Fast Indexing and Visualization of Metric Data Sets Using Slim-Trees

Caetano Traina Jr., *Member*, *IEEE*, Agma Traina, *Member*, *IEEE Computer Society*, Christos Faloutsos, *Member*, *IEEE*, and Bernhard Seeger, *Member*, *IEEE Computer Society*

Abstract—Many recent database applications must deal with similarity queries. For such applications, it is important to measure the similarity between two objects using the distance between them. Focusing on this problem, this paper proposes the Slim-tree, a new dynamic tree for organizing metric data sets in pages of fixed size. The Slim-tree uses the triangle inequality to prune distance calculations needed to answer similarity queries over objects in metric spaces. The proposed insertion algorithm uses new policies to select the nodes where incoming objects are stored. When a node overflows, the Slim-tree uses a Minimal Spanning Tree to help with the split. The new insertion algorithm leads to a tree with high storage utilization and improved query performance. The Slim-tree is the first metric access method to tackle the problem of overlap between nodes in metric spaces and to propose a technique to minimize it. The proposed "fat-factor" is a way to quantify whether a given tree can be improved and also to compare two trees. We show how to use the fat-factor to achieve accurate estimates of the search performance and also how to improve the performance of a metric tree through the proposed "Slim-down" algorithm. This paper also presents a new tool in the arsenal of resources of Slim-tree aimed at visualizing it. Visualization is a powerful tool for interactive data mining and for the visual tracking of the behavior of a tree under updates. Finally, we present a formula to estimate the number of disk accesses in range queries. Results from experiments with real and synthetic data sets show that the new algorithms of the Slim-tree lead to performance improvements. These results show that the Slim-tree outperforms the M-tree up to 200 percent for range queries. For insertion and split, the Minimal-Spanning-Tree-based algorithm achieves up to 40 times faster insertions. We observed improvements up to 40 percent in range queries after applying the Slim-down algorithm.

Index Terms—Metric databases, metric access methods, index structures, multimedia databases, selectivity estimation, similarity search.

1 Introduction

One of the new research directions for database management systems (DBMSs) is the handling of exotic data types, such as images (still or dynamic), sounds in their many formats, hypertexts, protein sequences, fingerprints, etc. These kinds of data are typically multimedia data and it is not simple to state their dimensionality. While there has been a large number of proposals for multidimensional access methods [15], almost none of them are applicable to multimedia databases since they assume that data belong to a multidimensional vector space. However, the data of multimedia databases often are not in vector spaces but in metric spaces. That is, the only information available are the objects and a dissimilarity function stating the distance between the objects.

This paper addresses the problem of designing efficient metric access methods (MAM). A MAM organizes a large metric data set allowing insertions, deletions, and searches. A metric data set is a set of objects and a distance function $d(\,)$ between two objects x, y. The distance function $d(\,)$ satisfies the three rules of a metric space (symmetry: $d(x,y) = d(y,x)$; nonnegativity: $0 < d(x,y) < \infty, x \neq y \; and \; d(x,x) = 0$; and triangle inequality: $d(x,y) \leq d(x,z) + d(z,y)$). Consequently, a MAM is not permitted to employ primitive operations such as addition, subtraction, or any type of geometric operation. MAMs usually support similarity queries, that is, *range queries* and *nearest neighbor queries*. Nearest neighbor queries ask for the first N nearest objects from a given object, for example, "Select the five nearest stars from the sun." Range queries ask for the objects within a given distance from a given object, for instance, "Find the stars that are within 10 light-years from the sun." In particular, range queries with a radius of zero are called *point queries.*

The efficiency of a MAM is determined by several factors. First, since the data set is generally too large to fit in main memory, one major factor for efficiency is the number of disk accesses required for processing queries and insertions. We assume here that a MAM organizes data in pages of fixed size on a disk and that disk access refers to read (write) one page from a disk into main memory. Second, the computational cost of the distance function can be very high such that the number of distance calculations has a major impact on efficiency. We expect, however, that there is a strong relationship between the number of disk accesses and the number of distance calculations. Third,

- *C. Traina Jr. and A. Traina are with the Department of Computer Science, University of São Paulo at São Carlos, C. Postal 668, 13560-970 São Carlos, SP, Brazil. E-mail: {caetano, agma}@icmc.sc.usp.br.*
- *C. Faloutsos is with the Department of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburg, PA 15213-3891. E-mail: christos@cs.cmu.edu.*
- *B. Seeger is with the Department of Mathematics and Computer Science, University Marburg, Hans-Meerwein-Str., D-35032 Marburg, Germany. E-mail: seegar@Mathematik.Uni-Marburg.de.*

Manuscript received 8 Dec. 1999; revised 21 June 2000; accepted 8 Sept. 2000; posted to Digital Library 25 July 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 111052.

storage utilization is another important factor, although it has rarely been previously considered in this context. The reason we are concerned about storage utilization is not because of the storage cost, but primarily because of the number of disk accesses required to answer "large" range queries. For those queries, the number of accesses is low only when the storage utilization is sufficiently high. In other words, a MAM may be slower than a simple sequential scan for such cases.

The basic structure of metric trees aims to partition the data space in regions using *representatives* or *centers* to which the other objects in each partition will be associated. Each partition has a covering radius and only objects within this radius are associated to the representative. Our proposed Slim-tree is a new dynamic MAM. The Slim-tree, like other metric trees, such as M-tree [11], stores the data in its leaves and creates an appropriate cluster hierarchy on top. The innovations in the Slim-tree are the following: First, based on the Minimal Spanning Tree (MST), a new node splitting algorithm is presented. This algorithm performs faster than other split algorithms without sacrificing search performance. Second, a new algorithm is presented to guide insertion of new objects to an appropriate subtree. In particular, our new algorithm leads to considerably higher storage utilization. Third, and probably the most important, the "Slim-down" algorithm is presented to make the metric tree tighter and faster in a postprocessing step. This algorithm was derived from our findings that high overlap in a metric tree is largely responsible for its inefficiency. Unfortunately, the well-known techniques to measure overlap of a pair of intersecting nodes (e.g., circles in a two-dimensional space) cannot be used for metric data. Instead, we propose the "absolute fat-factor" and the "relative fat-factor" to measure the degree of overlap. It is shown that the Slim-down algorithm reduces the "relative fat-factor" and, hence, improves the query performance of the metric tree.

A preliminary version of this work was presented at EDBT 2000 [28]. Here, we also describe two new tools. The first tool is a visualization algorithm that quickly maps the objects of the Slim-tree onto low-dimensional points, trying to preserve the distances. With this tool, one can visualize how "good" the tree is, as well as do visual data mining (e.g., detecting clusters and outliers). The second tool is a formula that estimates the average number of disk accesses needed to answer range queries. This formula takes advantage of the "absolute fat-factor," so the estimate can be performed on any tree and not only on ideal ones.

The remainder of the paper is structured as follows: In the next section, we first give a brief history of MAMs, including a concise description of the data sets we used in our experiments. Section 3 introduces the Slim-tree, and Section 4 presents its new splitting algorithm based on minimal spanning trees. Section 5 introduces the "absolute fat-factor" and the "relative fat-factor" and Section 6 presents the visualization kit developed for the Slim-tree. The Slim-down algorithm is described in Section 7, while Section 8 presents the development of a formula which allows the estimation of selectivity for range queries. Section 9 presents a performance evaluation of the Slim-tree and Section 10 gives the conclusions of this paper.

2 Survey and Data Sets

The design of efficient access methods has interested researchers for more than three decades. An excellent survey of multidimensional access methods can be found in [15]. However, most of these access methods apply only to a one or multidimensional vector data set.

Metric data sets have attracted the attention of researchers, mainly regarding the problem of supporting nearest neighbor and range queries. The pioneering work of Burkhard and Keller [7] provided other interesting techniques for partitioning a metric data set in a recursive fashion where the recursive process is materialized as a tree. The first technique partitions a data set by choosing a representative from the set and grouping the elements with respect to their distance from the representative. The second technique partitions the original set into a fixed number of subsets and chooses a representative from each of the subsets. The representative and the maximum distance from the representative to a point of the corresponding subset are also maintained. The metric tree of Uhlmann [29] and the Vantage-point tree (vp-tree) of Yanilos [32] are somewhat similar to the first technique of [7] as they partition the elements into two groups according to a representative, called a "vantage point." In [32], the vp-tree has also been generalized to a multiway tree and, in [8], it was further improved through using a better algorithm to choose the vantage points and to answer nearest neighbor queries. In order to reduce the number of distance calculations, Baeza-Yates et al. [1] suggested using the same vantage point in all nodes that belong to the same level. Then, a binary tree degenerates into a simple list of vantage points. Another method of Uhlmann [29] is the generalized hyper-plane tree (gh-tree). The gh-tree partitions the data set into two by picking two objects as representatives and assigning the remaining to the closest representative. Bozkaya and Özsoyoglu [4], [5] proposed an extension of the vp-tree called the "multivantage-point tree" (mvp-tree), which carefully chooses m vantage points for a node which has a fanout of m^2. The Geometric Near Access Tree (GNAT) of Brin [6] can be viewed as a refinement of the second technique presented in [7]. In addition to the representative and the maximum distance, it is suggested that the distances between pairs of representatives be stored. These distances can be used to prune the search space using the triangle inequality. Also, an approach to estimate distances between objects using precomputed distances on selected objects of the data set is proposed by Shasha and Wang in [25].

All methods presented above are static in the sense that they do not support insertions and deletions. The M-tree of Ciaccia, et al. [11] overcomes this deficiency allowing further insertions. The M-tree is a height-balanced tree where the data elements are stored in the leaves. An internal node is a set of entries where each entry consists of a pointer to a subtree, the routing object, and the covering radius r of the subtree. Each node is filled with at least c and at most C entries, where $c \leq C/2$. Notice that the M-tree supports insertions similar to R-trees [17]. Experimental results comparing the M-tree and the mvp-tree [9] and the M-tree and the R*-tree [10] are provided, as well as a cost model based on I/O and distance distribution for the M-tree [12].

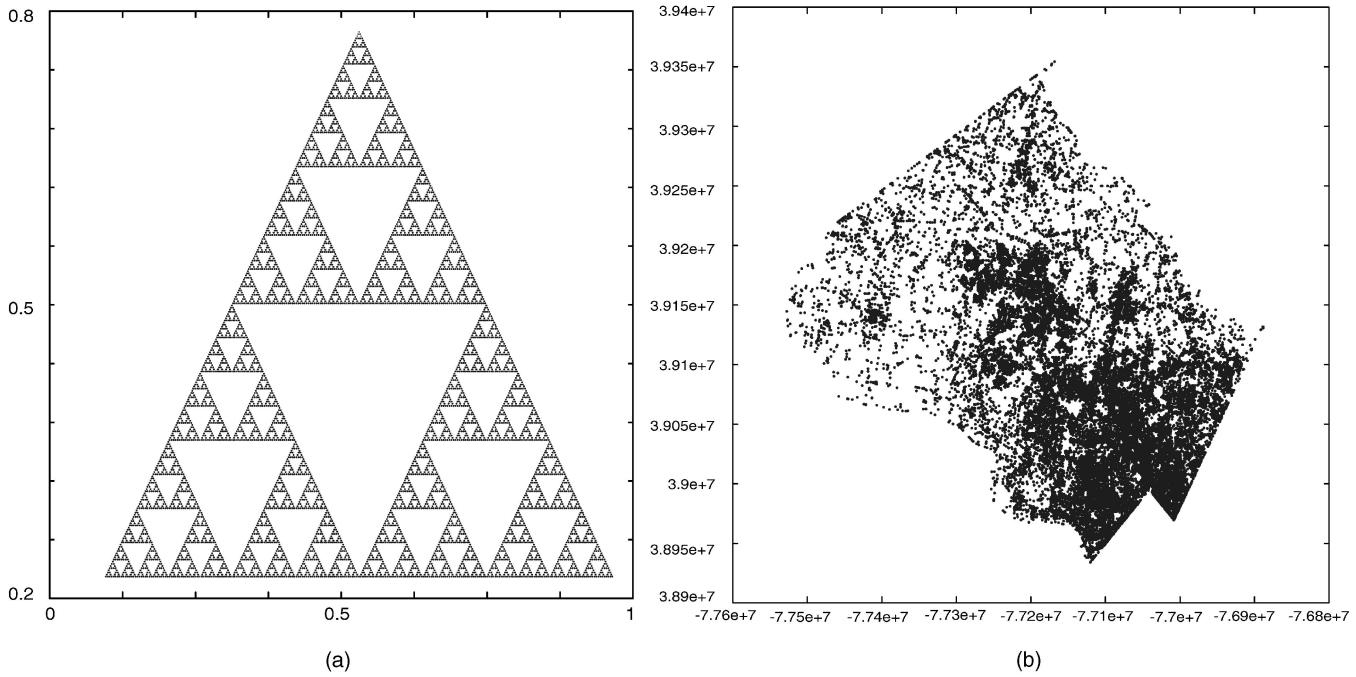

Fig. 1. The two-dimentional vector data sets used in the experiments. (a) Sierpinsky triangle. (b) MGCounty.

The Slim-tree as well as the M-tree are dynamic MAMs in the sense that they allow dynamic insertions. Deletions are not supported by Slim-tree nor by M-tree yet. Nevertheless, deletions can be handled in a similar way as it is in some variations of B+-trees where the objects in the index nodes could be marked as "deleted" [19].

With respect to visualization, there are several tools for visualizing index trees in the literature [15]. An integrated framework for visual debugging of access methods is presented in [24]. In that work, the tool "amdb" allows the user to visualize the structure of the whole tree, the nodes, and even the data distribution. However, it does not support metric data sets. An approach to visualizing a metric set is to use a distance-preserving mapping algorithm. This kind of algorithm takes a data set and a distance function and maps each object in the data set to a point in a k-dimensional space, trying to preserve the distances. There are many distance-preserving mapping algorithms in literature, such as the "FastMap" [14], Cofe [18], which was developed to map protein data sets, and the "Metric-Map" [31], aiming toward RNA data. In this work, we used the "FastMap" algorithm just for visualization because it performs well as a generic mapping algorithm. However, any of the other algorithms could also be employed.

In order to illustrate the performance of the different MAMs, we use six synthetic and real data sets throughout the paper. The distance distribution and intrinsic dimensionality of these data sets are presented in [26]. The data sets are the following:

- "Sierpinsky"—a synthetic set of 106,288 points in a two-dimensional space from the Sierpinsky triangle (see Fig. 1a), a well-known data set, which is self-similar ("fractal").
- "MGCounty"—a set of 15,559 geographical points in a two-dimensional space describing the coordinates of the road intersections in Montgomery County, Maryland (see Fig. 1b).
- "Eigenfaces"—a set of 11,900 face vectors (16-dimensions) from the Informedia Project [30] at Carnegie Mellon University.
- "FaceIt"—a data set constructed with a distance matrix given by the FaceIt™ software, version 2.51. FaceIt is a commercial product from Visionics Corporation and their distance function was *not disclosed*. The set of 1,056 faces that generate this distance matrix was also given by the Informedia Project. Notice that the distance matrix corresponds to a metric function.
- "EnglishWords"—a set of 25,143 objects from the English language dictionary (/usr/share/lib/dict from Unix systems).
- "PortugueseWords"—a set of 429,434 objects from the Portuguese language dictionary [21].

Note especially that, for the FaceIt data set, we have used a distance function from a commercial software product. In general, we have used the L_2 metric for the vector data sets and all of them are normalized in a unit cube. For the "EnglishWords" and "PortugueseWords" data set, the Levenshtein, or string edit distance (L_{Edit}), was used. $L_{Edit}(\mathrm{x}, \mathrm{y})$ is a metric function which counts the minimal number of symbols that have to be inserted, deleted, or substituted to transform string x into string y (e.g., L_{Edit} ("head," "hobby") = 4 $\rightarrow$ three substitutions and one insertion). We are using the same unit cost for each operation, although different costs can be associated with each edit operation without any change in our proposed methods.

TABLE 1
Summary of Symbols and Definitions

Symbols	Definitions
$d(x,y)$	distance function between objects x and y
T	metric tree
N	number of objects in the dataset
$\mathcal{D}$	intrinsic dimensionality of the dataset
M	number of nodes in a metric tree
M_h	number of nodes in level h of a metric tree (root is at h=0)
M_{min}	minimal number of nodes for a given metric tree with N objects
H	height of the metric tree
H_{min}	minimal height for an optimal metric tree of N objects
I_C	total number of node accesses required to answer a point query for each object
C	capacity of a metric tree node (maximum number of objects stored in a non-root node)
$fat(T)$	absolute fat-factor for the metric tree T
$rfat(T)$	relative fat-factor for the metric tree T
r_q	radius of a range query q
q	a range query
$DA_T(r_q)$	number of disk accesses for all nodes in a metric tree T for a range query of radius r_q

3 The Slim-Tree: An Improved Performance Metric Tree

The Slim-tree is a balanced and dynamic tree that grows bottom-up from the leaves to the root. Like other metric trees, the objects of the data set are grouped into fixed size disk pages, each page corresponding to a tree node. The objects are stored in the leaves. The main intent is to organize the objects in a hierarchical structure using a *representative* as the center of each minimum bounding region which covers the objects in a subtree. The Slim-tree has two kinds of nodes, data nodes (or leaves) and index nodes. As the size of a page is fixed, each type of node holds a predefined maximum number of objects C. For simplicity, we assume that the capacity C of the leaves is equal to the capacity of the index nodes. Table 1 summarizes the symbols used in this paper.

The leaf nodes hold all objects stored by the Slim-tree, and their structure is

$$leafnode\ [\text{array of} < Oid_i, d(S_i, S_{rep}), S_i >],$$

where Oid_i is the identifier of the object S_i and $d(S_i, S_{rep})$ is the distance between the object S_i and the representative object of this leaf node S_{rep}. The structure of an index node is

$$indexnode\ [\text{array of} < S_i, R_i, d(S_i, S_{rep}), Ptr(TS_i), NEntries(Ptr(TS_i)) >],$$

where S_i keeps the object that is the representative of the subtree pointed by $Ptr(TS_i)$ and R_i is the covering radius of that region. The distance between S_i and the representative of this node S_{rep} is kept in $d(S_i, S_{rep})$. The pointer $Ptr(TS_i)$ points to the root node of the subtree rooted by S_i. The number of entries in the node pointed to by $Ptr(TS_i)$ is held by $Nentries(Ptr(TS_i))$. Fig. 2 graphically shows the Slim-tree structure (Fig. 2a) and an example of a tree with seven "word" objects using the L_{Edit} distance function (Fig. 2b).

The regions that correspond to each node of the Slim-tree can overlap each other. The increase of overlap also increases the number of paths to be traversed when a query is issued, as well as the number of distance calculations to answer queries. The Slim-tree was developed to reduce the overlap between regions in each level.

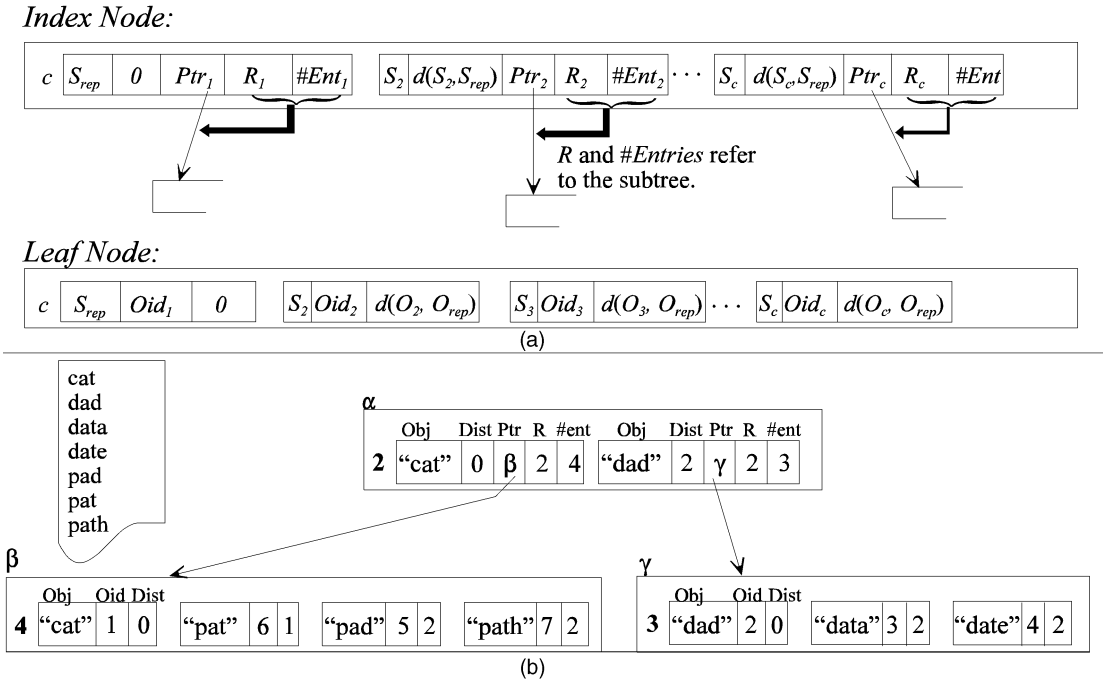

Fig. 2. (a) Graphical representation of the structure on index and leaf nodes of the Slim-tree. (b) An example of a slim-tree storing seven "words" using the L_{Edit} distance functions.

Algorithm 1 - *Splitting of a node using MST strategy*
begin
1. Build the MST on the C objects of the node.
2. Delete the longest edge.
3. Report the connected components as two groups.
4. Choose the representative of each group, i.e., the object whose maximum distance to all other objects of the group is the shortest.

end

Fig. 3. Algorithm for splitting a node using a minimal spanning tree.

3.1 Building the Slim-Tree

Objects are inserted in a Slim-tree in the following way: Starting from the root node, the algorithm tries to locate a node that can cover the new object. If none qualifies, select the node whose center is nearest to the new object. If more than one node qualifies, execute the *ChooseSubtree* algorithm to select one of them. This process is recursively applied for all levels of the tree. When a node m overflows, a new node m' is allocated at the same level and the objects are distributed among the nodes. When the root node splits, a new root is allocated and the tree grows one level.

The Slim-tree provides three options for the *ChooseSubtree* algorithm:

- **random**—Randomly choose one of the qualifying nodes.
- **mindist**—Choose the node that has the minimum distance from the new object and the representative (center) of the node.
- **minoccup**—Choose the node that has the minimum occupancy among the qualifying ones. This is the default method due to its better performance, as we see next.

The number of entries in each child node *(NEntries*) is maintained in its *indexnode.* The field *NEntries* is intended to be used by the *minoccup ChooseSubtree* algorithm. Although it uses some memory space in each *indexnode*, this is usually a small proportion of the total memory used for each entry (one byte is usually enough), but, as we will show later, this *ChooseSubtree* algorithm generated trees with higher node occupation rates, leading to a smaller number of disk accesses. It also helps in the Slim-down algorithm, as presented later.

The splitting algorithms for the Slim-tree are:

- **random**—The two new center objects are randomly selected and the existing objects are distributed among them. Each object is stored in the new node whose center is closest to this object. This is not a wise strategy, but it is very fast.
- **minMax**—All possible pairs of objects are considered as potential representatives. For each pair, a linear algorithm assigns the objects to one of the representatives. The pair which minimizes the covering radius is chosen. The complexity of the algorithm is $\Theta(C^3)$, using $\Theta(C^2)$ distance calculations. This algorithm has already been used for the M-tree and it was found to be the most promising splitting algorithm regarding query performance [10].
- **MST**—The minimal spanning tree [20] of the objects is generated and one of the longest arcs of the tree is dropped. This algorithm is one of the contributions of this paper. It produces Slim-trees almost as good as the minMax algorithm in a fraction of the time.

The default algorithms to build a Slim-tree are: "*minoccup*" for the *Choose Subtree* algorithm, the MST strategy for splits, the node *capacity C* is 60 for vector (L_2 distance) and word data sets (L_{edit} distance) and 25 for FaceIt data set. Next, the new MST split algorithm is described.

4 The Splitting Algorithm Based on Minimal Spanning Tree

This section addresses the following problem. Given a set of C objects in a node to be split, quickly divide them in two groups so that the resulting Slim-tree leads to low search times. We propose a split algorithm based on the minimal spanning tree (MST) which has been successfully used in clustering [20]. We consider the full graph consisting of C objects and $C(C-1)$ edges, where the weight of the edges is the distance between the connecting objects. Thus, we proceed with the steps as shown in Fig. 3. Unfortunately, this algorithm does not guarantee that each group will receive a minimum percentage of objects. To obtain more even distribution, we choose the most appropriate edge from among the longest ones. If none exists (as in a star-shaped set), the uneven split is accepted and the largest edge is removed.

Fig. 4 illustrates our approach applied to a vector space. The node to be split is presented in Fig. 4a. After building the MST (Fig. 4b), the edge between objects A and E will be deleted and one node will keep the objects A, B, C, and D, having B as the representative. The other node will have the objects E, F, G, and H, having F as the representative. Fig. 4c presents the two resulting nodes after the split by the MST approach.

Experiments on real data sets show that the new MST-splitting method is considerably faster than the minMax-splitting method, which has been considered the best for the M-tree [11], [9], while the query performance is minimally affected. For a node with capacity C, the runtime of the minMax-splitting method is $O(C^3)$, whereas the runtime of the MST-splitting method is $O(C^2\ log\ C)$. The performance difference is reflected in our experiments, where the time to build a Slim-tree using the MST-splitting method is much faster than using the minMax-splitting method. Both splitting methods result in Slim-trees with almost the same query performance. We observed that the query performance suffered a little by using the MST-splitting method, but only for small node capacities (less than 20).

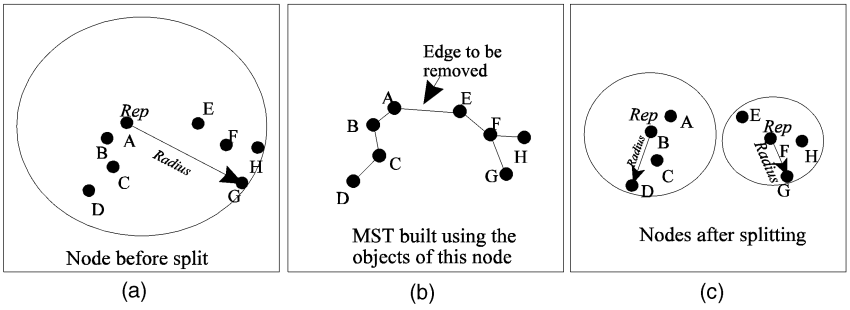

Fig. 4. Exemplifying a node split using the MST algorithm.

5 Overlap Optimization

In this section, we present the theoretical underpinnings behind the Slim-down algorithm. The Slim-down algorithm is an easy-to-use approach to reduce overlaps in an existing Slim-tree. Before presenting the algorithm, we have to define the meaning of overlap in a metric space. Obviously, the notion of overlap in a vector space cannot be applied to a metric space [10].

When someone estimates the number of distance calculations or disk accesses from an index tree, a typical assumption is that the tree is "good" [13], [26]. That is, the nodes are tight and the overlaps of the nodes are minimal. Real trees are not necessarily "good." The present work directly tackles this issue. So, the major motivation behind this work is to solve the following problem: *"Given N objects organized in a metric tree, how can we express its 'goodness'/ 'fitness' with a single number?"*

We also show that our approach to measuring overlap in a metric space leads to the "absolute fat-factor" and to the "relative fat-factor." Both of these factors are suitable for measuring the "goodness" of the Slim-tree and other metric trees. After discussing the properties of these factors, we will present the Slim-down algorithm.

5.1 Computing Overlap in a Metric Access Method

Let us consider two index entries stored in a node of the Slim-tree. In vector spaces, the overlap of two nodes is the amount of common space which is covered by both of the bounding regions. That is, we simply compute the overlap as the volume of the intersection. Since the notion of volume is not available in a metric space, we pursue a different approach. Instead of measuring the amount of space, we *propose counting the number of objects that are covered by both regions*. Thus, we can state the following definition:

Definition 1. *Let I1 and I2 be two index entries (nodes of the metric tree). The overlap of I1 and I2 is defined as the number of objects in the corresponding subtrees which are covered by both regions, divided by the number of objects in both subtrees.*

This definition provides a generic way to measure the intersection between regions of a metric tree, enabling the use of optimization techniques developed for vector spaces on metric trees.

5.2 The Absolute Fat-Factor

Analogous to Definition 1 of overlap in a metric space, in this section we present a method to measure the "goodness" of a metric tree. The basic premise of the following definition of the absolute fat-factor is that a good tree has very little or, ideally, no overlap between its index entries. Such an approach is compatible with the design goals of index structures, like the R+-tree [23] and the R*-tree [2], which strive to minimize overlap.

Our definition of the absolute fat[1] makes two reasonable assumptions. First, we take into account only range queries to estimate the "goodness" of a tree. This assumption is not restrictive since nearest neighbor queries can be viewed as special cases of range queries [3] and nearest neighbor queries are also implemented on Slim-trees. Second, we assume that the distribution of the centers of range queries follows the distribution of data objects. This seems to be reasonable since we expect the queries to be issued most likely in regions of space where the density of objects is high. Moreover, there is no other alternative in a metric space.

Assuming the above, it is easy to state how an ideal metric-tree should behave. For a point query (a range query with radius zero), the ideal metric-tree requires that one node be retrieved from each level. Thus, the " absolute fat " should be zero. The worst possible tree is the one which requires the retrieval of all nodes to answer a point query. In this situation, the "absolute fat" should be one.

Definition 2. *Let T be a metric tree with height H and M nodes, $M \geq 1$. Let N be the number of objects. Then, the* absolute fat-factor*, or simply the* fat-factor*, of a metric tree T is*

$$fat(T) = \frac{I_C - H^* N}{N} \cdot \frac{1}{(M - H)}, \qquad (1)$$

where I_C denotes the total number of node accesses required to answer a point query for each of the N objects stored in the metric tree. Definitions 1 and 2 lead to the next lemma.

Lemma 1. *Let T be a metric tree. Then, fat(T) returns a value in the range $[0, 1]$. The worst possible tree returns one, whereas an ideal tree returns zero.*

Proof. Let us consider a point query for an object stored in the tree. Such a query has to retrieve at least one node from each level of the tree. In particular, the nodes on the insertion path of the object qualify and are required to be read from disk into memory. A lower limit for I_C (the total number of disk accesses for all point queries) is then H^*N, resulting in an absolute fat of zero. The worst case

1. The "absolute fat-factor" will be used interchangeably with absolute fat and "relative fat-factor" with relative fat.

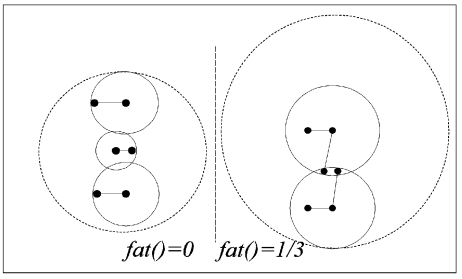

Fig. 5. Two trees storing the same data set with a different number of nodes and absolute fat-factors. Root nodes are indicated by broken lines.

occurs when each node has to be read for each of the queries. An upper limit of I_C is then M^*N, resulting in an absolute fat-factor of one. Since the absolute fat-factor is a linear function in I_C and $H^*N \leq I_C \leq M^*N$, it follows that the absolute fat-factor has to be in the range $[0, 1]$.□

Fig. 5 shows two trees and their absolute fat-factor. In order to illustrate the relationship between the representative and its associated objects, we have drawn a connecting line between them. Calculating the absolute fat for these trees is straightforward, e.g., for the tree in Fig. 5a, we have $I_C = 12$, $H = 2$, $N = 6$, and $M = 4$, leading to an absolute $\text{fat} = 0$. For the tree in Fig. 5b, we have $I_C = 14$, $H = 2$, $N = 6$, and $M = 3$, leading to an absolute $\text{fat} = 1/3$.

5.3 Comparing Different Trees for the Same Data Set: The Relative Fat-Factor

The absolute fat-factor is a measure of the amount of objects that lie inside intersecting regions defined by nodes at the same level of a metric tree. If two trees store the same data set and have the same number of nodes but different absolute fat, the tree with the smaller factor will have fewer points in intersecting regions and, thus, it will need fewer disk accesses and distance calculations to perform a given query. However, if two trees storing the same data set have a *different* number of nodes, the direct comparison of the corresponding absolute fat will not give such an indication. This is due to the fact that a tree with fewer nodes may lead to a tree with more objects lying inside intersection regions and, thus, a bigger absolute fat-factor. However, the average number of disk accesses needed to answer the queries can also be smaller because there are fewer nodes to be read (see Fig. 5).

To enable the comparison of two trees that store the same data set (but that use different splitting and/or different promotion algorithms leading to different trees), we need to "penalize" trees that use more than the minimum required number of nodes (and, so, disk pages). To do that, we propose a new measure, called the "relative fat-factor." In a similar way to the absolute fat-factor, the relative fat considers not the height and number of nodes in the real tree, but that of the minimum tree. Among all possible trees, the minimum tree is the one with the minimum height H_{min} possible and the minimum number of nodes $M_{\min}$. Mathematically, we have:

Definition 3. *The relative fat-factor of a metric tree T with more than one node ($M_{min} > 1$) is defined as*

$$rfat(T) = \frac{I_C - H_{\min}{}^* N}{N} \cdot \frac{1}{(M_{\min} - H_{\min})}. \quad (2)$$

This factor will vary from zero to a positive number that may be greater than one. Although not limited to one, this factor enables the direct comparison of two trees with different relative fat as the tree with the smaller factor will always lead to fewer disk accesses.

The minimum height of a tree organizing N objects is $H_{\min} = \lceil \log_C N \rceil$, and the minimum number of nodes for a given data set can be calculated as $M_{\min} = \sum_{i=1}^{H_{\min}} \lceil N/C^i \rceil$, where C is the capacity of the nodes.

It is worth emphasizing that both the absolute and the relative fat are directly related to the average amount of overlap between regions in the same level of the tree, represented by I_C. The absolute fat measures how good a given tree is with respect to its amount of overlap, regardless of a possible waste of disk space due to lower occupation of its nodes. Relative fat enables us to compare two trees, considering both the number of overlaps *and* efficient occupation of the nodes.

6 Visualization of Slim-Trees

In order to make metric trees more intuitive and practical, this section presents a new tool for the Slim-tree which aims at visualizing it. Visualization is a powerful tool which can help us in interactive data mining (detection of clusters, testing of hypotheses, etc.) and even in visually checking whether one metric tree is better than another. Ideally, we would like to have an algorithm which will operate on a Slim-tree of N objects and will print N points in k dimensions (k is user-defined, e.g., k = 2 or 3). These plotted points should preserve the distances as much as possible. Having plotted the objects, it is useful to see how they are distributed in the nodes of the tree.

Fig. 6a shows a three-dimensional view of the set of English words using the Levenshtein distance function (originally a nondimensional, i.e., metric, space). This figure also shows the quantization effect because the distance function always gives an integer value within a very small range. Fig. 6b shows the same data set overlaid with the bounding regions which involve the leaf nodes of a Slim-tree built over this data set. A shape for the Levenshtein distance function is not apparent as it is nondimensional, so the bounding regions have been depicted as circles, aiming to represent the nodes of the Slim-tree as spheres. The tree was built using the minMax splitting algorithm. Only its leaf nodes are shown. Here, it can be seen that this is not a "good" tree as each bounding region covers many more objects than just its "proper" ones. This occurs even if the real shape of the balls turns out to be, in fact, not a circle at all, but, perhaps, a "star" shape in some dimension.

6.1 Algorithm

The basic algorithm used to visualize a metric tree is based on *FastMap* [14]. This is an iterative process, where the

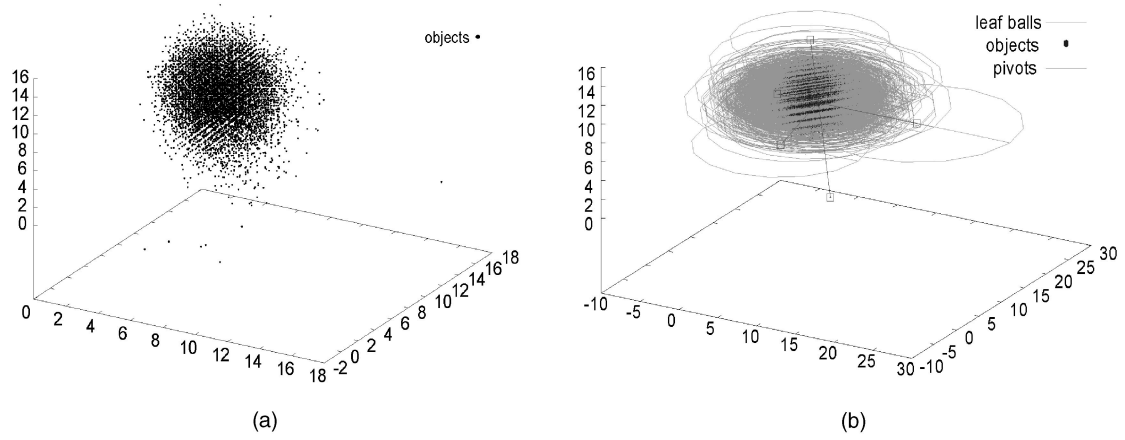

Fig. 6. A three-dimensional view of the English Words data set. (a) The mapping of the data set only. (b) The mapping of the data set, with the pivots of the FastMap Algorithm and the balls of the leaf nodes of a Slim-tree built using the MinMax splitting algorithm, choosing a minimum radius and 60 objects per node.

number of iterations is k (the target number of dimensions). In each iteration, two objects which are far away from each other are chosen to be the "pivots" of the target dimension being generated and both are arbitrarily put in the axis of this dimension, considering only the distance between them. Following that, the projections of all other objects in this axis are calculated by the triangulation of the object and the two pivots. The method maps objects into k-dimensional points, trying to preserve the original distances. We distinguish two entry points for the FastMap library of routines:

1. *FastMap*( ), which receives a data set with N objects, a distance function d( ), and the target number of dimensions. It scans the data set to find k pairs of objects (pivots) that are far from each other.
2. *Coordinate FastMap*( ), which receives an object, a distance function d( ), and the k pairs of pivots. It maps the given object to a point in a k-dimensional space.

This process exhibits three interesting characteristics that suggest its use as part of a visualizing tool for metric trees. First, it is linear on the number of objects $O(N)$, so it is scalable in arbitrarily large data sets. Second, the set of pivots can be stored for reuse in further executions of the algorithm. In this way, objects can be mapped incrementally and labeled, enabling the mapping of the different layers of the tree, different subtrees, and/or different query results, in a unique, consistent, target space. Third, as it preserves the original distances as much as possible, it enables the representation of geometric structures overlaying the object sets, similar to the visualization of the hyper-bounding regions defined by the Slim-tree nodes. To achieve better mapping, it is important that the pivots for each dimension be separated as much as possible. However, finding a pair of points far apart can be the most time consuming part of the FastMap algorithm. Considering that the objects within the set of routing objects of any given nonleaf level of the tree contain objects naturally distant from each other, we decided to choose the anchors from the routing objects of the first level of a tree that has at least 50 objects. The selection of pairs of objects which are far apart is restricted to a much smaller number, and this speeds up the process. The complete algorithm used to build the visualization can be seen in Fig. 7.

6.2 Usage of Visualization for Debugging and Better Design

We show here an example of the visualization of Slim-trees. Additional figures are presented in the following sections. Fig. 8 shows the Sierpinsky data set, overlaid with the circles corresponding to the leaf nodes of the Slim-tree. It shows the points used as pivots of each dimension, too, together with the respective axes generated. This tree was built using the MST splitting algorithm and the *choose minimum occupancy* insertion police. The occupancy C of the nodes is 60. This tree presents the absolute and the relative fat equal to 0.01. In fact, we can visually see that this is a "good" tree.

7 The Slim-Down Algorithm

In this section, we present an algorithm that produces a "tighter" tree. The absolute and relative fat-factors indicate whether a tree has room for improvement. It is clear from Definition 3 that if we want to construct a tree with smaller relative fat, we need first to decrease the number of objects that fall within the intersection of two regions in the same level. Second, we may need to decrease the number of nodes in the tree.

We propose the Slim-down algorithm to postprocess a tree, aiming to reduce these two numbers in an already constructed tree. This algorithm is described in Fig. 9. Fig. 10 graphically illustrates it.

In this way, if the object c was moved from node i to node j in Step 2, and it is the only object in node i at this distance from the original center, then the correction of the radius of node i will reduce the radius of this node without increasing any other radii. As Fig. 10 illustrates, we can assume that object e is the next farthest object from the representative of the node i. Thus, after the reduction, the

Algorithm 2.1 - *VisualizeSlimTree*
input: a built metric tree T, the level l of T to be mapped, the array of pivots (which can be null));
output: the array of pivots;
Begin
1. If the array of pivots is null, execute *FindPivots*
2. Retrieve the objects in level *l* of *T* and, if this level is not the leaf level, the corresponding radius associated with each object
3. Execute *Coordinate FastMap( )* for each object retrieved.
4. For each selected object, plot the point and the corresponding bounding region
5. Return the array of pivots to be re-used in further executions of this algorithm.

end

Algorithm 2.2: *FindPivots*
input: tree T; output: array of pivots;
Begin
1. Count the number of objects stored in each level of *T*
2. Identify the lowest level *i* with at least 50 objects
3. Select all objects of this level *i*
4. execute *FastMap( )* for these selected objects.
5. Store the pivots found in the array of pivots.

end

Fig. 7. The Visualization algorithm for Slim-tree using the *FastMap* method.

new radius of node i will be that shown with a solid line. With this reduction, object c will go out of the region of this node which intersects with the region of node j, reducing I_C counting.

During the execution of this algorithm, the minimum occupancy in the nodes of the tree is not guaranteed, so, eventually, some nodes can become empty (Step 3), further reducing the number of nodes in the tree. After applying the Slim-down algorithm, the nodes that became under-occupied can be removed and its objects reinserted. This approach proved to be quite effective during the slimming down experiments.

A subtle problem may appear in Step 4 if the situation shown in Fig. 11 occurs. In this case, objects f, d, and e will synchronously move from nodes i, j, and k to nodes j, k, and I, respectively, and then again to their original positions. This is illustrated in Fig. 11 by the sets of solid and broken lines. As this can lead to an infinite loop, we limited the number of executions of Step 3, which holds the moving objects to three times the number of objects in the node in the preceding level. The experiments performed indicated this value was a good choice.

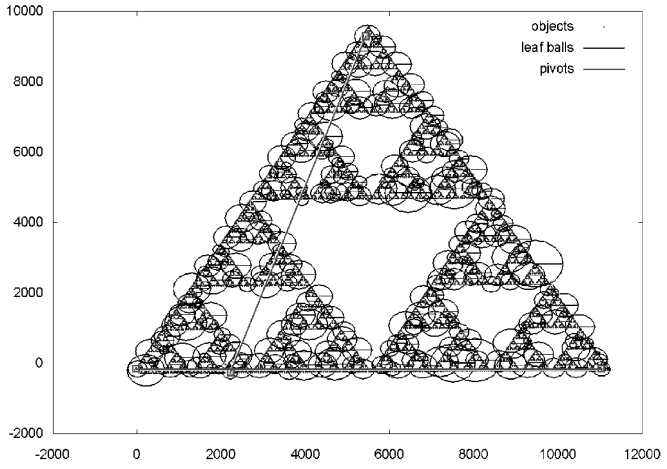

Fig. 8. A view of a Slim-tree indexing the Sierpinsky triangle data set and the pivots of the FastMap algorithm. This tree was built with the MST splitting algorithm, choosing minimum occupancy and a maximum of 60 objects per node (the default parameters for a Slim-tree).

We implemented the algorithm to manipulate the leaf nodes after indexing the full data set. Fig. 12 plots the regions of the leaves of trees created with the Sierpinsky data set using the random splitting algorithm. Fig. 12a shows the tree before the slimming-down and Fig. 12b shows it after correction by the Slim-down algorithm. The regions shown correspond to the minimum bounding circles at the leaf level only (to avoid cluttering). The tree in Fig. 12b clearly has fewer and tighter nodes (circles); therefore, it should perform better. This is confirmed by the relative fat-factor values, which are 0.03 for the tree in Fig. 12a and 0.01 for the tree in Fig. 12b.

The Slim-down algorithm can be executed at many different phases of the evolution of the tree. We list the following variations:

1. As described earlier, the Slim-down algorithm is only applied to the leaves of a full tree.
2. A similar algorithm can be applied to the higher levels of the tree.
3. The algorithm can be dynamically applied for slimming down the subtree stored in a node immediately after one of its direct descendants has been split.
4. When a new object needs to be inserted in a node which is full, a single relocation of the farthest object from one node should be tried instead of splitting.

Notice that introducing the concept of overlap between nodes in metric trees allows the use of optimizations already developed for spatial index structures in metric access methods. For example, Variation 4 corresponds to the shift-split technique already studied for spatial access methods, like R-tree [16]. We believe that Variation 2 should

Algorithm 3 - *Slim-down*
input: a built metric tree T, level h of the tree;
output: an improved metric tree T';
begin
1. For each node i in the level h of the tree, find the farthest object c from the representative b.
2. Find a sibling node j of i, that also covers object c. If such a node j exists and it is not full, remove c from node i and insert it into node j.
3. If node i is not empty then correct the radius of node i.
   else delete node i
4. Steps 1 to 3 must be applied sequentially over all nodes of a given level of the tree. If after a full round of these three steps, an object moves from one node to another, another full round from step 1 to 3 must be re-applied.

end

Fig. 9. The Slim-down algorithm.

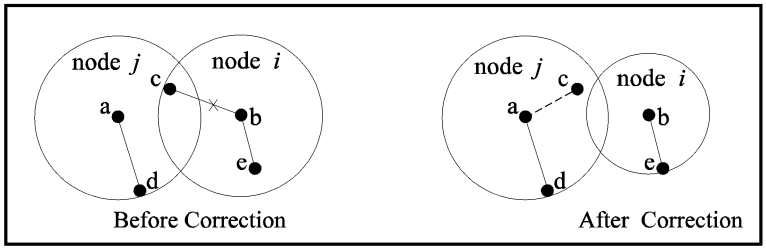

Fig. 10. How the Slim-down algorithm works.

improve the efficiency even more of the tree and it is a promising direction for future research.

The treatment of nodes that become underoccupied after the execution of the Slim-down algorithm depends on the phase where it is executed. In Variation 1, under-occupied nodes are removed and their objects are reinserted. Through the field "number of entries in each child node" *(NEntries)*, this reinsertion operation becomes easier. We implemented and tested Variations 1 and 3 and found that both have similar results. Both variations can be applied isolated or together and we found that, in general, the improvements are cumulative when the variations are applied together. In this paper, we present only the results of Variation 1.

8 A Formula for Estimation of Disk Accesses for Range Queries

The same concept of measures presented in Section 5 will be used in order to predict the number of disk accesses that a range query of radius r_q would require. In [26], a formula was developed to predict the average number of disk accesses for an optimal metric tree and, in the Appendix, the development of this formula is shown. One of the assumptions of that formula is that the expected probability $P_i(0)$ of any point query q_0 to fetch a given node m_i can be measured by taking the (hyper)volume covered by that node divided by the overall volume covered by the metric-tree. In [27] and [22], it was shown that, often, a real data set behaves as a manifold with intrinsic dimensionality $\mathcal{D} \in \mathbb{R}$. Following this idea, a formula that allows estimation of the number of disk accesses (on average) for range queries for an optimal metric index tree is as follows:

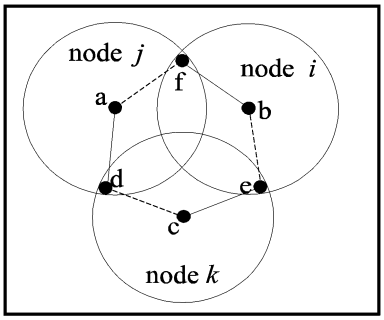

Fig. 11. A cyclic move of objects without reducing radii.

Lemma 2. *For an optimal metric tree, the average number of disk accesses $DA(r)$ on all nodes of the metric tree which are needed to answer any biased range queries of radius r_q can be estimated as:*

$$DA_{Optimal}(r_q) \approx \frac{1}{{r_0}^{\mathcal{D}}} \cdot \sum_{h=0}^{H-1} N^{\frac{h}{H}} \left(\sqrt[\mathcal{D}]{N^{\frac{-h}{H}}} + r_q \right)^{\mathcal{D}}. \quad (3)$$

Proof. See [26]. □

However, this formula assumes that the overlap between nodes is minimal, a typical assumption made when developing a cost function. Because the absolute fat allows measurement of the overlap between nodes for a given metric tree, such optimistic assumptions are not necessary anymore. Moreover, by taking advantage of the absolute fat and estimating the average number of disk accesses for range queries, the following conjecture can be made:

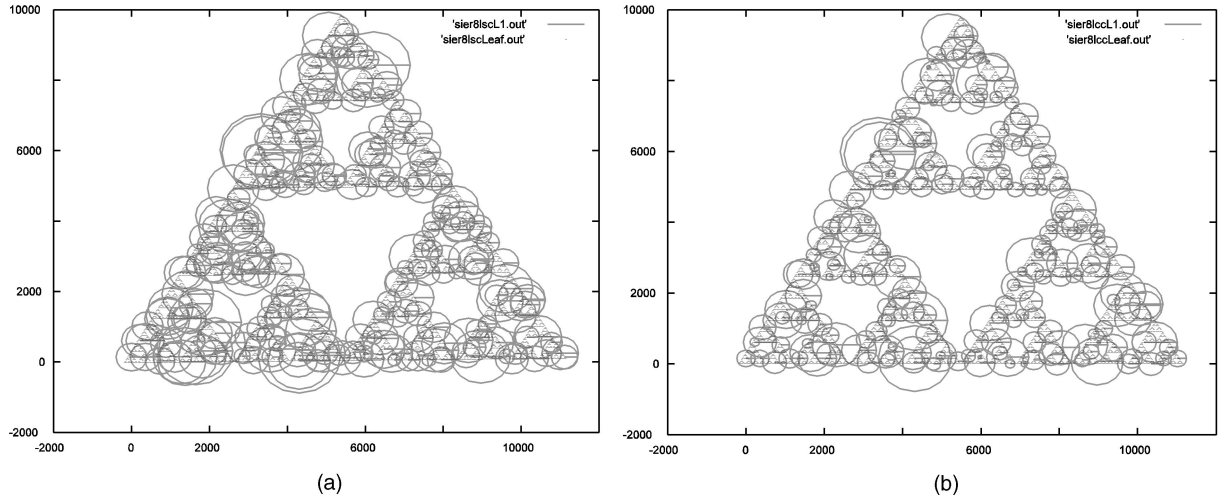

Fig. 12. A tree indexing the Sierpinsky triangle using the random splitting algorithm. (a) Before the correction, the relative fat-factor is 0.03 and (b) after the correction, the relative fat-factor is 0.01. Notice that there are fewer and tighter circles after the slimming-down.

Conjecture. *Given a metric tree T, the number of disk accesses $DA_T(r_q)$ on all nodes of the metric tree which are needed to answer any range query q with covering radius r_q can be estimated as*

$$DA_{\mathrm{T}}(r_q) \approx \frac{1}{{r_0}^{\mathcal{D}}} \cdot \sum_{h=0}^{H-1} (1 + fat(T) \cdot (M^{\frac{h}{H-1}} - 1)) \cdot N^{\frac{h}{H}} \left(\sqrt[\mathcal{D}]{N^{-\frac{h}{H}}} + r_q \right)^{\mathcal{D}}. \quad (4)$$

Justification. The reasoning for this conjecture is as follows: Given that a tree with H levels has M total nodes, it will have an average of $M_h = M^{\frac{h}{H-1}}$ nodes in the h level. When accessing the nodes to answer a query, at most $M_h - 1$ are due to overlap because only a single node corresponds to the correct one. Due to construction, the absolute fat-factor gives the average proportion of the nodes that overlap in each node of the tree. So, the term $fat(T) \cdot (M_h - 1)$ represents, for each level h of the tree, the average number of extra disk accesses that occur due to overlap. Adding one to this number gives the total number of disk accesses for each level h. Thus, modifying (3) with this number to increase the predicted number of disk accesses in each level of the tree will lead to (4).

Equation (4) holds for any tree because the overlap between nodes is quantified by the absolute fat-factor of the tree. When the metric tree is optimal, its absolute fat-factor is zero and (4) turns into (3). When the metric tree is as "bad" as possible (all nodes overlapping), its absolute fat-factor is equal to one. In this case, all nodes must be retrieved in order to answer any range query and this can be seen by the summation on the number of nodes per level M_h in all levels of the tree. Section 9 of this paper presents plots comparing numbers of disk accesses predicted by (4) and the real measurements obtained with real Slim-trees built with an assortment of data sets. Given the average values that are used, as well as the assumptions made to obtain (3), a mathematical proof that (4) is the real one is not available. However, it makes sense as conjecture and the experimental results indicate that it is at least a very close approximation.

9 Experimental Evaluation of the Slim-Tree

This section provides experimental results of the performance of the Slim-tree. The experiments were designed to answer the following questions:

1. How good is the performance of the Slim-tree compared with the M-tree?
2. How good is the performance of the new split method MST compared with the minMax for both insertion and query times?
3. What is the improvement achieved by the Slim-down algorithm?
4. How accurate is the proposed formula for estimating the selectivity for range queries?

We implemented the Slim-tree from scratch in C++ under Windows NT. The experiments were performed on a Pentium II 450MHz PC with 128 MB of main memory. We instrumented our implementation with counters for node accesses and for distance computations. Since the performance of insertions is largely determined by the CPU-time (because of the required distance computations and the complexity of split operations), only the total runtime for creating metric trees is reported in the following. The six previously introduced data sets were used in the experiments. In each case, the metric tree was built by inserting objects one by one. Thereafter, 13 sets of 500 range queries were run for each data set, where the size of the range queries was fixed for each set. In the following graphs, we report the average number of disk accesses and distance calculations obtained from a set of queries as the function of the query size.

9.1 Comparing the Slim-Tree and the M-Tree

Since the M-tree is the only dynamic metric tree available, its alleged best configuration was compared to the correspondent Slim-tree. Figs. 13 and 14 show the query performance of the Slim-tree and the M-tree for the six data sets. Both trees were built using the minMax-splitting algorithm. The corresponding capacities of the nodes used in these experiments are reported in Table 2. Note that, for

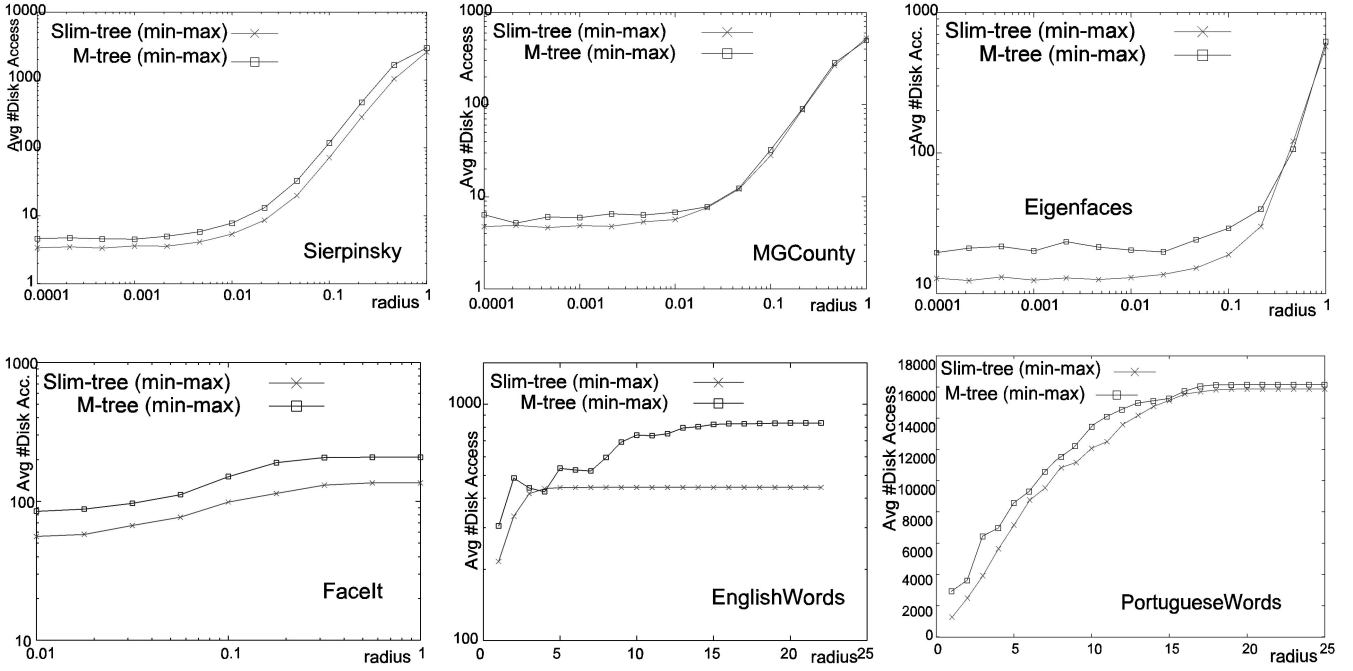

Fig. 13. The query performance (given by an average number of disk accesses) for the M-tree and the Slim-tree as a function of the query radius, where each of the plots refer to one of our data sets.

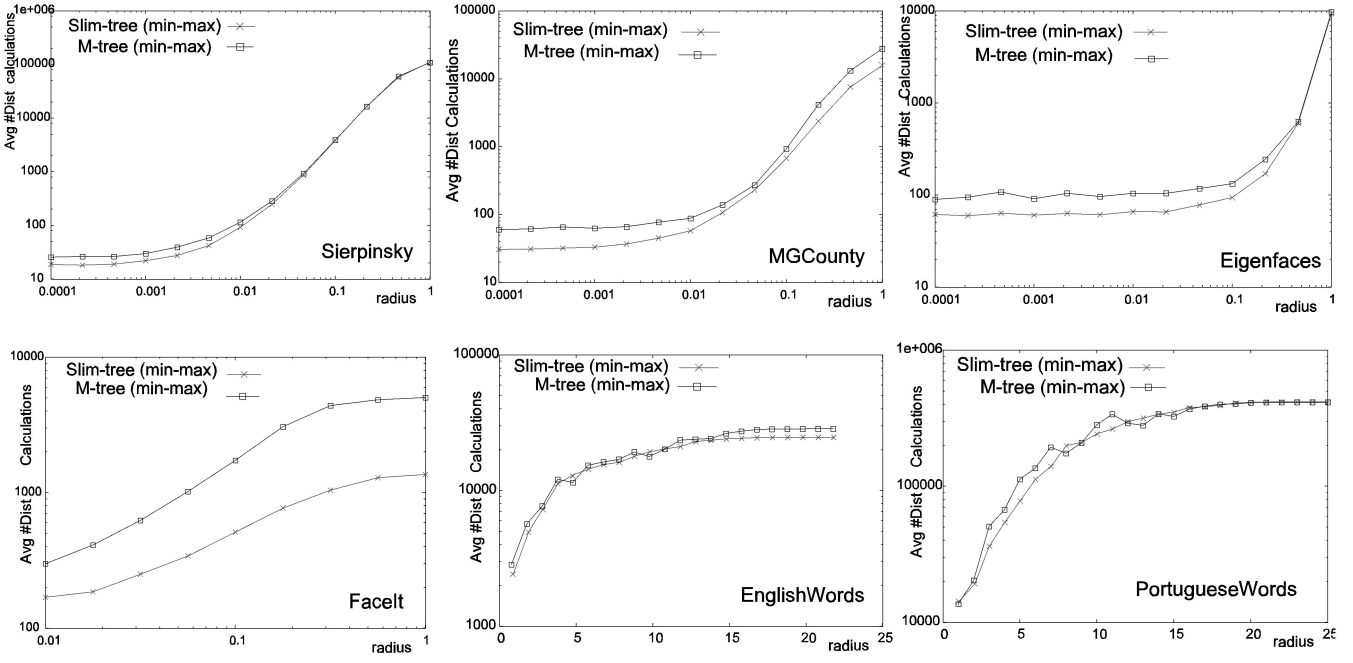

Fig. 14. The query performance (given by an average number of distance calculations) for the M-tree and the Slim-tree as a function of the query radius, where each plot refers to one of our data sets.

both trees, the same settings of the parameters were used, leading to a fair comparison.

Fig. 13 presents the average number of disk accesses for different query radii. For the EnglishWords and Portuguese-Words data sets, we show the results for each possible distance (from one to the maximum number of letters in a word). For the other data sets, we show different radii from 1/10,000 [2] of the maximum data set radius, up to the full data set—we use log scale for these radii to highlight the behavior for different magnitude of radii. The resulting average number of disk accesses is plotted in log scale to minimize the large difference resulting from queries with small and large radii, which makes the comparison easier. Similarly, Fig. 14 shows the average distance calculation for answering the same range queries issued in Fig. 13. Notice that the query objects are randomly taken and are different for each radius, but the same for both trees.

2. For the FaceIt data set, the radius begins at 1/100 because it reaches the top of the resolution of the distance matrix.

TABLE 2
Parameters Used to Compare Slim-Tree with M-Trees

Dataset	Num. of objects N	Objects per node C
Sierpinsky	106,288	52
MGCounty	15,559	52
Eigenfaces	11,900	24
FaceIt	1,056	11
EnglishWords	25,143	60
PortugueseWords	429,434	60

It can be seen from the plots in Fig. 13 that the Slim-tree consistently outperforms the M-tree in number of disk accesses. One of the reasons is that occupation of the nodes is higher for the Slim-tree and, therefore, the total number of nodes is smaller. This effect is visible for the FaceIt and EnglishWords data sets, where a large number of pages is required for the large range queries. For the vector data sets, however, both trees perform similarly for large query radii. This is because the overlap of the entries is low and, therefore, the different insertion strategies of the M-tree and the Slim-tree perform similarly. Note also that, for large query radii, it might be more effective to read the entire file into memory (using sequential I/Os). However, it is common to expect that the majority of queries radii are rather small so that it is beneficial to use a metric tree. Notice that, for distance calculations, both Slim-tree and M-tree perform similarly for the words data sets.

It is noteworthy to compare the most efficient M-tree (as shown in Fig. 13) with the most efficient Slim-tree for the same configuration (after Slim-down). Therefore, Fig. 15 presents a comparison of three trees; an M-tree and a Slim-tree both before and after execution of the Slim-down algorithm. The improvement of the trees is visible and the absolute and relative fat corroborate this. For the Eigenfaces data set, the absolute fat drops from 0.33 to 0.31 and the relative fat from 0.55 to 0.50, after running the Slim-down algorithm. For the EnglishWords data set, the absolute fat drops from 0.48 to 0.38 and the relative fat from 0.53 to 0.39. For the FaceIt data set, the absolute fat drops from 0.42 to 0.41 and the relative fat from 0.69 to 0.59, after running the Slim-down algorithm. Notice that the difference in relative fat represents the difference in query performance between the trees.

9.2 Comparing minMax and MST Splitting Algorithms

Fig. 16 compares the query performance of two Slim-trees, where one uses the minMax-splitting algorithm and the other uses the MST-splitting algorithm. Such Slim-trees were built using the default parameter values (see Section 3.1) and no slimming down was performed. The plots in Figs. 16a and 16b show the results for Sierpinsky and FaceIt, respectively, and show that both Slim-trees perform similarly. Table 3 gives more details about the

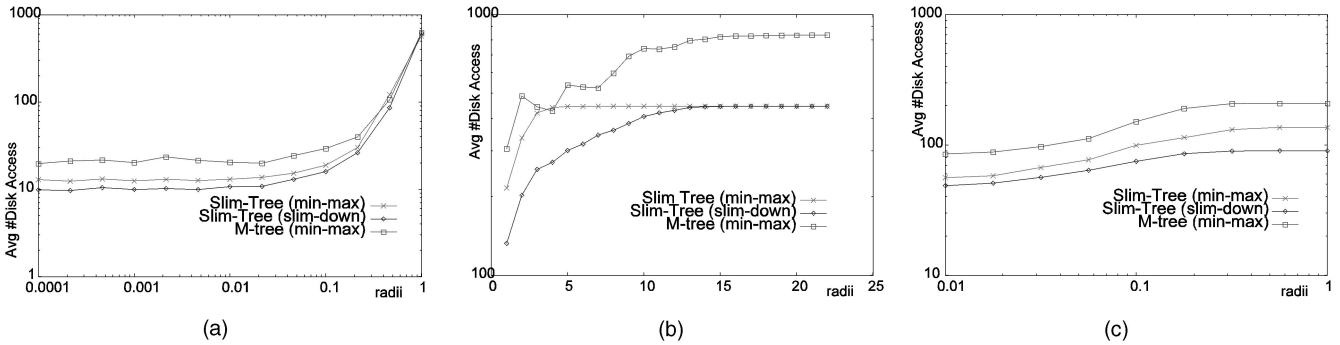

Fig. 15. The query performance for an M-tree, a Slim-tree before slimming down, and a Slim-tree after slimming down. The average number of disk accesses as a function of the query radius are shown. (a) Eigenfaces data set. (b) EnglishWords data set. (c) FaceIt data set.

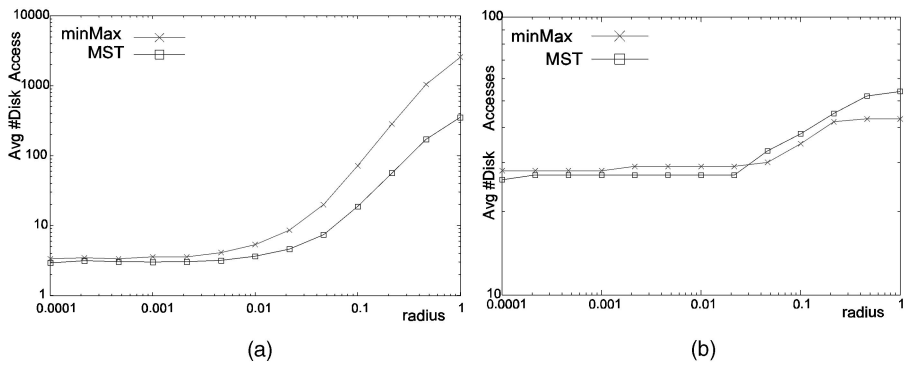

Fig. 16. The query performance of two Slim-trees using the minMax-splitting algorithm and the MST-splitting algorithm. (a) Sierpinsky data set. (b) FaceIt data set.

TABLE 3
A Time Comparison of the Slim-Trees Using the minMax-Splitting Algorithm and the MST-Splitting Algorithm

Datasets	Slim-tree using the minMax-splitting algorithm (wall-clock time in sec.)		Slim-tree using the MST-splitting algorithm (wall-clock time in sec.)	
	build	range queries	build	range queries
Sierpinsky	1,102.49	65.03	38.01	69.52
MGCounty	292.75	20.13	7.79	20.47
Eigenfaces	110.13	45.73	21.42	43.84
FaceIt	12.90	3.35	10.68	3.34
EnglishWords	1,743.64	3,666.05	36.20	3647.41
PortugueseWords	65,739.70	96,192.30	1,700.19	96,576.15

Both trees were built using the default parameters. The numbers are wall-clock times in seconds.

comparison of the different splitting strategies. Here, the columns "range queries" refer to the total wall clock time required to perform the 500 queries for 13 radii (for the EnglishWords data set, we used 22 radii and, for the Portuguese, 25 radii). Note that the MST-splitting strategy suffers slightly when the number of objects per node (the capacity) is small. The columns labeled "build" show the time taken to create the Slim-trees. The MST-algorithm is clearly faster than the minMax-splitting algorithm. For example, the MST-algorithm is faster by a factor of 40 for the two-dimensional data sets. Overall, the MST-splitting algorithm provides considerable savings when a Slim-tree is created, and it gives almost the same performance as the minMax-splitting algorithm for range queries.

The experiments comparing the splitting algorithms show that the runtime of the MST-splitting algorithm is increasingly better than the minMax-splitting algorithm as the number C of entries per node increases. From the results of the experiments, we propose the following rule of thumb: When C (the capacity of the nodes) is lower than 20, it is beneficial to use the minMax splitting strategy; otherwise, use the MST-splitting algorithm. It is also important to mention that the *ChooseSubtree* algorithm also influences the splitting algorithms.

9.3 Experiments with the Slim-Down Algorithm

The Slim-down algorithm improves the number of disk accesses for range queries on average between 10 to 20 percent for vector data sets. These data sets already have a low relative fat, which indicates that there is little room for improvement by using the Slim-down algorithm. For data sets with bigger relative fat, such as the metric data sets FaceIt and EnglishWords, the average improvement is between 25 and 40 percent.

We run the Slim-down algorithm on trees built with the Slim-tree default parameters. Fig. 17 compares the query performance of the Slim-trees before and after running the Slim-down algorithm. Fig. 17a shows the results when the minMax-splitting algorithm is used, whereas the results of the MST-splitting algorithm are presented in Fig. 17b. Both graphs show that the Slim-down algorithm improves the Slim-trees. In general, only a small fraction of the build time is required to slim down a Slim-tree (less than four seconds for all data sets but EnglishWords and PortugueseWords). For the EnglishWords and PortugueseWords data sets, the majority of the time spent is in Step 4 of the Slim-down algorithm (Fig. 9) trying to correct the object migration flaw, which occurs because the distance used provides values in a small range. In this case, the time to slimming down the tree is about 10 percent of the build time.

Although the measurement of I_C takes some computing time, the alternative way to obtain values that represent the performance of a metric tree is to issue several queries for each given radius, accumulate the average number of disk accesses or distance calculations, their standard deviations, and then generate the corresponding plots for many radii. The times spent calculating

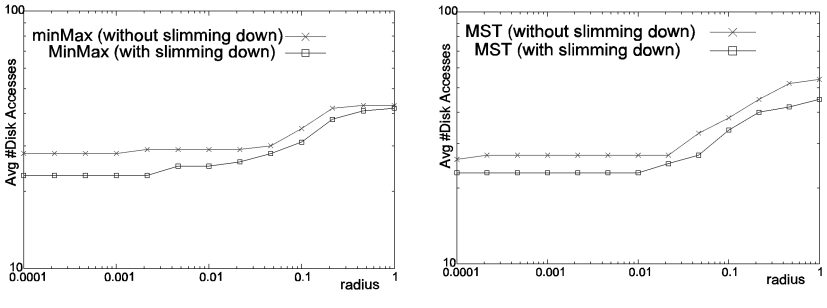

Fig. 17. Comparing the improvements given by the Slim-down algorithm to answer range queries. (a) minMax-splitting algorithm. (b) MST-splitting algorithm.

TABLE 4
Slim-Trees Built Using the MST-Splitting Algorithm with the Default Parameters without Running the Slim-Down Algorithm

Dataset	Num. of objects N	Max. # Objects per node C	Number of nodes M	M_{min}	Height of the tree H	H_{min}	Absolute fat *fat()*	Relative fat *rfat()*
Sierpinsky	106,288	60	3015	1803	4	3	0.01	0.01
MGCounty	15,559	60	516	266	3	3	0.01	0.02
Eigenfaces	11,900	30	602	412	3	3	0.33	0.55
FaceIt	1,056	25	72	46	3	3	0.42	0.69
EnglishWords	25,143	60	586	428	3	3	0.48	0.53
PortugueseWords	429,434	60	12829	7281	4	4	0.11	0.19

both the absolute fat and the average of disk accesses on 500 randomly generated queries was measured. The times to obtain the absolute and relative fat is about 10 percent of the total time required to ask 500 range queries on the trees. Thus, it can be seen that the calculation of the fat-factor is usually much faster than the other alternative.

The last two columns of Table 4 show the absolute fat and the relative fat calculated for the Slim-trees using the MST-splitting algorithm. Moreover, the number of objects, the capacity, the number of nodes, and the height of the tree are also presented. Note that parameter M refers to the actual number of nodes and parameter M_{min} gives the minimum number of nodes. Analogously, parameters H and H_{min} refer to the height of the Slim-tree.

The experiments show that the two splitting algorithms lead to similar search performance, with MST having significantly faster insertions. The experimental results confirmed that the absolute fat-factor is suitable for measuring the quality of a Slim-tree. As a rule of thumb, it seems that a metric tree with an absolute fat between 0 and 0.1 can be considered a "good" tree. Table 4 shows that the Slim-trees for Sierpinsky and MGCounty are indeed "good" trees, but the trees for Eigenfaces, FaceIt, EnglishWords, and PortugueseWords have room for improvement.

9.4 The Accuracy of Our Proposed Selectivity Formulas

Here, the results obtained from the measurement of real queries in a Slim-tree are compared with estimations made using (4). This measurement plots the average number of disk accesses evaluated from the Slim-tree to answer 500 queries with each given radius. Fig. 18 shows the measurements obtained from the six data sets and the respective estimations using (4) versus the query radius. There is no competing method for purposes of comparison. For metric data sets, it is impossible to use even the uniformity assumption.

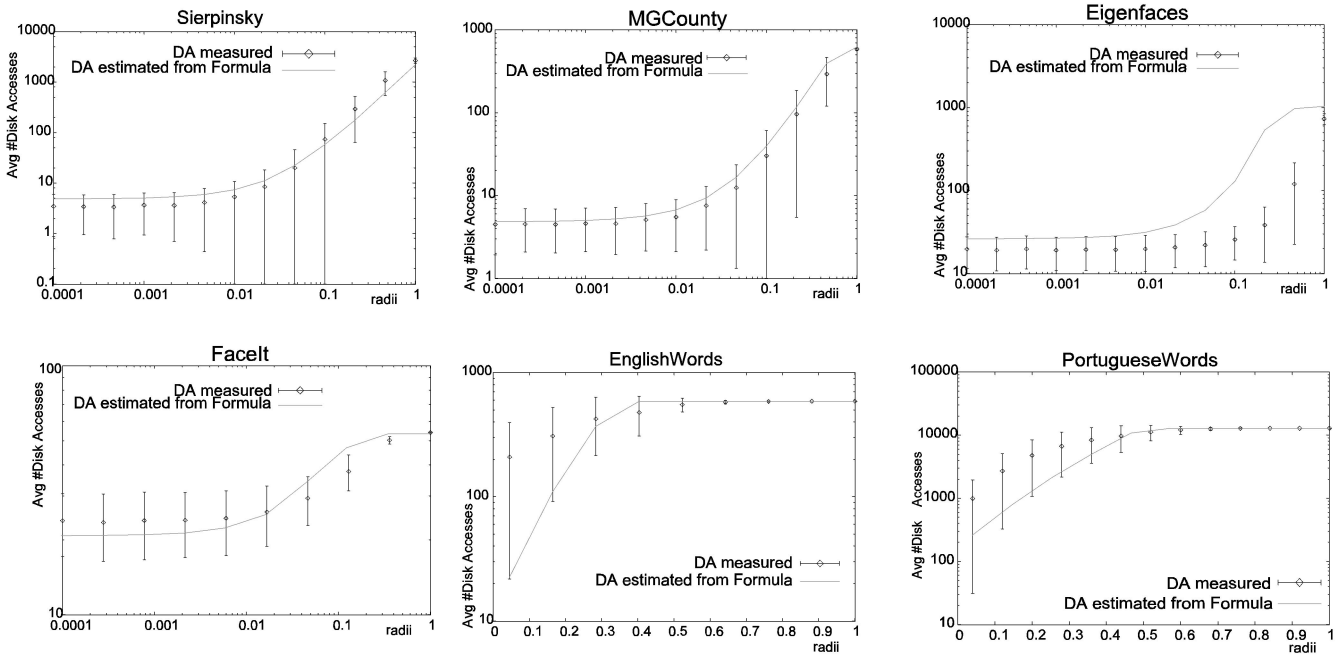

Fig. 18. Slim-trees for the six data sets using the MST splitting strategy. The plots are shown in log-log scales (except for the "EnglishWords" and PortugueseWords data sets in linear-log) comparing the actual number of the disk accesses (bullets) inside one standard deviation (error bars) with the estimated number of disk accesses using the absolute fat-factor (line).

10 Conclusions

We have presented the Slim-tree, a dynamic metric access method which uses new approaches to efficiently index metric data sets. Our main contributions are as follows:

- new measurements, the *absolute* and the *relative fat-factor*, to evaluate the overlap between nodes of a metric tree,
- the Slim-down algorithm which leads to better trees, decreasing the absolute and relative fat-factors proposed,
- a new, significantly faster splitting algorithm based on the minimal spanning tree (MST),
- a formula (4) that allows estimating the selectivity for range queries using only a few parameters from the tree,
- a visualization algorithm that helps the user to understand how metric data is organized internally by the Slim-tree, which can also be used as a tool for data mining on the already stored data,
- a new *ChooseSubtree* algorithm for the Slim-tree (*minoccup*) which leads to tighter trees, fewer disk pages, and faster retrievals.

The Slim-down algorithm is designed for application to a poorly constructed metric tree in order to improve its query performance. The theoretical underpinning of the Slim-down algorithm is our approach for computing overlap in a metric tree. Although overlap is identified as an important tuning parameter for improving query performance for spatial access methods, it has not been previously used for metric trees due to the inability to compute the volume of intersecting regions. In order to overcome this deficiency, we propose using the relative number of objects covered by two (or more) regions to estimate their overlap. This concept is used in the design of the Slim-down algorithm. In this paper, we used the Slim-down algorithm in a postprocessing step, just after the insertion of all objects. This approach does not impede subsequent insertions since it could also be used when objects have yet to be inserted. In our experiments, the Slim-down algorithm improves query performance up to 40 percent. The experiments also show that the Slim-tree outperforms the M-tree up to 200 percent in terms of number of disk accesses when performing range queries.

Our concept of overlap also leads to the introduction of two factors, each of which expresses the quality of a Slim-tree for a given data set using only a single number. The absolute fat-factor measures the quality of a tree with a fixed number of nodes, whereas the relative fat-factor enables a comparison of trees where the number of nodes is different. Moreover, we foresee that the proposed method of treating overlaps in metric spaces allows us to apply to MAM many well-known fine-tuning techniques developed for SAMs. Taking advantage of the absolute fat-factor, we derived a formula to estimate the selectivity for range queries with considerable accuracy, typically within one standard deviation of the actual value for meaningful radius values.

The visualization algorithm developed for the Slim-tree allows graphic perception of how the data is organized, even if the data is in a high-dimensional or metric space. This tool can also be used to help in data mining techniques.

Future work could focus on the use of the absolute and relative fat for bulk-loading, as well as for analysis of other operations, such as the nearest-neighbor queries.

Acknowledgments

The authors would like to thank Pavel Zezula, Paolo Ciaccia, and Marco Patella for giving us the code of the M-tree and for their generous help and advice with our clarification questions. They also wish to thank the anonymous referees for their valuable comments to improve this paper. A preliminary version of this paper appeared in EDBT-2000 [28]. C. Traina Jr.'s work has been funded by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo, Brazil, under Grant 98/05556-5) and he is on leave at Carnegie Mellon University. A. Traina's work has been funded by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo, Brazil, under Grant 98/0559-7). She is also currently on leave at Carnegie Mellon University. B. Seeger's work has been supported by Grant No. SE 553/2-1 from DFG (Deutsche Forschungsgemeinschaft). This material is based upon work supported by the US National Science Foundation under Grant Nos. IRI-9625428, DMS-9873442, IIS-9817496, and IIS-9910606, and by the US Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-97-C-8517. Additional funding was provided by donations from NEC and Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US National Science Foundation, DARPA, or other funding parties.

References

[1] R.A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity Matching Using Fixed-Queries Trees," *Proc. Fifth Ann. Symp. Combinatorial Pattern Matching (CPM)*, pp. 198-212, 1994.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM Int'l Conf. Data Management (SIGMOD)*, pp. 322-331, 1990.

[3] S. Berchtold, C. Böhm, D.A. Keim, and H.-P. Kriegel, "A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space," *Proc. ACM Symp. Principles of Database Systems (PODS)*, pp. 78-86, 1997.

[4] T. Bozkaya and Z.M. Özsoyoglu, "Distance-Based Indexing for High-Dimensional Metric Spaces," *Proc. ACM Int'l Conf. Data Management (SIGMOD)*, pp. 357-368, 1997.

[5] T. Bozkaya and Z.M. Özsoyoglu, "Indexing Large Metric Spaces for Similarity Search Queries," *ACM Trans. Database Systems (TODS)*, vol. 24, no. 3, pp. 361-404, Sept. 1999.

[6] S. Brin, "Near Neighbor Search in Large Metric Spaces," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 574-584, 1995.

[7] W.A. Burkhard and R.M. Keller, "Some Approaches to Best-Match File Searching," *Comm. ACM*, vol. 16, no. 4, pp. 230-236, 1973.

[8] T.-C. Chiueh, "Content-Based Image Indexing," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 582-593, 1994.

[9] P. Ciaccia and M. Patella, "Bulk Loading the M-Tree," *Proc. ADC Australasian Database Conf.*, pp. 15-26, 1998.

[10] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula, "Indexing Metric Spaces with M-Tree," *Proc. Atti del Quinto Convegno Nazionale SEBD*, pp. 67-86, June 1997.

[11] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. Int'l Conf. Very Large Databases (VLDB),* pp. 426-435, 1997.
[12] P. Ciaccia, M. Patella, and P. Zezula, "A Cost Model for Similarity Queries in Metric Spaces," *Proc. ACM Symp. Principles of Database Systems (PODS),* pp. 59-68, 1998.
[13] C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-Trees Using the Concept of Fractal Dimension," *Proc. ACM Symp. Principles of Database Systems (PODS),* pp. 4-13, 1994.
[14] C. Faloutsos and K. Lin, "FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets," *Proc. ACM Int'l Conf. Data Management (SIGMOD),* pp. 163-174, 1995.
[15] V. Gaede and O. Gunther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, 1998.
[16] Y.J. García, M.A. Lopez, and S.T. Leutenegger, "On Optimal Node Splitting for R-Trees," *Proc. Int'l Conf. Very Large Databases (VLDB),* pp. 334-344, 1998.
[17] A. Guttman, "R-Tree: A Dynamic Index Structure for Spatial Searching," *Proc. ACM Int'l Conf. Data Management (SIGMOD),* pp. 47-57, 1984.
[18] G. Hristescu and M. Farach-Colton, "Cluster-Preserving Embedding of Proteins," DIMACS, Technical Report 99-50, 1999.
[19] T. Johnson and D. Shasha, "Utilization of B-Trees with Inserts, Deletes and Modifies," *Proc. ACM Symp. Principles of Database Systems (PODS),* pp. 235-246, 1989.
[20] J.B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proc. Am. Math Soc.,* vol. 7, pp. 48-50, 1956.
[21] R.T. Martins, R. Hasegawa, M.G.V. Nunes, G. Montilha, and O.N. Oliveira, "Linguistic Issues in the Development of ReGra: A Grammar Checker for Brazilian Portuguesse," *Natural Language Eng.,* vol. 4, no. 4, pp. 287-307, Dec. 1997.
[22] B.-U. Pagel, F. Korn, and C. Faloutsos, "Deflating the Dimensionality Curse Using Multiple Fractal Dimensions," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 589-598, 2000.
[23] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. Int'l Conf. Very Large Databases (VLDB),* pp. 507-518, 1987.
[24] M.A. Shah, M. Kornacker, and J.M. Hellerstein, "amdb: A Visual Access Method Development Tool," *Proc. Int'l Workshop User Interfaces to Data Intensive System,* pp. 130-140, 1999.
[25] D. Shasha and T.L. Wang, "New Techniques for Best-Match Retrieval," *ACM Trans. Information Systems,* vol. 8, no. 2 pp. 140-158, 1990.
[26] C. Traina, A.J.M. Traina, and C. Faloutsos, "Distance Exponent: A New Concept for Selectivity Estimation in Metric Trees," Technical Report CMU-CS-99-110, Carnegie Mellon Univ., Pittsburgh, Pa., Mar. 1999.
[27] C. Traina, A.J.M. Traina, and C. Faloutsos, "Distance Exponent: A New Concept for Selectivity Estimation in Metric Trees," *Proc. Int'l Conf. Data Engineering (ICDE),* p. 195, 2000.
[28] C. Traina, A.J.M. Traina, B. Seeger, and C. Faloutsos, "Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes," *Proc. Int'l Conf. Extending Database Technology,* pp. 51-65, 2000.
[29] J.K. Uhlmann, "Satisfying General Proximity/Similarity Queries with Metric Trees," *Information Processing Letter,* vol. 40, no. 4, pp. 175-179, Nov. 1991.
[30] H.D. Wactlar, T. Kanade, M.A. Smith, and S.M. Stevens, "Intelligent Access to Digital Video: Informedia Project," *Computer,* vol. 29, no. 3, pp. 46-52, Mar. 1996.
[31] J.T.-L. Wang, X. Wang, K.-I. Lin, D. Shasha, B.A. Shapiro, and K. Zhang, "Evaluating a Class of Distance-Mapping Algorithms for Data Mining and Clustering," *Proc. ACM Int'l Conf. Knowledge Discovery and Data Mining (KDD),* pp. 307-311, 1999.
[32] P.N. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. Fourth Ann. ACM/SIGACT-SIAM Symp. Discrete Algorithms—SODA,* pp. 311-321, 1993.

Caetano Traina Jr. received the BSc degree in electrical engineering and the MSc and PhD degrees in computer science from the University of São Paulo, Brazil, in 1978, 1982, and 1987, respectively. He is currently an associate professor in the Computer Science Department at the University of São Paulo in São Carlos, Brazil. His research interests include physical database design, data modeling, indexing methods for medical, and multimedia databases. Dr. Traina is a member of the IEEE, SIGMOD, and the SBC.

Agma Traina received the BSc, MSc, and PhD degrees in computer science from the University of São Paulo, Brazil, in 1983, 1987, and 1991, respectively. She is currently an associate professor in the Computer Science Department at the University of São Paulo in São Carlos, Brazil. Her research interests include image databases, indexing methods for multidimensional data, and image processing. Dr. Traina is a member of the IEEE Computer Society, ACM, SIGMOD, and SBC.

Christos Faloutsos received the BSc degree in electrical engineering in 1981 from the National Technical University of Athens, Greece, and the MSc and PhD degrees in computer science from the University of Toronto, Canada, in 1982 and 1987, respectively. From 1985-1997, he was with the Department of Computer Science at the University of Maryland, College Park. He is currently a professor in the Computer Science Department at Carnegie Mellon University in Pittsburgh, Pennsylvania. In 1989, he received the US National Science Foundation Presidential Young Investigator Award. His research interests include physical database design, searching methods for text, geographic information systems, indexing methods for medical and multimedia databases, and data mining. He is a member of the IEEE.

Bernhard Seeger received the MS degree in 1986 from the University of Würzburg and the PhD degree in 1989 from the University of Bremen. He is a professor of database systems in the Institute for Computer Science at the University of Marburg, Germany. His research interests include extensible database systems, generic query processing techniques, and indexing methods. He is a member of the IEEE Computer Society.