D-cache: Universal Distance Cache for Metric Access Methods

Tomáš Skopal, Jakub Lokoč, and Benjamin Bustos

Abstract—The caching of accessed disk pages has been successfully used for decades in database technology, resulting in effective amortization of I/O operations needed within a stream of query or update requests. However, in modern complex databases, like multimedia databases, the I/O cost becomes a minor performance factor. In particular, metric access methods (MAMs), used for similarity search in complex unstructured data, have been designed to minimize rather the number of distance computations than I/O cost (when indexing or querying). Inspired by I/O caching in traditional databases, in this paper we introduce the idea of *distance caching* for usage with MAMs – a novel approach to streamline similarity search. As a result, we present the *D-cache*, a main-memory data structure which can be easily implemented into any MAM, in order to spare the distance computations spent by queries/updates. In particular, we have modified two state-of-the-art MAMs to make use of D-cache – the M-tree and Pivot tables. Moreover, we present the D-file, an index-free MAM based on simple sequential search augmented by D-cache. The experimental evaluation shows that performance gain achieved due to D-cache is significant for all the MAMs, especially for the D-file.

Index Terms—metric indexing, similarity search, distance caching, metric access methods, D-cache, MAM, index-free search

1 Introduction

In database technology, the majority of problems concerns the efficiency issues, that is, the performance of a DBMS. For decades, the number of accesses to disk (required by I/O operations) was the dominant factor affecting the DBMS performance. There were developed indexing structures [1], [2], storage layouts [3], and also disk caching/buffering techniques [4]; all of these designs aimed to minimize the number of physical I/Os spent within a database transaction flow. In particular, disk caching was proven to be extremely effective in situations where access to some disk pages happens repeatedly during a single runtime session.

However, the situation is dramatically different in modern complex databases consisting of snapshots of nature (i.e., images, sounds, or other signals), like multimedia databases, bioinformatic databases, time series, etc. Here we often adopt the similarity search within the content-based retrieval paradigm, where a similarity function $\delta(q, o)$ serves as a measure saying how much a database object $o \in \mathbb{S}$ is relevant to a query object $q \in \mathbb{U}$ (where $\mathbb{S}$ is the database and $\mathbb{U}$ is the object universe, $\mathbb{S} \subset \mathbb{U}$). To speed up similarity search in such a database, there have been many indexing techniques developed – some of them domain-specific and some others more general. Also, there were distributed indexing techniques developed [5] that use parallelism to speed up similarity queries. An important fact is that the retrieval performance of such a system is more affected by CPU cost than by I/O cost. In particular, in similarity-search community the computation of a single value δ is employed as the logical unit for indexing/retrieval cost, because of its dominant impact on the overall performance [6], [7]. Thus, the I/O cost is mostly regarded as a minor component of the overall cost. The number of computations δ needed to answer a query (or to index a database) is referred to as the *computation cost*.

Among general techniques, the *metric access methods* (MAMs) are suitable in situations where the similarity measure δ is a *metric* distance (in mathematical meaning). The metric properties (1), (2), (3), (4) allow us to organize a database $\mathbb{S}$ within equivalence classes, embedded in a data structure which is stored in an *index file*.

$$\begin{aligned} \delta(x,y) &= 0 \Leftrightarrow x = y && \text{identity (1)} \\ \delta(x,y) &> 0 \Leftrightarrow x \neq y && \text{non-negativity (2)} \\ \delta(x,y) &= \delta(y,x) && \text{symmetry (3)} \\ \delta(x,y) + \delta(y,z) &\geq \delta(x,z) && \text{triangle inequal.(4)} \end{aligned}$$

The index is later used to quickly answer typical similarity queries – either a *k nearest neighbors* (kNN) query like "return the 3 most similar images to my image of a horse", or a *range query* like "return all voices more similar than 80% to the voice of a nightingale". In particular, when issued a similarity query, the MAMs exclude many non-relevant equivalence classes from the search (based on metric properties of δ), so only several candidate classes of objects have to be exhaustively (sequentially) searched. In consequence, searching a small number of candidate classes turns out in reduced computation cost of the query. For a comprehensive survey on MAMs, we refer to [7], [8] or monographs [6], [9]. Again, we have to emphasize the assumption on computationally *expensive* distance metric δ (i.e., $> O(n)$, where n is the size of object o_i). In other words, the real time spent in distance

- *T. Skopal and J. Lokoč are with the Dept. of Software Engineering, Faculty of Mathematics and Physics, Charles University, Malostranske nam. 25, 118 00 Prague, Czech Republic. E-mail: {skopal, lokoc}@ksi.mff.cuni.cz.*
- *B. Bustos is with the Dept. of Computer Science, University of Chile, Av. Blanco Encalada 2120, Santiago, Chile. E-mail: bebustos@dcc.uchile.cl*

Manuscript received XXXXXX XX, 2010; revised XXXX XX, 2010; accepted XXXX XX, 2010
Recommended for acceptance by XXXXXX.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-XXXXXXXXXX. Digital Object Identifier no. XXXXXXXXXXXX

computations is assumed to dominate the real time spent in other parts of MAMs' algorithms (including I/O cost).

1.1 Motivation for Distance Caching

Importantly, after a metric index is built, the existing MAMs solve every query request *separately*, that is, every query is evaluated as it would be the only query to be answered. In general, no optimization for a *stream of queries* (query requests spread in time) has been considered for MAMs up to date. Instead, huge efforts were given to "materializing" the filtering knowledge into the index file itself.

In this paper, we change this paradigm and propose a structure for *caching distances* computed during the current runtime session. The distance cache ought to be an analogy to the classic disk cache widely used in database management to optimize I/O cost. Hence, instead of sparing I/Os, the distance cache should spare distance computations. A desired feature of distance cache should be its universal usage with all MAMs, similarly like disk caching is universal for standard I/O management. The main idea behind the distance caching resides in approximating the requested distances by providing their lower and upper bounds "for free". Since some "useful" distances could have been computed during previous querying/indexing, such distances could still "sit" in the distance cache and thus could be used to infer (more or less tight) approximations of distances we request.

1.2 Paper Contribution

We present D-cache (distance cache), a tool for general metric access methods that helps to reduce the cost of both, indexing and querying. The basic task of D-cache is to cheaply determine tight lower- and upper bound of an unknown distance between two objects, based on stored distances computed during previous querying and/or indexing. Although the D-cache was already introduced in our preliminary work [10], it was applied in a more narrowed context – as a tool for efficient index-free similarity search (resulting in a new method, the D-file). Moreover, in this paper we not only employ the D-cache in various MAMs, but we present a completely redesigned D-cache variant that is more effective (provides tighter lower/upper bounds) and also more efficient (faster bound determination) than the previous version.

2 Metric Access Methods

In the following, we consider three out of dozens of existing MAMs – the *sequential file* (a trivial MAM), the *Pivot Tables*, and the *M-tree*. Later in the paper we will consider extensions of these MAMs by the announced D-cache structure.

2.1 Sequential File

The sequential file is simply the original database, where any query involves a sequential scan over all the database objects. For a query object q and every database object o_i, a distance $\delta(q, o_i)$ must be computed (regardless of query selectivity). Although this kind of "MAM" is not very smart, it does not require any index (and no indexing), which can be useful in many situations (as discussed in Section 4.1).

2.2 Pivot Tables

A simple but efficient solution to similarity search represent methods called *pivot tables* (or distance matrix methods). In general, a set of p objects (so-called pivots) is selected from the database, while for every database object a p-dimensional vector of distances to the pivots is created. The vectors belonging to the database objects then form a distance matrix – the pivot table. When performing a range query (q, rad), a distance vector for the query object q is determined the same way as for a database object. From the query vector and the query radius rad a p-dimensional hyper-cube is created, centered in the query vector (query point, actually) and with edges of length $2rad$. Then, the range query is processed on the pivot table, such that vectors of database objects that do not fall into the query cube are filtered out from further processing. The database objects that cannot be filtered have to be subsequently checked by the usual sequential search.

There have been many MAMs developed based on pivot tables. The AESA [11] treats all the database objects as pivots, so the resulting distance matrix has quadratic size with respect to the database size. Also, the search algorithms of AESA is different, otherwise the determination of a query vector would turn out in sequential scan of the entire database. The advantage of AESA is empirical average constant complexity of nearest neighbor search. The drawback is quadratic space complexity and also quadratic time complexity of indexing (creating the matrix) and of the external CPU cost (loading the matrix when querying). The LAESA [12] is a linear variant of AESA, where the number of pivots is assumed far smaller than the size of the database (so that query vector determination is not a large overhead). The concept of LAESA was implemented many times under different conditions, we name, e.g., TLAESA [13] (pivot table indexed by GH-tree-like structure), Spaghettis [14] (pivot table indexed by multiple sorted arrays), OMNI family [15] (pivot table indexed by R-tree), PM-tree [16] (hybrid approach combining M-tree and pivot tables). In the rest of the paper we consider the simplest implementation of pivot tables – the original LAESA.

2.3 M-tree

The *M-tree* [17] is a dynamic index structure that provides good performance in secondary memory (i.e., in database environments). The M-tree is a hierarchical index, where some of the data objects are selected as centers (local pivots) of ball-shaped regions, while the remaining objects are partitioned among the regions in order to build up a balanced and compact hierarchy of data regions, see Figure 1. Each region (subtree) is indexed recursively in a B-tree-like (bottom-up) way of construction.

The inner nodes of M-tree store *routing entries* $rout_l(o_i) = [o_i, rad_{o_i}, \delta(o_i, Par(o_i)), ptr(T(o_i))]$, where $o_i \in \mathbb{S}$ is a data object representing the center of the respective ball region, rad_{o_i} is a *covering radius* of the ball, $\delta(o_i, Par(o_i))$ is the so-called *to-parent* distance (the distance from o_i to the object of the parent routing entry), and finally $ptr(T(o_i))$ is a pointer to the entry's subtree. The data is stored in the leaves of M-tree. Each leaf contains *ground entries* $grnd(o_i) =$

$[o_i, \delta(o_i, Par(o_i))]$, where $o_i \in \mathbb{S}$ is an indexed database object and $\delta(o_i, Par(o_i))$ is, again, the to-parent distance.

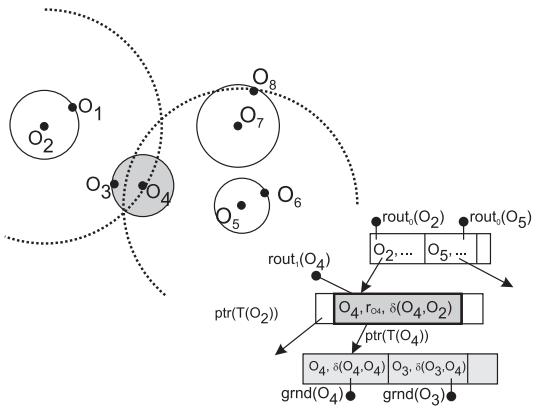

Fig. 1. M-tree (hierarchical space decomposition and the tree structure).

Range and kNN queries are implemented by traversing the tree, starting from the root. Those nodes are accessed, whose parent regions (described by the routing entries) are overlapped by the query ball (q, rad). In case of a kNN query the radius rad is not known beforehand, so we have to additionally employ a heuristics to dynamically decrease the radius during the search (initially set to ∞). The kNN algorithm performs a best-first traversal of the index, where regions are accessed in the order of increasing lower bound distance to q.

2.3.1 M-tree Construction

In the original M-tree proposal [17], the index was constructed by multiple dynamic insertions, which consisted of two steps. First, an appropriate leaf node for the newly inserted object is found by traversing a single path in the tree (so-called *single-way* leaf selection). Second, if a leaf gets overfull after the insertion, it is split, such that two objects from the split leaf are selected as centers of the new two leafs, while the remaining objects within the split leaf are distributed among the new leafs. Simultaneously, the new centers form new routing entries that are inserted into the parent node (if the parent gets overfull as well, the splitting proceeds recursively).

In addition to the original M-tree, in this paper we consider also recent advanced techniques of dynamic M-tree construction [18]. In particular, we consider the *multi-way* leaf selection. Although the multi-way selection is more expensive than the single-way variant, the target leaf is more appropriate for the newly inserted object. Specifically, a point query is issued, such that from all the "touched" leaves the selected one has its center closest to the newly inserted object. Another improvement in M-tree construction is adopting the well-known technique of *forced reinsertions*. When a leaf is about to split after a new insertion, some objects are removed from the leaf and inserted again into the M-tree under the hope they will not all arrive into the same leaf again (thus avoiding the split). Both of the advanced construction techniques (multi-way leaf selection and forced reinsertions) lead to more compact M-tree hierarchies, which, in turn, leads to faster query processing.

3 D-CACHE

We propose a non-persistent (main-memory) structure called *D-cache* (distance cache), that stores distances already computed by a MAM. We consider a single runtime session of a search engine, that is, a contiguous usage of a MAM for a sequence of queries, insertions, or both. The track of distance computations is stored as a set of triplets, each of form:

$$[id(o_i), id(o_j), \delta(o_i, o_j)]$$

where $id(o_i), id(o_j)$ are unique identifiers of objects o_i, o_j, and $\delta(o_i, o_j)$ is their distance.

To distinguish between the roles of "active and passive objects", we use the term *runtime object*, that denotes an object that is currently subject to an operation on MAM (either query or insertion). Once the operation is finished, the respective object becomes *past runtime object*, meaning either a regular database object (after an insertion) or a past query object. All objects are uniquely identified, regardless of their role (query, inserted object, database object). For instance, the runtime objects could be identified by the order they enter the index (forever, i.e., also for their past-runtime role), where as "entering" we mean either an insertion or a query.

Instead of considering a set of triplet entries, we can view the content of D-cache as a sparse matrix

$$D = \begin{array}{c} \\ o_1 \\ o_2 \\ o_3 \\ \dots \\ o_m \end{array} \begin{array}{c} \begin{array}{ccccc} o_1 & o_2 & o_3 & \dots & o_n \end{array} \\ \begin{pmatrix} & \delta_{12} & \delta_{13} & \dots & \\ \delta_{21} & & & \dots & \delta_{2n} \\ & & & \dots & \\ \dots & \dots & \dots & \dots & \dots \\ \delta_{m1} & & \delta_{m3} & \dots & \end{pmatrix} \end{array}$$

where the rows and columns refer to objects, and the cells store the respective object-to-object distances. Naturally, as new runtime objects appear during the session, the matrix gets larger (in number of rows and/or columns). At the beginning of the session the matrix is empty, while during the session the matrix is being extended and filled. Note that runtime objects do not have to be external, that is, a runtime object could originate from the database (e.g., a query or a re-inserted object). From this point of view, an object could have different roles at different moments, however, the unique objects identification ensures the D-cache content is correct.

Because of frequent insertions of triplets into D-cache, the matrix should be efficiently updatable. Moreover, due to operations described in the next subsection, we should be able to quickly retrieve the value of a particular cell.

3.1 Principle of D-cache

The desired functionality of D-cache is twofold:

First, given a pair runtime object/database object $\langle r, o \rangle$, the D-cache should quickly determine the *exact* value $\delta(r, o)$ in case the distance is stored in the D-cache. However, as the exact value could only be found when the actual distance was already computed previously in the session, this functionality is limited to rather special cases, like re-indexing of data objects (or index rearrangements), repeated queries or querying by database objects.

The **second** functionality, which is the main D-cache contribution, is more general. Given a runtime object r and a database object o on input, the D-cache should quickly determine the tightest possible *lower* or *upper bound* of $\delta(r, o)$ without the need of an explicit distance computation. This cheap determination of lower/upper bound distances then serves a MAM in order to filter out a non-relevant database object or even a whole part of the index. Let us denote a lower-bound distance of $\delta(r, o)$ as $\delta_{LB}(r, o) \leq \delta(r, o)$ and an upper-bound distance as $\delta_{UB}(r, o) \geq \delta(r, o)$.

In order to facilitate the second functionality, we have to feed the D-cache with relevant information about the involved objects. In particular, we would like to know distances to some past runtime objects dp_i^r which are very close to or very far from the current runtime r, that is, suppose for a while we know some $\delta(dp_1^r, r), \delta(dp_2^r, r), \ldots$ These past runtime objects will serve as *dynamic pivots* made-to-measure to r. Formally defined, $dp_i^r \in DP \subseteq PR \subset \mathbb{U}$, where PR is the set of all past runtime objects within the current session and DP is an actual set of selected dynamic pivots (see the next section). Regarding the size of DP, we could choose either $DP = PR$, or set a fixed size $|DP| = k < |PR|$. Note that dynamic pivots could originate outside the database $\mathbb{S}$ (necessary for queries and newly inserted objects).

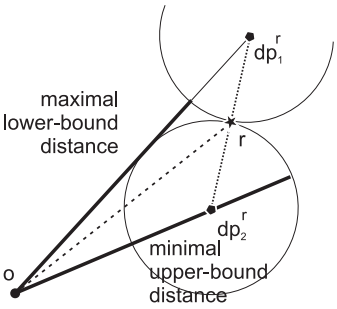

Fig. 2. Lower/upper bounds to $\delta(r, o)$.

Since the dynamic pivots are supposed either close to r or far from r, they should be effective for pruning by a MAM (they provide tight approximations of $\delta(r, o_i)$ distances). After the dynamic pivots are selected, the lower/upper bound distances are constructed using the distances $\delta(dp_i^r, o)$ still "sitting" in the D-cache matrix, where they were inserted earlier during the session. In particular, with respect to dp_i^r and available distances $\delta(dp_i^r, o)$ in the matrix, $max_{dp_i^r}\{|\delta(dp_i^r, o) - \delta(dp_i^r, r)|\}$ is the tightest lower-bound distance $\delta_{LB}(r, o)$. Similarly, $min_{dp_i^r}\{\delta(dp_i^r, o) + \delta(dp_i^r, r)\}$ is the tightest upper-bound distance $\delta_{UB}(r, o)$. See the situation in Figure 2.

3.1.1 Selection of Dynamic Pivots

In the past decade, there were many sophisticated techniques for selection of effective pivots developed, allowing an efficient similarity search [19], [20]. This classic approach assumes the pivot selection procedure as a part of the indexing/preprocessing phase (e.g., before the distance matrix for pivot tables is established). However, in D-cache the dynamic pivots have to be selected at the moment a query or insertion starts. So, there is not much room for preprocessing, such as an expensive pivot selection, even though we select pivots from a rather small set of past runtime objects. Hence, we propose the following cheap pivot selection technique.

We need to choose some k runtime objects from all of the past runtime objects before the current runtime processing actually starts (i.e., before processing a query or insertion). Based on observations taken from the preliminary work on D-cache [10], we consider just the *recent selection policy*. That is, the k *most recent* runtime objects are selected as dynamic pivots, because it is more probable that recent runtime objects have more distances stored in the D-cache than the older ones (i.e., not replaced by other distances, see Section 3.3.2).

After the dynamic pivots are determined, their distances to r have to be computed. Note that this is the *only moment* where some extra distances are explicitly computed, that would not be computed when not using D-cache.

3.2 Distance Matrix Structure

Because the main memory is always limited and the distance matrix could expand to an enormous size, we need to choose a compact data structure that consumes a user-defined portion of main memory. In order to provide also fast retrieval, the D-cache implements the distance matrix as a linear *hash table* consisting of entries $[id1, id2, \delta(o_{id1}, o_{id2})]$. The hash key (pointing to a position in the hash table) is derived from the two ids of objects whose distance is being retrieved or stored.

In addition, there is a constant-size *collision interval* defined, that allows to move from the hashed position to a more suitable one (due to replacement policies, see below). However, in order to keep the D-cache as fast as possible, the collision interval should be very small, preferably just one position in the hash table (i.e., only the hashed position).

3.2.1 Hashing Function

To achieve uniform distribution of the hashed distances, we consider two variants of hashing function f, both taking two integer numbers $id1, id2$ as arguments (the ids of objects).

Simple. The faster variant of f multiplies the ids (modulo the size D of hash table), i.e., $f(id1, id2) = (id1 \cdot id2) \bmod D$. The motivation here is that we expect the ids entering the hashing function are random combinations, so the simple multiplication should produce distribution uniform enough.

Universal. A slightly slower variant of f is based on the Simple variant and on *universal hashing* [21]. Let p be a large prime ($p > D$), and let a, b be two random integer numbers smaller than p. All the numbers p, a, b are fixed during D-cache lifetime. Then, the hashing function is defined as $f(id1, id2) = ((a \cdot id1 \cdot id2 + b) \bmod p) \bmod D$.

3.3 Operations on D-cache

The D-cache is initialized by a MAM when loading the index (the session begins). Besides the initialization, the D-cache is also notified by a MAM whenever a new query/insertion is to be started (the MAM calls method StartRuntimeProcessing on D-cache). At that moment, new runtime object r is announced to be processed, which also includes the computation of distances from r to the k actual dynamic pivots dp_i^r.

3.3.1 Distance Retrieval

The main D-cache functionality is operated by methods GetDistance and GetLowerBoundDistance[1], see Algorithm 1.

Algorithm 1: (GetDistance, GetLowerBoundDistance)

```
double GetDistance(r, o_i) {
  let minId = min(id(r), id(o_i)), maxId = max(id(r), id(o_i))
  let CI = size of collision interval
  let h = GetHash(minId, maxId)                    // hashing function f, see Sec. 3.2.1
  for i = 1 to CI                                  // every + is modulo hash table size
    if hashTable[h + i].id1 = minId and hashTable[h + i].id2 = maxId then
      return hashTable[h + i].distance
  return nil }

double GetLowerBoundDistance(r, o_i) {
  let k be the number of pivots to use
  let DP be the set of k dynamic pivots and their distances to r
  if GetDistance(r, o_i) ≠ nil then
    return GetDistance(r, o_i)
  let value = 0
  for each p in DP do
    if GetDistance(p, o_i) ≠ nil then
      value = max(value, |GetDistance(p, o_i) −δ(r, p)|)
  return value }
```

The number of dynamic pivots ($k = |DP|$) used to evaluate GetLowerBoundDistance is set by the user, while this parameter is an exact analogy to the number of pivots used by Pivot tables, e.g., LAESA. There exists no general rule for the automatic determination of the number of pivots [19], [20], especially when minimizing the realtime cost rather than just the number of distance computations. In general, the effective number of pivots depends on the (expected) size of the database, its intrinsic dimensionality (see Section 6.1.1), the computational complexity of the used metric, the pivot set quality itself, etc. The same reasons apply also for D-cache.

3.3.2 Distance Insertion

Every time a distance $\delta(r, o_i)$ is computed by the MAM, the triplet $[id(r), id(o_i), \delta(r, o_i)]$ is inserted into the D-cache (the MAM calls method InsertDistance on D-cache). Since the storage capacity of D-cache is limited, at some moment the collision interval in the hash table for a newly inserted distance entry is full. Then, some older entry within the collision interval has to be replaced by the new entry. Or, alternatively, if it turns out the newly inserted distance is less useful than all the distances in the collision interval, the insertion of the new distance is canceled.

Note that we should prioritize replacing of such entries $[id1, id2, \delta(o_{id1}, o_{id2})]$ where none of the objects o_{id1}, o_{id2} belongs to the current set of k dynamic pivots anymore. Naturally, the distances of such *obsolete entries* cannot be effectively utilized to determine a lower- or upper bound distance, because for a current runtime r only the distances to the k most recent runtimes are useful. In particular, we consider two policies for replacement by a new entry:

Obsolete. The first obsolete entry (i.e., not containing id of a current dynamic pivot) in the collision interval is replaced. In case none of the entries in the collision interval is obsolete, the first entry is replaced by the new entry.

ObsoletePercentile. This policy includes two steps. First, we try to replace the first obsolete entry as in the Obsolete policy. If none of the entries is obsolete, we replace an entry with the least useful distance. As we have mentioned in Section 3.1, a good pivot is either very close to or very far from the database objects. Because the entries in D-cache consist of distances from dynamic pivots to database objects, we should preserve entries with large and small distances and get rid of those close to a "middle" distance. Hence, among all entries in the collision interval the entry that is closest to the "middle" distance is the least useful, thus it is replaced. Of course, it might turn out the least useful distance (closest to the "middle" distance) is that of the newly inserted entry. In such case the D-cache is not updated by the new entry at all.

Ideally, the "middle" distance should be represented by the *median* distance among objects in the database, that is, a distance value d_m where 50% of the computed distances are greater and the other 50% distances are smaller than d_m. However, it might turn out that an optimal value for D-cache is not the median distance but a distance belonging to another percentile. Hence, we relax the term "middle" distance to allow not only the fixed median distance (i.e., 50% percentile) but also a distance belonging to a user-defined percentile[2].

The method InsertDistance, including entry replacement, is precisely described in Algorithm 2. The method IsObsolete checks if either of the two ids appears in the actual set of k dynamic pivots' ids (if not, it is an obsolete entry).

Algorithm 2: (InsertDistance)

```
InsertDistance(r, o_i, d_new) {
  let minId = min(id(r), id(o_i)), maxId = max(id(r), id(o_i))
  let CI = size of collision interval
  let d_m be the distance belonging to a user-defined percentile
  let h = GetHash(minId, maxId)
  let finalH = h
  if entry replacement heuristics is Obsolete then {
    for i = 1 to CI                                // every + is modulo hash table size
      if IsObsolete(hashTable[h+i]) then
        finalH = h + i
        break for
  } else if entry replacement heuristics is ObsoletePercentile then {
    let fitness = 0
    let dNew = |d_m − d_new|
    for i = 1 to CI                                // every + is modulo hash table size
      if not IsObsolete(hashTable[h + i]) then
        dOld = |d_m − hashTable[h + i].distance|
        if dNew > dOld and fitness < dNew − dOld then
          finalH = h + i
          fitness = dNew − dOld                    // the greater fitness, the better
      else                                         // obsolete entry found
        finalH = h + i, fitness = 1
        break for
    if fitness = 0 then return }                   // do not replace (new distance is bad)
  set hashTable[finalH] = [minId, maxId, d_new] }  // finally, replace the entry
```

3.4 Filtering by D-cache

The D-cache can be widely used with any metric access method. In particular, MAMs index data either in ball-shaped metric regions (e.g., (m)vp-tree, (P)M-tree, D-index) or in Voronoi-based regions (e.g., gh-tree, GNAT). Hence, there are basically three low-level *filtering predicates* used by MAMs

1. GetUpperBoundDistance is similar, but the value is initialized to ∞ and updated as value = min(value,**GetDistance**$(p, o_i) + \delta(r, p)$).

2. The calculation of percentile distances d_m is obtained for free during the indexing phase of a MAM, using the distance distribution histogram.

to answer a query (or to insert new database object) – two predicates for ball-shaped and one for Voronoi-based regions. The most common queries (range and kNN) have also the shape of a ball.

When employing D-cache, the MAMs' filtering predicates can be weakened to be used with lower/upper bound distances inferred from D-cache, instead of computing an exact δ distance. Generally, the predicates can be weakened such that any form $\delta(\cdot,\cdot)+... <$ is turned into $\delta_{UB}(\cdot,\cdot)+... <$ and any $\delta(\cdot,\cdot)-... >$ into $\delta_{LB}(\cdot,\cdot)-... >$. This adjustment is correct, since it underestimates the filtering hits, that is, weakened form implies the original one, but not vice versa (see Figure 3).

3.4.1 Filtering of Ball-shaped Regions

The ball-shaped regions are generally of two kinds – a simple ball and/or a ring.

(A) Ball data regions

Querying: Having a query ball (q, rad_q) and a ball-shaped data region (o_i, rad_{o_i}), the data region can be excluded (filtered) from the search if the two balls do not overlap, that is, in case that predicate

$$\delta(q, o_i) \quad > \quad rad_q + rad_{o_i} \tag{1}$$

is true (see Figure 3a). Note that this simple predicate applies also on filtering database objects themselves (rather than regions), considering just DB object o_i, i.e. $rad_{o_i} = 0$.

Indexing: Let us now consider q as a new object to be inserted and $rad_q = 0$. Then the predicate (1) can be used to filter a data region which cannot cover the new object (without enlarging the radius rad_{o_i}).

D-cache usage: Prior to an application of predicate (1), a MAM could use its weakened form

$$\delta_{LB}(q, o_i) \quad > \quad rad_q + rad_{o_i} \tag{2}$$

Since D-cache provides the lower bound $\delta_{LB}(q, o_i)$ for free, the index region (o_i, rad_{o_i}) could be filtered out by predicate (2) without the need of computing $\delta(q, o_i)$ otherwise required to apply predicate (1).

(B) Ring data regions

Some MAMs ((m)vp-tree, (P)M-tree, D-index) combine two balls to form a ring, which is a pair of two concentric balls – the smaller one is regarded as a hole in the bigger. In order to determine an overlap with query ball (or inserted object), the predicate (1) alone cannot be used to determine that a query ball is entirely inside the hole. Hence, we use predicate

$$\delta(q, o_i) \quad < \quad rad_{o_i} - rad_q \tag{3}$$

to determine whether the query ball is entirely inside the hole (see Figure 3b). A query ball is not overlapped by a ring region in case (1) is true for the bigger ball or (3) is true for the smaller ball (hole). For insertion of a new database object the predicates (1) and (3) are used in a similar way.

D-cache usage: Prior to an application of predicate (3), a MAM could use its weakened form

$$\delta_{UB}(q, o_i) \quad < \quad rad_{o_i} - rad_q \tag{4}$$

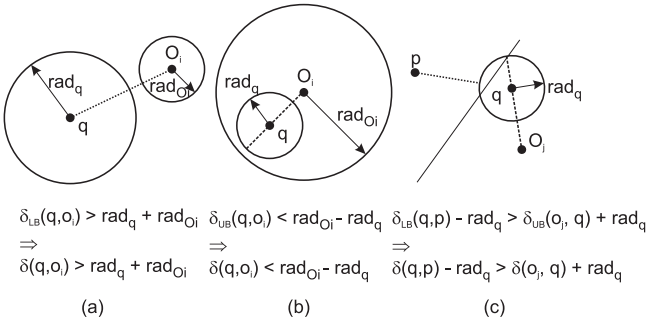

Fig. 3. (a) Ball-ball overlap (b) Hole-ball containment (c) Halfspace-ball overlap.

3.4.2 Filtering of Voronoi-based Regions

Several MAMs (gh-tree, GNAT, M-index) partition the metric space by use of a border composed of "Voronoi hyperplanes". Given m pivot objects, the border is formed by all such points of the universe, which are equally distant to two of the pivot objects and farther from the rest of objects.

A region assigned to pivot object p does not overlap a query region (q, rad_q) if the following predicate is true

$$\forall o_j : \quad \delta(q, p) - rad_q \quad > \quad \delta(q, o_j) + rad_q \tag{5}$$

where $\forall o_j$ are the remaining pivot objects (see Figure 3c).

D-cache usage: Prior to an application of predicate (5), a MAM could use its weakened form

$$\forall o_j : \quad \delta_{LB}(q, p) - rad_q \quad > \quad \delta_{UB}(q, o_j) + rad_q \tag{6}$$

3.5 Use of D-cache in Approximate Similarity Search

In addition to exact search by MAMs, the D-cache may also be used to improve the efficiency of approximate similarity search techniques [22]. In these techniques, the search algorithm saves search cost at the cost of possibly not retrieving the exact answer (i.e., all relevant objects for the given query). That is, they provide a trade-off between the efficiency and the effectiveness of the similarity search.

Similarly to search algorithms in MAMs, approximate algorithms can take advantage of lower or upper bound distance estimations to avoid distance computations. For example, the probabilistic incremental search approach [23] fixes a number of distance computations to be performed by the search algorithm. Once a distance is computed, the algorithm decides if it must continue searching or not in a particular branch of the search hierarchy. By using D-cache, the discarding process could be done without actually computing that distance (using the returned lower bound distance), thus saving it for further searching in the hierarchy. This will improve the effectiveness of the search, as more branches of the hierarchy will be visited.

3.6 Analysis of D-cache Performance

A fast implementation of D-cache functionality is crucial for its efficient employment by MAMs. Specifically, this requirement applies to the function GetLowerBoundDistance and method InsertDistance due to their frequent use during querying/indexing. A D-cache-enhanced MAM would be

faster in realtime only in case the D-cache overhead would not be dominant. In particular, employing computationally expensive distance functions δ promises the speed-up in realtime will approach the reduction in distance computations (which is the theoretical speed-up maximum). Although the mentioned functions do not compute even a single distance δ, for an improper parameterization their realtime overhead might be significant. First of all, the overall cost of GetLowerBoundDistance and InsertDistance is proportional to the number of dynamic pivots k. Thus, to obtain effective usage of D-cache, k must be reasonably small. Second, the size of collision interval can heavily affect the D-cache performance, because sequential processing of the collision interval affects the realtime cost linearly. Although a large collision interval usually leads to better replacement of distances, the resulting heavy slowdown may not be a good trade-off. Third, the hashing function is called frequently in GetLowerBoundDistance, so it should be as fast as possible but, at the same time, providing good enough distribution of keys.

In the experimental evaluation, we present different settings affecting the discussed performance issues. Basically, for smaller D-cache the collision interval of size 1 and simple hashing is the best, while for larger D-cache the interval of size 5 and universal hashing is slightly better. Regarding the dynamic pivots, their optimal number is heavily dependent on the database settings, while in our experiments it turns out that several tens to a few hundred pivots perform the best.

4 Enhancing MAMs by D-cache

In this section, we discuss the modifications of three MAMs that take advantage of D-cache for both querying and indexing.

4.1 Enhancing Sequential Search – the D-file

Although not a proper MAM, the sequential search over the database can be enhanced by D-cache to speed the search. In Algorithms 3, 4 see the adjusted range and kNN query.

Algorithm 3: (D-file range query)

```
set ScanRangeQuery(q, rad_q) {
  Dcache.StartRuntimeProcessing(q)
  for each o_i in database do
    if Dcache.GetLowerBoundDistance(q, o_i) ≤ rad_q then        // D-cache filt.
      compute δ(q, o_i); Dcache.InsertDistance(q, o_i, δ(q, o_i))
      if δ(q, o_i) ≤ rad_q then add o_i to the query result }    // basic filtering
```

We have to emphasize that the D-cache together with sequential search could be used as a standalone metric access method that requires no indexing at all. We call the enhanced sequential search as the *D-file*, introduced recently in its preliminary version as a tool for index-free similarity search [10]. Hence, it could be used in situations where indexing is not possible or too expensive. Generally, any form of indexing requires at least linear time to construct an index for a database (but typically more, e.g., $O(n\ log\ n)$ or $O(n^2)$). Thus, indexing is beneficial just in case we assume many queries, so the indexing cost will be amortized by the overall decreased query cost.

Algorithm 4: (D-file kNN query)

```
set kNNQuery(q, k) {
  Dcache.StartRuntimeProcessing(q)
  let NN be array of k pairs [o_i, δ(q, o_i)] sorted asc. wrt δ(q, o_i),
                                  initialized to NN = [[−, ∞], ..., [−, ∞]]
  let rad_Q denotes the actual distance component in NN[k]
  for each o_i in database do
    if Dcache.GetLowerBoundDistance(q, o_i) ≤ rad_q then   // D-cache filtering
      compute δ(q, o_i); Dcache.InsertDistance(q, o_i, δ(q, o_i))
      if δ(q, o_i) ≤ rad_q then insert [o_i, δ(q, o_i)] into NN        // basic filtering
  return NN as result }
```

4.1.1 Motivation for Index-free Similarity Search

In some scenarios, the indexing (and even dynamic updates) represents an obstacle. In the following, we briefly discuss three such scenarios.

"Changeable" databases. In many applications we encounter databases intensively changing over time, like streaming databases, archives, logs, temporal databases, where new data arrives and old data is discarded frequently. Alternatively, we can view any database as "changeable" if the proportion of changes to the database exceeds the number of query requests. In highly changeable databases the indexing efforts lose their impact, since the expensive indexing is compensated by just a few efficient queries. In the extreme case (e.g., sensory-generated data), the database could have to be massively updated in real time, so that any indexing is unpractical.

Isolated searches. In complex tasks, e.g., in data mining, a similarity query over a single-purpose database is used just as an isolated operation in the chain of all required operations to be performed. In such case the database might be established for a single or several queries and then discarded. Hence, index-based methods cannot be used, because, in terms of the overall costs (indexing+querying), the simple sequential search would perform better.

Arbitrary similarity function. Sometimes the similarity measure is not defined a priori and/or can change over the time. This includes learning, user-defined or query-defined similarity. In such case, any indexing would lead to many different indexes, or is not possible at all.

To address the three scenarios, the D-file, as the "founding father" of index-free MAMs, could be the solution. We also emphasize that D-file should not be viewed as an index-based MAM that just maintains its temporary index in main memory. We see the difference between *index-based* and *index-free* methods not only in the main-memory organization, but mainly in the fragmentation of "indexing". While index-based MAMs cannot search the database before the indexing is finished, the index-free MAMs are allowed to search the database instantly. Moreover, as the "indexing step" (updating the D-cache) is performed during the query, the indexing vs. query efficiency trade-off remains balanced at any time.

4.2 Enhancing Pivot Tables

When considering range queries in Pivot tables, like LAESA, we have to discuss two steps (for details see Section 2.2). First, there is filtering by pivots[3] performed, where a query vector

3. Here we consider regular (static) pivots of Pivots tables. Please, do not confuse pivot tables' (static) pivots with D-cache's *dynamic* pivots.

is computed, a query box is established, and all the distance matrix rows are checked if they fall inside the query box. If not, these objects are filtered out from further processing, while the non-filtered objects are processed in the second (refinement) step by usual sequential search.

In the first step (filtering by pivots) the only moment of computing δ is the construction of query vector. Then, the pivot table is checked against the query box which does not require any distance computation. Hence, the D-cache is not needed in the first step. On the other hand, it could be utilized in the second (refinement) step when sequentially searching the non-filtered candidate objects. In fact, we can view the set of non-filtered objects as a (small) sequential file, where the D-cache could be utilized the same way as in the D-file.

Since the distance matrix consists of exact distance values that are not repeating, the D-cache cannot be used for indexing. We implemented the D-cache-enhanced querying into Pivot tables and called the new MAM as *D-Pivot Tables* (or *D-PT*).

4.3 Enhancing M-tree

In M-tree, the cheap filtering step based on D-cache is placed between the *parent filtering* (also cheap) and the *basic filtering* (expensive), see Algorithm 5.

Algorithm 5: (D-M-tree range query)

```
D-MtreeRangeQuery(Node N, RQuery (q, rad_q)) {
  let rout(p) be the parent routing entry of N
                                  // if N is root then let δ(o_i, p)=δ(p, q)=0
  if N is root then Dcache.StartRuntimeProcessing(q)
  if N is not a leaf then {
    for each rout(o_i) in N do
      if |δ(p, q) − δ(o_i, p)| ≤ rad_q + rad_{o_i} then          // parent filtering
                                                                 // D-cache filtering
        if Dcache.GetLowerBoundDistance(q, o_i) ≤ rad_q + rad_{o_i} then
          compute δ(q, o_i); Dcache.InsertDistance(q, o_i, δ(q, o_i))
          if δ(o_i, q) ≤ rad_q + rad_{o_i} then                  // basic filtering
            D-MtreeRangeQuery(ptr(T(o_i)), (q, rad_q))
  } else {
    for each grnd(o_i) in N do
      if |δ(p, q) − δ(o_i, p)| ≤ rad_q then                      // parent filtering
                                                                 // D-cache filtering
        if Dcache.GetLowerBoundDistance(q, o_i) ≤ rad_q then
          compute δ(q, o_i); Dcache.InsertDistance(q, o_i, δ(q, o_i))
          if δ(o_i, q) ≤ rad_q then                              // basic filtering
            add o_i to the query result } }
```

Furthermore, the D-cache can be used also to speed up the construction of M-tree, where we use both the exact retrieval of distances (method GetDistance) and also the lower-bounding functionality. The node splitting in M-tree often uses the expensive `mM_RAD` heuristics, where a distance matrix is computed for all pairs of node entries. The values of this matrix can be stored in D-cache and some of them reused later, when node splitting is performed on the child nodes of the previously split node. Similarly, when using forced reinsertions, the distances related to the inserted objects reside in the D-cache and could be used when reinserting some of the objects in the future. Moreover, when employing the expensive multi-way insertion, the D-cache could be used also in the "non-exact" way (using lower bounds similarly as by querying). We implemented the D-cache-enhanced indexing+querying into M-tree and called the new MAM as *D-M-tree*.

5 Related Work

To the best of our knowledge, there are no other approaches to distance caching for general MAMs. The most similar proposed approaches are in the line of bulk loading for batch insertion or processing multiple queries at once. However, in this case the data/queries need to be available beforehand, i.e., we cannot consider a continuous stream of queries or insertions. While in the literature it has been proposed the use of different kinds of "caching" in the context of multidimensional databases (e.g., caching in distributed systems [24], or a "L2 cache conscious" main-memory multidimensional index [25]), they do not take advantage of the computed distances at query time to speed up (future) similarity queries.

5.1 Caching Distances in M-tree

One idea that actually uses cached distances for range queries with an M-tree was proposed by Kailing et al. [26]. For each query, the distances computed from each routing object to the query are cached. In this way, if there are duplicated routing objects at different levels of the M-tree, their distance to the query object will be computed only once. As the memory cost of saving these distances is proportional to the height of the tree, the extra space needed is "tolerable" [26]. However, the computed distances at each node are deleted from the cache once the recursive search function leaves a node. Therefore, no distances are saved for future queries, and they are only useful if the cached distance between the same two objects needs to be computed again. Moreover, as duplicate routing objects are rare in M-tree (with respect to all examined objects), the savings in distance computations are rather negligible.

5.2 Batch indexing and querying

The basic idea of bulk loading is to create the index from scratch but knowing beforehand the database, thus 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 [27] 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. Recently, a parallel approach to insertion of a batch of objects was proposed for the M-tree [28].

Another approach for improving the efficiency of MAMs is the simultaneous processing of multiple queries [29]. Instead of issuing many single queries, the idea is to process a batch of similarity queries aiming at reducing I/O cost and computation cost. The proposed technique reduces the I/O cost by reading each disk page only once per batch of similarity queries, and it reduces the CPU cost by avoiding distance computations. An avoidable distance computation is detected by computing the distances between query objects and then using these distances, together with the triangle inequality, to compute lower bounds of the distances between queries and database objects. If the lower bound distance is greater

than a given tolerance radius for the similarity search, then the distance calculation is avoidable. The proposed technique is general, and it can be implemented based on a MAM or using a sequential file. However, besides the requirement to know all the queries beforehand, it also requires computing the distances between each pair of query objects to reduce the CPU cost, and it does not take advantage of distance computations between queries and database objects.

In [30] an approach to compute the k nearest neighbor graph in metric spaces was proposed, which is equivalent to compute n kNN queries, in (empirical) subquadratic time. However, the set of query objects was restricted to the database objects, which is only marginally meaningful in our framework.

5.3 Query Result Caching

In order to speed up the similarity searches, a recent approach provides a mechanism of caching query results [31], [32]. Basically, the *metric cache* stores a history of similarity queries and their answers (ids and descriptors of database objects returned by the query). When a next query is to be processed, the metric cache either returns the exact answer in case the same query was processed in the past and its result still sits in the cache. Or, in case of a new query, such old queries are determined from the metric cache, that spatially contain the new query object inside their query balls. If the new query is entirely bounded by a cached query ball, a subset of the cached query result is returned as an exact answer of the new query. If not, the metric cache is used to combine the query results of spatially close cached queries to form an approximate answer. In case the approximate answer is likely to exhibit a large retrieval error, the metric cache gives up and forwards the query processing to the underlying retrieval system/MAM (updating the metric cache by the query answer afterwards). We emphasize that metric cache is a higher-level concept that can be combined with any MAM employed in a search engine. Hence, metric cache is just a standalone front-end part in the whole retrieval system, while the underlying MAM alone is not aware of the metric cache at all. On the other hand, the following proposal of D-cache is a low-level concept that plays the role of integral part of a metric access method (that has to be adjusted to use D-cache functionality). Nevertheless, both approaches could be combined in the future (i.e., the metric cache in front of D-cache-enhanced MAMs).

6 Experimental Evaluation

We have extensively tested the D-cache-enhanced MAMs (D-file, D-M-tree, and D-Pivot tables) and their non-cached counterparts (sequential search, M-tree, Pivot tables) to examine the expected performance gain in indexing and querying. A substantial attention in the experiments was given to the D-file, which in some cases outperformed the index-based MAMs. We have observed both the number of distance computations as well as the real time spent by indexing/querying.

6.1 The Testbed

In order to examine the D-cache in very different conditions, we used four databases (two vector spaces, one string space, and one set space) and four metric distances (three expensive and one cheap), as follows:

(1) A part of the *CoPhIR* [33] database (descriptors of selected images from Flickr.com), namely, one million 282-dimensional vectors (representing five MPEG7 features), and the Euclidean distance as the similarity function (i.e., time complexity $O(n)$).

(2) A database of *Histograms* (descriptors of images downloaded from Flickr.com, but different to CoPhIR), namely, one million 512-dimensional histograms, and the quadratic form distance [34] as the similarity function (i.e., complexity $O(n^2)$). As the image representation we used the standard RGB histogram of dimensionality 512, where the R,G,B components were divided in 8 bins each, thus 8*8*8 = 512 bins. Each histogram was normalized to have the sum equal to 1, while the value of each bin was stored in a float. The similarity matrix used for the quadratic form distance was computed as described in [34], using similarity of colors in the CIE Lab color space [35].

(3) The *Listeria* [36] database, namely 20,000 DNA sequences of Listeria monocytogenes of lengths 200–7,000, and the edit distance [37] as the similarity (i.e., complexity $O(n^2)$).

(4) A synthetic *Clouds* database [38], namely 100,000 clouds (sets) of 60 6-dimensional points (embedded in a unitary 6D cube). For each cloud, its center was generated at random, while the 59 remaining points were generated under normal distribution around the center (the mean and variance in each dimension were adjusted to not generate points outside the unitary cube). Usually, clouds of points are used for simplified representations of complex objects or objects consisting of multiple observations [39]. As an appropriate distance metric, we used the symmetric Hausdorff distance [40] for measuring similarity between sets (maximum distance between a point in one cloud to the nearest point in the other cloud). We used the Euclidean distance as the internal point-to-point distance within the Hausdorff distance (hence, leading to overall complexity $O(n^2)$). The Clouds database was included into the experiments in order to examine a non-vectorial alternative to the usual synthetic database of normally distributed vectors.

In experiments where the growing database size was considered, the particular database subsets were sampled from the respective largest database at random. In the other experiments, we used the entire databases in the case of Clouds and Listeria, and random subsets of size 100,000 in the case of CoPhIR and Histograms. Unless otherwise stated, each query cost was an average over 500 queries using query objects not included in the database.

6.1.1 Database Indexability

As the fundamental assumption on metric access methods is their universal applicability on various kinds of data, the experimental databases were chosen to represent very different metric spaces. In addition to employing cheap (Euclidean) and expensive (quadratic form, edit, Hausdorff) distance functions, the databases also exhibited different *intrinsic dimensionalities*. The intrinsic dimensionality [7] is a concept generalizing the phenomenon of the *curse of dimensionality* into metric

spaces, and is defined as $\rho(\mathbb{S}, \delta) = \frac{\mu^2}{2\sigma^2}$, where μ and σ^2 are the mean and the variance of the distance distribution in the database. Informally, a database where most of the objects are far away from each other exhibits high intrinsic dimensionality and so it is hard to index by any MAM. Conversely, a database where some of the objects are close and some are distant exhibits a low intrinsic dimensionality (i.e., there exist distinct clusters). In this case, the MAMs are able to better separate the data, thus performing similarity queries in an efficient way.

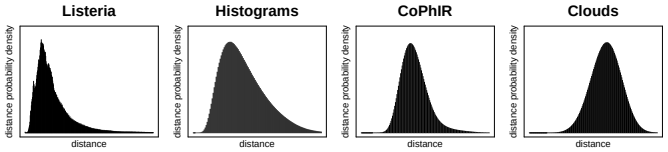

Fig. 4. Distance distribution in the databases.

The Clouds, CoPhIR, and Histograms databases exhibited high intrinsic dimensionalities (11.64, 7.5, 7.56, respectively), and the Listeria database exhibited low intrinsic dimensionality (1.19). Figure 4 shows the distance distribution on each particular database, where a wide and left-shifted "bell" means lower intrinsic dimensionality, and vice versa. Since the intrinsically low-dimensional databases were already efficiently indexed by the MAMs not enhanced by D-cache, there was not as much room to improve the indexing/search by the D-cache as in the case of the high-dimensional databases.

6.1.2 MAM Settings

The M-tree, Pivot Tables, and sequential scan were tested against their D-cache enhanced versions on all the databases. For (D-)M-tree, the node degree was 25 in leaf nodes and 24 in inner nodes, while for its construction the `mM_RAD` node splitting [17] and various object insertion policies were employed [18]. The static pivots of (D-)Pivot Tables were selected from the respective database at random.

6.1.3 D-cache Settings

Unless otherwise stated, the D-cache used 1,280,000 entries (i.e., 19.5 MB of main memory) and 160 dynamic pivots for the CoPhIR, Histograms, and Clouds databases, and it used 64,000 entries (i.e., 1 MB of main memory) and 50 dynamic pivots for the Listeria database. When using the ObsoletePercentile replacement policy, the percentile was set to 36% for Clouds, 4% for Listeria, 15% for Histograms, and 50% for CoPhIR (these values were observed as optimal, as discussed later). The D-cache was reset/initialized before every query batch was started.

Table 1 describes the labels of particular MAM and D-cache configurations used in the following figures.

6.2 Indexing

Table 2 presents the index construction times for MAMs not employing D-cache. As the sequential search and the D-file are index-free methods, they were not included in the indexing experiments.

TABLE 1
Labels used in the figures.

Label	Description
M-tree_SW	M-tree built using single-way insertion [18]
M-tree	the same as M-tree_SW
M-tree_SW_RI	M-tree built using single-way insertion + forced reinsertions [18]
M-tree_MW	M-tree built using multi-way insertion [18]
M-tree_MW_RI	M-tree built using multi-way insertion + forced reinsertions [18]
PT_x	Pivot Tables using x static pivots
D-mam	a particular MAM enhanced by D-cache (see Section 4)
Obs(cfg)	D-cache's Obsolete replacing policy (see Section 3.3.2))
ObsPct(cfg)	D-cache's ObsoletePercentile policy (see Section 3.3.2)
	cfg:
	CI=x: D-cache's collision interval (see Section 3.2), default is CI=5
	H=Simple/Universal: D-cache's hashing function (see Sec. 3.2.1), default is H=Universal
	Dc.size=x: size of D-cache in the number of distance entries
DB(x)	database containing x objects

TABLE 2
Index construction times (D-cache not used).

Database	MAM	Indexing time (seconds)
Clouds of points (100k)	PT_10	248.03
	M-tree	1509.17
Histograms (100k)	PT_10	307.23
	M-tree	1724.84
Listeria (20k)	PT_10	1277.88
	M-tree	13050.98
CoPhIR (1M)	PT_10	88.75
	M-tree	335.89

The index construction times for M-tree and D-M-tree are presented in Figure 5, showing the single-way leaf selection variants (left figure) and multi-way leaf selection variants (right figure). The results show that larger D-cache considerably speeds the M-tree construction (up to 1.7x). Both of the D-M-tree variants use the GetDistance method for indexing. Since the multi-way leaf selection technique issues a point query, it is reasonable to use also the GetLowerBoundDistance in the D-M-tree_MW variant. Also note that the D-M-tree_SW_RI that uses extra forced reinsertions is even faster than simple M-tree_SW.

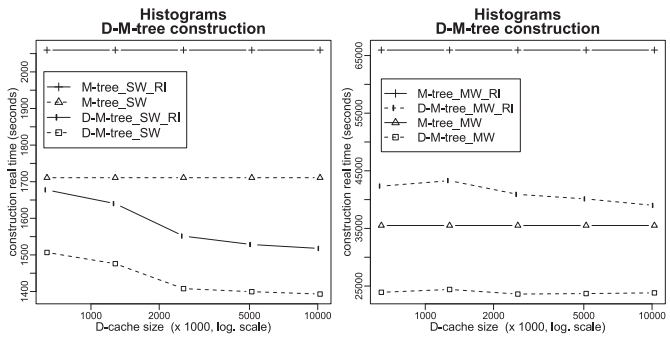

Fig. 5. M-tree and D-M-tree construction, using single-way (left figure) and multi-way (right figure) leaf selection.

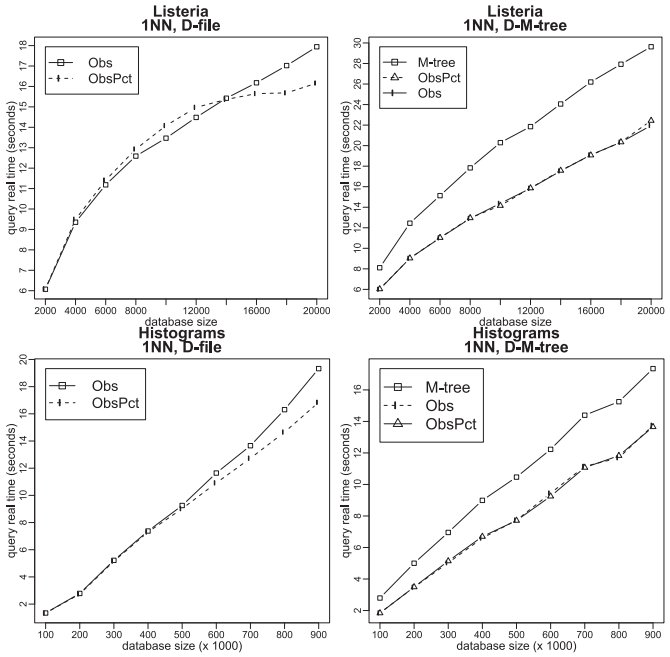

Fig. 6. 1NN queries on growing databases.

TABLE 3
Real times of sequential search.

Database	Query time (seconds)
Histograms (100k)	29.03
Clouds (100k)	21.10
Listeria (20k)	134.53
CoPhIR (1M)	4.60

6.3 Queries

The largest set of experiments was focused on kNN queries under different D-cache and retrieval settings. Unless otherwise stated, on databases that employed expensive distances we present just the real times for queries, because the numbers of distance computations followed exactly the same proportion. In other words, when queried by the expensive distance metrics, the real time spent outside the code computing distances was negligible. Also note that because the query objects were outside the database (i.e., unknown to D-cache), the speed-up achieved by D-cache was solely based on the lower-bounding functionality (see Section 3.3).

Table 3 shows the baseline real times when searching a database sequentially, regardless of the query selectivity. The results confirm that the Euclidean distance (used on CoPhIR) is very efficient, while the edit distance used on the long Listeria sequences is very expensive.

6.3.1 Database Size

The first querying experiment was focused on the growing database size while fixing the size of the D-cache used (see Figure 6). We observe that for small databases there is enough space in D-cache, so that distance replacements are not often needed. However, for larger databases the D-cache gets filled and the distance replacements are necessary. In such case, for D-file the ObsPct replacement policy turns out to be more effective for replacing "bad" distances, which results in a better filtering and so in a faster query processing. On the other hand, for the same D-cache size but the D-M-tree, the filling of D-cache with distances is slower (because of more aggressive filtering), so distance replacements are not often.

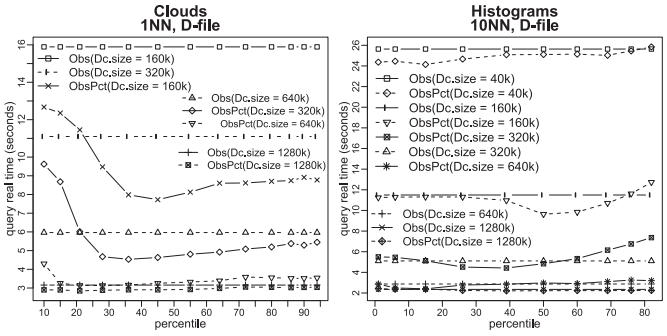

Fig. 7. Impact of various percentile distances used by ObsPct replacement heuristics.

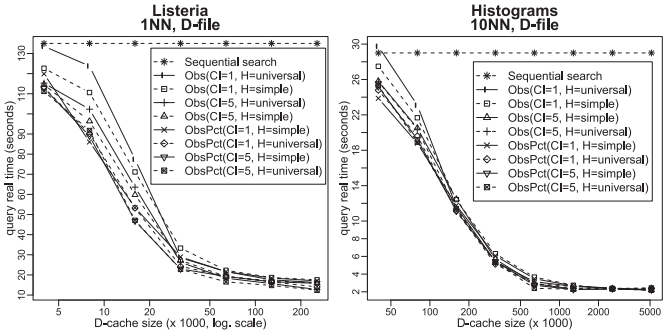

Fig. 8. Impact of D-cache size on distance replacement.

6.3.2 Percentile Distances

In the second experiment we have investigated the percentile distances that optimize the replacement of the least useful distances in the D-cache, see Figure 7. Since the Clouds database exhibits large intrinsic dimensionality, the proportion of possibly "bad" distances that cannot be used for effective lower-bound filtering is larger than in the Histogram database. Hence, for Clouds database the ObsPct replacement policy that prevents from storing the bad distances leads to faster querying than the Obs policy. This effect is even magnified for smaller D-cache sizes (up to 2x query speed-up).

6.3.3 D-cache Size

Next, we performed experiments with growing D-cache size for various replacement policies, collision intervals, and hashing functions (see Figure 8, where the D-cache size is the number of distance entries allocated). Although for smaller D-cache sizes the different settings lead to slightly different querying performance, for larger D-cache sizes the differences are negligible. Nevertheless, the ObsPct(CI=5, H=universal) policy performed well under all circumstances.

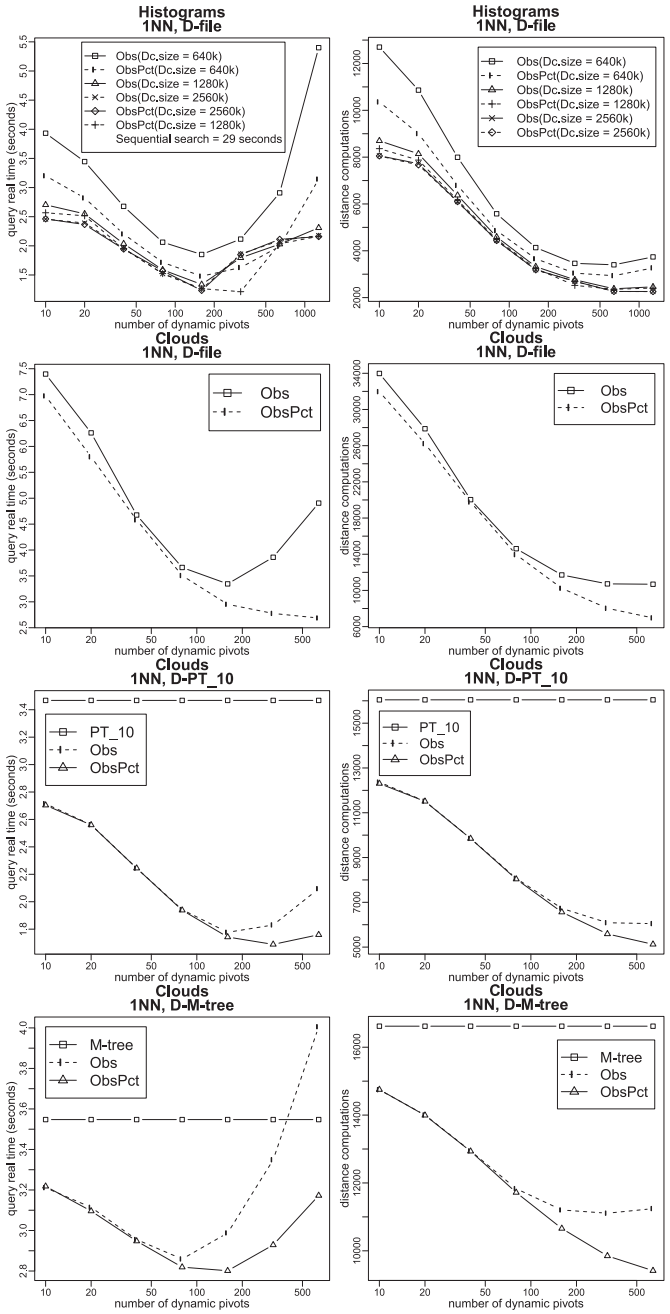

Fig. 9. Impact of the growing number of dynamic pivots.

6.3.4 Number of Dynamic Pivots

Figure 9 shows the impact of increasing number of dynamic pivots used by D-cache. Instead of the usual 500 queries, we present the averaged results over 1,000 queries for Clouds and 2,500 queries for Histograms, in order to justify the larger numbers of dynamic pivots. Also note that in this experiment we present both the real time and the number of distance computations.

The superiority of ObsPct replacement policy is here confirmed. For a large number of dynamic pivots and when replacing an entry in D-cache, the likelihood that the collision interval contains an obsolete entry will be low because most of the past runtime objects are still dynamic pivots. In such a case when no obsolete entry is available, the Obs policy just replaces the first entry found in the collision interval. On the other hand, the ObsPct policy replaces the least "useful" entry in the interval, based on the selected percentile distance.

To mention also the negative results, with a growing number of dynamic pivots the time complexity of the methods GetLowerBoundDistance and InsertDistance increases, resulting in increased real time spend for querying. The overhead of the mentioned methods is even more visible when using cheap metric distances. To show a clear fail of D-cache, we present results for the CoPhIR database searched under the Euclidean distance. Observe in Figure 10 the discrepancy between the real time and the number of distance computations spent for query processing.

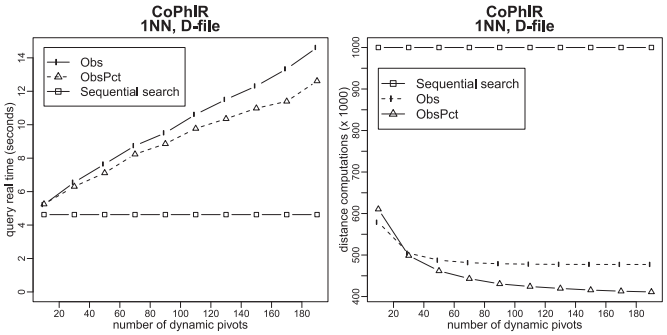

Fig. 10. Negative results on the CoPhIR database.

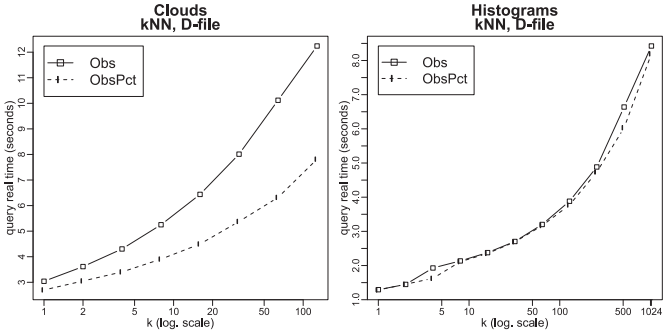

Fig. 11. kNN queries.

6.3.5 kNN queries

The next experiment investigated the performance of kNN queries on D-file, see Figure 11. Since for larger k the number of distance computations spent by querying increases, the D-cache size becomes insufficient. This leads to less effective query processing.

6.3.6 Number of Objects in the Query Batch

In the last test we examined the "warming" of D-cache, that is, how many queries are needed to populate the D-cache to be useful enough for filtering. Figure 12 shows the impact of the growing query batch size, that is, the average cost of a 10NN query when running queries in differently sized batches (each query batch runs as a single D-cache session). The trend is obvious: the more queries, the more distances get into the

D-cache which the subsequent queries can benefit from. The difference is quite significant for the D-file: the average cost of a query within a 2,500 queries batch falls down to 70% of the average query cost within a 700 queries batch. Moreover, in the right graph of Figure 12 see the total query cost (not the average as usual) for differently sized batches of queries, including also the indexing cost. This test aims to show the overall cost when searching a database for a limited number of queries. Obviously, when only a small number of queries is needed, say up to 300, the D-file is the clear winner because of its index-free concept. On the other hand, when reaching a sufficiently large number of queries, the index-based MAMs begin to amortize the huge initial indexing cost by the efficient query processing (but the D-file still keeps up with them). In case of MAMs employing D-cache, the amortization is quicker.

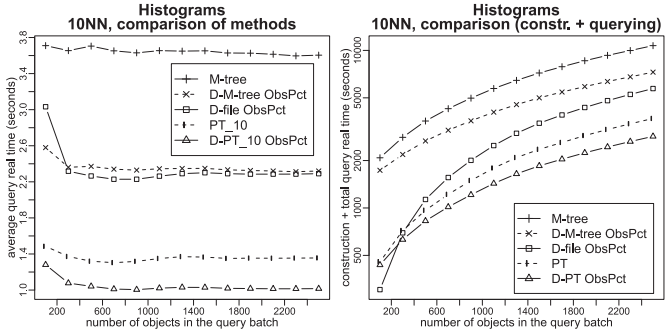

Fig. 12. Impact of the growing volume of query batch.

6.4 Summary

We have shown that the D-cache accelerates the indexing of M-tree significantly. When querying, the D-cached-enhanced MAMs perform up to two times faster than their non-cached counterparts (up to 24 times in case of D-file).

A special attention should be devoted to the D-file, which is not only the first index-free MAM, but it can compete with efficient competitors like the Pivot tables or the M-tree.

The D-cache proved its benefits in most of the experiments. On the other hand, when a cheap metric distance is used, the overhead of D-cache is too large. Regarding the D-cache tuning, we observed that the ObsoletePercentile replacement strategy works the best in most of the cases, while the number of dynamic pivots ranging from tens to a few hundreds is sufficient. When considering the D-file, the size of employed D-cache should be proportional to the database size (e.g., 10%) in order to achieve an optimal performance.

7 Conclusions

In this paper we presented the D-cache, a main-memory data structure which tracks computed distances while inserting objects or performing similarity queries in the metric space model. Since distance computations stored in the D-cache may be re-used in further database operations, it is not necessary to compute them again. Also, the D-cache can be used to estimate distance functions between new objects and objects stored in the database, which can also avoid expensive distance computations. The D-cache aims to amortize the number of distance computations spent by querying/updating the database, similarly like disk page buffering in traditional DBMSs aims to amortize the I/O cost.

The D-cache structure is based on a hash table, thus making efficient to retrieve stored distances for further usage. Additionally, the D-cache maintains the set of previously processed runtime objects (i.e., inserted or query objects), while the most recent of them are used as dynamic pivots. The D-cache supports three functions useful for metric access methods (MAMs) – the GetDistance (returning the exact distance between two objects, if available), the GetLowerBoundDistance (returning the greatest lower-bound distance between two objects, by means of the dynamic pivots), and the GetUpperBoundDistance (returning the lowest upper-bound distance). With these functions, the D-cache may be used to improve the construction of MAMs' index structures and the performance of similarity queries.

Our depiction of the D-cache is general, and may be used with any metric access method or even to aid a sequential scan of the database – forming a brand new concept of index-free MAM, the D-file. We have presented replacement policies for the distances stored in the cache as well as algorithms for the computation of the lower- and upper-bound distances. We have also described in detail how to enhance some of the existing metric access methods (M-tree, Pivot tables) with the D-cache.

Finally, we presented the results of an experimental evaluation with different databases using expensive and cheap metric distance functions. When considering expensive enough distance functions ($\geq O(n^2)$), the D-cache substantially improves the real times needed to query/update metric databases.

Acknowledgments

This research has been supported by Czech Science Foundation grant GAČR 201/09/0683 (first and second author), and by FONDECYT (Chile) Project 11070037 (third author).

References

[1] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, 2001. [Online]. Available: citeseer.ist.psu.edu/vitter01external.html

[2] C. Böhm, S. Berchtold, and D. Keim, "Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases," *ACM Computing Surveys*, vol. 33, no. 3, pp. 322–373, 2001.

[3] S. D. Carson, "A System for Adaptive Disk Rearrangement," *Software - Practice and Experience (SPE)*, vol. 20, no. 3, pp. 225–242, 1990.

[4] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 4, pp. 560–595, 1984.

[5] M. Batko, D. Novak, F. Falchi, and P. Zezula, "Scalability Comparison of Peer-to-Peer Similarity Search Structures," *Future Gener. Comput. Syst.*, vol. 24, no. 8, pp. 834–848, 2008.

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

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in Metric Spaces," *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, 2001.

[8] G. R. Hjaltason and H. Samet, "Index-driven Similarity Search in Metric Spaces," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 517–580, 2003.

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

[10] T. Skopal and B. Bustos, "On Index-Free Similarity Search in Metric Spaces," in *DEXA, LNCS 5690*. Springer, 2009, pp. 516–531.

[11] E. Vidal, "New Formulation and Improvements of the Nearest-neighbour Approximating and Eliminating Search Algorithm (AESA)," *Pattern Recognition Letters*, vol. 15, no. 1, pp. 1–7, 1994.

[12] M. L. Micó, J. Oncina, and E. Vidal, "An Algorithm for Finding Nearest Neighbour in Constant Average Time with a Linear Space Complexity," in *Int. Cnf. on Pattern Recog.*, 1992.

[13] M. L. Micó, J. Oncina, and R. C. Carrasco, "A Fast Branch & Bound Nearest-neighbour Classifier in Metric Spaces," *Pattern Recogn. Lett.*, vol. 17, no. 7, pp. 731–739, 1996.

[14] E. Chávez, J. L. Marroquín, and R. Baeza-Yates, "Spaghettis: An Array Based Algorithm for Similarity Queries in Metric Spaces," in *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*. IEEE Computer Society, 1999, p. 38.

[15] C. Traina, Jr., R. F. Filho, A. J. Traina, M. R. Vieira, and C. Faloutsos, "The Omni-family of All-purpose Access Methods: a Simple and Effective Way to Make Similarity Search More Efficient," *The VLDB Journal*, vol. 16, no. 4, pp. 483–505, 2007.

[16] T. Skopal, "Pivoting M-tree: A Metric Access Method for Efficient Similarity Search," in *Proceedings of the 4th annual workshop DATESO, Desná, Czech Republic, ISBN 80-248-0457-3, also available at CEUR, Volume 98, ISSN 1613-0073,* `http://www.ceur-ws.org/Vol-98`, 2004, pp. 21–31.

[17] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," in *VLDB'97*, 1997, pp. 426–435.

[18] T. Skopal and J. Lokoč, "New Dynamic Construction Techniques for M-tree," *Journal of Discrete Algorithms*, vol. 7, no. 1, pp. 62–77, 2009.

[19] B. Bustos, G. Navarro, and E. Chávez, "Pivot Selection Techniques for Proximity Searching in Metric Spaces," *Pattern Recognition Letters*, vol. 24, no. 14, pp. 2357–2366, 2003.

[20] J. Venkateswaran, T. Kahveci, C. Jermaine, and D. Lachwani, "Reference-based Indexing for Metric Spaces with Costly Distance Measures," *The VLDB Journal*, vol. 17, no. 5, pp. 1231–1251, 2008.

[21] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143 – 154, 1979.

[22] M. Patella and P. Ciaccia, "The Many Facets of Approximate Similarity Search," in *Proc. 1st International Workshop on Similarity Search and Applications (SISAP'08)*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 10–21.

[23] B. Bustos and G. Navarro, "Probabilistic Proximity Search Algorithms Based on Compact Partitions," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 115–134, 2004.

[24] B. Nam, H. Andrade, and A. Sussman, "Multiple Range Query Optimization with Distributed Cache Indexing," in *Proc. ACM/IEEE Conference on High Performance Networking and Computing (SC'06)*. IEEE, 2006, p. 100.

[25] J. M. Shim, S. I. Song, Y. S. Min, and J. S. Yoo, "An Efficient Cache Conscious Multi-dimensional Index Structure," in *Proc. International Conference on Computational Science and Its Applications (ICCSA'04) (4)*, 2004, pp. 869–876.

[26] K. Kailing, H.-P. Kriegel, M. Pfeifle, and S. Schnauer, "Extending Metric Index Structures for Efficient Range Query Processing," *Knowledge and Information Systems*, vol. 10, no. 2, pp. 211–227, 2006.

[27] 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)*. Morgan Kaufmann, 2001, pp. 461–470.

[28] J. Lokoč, "Parallel Dynamic Batch Loading in the M-tree," in *Proc. 2nd International Workshop on Similarity Search and Applications (SISAP 2009)*. IEEE CS, 2009, pp. 117–123.

[29] B. Braunmüller, M. Ester, H.-P. Kriegel, and J. Sander, "Multiple Similarity Queries: A Basic DBMS Operation for Mining in Metric Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 79–95, 2001.

[30] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, "Practical Construction of k-Nearest Neighbor Graphs in Metric Spaces," in *Proc. 5th International Workshop on Experimental Algorithms (WEA'06)*, ser. LNCS 4007. Springer, 2006, pp. 85–97.

[31] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti, "A Metric Cache for Similarity Search," in *LSDS-IR '08: Proceeding of the 2008 ACM workshop on Large-Scale distributed systems for information retrieval*. New York, NY, USA: ACM, 2008, pp. 43–50.

[32] ——, "Caching Content-based Queries for Robust and Efficient Image Retrieval," in *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*. New York, NY, USA: ACM, 2009, pp. 780–790.

[33] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti, "CoPhIR: a Test Collection for Content-based Image Retrieval," *CoRR*, vol. abs/0905.4627v2, 2009. [Online]. Available: http://cophir.isti.cnr.it

[34] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack, "Efficient Color Histogram Indexing for Quadratic Form Distance Functions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 729–736, 1995.

[35] Y. Rubner, J. Puzicha, C. Tomasi, and J. M. Buhmann, "Empirical Evaluation of Dissimilarity Measures for Color and Texture," *Comput. Vis. Image Underst.*, vol. 84, no. 1, pp. 25–43, 2001.

[36] K. Figueroa, G. Navarro, and E. Chavez, "Metric Spaces Library, `http://www.sisap.org/library/metricSpaces/docs/manual.pdf`," 2007.

[37] I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics-Doklady 10*, pp. 707–710, 1966.

[38] J. Lokoč, "Cloud of Points Generator, SIRET Research Group, `http://siret.ms.mff.cuni.cz/projects/pointgenerator/`," 2010.

[39] F. Mémoli and G. Sapiro, "Comparing Point Clouds," in *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. New York, NY, USA: ACM, 2004, pp. 32–40.

[40] D. Huttenlocher, G. Klanderman, and W. Rucklidge, "Comparing Images Using the Hausdorff Distance," *IEEE Patt. Anal. and Mach. Intell.*, vol. 15, no. 9, pp. 850–863, 1993.

Tomáš Skopal is an Associate Professor at the Department of Software Engineering at the Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic. He is the head of the SIRET Research Group. His research interests are metric access methods, database indexing, multimedia databases and similarity modeling. He has a doctoral degree in computer science and applied mathematics from the Technical University of Ostrava, Czech Republic.

Jakub Lokoč is a Researcher at the Department of Software Engineering at the Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic. His research interests are metric access methods, parallel database indexing, multimedia databases and similarity modeling. He has a doctoral degree in software systems from the Charles University in Prague, Czech Republic.

Benjamin Bustos is an Assistant Professor at the Department of Computer Science, University of Chile. He is head of the PRISMA Research Group. He leads research projects in the domains of multimedia retrieval, video copy detection, sketch-based image retrieval, and retrieval of handwritten documents. His interests include similarity search, multimedia retrieval, 3D object retrieval, and (non)-metric indexing. He has a doctoral degree in natural sciences from the University of Konstanz, Germany.