Journal of Discrete Algorithms 7 (2009) 62–77

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

# New dynamic construction techniques for M-tree

Tomáš Skopal *, Jakub Lokoč

*Charles University in Prague, FMP, Department of Software Engineering, Malostranské nám. 25, 118 00 Prague, Czech Republic*

ARTICLE INFO

*Article history:*
Available online 26 September 2008

*Keywords:*
Metric access methods
M-tree
Forced reinsertions
Dynamic insertion

ABSTRACT

Since its introduction in 1997, the M-tree became a respected metric access method (MAM), while remaining, together with its descendants, still the only database-friendly MAM, that is, a dynamic structure persistent in paged index. Although there have been many other MAMs developed over the last decade, most of them require either static or expensive indexing. By contrast, the dynamic M-tree construction allows us to index very large databases in subquadratic time, and simultaneously the index can be maintained up-to-date (i.e., supports arbitrary insertions/deletions). In this article we propose two new techniques improving dynamic insertions in M-tree—the forced reinsertion strategies and so-called hybrid-way leaf selection. Both of the techniques preserve logarithmic asymptotic complexity of a single insertion, while they aim to produce more compact M-tree hierarchies (which leads to faster query processing). In particular, the former technique reuses the well-known principle of forced reinsertions, where the new insertion algorithm tries to re-insert the content of an M-tree leaf that is about to split in order to avoid that split. The latter technique constitutes an efficiency-scalable selection of suitable leaf node wherein a new object has to be inserted. In the experiments we show that the proposed techniques bring a clear improvement (speeding up both indexing and query processing) and also provide a tuning tool for indexing vs. querying efficiency trade-off. Moreover, a combination of the new techniques exhibits a synergic effect resulting in the best strategy for dynamic M-tree construction proposed so far.



## 1. Introduction

The methods of similarity search are becoming a standard tool in various data-oriented research areas, like multimedia databases, data mining, bioinformatics, biometric databases, text retrieval, etc. Basically, the task of similarity search is expressed as follows: Given a universe $\mathbb{U}$ of database object descriptors $O_i \in \mathbb{U}$ (e.g., MPEG7 features in case of images), a database of objects $\mathbb{S} \subset \mathbb{U}$, and a similarity measure $\delta(O_i, O_j)$ computing the similarity score between any two objects of the universe, then we want to query the database in order to retrieve the most similar objects to our query object $Q \in \mathbb{U}$. At the same time, since a single computation of the similarity score $\delta(\cdot,\cdot)$ is considered as CPU-intensive (often quadratic in object size, or worse), we would like to avoid the sequential search over the entire database. Unfortunately, an efficient processing of a similarity query cannot be accomplished by conventional access/indexing methods (like B-trees), because there does not exist a meaningful canonical ordering on the database objects.

Here come the *metric access methods* (MAMs) into play, a class of indexing methods aimed at similarity search of various kinds. The similarity measure $\delta$ (dissimilarity or distance, actually) is required to be a metric function, that is, it must satisfy the reflexivity, non-negativity, symmetry and triangle inequality. Based on these properties, the MAMs par-

* Corresponding author.
*E-mail addresses:* skopal@ksi.mff.cuni.cz (T. Skopal), lokoc@ksi.mff.cuni.cz (J. Lokoč).

1570-8667/$ – see front matter 
doi:10.1016/j.jda.2008.09.013

tition (or index) the database $\mathbb{S}$ into classes, so that only some classes have to be sequentially searched when querying. This results in less distances $\delta(\cdot,\cdot)$ computed at query time, and thus in more efficient retrieval. The number of distance computations spent during index construction is referred to as the *construction costs*, while *query costs* represent the computations spent by processing a query. The already developed MAMs address various aspects—main-memory/database-friendly methods, static/dynamic indexing, exact/approximate search, centralized/distributed indexing, etc. (see monographs [10,19] and survey [3]). The M-tree [6] represents a centralized, dynamic, and database-friendly MAM. Although there exist MAMs more efficient in querying performance, the M-tree (and its descendants) is still the only solution applicable to very large databases, due to its cheap B-tree-like construction. The M-tree is also dynamic, so it is easily updatable by arbitrary insertions or deletions of individual objects.

In this article we propose two new techniques improving dynamic insertion in M-tree—the forced reinsertions and so-called hybrid-way leaf selection. The former one applies the well-known principle of forced reinsertions into M-tree, the latter represents a scalable selection of leaf wherein a new object has to be inserted. The rest of the paper is structured as follows—in Section 2 we review the M-tree, in Section 3 we discuss alternative ways of M-tree construction. In Sections 4 and 5 we describe the newly proposed techniques. The experimental evaluation is included in Section 6, while Section 7 concludes the paper.

## 2. M-tree

Based on properties well-tried in $B^+$-tree and $R^*$-tree, the M-tree [6] is a dynamic metric access method suitable for indexing of large metric databases. The structure of M-tree represents a hierarchy of nested ball regions, where data is stored in leaves, see Fig. 1a. Every node has a capacity of $m$ entries and a minimal occupation $m_{min}$; only the root node is allowed to be underflowed below $m_{min}$. The inner nodes consist of routing entries $rout(R)$:

$$rout(R) = \big[R, ptr\big(T(R)\big), r_R, \delta\big(R, Par(R)\big)\big],$$

where $R \in \mathbb{U}$ is a routing object, $ptr(T(R))$ is a pointer to the subtree $T(R)$, $r_R$ is a covering radius, and the last component is a distance to the parent routing object $Par(R)$ (so-called *to-parent distance*[1]) denoted as $\delta(R, Par(R))$. In order to correctly bound the data in $T(R)$'s leaves, the routing entry must satisfy the *nesting condition*: $\forall O_i \in T(R), r_R \geqslant \delta(R, O_i)$. The routing entry can be viewed as a ball region in the metric space, having its center in the routing object $R$ and radius $r_R$. A leaf (ground) entry has a format:

$$grnd(D) = \big[D, oid(D), \delta\big(D, Par(D)\big)\big],$$

where $D \in \mathbb{S}$ and $\delta(D, Par(D))$ are similar as in the routing entry, and $oid(D)$ is an external identifier of the original object ($D$ is just an object descriptor).

### 2.1. Similarity queries in M-tree

Similarly like the data regions described by routing entries, also the two most common similarity queries are described by ball-shaped regions. The *range query* is defined as a ball centered in a query object $Q$ with fixed query radius $r_Q$, hence, we search for objects similar to a query object more than a user-defined threshold. The $k$ nearest neighbor (kNN) queries retrieve the $k$ most similar objects to $Q$. The kNN query region is also ball-shaped, however, the query radius, being the distance to the $k$th closest object, is not known in advance, so it must be iteratively refined during kNN query processing.

The queries are implemented by traversing the tree, starting from the root.[2] Those nodes are accessed, the parent regions of which are overlapped by the query ball. The check for region-and-query overlap requires an explicit distance computation $\delta(R, Q)$ (called *basic filtering*), see Fig. 1b. In particular, if $\delta(R, Q) \leqslant r_Q + r_R$, the data ball $(R, r_R)$ overlaps the query ball $(Q, r_Q)$, thus the child node has to be accessed. If not, the respective subtree is filtered from further processing. Moreover, each node in the tree contains the distances from the routing/ground entries to the center of its parent routing entry (the to-parent distances). Hence, some of the non-relevant M-tree branches can be filtered without the need of a distance computation (called *parent filtering*, see Fig. 1c), thus avoiding the "more expensive" basic overlap check. In particular, if $|\delta(P, Q) - \delta(P, R)| > r_Q + r_R$, the data ball $R$ cannot overlap the query ball, thus the child node has not to be re-checked by basic filtering. Note $\delta(P, Q)$ was computed in the previous (unsuccessful) parent's basic filtering.

### 2.2. Compactness of M-tree hierarchy

Since the ball metric regions described by routing entries are restricted just by the nesting condition, the M-tree hierarchy is very loosely defined, while for a single database we can obtain many correct M-tree hierarchies. However, not every M-tree hierarchy built on a database is *compact* enough. In more detail, when the ball regions are either too large and/or highly overlap "sibling" regions, the query processing is not efficient because routing entries of many nodes overlap the

[1] The to-parent distance is not defined for entries in the root.

[2] We outline just the principles, for details see the original M-tree algorithms [6,15].