecting a subset of the dataset that fits in main memory, employing a buffer-tree technique [Arge 2003]. Its biggest disadvantage is that the performance depends on the input ordering of the elements. Moreover, its objective is only to reduce the number of disk accesses, ignoring the CPU cost [Ciaccia and Patella 1998]. Sample-based bulk-load chooses a sample with size low enough to fit in main memory and builds the structure for the full dataset using estimations provided by the sample to build the full tree [Lang and Singh 2001]. The main problem of this technique is that it relies on the quality of the sample, which at times can lead to bulk loading times higher than that required by a sequential insertion [Bercken and Seeger 2001].

The traditional sort-based bulk-load methods cannot be directly applied to bulk loads MdAM and MAM. Some techniques presented in the literature to bulk load R-trees and its variants [Berchtold et al. 1998] [Arge et al. 2002] [Lin and Su 2004] use space-filling curves to attach an ordering over the space and allow applying a sort-based algorithm. In [Leal 2001], the authors use buffers maintained in main memory to bulk load

a Slim-tree. The buffer corresponding to several sub-trees are swapped until a fitting arrangement leads to a balanced tree. In [Bercken and Seeger 2001] two generic sample-based bulk-load algorithms are described, which can be applied over both MdAM and MAM. The proposed implementations employ an application programming interface that can be constructed for a broad class of index structures, known as GP-tree (Grow and Post trees) and OP-trees (Overlapping Predicate trees). The first algorithm can be applied over most of the tree-based index structures, and the second one over trees based on the concept of node splitting and covering regions overlaps.

Besides these two last generic algorithms, there is another example of a sample-based technique for MAM developed targeting the M-tree [Ciaccia and Patella 1998], known as the M-tree bulk-load algorithm. It can be described as composed of two steps. The first randomly chooses samples as representatives and assigns the remaining elements to each representative. This is made recursively for the subsets associated to each representative, until the subset is small enough to fit in one node. However, the structure generated by this step may have under-filled nodes and is unbalanced, violating the M-tree properties. Therefore, a second step employs two strategies to provide deep-balancing and reduce the under-filled nodes: reassign the elements in under-filled sub-trees to other ones, and split the taller sub-trees, obtaining a number of shorter ones. The roots of the newly obtained sub-trees are inserted in the current sample, replacing the original representatives. This technique can produce a tree with only one representative and a single set of elements in many leaf nodes. When this occurs, the authors suggest repeating the overall process, re-starting from the beginning with a new sampling step. This process can be very expensive, and there are chances that the algorithm never creates a M-tree. When a tree is obtained, it can be better than the tree obtained by the sequential insertion to answer range an k-nearest neighbor queries. However, as the algorithm does not guarantee that a M-tree is produced, it cannot be used as part of a DBMS.

3. Bulk loading Slim-Tree

In this section we propose a bulk-load algorithm specific for dynamic MAM. It constructs the index structure in a top-down fashion, using a sampling technique too. However, distinctly from the M-tree bulk-load algorithm, we do not construct an unbalanced tree trying to balance it thereafter. Instead, our algorithm constructs a balanced tree from the start, limiting the number of elements associated with each element sampled and it always produces a corret tree, as it is described following.

Our main idea considerates that a tree can be described specifying the following three data: the number of nodes, the number of elements at each node, and the list of elements at each node. The existing algorithms works on the data to produces the lists of elements at each node, so it derives the other data as consequences of these lists. Our proposal is to estimate the number of nodes and the number of elements at each node beforehand, and only then work to distribute the elements among the nodes.

The first step of our technique estimates an approximate number of nodes that will compose the final tree. To this intent, we simulate building the tree in a bottom-up fashion, but we really build it in top-down. The parameters used in this step are the number N of elements to be indexed, the maximal capacity C_M of the nodes and the minimum occupancy C_m to be guaranteed. There are three approaches to obtain the

estimate: assuming fixed-size nodes for the whole tree, fixed-size nodes at each level of the three, and bounded-size nodes, as described following.

Fixed-Size bulk loading (FS bulk loading) The approach considering fixed-size nodes for the whole tree can be generated in a top-down or in a bottom-up fashion. The top-down approach tends to produce a tree with many leaf nodes. Suppose $N = 101$ elements need to be inserted into a tree with $C_M = 10$ elements per node. In a top-down way, assigning 10 elements to the root, there will be 10 nodes with 10 elements each in second level and 99 leaf nodes with 1 element and one leaf node with 2 elements, generating a tree with 111 nodes and much free space on the leaf nodes but no space at the upper levels (Figure 1(a)). The same estimate following a bottom-up approach would produce a tree with 11 leaf nodes: 2 nodes with 10 elements and 9 nodes with 9 elements. In the second level it would have 2 nodes: one with 5 and the other with 6 nodes, and the root would have two elements. The bottom-up tree would have only 14 nodes, improving disk access on queries in the tree, as you can see in Figure 1(b).

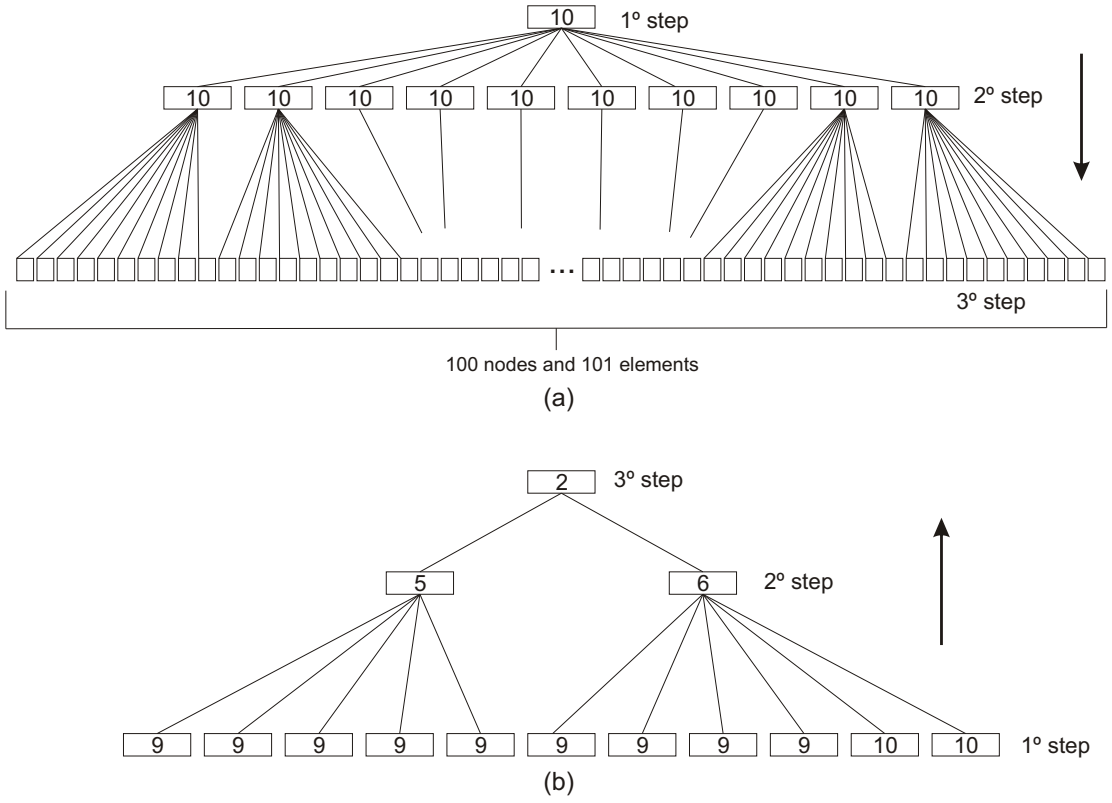

Figure 1. Fixed-Size bulk loading example with $N = 101$ elements inserted and maximum node occupancy $C_M = 10$. In (a) we have a tree built in a top-down fashion with 111 nodes and in (b) a tree built bottom-up with only 14 nodes and the same number of elements in the leaf nodes that we have in top-down tree.

Despite of the little number of nodes, we cannot guarantee that the bottom-up generated tree will have a good performance on queries. In fact, the top-down approach builds the tree always considering the clustering properties when it chooses samples but, when we are building a tree from the bottom, there is no way to foresee how to construct the upper levels with good clustering. That is the reason why the next two bulk-load algorithms build the tree from the top, choosing samples to group the elements.

Level-Fixed-Size bulk loading (LFS bulk loading) The second approach fixes the size of each node regarding its level in the tree. Knowing how much elements per level we have in a bottom-up FS bulk loading, we can fix the number of elements considering this and the level of the tree. The number of nodes in each level is giving by $\eta = \left\lceil \frac{N}{(C_M)^{H-l}} \right\rceil$, where H is the height and l the level of the tree. Consequently, we have the number of elements in each node by level: $\kappa = \frac{T}{\eta}$, where T is the total number of elements to be inserted in each step of the algorithm.

Although this approach leads to a good space utilization and a small number of nodes, it may produce a tree with high node overlapping because we are constructing the tree in a top-down way, with the same number of node elements in each level as in the bottom-up way, we are fixing the number of objects of each node and, for this reason, we are not considering the clustering properties and data distribution of the elements as a whole. This implies that elements in denser regions will follow the same construction parameters as the element in sparser regions.

Bounded-Size bulk loading (BS bulk loading) The previous approaches do not take into account the data distribution. So, the third approach follows the data distribution in the space, grouping more elements in denser and less elements in sparser regions, using a bounding limit to define the number of elements needed to keep the tree balanced. This approach uses the proximity between elements to define their numbers per node, always considering the bounding limit to construct a balanced tree. So, we can improve the performance of the tree finding the nearest element to each sample.

A good choice of samples can lead to a better clustering, improving the tree performance. The bounding limit eliminates the samples with few elements associated. The previous approaches don't take into account this effect, leading to a bad data distribution across the tree. This bounding limit is obtained by the minimum and maximum amount of elements to keep the tree height $H = \lceil \log_{C_M} N \rceil + 1$. For this intent, we use two algorithms: *balancedMin* (Algorithm 1) and *balancedMax* (Algorithm 2).

Algorithm 1 *balancedMin - balancedMin(S,C_m)*

Require: S: elements to be inserted
C_m: minimum node occupancy
Ensure: Returns true if the resulting subtree will unbalance the tree, returns false, otherwise
1: let ℓ be the level of the sub-tree where the elements of S will be inserted
2: $h \leftarrow \lceil \log_{C_M} N \rceil$, this is the height-1 of the future bulk loaded tree
3: $\upsilon \leftarrow \lceil \log_{C_m} \|S\| \rceil$, this is the maximum height-1 of the resulting subtree
4: **if** $\upsilon > h - \ell$ **then**
5: return true
6: **else**
7: return false
8: **end if**

The *balancedMin* and *balancedMax* will return false if the numbers of elements in a set will construct a balanced tree, and true otherwise. These two functions work as a bounding limit to the number of elements inserted in each set associated with a

Algorithm 2 *balancedMax - balancedMax(S,C_M)*

Require: S: elements to be inserted
C_M: maximum node occupancy
Ensure: Returns true if the resulting subtree will unbalance the tree, returns false, otherwise
1: let ℓ be the level of the sub-tree where the elements of S will be inserted
2: $h \leftarrow \lceil \log_{C_M} N \rceil$, this is the height-1 of the future bulk loaded tree
3: $\upsilon \leftarrow \lceil \log_{C_M} \|S\| \rceil$, this is the minimum height-1 of the resulting subtree
4: **if** $\upsilon < h - \ell$ **then**
5: return true
6: **else**
7: return false
8: **end if**

sample. The numbers used by these two functions can be easily obtained: we determine the minimum/maximum height of a sub-tree that can store the given number of elements (line 3) and we compare it (line 4) with the desirable height (line 2). With this information, we can control the maximum and minimum of elements to construct a balanced tree.

The third approach is the one employed in the proposed bulk loading method. It builds a balanced tree in a top-down fashion and does not fix the number of elements per node, using a distance function and a bound limit to group and determine the number of elements per set in each step of the algorithm, which is shown as Algorithm 3.

The algorithm can be divided into three stages: sampling, assignment and refinement. The sampling stage chooses the samples to start the algorithm (Figure 2 (a)). The assignment stage uses the *balancedMax* and proximity between each element of the dataset and the samples to create a set associated with each sample (Figure 2 (b)). The refinement stage reassigns the elements that does not meet the criteria defined by *balancedMin* function (Figure 2 (c) and (d)).

The LFS Bulk-load Algorithm (Algorithm 3) creates the tree receiving as parameters the whole dataset (S), a null value as representative (r) and the maximum (C_M) and minimum (C_m) node occupancy as arguments to the *BulkLoadSlim* algorithm. The algorithm starts randomly choosing elements (samples) in the whole dataset as representatives in a number equal to the maximum node occupancy (line 1 and 4) - sampling stage. Next, it partitions the whole data and associates each subset to a representative (line 5). Each element is associated to the nearest representative (line 8 and 9), respecting the maximum node occupancy to maintain the tree balanced (line 10 to 17) - assignment stage. If a subset do not have the minimum quantity to maintain the tree balanced (line 20) they are discarded and their elements are re-inserted in the others sets, always keeping the tree balanced and the minimal distance between the elements and its representative (line 23) - refinement stage. Thereafter, we create a node with their elements (line 27), choosing a well-fitted representative for them (line 28), so the process can be repeated, constructing the tree recursively (line 30). When we get a null value as representative (line 33), it means this is the root node (line 34) of the tree.

Algorithm 3 LFS Bulk-load Algorithm - *LFSBulkLoad(S,r,C_M,C_m)*

Require: S: set of elements to be bulk loaded
r: representative of the sub-tree
C_M: maximum node occupancy
C_m: minimum node occupancy

Ensure: Insert all elements $i_n \in S$ with the representative r

1: **if** $\|S\| \leq C_M$ **then**
2: insert all elements $i_n \in S$ into a leaf node N and set r as representative of the node
3: **else**
4: choose $k = C_M$ sample elements $s_1 \ldots s_k$ from S
5: create k empty sets $S_1 \ldots S_k$ and associate each set with the respective sample element (representative of set) $L = \{s_1, S_1\} \ldots \{s_k, S_k\}$
6: create a empty set $K = \{\}$
7: **for** $p = 0$ to $\|S\|$ **do**
8: assign i_p to the set S_j, where s_j is the closest sample element to i_p associated with S_j
9: $V \leftarrow \{s_j, S_j\}$
10: **while** $V \in K$ **do**
11: let be $V = \{s_p, S_p\}$, we get the farthest element of S_p and try to insert in other set S_m, where $m \neq p$
12: $V \leftarrow \{s_m, S_m\}$
13: **end while**
14: **if** *balancedMax(S_j,C_M)=true* **then**
15: insert $\{s_j, S_j\}$ into a K set
16: remove $\{s_j, S_j\}$ from L
17: **end if**
18: **end for**
19: **for** $q = 0$ to $\|L\|$ **do**
20: **if** *balancedMin(S_q,C_m)=true* **then**
21: distribute all elements of S_q in the other sets, with the same strategy used in lines 10 to 19
22: **else**
23: insert $\{s_q, S_q\}$ into a K set
24: **end if**
25: remove $\{s_q, S_q\}$ from L
26: **end for**
27: insert $s_1 \ldots s_k$ into a node N
28: set r as representative of the node
29: **for** $m = 0$ to $\|K\|$ **do**
30: *LFSBulkLoad(S_m, s_m, C_M, C_m)*
31: **end for**
32: **end if**
33: **if** r = null **then**
34: define N as root of the tree
35: **end if**

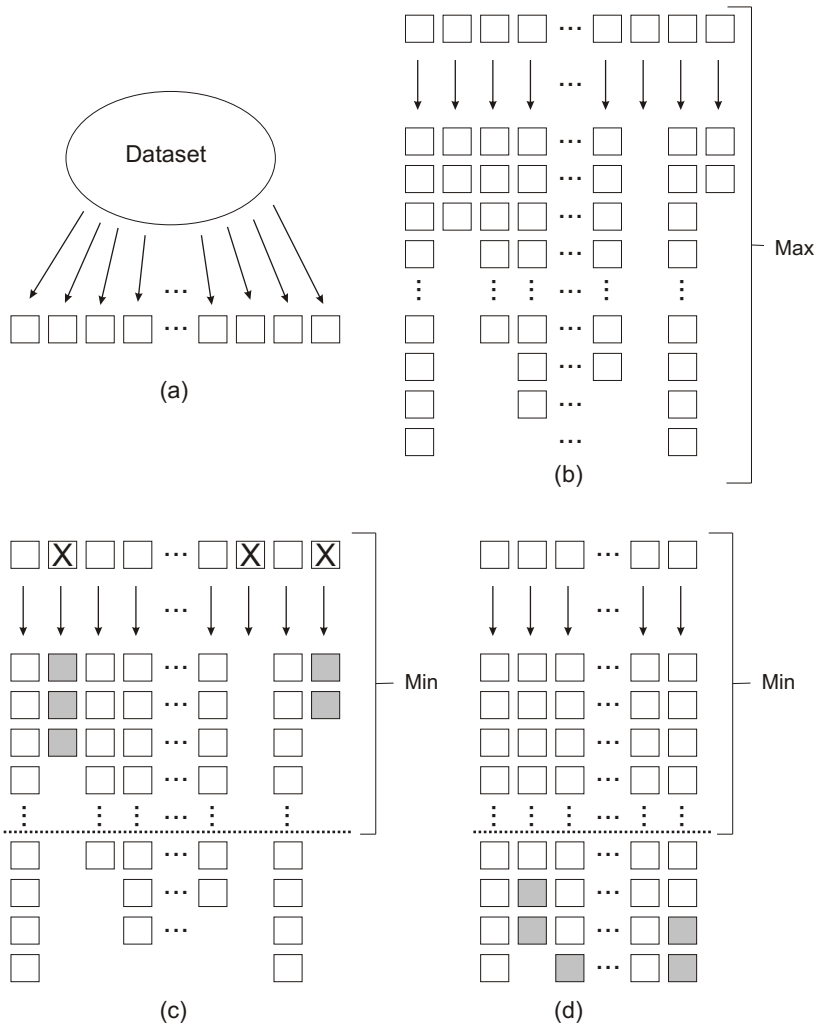

Figure 2. The three stages of the LFS Bulk-load Algorithm: (a) the sampling stage chooses C_M samples; (b) the assignment stage assigns the elements considering the *balancedMax* function as maximum bound limit (Max); (c) the refinement stage reassigns (d) the elements considering a minimum bound limit (Min) using *balancedMin* function.

4. Experimental Evaluation

In this section we report the experimental results obtained by the proposed bulk-load algorithm and compare it with the sequential insertion algorithm available for the M-tree and Slim-tree, and with the M-tree bulk-load algorithm. The LFS bulk-load algorithm was compared with the bulk-load M-tree algorithm because it is the only sample based bulk-load method known in literature dedicated to MAM. The performance evaluation was done using synthetic and real datasets, which are described in Table 1, showing their names, number of elements (#Objs), embedding dimensionality (D), page size (Pg) set to build the corresponding trees, metric ($d()$) and a short description of each set. The maximum node capacity C_M is derived from the page size and the size of the object stored.

The experiments were executed in an Intel Pentium 4 HT 3.0 GHz processor with 2 GB of RAM and 160 GB of disk space. The proposed bulk-load

Name	# Objs.	D	Pg	$d()$	Description
Cities	5,507	2	1KB	L_2	Geographical coordinates of the Brazilian cities (www.ibge.gov.br)
Eigenfaces	11,900	16	4KB	L_2	Informedia Project at Carnegie Mellon University [Wactlar et al. 1996]
MedHisto	4,247	-	4KB	L_M	Metric histograms of medical gray-level images. This dataset was generated at GBDI, ICMC, USP [Traina et al. 2002].
Synt256D	10,000	256	10KB	L_2	Synthetic clustered datasets consisting of 256-dimensional vectors normally-distributed in 20 clusters over the unit hypercube. The process to generate this dataset is described in [Ciaccia et al. 1998].

Table 1. Datasets used in the experiments.

algorithm was implemented usign C++ language in the Arboretum MAM library (http://www.gbdi.icmc.usp.br/arboretum), which contains the Slim-tree and the M-tree algorithms, thus providing a uniform workbench for the experiments. In every test we set the configuration of both the M-tree and the Slim-tree as: *minMax* for the split algorithm; *minDist* for the *ChooseSubtree* algorithm; and 25% as the minimal occupation node capacity C_m. For the Slim-tree we executed the *Slim-Down* algorithm and performed measurements after that.

To measure the query performance we prepared a set 500 elements to be used as queries centers. They were randomly chosen from the original dataset, and 250 of them where removed from the original dataset before creating the tree. Therefore, half of the query centers are indexed, and the other half are not, but all of them are highly probable to occur in real queries. For every measured point in each plot corresponding to each tree we have calculated the average number of disk access, average number of distance calculations and total processing time (in seconds) to perform 500 queries with varying radius ξ or number of elements k with the set of 500 query centers. The Range queries graphics are in log scale for the radius abscissa, to emphasize the relevant part of the graph. The number k for the k-Nearest Neighbor queries varied from 5 to 30 for each measurement, and the radius ξ varied from 0.01% to 0.32% of the dataset diameter.

4.1. Performance Comparison

Comparing the sequential insertion with the LFS bulk loading technique in the Slim-tree (Table 2), the total time to insert 5,257 cities in the Cities dataset using the bulk-load method was 6.3 times faster than sequential insertion. In the Eigenfaces dataset, the LFS bulk-load was 4.88 faster, in the MedHisto dataset was 3.55 faster, and in Synth256D dataset was 1.96 faster than sequential insertion. Considering disk access the LFS bulk-load requires 44 times less disk accesses in Cites dataset, 22.81 times less in EigenFaces, 5.22 times less in MedHisto, and 3.38 times less disk accesses in Synth256D. Bulk loading the cities dataset requires 6.1 times less distance calculations, Eigenfaces requires 3.63 times less, MedHisto requires 1.88 times less, and Synth256D requires 1.79 times

Dataset	Total Time (s)	Disk Access	Distance Calculation
Cities	0.593	36,524	206,683
Cities Bulk-loaded	0.094	830	33,852
Eigenfaces	25.703	96,646	1,594,920
Eigenfaces Bulk-loaded	5.265	4,236	439,134
MedHisto	47.659	79,140	2,845,373
MedHisto Bulk-loaded	13,419	15,153	1,511,716
Synt256D	55,978	99,446	4,463,498
Synt256D Bulk-loaded	28,563	29,446	2,487,495

Table 2. Insertion performance comparison between sequential and LFS bulk loaded insertion in the Slim-tree.

less distance calculations, as compared to the sequential insertion. We do not show the measurements to bulk load the M-tree, as sometimes it is required to start over its construction, because the algorithm do not always produces a good tree.

To evaluate the behavior of the trees created by the sequential insertion and the bulk-load methods to answer queries, we submitted both to a set of queries. Figure 3 shows measurements to answer $kNNq$ queries over the Cities, MedHisto and Synth256D datasets. Figures 3 (a), (d) and (g) shows the average number of disk accesses (for the Cities, MedHisto and Synth256D datasets respectively). It is possible to note that tree created by the LFS bulk-load algorithm reduces up to 41% the number of disk accesses, when comparing with trees created by the sequential insertion technique in the Slim-tree and 28% when compared with the M-tree bulk loading, in the MedHisto dataset.

Figures 3 (b), (e) and (h) shows the average number of distance calculations and Figures 3 (c), (f) and (i) shows the total time to perform the queries (both for the Cities, MedHisto and Synth256D datasets respectively). Comparing the sequential insertion algorithm with the LFS bulk-load algorithm we have almost 34% less distance calculations, and it is up to 41% faster regarding total processing time. Considering the M-tree bulk loading and the LFS bulk loading the latter executed up to 24% less distance calculations and it is 26% faster in processing time. For the Cities dataset the LFS bulk-load algorithm always wins when compared to any other algorithm considering disk access, distance calculations and processing time.

For small radius in the range queries, the LFS bulk-load algorithm has a performance similar to that of a sequential insertions algorithms and to the M-tree bulk-load algorithm, but has better performance as the radius increases, as it can be see in Figure 4. With a radius $\xi = 32\%$ of the diameter, the LFS bulk-load algorithm requires 26% less disk access, 13% less distance calculations and 11% less processing time when compared to the sequential insertion. Comparing the two bulk-load algorithms at 32% of the radius, we have 32% less disk access, 15% less distance calculations and 5% less processing time with the LFS bulk-loading.

5. Conclusions and Future Work

In this paper we propose a fast bulk loading technique for Metric Access Methods, and show an algorithm based on this technique to construct a Slim-tree from scratch. This is the first sample-based bulk-load algorithm for Slim-tree and can it be used in others

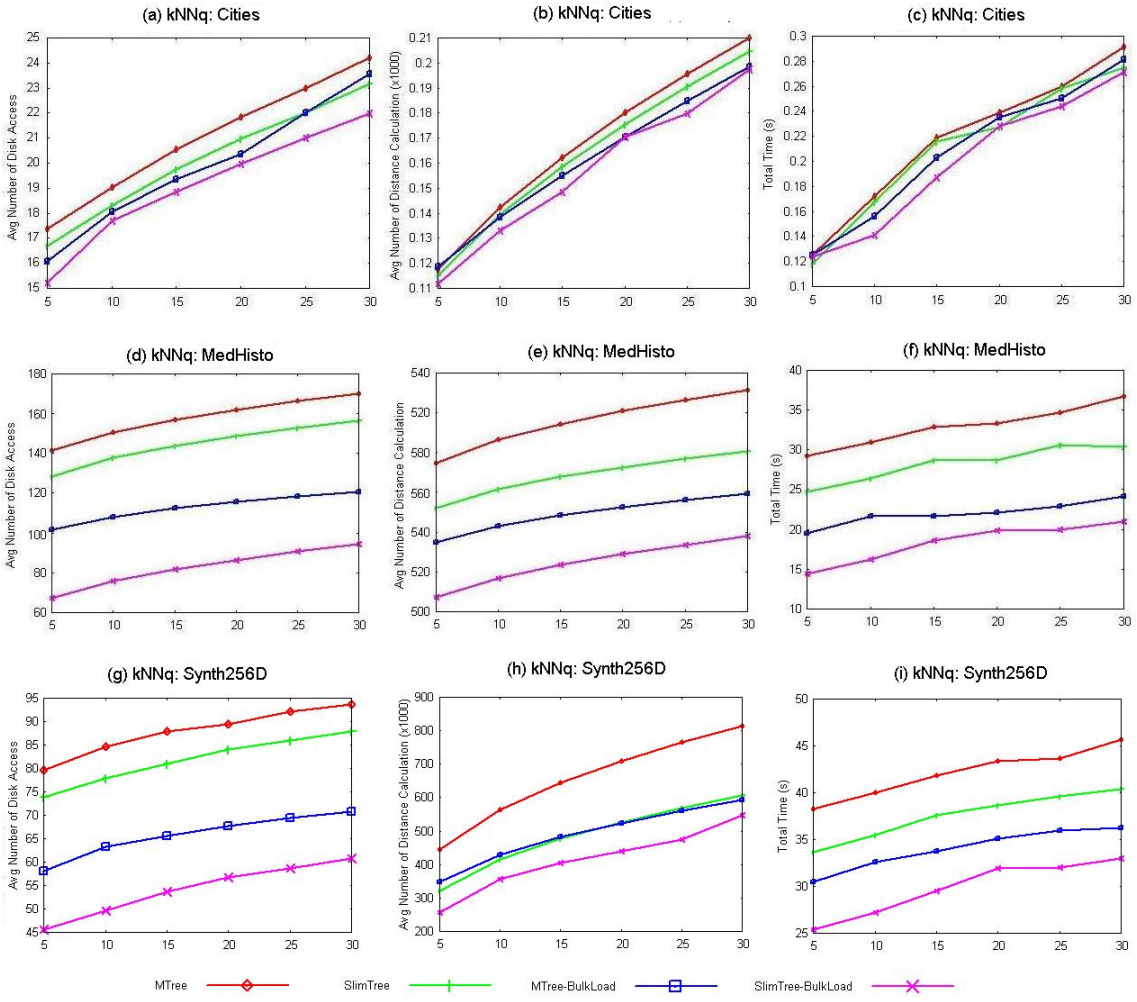

Figure 3. Comparison of average number of disk access (first column), distance calculations (second column) and total processing time in seconds (third column) for $kNNq$ queries performed over trees constructed by the sequential insertion in M-tree and Slim-tree, the M-tree bulk-load algorithm and the LFS bulk-load algorithm to index the Cities (first row), MedHisto (second row) and Synth256D (third row) datasets.

MAM as well. As it always produces a good tree, this is the first bulk-load algorithm for MAM described in the literature that can be included in a DBMS.

The experiments showed that the proposed bulk-load algorithm (LFS bulk loading) achieves good data clustering and outperforms the sequential insertion method regarding both construction and search performance. In fact, the LFS bulk-load algorithm is up to 6 times faster to construct a tree than using the sequential insertion. The trees generated by our proposed algorithm requires up to 34% less distance calculations, and it is 26% faster than the trees generated by the M-tree bulk-load algorithm to answer range queries and requires up to 26% less disk access and 13% less distance calculation, and it is up to 5% faster than M-tree bulk-load algorithm to answer $kNNq$ queries.

Future work includes the development of a method to verify if the samples chosen are the centers of their corresponding cluster and change them accordingly, as well as the

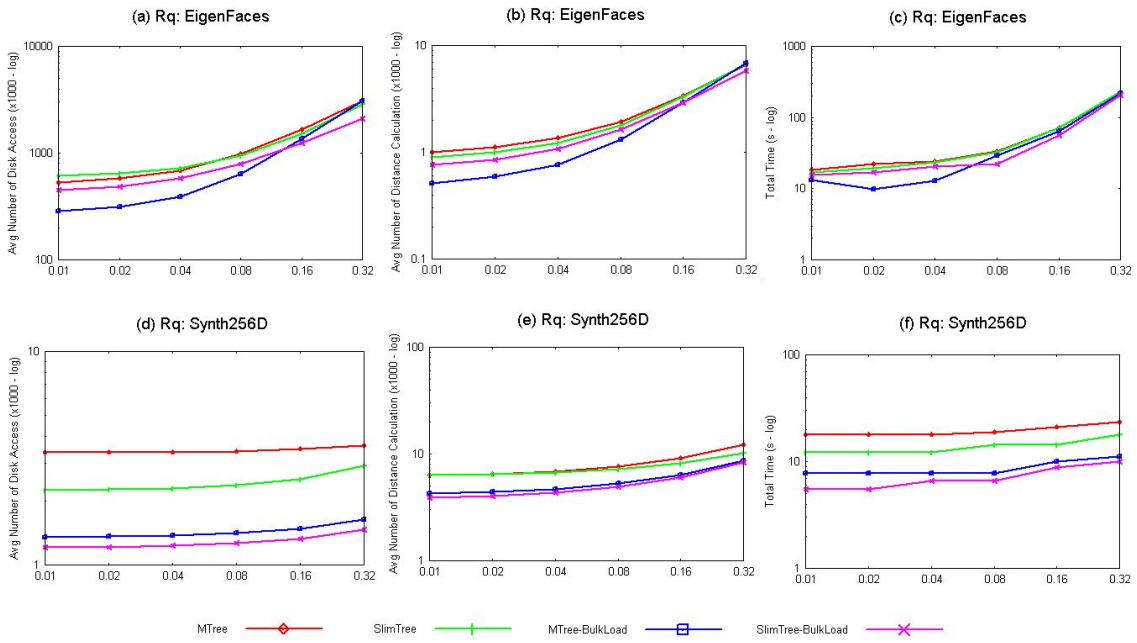

Figure 4. Comparison of the average number of disk access, of distance calculations and total processing time in seconds for range queries performed over trees constructed by the sequential insertion in M-tree and Slim-tree, the M-tree bulk-load algorithm and the LFS bulk-load algorithm to index the Eigenfaces and Synth256D datasets.

development of, and an additional algorithm to use clustering techniques to choose the best samples to bulk load the tree.

References

Aggarwal, A. and Vitter, J. S. (1988). The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127.

Arge, L. (2003). The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24.

Arge, L., Hinrichs, K., Vahrenhold, J., and Vitter, J. S. (2002). Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128.

Berchtold, S., Böhm, C., and Kriegel, H. P. (1998). Improving the query performance of high-dimensional index structures by bulk load operations. *Lecture Notes in Computer Science*, 1377:216–?

Bercken, J. V. d. and Seeger, B. (2001). An evaluation of generic bulk loading techniques. *International Conference on Very Large Databases (VLDB)*, pages 461–470.

Ciaccia, P. and Patella, M. (1998). Bulk loading the m-tree. *Proceedings of the 9th Australasian Database Conference (ADC 98)*, pages 15–26.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*, pages 426–435.

Ciaccia, P., Patella, M., and Zezula, P. (1998). A cost model for similarity queries in metric spaces. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 59–68, New York, NY, USA. ACM Press.

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

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

Hjaltason, G. R. and Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 21(4):517 – 580.

Johnson, T. and Shasha, D. (1993). The performance of current b-tree algorithms. *ACM Transactions on Database Systems (TODS)*, 18(1):51–101.

Lang, C. A. and Singh, A. K. (2001). Modeling high-dimensional index structures using sampling. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):389–400.

Leal, E. (2001). Definition of persistence and building of a bulk loading algorithm for the slim-tree. Master's thesis, Federal University of São Carlos, São Carlos, Brazil.

Lin, B. and Su, J. (2004). On bulk loading tpr-tree. *IEEE International Conference on Mobile Data Management (MDM'04)*, pages 114–124.

Traina, A. J. M., Traina Jr, C., Bueno, J. M., and Marques, P. M. A. (2002). The metric histogram: A new and efficient approach for content-based image retrieval. *IFIP Working Conference on Visual Database Systems (VDB)*.

Traina Jr., C., Santos Filho, R. F., Traina, A. J. M., Vieira, M. R., and Faloutsos, C. (2007). The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *The International Journal on Very Large Data Bases (VLDBJ)*. Avaliable online: http://dx.doi.org/10.1007/s00778-005-0178-0.

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

Traina Jr., C., Traina, A. J. M., Seeger, B., and Faloutsos, C. (2000). Slim-trees: High performance metric trees minimizing overlap between nodes. *International Conference on Extending Database Technology, v. 1777 of Lecture Notes in Computer Science*, pages 51–65.

Vieira, M. R., Traina Jr., C., Chino, F. J. T., and Traina, A. J. M. (2006). Dbm-tree: Trading height-balancing for performance in metric access methods. *Journal of the Brazilian Computer Society*, 11(3):20.

Wactlar, H. D., Kanade, T., Smith, M. A., and Stevens, S. M. (1996). Intelligent access to digital video: Informedia project. *Computer*, 29(3):46–52.