On Index-free Similarity Search in Metric Spaces

Tomáš Skopal[1] and Benjamin Bustos[2]

[1] Department of Software Engineering, FMP, Charles University in Prague,
Malostranské nám. 25, 118 00 Prague, Czech Republic
`skopal@ksi.mff.cuni.cz`
[2] Department of Computer Science, University of Chile,
Av. Blanco Encalada 2120 3er Piso, Santiago, Chile
`bebustos@dcc.uchile.cl`

Abstract. Metric access methods (MAMs) serve as a tool for speeding similarity queries. However, all MAMs developed so far are index-based; they need to build an index on a given database. The indexing itself is either static (the whole database is indexed at once) or dynamic (insertions/deletions are supported), but there is always a preprocessing step needed. In this paper, we propose *D-file*, the first MAM that requires no indexing at all. This feature is especially beneficial in domains like data mining, streaming databases, etc., where the production of data is much more intensive than querying. Thus, in such environments the indexing is the bottleneck of the entire production/querying scheme. The idea of D-file is an extension of the trivial sequential file (an abstraction over the original database, actually) by so-called *D-cache*. The D-cache is a main-memory structure that keeps track of distance computations spent by processing all similarity queries so far (within a runtime session). Based on the distances stored in D-cache, the D-file can cheaply determine lower bounds of some distances while the distances alone have not to be explicitly computed, which results in faster queries. Our experimental evaluation shows that query efficiency of D-file is comparable to the index-based state-of-the-art MAMs, however, for zero indexing costs.

1 Introduction

The majority of problems in the area of database systems concern the efficiency issues – the performance of DBMS. For decades, the number of accesses to disk required by I/O operations was the only important factor affecting the DBMS performance. Hence, there were developed indexing structures [16, 2], storage layouts [4], and also disk caching/buffering techniques [7]; all of these designs aimed to minimize the number of physical I/Os spent during a database transaction flow. In particular, disk caching is extremely effective in situations where repeated access to some disk pages happens within a single runtime session.

In some modern databases, like multimedia DBs (MMDBs), DNA DBs, time series DBs, etc., we need to use a similarity function $\delta(\cdot, \cdot)$ which serves as a relevance measure, saying how much a DB object is relevant to a query object. To speedup similarity search in MMDBs, there have been many indexing techniques developed, some being domain-specific and some others more general. The new important fact is that the performance of MMDBs is more affected by CPU costs than by I/O costs. In particular, in MMDBs community a single computation of similarity value $\delta(\cdot, \cdot)$ is employed as the logical unit for indexing/retrieval cost, because of its dominant impact on overall MMDB performance [18, 5] (algorithms computing $\delta(\cdot, \cdot)$ are often super-linear in terms of DB object size). Thus, the I/O costs are mostly regarded as a minor component of the overall cost because of the computational complexity of similarity measures. The number of computations $\delta(\cdot, \cdot)$ needed to answer a query or to index a database is referred to as the *computation costs*.

1.1 Metric Access Methods

Among the similarity search techniques, *metric access methods* (MAMs) are suitable in situations where the similarity measure δ is a *metric* distance (in mathematical meaning). The metric postulates – reflexiveness, positiveness, symmetry, triangle inequality – allow us to organize the database within classes that represent some occupied partitions of the underlying metric space. The classes are usually organized in a data structure (either persistent or main-memory), called *index*, that is created during a preprocessing step (the indexing).

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 horse", or a *range query* like "return all voices more than 80% similar to the voice of nightingale". In particular, when issued a similarity query[1], the MAMs exclude many non-relevant classes from the search (based on metric properties of δ), so only several candidate classes of objects have to be sequentially searched. In consequence, searching a small number of candidate classes turns out in reduced computation costs of the query.

There were developed many MAMs so far, addressing various aspects – main-memory/database-friendly methods, static/dynamic indexing, exact/approximate search, centralized/distributed indexing, etc. (see [18, 12, 5, 11]). Although various MAMs often differ considerably, they all share the two following properties:

1. MAMs are all *index-based*. For a given database, an index must exist in order to be able to process queries. Hence, the first query must be always preceded by a more or less expensive data preprocessing which results in an index (either main-memory or persistent).
2. Once its index is built, a MAM solves every query request *separately*, that is, every query is evaluated as it would be the only query to be ever answered. In general, no optimization for a *stream of queries* is considered by MAMs up to date. Instead, enormous research has been spent in "materializing" the data-pruning/-structuring knowledge into the index file itself.

In the following, we consider three representatives out of dozens of existing MAMs – the *M-tree*, the *PM-tree*, and *GNAT*.

[1] A range or kNN query can be viewed as a ball in the metric space (centered in query object Q with radius of the range/distance to kNN), so we also talk about *query ball*.

M-tree. The *M-tree* [6] is a dynamic (easily updatable) index structure that provides good performance in secondary memory, i.e., in database environments. The M-tree index is a hierarchical structure, where some of the data objects are selected as centers (also called references or 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.

PM-tree. The idea of PM-tree [13, 14] is to enhance the hierarchy of M-tree by using information related to a static set of p global pivots P_i. In a PM-tree's non-leaf region, the original M-tree-inherited ball region is further cut off by a set of rings (centered in the global pivots), so the region volume becomes smaller. Similarly, the PM-tree leaf entries are extended by distances to the pivots, which are also interpreted as rings due to quantization. Each ring stored in a non-leaf/leaf entry represents a distance range bounding the underlying data with respect to a particular pivot. The combination of all the p entry's ranges produces a p-dimensional minimum bounding rectangle, hence, the global pivots actually map the metric regions/data into a "pivot space" of dimensionality p.

GNAT. The Geometric Near-Neighbor Access Tree (GNAT) [3] is a metric access method that extends the Generalized-Hyperplane Tree [15]. The main idea behind GNAT is to partition the space into zones that contain close objects. The root node of the tree contains m objects selected from the space, the so-called *split-points*. The rest of the objects is assigned to their closest split-point. The construction algorithm selects with a greedy algorithm the split-points, such that they are far away from each other. Each zone defined by the selected split-points is partitioned recursively in the same way (possibly using a different value for m), thus forming a search hierarchy. At each node of the tree, a $O(m^2)$ table stores the range (minimum and maximum distance) from each split-point to each zone defined by the other split-points.

1.2 Motivation for Index-free Similarity Search

As mentioned earlier, the existing MAMs are all index-based. However, there emerge many real and potential needs for access methods that should provide index-free similarity search. We briefly discuss three cases where any data pre-processing (like indexing) is undesirable:

"Changeable" databases. In many applications, there are databases which content is 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 not only slow but even impossible.

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, while in such case any indexing would lead to many different indexes, or it is not possible at all.

1.3 Paper Contribution

In this paper, we propose the D-file, an index-free MAM employing a main-memory structure – the D-cache. The D-cache (distance cache) stores distances computed during querying within a single runtime session. Hence, the aim of D-file is not to use an index, but to amortize the query costs by use of D-cache, similarly like I/O-oriented access methods amortize the I/O costs using disk cache. As in the case of simple sequential search, querying the D-file also means a sequential traversal of the entire database. However, whenever a DB object is to be checked against a query, instead of computing the DB object-to-query object distance, we request the D-cache for its tightest lower bound. This lower-bound distance is subsequently used to filter the DB object. Since many distances could have been computed during previous querying (for other query objects, of course), the lower bounds could be transitively inferred from the D-cache "for free", which results in reduced query costs.

2 Related Work

In this section, we briefly discuss existing index-free attempts to metric similarity search. In fact, to the best of our knowledge, there exist just one non-trivial approach applicable directly to the index-free metric search, as mentioned in Section 2.2. But first, in the following section we discuss the simple sequential scan – in a role of trivial index-free MAM.

2.1 Simple Sequential Scan

If no index structure is provided, the only way of answering a similarity query in metric spaces is to perform a sequential search of the database. By this approach, for both range and k-NN queries the search algorithm computes the distances between the query object and all objects in the database. With the computed distances it is trivial to answer both types of queries. This approach has, of course, the advantage that neither space nor preprocessing CPU time is required to start performing similarity queries. It follows that the cost of a sequential scan is linear in the size of the database. This, however, may be already prohibitively expensive, for example if the database is too large, e.g., it contains tens of millions of objects, or if the distance function is expensive to compute, e.g., the edit distance between strings is $O(nm)$ for sequences of lengths n and m.

For the particular case of vector spaces, the VA-file [17] is a structure that stores compressed feature vectors, providing thus an efficient sequential scan of the database. At query time, an approximation of the distances between query objects and compressed features are computed, discarding at this filtering step as many objects as possible. The search algorithm refines this result by computing the real distances to database objects only for the non-discarded vectors. While this approach could be a good alternative to the plain sequential scan, it only works in vector spaces and cannot be easily generalized to the metric case. Moreover, though the idea is based on sequential search, it is not index-free approach, because the compressed vectors form a persistent index – the VA-file.

2.2 Query Result Caching

Recently, the concept of *metric cache* for similarity search was introduced, providing a caching mechanism that prevents any underlying MAM (i.e., also simple sequential scan) to process as many queries as possible [8, 9]. 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 already 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 of the new query. In case the approximated 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 have to emphasize that metric cache is a higher-level concept that can be combined with any MAM employed in a content-based retrieval system. Hence, metric cache is just a standalone front-end subpart in the whole retrieval system, while the underlying MAM alone is not aware of the metric cache. On the other hand, the proposal of D-cache in the following text is a low-level concept that plays the role of integral part of a metric access method (the D-file, actually).

3 D-file

We propose an index-free metric access method, the *D-file*, which is a set of methods extending simple sequential search over the database. Unlike the VA-file mentioned earlier, we emphasize that D-file is just an abstraction above the original database, hence, there is no additional “file” materialized alongside the database, that is, no additional *persistent* data structure is maintained, nor any preprocessing is performed. In other words, the D-file is the original database file equipped by a set of querying methods. Instead, the D-file uses a main-memory structure called D-cache (described in the next section). The D-cache has a simple objective – to gather (cache) distances already computed between DB and query objects within a single runtime session. Based on the stored distances, the D-cache can be asked to cheaply infer lower bound of some distance between a query object and a DB object. The D-file's query algorithms then use these lower bounds when filtering DB objects, see Algorithms 1 and 2.

Algorithm 1 (D-file kNN query)

```
set kNNQuery($Q, k$) {
  Dcache.StartQueryProcessing($Q$)
  let NN be array of $k$ pairs $[O_i,\ \delta(Q, O_i)]$ sorted asc. wrt $\delta(Q, O_i)$, initialized to NN = $[[-, \infty], ..., [-, \infty]]$
  let $r_Q$ denotes the actual distance component in NN$[k]$
  for each $O_i$ in database do
    if Dcache.GetLowerBoundDistance($Q$, $O_i$) $\leq r_Q$ then                    // D-cache filtering
      compute $\delta(Q, O_i)$; Dcache.AddDistance($Q$, $O_i$, $\delta(Q, O_i)$)
      if $\delta(Q, O_i) \leq r_Q$ then insert $[O_i, \delta(Q, O_i)]$ into NN           // basic filtering
  return NN as result }
```

Algorithm 2 (D-file range query)

```
set RangeQuery($Q, r_Q$) {
  Dcache.StartQueryProcessing($Q$)
  for each $O_i$ in database do
    if Dcache.GetLowerBoundDistance($Q$, $O_i$) $\leq r_Q$ then                    // D-cache filtering
      compute $\delta(Q, O_i)$; Dcache.AddDistance($Q$, $O_i$, $\delta(Q, O_i)$)
      if $\delta(Q, O_i) \leq r_Q$ then add $O_i$ to the query result }              // basic filtering
```

4 D-cache

The main component of D-file is the *D-cache* (distance cache) – a non-persistent (memory resident) structure, which tracks distances computed between query objects and DB objects, considering a single runtime session, i.e., contiguous sequence of queries. The track of distance computations is stored as a set of triplets, each of the form $[id(Q_i), id(O_j), \delta(R_i, O_j)]$, where Q_i is a query object, O_j is a DB object, and $\delta(Q_i, O_j)$ is their distance computed during the current session. We assume query as well as DB objects are uniquely identified.

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

$$D = \begin{array}{c} \\ Q_1 \\ Q_2 \\ Q_3 \\ \dots \\ Q_m \end{array} \begin{array}{c} \begin{array}{ccccc} O_1 & O_2 & O_3 & \dots & O_n \end{array} \\ \left(\begin{array}{ccccc} & d_{12} & d_{13} & \dots & \\ d_{21} & & & \dots & d_{2n} \\ & & & \dots & \\ \dots & \dots & \dots & \dots & \dots \\ d_{m1} & & d_{m3} & \dots & \end{array} \right) \end{array}$$

where the columns refer to DB objects, the rows refer to query objects, and the cells store the respective query-to-DB object distances. Naturally, as new DB objects and query objects arrive into D-cache during the session, the matrix gets larger in number of rows and/or columns. Note that at the beginning of session the matrix is empty, while during the session the matrix is being filled. However, at any time of the session there can still exist entirely empty rows/columns.

Note that query objects do not need to be external, that is, a query object could originate from the database. From this point of view, an object can have (at different moments) the role of query as well as the role of DB object, however, the unique identification of objects ensures the D-cache content is correct.

Because of frequent insertions into D-cache, the matrix should be efficiently updatable. Moreover, due to operations described in the next subsection, we have to be able to quickly retrieve a cell, column, or row values. To achieve this goal, we have to implement the matrix by a suitable data structure(s).

4.1 D-cache Functionality

The desired functionality of D-cache is twofold:

– First, given a query object Q and a DB object O on input (or even two DB objects or two query objects), the D-cache should quickly determine the *exact* value $\delta(Q, O)$, provided the distance is present in D-cache.
– The second functionality is more general. Given a query object Q and a DB object O on input, the D-cache should determine the tightest possible *lower bound* of $\delta(Q, O)$ without the need of an explicit distance computation.

Both of the functionalities allow us to filter some non-relevant DB objects from further processing, making the search more efficient. However, in order to facilitate the above functionality, we have to feed the D-cache with information about the involved objects. To exploit the first functionality, the current query object could have to be involved in earlier queries either as query object or as DB object the distance of which was computed against another query object.

To exploit the second functionality, we need to know distances to some past query objects DP_i^Q which are very close to the current query Q. Suppose for a while we know $\delta(Q, DP_1^Q), \delta(Q, DP_2^Q), \ldots$ – these will serve as *dynamic pivots* made-to-measure to Q. Since the dynamic pivots are close to Q, they should be very effective when pruning as they provide tight approximations of $\delta(Q, O_i)$. Having the dynamic pivots DP_i^Q, we can reuse some distances $\delta(DP_i^Q, O)$ still sitting in the D-cache matrix, where they were inserted earlier during the current session. Then, with respect to the pivots and available distances $\delta(DP_i^Q, O)$ in the matrix, the value $max_{DP_i^Q}\{\delta(DP_i^Q, O) - \delta(DP_i^Q, Q)\}$ is the tightest lower-bound distance of $\delta(Q, O)$. Similarly, $min_{DP_i^Q}\{\delta(DP_i^Q, O) + \delta(DP_i^Q, Q)\}$ is the tightest upper-bound distance of $\delta(Q, O)$. See the situation in Figure 1a.

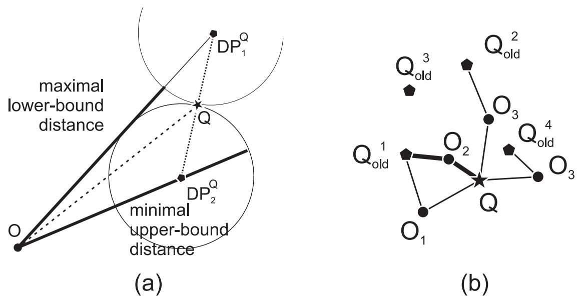

Fig. 1. (a) Lower/upper bounds to $\delta(Q, O)$. (b) Internal selection of dynamic pivot ($k = 1$) by closeness approximation (Q_{old}^1 is the winner).

4.2 Determining Dynamic Pivots

In principle, we consider two ways to obtain k dynamic pivots out of (all) previously processed queries:

(a) Recent. We choose k past query objects immediately, that is, before the current query processing actually starts. More specifically, we choose the k recently processed (distinct) query objects.

(b) Internal. When the current query processing is started, the first x distance computations of $\delta(Q, O_i)$ are used just to update D-cache, that is, the D-cache is still not used for filtering. After insertion of the respective x triplets into D-cache, the D-cache could be requested to select the k most promising dynamic pivots out of the past query objects Q_{old}^i.

The reason for computing x triplets is based on an expectation, that there could appear so-called *mediator objects* in D-cache, that is, objects O_m for which distances $\delta(Q, O_m)$ and $\delta(O_m, Q_j)$ will appear in D-cache. Such mediator objects provide an indirect distance approximation of $\delta(Q, Q_j)$ (remember that because of the triangle inequality $\delta(Q, Q_j) \leq \delta(Q, O_m) + \delta(Q_m, R_j)$). The mediators can be used for selection of dynamic pivots as follows:

A dynamic pivot DP_j is chosen as $DP_j \in \{Q_{old}^i | \exists O_m(\delta(Q, O_m) + \delta(O_m, Q_{old}^i) \leq k\text{-}min_{O_l}\{\delta(Q, O_l) + \delta(O_l, Q_{old}^i))\}\}$. The $k\text{-}min_{O_l}\{\cdot\}$ is the kth minimum value; with respect to any O_l for which D-cache stores distances $\delta(Q, O_l)$ and $\delta(O_l, Q_{old}^i)$. This means that an old query object will be selected as a dynamic pivot if its associated smallest "through-mediator approximation" of $\delta(Q, DP_j)$ is within the k smallest among all old query objects.

Figure 1b shows an example for $k = 1$. The really closest pivot Q_{old}^4 was not selected because the mediator O_3 is an outlier, while no better mediator for Q_{old}^4 was found in D-cache. Also note that for Q_{old}^3 there is no mediator at all.

After we determine the dynamic pivots, we compute their distances to Q. Note that this is the *only place* where we explicitly compute some extra distance computations (not computed when not employing D-cache).

4.3 D-cache Implementation

The D-cache is initialized by D-file when the session begins. Besides this global initialization, the D-cache is also notified by D-file that a query has been started (method StartQueryProcessing). At that moment, a new query object is being processed so the current dynamic pivots have to be dismissed. Every time a distance $\delta(Q, O_i)$ value is explicitly computed, the triplet $[id(Q), id(O_i), \delta(Q, O_i)]$ is inserted into the D-cache (method AddDistance).

Besides the retrieval of the hopefully available exact distance between objects Q, O_i (method GetDistance(Q, O_i)), the main functionality is operated by method GetLowerBoundDistance, see Algorithm 3.

Algorithm 3 (GetLowerBoundDistance)

```
double GetLowerBoundDistance($Q$, $O_i$) {
  let $x$ be the number of computations ignored
  let $k$ be the number of pivots to use
  let $DP$ be the set of dynamic pivots and their distances to $Q$
  mComputed = mComputed + 1                                   // mComputed=0 initialized at query start
  value = 0
  if mComputed $\leq x$ and determineMethod = internal then {            // internal pivot selection
    value = compute $\delta(Q, O_i)$
    AddDistance($Q$, $O_i$, value)
    if mComputed = $x$ then
      $DP$ = DeterminePivotsByMediatorDistances($Q$, $k$)
  } else {                                                    // lower bound construction/update
    for each $P$ in $DP$ do
      if $cell(P, O_i)$ is not empty then
        value = $max$(value, $cell(P, O_i) - \delta(Q, P)$) }
  return value }
```

The structure of D-cache itself is implemented by two substructures – the CellCache and the RowCache:

CellCache Structure. As the main D-cache component, the *CellCache* stores the distance matrix as a set of triplets $(id1, id2, \delta(Q_{id1}, O_{id2}))$ in a hash table, and provides retrieval of individual triplets. As a hash key, $(min(id1, id2)$, $max(id1, id2))$ is used. When applying the *recent* dynamic pivot selection, as defined in Section 4.2, the CellCache is the only D-cache component. Naturally, the hash-based implementation of CellCache is very fast (constant access time).

RowCache Structure. However, when applying *internal* selection of dynamic pivots, we need to retrieve rows from the distance matrix. This cannot be efficiently accomplished by CellCache, hence, we use the *RowCache* as a redundant data structure. In particular, the RowCache aims at efficiently computing the dynamic pivots to be used with the current query object. Thus, it must determine the mediator objects and compute the intersection between rows of the D-cache. It could also be used to obtain bounds of the distances between objects as with the CellCache, however the CellCache may be more efficient than the RowCache for this particular function.

The RowCache is implemented as a main-memory inverted file. Each row of this cache stores all the computed distances for a single query, and it is implemented as a linked list. Each time a distance between the query and an object from the database is computed, a node is added to the list. When a new query object is inserted in the cache, the algorithm creates a new row and stores there the computed distances to the new query object.

To compute the best dynamic pivots for a given query object, the RowCache determines firstly its mediator objects. That is, given two query objects, the current one and one from the cache, it returns the intersection of the corresponding rows in the RowCache. This is repeated for all query objects in the cache. Once the mediator objects are found, the algorithm determines the best dynamic pivots by selecting the k query objects with smallest indirect distance (i.e., obtained through a mediator) to the current query (see Algorithm 4). This algorithm can be efficiently implemented with a priority queue (max-heap), that keeps the k best pivots found so far, and replaces the worst of them when a better pivot is found. With our actual implementation of RowCache[2], in the worst case this algorithm takes $O(A * \log(A) + A * C + A * \log(k))$ time, where A is the number of rows in the RowCache, C is the maximum number of cells per row, and k is the number of pivots. In practice, however, there are usually only a few valid mediators per query objects, thus the average cost is closer to $O(\log(k))$.

Algorithm 4 (DeterminePivotsByMediatorDistances)

```
set DeterminePivotsByMediatorDistances($Q, k$) {
  old = the set of past query objects $Q^i_{old}$
  winners = $\emptyset$ // set of $k$ pairs [object, distance] ordered ASC on distance
  for each $Q^i_{old}$ in old do
    // determine mediators, i.e. objects having distance to both $Q$ and $Q^i_{old}$ in D-cache
    // $row(\cdot) \cap row(\cdot)$ stands for all DB objects having defined both values on their position in the rows
    mediators = $row(Q^i_{old}) \cap row(Q)$
    for each $M$ in mediators do
      update winners by $[Q^i_{old}, cell(Q, M) + cell(Q^i_{old}, M)]$
  return [winners, computed distances $\delta(Q,$ winners$(i))$] }
```

The CellCache and RowCache are not necessarily synchronized, that is, both caches may not contain the distances for the same pair of objects. However, if a query is deleted from the CellCache this is also reflected in the RowCache.

Distance Replacement Policies. Since the storage capacity of D-cache is limited, the hash table of CellCache as well as the inverted list of RowCache are of user-defined size (in bytes, usually equally divided between CellCache and RowCache[3]). Once either of the structures has no space to accommodate a new distance, some old distance (near to the intended location) has to be released.

[2] The depicted implementation of the RowCache structure is not optimized. However, in the experimental evaluation we will show that the effectiveness of the "internal pivot determination" is worse than the simple "recent query objects", anyways.

[3] If the internal dynamic pivot selection (and RowCache) is not used, the entire D-cache capacity is given to CellCache.

We consider two policies for distance replacement:

LRU. A distance is released, which has been least recently used (read). The hypothesis is that keeping just the frequently used distances leads to tighter distance lower/upper bounds, thus to better pruning.

Smallest distance. The hypothesis is an anticipation that small distances between queries and DB objects represent overlapped query and data regions; in such case (even the exact) small distance is useless for pruning, so we release it.

5 Experimental Evaluation

We have extensively tested the D-file, while we have compared its performance with M-tree, PM-tree, and GNAT. We have observed just the computation costs, that is, the number of distance computations spent by querying. For the index-based MAMs, we have also recorded the construction costs in order to give an idea about the indexing/querying trade-off.

5.1 The Testbed

We used 3 databases and 3 metrics (two continuous and one discrete):
– A subset of *Corel features* [10], namely 65,615 32-dimensional vectors of color moments, and the L_1 distance (the sum of the difference of coordinate values between two vectors). Note: L_2 is usually used with color histograms, but from the indexing point of view any L_p norm ($p \geq 1$) gives similar results.
– A synthetic *Polygons* set; 500,000 randomly generated 2D polygons varying in the number of vertices from 10 to 15, and the Hausdorff distance (maximum distance of a point set to the nearest point in the other set). This set was generated as follows: 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 the maximum distance in the space. Since we have used the Hausdorff distance, one could view a polygon as a cloud of 2D points.
– A subset of *GenBank* file `rel147` [1], namely 50,000 protein sequences of lengths from 50 to 100, and the edit distance (minimum number of insertions, deletions, and replacements needed to convert a string into another).

Index-based MAM settings. The databases were indexed with M-tree and PM-tree, while GNAT was used to index just Corel and Polygons. For (P)M-tree, the node size was set to 2kB for Corel, and to 4kB for Polygons and GenBank databases (the node degree was 20–35). The PM-tree used 16 (static) pivots in inner nodes and 8 pivots in leaf nodes. Both M-tree and PM-tree used `mM_RAD` node splitting policy and the single-way object insertion. The GNAT arity (node degree) was set to 50 for Corel and to 20 for Polygons. For most querying experiments, we have issued 1,000 queries and averaged the results.

D-file settings. Unless otherwise stated, the D-cache used 100 MB of main memory and unlimited history of query objects' ids, i.e., we keep track of all the queries issued so far (within a session). The `recent` and `internal` dynamic pivot selection techniques were considered. Concerning `internal` selection, the number of initial fixed distance computations was set to $x = 1,000$. The smallest distance replacement policy was used in all tests. Furthermore, unless otherwise stated, the D-file used 100 dynamic pivots. The D-cache was reset/initialized before every query batch was started.

5.2 Indexing

The first experiment was focused on indexing – the results are summarized in Table 1. Because of its static nature, note that GNAT is an order of magnitude more expensive than (P)M-tree. By the way, due to the expensive construction and the expensive edit distance, we could not index GenBank by GNAT.

index	`Corel`	`Polygons`	`GenBank`
M-tree	4,683,360	38,008,305	3,729,887
PM-tree	7,509,804	55,213,829	6,605,421
GNAT	60,148,055	497,595,605	n/a
D-file	0	0	0

Table 1. Index construction costs (total distance computations).

5.3 Unknown Queries

The second set of tests was focused on the impact of D-file on querying when considering "unknown" queries, that is, query objects outside the database. It has to be emphasized that for unknown queries the D-file cannot take advantage of the trivial method GetDistance, because for an arriving query object there cannot be any record stored within D-cache at the moment the query starts. Thus, D-file can effectively use just the D-cache's non-trivial method GetLowerBoundDistance.

First, for the Corel database we have sampled queries with "snake distribution" – for an initially randomly sampled query object, its nearest neighbor was found, then the nearest neighbor's nearest neighbor, and so on. The intention for snake distribution was led by an anticipation that processing of a single "slowly moving" query will benefit from D-cache (which might provide tighter lower bounds). Because the D-cache should benefit from a long sequence of queries, we were interested in the impact of growing query batch size, see Figure 2a. As we can see, this assumption was not confirmed, the D-file costs generally follow the other MAMs – the performance gain of D-file queries is rather constant. Although in this test the query performance of D-file is not very good when compared to the other MAMs, in Figure 2b the situation takes into account also indexing costs. When just a small- or medium-sized query batch is planned for a database, the costs (total number of distance computations spent on indexing + all queries) show the D-file beats GNAT and PM-tree considerably. Note that

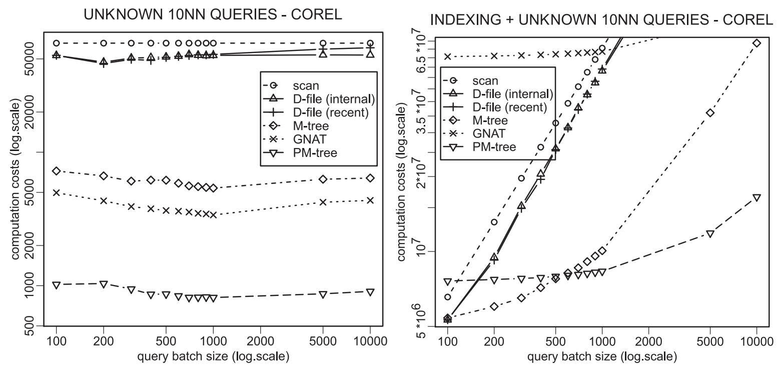

Fig. 2. Unknown 10NN queries on Corel: (a) queries only (b) indexing + queries.

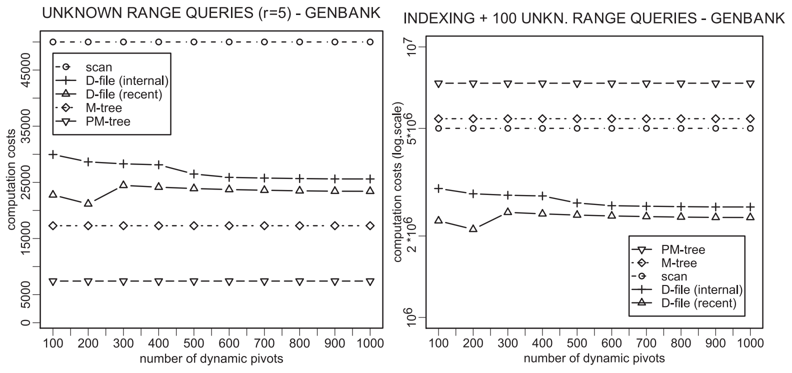

Fig. 3. Unknown range queries on GenBank: (a) queries only (b) indexing + queries.

in Figure 2b the costs are not averaged per query but total (because of correct summing with indexing costs).

The second "unknown" set of queries (randomly sampled range queries having radius $r = 5$) was performed on the GenBank database, considering growing number of D-cache's dynamic pivots, see Figure 3a. The costs decrease with growing number of pivots, the D-file shows a reduction of costs by 45%, when compared to the sequential search. The `recent` dynamic pivot selection is the winning one. In Figure 3b the situation is presented in indexing + querying costs (total costs for 100 queries were considered).

5.4 Database Queries

For the third set of experiments, we have considered database (DB) queries. As opposed to unknown queries, the DB queries consisted of query objects randomly sampled from the databases and having also the same ids as in the database. Although not as general as unknown queries, the DB queries are legitimate queries

– let us bring some motivation. In a typical general-purpose image retrieval scenario, the user does not have the perfect query image (s)he wants. Instead, (s)he issues any available "yet-satisfactory" image query (being an unknown query) and then iteratively browses (navigates) the database by issuing subsequent DB queries given by an image from the previous query result.

The browsing is legitimate also for another reason. Even if we have the right query image, the similarity measure is often imperfect with respect to the specific user needs, and so the query result may be unsatisfactory. This could be compensated by further issuing of DB queries matching the user's intent better.

For DB queries, we have anticipated much greater impact of D-cache, because distances related to a newly arriving query (being also DB object) could reside within D-cache since previous query processing. Consequently, for DB queries also the trivial GetDistance method can effectively take advantage, so that we could obtain an exact distance for filtering, rather than only a lower bound.

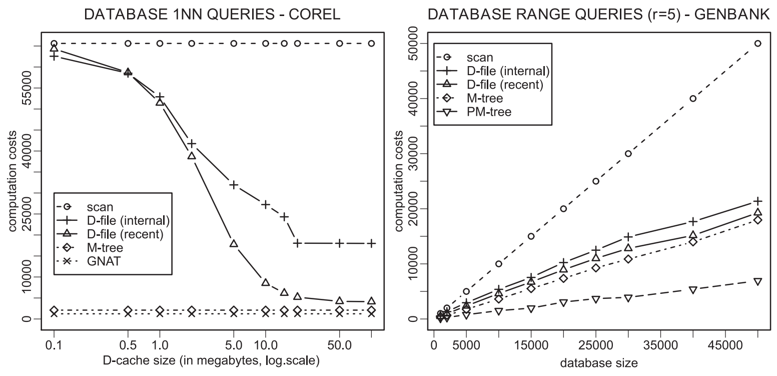

Fig. 4. (a) DB 10NN queries on Corel. (b) DB range queries on GenBank.

In Figure 4a, the results of 1NN queries on Corel are presented, considering varying D-cache storage capacity (we actually performed 2NN search, because the query object is DB object). We observe that a small D-cache is rather an overhead than a benefit, but for growing D-cache size the D-file costs dramatically decrease (to 6% of seq. search). At 10–20 MB the D-cache is large enough to store all the required distances, so beyond 20 MB the performance gain stagnates. However, note that there is a significant performance gap between D-files employing `recent` and `internal` pivot selection. This should be an evidence that exact distances retrieved from D-cache are not the dominant pruning factor even for DB queries, because the GetDistance method is pivot-independent. Hence, the effect of non-trivial "lowerbounding" is significant also for DB queries.

In the fourth experiment, see Figure 4b, the growing GenBank database was queried on range. Here, the D-file performance approaches M-tree, while the performance scales well with the database growth. In the last experiment, see Figure 5a, we tested the effect of the number of dynamic pivots on the largest

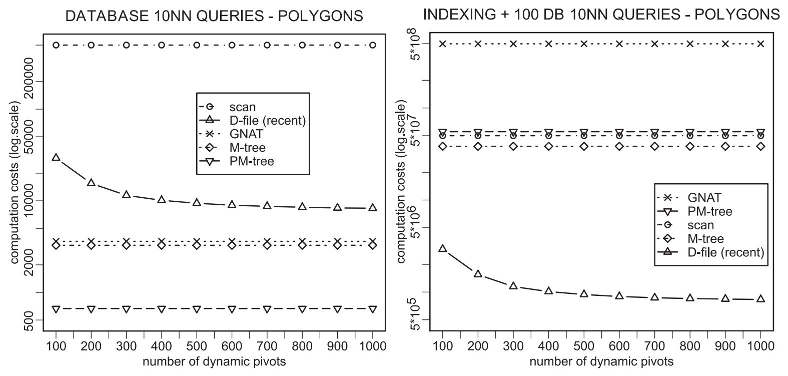

Fig. 5. DB 10NN queries on Polygons: (a) queries only (b) indexing + queries.

database – Polygons. For D-file, the costs fell down to 1.7% of sequential search, while they were decreasing with the increasing number of D-cache's dynamic pivots. In Figure 5b, the total indexing + querying costs are presented for 100 10NN queries, beating the competitors by up to 2 orders of magnitude.

6 Conclusions

In this paper, we presented the D-file – the first index-free metric access method for similarity search in metric spaces. The D-file operates just on the original database (i.e., it does not use any indexing), while, in order to support efficient query processing, it uses lower bound distances cheaply acquired from the D-cache structure. The D-cache is a memory-resident structure which keeps track of distances already computed during the actual runtime session.

The experiments have shown that D-file can compete with the state-of-the-art index-based MAMs (like M-tree, PM-tree, GNAT). Although in most cases the separate query costs are higher for D-file, if we consider the total indexing+querying costs, the D-file performs better for small- and middle-sized query batches. Thus, the usage of D-file could be beneficial either in tasks where only a limited number of queries is expected to be issued, or in tasks where the indexing is inefficient or not possible at all (e.g., highly changeable databases).

Future Work. In the future, we would like to employ the D-cache also by the index-based MAMs, since its ability to provide lower/upper bounds to distances is not limited just to D-file. Moreover, by using D-cache the index-based MAMs could take advantage not only from the improved query processing, but also from an increased performance of indexing.

Acknowledgments

This research was supported by Czech Science Foundation project no. 201/09/0683 (first author), and by FONDECYT (Chile) Project 11070037 (second author).

References

1. D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res*, 28(1):15–18, January 2000.
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*, 33(3):322–373, 2001.
3. S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584. Morgan Kaufmann, 1995.
4. S. D. Carson. A system for adaptive disk rearrangement. *Software - Practice and Experience (SPE)*, 20(3):225–242, 1990.
5. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
6. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97*, pages 426–435, 1997.
7. W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, 1984.
8. 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*, pages 43–50, New York, NY, USA, 2008. ACM.
9. F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti. Caching content-based queries for robust and efficient image retrieval. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 780–790, New York, NY, USA, 2009. ACM.
10. S. Hettich and S. Bay. The UCI KDD archive [http://kdd.ics.uci.edu], 1999.
11. G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.
12. H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.
13. 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`, pages 21–31, 2004.
14. T. Skopal, J. Pokorný, and V. Snášel. Nearest Neighbours Search using the PM-tree. In *DASFAA '05, Beijing, China*, pages 803–815. LNCS 3453, Springer, 2005.
15. J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.
16. J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
17. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
18. 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.