Revisiting M-Tree Building Principles

Tomáš Skopal[1], Jaroslav Pokorný[2], Michal Krátký[1], and Václav Snášel[1]

[1] Department of Computer Science
VŠB–Technical University of Ostrava, Czech Republic
{tomas.skopal,michal.kratky,vaclav.snasel}@vsb.cz
[2] Department of Software Engineering
Charles University, Prague, Czech Republic
jaroslav.pokorny@ksi.ms.mff.cuni.cz

Abstract. The M-tree is a dynamic data structure designed to index metric datasets. In this paper we introduce two dynamic techniques of building the M-tree. The first one incorporates a multi-way object insertion while the second one exploits the generalized slim-down algorithm. Usage of these techniques or even combination of them significantly increases the querying performance of the M-tree. We also present comparative experimental results on large datasets showing that the new techniques outperform by far even the static bulk loading algorithm.

Keywords: M-tree, bulk loading, multi-way insertion, slim-down algorithm

1 Introduction

Multidimensional and spatial databases have become more and more important for different industries and research areas in the past decade. In the areas of CAD/CAM, geography, or conceptual information management, it is often to have applications involving spatial or multimedia data. Consequently, data management in such databases is still a hot topic of research. Efficient indexing and querying spatial databases is a key necessity to many interesting applications in information retrieval and related disciplines.

In general, the objects of our interests are spatial data objects. Spatial data objects can be points, lines, rectangles, polygons, surfaces, or even objects in higher dimensions. Spatial operations are defined according to the functionality of the spatial database to support efficient querying and data management. A spatial access method (SAM) organizes spatial data objects according to their position in space. As the structure of how the spatial data objects are organized can greatly affect performance of spatial databases, SAM is an essential part in spatial database systems (see e.g. [12] for a survey of various SAM).

So far, many SAM were developed. We usually distinguish them according to which type of space is a particular SAM related. One class of SAM is based on vector spaces, the second one uses metric spaces. For example, well-known data structures like kd-tree [2], quad-tree [11], and R-tree [8], or more recent ones like UB-tree [1], X-tree [3], etc. are based on a form of vector space. Methods for

L. Kalinichenko et al. (Eds.): ADBIS 2003, LNCS 2798, pp. 148–162, 2003.

indexing metric spaces include e.g. metric tree [14], vp-tree [15], mvp-tree [5], Slim-tree [13], and the M-tree [7].

Searching for objects in multimedia databases is based on the concept of similarity search. In many disciplines, similarity is modelled using a distance function. If the well-known triangular inequality is fulfilled by this function, we obtain metric spaces. Authors of [9] remind that if the elements of the metric space are tuples of real numbers then we get a finite dimensional vector space.

For spatial and multimedia databases there are three interesting types of queries in metric spaces: range queries, nearest neighbours queries, and k-nearest neighbours queries. A performance of these queries differs in vector and metric spaces. For example, the existing vector space techniques are very sensitive to the space dimensionality. Closest point search algorithms have an exponential dependency on the dimensionality of the space (this is called the curse of dimensionality, see [4] or [16]).

On the other hand, metric space techniques seem to be more attractive for a large class of applications of spatial and multimedia databases due to their advantages in querying possibilities. In the paper, we focus particularly on improvement of the dynamic data structure M-tree. The reason for M-tree lies in the fact that, except Slim-trees, it is still the only persistent metric index. In existing approaches to M-tree algorithms there is a static bulk loading algorithm with a small construction complexity. Unfortunately, a querying performance of above-mentioned types of queries is not too high on such tree.

We introduce two dynamic techniques of building the M-tree. The first one incorporates a multi-way object insertion while the second one exploits the generalized slim-down algorithm. Usage of these techniques or even combination of them significantly increases the querying performance of the M-tree. We also present comparative experimental results on large datasets showing that the new techniques outperform by far even the static bulk loading algorithm. By the way, the experiments have shown that the querying performance of the improved M-tree has grown by more than 300%.

In Section 2 we introduce shortly general concepts of the M-tree, discuss the quality of the M-tree structure, and introduce the multi-way insertion method. In Section 3 we repeat the slim-down algorithm and we also introduce here a generalization of this algorithm. Experimental results and their discussion are presented in Section 4. Section 5 concludes the results.

2 General Concepts of the M-Tree

M-tree, introduced in [7] and elaborated in [10], is a dynamic data structure for indexing objects of metric datasets. The structure of M-tree was primarily designed for multimedia databases to natively support the similarity queries.

Let us have a metric space $\mathcal{M} = (D, d)$ where D is a domain of feature objects and d is a function measuring distance between two feature objects. A feature object $O_i \in D$ is a sequence of features extracted from the original database object. The function d must be a metric, i.e. d must satisfy the following metric axioms:

$$\begin{aligned} d(O_i, O_i) &= 0 && && \text{reflexivity} \\ d(O_i, O_j) &> 0 && (O_i \neq O_j) && \text{positivity} \\ d(O_i, O_j) &= d(O_j, O_i) && && \text{symmetry} \\ d(O_i, O_j) + d(O_j, O_k) &\geq d(O_i, O_k) && && \text{triangular inequality} \end{aligned}$$

The M-tree is based on a hierarchical organization of feature objects according to a given metric d. Like other dynamic and persistent trees, the M-tree structure is a balanced hierarchy of nodes. As usually, the nodes have a fixed capacity and a utilization threshold. Within the M-tree hierarchy, the objects are clustered into metric regions. The leaf nodes contain entries of objects themselves (here called the ground objects) while entries representing the metric regions are stored in the inner nodes (the objects here are called the routing objects). For a *ground object* O_i, the entry in a leaf has a format:

$$grnd(O_i) = [O_i, oid(O_i), d(O_i, P(O_i))]$$

where $O_i \in D$ is the feature object, $oid(O_i)$ is an identifier of the original DB object (stored externally), and $d(O_i, P(O_i))$ is a precomputed distance between O_i and its parent routing object.

For a *routing object* O_j, the entry in an inner node has a format:

$$rout(O_j) = [O_j, ptr(T(O_j)), r(O_j), d(O_j, P(O_j))]$$

where $O_j \in D$ is the feature object, $ptr(T(O_j))$ is pointer to a covering subtree, $r(O_j)$ is a covering radius, and $d(O_j, P(O_j))$ is a precomputed distance between O_j and its parent routing object (this value is zero for the routing objects stored in the root). The entry of a routing object determines a metric region in space $\mathcal{M}$ where the object O_j is a center of that region and $r(O_j)$ is a radius bounding the region. The precomputed value $d(O_j, P(O_j))$ is redundant and serves for optimizing the algorithms upon the M-tree. In Figure 1, a metric region and

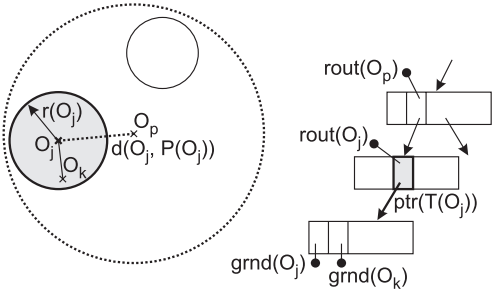

Fig. 1. A metric region and its routing object in the M-tree structure.

its appropriate entry $rout(O_j)$ in the M-tree is presented. For the hierarchy of metric regions (routing objects $rout(O)$ respectively) in the M-tree, only one invariant must be satisfied. The invariant can be formulated as follows:

• All the ground objects stored in the leafs of the covering subtree of $rout(O_j)$ must be spatially located inside the region defined by $rout(O_j)$. •

Formally, having a $rout(O_j)$ then $\forall O \in T(O_j), d(O, O_j) \leq r(O_j)$. If we realize, this invariant is very weak since there can be constructed many M-trees of the same object content but of different structure. The most important consequence is that many regions on the same M-tree level may overlap. An example

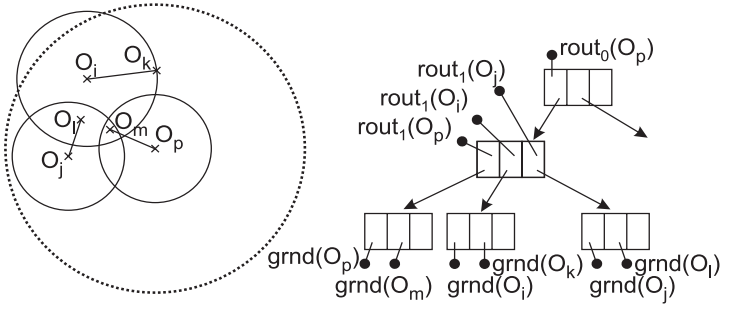

Fig. 2. Hierarchy of metric regions and the appropriate M-tree.

in Figure 2 shows several objects partitioned into metric regions and the appropriate M-tree. We can see that the regions defined by $rout_1(O_p)$, $rout_1(O_i)$, $rout_1(O_j)$ overlap. Moreover, object O_l is located inside the regions of $rout(O_i)$ and $rout(O_j)$ but it is stored just in the subtree of $rout_1(O_j)$. Similarly, the object O_m is located even in three regions but it is stored just in the subtree of $rout_1(O_p)$.

2.1 Similarity Queries

The structure of M-tree natively supports similarity queries. A similarity measure is here represented by the metric function d. Given a query object O_q, a similarity query returns (in general) objects close to O_q. The similarity queries are of two basic kinds: a *range query* and a *k-nearest neighbour query*.

Range Queries. A range query is specified as a *query region* given by a query object O_q and a query radius $r(O_q)$. The purpose of a range query is to return all the objects O satisfying $d(O_q, O) \leq r(O_q)$. A query with $r(O_q) = 0$ is called a *point query*.

***k*-Nearest Neighbours Queries.** A k-nearest neighbours query (k-NN query) is specified by a query object O_q and a number k. A k-NN query returns the first k nearest objects to O_q. Technically, the k-NN query can be implemented

using the range query with a dynamic query radius. In practice, the k-NN query is used more often than the range query since the size of the k-NN query result is known in advance.

By the processing of a range query (k-NN query respectively), the M-tree hierarchy is being passed down. Only if a routing object $rout(O_j)$ (its metric region respectively) intersects the query region, the covering subtree of $rout(O_j)$ is relevant to the query and thus further processed.

2.2 Quality of the M-Tree

As of many other indexing structures, the main purpose of the M-tree is its ability to efficiently process the queries. In other words, when processing a similarity query, a minimum of disk accesses as well as computations of d should be performed. The need of minimizing the disk access costs[1] (DAC) is a requirement well-known from other index structures (B-trees, R-trees, etc.). Minimization of the computation costs (CC), i.e. the number of the d function executions, is also desirable since the function d can be very complex and its execution can be computationally expensive. In the M-tree algorithms, the DAC and CC are highly correlated, hence in the following we will talk just about "costs".

The key problem of the M-tree's efficiency resides in a quantity of overlaps between the metric regions defined by the routing objects. If we realize, the query processing must examine all the nodes the parent routing objects of which intersect the query region. If the query region lies (even partially) in an overlap of two or more regions, all the appropriate nodes must be examined and thus the costs grow.

In generic metric spaces, we cannot quantify the volume of two regions overlap and we even cannot compute the volume of a whole metric region. Thus we cannot measure the goodness of an M-tree as a sum of overlap volumes. In [13], a *fat-factor* was introduced as a way to classify the goodness of the Slim-tree, but we can adopt it for the M-tree as well. The fat-factor is tightly related to the M-tree's query efficiency since it informs about the number of objects in overlaps using a sequence of point queries.

For the fat-factor computation, a point query for each ground object in the M-tree is performed. Let h be the height of an M-tree T, n be the number of ground objects in T, m be the number of nodes, and I_c be the total DAC of all the n point queries. Then,

$$fat(T) = \frac{I_c - h \cdot n}{n} \cdot \frac{1}{(m - h)}$$

is the fat-factor of T, a number from interval $\langle 0, 1 \rangle$. For an ideal tree, the $fat(T)$ is zero. On the other side, for the worst possible M-tree the $fat(T)$ is equal to one. For an M-tree with $fat(T) = 0$, every performed point query costs h disk

[1] considering all logical disk accesses, i.e. disk cache is not taken into account

accesses while for an M-tree with $fat(T) = 1$, every performed point query costs m disk accesses, i.e. the whole M-tree structure must be passed.

2.3 Building the M-Tree

By revisiting the M-tree building principles, our objective was to propose an M-tree construction technique keeping the fat-factor minimal even if the building efforts would increase.

First, we will discuss the dynamic insertion of a single object. The insertion of an object into the M-tree has two general steps:

1. Find the "most suitable" leaf node where the object O will be inserted as a ground object. Insert the object into that node.
2. If the node overflows, split the node (partition its content among two new nodes), create two new routing objects and promote them into the parent node. If now the parent node overflows, repeat step 2 for the parent node. If a root is split the M-tree grows by one level.

Single-Way Insertion. In the original approach presented in [7], the basic motivation used to find the "most suitable" leaf node is to follow a path in the M-tree which would avoid any enlargement of the covering radius, i.e. at each level of the tree, a covering subtree of $rout(O_j)$ is chosen, for which $d(O_j, O) \leq r(O_j)$. If multiple paths with this property exist, the one for which object O is closest to the routing object $rout(O_j)$ is chosen.

If no routing object for which $d(O_j, O) \leq r(O_j)$ exists, an enlargement of a covering radius is necessary. In this case, the choice is to minimize the increase of the covering radius. This choice is thightly related to the heuristic criterion that suggests to minimize the overall "volume" covered by routing objects in the current node.

The single-way leaf choice will access only h nodes, one node on each level, as depicted in Figure 3a.

Multi-way Insertion. The single-way heuristic was designed to keep the building costs as low as possible and simultaneously to choose a leaf node for which the insertion of the object O will not increase the overall "volume". However, this heuristic behaves very locally (only one path in the M-tree is examined) and thus the most suitable leaf may be not chosen.

In our approach, the priority was to choose the most suitable leaf node at all. In principle, a point query defined by the inserted object O is performed. For all the relevant leafs (their routing objects $rout(O_j)$ respectively) visited during the point query, the distances $d(O_j, O)$ are computed and the leaf for which the distance is minimal is chosen. If no such leaf is found, i.e. no region containing the O exists, the single-way insertion is performed.

This heuristic behaves more globally since multiple paths in the M-tree are examined. In fact, all the leafs the regions of which spatially contain the object O are examined. Naturally, the multi-way leaf choice will access more nodes than h as depicted in Figure 3b.

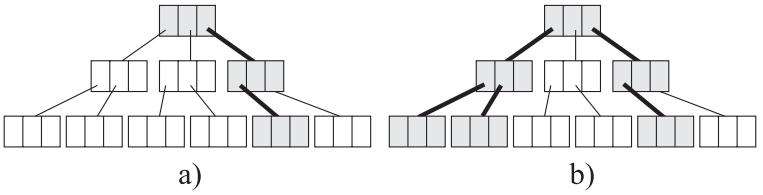

Fig. 3. a) Single path of the M-tree is passed during the single-way insertion. b) Multiple leafs are examined during the multi-way insertion.

Node Splitting. When a node overflows it must be split. According to keep the minimal overlap, a suitable splitting policy must be applied. Splitting policy determines how to split a given node, i.e. which objects to choose as the new routing objects and how to partition the objects into the new nodes.

As the experiments in [10] have shown, the `minMAX_RAD` method of choosing the routing objects causes the best querying performance of the M-tree. The `minMAX_RAD` method examines all of the $\frac{n(n-1)}{2}$ pairs of objects candidating to the two new routing objects. For every such a pair, the remaining objects in the node are partitioned according to the objects of the pair. For the two candidate routing objects a maximal radius is determined. Finally, such a pair $(rout(O_i), rout(O_j))$ for which is the maximal radius (the greater of the two radii $r(O_i), r(O_j)$) minimal is chosen as the two new routing objects.

For the object partition, a distribution according to *general hyperplane* is used as the beneficial method. An object is simply assigned to the routing object that is closer. For preservation of the minimal node utilization a fixed amount of objects is distributed according to the balanced distribution.

2.4 Bulk Loading the M-Tree

In [6] a static algorithm of the M-tree construction was proposed. On a given dataset a hierarchy is built resulting into a complete M-tree.

The basic bulk loading algorithm can be described as follows: Given the set of objects $\mathcal{S}$ of a dataset, we first perform an initial clustering by producing k sets of objects $\mathcal{F}_1, \ldots, \mathcal{F}_k$. The k-way clustering is achieved by sampling k objects $O_{f_1}, \ldots, O_{f_k}$ from the $\mathcal{S}$ set, inserting them in the sample set $\mathcal{F}$, and then assigning each object in $\mathcal{S}$ to its nearest sample, thus computing $k \cdot n$ distance matrix. In this way, we obtain k sets of relatively "close" objects. Now, we invoke the bulk loading algorithm recursively on each of these k sets, obtaining k subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_k$. Then, we have to invoke the bulk loading algorithm one more time on the set $\mathcal{F}$, obtaining a super-tree $\mathcal{T}_{sup}$. Finally, we append each sub-tree $\mathcal{T}_i$ to the leaf of $\mathcal{T}_{sup}$ corresponding to the sample object O_{f_i}, and obtain the final tree $\mathcal{T}$.

The algorithm, as presented, would produce a non-balanced tree. To resolve this problem we use two different techniques:

- Reassign the objects in underfull sets $\mathcal{F}_i$ to other sets and delete corresponding sample object from $\mathcal{F}$.
- Split the taller sub-trees, obtaining a shorter sub-trees. The roots of the sub-trees will be inserted in the sample set $\mathcal{F}$, replacing the original sample object.

A more precise description of the bulk loading algorithm can be found in [6] or [10].

3 The Slim-Down Algorithm

Presented construction mechanisms incorporate decision moments that regard only a partial knowledge about the data distribution. By the dynamic insertion, the M-tree hierarchy is constructed in a moment when the nodes are about to split. However, splitting a node is only a local redistribution of objects. From this point of view, the dynamic insertion of the whole dataset will raise a sequence of node splits – local redistributions – which may lead to a hierarchy that is not ideal.

On the other side, the bulk loading algorithm works statically with the whole dataset, but it also works locally – according to a randomly chosen sample of objects.

In our approach we wanted to utilize a global mechanism of (re)building the M-tree. In [13] a post-construction method was proposed for the Slim-tree, called as *slim-down* algorithm. The slim-down algorithm was used for an improvement of a Slim-tree already built by dynamic insertions. The basic idea of the slim-down algorithm was an assumption that a more suitable leaf exists for a ground object stored in a leaf. The task was to examine the most distant objects (from the routing object) in the leaf and try to find a better leaf. If such a leaf existed the object was inserted to the new leaf (without the need of its covering radius enlargement) and deleted from the old leaf together with a decrease of its covering radius. This algorithm was repeatedly applied for all the ground objects as long as the object movements occured.

However, the experiments have shown that the original (and also cheaper) version of the slim-down algorithm presented in [13] improves the querying performace of the Slim-tree only by 35%.

3.1 Generalized Slim-Down Algorithm

We have generalized the slim-down algorithm and applied it for the M-tree as follows:

The algorithm separately traverses each level of the M-tree, starting on the leaf level. For each node N on a given level, a better location for each of the objects in the node N is tried to find. For a ground object O in a leaf N, a set of relevant leafs is retrieved, similarly like the point query does it by the multi-way insertion. For a routing object O in a node N, a set of relevant nodes (on the appropriate level) is retrieved. This is achieved by a modified range query, where

the query radius is $r(O)$ and only such nodes are processed the routing objects of which *entirely* contain $rout(O)$. From the relevant retrieved nodes a node is chosen the parent routing object $rout(O_i)$ of which is closest to the object O. If the object O is closer to $rout(O_i)$ more than to the routing object of N (i.e. $d(O, rout(O_i)) < d(O, rout(N))$, the object O is moved from N to the new node. If O was the most distant object in N, the covering radius of its routing object $rout(N)$ is decreased. Processing of a given level is repeated as long as any object movements are occuring. When a level is finished the algorithm for the next higher level starts.

The slim-down algorithm reduces the fat-factor of the M-tree via decreasing the covering radii of routing objects. The number of nodes on each M-tree level is preserved since only redistribution of objects on the same level is performed during the algorithm and no node overflows or underflows (and thus node splitting or merging) by the object movements are allowed.

Example (generalized slim-down algorithm):
Figure 4 shows an M-tree before and after the slim-down algorithm application.

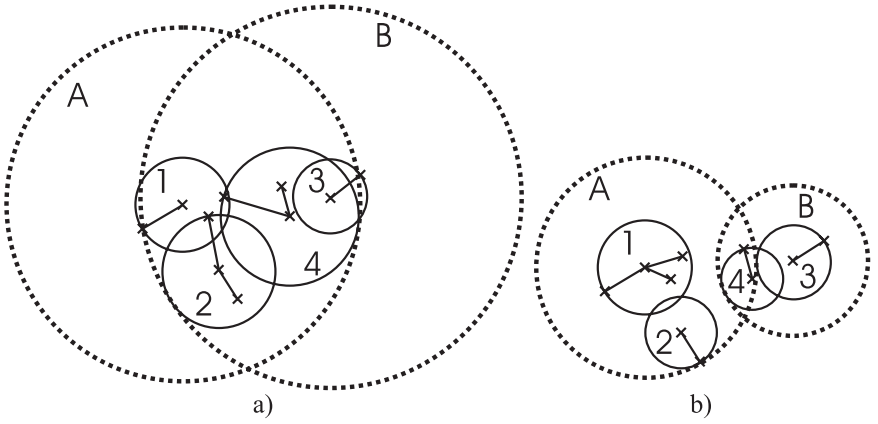

Fig. 4. a) M-tree before slimming down. b) M-tree after slimming down.

Routing objects stored in the root of the M-tree are denoted as A, B while the routing objects stored in the nodes of first level are denoted as 1, 2, 3, 4. In the leafs are stored the ground objects (denoted as crosses). Before slimming down, the subtree of A contains 1 and 4 while the subtree of B contains 3 and 2. After slimming down the leaf level, one object was moved from 2 to 1 and one object was moved from 4 to 1. Covering radii of 2 and 4 were decreased. After slimming down the first level, 4 was moved from A to B, and 2 was moved from B to A. Covering radii of A and B were decreased.

4 Experimental Results

We have completely reimplemented the M-tree in C++, i.e. we have not used the original GiST implementation (our implementation is stable and about 15-times faster than the original one). The experiments ran on an Intel Pentium®4 2.5GHz, 512MB DDR333, under Windows XP.

The experiments were performed on synthetic vector datasets of clustered multidimensional tuples. The datasets were of variable dimensionality, from 2 to 50. The size of dataset was increasing with the dimensionality, from 20,000 2D tuples to 1 million 50D tuples. The integer coordinates of the tuples were ranged from 0 to 1,000,000.

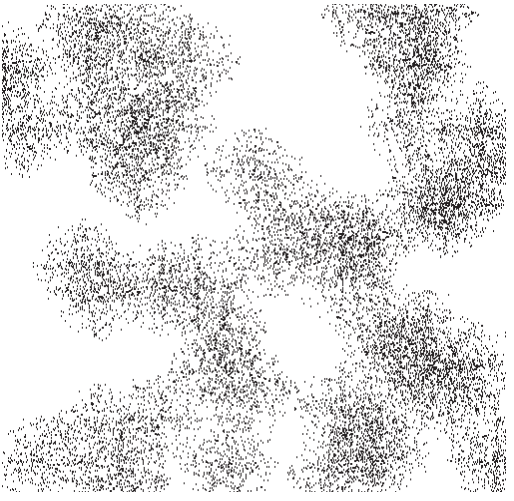

Fig. 5. Two-dimensional dataset distribution.

The data were randomly distributed inside hyper-spherical (L_2) clusters (the number of clusters was increasing with the increasing dimensionality – 50 to 1,000 clusters) with radii increasing from 100,000 (10% of the domain extent) for 2D tuples to 800,000 (80% of the domain extent) for 50D tuples. In such distributed datasets, the hyper-spherical clusters were highly overlapping due to their quantity and large radii. For the 2D dataset distribution, see Figure 5.

4.1 Building the M-Tree

The datasets were indexed in five ways. The single-way insertion method and the bulk loading algorithm (in the graphs denoted as SingleWay and Bulk Loading) represent the original methods of the M-tree construction. In addition to these

Table 1. M-tree statistics.

Metric: L_2 (euclidean)	Node capacity: 20	Dimensionality: 2 – 50
Tuples: 20,000 – 1,000,000	Tree height: 3 – 5	Index size: 1 – 400 MB

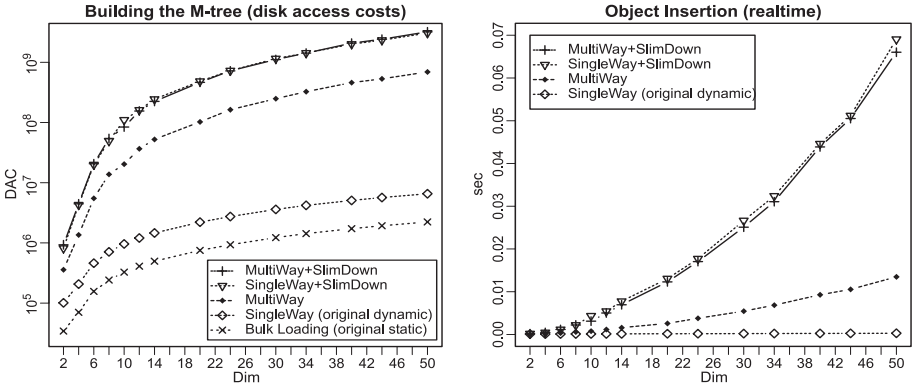

Fig. 6. Building the M-tree: a) Disk access costs. b) Realtime costs per one object.

methods, the multi-way insertion method (denoted as MultiWay) and the generalized slim-down algorithm represent the new building techniques introduced in this article. The slim-down algorithm, as a post-processing technique, was applied on both SingleWay and MultiWay indexes which resulted into indexes denoted as SingleWay+SlimDown and MultiWay+SlimDown. Some general M-tree statistics are presented in Table 1.

The first experiment shows the M-tree building costs. In Figure 6a, the disk access costs are presented. We can see that the SingleWay and Bulk Loading indexes were built much cheaply than the other ones, but the construction costs were not the primary objective of our approach. Figure 6b illustrates the average realtime costs per one inserted object. In Figure 7a, the fat-factor characteristics of the indexes are depicted. The fat-factor of SingleWay+SlimDown and MultiWay+SlimDown indexes is very low, which indicates that these indexes contain relatively few overlapping regions. An interesting fact can be observed from the Figure 7b showing the average node utilization.

The MultiWay index utilization is by more than 10% better than the utilization of the SingleWay index. Studying this value is not relevant for the SingleWay+SlimDown and MultiWay+SlimDown indexes since the "slimming-down" does not change the average node utilization, thus the results are the same as those achieved for SingleWay and MultiWay.

4.2 Range Queries

The objective of our approach was to increase the querying performace of the M-tree. For the query experiments, sets of query objects were randomly selected

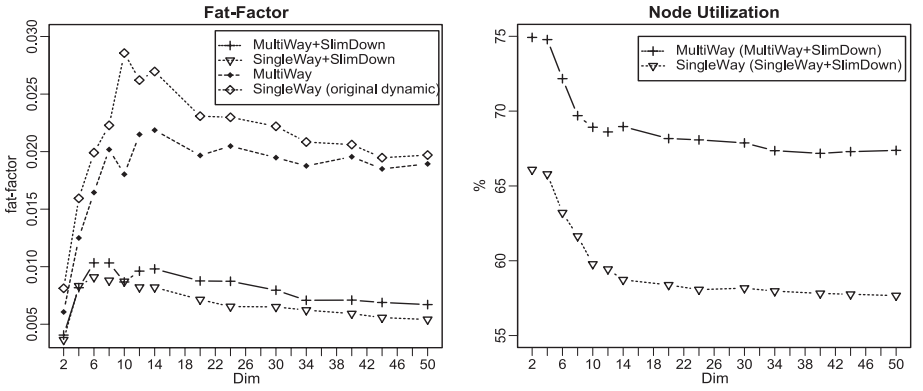

Fig. 7. Building the M-tree: a) Fat-factor. b) Node utilization.

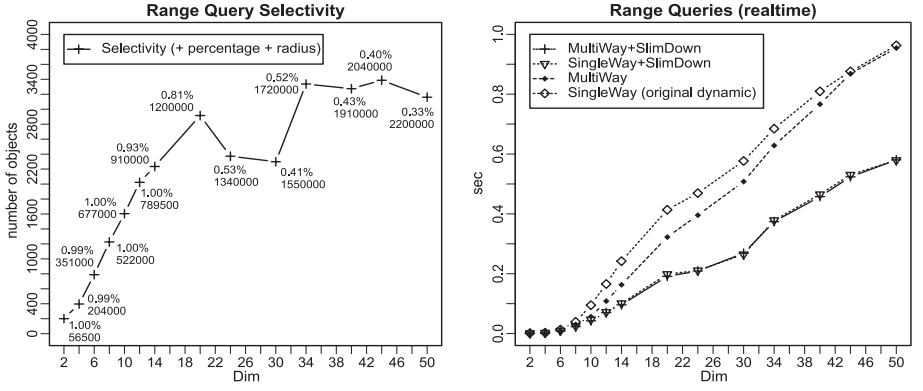

Fig. 8. Range queries: a) Range query selectivity. b) Range query realtimes.

from the datasets. Each query test consisted from 100 to 750 queries (according to the dimensionality and dataset size). The results were averaged.

In Figure 8a, the average range query selectivity is presented for each dataset. The selectivity was kept under 1% of all the objects in the dataset. For an interest, we also present the average query radii. In Figure 8b, the realtime costs are presented for the range queries. We can see that the query processing of the SingleWay+SlimDown and MultiWay+SlimDown indexes is almost twice faster when compared with the SingleWay index.

The disk access costs and the computation costs for the range queries are presented in Figure 9. The computation costs comprise the total number of the d function executions.

4.3 k-NN Queries

The performace gain is even more noticeable by the k-NN queries processing. In Figure 10a, the disk access costs are presented for 10-NN queries.

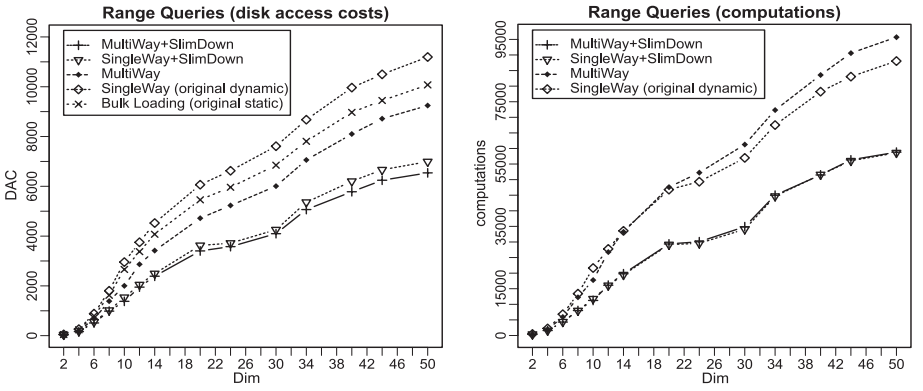

Fig. 9. Range queries: a) Disk access costs. b) Computation costs.

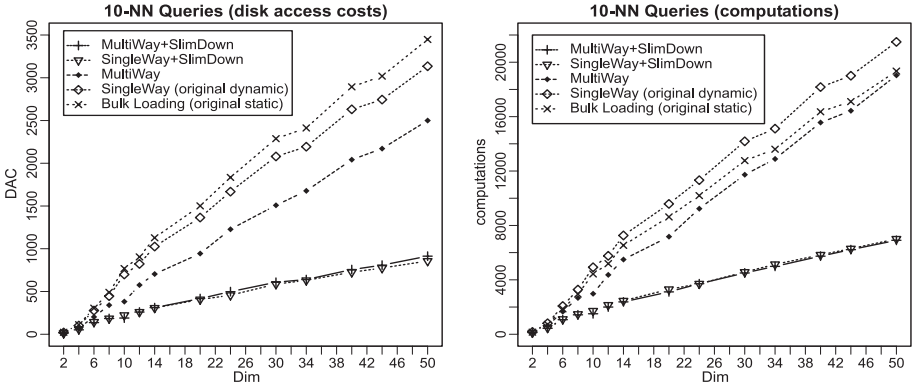

Fig. 10. 10-NN queries: a) Disk access costs. b) Computation costs.

As the results show, querying the SingleWay+SlimDown index consumes 3.5-times less disk accesses than querying the SingleWay index. Similar behaviour can be observed also for the computation costs presented in Figure 10b. The most promising results are presented in Figure 11 where the 100-NN queries were tested. The querying performance of the SingleWay+SlimDown index is here better by more than 300% than the performance of the SingleWay index.

5 Conclusions

In this paper we have introduced two dynamic techniques of building the M-tree. The cheaper multi-way insertion causes superior node utilization and thus smaller indexes, while the querying performance for the k-NN queries is improved by up to 50%. The more expensive generalized slim-down algorithm causes superior querying performance for both the range and the k-NN queries, for the 100-NN queries even by more than 300%.

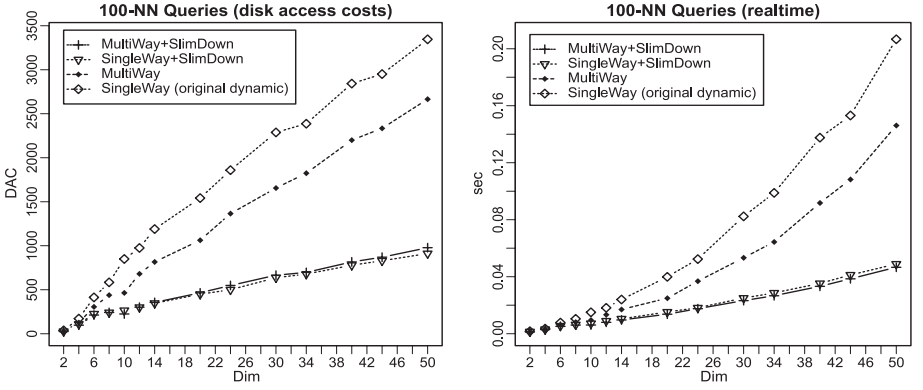

Fig. 11. 100-NN queries: a) Disk access costs. b) Realtime costs.

Since the M-tree construction costs used by the multi-way insertion and mainly by the generalized slim-down algorithm are considerable, the methods proposed in this paper are suited for DBMS scenarios where relatively few insertions to a database are requested and, on the other hand, many similarity queries must be quickly answered at a moment.

From the DBMS point of view, the static bulk loading algorithm can be considered as a transaction, hence the database is not usable during the bulk loading algorithm run. However, the slim-down algorithm, as a dynamic post-processing method, is not a transaction. Moreover, it can operate continuously in a processor idle time and it can be, whenever, interrupted without any problem. Thus the construction costs can be spread over the time.

References

1. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of World-Wide Computing and its Applications'97, WWCA'97, Tsukuba, Japan*, 1997.
2. J. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communication of the ACM*, 18(9):508–517, 1975.
3. S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22nd Intern. Conf. on VLDB, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.
4. C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.
5. T. Bozkaya and Z. M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, 1999.
6. P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australian Conference (ADC'98)*, pages 15–26, 1998.
7. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd Athens Intern. Conf. on VLDB*, pages 426–435. Morgan Kaufmann, 1997.

8. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.
9. E. Navarro, R. Baeza-Yates, and J. Marroquin. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
10. M. Patella. *Similarity Search in Multimedia Databases*. Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.
11. H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(3):184–260, 1984.
12. H. Samet. *Spatial data structures in Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 361–385. Addison-Wesley/ACM Press, 1995.
13. C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777, 2000.
14. J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.
15. P. N. Yanilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms - SODA*, pages 311–321, 1993.
16. C. Yu. *High-Dimensional Indexing*. Springer–Verlag, LNCS 2341, 2002.