Metric Index: An Efficient and Scalable Solution for Similarity Search

David Novak  
Masaryk University  
Brno, Czech republic  
Email: david.novak@fi.muni.cz

Michal Batko  
Masaryk University  
Brno, Czech republic  
Email: batko@fi.muni.cz

Abstract—Metric space as a universal and versatile model of similarity can be applied in various areas of non-text information retrieval. However, a general, efficient and scalable solution for metric data management is still a resisting research challenge. We introduce a novel indexing and searching mechanism called Metric Index (M-Index), that employs practically all known principles of metric space partitioning, pruning and filtering. The heart of the M-Index is a general mapping mechanism that enables to actually store the data in well-established structures such as the B+-tree or even in a distributed storage. We have implemented the M-Index with B+-tree and performed experiments on a combination of five MPEG-7 descriptors in a database of hundreds of thousands digital images. The experiments put under test several M-Index variants and compare them with two orthogonal approaches – the PM-Tree and the iDistance. The trials show that the M-Index outperforms the others in terms of efficiency of search-space pruning, I/O costs, and response times for precise similarity queries. Furthermore, the M-Index demonstrates an excellent ability to keep similar data close in the index which makes its approximation algorithm very efficient – maintaining practically constant response times while preserving a very high recall as the dataset grows.

Keywords—metric space; similarity search; data structure; approximation; scalability

I. INTRODUCTION

There are many techniques that focus on searching in textual or vector-based data. However, these techniques are not always sufficient for current digital data types or their efficiency is significantly reduced, for example because of the phenomenon referred to as the curse of dimensionality. A more general metric-space abstraction allows to grasp a wider variety of these data types. However, after more than a decade of research, efficiency and scalability of metric access methods is still an issue.

In this paper, we introduce a novel index structure for metric data which builds upon a long-term research in this area. Metric Index (M-Index) defines a universal mapping schema from a generic metric space to a real numeric domain. Importantly, this schema has the ability to preserve the proximity of data, i.e. it maps similar metric objects to close numbers in the numeric domain. The M-Index indexing and searching mechanisms use a set of reference objects and synergically exploit practically all known metric-based principles of data partitioning, pruning and filtering.

The mapping character of the M-Index enables to use well-established techniques such as the B+-tree or the distributed hash table Skip Graphs [1] to efficiently manage and search the actual data. We also propose an approximate search strategy that utilizes the M-Index structure and has a tunable tradeoff between the response time and the recall of the approximate answer.

We implemented the M-Index on top of a B+-tree structure and used it to prove our concept by numerous experiments on a complex real-life dataset – a collection of 200,000 images taken from the photo-sharing site Flickr. These images are compared by a combination of five MPEG-7 visual descriptors [2] extracted from every image. The results confirm that the M-Index outperforms current best metric indexing techniques. Our experiments with approximate search strategies show that the M-Index can preserve almost precise recall while saving a significant portion of the search costs.

Section II of this paper introduces the basics of metric-based searching. In Section III we thoroughly describe the M-Index – its core ideas, structure, and search algorithms. Section IV is dedicated to experiments that test the efficiency and scalability of various M-Index settings and variants. Our technique is compared to other related work in Section V and the paper concludes in Section VI.

II. PRELIMINARIES: METRIC SEARCHING

Metric space \( \mathcal{M} \) is a pair \( \mathcal{M} = (\mathcal{D}, d) \), where \( \mathcal{D} \) is a domain of objects and \( d \) is a total distance function \( d : \mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R} \) satisfying these postulates for all \( o, p, q \in \mathcal{D} \):

\[
\begin{align*}
    d(o, p) & \geq 0 \quad \text{(non-negativity)}, \\
    d(o, p) & = 0 \quad \text{iff} \quad o = p \quad \text{(identity)}, \\
    d(o, p) & = d(p, o) \quad \text{(symmetry)}, \\
    d(o, p) & \leq d(o, q) + d(p, q) \quad \text{(triangle inequality)}.
\end{align*}
\]

Throughout this paper, we assume that the distance function \( d \) is normalized: \( d : \mathcal{D} \times \mathcal{D} \rightarrow [0, 1] \). This can be achieved by dividing the distance by a constant greater than the maximal value of \( d \) expected in the data domain. In practice, such constant can always be determined and the transformed space retains all properties of the original one.
**Similarity Queries**

The metric space as a model of similarity is typically searched according to the query-by-example paradigm – the query is formed by an object \( q \in D \) and some constraint on the data to be retrieved from the indexed dataset \( X \subseteq D \). We focus on two basic types of these queries: (1) the range query \( R(q, r) \), which retrieves all objects \( o \in X \) within the range \( r \) from \( q \) (i.e. \( \{ o \in X | d(q, o) \leq r \} \)), and (2) the nearest neighbors query \( k\text{NN}(q, k) \), which returns the \( k \) objects from \( X \) with the smallest distances to \( q \).

Evaluating similarity queries precisely may be costly in terms of both I/O and CPU costs. The approximate strategy with a certain level of inaccuracy may be tolerable for many applications. Quality of query answer can be measured by recall, i.e. the percentage of precise answer that is returned by the approximate search. See relevant literature [3], [4] for details on metric indexing and searching.

III. **Metric Index**

In this section, we introduce principles, architecture and search algorithms of the M-Index. We uncover the concepts in three steps following their natural development.

A. **M-Index Level One**

The primary concept of the M-Index was inspired by iDistance [5], which is an index method for similarity search in vector spaces. Having a sample set \( S \subseteq D \), the iDistance partitions \( S \) into \( n \) clusters and establishes a reference point \( p_i \) for each cluster \( C_i \), \( i \in \{0, \ldots, n-1\} \). Every object \( o \in X \) is then assigned a numeric key according to the distance to its cluster’s reference object. Having a large-enough constant \( c \) to separate individual clusters, the iDistance key for an object \( o \in C_i \) is

\[
i\text{Dist}(o) = d(p_i, o) + i \cdot c.
\]

This formula maps all objects in any cluster \( C_i \) to interval \([i \cdot c, (i+1) \cdot c)\) – see Figure 1a. Data objects are then stored in a B\(^+\)-tree according to their \( i\text{Dist} \) keys. During a search, the iDistance space is pruned by the principle illustrated in Figure 1b – for a range query \( R(q, r) \), we can determine several intervals of the \( i\text{Dist} \) keys which need to be accessed in order to process the query. Further details can be found in relevant literature [5].

M-Index generalizes the iDistance so that it can be used for general metric spaces – not restricting it purely to vector spaces. This is done by selecting a set of \( n \) pivots \( p_0, p_1, \ldots, p_{n-1} \) a priori (from a sample dataset \( S \subseteq D \)) and then applying the Voronoi-like partitioning to divide the space into \( n \) partitions (we will continue using the term clusters). Figure 2 shows an example of this partitioning in 2D for four pivots (\( n = 4 \)).

The mapping schema follows the idea of iDistance; since we assume a normalized metric space (Section II), separation constant \( c \) is not necessary any more – see Figure 2 (right).

B. **Multi-level M-Index**

In order to make the M-Index scalable for growing amounts of data, we would like to apply further partitioning of the clusters. Before we describe the principles of multi-level M-Index, let us state one preliminary definition. Having a fixed set of \( n \) pivots \( \{p_0, p_1, \ldots, p_{n-1}\} \) and an object \( o \in D \), let

\[
(\cdot)_o : \{0, 1, \ldots, n-1\} \longrightarrow \{0, 1, \ldots, n-1\}
\]

be a permutation of indexes such that

\[
d(p_{(0)_o}, o) \leq d(p_{(1)_o}, o) \leq \cdots \leq d(p_{(n-1)_o}, o).
\]

In other words, sequence \( p_{(0)_o}, p_{(1)_o}, \ldots, p_{(n-1)_o} \) is ordered with respect to distances between the pivots and object \( o \).

The M-Index with \( l \) levels, where \( l \) is an integer \( 1 \leq l \leq n \), partitions space \( D \) into \( n \cdot (n-1) \cdot \cdots \cdot (n-l+1) \) partitions (clusters) using the following recursive Voronoi partitioning: On the first level, each object \( o \in D \) is assigned to its closest pivot \( p_{(0)_o} \) – clusters \( C_i \) are formed in this way. On the second level, each cluster \( C_i \) is partitioned into \( n-1 \) clusters by the same procedure using set of \( n-1 \) pivots \( \{p_1, p_2, \ldots, p_{n-1}\} \) creating clusters \( C_{i,j} \). In other words, cluster \( C_{i,j} \) is formed by objects \( o \in X \) for which pivot \( p_j \) is the closest and \( p_i \) the second closest one: \( (0)_o = i \) and \( (1)_o = j \). This partitioning process is repeated \( l \)-times. Figure 3 (left) shows an example of the M-Index partitioning for \( l = 2 \).

Following the pattern sketched in the previous section, M-Index defines a mapping \( k\text{ey}_l : D \longrightarrow \mathbb{R} \). Let \( o \in D \) belong to cluster \( C_{i_0,i_1,\ldots,i_{l-1}} \). The integral part of \( k\text{ey}_l(o) \) identifies the cluster: it is equal to a number “\( i_{0}i_{1} \ldots i_{l-1} \)” written in numeral system with base \( n \). The fractional part

![Figure 1. Principles of the iDistance.](image-url)
of $key_l(o)$ is the distance between object $o$ and its closest pivot $d(p_{(0)}o, o)$. The overall M-Index mapping is defined by the following formula:

$$key_l(o) = d(p_{(0)}o, o) + \sum_{i=0}^{l-1} (i)_{o} \cdot n^{(l-1-i)}.$$  

(1)

Figure 3 (right) shows the mapping for a two-level M-Index ($l = 2$). In order to keep the figure lucid, the mapping is shown only for clusters $C_{1,*}$. The size of the domain is $4^2 = 16$ and objects from cluster $C_{i,j}$ are mapped to interval $[i \cdot n + j, i \cdot n + j + 1]$). For example, if the structure has four pivots ($n = 4$), an object $o \in C_{1,2}$ with $d(p_1, o) = 0.2$ is assigned key $key_2(o) = 0.2 + (12)_4 = 6.2$.

Note that not all partitions exist for particular space and set of pivots – in this example, clusters $C_{0,2}$ and $C_{2,0}$ do not exist. Corresponding intervals of the real domain are empty (as well as intervals reserved for clusters $C_{i,i,...}$). These gaps in the M-Index key domain do not cause any problems as long as the data is indexed by a B$^+$-tree or a similar efficient structure. The question of the number of existing partitions, or distance permutations, was studied recently [6] and depends on the number of pivots, complexity of the space and other factors.

C. M-Index with Dynamic Level

The multi-level M-Index, as described in the previous section, improves the clustering ability over the single-level M-Index by gathering close objects to smaller partitions. As we will see below, this can improve both precise and approximate searching strategies. Theoretically speaking, the more levels the M-Index has, the more efficient the searching is, because the search space is pruned better. On the other hand, large numbers of small clusters lead to a higher fragmentation of the data accessed by a query and to more demanding management during a query processing. In practice, this can exceed the profit gained from more levels.

Let us enhance the concept of M-Index by introducing the dynamic number of levels which enables (1) to dynamically
increase the number of levels (depth) only for large clusters, while (2) the increase requires only local modifications of the M-Index keys. This improvement requires the following modifications of the M-Index concept:

- a fixed “maximal M-Index level” \( 1 \leq l_{\text{max}} \leq n \) is chosen and the domain for this maximal level \( \mu^{l_{\text{max}}} \) is allocated.
- the key formula (1) for level \( l \), \( 1 \leq l \leq l_{\text{max}} \) is modified to:

\[
key_l(o) = d(p_{(0)\to l}, o) + \sum_{i=0}^{l-1} (i)_{\to l} \cdot n^{l_{\text{max}}-1-i} \cdot 2^j, \quad (2)
\]

- a dynamic cluster tree is created which specifies actual depth for specific M-Index clusters.

The schema of this tree-like structure for \( l_{\text{max}} = 3 \) is sketched in Figure 4. Objects are assigned keys according to the lowest levels at specific parts of the tree. Typically, we set a limit on the data volume stored in a cluster and increase the level of a cluster when its limit is exceeded.

![Dynamic cluster tree, \( l_{\text{max}} = 3 \).](image)

Partitioning of a cluster by one level means local reordering of the data belonging to the cluster. If we align the number of pivots \( n \) to a power of two then each level of the key, \( 1 \leq l \leq l_{\text{max}} \) defines a certain number of bits of the integral part of key\(_{l_{\text{max}}}^{*}\). Then this key-assignment approach becomes similar to extensible hashing [7] and partitioning of a cluster only means considering more bits of the key\(_{l_{\text{max}}}^{*}\). In practice, the \( l_{\text{max}} \) is only limited by \( n \) and eventually by the range of the numeric representation of key.

Let us denote \( C_{p,*} \) all clusters having pivot \( p \) on the first level. Recall that all keys in these clusters contain information about distance between the objects and pivot \( p \). We let the leaf nodes of the cluster tree store minimal and maximal keys in the corresponding cluster. These bounds represent the minimal/maximal object-pivot distances and are used for space pruning, as discussed below.

**D. M-Index Search Principles**

In general, indexing and search-space pruning in metric spaces can be done only on the basis of mutual distances of the objects and some designated objects (pivots). There are several theoretical principles of space pruning and data filtering [3] and the M-Index tries to exploit all of these. Let us focus on algorithm for range query \( R(q, r) \) as the basic similarity query.

The first step is to calculate distances \( d(p_i, q), \ i = 1, \ldots, n \), i.e., the distances from the query object to all the \( n \) pivots. The algorithm considers clusters on levels \( 1, \ldots, l \) (for the fixed-level M-Index) or by traversing the cluster tree and tries to prune whole clusters. Due to the repetitive Voronoi partitioning and according to **Double-Pivot Distance Constraint** [3], cluster \( C_i \) can be skipped when

\[
d(p_i, q) - d(p_{(0)\to l}, q) > 2 \cdot r
\]

where \( p_{(0)\to l} \) is the pivot closest to \( q \). Because the Voronoi partitioning is repeated \( l \)-times for level \( l \), this rule can be applied \( l \)-times for cluster \( C_{(p_0, \ldots, p_{l-2})} \). On each level \( j, \ 1 \leq j \leq l \) pivots \( p_{(l_{-j-2})}, \ldots, p_{(l_{-1})} \) are ignored by the pruning mechanism because they were skipped by Voronoi partitioning on level \( j \). A similar pruning mechanism is used in Spatial Approximation Tree [8], the M-Index just uses the same set of pivots repeatedly.

Knowing the minimal and maximal objects-pivot distance for objects in clusters (stored in leaf nodes of the cluster tree), we can apply **Range-Pivot Distance Constraint** [3] and skip accessing a cluster \( C_{p,*} \) if

\[
d(p, q) + r < r_{\text{min}} \quad \text{or} \quad d(p, q) - r > r_{\text{max}}
\]

where \( r_{\text{min}} \) and \( r_{\text{max}} \) are minimal and maximal distances of objects within a particular cluster, i.e. \( \min\{d(p, o) | o \in C_{p,*}\} \). Similar approach is used in GNAT [9] for both leaf and internal nodes of a Voronoi-like tree.

When none of the above-mentioned mechanisms filters out cluster \( C_{(p_0, \ldots, p_{l-1})} \) on leaf level then we determine an interval of the key domain to be searched within that cluster:

\[
[\min\{d(p, o) | o \in C_{(p_0, \ldots, p_{l-1})}\}, \min\{d(p, o) + r | o \in C_{(p_0, \ldots, p_{l-1})}\}]
\]

where both limits are shifted by the integral part of the keys for cluster \( C_{(p_0, \ldots, p_{l-1})} \) according to key formulas (1) or (2). This mechanism is taken from iDistance and it is a direct application of **Object-Pivot Distance Constraint** [3].

In some spaces, computation of the distance function \( d \) can be extremely time-consuming and an important objective of the data structures is to avoid distance computations at the price of storing additional metadata with the objects. For each object \( o \in X \), the M-Index can store distances \( d(p_0, o), \ldots, d(p_{n-1}, o) \) (computed during insertion) and use them at query time to avoid computation of \( d(q, o) \) if

\[
\max_{i \in \{0, \ldots, n-1\}} |d(p_i, q) - d(p_i, o)| > r.
\]

This standard mechanism is referred to as **Pivot Filtering** [3].

**Nearest-Neighbors Search**

Algorithm for \( kNN(q, k) \) queries follows the general schema applied in the iDistance. Initially, it accesses intervals of the key domain that correspond to a range query with
a small radius. Then in each iteration, the query radius is increased and the data already accessed is skipped. When current query answer contains (any) \( k \) objects, distance between \( q \) and the \( k \)-th answer object is an upper bound on the distance to the actual \( k \)-th nearest neighbor of \( q \). This distance can be used as current query radius to prune the cluster tree by the mechanisms mentioned above.

\[ \text{penalty}(C_q) = \sum_{j=0}^{l-1} \max \{ d(p_{i_j}, q) - d(p_{(j+1)_{q}}, q), 0 \} . \]

Figure 5 provides an example in which \( C_q = C_{0.3} \) and the query-pivots distances are \( d(p_0, q) = 0.2 \), \( d(p_1, q) = 0.3 \), \( d(p_2, q) = 0.5 \). We know that \( \text{penalty}(C_{0.3}) = 0 \) and penalties for other “close” clusters are, for instance:

\[ \text{penalty}(C_{0.1}) = d(p_0, q) - d(p_0, q) + (d(p_1, q) - d(p_3, q)) = 0.05, \]

\[ \text{penalty}(C_{3.0}) = (0.25 - 0.2) + \max \{ 0.2 - 0.25, 0 \} = 0.05, \]

\[ \text{penalty}(C_{1.0}) = (0.3 - 0.2) + \max \{ 0.2 - 0.25, 0 \} = 0.1. \]

Basically, the algorithm visits the clusters in the order of this penalty, which grows with the intuitive “distance” of the clusters from \( q \). Specifically, the algorithm maintains a dynamic queue of the cluster tree nodes; the queue is ordered nondecreasingly by the penalty of the clusters (nodes). Initially, the top-level clusters are put into the queue and then, in each step, the first node is removed and either (1) its child nodes are put into the queue or, if it is a leaf cluster, (2) its data is checked against the query condition. In order to make the penalties on various levels comparable, the penalty values are normalized by \( d(p_{(0)_{q}}, q) + \ldots + d(p_{(l-1)_{q}}, q) \).

The algorithm can stop at any time – for instance when a certain volume of data is processed. The approximate algorithm can prune the space by means of the distance between \( q \) and current \( k \)-th answer object, as mentioned in the previous section for \( k \text{NN} \) queries. The approximate algorithm can also run until the precise answer is reached.

**E. Approximate Search with M-Index**

Let us introduce the M-Index approximate search strategy for \( k\text{NN}(q, k) \) queries. The keystone of this technique is a heuristic which determines the order in which the algorithm explores individual M-Index clusters. This heuristic is based on analysis of the \( d(p_0, q), d(p_1, q), \ldots, d(p_{n-1}, q) \) distances and of the M-Index cluster tree (Figure 4).

Cluster \( C_{(i_0, \ldots, i_{l-1})_{q}} \), in which object \( q \) would be stored in the M-Index, is always searched (let us denote it \( C_q \)). The heuristic assigns the lowest penalty to this cluster. The penalty of a cluster is to reflect the “proximity” of the cluster to \( q \). Cluster \( C_q \) has the smallest sum of distances between \( q \) and the cluster’s pivots: \( d(p_{(i_1)_{q}}, q) + \ldots + d(p_{i_{l-1}q}, q) \). This sum grows for other clusters and this is reflected by the penalty. There are clusters with the same distance sum as \( C_q \), for instance \( C_{3.0} \) for \( C_q = C_{0.3} \). These clusters are also assigned higher penalties than \( C_q \). Specifically:

The key-indexing structure can be also distributed. An example of a structure which efficiently evaluates interval queries on a simple key domain is the Skip Graphs [1], which is a structured peer-to-peer network. Although the design of a distributed version of the cluster tree and of the query navigation algorithm are out of the scope of this paper, a single-level distributed M-Index restricted to the iDistance pruning corresponds to structure called M-Chord [10].

**F. Architectural of the Index**

The M-Index principles and algorithms introduced in the previous sections can be exploited to design various specific data structures. Data objects can be stored in any structure that would index them according to their M-Index keys, ideally with efficient evaluation of interval queries – \( B^+ \)-tree is a typical example. The data structure is to ensure actual efficient memory and disk management.

Implementation of the cluster tree is straightforward. Entry for cluster \( C_{i_0, \ldots, i_{l-1}} \) consists of: \( l \) tuple \( (i_0, \ldots, i_{l-1}) \), and either sub-tree pointers \( (\text{sub}_{i_0}, \ldots, \text{sub}_{i_{l-1}}) \) for an inner-node entry or \( r_{\text{min}}, r_{\text{max}} \) for a leaf entry – see Section III-D. The cluster tree is expected to be kept in main memory: in our implementation, the tree for 100,000 objects occupied about 60 kB of memory (\( l_{\text{max}} = 5, n = 20 \), leaf-cluster contained about 500 objects at the most).

IV. EXPERIMENTAL EVALUATION

In this section, we present experiments conducted on the \( B^+ \)-tree version of the M-Index. The structure was implemented in Java using the MESSIF [11] framework which provides extensive support for realization of metric-based indexing techniques. We evaluated the performance of the M-Index with various settings and compared these with two existing approaches – the iDistance [5] generalized for metric spaces (see Section III-A) and the PM-Tree [12]. The PM-Tree extends a popular structure M-Tree [13] by adding
space pruning and filtering using a static set of preselected pivots. In order to make the results comparable, all tested structures always use the same set of pivots (PM-Tree uses additional dynamically selected pivots [13]). All the structures were implemented using the MESSIF framework, use the same datasets, and run in the same environment.

A. Settings

The experiments were realized on a dataset of visual descriptors retrieved from digital images. Five MPEG-7 features were extracted from each image: Scalable Color, Color Structure, Color Layout, Edge Histogram, and Homogeneous Texture [2]. The images and the descriptors were taken from the CoPhIR Database [14]. Each of these descriptors is compared using a metric function and we use a weighted sum of these distances to aggregate them into a single metric space [14]. In total, this representation can be viewed as a 280-dimensional vector together with a very complex distance function. It requires about 1 kB of memory and evaluation of one aggregate distance function takes approx. 0.01 ms on standard hardware. The intrinsic dimensionality [15] of the dataset is 12.9 which makes it rather difficult to index. We used sets of 20,000 to 200,000 images – actual size is specified for each experiment. All similarity queries were processed for fifty query objects and we always present average results over these fifty queries.

Search efficiency of the structures is measured by I/O costs, computational costs, and response times. Specifically, the I/O costs express the number of 4 kB-block reads realized during the search. The computational costs are measured as the number of evaluations of the distance function \( d \) (distance computations). This number of DC is influenced by the pivot filtering (see Section III-D) applied by all structures under test. When we want to eliminate this influence, we report on the number of data objects accessed during the search. For the approximate evaluation strategies, we measure the answer quality by the recall (see Section II).

B. Construction Costs

Due to the mapping nature of the M-Index, the computational construction costs can be directly derived from the size of the dataset and the number of pivots used. Table I compares construction costs of M-Index and PM-Tree when both use the same set of 20 pivots.

Table I

<table>
<thead>
<tr>
<th>dataset size</th>
<th>20,000</th>
<th>80,000</th>
<th>140,000</th>
<th>200,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-Index</td>
<td>400,000</td>
<td>1,600,000</td>
<td>2,800,000</td>
<td>4,000,000</td>
</tr>
<tr>
<td>PM-Tree</td>
<td>1,205,538</td>
<td>6,299,207</td>
<td>11,627,729</td>
<td>16,897,996</td>
</tr>
</tbody>
</table>

The PM-Tree node-splits were realized using Slim-Tree algorithm [16]. We can see that the PM-Tree costs are \( 3-4 \times \) higher than for the M-Index. The I/O costs of the M-Index depend on the specific data-indexing structure. Our prototype uses the \( B^+ \)-tree, which has a very efficient disk management, so it required about 80% I/O operations compared to building costs of the PM-Tree.

C. Precise Similarity Search

In the first set of the experiments, we test efficiency of the data-space partitioning and subsequent pruning applied by the precise search strategies. In the experiment, we alter (1) the set of the M-Index pivots \( p_0, \ldots, p_{n-1} \) and (2) the size of the dataset stored. We put under test the static M-Indexes with various levels and the dynamic M-Index.

1) Number of Pivots: Figure 6 reports on the influence of the number of pivots \( n \) on M-Index search efficiency. We processed a set of \( kNN(q, 50) \) queries on three static indexes \( (l = 1, 2, 3) \) and on a dynamic M-Index with \( l_{\text{max}} = 5 \). All structures managed a dataset of 100,000 objects and the graph shows the number of accessed objects varying \( n \) from 1 to 50 (with the limitation that \( n \geq l \) and \( n \geq l_{\text{max}} \)).

![Figure 6. Pruning efficiency for various \( n \).](image)

We can observe the expected positive influence of growing \( n \) on the pruning efficiency and the trend is similar for all tested M-Index variants. In this experiment, the set of pivots was selected from a sample set by a variant of a standard incremental pivot-selection algorithm [17], which tries to maximize efficiency of pivot filtering. Other pivot selection approaches we tested, including random selecting, had similar performance and the trends are practically identical. In the rest of the trials, we will fixate the set of pivots to the first twenty from the collection used in this experiment.

2) Dataset Size: Figure 7 compares efficiency of the \( kNN \) query processing for several M-Index variants, the iDistance, and the PM-Tree. The three graphs show the number of data objects accessed (space pruning efficiency), I/O costs, and response times, respectively, while increasing the indexed data volume. In general, all these graphs have linear trend which is typical for all centralized metric structures that adopt precise search strategies.

Looking at graph (a), we can see that the M-Index with \( l = 1 \) is slightly more efficient than the iDistance and
that the multi-level M-Indexes prune the search space even better. The M-Index with \( l = 3 \) and the dynamic M-Index \( (l_{\text{max}} = 5) \) exhibit best pruning effect – they access 25% fewer objects than the PM-Tree. For some of these accessed objects, evaluation of the distance function is skipped due to the pivot filtering. In our experiments, the number of distances computations were about 2/3 of the number of accessed objects, but the differences between individual structures were smaller because all the structures use the same set of pivots for filtering.

The I/O costs, depicted in graph (b), reflect the accessed objects but this graph depicts the actual page reads realized during the query processing. We can see that all M-Index settings have similar results which confirms that the space-pruning improvement achieved by recursive partitioning is paid by higher data fragmentation. In general, the M-Index has about 15 % lower I/O costs than the PM-Tree.

As we mentioned in Section III-C, static M-Index with higher \( l \) has the disadvantage of creating too many small clusters for smaller datasets. Table II shows the number of clusters for different M-Index variants and various dataset sizes. As expected, static M-Indexes have rather stable number of clusters regardless of the dataset size while these numbers grow practically linearly for the dynamic M-Index (the maximal cluster size for dynamic M-Index was set to 512 kB). Higher fragmentation and management of many small clusters have negative effect on the overall efficiency of the search, which can be observed in Figure 7 (c) showing \( k\text{NN}(q, 30) \)-query response times. We can see that the M-Index with \( l = 3 \) is less efficient for smaller datasets and more efficient for larger ones than M-Index with \( l = 1, 2 \). The dynamic M-Index has the best response times – about 20 % better than the PM-Tree for the 200,000-objects dataset.

3) Observations: Generally, we can say that the M-Index fulfilled our expectations – its precise search strategies prune the search space well and outperforms the current top-end metric index structures. The number of pivots and levels has the expected influence and the dynamic M-Index seems to be the best in respect of both I/O costs and response times. Even though the M-Index lowers the search costs down, the percentage of accessed data is still relatively high. This fact confirms the well-known fact that precise metric search is expensive having typically linear scalability and this is the major motivation for approximate metric searching.

D. Approximate Search

In this section, we focus on performance of the M-Index approximate search. The efficiency of the technique depends on the potential to keep close data close in the index (clustering ability) and on the ability to identify “promising” parts of the index. We compare several M-Index variants and the PM-Tree structure. The PM-Tree employs an approximate algorithm [18] which uses a priority queue of structure nodes sorted according to a heuristic.

1) Maximal Percentage of Data: In the first experiment, we limit the percentage of data volume accessed by the approximate algorithms. Figure 8 shows the recall of approximate search while varying this percentage for dataset of 100,000 objects.

The graph demonstrates a good clustering ability of the M-Index which grows with level \( l \) and is best for the dynamic M-Index. Limiting the percentage of accessed data to 5%, the dynamic M-Index reaches 93% recall while the PM-Tree has recall about 50%. The PM-Tree has to access four times more data to reach 90% recall. These results also show that the approximate heuristic described in Section III-E works even in the complex metric space we use. Note that the M-Index pruning mechanisms are applied also for approximate search and, thus, the searching may stop even before reaching the accessed-data limit.
2) Maximal Fixed Volume of Data: In this final experiment, we fixate the limit of data accessed by the algorithm regardless of the size of the data being indexed. Graph in Figure 9 shows the search recall for this experiment limiting the number of accessed objects to 10,000.

We can see that PM-Tree recall is falling from 99% for the 20,000 dataset down to about 60% for the 200,000 dataset. As the M-Index level $l$ grows from 1 to 3, the recall becomes more stable and the dynamic M-Index reaches more than 93% recall even for the database of 200,000 objects.

The graph in Figure 10 shows that the approximate-search response times of the M-Index are very stable and significantly shorter than for the PM-Tree. The PM-Tree manages a dynamic queue of tree nodes sorted according to distances from the nodes’ pivots [18] and this proves to be more demanding than the M-Index approach. Moreover, the M-Index applies all its pruning and filtering mechanisms which may result in reaching the precise answer even before the stop condition applies.

Observations: The experiments demonstrated that the M-Index has a good clustering ability and that this feature is well utilized by the approximate algorithm. The efficiency of the approximate search grows with the M-Index level and is best for the dynamic M-Index. The processing costs of the algorithm are stable and very low and the dynamic M-Index reaches nearly constant approximate search efficiency as the volume of indexed data grows.

V. RELATED WORK

As described in Section III-A, the M-Index shares one of its core space pruning principles – the Object-Pivot Distance Constraint – with the dDistance technique [5]. However, the M-Index is defined for more general metric space domains and it exploits several other pruning techniques which result in more efficient range and $kNN$ similarity search in comparison with the dDistance. We have also compared the M-Index with the PM-Tree [12] structure, which is an improved variant of the well-established Metric Tree (M-Tree) [13]. It is a balanced disk-oriented tree structure that supports also approximate similarity search [18]. Our experiments show that the M-Index is noticeably faster when using precise evaluation strategy and it has a significantly better recall when applying approximate evaluation strategies; the M-Index recall has a significantly more stable trend when the dataset size increases. A Voronoi-like partitioning of the metric space is the core of the GNAT [9] technique, which is a multi-way tree structure that exploits the Range-Pivot Distance Constraint similarly to the M-Index. A faster and approximation-oriented structure Spatial Approximation Tree [8] builds a spanning tree of the stored objects based on their mutual metric distances. During query evaluation, this structure exploits the Double-Pivot Distance Constraint that is used also in the M-Index. This technique, however, is static in its nature and, although a dynamic extension [19] exists, its approximation recall and performance is in general not better than that of the PM-Tree. The D-Index [20] structure uses a hashing-like approach where objects are mapped to well-separated partitions by several split-functions. Since the M-Index assigns a simple number to each indexed object, it can be also viewed as a hashing technique. Moreover, objects that are close in the original metric space are mapped to close numbers by the M-Index, so the core property of the Locality Sensitive Hashing (LSH) [21] is satisfied. Compared with LSH, which is currently defined only for certain specific data spaces (distance functions), the M-Index broadens the usability of LSH approach to any generic metric space. Contrary to classic LSH, the M-Index has the ability to stop the search when precise answer is gathered.
VI. CONCLUSIONS AND FUTURE WORK

We have presented the M-Index – a novel indexing technique that exploits practically all known principles of metric space partitioning, pruning and filtering. The M-Index maps objects from any metric space to a simple numeric domain and then employs interval queries to implement its precise and approximate search algorithms. There are many techniques, such as B+-tree, that can speed up processing of interval queries and they can be directly employed by the M-Index. As such, the M-Index is highly extensible and can be used in many application areas.

Our experiments compare the M-Index with two structures – the PM-Tree and the iDistance. We show that the M-Index outperforms both of them in terms of I/O, computational costs and query response times. The dynamic variant of M-Index proves to be the best one and its performance indicators are the most stable with respect to a growing dataset. M-Index approximate search strategy reaches nearly constant response times and recall while the dataset size increases. Since the indexing directory grows logarithmically with the dataset size and since the storage can use a standard database technology, the M-Index is also highly scalable.

The principles introduced by the M-Index are directly applicable to a distributed environment. Our ongoing work is to apply the M-Index technique to the full CoPhIR database (100 million images) by employing a structured peer-to-peer network called Skip Graphs. We believe that a distributed M-Index will enable efficient similarity searches on datasets of billions of data objects and we would like to focus our future research in this area. Another stream of our research is to study and develop the M-Index mapping principle as a universal technique of locality sensitive hashing.

ACKNOWLEDGMENTS

This work has been supported by EU IST FP6 project 045128 (SAPIR), the national research projects GACR 201/08/P507, GACR 201/09/0683 and MSMT 1M0545. The hardware infrastructure was provided by the METACentrum under the research intent MSM683917201.

REFERENCES