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

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 persisted 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.

Key words: metric access methods, M-tree, forced reinsertions, dynamic insertion

* Corresponding author
Email addresses: skopal@ksi.mff.cuni.cz (Tomáš Skopal), lokoc@ksi.mff.cuni.cz (Jakub Lokoč).

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 δ (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 partition (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 [19,10] 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 Figure 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 $rout(R)$:

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

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*: $\forall O_i \in T(R), r_R \geq \delta(R, O_i)$. 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:

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

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).

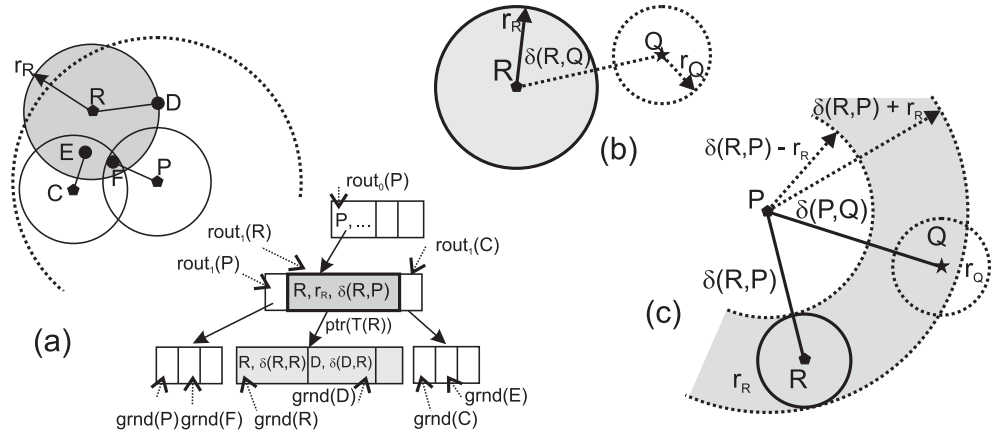

Fig. 1. (a) An M-tree hierarchy (b) Basic filtering (c) Parent filtering

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*

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

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 kth 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 Figure 1b. In particular, if $\delta(R, Q) \leq r_Q + r_R$, 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 Figure 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 query ball. In consequence, large portion of the M-tree hierarchy must be traversed, and so many query-to-object distances have to be computed (resulting in high query costs).

Even if we use always the same construction method, the resulting M-tree hierarchy will still heavily depend on the order in which data objects are inserted (in case of dynamic insertions). A maximally compact M-tree hierarchy(ies) surely exist(s), however, such a construction would require static indexing, and, above all, an exponential construction time. Hence, we would rather prefer an efficient sub-optimal dynamic construction, yet producing sufficiently compact hierarchies.

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

During the last decade, many methods have been developed to challenge the problem of compact M-tree hierarchies. Besides the original M-tree construction (see Section 2.3), we overview some recent M-tree enhancements in Section 3 and present our contributing methods in Sections 4 and 5.

2.3 Building the M-tree

An M-tree is built in the bottom-up fashion (like B-tree or R-tree), so the data objects are inserted into the leaf nodes. When a leaf overflows, a split is performed – a new leaf is created and some objects are moved from the original leaf into the new one. Two new routing entries are created, one for the original updated leaf and one for the new leaf, and inserted into the parent node (entry for the original leaf is just replaced). All to-parent distances to the new routing objects are computed and replaced in the new leaves' entries. Because of inserting new routing entries, the parent node might overflow as well. In such case a split is performed in a similar way, recursively. If the root node is split, the M-tree grows by one level.

When building an M-tree by dynamic insertions, two main problems have to be solved – the leaf selection and the node splitting.

2.3.1 Leaf Selection

In the original M-tree, a process similar to a point query is performed, in order to find an appropriate leaf for object placement. However, in contrast to a point query, only one vertical path of the tree is passed. This approach is also referred to as the *single-way* (or deterministic) insertion, see Figure 2a. When navigating the tree, the next node in the path is chosen such that the inserted object fits the appropriate region best (for details see [6,15]).

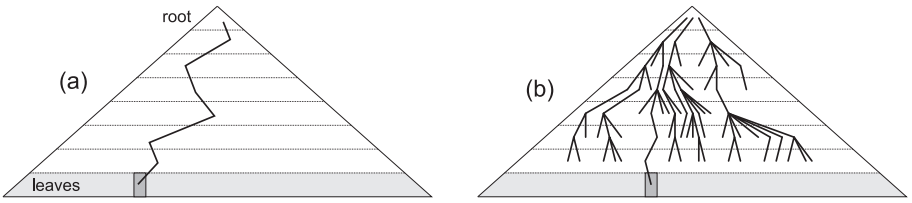

Fig. 2. (a) Single-way leaf selection (b) Multi-way leaf selection

2.3.2 Node Splitting

The node splitting policy is a significant factor of the M-tree building process. When a node is split, two new routing entries have to be created, representing new ball regions. To guarantee a compact M-tree hierarchy, the splitting

process must ensure the new regions are separated as much as possible, they overlap as least as possible, and they are of minimum volumes (radii).

To best fit these requirements, all the objects in the node are candidates to the routing objects. For each pair of candidate routing objects, the resulting nodes are temporarily created and radius of the greater region is determined. Such pair of candidate routing objects is finally chosen, which has the smallest radius of the greater region (so-called *mM_Rad* choice). This *CLASSIC* approach exhibits $O(m^2)$ complexity, where m is the capacity of the node. To avoid the quadratic complexity, there were alternative heuristics developed:

– The *RANDOM* approach directly selects two new routing objects at random, which cuts the complexity down to $O(m)$.
– Instead of considering all objects in the node as candidate routing objects, the *SAMPLING* approach selects randomly just s candidates ($s < m$). Then the complexity of node splitting is $O(ms)$.

In Figure 3 see the result of leaf splitting, comparing the CLASSIC and SAMPLING heuristics. A splitting of non-leaf nodes is similar, though it must take also the radii of the redistributed routing entries into account.

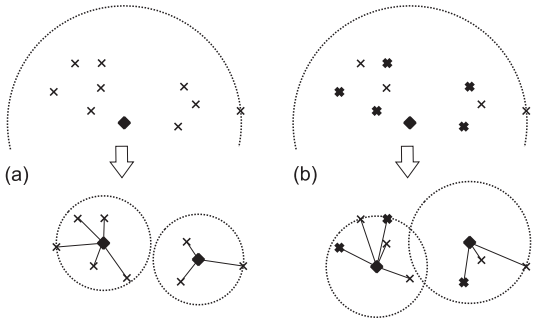

Fig. 3. Leaf split using (a) *CLASSIC* and (b) *SAMPLING* heuristics

3 Related Work

The success of M-tree can be supported by the existence of its many descendants that have appeared during the past decade. We could chronologically name the Slim-tree [17] (discussed in Section 3.1), and the M$^+$-tree [20] which exploits further partitioning of the node by a hyper-plane (i.e., an approach limited to Euclidean spaces). Furthermore, let us mention the PM-tree [12,16] which combines the M-tree with pivot-based techniques, the M^2-tree [5] and M^3-tree [2] which exploit an aggregation of multiple metrics. As the most recent ones we point out to M*-tree [13], where each node is additionally equipped by a nearest-neighbor graph, and the NM-tree [14] which allows also nonmetric distances. In the rest of the paper we consider the original structural properties of M-tree [6] (i.e., modified algorithms, not the structure).

The effectiveness of query processing in M-tree heavily depends on the M-tree compactness, hence, on the construction algorithm used. Intuitively, to improve the search performance, the construction should be more expensive, and vice versa. For example, if we use the *RANDOM* node split heuristic, we obtain low construction costs, but the region volumes/overlaps will increase, and so the query costs will rapidly increase as well. In the following we present three approaches of compact M-tree construction.

3.1 Slim-down Algorithm

The authors of Slim-tree [17] proposed two new M-tree construction techniques. First, a node splitting policy was introduced, based on minimum spanning tree. Instead of choosing many candidate pairs to new routing entries and then temporarily partitioning the node entries for each candidate pair, in Slim-tree the complete distance graph between node entries is used to construct the minimum spanning tree (MST). Then, the longest edge in MST is removed in order to obtain two separate sets of entries – these sets directly constitute the two new nodes. Although the MST splitting still needs $O(m^2)$ distance computations due to the complete distance graph, there are $O(m)$ external CPU costs saved, otherwise used for the temporary partitioning. Second, the slim-down algorithm was presented, a post-processing method trying to redistribute ground entries into more suitable leaves. This technique produces very compact M-tree/Slim-tree hierarchies, however, it is also very expensive – up to linear with database size for a single ground entry redistribution.

The slim-down algorithm was later generalized in order to redistribute also routing entries at higher M-tree levels [15], which leads to even more compact hierarchies (see Figure 4).

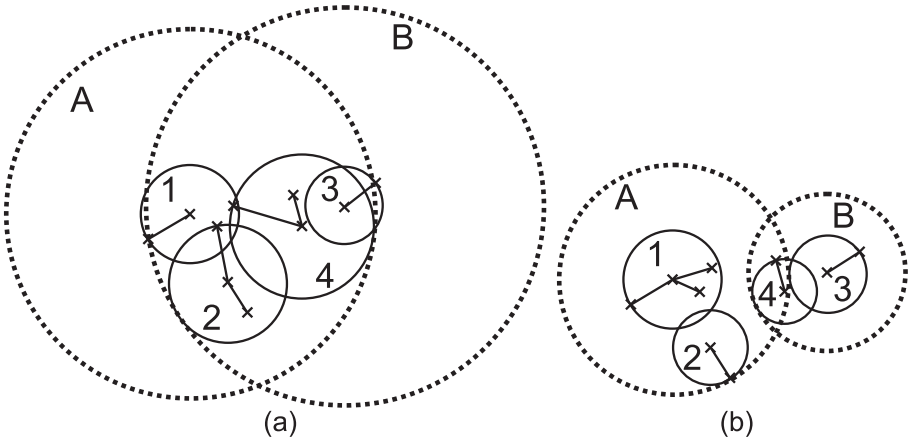

Fig. 4. An M-tree before (a) and after (b) generalized slim-down algorithm run

3.2 Multi-way Leaf Selection

Another way of improving M-tree compactness is an employment of more sophisticated selection of target leaf wherein a new object will be inserted. In [15] the *multi-way* (non-deterministic) leaf selection was proposed. The target leaf is found such that a point query (i.e., range query with $r_Q = 0$) is issued having the new inserted object in the role of Q. All the "touched" and non-full leaves serve as candidates to the target leaf. Among the touched non-full leaves the one is chosen which has its parent routing object closest to the inserted object (see Figure 2b).

The multi-way leaf selection has positive impact on the M-tree compactness, though not as large as the generalized slim-down algorithm. On the other hand, the insertion employing multi-way leaf selection is by far less expensive than the generalized slim-down algorithm, though still up to linearly expensive with database size for a single insertion.

3.3 Bulk Loading

The basic idea of bulk loading is to statically create the index from scratch but knowing beforehand the database. Then some optimizations may be performed to obtain a "good" index for that database. Usually, the proposed bulk loading techniques are designed for specific index structures, but there have been proposals for more general algorithms. For example, in [7] the authors propose two generic algorithms for bulk loading, which were tested with different index structures like the R-tree and the Slim-tree. Note that the efficiency of the index may degrade if new objects are inserted after its construction. Specific bulk loading techniques for M-tree were introduced in [4,11], the latter one furthermore introduces, for the first time, dynamic deletions on M-tree (renamed to SM-tree here). Another bulk loading algorithm for Slim-tree was recently proposed in [18].

Sometimes the bulk loading is viewed as a technique for fast index construction, rather than a tool for building a compact index hierarchy.

4 Forced Reinsertions

The first contribution we propose in this article is an adaptation of forced reinsertion into the process of dynamic insertions in M-tree. The *forced reinsertions* is a well-known technique from the R*-tree [1]. The idea is based on

an easy principle. Some objects are removed from a leaf to avoid a split operation and then inserted in a common way under a hope that the reinserted objects will arrive into more "suitable" leaf(s). There are two basic motivations to consider forced reinsertion as beneficial, considering any B-tree-based spatial/metric index structure. The straightforward (but also weaker) motivation is better node occupancy, hence, forced reinsertions lead to fuller nodes. Second, due to unavoidable node splitting over the time, the compactness of spatial/metric region hierarchy deteriorates – the region volumes and overlaps grow because of spatial aggregations mixing old and new objects/regions. Here the forced reinsertions could serve as an opportunity to move some "bad" (volume- or overlap-inflating) objects from the leaf.

In M-tree, we have to face some specific issues when implementing forced reinsertions. Basically, when a new object is inserted into a leaf that is now about to split, some suitable objects from the leaf must be selected and reinserted. The crucial goal is to propose a method aiming to decrease the covering radius of the reinserted leaf as much as possible, while simultaneously aiming to grow the radii of leaves accepting the reinserted objects as little as possible. Here we have to take also the induced leaf splits/reinsertions into account, that is, a forced reinsertion attempt could raise a chain of reinsertions terminated by regular splits "after a while".

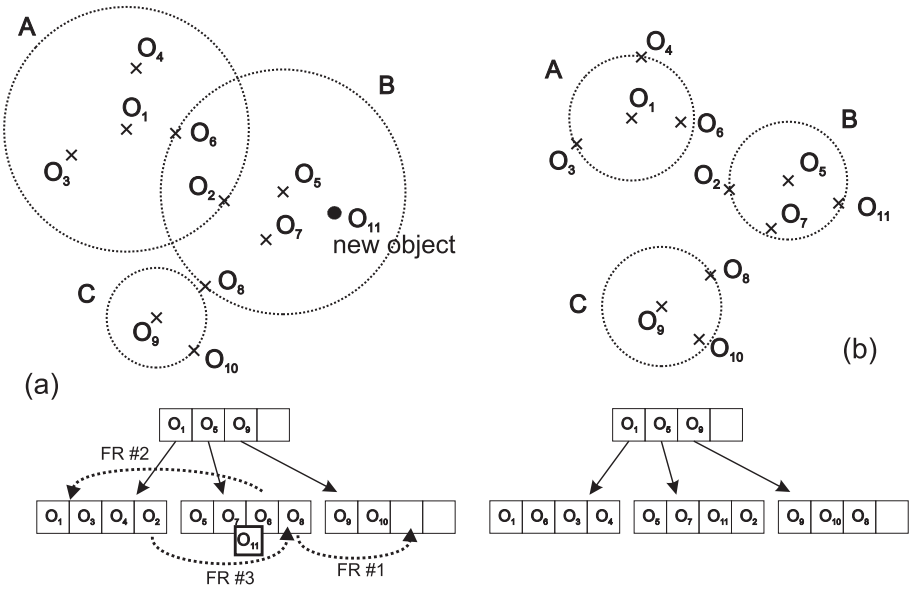

Fig. 5. (a) Before reinsertions (b) Decreased overlaps/volumes after 3 reinsertions

As a fundamental assumption, we expect objects located close to the region's "border" have higher probability to be suitably reinserted than the more "centered" ones. Since in an M-tree node the entries are ordered according to their distances to the parent routing object (region's center), we can select the fur-

thest ones (close to the border) easily.[3] In Figure 5 see a motivation example – situation just before a leaf split, and how the split is avoided after a series of induced reinsertions, denoted as FR#1, FR#2, FR#3. We can see that not only the split was prevented, but the M-tree compactness was improved, too.

We propose two variants using forced reinsertions for M-tree – the full reinsertions and conservative reinsertions.

4.1 Full Reinsertions

As mentioned before, we assume the most suitable entries for reinserting are the furthest ones from the parent routing object. To avoid an overfull leaf split, some of its furthest entries are removed from the leaf and pushed onto a temporary main memory stack $\mathcal{S}$. The covering radius of the leaf's parent routing entry is then immediately reduced to the distance of the routing object to the new furthest entry in the leaf (and so the covering radii of all ancestors). Then, the current entry on the top of $\mathcal{S}$ is reinserted in a standard way as it would be a regular new object to be inserted. Naturally, a forced reinsertion could possibly induce further reinsertion attempts (i.e., the top of the stack grows). The reinsertions are repeated until the stack becomes empty.

4.1.1 Recursion Depth

Since a single reinsertion attempt could generally raise a long chain of subsequent reinsertions (the stack is inflating instead of emptying), we would like to limit the number of forced reinsertion attempts to keep the construction costs reasonable and scalable. We denote the limit as a user-defined *recursion depth* parameter. When the limit of reinsertion attempts is reached, the remaining entries on the stack are popped and reinserted such that only regular splits are allowed from now on (i.e., the stack does not grow anymore).

4.1.2 Entries Removing

As to the entries removing mentioned before, we remove at most k furthest entries from the leaf in the direction from closer to further ones (where k is user-defined). However, if the newly inserted object is within the k entries, we remove just the more distant of the k entries (i.e., we do not remove the new one and all closer). We called such a removing of entries as the *reverse pessimistic* entries removing, see Figure 6.

[3] Remember the precomputed distances to the routing entry (the to-parent distances) are stored in all entries except those in the root node.

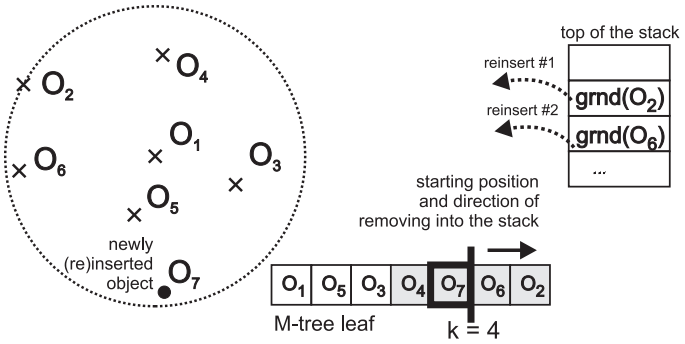

Fig. 6. Reverse pessimistic entries removing

As a motivation for the reverse pessimistic removing, we suppose that the reinsertion of an object being just newly inserted (and all closer ones) would cause insertion back to the same leaf (the pessimistic assumption). As to the direction of entries removing, being "reverse" due to starting from the leaf's "middle", we assume the furthest entries (being the outlying "losers") should be reinserted first. This heuristics aims to increase the likelihood of finding a suitable non-full leaf, being otherwise possibly occupied by the other removed "sibling" entries on the stack. Although we tried also other variants of entries removing than the reverse pessimistic (as described and evaluated in [9]), they performed worse, so we do not consider them anymore in this article.

Algorithm 1 (insertion with full forced reinsertions)

```
let maxRemoved be maximal number of removed entries (user-defined)        // denoted k in Section 4.1.2
let recursionDepth be the maximal depth of recursion (user-defined)

method Insert(O_new) {
  find leaf L for O_new                                                   // "either-way" leaf selection
  insert O_new into L

  if L is not overfull then return
  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)
  exclude grnd(O_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then {
    for (j = 0; j < |𝔼|; j++) {                                           // remove furthest entries from leaf
      𝒮.Push(𝔼.GetEntry(1))
      𝔼.DeleteEntry(1)
    }
    decrease radius of L (and possibly of its ancestors)

    while (𝒮 is not empty) {                                              // reinsert removed entries
      recursionDepth = recursionDepth − 1
      Insert(𝒮.Pop())
    }
  } else {
    perform regular split of L (and possibly of its ancestors)
  }
}
```

To provide a comprehensive description, in Algorithm 1 see the pseudocode of dynamic insertion enhanced by full forced reinsertions.

4.2 Conservative Reinsertions

As observed in [9] and as shown in experiments, the full forced reinsertion variant is effective in producing compact M-tree hierarchy, though it is still quite expensive in terms of M-tree construction costs. The reverse pessimistic strategy ensures the newly inserted entry and all closer entries will not be reinserted (probably back to the same leaf). However, there can still occur reinsertions of more distant entries into the same leaf, being thus ineffective. In this section, therefore, we introduce an improvement of full forced reinsertions, called the *conservative forced reinsertions*, better avoiding reinsertions of entries back into the same leaf (see also Algorithm 2 for pseudocode).

The improvement requires a slight extension of the M-tree's ground entry format, as $grnd(D) = [D, oid(D), \delta(D, Par(D)), SplitNumber]$. The $SplitNumber$ is the number of node splits occurred before (re)inserting this entry into the current leaf. In fact, the $SplitNumber$ represents a logical time related to the amount of structural changes in M-tree during its construction.

The way of removing of entries onto the stack $\mathcal{S}$ is the same as used in the full forced reinsertion variant (i.e., using the reverse pessimistic strategy). The difference is in the processing of popped entries from stack, and in the structure of a stack entry. Instead of storing pure objects $O_i \in \mathbb{S}$ on the stack, now we store the entire ground entry $grnd(O_i)$ and, additionally, a pair of co-identifiers determining wherefrom the entry came. The first identifier is an id of the source leaf, while the second one is the id of the source leaf's routing entry (see an example in Figure 7 right). The reason for two identifiers of a leaf is that we want to distinguish leaves whose node id remained the same but they obtained a different parent routing entry (caused by a possible split).

4.2.1 The Stack Processing

After the entries are removed onto the stack, the top entry is popped and reinserted in the usual way. We check whether it falls back to the leaf it came from, that is, whether the leaf+parent entry co-identifiers are equal to that of the entry popped from stack. If so, the top of stack is checked for a contiguous block of entries. In such a contiguous block all entries must become also from the same leaf as the first reinserted entry, and all must be "younger" (their $SplitNumber$ is higher). If such a block exists, its entries are popped and directly *moved* to the respective leaf, following the first reinserted entry. Actually, they are returned to their original leaf for free; no regular insertion is performed for them (no distance has to be computed). Otherwise, if such a contiguous block of entries does not exist (or its processing was finished), the entries on the top of stack are reinserted in the usual way.

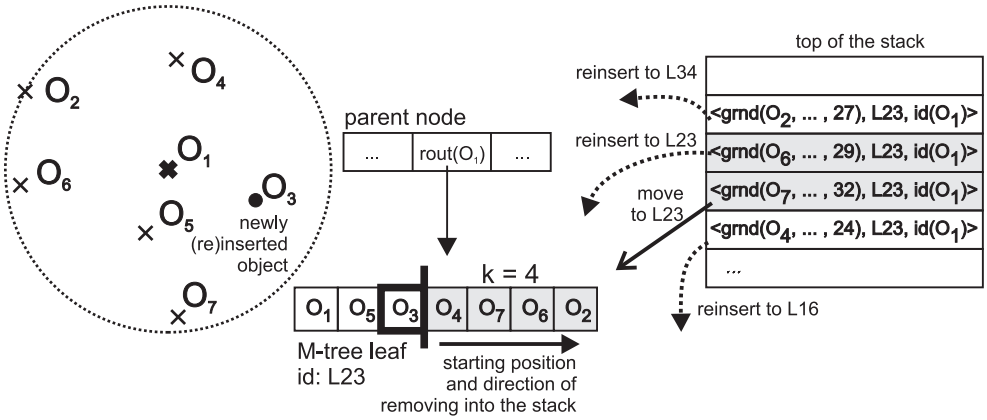

Fig. 7. Conservative forced reinsertions

Algorithm 2 (insertion with conservative forced reinsertions)

```
let maxRemoved be maximal number of removed entries (user-defined)        // denoted k in Section 4.1.2
let recursionDepth be the maximal depth of recursion (user-defined)

method Insert(O_new, SplitCount) {
  find leaf L for O_new                                                   // "either-way" leaf selection
  insert O_new into L
  let rout(O_p) be L's parent routing entry

  while (S is not empty and S.TopEntry.NodeId = L.NodeId and
        S.TopEntry.GroundEntry.RoutId = L.RoutId and S.TopEntry.GroundEntry.SplitCount ≥ SplitCount) {
    L.Insert(S.Pop().GroundEntry)                                         // move the entries back to the original leaf
    if (L is overfull) then {
      perform regular split of L (and possibly of its ancestors)
      return
    }
  }
  if L is not overfull then return
  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)
  exclude grnd(O_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then {
    for (j = 0; j < |𝔼|; j++) {                                           // remove furthest entries from leaf
      S.Push(⟨𝔼.GetEntry(1), L.id, rout(O_p).id⟩)
      𝔼.DeleteEntry(1)
    }
    decrease radius of L (and possibly of its ancestors)

    while (S is not empty) {                                              // reinsert removed entries
      recursionDepth = recursionDepth − 1
      let entry = S.Pop().GroundEntry
      Insert(entry.object, entry.SplitCount)
    }
  } else {
    perform regular split of L (and possibly of its ancestors)
  }
}
```

For an example, in Figure 7 the top entry on the stack was reinserted into a leaf 34, however, the next one arrived into the same leaf 23 as it came from. Therefore, a single-entry block on the stack was identified, fulfilling the block conditions (i.e., originating also from leaf 23 and being younger by 3 splits),

and moved back to the leaf. The next entry on the stack also came from leaf 23 but was older, so for this entry there is a greater probability that during its longer "sitting" in leaf 23 there appeared better leaves in the M-tree, so this one is properly reinserted to the leaf 16.

The motivation for the above described optimization is a conservative assumption, that if a set of entries was removed onto the stack from the same node at the same time, and one of them was reinserted back into the same leaf, then also some other entries in this set would be probably reinserted into the same leaf as well. Hence, instead of ineffective costly reinsertions we rather "give up" and move the entries directly back.

Additional notes:
– The to-parent distance stored in a moved ground entry is still valid. Actually, this is another reason why we use additionally the parent routing entry identifier to co-identify the source leaf.
– When a ground entry is reinserted, its $SplitNumber$ is updated only in case it has not been reinserted/moved back into the same leaf.
– Due to the reverse pessimistic strategy, the moved entries are always closer to the parent routing entry than the entry reinserted to the same leaf as first. Hence, the leaf's covering radius cannot be inflated due to entries moving.

4.3 Construction vs. Query Efficiency

The rationale for forced reinsertions is two-fold. First, reinsertions could clearly improve the compactness of M-tree (thus the query performance) at the cost of (a bit) more expensive construction. The second reason considers the trade-off between indexing and querying performance. In contrast to the first reason (speeding up querying), sometimes we would like to decrease construction costs but simultaneously keep the query costs as low as if used more expensive construction. With forced reinsertions this goal could be carried out. For example, the *CLASSIC* splitting of M-tree node is expensive but brings faster queries, while the *SAMPLING* splitting is cheaper but also leads to slower queries. Since the *CLASSIC* splitting could produce M-tree which is compact enough, at some scenarios the employment of forced reinsertions could not bring any further improvement – so only the construction costs grow, but the retrieval performance is not improved. In such case we might rather employ the forced reinsertions together with the *SAMPLING* splitting. This way we could achieve retrieval costs similar to that of *CLASSIC* splitting, however, for cheaper construction – somewhere between *SAMPLING* and *CLASSIC* without forced reinsertions. In other words, forced reinsertions might cheaply fix the bad data partitioning caused by *SAMPLING* splitting.

Finally, we have to emphasize that when combined with the single-way leaf selection (or the hybrid-way leaf selection, see next section), the asymptotic

complexity of a single insertion is still logarithmic with database size. In more detail, the number of reinsertion attempts is limited by a constant (recursion depth), the maximal number of entries in a leaf is constant, while the single/hybrid-way leaf selection is of logarithmic complexity.

5 Hybrid-way Leaf Selection

As the second contribution, we introduce so-called *hybrid-way leaf selection*. The rationale for this effort was the performance gap between single-way and multi-way leaf selection (see Sections 2.3.1 and 3.2). On the first hand, the single-way selection is very cheap (logarithmic with database size) but often selects a leaf that is not optimal, thus the resulting M-tree compactness is not very good. On the other hand, the multi-way technique selects an optimal leaf (though only a non-full one), but it is expensive, up to linear with DB size.

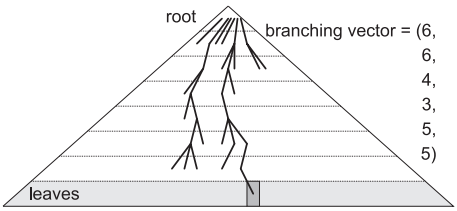

Fig. 8. Hybrid-way leaf selection

Instead of the multi-way's expensive traversal to all leaves whose regions could cover the new object, the hybrid-way technique selects only a limited number of the "best" candidate nodes at each level. Such nodes become the candidates, the parent routing regions of which cover the newly inserted object and their routing objects are as close to the new object as possible. All covering child nodes of the selected candidate nodes are then followed down to the next M-tree level, while, again, only a limited number of the best ones are selected as the candidate nodes, and so on. After the pre-leaf level is reached, the candidate pre-leaves are checked for the best routing entry and the respective leaf is returned as the finally selected leaf. In the rather unlikely situation when no candidate nodes are selected at a level (i.e., the new object is not covered by any node's ball), the hybrid-way technique gives up and selects the leaf by single-way selection. The limits of candidates at all levels are described by so-called *branching vector*. The branching vector determines how many paths in M-tree the hybrid-way selection traverses, see an example in Figure 8.

The hybrid-way solution represents a scalable technique, actually generalizing both the single- and multi-way selections. If the branching vector contains only 1s, we obtain a single-way-like behavior, though not the same – the single-way always selects a node at any level, the hybrid-way does not need. If the branching vector contains only ∞s, we obtain a multi-way-like behavior (though the original multi-way selects only non-full leaves). If all the numbers in the vector

are equal, we can talk just about a *branching factor* $f \in \langle 1, \infty \rangle$. In Algorithm 3 see the hybrid-way leaf selection. Note the algorithm uses the parent filtering (Section 2.1) to efficiently filter out the non-covering nodes. Because the number of traversed paths is limited by the branching vector (consisting of constants), the hybrid-way selection is still of logarithmic complexity, though more expensive than the single-way selection by a constant factor.

Algorithm 3 (hybrid-way leaf selection)

```
method FindLeafHybridWay(O_new, branching vector v) {
  let nodeCandidates = {⟨root; 0⟩} // the 1st value in ⟨·,·⟩ is denoted .Node, the 2nd .ObjectToParentDistance
  let targetLeaf = ∅
  let minDistance = ∞

  for (level = 0; level < treeHeight; level++) {
    let levelCandidates = ∅
    for each can in nodeCandidates {
      for each entry in can.Node {
        if |can.ObjectToParentDistance − entry.RoutingToParentDistance| ≤ entry.radius then {   // parent filt.
          compute δ(entry.object, O_new)
          if δ(entry.object, O_new) < entry.radius then {                                      // basic filtering
            if level = treeHeight − 1 then {                                  // at pre-leaf level select the winning leaf
              if δ(entry.object, O_new) < minDistance then {
                minDistance = δ(entry.object, O_new)
                targetLeaf = entry.childNode
              }
            } else {                                                          // at higher levels follow the child nodes
              read entry.childNode
              add ⟨entry.childNode; δ(entry.object, O_new)⟩ to levelCandidates
            } /* end if level */
          }
        }
      } /* for each entry */
    } /* for each can */
    if level = treeHeight − 1 {
      if targetLeaf is ∅ then return FindLeafSingleWay(O_new) else return targetLeaf
    }
    sort levelCandidates by .ObjectToParentDistance ASC
    let nodeCandidates = levelCandidates[0..v[level]−1]                  // pick the best node candidates
  } /* for each level */
}
```

6 Experimental Evaluation

We performed an extensive experimentation with the two new techniques and their combination. We compared them against the original M-tree dynamic construction and also against the previously proposed techniques including multi-way leaf selection and generalized slim-down algorithm. Only the distance computation costs are included in the experiments. Since the I/O costs correlate with the computation costs, their inclusion would be redundant.

6.1 The Testbed

We have used two databases, a subset of the *CoPhIR* database [8] of MPEG7 image features extracted from images downloaded from `flickr.com`, and a

synthetic database of polygons. The CoPhIR subset consisted of 1,000,000 feature vectors formed by two MPEG7 features (12-dimensional color layout and 64-dimensional color structure, i.e., total 76 dimensions). As a distance function the Euclidean (L_2) distance has been employed. The *Polygons* database was a synthetic randomly generated set of 250,000 2D polygons, each polygon consisting of 5–15 vertices. The Polygons should serve as a non-vectorial analogy to uniformly distributed points. The first vertex of a polygon was generated at random. The next one was generated randomly, but the distance from the preceding vertex was limited to 10% of max. distance. We used the Hausdorff distance for measuring two polygons (where the order of vertices does not matter), so here a polygon could be interpreted as a cloud of points.

6.2 Experiment Settings

The query costs were always averaged for 200 query objects, while the queries followed the distribution of database objects. We did not perform an inter-MAM comparison; we focused just on various configurations of M-tree – with or without forced reinsertions under single-, multi-, or hybrid-way leaf selection. As the parameters we observed various data dimensionalities, database sizes, M-tree node capacities, hybrid-way branching factor, as well as various forced reinsertion settings. The M-tree node capacities ranged from 20 to 80, the index sizes took 1–138 MB, the M-tree heights were 2–5 (3–6 levels). The minimal M-tree node utilization was set to 20% of node capacity. On average, the methods utilizing forced reinsertions achieved 80% leaf utilization (87% in case of multi-way leaf selection), while the "non-reinserting" ones got to 70% (75% for multi-way selection). The index size is connected with the leaf utilization, so the forced reinsertions produced indexes smaller by 15%. Unless otherwise stated, the database size in experiments was set to 250,000 objects.

stage of insertion	*label*	*description*
leaf selection	Single	single-way leaf selection (default)
	Multi	multi-way LS
	Hybrid(b)	hybrid-way LS, b stands for the branching factor
	Hybrid(b).Nonfull	hybrid-way LS, restricted to select only non-full leaves, i.e., Hybrid(∞).Nonfull = Multi
node splitting	Classic	classic *mM_Rad* node splitting (default)
	Sampling	sampling *mM_Rad* (10% of node's entries in sample)
forced reinsertions	Full_RI	full FR, recursion depth = 10, removed entries = 4
	Cons_RI(x,y)	conservative FR, x is recursion depth, y is number of removed entries – if not specified, $x = 10, y = 4$
	Cons_RI(x,y).NoHistory	moving of entries is not affected by *SplitCount*
	(nothing specified)	FR not used (default)
generalized slim-down algorithm	GeneralizedSlimDown	generalized slim-down algorithm used on M-tree built using Single.Classic

Table 1. Description of labels in the figures' legends

Because of the many tested M-tree construction variants, we have formed a set of labels denoting certain alternatives within each stage of the insertion process (leaf selection, node splitting, forced reinsertions), see Table 1. A combination of labels belonging to each stage of construction constitutes a complete variant of insertion, these composed labels are used in the following figure legends. Within each stage a default value is marked, which applies in case that no of the respective stage's possibilities is specified in the composed label. Hence, the very original M-tree dynamic insertion methods [6] are denoted as Single.Classic and Single.Sampling.

6.3 The Results

In the first experiment (see Figure 9) we have observed varying branching factor b applied to hybrid-way leaf selection. The greater the b, the better the query performance of Hybrid(b) variants but also the slower construction. For example, Hybrid(∞).Cons_RI is 35% faster in querying than Multi, but slower in construction in a similar proportion. Nevertheless, Hybrid(50).Cons_RI beats Multi and Multi.Cons_RI in both construction and query performance. The results for Multi.Cons_RI and Hybrid(∞).Cons_RI differ because the multi-way selects only non-full leaves, while hybrid-way selects also the full ones.

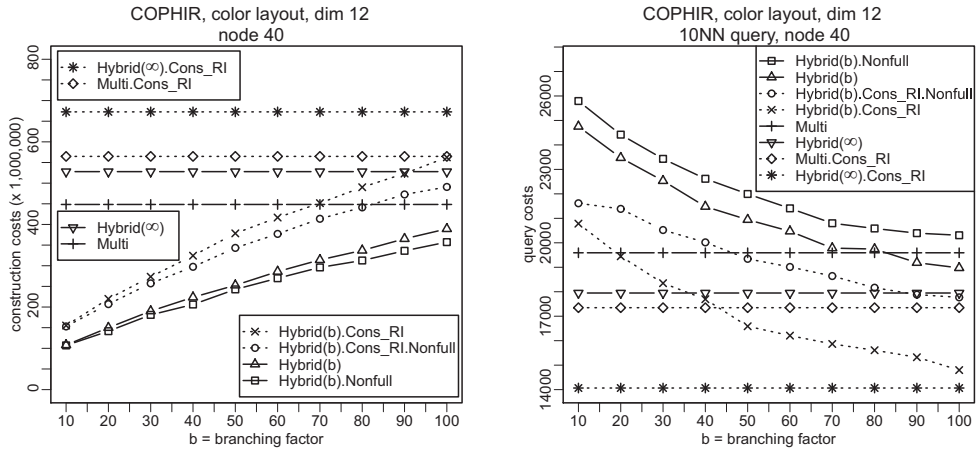

Fig. 9. Hybrid-way branching factor: (a) Construction costs (b) 10NN query costs

In the second experiment we have examined varying database dimensionalities, ranging from 8 to 64 (see Figure 10). We can observe that with increasing dimensionality the queries become less efficient – almost exponentially with the dimension. Hence, we experience the effects of dimensionality curse. However, note the construction costs of all methods except GeneralizedSlimDown and Hybrid(∞).Cons_RI are almost constant. This observation is a nice evidence of the logarithmic construction complexity of hybrid-way leaf selection and forced reinsertions, which is further supported by the worse results of GeneralizedSlimDown and Hybrid(∞).Cons_RI (being super-logarithmic methods).

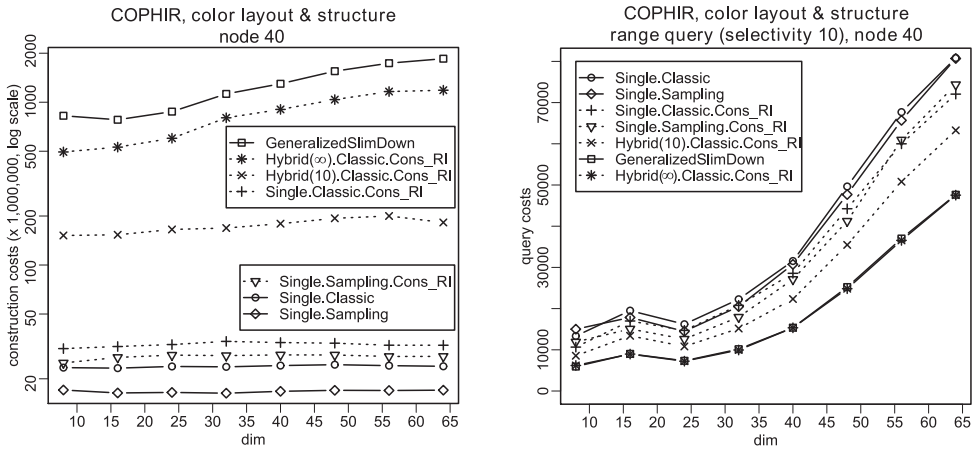

Fig. 10. Varying dimensionality: (a) Construction costs (b) Range query costs

Moreover, note that for dimension 64 the method Hybrid(10).Cons_RI is 1.3× slower in query processing than GeneralizedSlimDown and Hybrid(∞).Cons_RI, but 20× (10×, respectively) faster in construction.

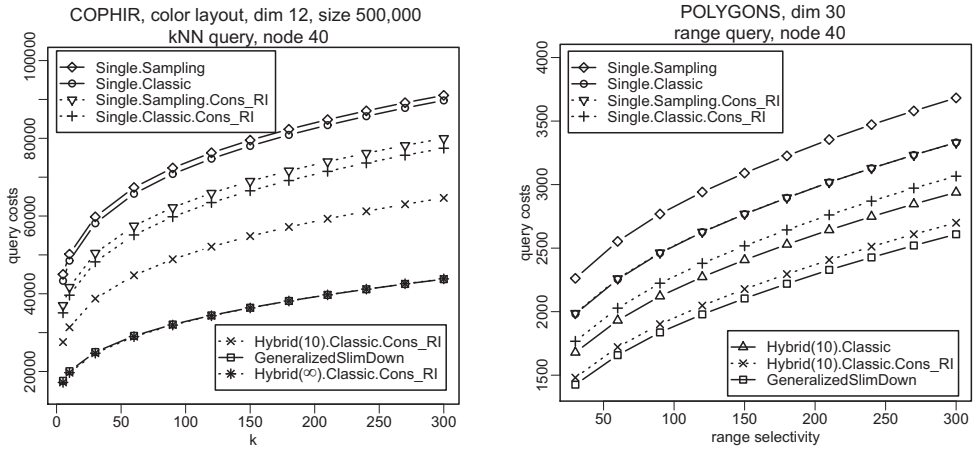

Fig. 11. Varying query selectivity: (a) kNN queries (b) range queries

COPHIR constructions costs, size 500,000, dim 12, node 40

Generalized SlimDown	Hybrid(∞) .Cons_RI	Hybrid(10) .Cons_RI	Classic	Classic .Cons_RI	Sampling .Cons_RI	Sampling
3,981,370,880	2,182,645,760	371,376,416	52,274,784	66,667,772	55,477,300	36,678,464

POLYGONS constructions costs, size 250,000, dim 30, node 40

Generalized SlimDown	Hybrid(10)	Hybrid(10) .Cons_RI	Classic	Classic .Cons_RI	Sampling .Cons_RI	Sampling
103,621,584	59,779,960	87,028,288	22,256,964	28,538,876	22,451,152	15,371,245

Table 2. Construction costs for Figure 11

The third experiment was focused on varying query selectivity – in Figure 11 see the results for kNN and range queries (for the construction costs see Table 2). Note that for Hybrid(∞).Cons_RI in case of COPHIR database the achieved query costs are exactly the same as for GeneralizedSlimDown, but

the construction is almost 50% cheaper. Also note that in case of COPHIR the Single.Sampling.Cons_RI is significantly faster in query processing than Single.Classic, while having almost the same construction costs.

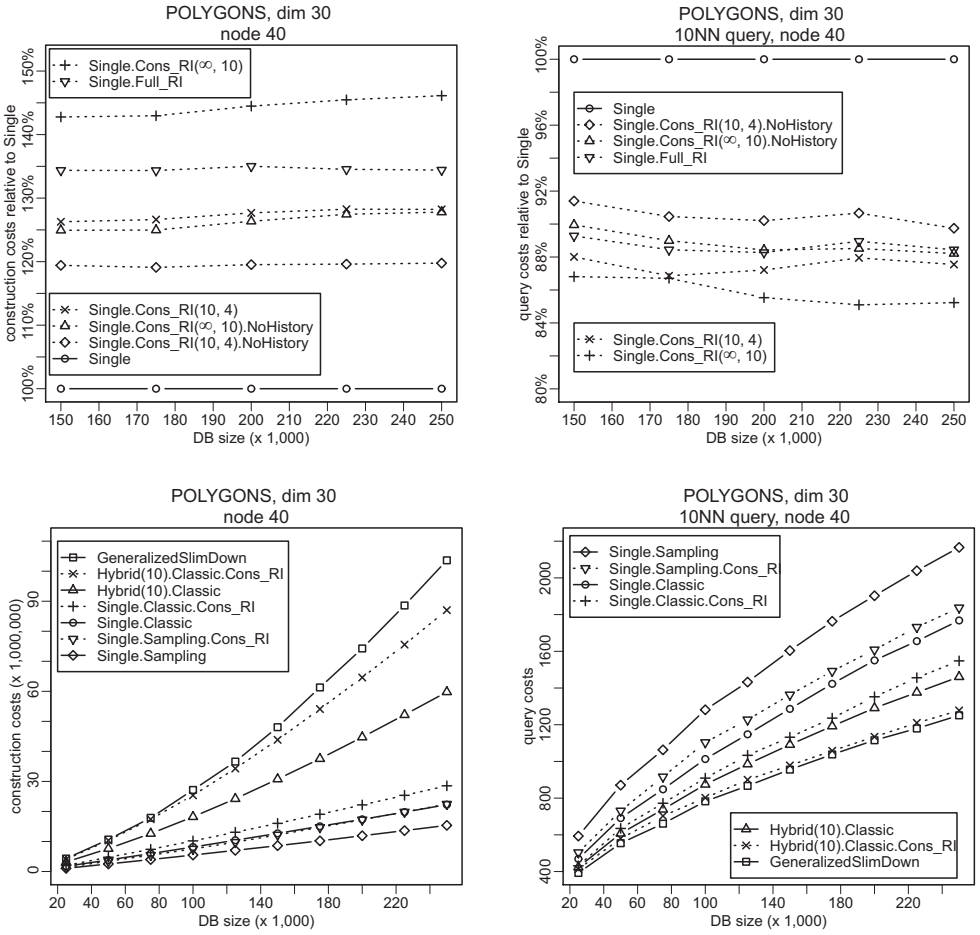

Fig. 12. Varying database size: (a) Construction costs (b) 10NN query costs

In the fourth experiment we have observed the impact of database size on indexing (see Figure 12). In the upper part the construction/query costs for the Single.* variants are presented relatively to the baseline Classic method. We can see that the improvement over the baseline is quite stable with increasing DB size in terms of construction costs, however, the query performance improves a bit faster with increasing DB size. Also note the Single.Cons_RI(10,4) proposed in this article clearly beats the Single.Full_RI (already proposed in [9]) in both construction and query costs. In the bottom part of Figure 12 the construction/query costs are presented in absolute numbers but now for the Hybrid.*, *.Sampling.* and GeneralizedSlimDown variants.

The fifth experiment (Figure 13) inspected the impact of the maximal number of reinserted entries (ground entries removed onto the stack, respectively) on the conservative forced reinsertion strategy. We considered various node capacities (20–60), while the results show that a reasonable value is around

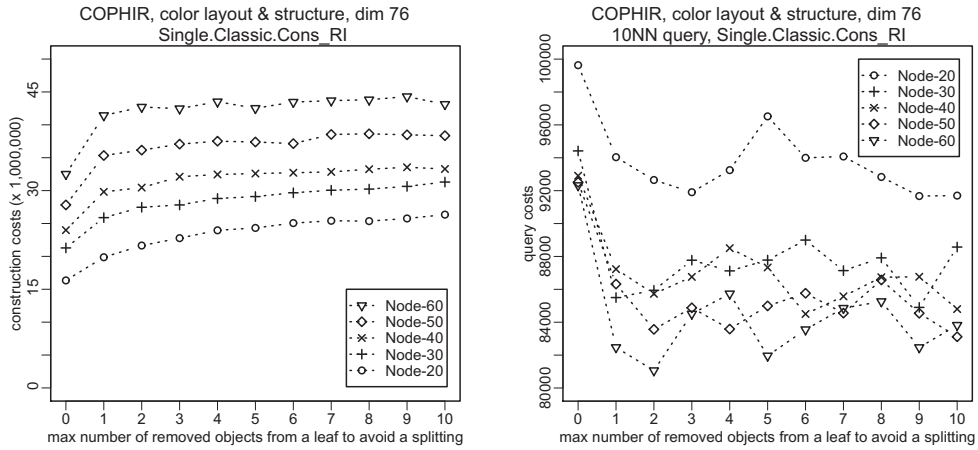

Fig. 13. Max. number of removed objs: (a) Construction costs (b) 10NN query costs

3 or 4 – higher values slightly increase construction costs but do not bring a clear improvement in querying (there is cca 0.5% variance in query costs).

costs	Hybrid(∞) .Cons_RI	Multi	Classic .Cons_RI	Sampling .Cons_RI	Classic	Sampling
construction	3,535,264,768	2,990,771,968	106,378,144	93,059,856	74,684,496	57,775,288
10NN query	31,132	39,540	81,019	80,749	94,081	102,898

Table 3. Results for COPHIR, one million objects, dim 12, node size 20

In Table 3 see the results of the sixth experiment, considering the largest COPHIR database in the testbed – one million objects, indexed within 6-level M-trees, node capacity 20. We can observe the queries on Hybrid(∞).Cons_RI performed 3× faster than those on Classic, however, for 47× higher construction costs. Nevertheless, the Sampling.Cons_RI achieved 15% reduction in query costs with respect to Classic for just 125% of Classic's construction costs.

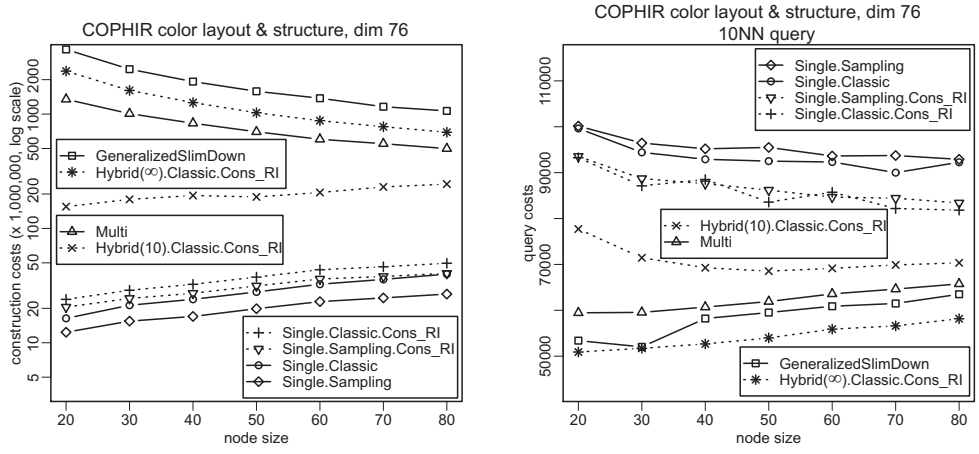

Fig. 14. Varying node capacity (a) Construction costs (b) 10NN query costs

In the last experiment we have tested the impact of various M-tree node capacities, see Figure 14. An interesting observation is that the expensive

techniques improve the construction costs with growing node capacity, while, on the other hand, the less expensive techniques slightly improve the query performance with growing node capacity.

The Figure 15 is maybe the most important outcome of the experimental results. Here the graphs from Figure 14 are aggregated into one, in order to show the construction vs. query performance trade-off. The closer to the bottom-left origin a technique is, the better the overall performance trade-off is. Hence, we can see the Single.Sampling.Cons_RI is clearly better than Single.Classic, and that Hybrid(∞).Classic.Cons_RI is much better than the GeneralizedSlim-Down. All the other techniques lie on a sort of skyline, hence, they represent meaningful trade-off choices applicable to various scenarios.

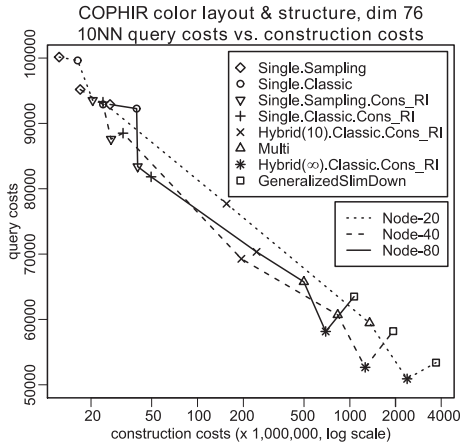

Fig. 15. Construction vs. query costs – aggregate results

6.4 Summary

The conservative forced reinsertions proved their usability when combined with "either-way" leaf selection. The query performance is always higher than that of techniques without forced reinsertions, while the construction is usually a few tens of percent more expensive. The conservative forced reinsertions also showed better results in both querying and construction when compared to the full forced reinsertions (as proposed in [9]). When used with single-way leaf selection, the conservative reinsertions are suitable also for indexing large and high-dimensional databases, where the feasibility of index construction is the crucial task. The hybrid-way leaf selection is beneficial for its scalability, while when used with unlimited branching factor and conservative forced reinsertions, it becomes a clear winner over the generalized slim-down algorithm, being much cheaper in construction and comparable or better in query costs.

7 Conclusions

In this article we have proposed two new techniques improving dynamic insertions in M-tree – the forced reinsertions and the hybrid-way leaf selection. Both of the techniques preserve logarithmic complexity of a single insertion, while they aim to produce more compact M-tree hierarchies. The proposed techniques experimentally proved their benefits. The experiments have also shown the problem of constructing compact M-trees cannot be solved by a simple solution or by a brute force. The increasing complexity of M-tree-related techniques developed over the last decade indicates it is worth to continue in designing even more complex algorithms within the realm of M-tree.

Acknowledgments. This research has been partially supported by Czech grants: "Information Society program" 1ET100300419 and GAUK 18208.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[2] B. Bustos and T. Skopal. Dynamic Similarity Search in Multi-Metric Spaces. In *Proceedings of the 8th ACM SIGMM International Workshop on Multimedia Information Retrieval (MIR'06), ACM Multimedia workshops, Santa Barbara, CA, USA*, pages 137–146. ACM Press, 2006.

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

[4] P. Ciaccia and M. Patella. Bulk loading the M-tree, In Proceedings of the 9th Australasian Database Conference (ADC), pages 15–26, Perth, Australia, 1998.

[5] P. Ciaccia and M. Patella. The M2-tree: Processing Complex Multi-Feature Queries with Just One Index. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97), Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[7] J. V. den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 461–470. Morgan Kaufmann, 2001.

[8] F. Falchi, C. Lucchese, R. Perego, and F. Rabitti. CoPhIR: COntent-based Photo Image Retrieval [http://cophir.isti.cnr.it/CoPhIR.pdf], 2008.

[9] J. Lokoč and T. Skopal. On Reinsertions in M-tree. In *Proceedings of the first international workshop on Similarity Search and Applicatons (SISAP'08), ICDE 2008 workshops, Cancun, Mexico*, pages 410–417. IEEE, 2008.

[10] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[11] A. P. Sexton and R. Swinbank. Bulk Loading the M-Tree to Enhance Query Performance. In *Proceedings of the 21st British National Conference on Databases (BNCOD'04), LNCS 3112*, pages 190–202. Springer, 2004.

[12] T. Skopal. Pivoting M-tree: A metric access method for efficient similarity search. In *4th workshop on Databases, Texts, Specifications and Objects (DATESO'04), CEUR vol. 98,* `www.ceur-ws.org/Vol-98`, pages 21–31, 2004.

[13] T. Skopal and D. Hoksza. Improving the performance of M-tree family by nearest-neighbor graphs. In *Proceedings of the 11th East-European Conference on Advances in Databases and Information Systems (ADBIS'07), LNCS 4690, Varna, Bulgaria*, pages 172–188. Springer, 2007.

[14] T. Skopal and J. Lokoč. NM-tree: Flexible Approximate Similarity Search in Metric and Non-metric Spaces. In *Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA'08), LNCS 5181, Turin, Italy*, pages 312–325. Springer, 2008.

[15] T. Skopal, J. Pokorný, M. Krátký, and V. Snášel. Revisiting M-tree Building Principles. In *Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS'04), LNCS 2798, Dresden, Germany*, pages 148–162. Springer, 2003.

[16] T. Skopal, J. Pokorný, and V. Snášel. Nearest Neighbours Search using the PM-tree. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA'05), LNCS 3453, Beijing, China*, pages 803–815. Springer, 2005.

[17] C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. In *Proceedings of the 7th Conference on Extending Database Technology (EDBT'00), LNCS 1777, Konstanz, Germany*, pages 51–65. Springer, 2000.

[18] T. G. Vespa, C. Traina, and A. J. M. Traina. Bulk-loading Dynamic Metric Access Methods. In *Proceedings of the 22nd Brazilian Symposium on Databases (SBBD'07), ACM SIGMOD DiSC*, pages 160–174, 2007.

[19] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[20] X. Zhou, G. Wang, J. Y. Xu, and G. Yu. M+-tree: A New Dynamical Multidimensional Index for Metric Spaces. In *Proceedings of the 14th Australasian database conference (ADC'03)*, pages 161–168, 2003.