Optimizing Query Performance with Inverted Cache in Metric Spaces

Matej Antol(✉) and Vlastislav Dohnal

Faculty of Informatics, Masaryk University, Botanicka 68a, Brno, Czech Republic
{xantol,dohnal}@fi.muni.cz
https://www.fi.muni.cz

Abstract. Similarity searching has become widely available in many on-line archives of multimedia content. Querying such systems starts with either a query object provided by user or a random object provided by the system, and proceeds in more iterations to improve user's satisfaction with query results. This leads to processing many very similar queries by the system. In this paper, we analyze performance of two representatives of metric indexing structures and propose a novel concept of reordering search queue that optimizes access to data partitions for repetitive queries. This concept is verified in numerous experiments on real-life image dataset.

Keywords: Similarity search · Nearest-neighbors query · Metric space · Inverted cache · Query optimization

1 Introduction

Multimedia retrieval systems have been becoming more and more applied to organize data archives of unstructured content, for example, photo stocks. Such systems provide content-based retrieval of data objects (e.g., images), so a user may find visually similar images to a given one. If he or she is not satisfied with the result, clicking on an interesting image in the answer may give better answer. This is called *browsing*. In another retrieval scenario, users may not have any particular search intent, but they rather like to inspect a multimedia collection. Here, a query-by-example search is not suitable in the first phases, because the user may not have any query object. So, the user would prefer a categorized view of data and then to dive into categories via regular query-by-example search to explore the collection. This is called *multimedia exploration* [4,15]. Such scenarios share the property that many queries issued to the system are alike, so search algorithms may optimize *repeated queries* to save computational resources.

In common database technology, the query efficiency is typically supported by various indexing structures, storage layouts and disk caching/buffering techniques. So the number of disk I/Os needed to answer a query is greatly reduced. In modern retrieval systems, analogous approaches are used too. However, to handle more complex and unstructured data, they are extended to high-dimensional spaces or even distance spaces where no implicit coordinate system

J. Pokorný et al. (Eds.): ADBIS 2016, LNCS 9809, pp. 60–73, 2016.
DOI: 10.1007/978-3-319-44039-2_5

is defined [19]. The problem of *dimensionality curse* then often appears [6]. In particular, it states that indexing structures stop exhibiting logarithmic complexity in query evaluation but rather become linear [7,8]. This is typically attributed to the fact that many data partitions must be visited by an indexing mechanism due to high overlaps among them. Efficiency is then improved by further filtering conditions and optimized node-splitting strategies in the indexing structures [9,22] or by sacrificing precision in query results (approximate querying) [1,12,13].

In this paper, we study the issue of evaluating repeated queries and propose a solution that prioritize data partitions during query evaluation to deliver query results earlier. Instead of caching answers to particular queries, our proposal stores *usefulness* of data partitions and localizes such information to increase effectiveness of accessing data partitions during evaluation of new queries. Moreover, this concept is generally applicable to any metric indexing structure [24].

The paper is structured as follows. In the next section, we summarize related work. The necessary background of similarity searching and indexing is given in Sect. 3. Analysis of performance of current indexes that motivates our work is presented in Sect. 4. The proposal of so-called Inverted Cache Index is described in Sect. 5 and its evaluation is in Sect. 6. Contributions of this paper and possible future extensions are summarized in Sect. 7.

2 Related Work

There are many approaches [8,24] for indexing metric spaces that were developed as generally applicable to a large variety of domains. To process large datasets, they are designed as disk oriented. The data partitioning principles are typically based on (i) hierarchical clustering (e.g. M-tree [9]), where each subtree is covered by a preselect data object (pivot) and a covering radius; (ii) voronoi partitioning (e.g. M-index [17]), where subtrees are formed by assigning objects to the closest pivot; and (iii) precomputed distances (e.g. LAESA [23]), where no explicit structure is built, but rather distances among data objects are stored.

Optimizations of query-evaluation algorithms are based on extending a hierarchical structure with additional precomputed distance to strengthen filtering capabilites, e.g. M*-tree [21], cutting local pivots [18]; or on exploiting large number of pivots in a very compact and reusable way, e.g. permutation prefix index [11]. These techniques, however, does not analyze the stored data and accesses to them, but rather constrain data partitions as much as possible.

Another way to make query evaluation much faster is to trade accuracy – approximate searching. There are many approaches that apply early-termination and relaxed-branching strategies to stop searching when query result does improve marginally. A recent approach called spatial approximation sample hierarchy [13] builds an approximated near-neighbor graph and does not exploit triangle inequality to filter out irrelevant data partitions. This was further improved and combined with cover trees to design Rank Cover Tree [12].

Distance Cache [20] is a main-memory structure that maintains dynamic information to determine tight lower- and upper-bounds of distances between

data objects. This information is collected based on previous querying and is applied to newly posed queries. So it is applicable to any metric indexing structure, which is the resemblance with the approach proposed in this paper. Distance Cache may independently provide further filtering power to our proposal. We expect it would mainly contribute to M-tree's performance rather than to M-index'es, so our results on M-tree with Distance Cache would approach the ones on M-index.

A cache-like structure for similarity queries, called Snake Table, was proposed in [2]. It is a dynamically-built structure for optimizing all queries corresponding to one user session. It remembers results of all queries processed so far and constructs a linear AESA over them. It accelerates future queries. This approach behaves clearly as a traditional *cache*.

3 Background

We assume unstructured data are modeled in metric space and organized in appropriate indexing techniques here. Before presenting experience with metric structures that motivated our work, we summarize the necessary background.

3.1 Metric Space and Similarity Queries

The metric space $\mathcal{M}$ is defined as a pair $(\mathcal{D}, d)$ of a domain $\mathcal{D}$ representing data objects and a pair-wise distance function $d : \mathcal{D} \times \mathcal{D} \mapsto \mathbb{R}$ that satisfies:

$\forall x, y \in \mathcal{D}, d(x, y) \geq 0$	non-negavity,
$\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$	symmetry,
$\forall x, y \in \mathcal{D}, x = y \Leftrightarrow d(x, y) = 0$	identity,
$\forall x, y, z \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality.

The distance function is used to measure similarity between two objects. The shorter the distance is, the more similar the objects are. Consequently, a similarity query can be defined. There are many query types [10] but the range query and k-nearest neighbor query are most important ones. The range query $R(q, r)$ specifies all database objects within the distance of r from q. In particular, $R(q, r) = \{o | o \in X, d(q, o) \leq r\}$, where $X \subset \mathcal{D}$ is the database to search in. In this paper, we primarily focus on k-nearest neighbors query since it is more convenient for users. The user wants to retrieve k most similar objects to a query: $kNN(q) = A, |A| = k \wedge \forall o \in A, p \in X - A, d(q, o) \leq d(q, p)$.

3.2 Indexing and Query Evaluation

To organize a database to answer similarity queries efficiently, many indexing structures have been proposed [24]. Their principles are twofold: (i) recursively applied data partitioning/clustering defined by a preselected data object called *pivot* and a distance threshold, and (ii) in effective object filtering using lower-bounds on distance between a database object and a query object. These principles are firstly surveyed in [8].

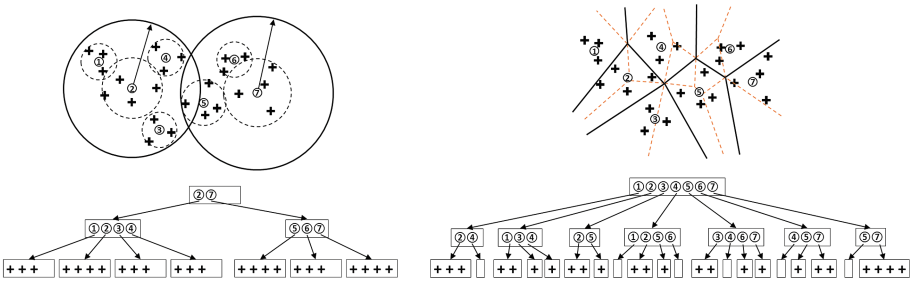


Fig. 1. Partitioning principles of M-tree (left) and M-index (right)

In this paper, we use a traditional index M-tree [9] and a more recent technique M-index [17]. Both these structures create an internal hierarchy of nodes partitioning data space into many buckets – an elementary object storage. Please refer to Fig. 1 for principles of their organization. M-tree organizes data objects in compact clusters created in the bottom-up fashion, where each cluster is represented by a pair (p, r^c) – a pivot and a covering radius, i.e. distance from the pivot to the farthest object in the cluster. On the other hand, M-index applies Voronoi-like partitioning using a predefined set of pivots in the top-down way. In this case, clusters are formed by objects that have the cluster's pivot as the closest one. On next levels, the objects are reclustered using the other pivots, i.e. eliminating the pivot that formed the current cluster. Buckets of both the structures store objects in leaf nodes, as is exampled in the illustration. So we use *leaf node* and *bucket* interchangeably.

An algorithm to evaluate a kNN query constructs a priority queue of nodes to access. The priority is defined in terms of a lower bound on distance between the node and the query object. So a probability of node to contain relevant data objects is estimated this way. In detail, the algorithm starts with initializing the queue with the root node of hierarchy. Then it repeatedly pulls the head of priority queue until the queue is empty. The algorithm terminates immediately, if the head's lower bound is greater than the distance of current k^{th} neighbor to the query object. If the pulled element represents a leaf node, its bucket is accessed and all data objects stored in there are checked against the query, so query's answer is updated. If it is a non-leaf node, all its children are inserted into the queue with correct lower bounds estimated. M-tree defines the lower bound for a node (p, r^c) and a query object q as the distance $d(q, p) - r^c$. For space constraints, we do not include additional M-tree's node filtering principles as well as the M-index's approach that is elaborate too.

4 Index Structure Effectiveness

Interactivity of similarity queries is the main driving force to make content-based information retrieval widely used [14]. In the era of Big Data, near real-time execution of similarity queries over massive data collections is even more

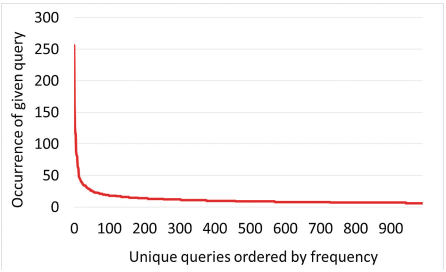


Fig. 2. Distribution of top-1000 unique queries ordered by their appearances

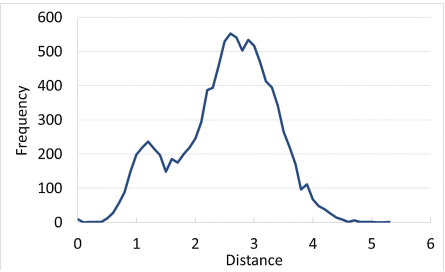


Fig. 3. Density of distances among top-1000 query objects

important, because it allows various analytics to be implemented [5]. In this section, we present motivating arguments based on experience with a real-life content-based retrieval system.

4.1 Query Statistics

From Google Analytics, we have obtained statistics about queries processed in a demonstration application [16]. This application implements content-based retrieval on the CoPhIR data-set [3] consisting of 100 million images. The application's web interface[1] shows similar images to a query image chosen randomly from 100 preselected images. Then the user may browse the collection by clicking "Visually similar", or obtain a new query by a regular keyword search. Thus this application fits our motivating browsing and exploring scenarios perfectly.

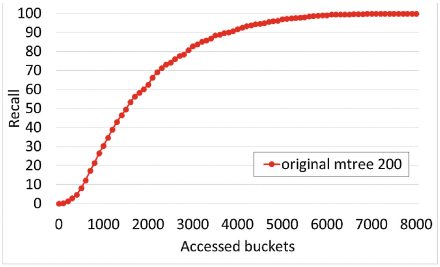
Figure 2 shows absolute frequencies of individual top-1000 queries that were executed during the application's life time (launched in Nov. 2008). This power-law like distribution is attributed to the way of presenting an initial search to a new website visitor. Figure 3 depicts density of distances among these queries, so the reader may observe there are very similar query objects as well as distinct ones. This proves that the users were also browsing the data collection.

4.2 Indexing Structure Performance

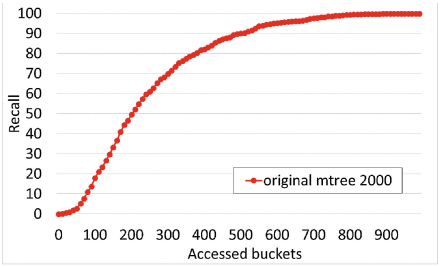
The main drawbacks of indexing structures in metric spaces are a high amount of overlaps of their substructures, and not very precise estimation of lower bounds on distances between data objects and a query object. So the kNN-query evaluation algorithm often accesses large portion of indexing structure's buckets to obtain precise answer to a query. In Fig. 4, we present the progress of recall while constraining the number of accessed buckets.

The selected indexing structure representatives were populated with 1 million data objects from the CoPhIR dataset and 30NN queries for the top-1000 query objects were evaluated. The figures present average values of recall of such

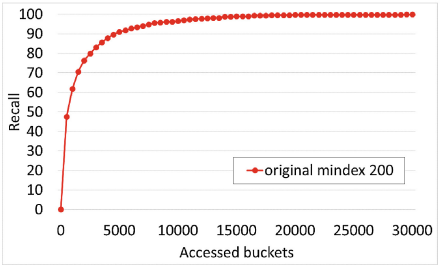
[1] http://mufin.fi.muni.cz/imgsearch/similar.



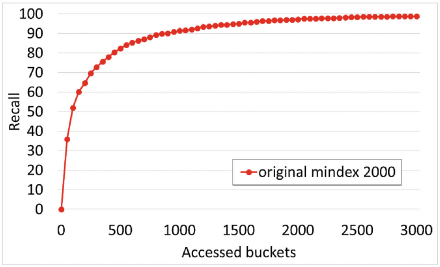
(a) M-tree 200



(b) M-tree 2000



(c) M-index 200



(d) M-index 2000

Fig. 4. Recall of 30NN for increasing number of accessed buckets of M-tree and M-index and different bucket capacities (200 and 2,000)

queries. We have tested two configurations for both M-tree and M-index. The capacity of buckets was constrained to 200 and 2,000 objects to have bushier and more compact structures. Table 1 summarizes information about them. To this end, M-index's building algorithm was initialized with 128 pivots picked at random from the dataset and the maximum depth of M-index's internal hierarchy was limited to 8. From the statistics, we can see that M-tree can adapt to data distribution better than M-index and does not create very low occupied buckets, so M-tree is more compact data structure.

From the query evaluation point of view, which is the main point of interest of this paper, both the structures need to access large amounts of buckets to

Table 1. Structure details of tested indexing techniques.

Indexing structure	Bucket capacity	Buckets in total	Avg. bucket occupation	Hierarchy height	Internal node capacity
M-tree 200	200	11,571	43 %	4	50
M-tree 2000	2,000	1,124	44 %	3	100
M-index 200	200	62,049	8 %	8	not defined
M-index 2000	2,000	10,943	4.6 %	8	not defined

obtain 100 % recall. M-tree needs to check objects in 8,100 (70 %) and 1,000 (89 %) buckets for 200 and 2,000 bucket capacities, respectively. M-index visits 30,000 (47 %) and 6,500 (58 %) buckets for 200 and 2,000 bucket capacities, respectively. To complete 95 % recall, the requirements are lower – 40 % and 53 % for M-tree versus 12 % and 13 % for M-index. From these results, we can conclude that both the structures are not very effective in accessing buckets with relevant data early. M-index's principle of partitioning, however, is much more effective in early stages of searching because it can get 50 % of correct objects within 1 percent of accessed buckets. M-tree locates only about 15 % of correct objects within the same ratio. In M-tree with 2,000 bucket size, the average number of leaf nodes containing 30 nearest neighbors is 17.

5 Inverted Cache Index

In this section, we propose a technique for prioritizing nodes in indexing hierarchies to locate relevant data objects earlier. This technique is based on exploiting knowledge of accessing data partitions during query evaluation. So, a query evaluation algorithm can adaptively re-order its priority queue with respect to *usefulness* of the current node, i.e. the node's chance to contribute to query result. We call this technique *Inverted Cache Index* (ICI), since it does not record the queries processed so far, but rather the number of times a given partition/bucket (or data object) contributed to the final result of such queries.

Each object and node in an indexing structure has a memory of its historical accesses. This memory is used for storing ICI value. After completing evaluation of a query, its final answer is checked and ICI value is increased for each object as well as for the object's leaf node and all its ancestors. ICI values are later used to update estimated lower bounds in the priority queue in the algorithm. In fact, mutual distances between data objects and queries are updated based on popularity. This procedure is captured in pseudo-code in Algorithm 1.

In the following, we propose two different procedures to apply ICI to the estimates of distances between a node and a query. General principle of such procedures is to create local attractive force to make accessed data parts closer to the query or repulsive force for unaccessed or distant data. In addition, we evaluate two ways of incrementing ICI in the experiments.

5.1 Naïve ICI

To modify priorities of individual nodes in algorithm's priority queue, we propose a naïve solution that mitigates influence of highly accesses data, but still respects the original distance:

$$log_{ICI} = log_{base}(ICI + base), \tag{1}$$

$$d_{ICI} = \frac{d_{orig}}{log_{ICI}}. \tag{2}$$

Algorithm 1. Algorithm for kNN query evaluation incorporating ICI.

Input: a query $Q = k\text{-NN}(q)$, an indexing structure hierarchy *root*
Output: List of objects satisfying the query $Q.res$

```
Q.res ← ∅ {init query result}
PQ ← {(root, 0)} {init priority queue with root and zero as the lower bound}
while PQ is not empty do
  e ← PQ.poll {get the first element from the priority queue}
  if Q.res[k].distance > e.lowerBound then
    break {terminate if e cannot contain objects closer than k^th neighbor}
  end if
  for all a ∈ e.getChildren() {check all child nodes} do
    if a.isLeaf() then
      update Q.res with a.objects
    else
      n.lowerBound ← get estimate of lower-bound on distance between a and Q
      {e.g. M-tree's original alg. uses (d(Q.q, a.pivot) − a.radius) here}
      n.distICI ← apply d_ICI on original distance between node's pivot and Q.q
      insert n into PQ
    end if
  end for
  sort PQ by distICI of each PQ's element
end while
for all o ∈ Q.res {increment ICI of object, its leaf node and all parents } do
  call incrICI on o {an integer stored at the object}
  call incrICI on o.getLeaf() and its parents {an integer stored at the node}
end for
return Q.res
```

To make the values of logarithm always positive, we add the value of *base* to ICI (which is zero for unaccessed data). It is also the only parameter of this method. Finally, the value of d_{ICI} is then used to sort the priority queue.

However, this procedure does not create the necessary attractive/repulsive forces with respect to distance. In particular, the shrinking factor applied on distance is constant for constant ICI. An example is given in Fig. 5.

5.2 Extended ICI

This procedure is inspired by the gravitation law and general dynamics of forces between physical objects. In this scenario, the value of ICI can be understood as a mass of an object/node, which determines an attraction force that pulls it to a query. The strength of it is straightforwardly updated with the power of distance. In naïve ICI, this force is constant regardless the distance to query. Extended ICI is defined as follows:

$$power_{ICI} = \frac{log_{ICI}}{(\frac{d_{orig}}{d_{max}})^{pwr} + 1}, \tag{3}$$

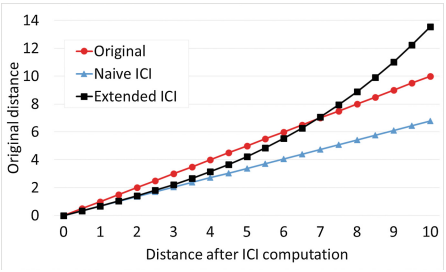


Fig. 5. Comparison of naïve and extended ICI = 20 for increasing original distance

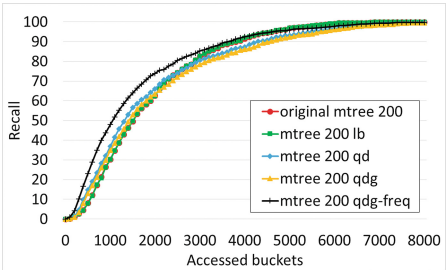


Fig. 6. Progress of recall for different strategies to order priority queue

$$d_{ICI} = \frac{d_{orig}}{power_{ICI}}, \tag{4}$$

where log_{ICI} is defined in Eq. 1 and d_{max} stands for the maximum distance in metric space (for CoPhIR dataset, it is 10).

This procedure introduces a new parameter *pwr*, which is subject to experimenting, but it brings necessary flexibility when different indexing structure is used. The behavior of Extended ICI is exampled in Fig. 5.

6 Experiments

We report on an extensive comparison of the proposed ICI techniques with a standard algorithm for precise kNN queries, i.e. no approximation was used.

The dataset used in experiments is a 1-million-object subset of CoPhIR dataset, where each object is formed by five MPEG-7 global descriptors (282 dimensional vector) and the distance function is a weighted sum of L_1 and L_2 metrics, for short. Please refer to [3] for complete description.

Since we focus on repeated queries, we used queries issued in the on-line image retrieval demo (see Sect. 4.1) during the year of 2009 and queries executed during January, 2010. The first set (Qy2009) contains 993 query objects and is used as the *learning set* to adapt ICI values. The second set (Qm1y2010) is the *testing set* to analyze the performance of metric indexing structures. In this set, there are 1000 query objects, where about 10 % queries appear in the learning set and the remaining 90 % queries are unique. All tests were performed for different settings and structures to evaluate precise 30NN queries:

- M-tree with capacities of leaf/non-leaf nodes set to 200/50, 400/100 and 2,000/100 objects;
- M-index built over 128 pivots and maximum tree depth of 8, node capacities set to 200 and 2,000 objects;
- naïve and extended ICI with different bases (5, 10) in log_{ICI} and exponents (2, 5, 10) in pwr_{ICI}.

Further statistics about the structures are given in Sect. 4.2.

6.1 Different Query Ordering Strategies

The first group of experiments focuses on determining the best setting of d_{ICI} distance measure. We used M-tree with leaf node capacity fixed to 200 only and the other parameters fixed to log base 10 and to power of 2. We studied the progress of recall at particular number of accessed nodes (buckets). The results are depicted in Fig. 6, where the following approaches where compared:

original – M-tree's algorithm for precise kNN evaluation (search queue ordered by lower-bound distance = $(d(q, pivot) - r_{covering})$;
lb – naïve ICI for $d_{orig} = d(q, pivot) - r_{covering}$;
qd – naïve ICI for $d_{orig} = d(q, pivot)$;
qdg – extended ICI for $d_{orig} = d(q, pivot)$, ICI updated for unique queries only;
qdg-freq – same as "qdg", but incrementing ICI for all queries (including repeated queries).

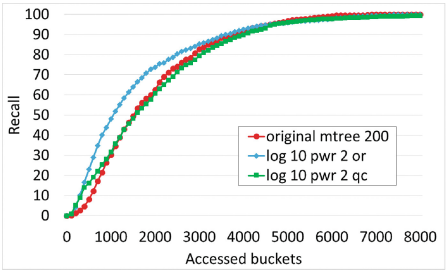
The results show that the concept of ICI is valid as the query recall rises faster. However, the original lower bound on distance must be replaced with the real distance between the query object and a pivot (node's representative). The best results are exhibited by the extended ICI strategy with values of ICI incremented for every query executed, i.e. including repeated queries. We will examine this strategy thoroughly in the following sections.

6.2 Influence of Indexing Structure Bushiness

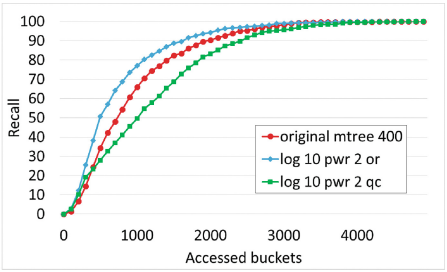
We focus on different leaf-node capacities of M-tree here. In particular, all three configurations (200, 400, and 2,000) are compared in Fig. 7. Results clearly show that the extended ICI with query frequency (blue curves in the figure) can outperform the original queue ordering regardless the number of leaf nodes. In addition, we have compared to variants of incrementing ICI values (lines $incrICI$ in Algorithm 1):

qc ICI value incremented by one in each node on the path from bucket to root; `incrICI(x):={x.ICI++}`
or each node's ICI value is increased by the normalized number of objects in the final query answer that were found in the node's subtree; the normalization is done by the cardinality of query answer, which is 30 in our scenario.
`incrICI(x):={x.ICI+=|subset(Q.res stored under x)| / |Q.res|}`

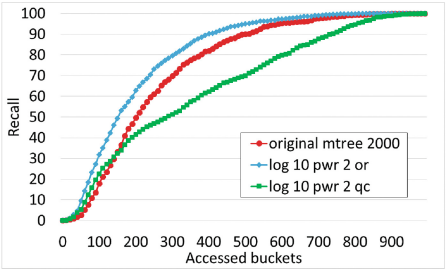
The variant *qc* apparently leads to very high values of ICI in nodes closer to the root node, which misleadingly attracts irrelevant nodes too near the query object. It has shown as ineffective in overall progress of recall. The variant *or* has a good property of having the sum of ICI values over all nodes on the same level equal to the number of processed queries, so we use it in all experiments if not stated otherwise.



(a) M-tree 200

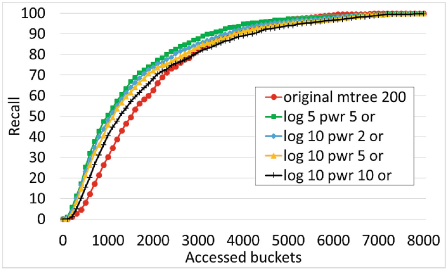


(b) M-tree 400

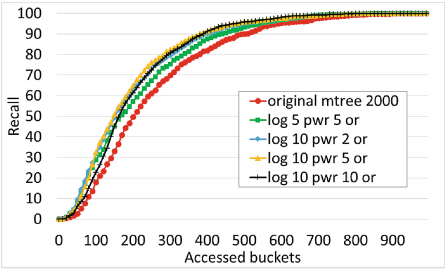


(c) M-tree 2,000

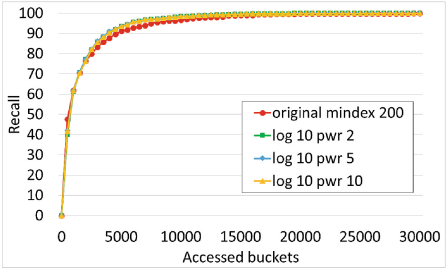
Fig. 7. Progress of recall for different M-tree configurations (qdg-freq) (Color figure online)



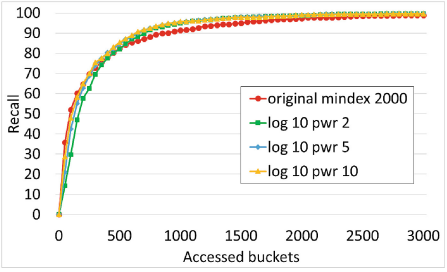
(a) M-tree 200



(b) M-tree 2,000



(c) M-index 200



(d) M-index 2,000

Fig. 8. Progress of recall while varying parameters of extended ICI (qdg-freq)

6.3 Varying Parameters of Extended ICI

The last group of experiments examines the parameter of extended ICI, namely the base of logarithm and the exponent of power. In Fig. 8, the progress of recall is presented for both M-tree and M-index with leaf node capacities 200 and 2,000 objects. From the large number of combinations of log base and exponent, we selected 5/5, 10/2, 10/5 and 10/10 only, because such settings were able to exceed the performance of original kNN algorithm. As for M-tree, the results quite clearly support the configurations 5/5 and 10/2 for 200 and 2,000 bucket capacities, respectively. The results for M-index look very similar to the original kNN algorithm in the figure. But we can still see higher efficiency for higher values of recall. In particular, starting from 80 % recall, the extended ICI queue ordering can access promising buckets earlier. Here, the best configuration is 10/5.

Table 2 presents details on the number of accessed buckets needed to obtain 50 % and 95 % recall of 30NN queries. It can be seen that the best results are dependent on the indexing structure setup (bucket capacity), which is mainly evident from the data concerning M-tree. High performance of original M-index's algorithm in early stages of query processing causes performance declination for 50 % recall. However, the improvement is eminent while considering higher values of recall, which calls for applying our method to approximate kNN evaluation. From the data, we can generally state that better results are obtained for M-index than for M-tree. It is also noticeable that greater bucket sizes increases the improvement achieved by ICI.

To sum up all the experiments, the concept of reordering priority queue with respect to previous usefulness of data partitions proved as valid. Since disk-oriented indexing structures prefer larger bucket capacities, the extended ICI with log base of 10 and exponent in power of 5 is a good and universal choice.

Table 2. Improvement in query costs for 50 % and 95 % recall.

Setup information		50 % query completion			95 % query completion		
Indexing structure	Best setup (log-pwr)	Original nodes needed	Nodes needed	Total improvement	Original nodes needed	Nodes needed	Total improvement
M-tree 200	5-5	1600	1000	37,5 %	4600	4200	8,7 %
M-tree 2000	10-2	210	160	23.8 %	590	470	20,5 %
M-index 200	10-5	600	800	−33 %	8000	6000	25 %
M-index 2000	10-5	100	130	−30 %	1500	950	37 %

7 Conclusion and Future Work

We have presented a new approach to query answering optimization in metric spaces called *Inverted Cache Index* (ICI). Previous accesses to data partitions

are recorded and their participation on query answering is later used to give search preference to such partitions. However, it is not blindly applied, but rather the distance values in metric space are reflected to create proper attractive or repulsive forces correspondingly.

Application of ICI presents multidimensional complexity as it is needed to analyze behavior on different datasets, different indexing structures, and different parameters of extended ICI formula. We have shown that more than 35 % improvement is achieved to obtain 95 % recall for a state-of-the-art indexing structure – M-index. We consider this to be the greatest contribution here.

Since the whole concept is applicable to any hierarchical organization, we plan to investigate it further. Additionally, ICI's optimization of approximate query evaluation is straightforward and we will investigate it in the future. Another issue to study is to vary the amount of historical bucket-access recordings to take into consideration. Its implementation is easy, but new findings may be obtained. The ultimate goal would be a definition of procedure that could automatically swap search queue ordering between ICI and the original priority depending on current data distribution.

Acknowledgements. This work was supported by Czech Science Foundation project GA16-18889S.

References

1. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. ACM Trans. Inf. Syst. (TOIS) **21**(2), 192–227 (2003)
2. Barrios, J.M., Bustos, B., Skopal, T.: Analyzing and dynamically indexing the query set. Inf. Syst. **45**, 37–47 (2014)
3. Batko, M., Falchi, F., Lucchese, C., Novak, D., Perego, R., Rabitti, F., Sedmidubsky, J., Zezula, P.: Building a web-scale image similarity search system. Multimedia Tools Appl. **47**(3), 599–629 (2009)
4. Beecks, C., Uysal, M.S., Driessen, P., Seidl, T.: Content-based exploration of multimedia databases. In: Proceedings of the 11th International Workshop on Content-Based Multimedia Indexing (CBMI), pp. 59–64. IEEE, June 2013
5. Beecks, C., Skopal, T., Schöffmann, K., Seidl, T.: Towards large-scale multimedia exploration. In: Proceedings of the 5th International Workshop on Ranking in Databases (DBRank), Seattle, WA, USA, pp. 31–33. VLDB Endowment (2011)
6. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3), 322–373 (2001)
7. Chávez, E., Marroquín, J.L., Navarro, G.: Overcoming the curse of dimensionality. In: Proceedings of the European Workshop on Content-Based Multimedia Indexing (CBMI), Toulouse, France, 25–27 October 1999, pp. 57–64 (1999)
8. Chávez, E., Navarro, G., Baeza-Yates, R.A., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. (CSUR) **33**(3), 273–321 (2001)

9. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., Loucopoulos, P., Jeusfeld, M.A. (eds.) Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), Athens, Greece, 25–29 August 1997, pp. 426–435. Morgan Kaufmann (1997)
10. Deepak, P., Prasad, M.D.: Operators for Similarity Search: Semantics, Techniques and Usage Scenarios. Springer, Heidelberg (2015)
11. Esuli, A.: Use of permutation prefixes for efficient and scalable approximate similarity search. Inf. Process. Manage. **48**(5), 889–902 (2012)
12. Houle, M.E., Nett, M.: Rank-based similarity search: reducing the dimensional dependence. IEEE Trans. Pattern Anal. Mach. Intell. **37**(1), 136–150 (2015)
13. Houle, M.E., Sakuma, J.: Fast approximate similarity search in extremely high-dimensional data sets. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 619–630, April 2005
14. Lew, M.S., Sebe, N., Djeraba, C., Jain, R.: Content-based multimedia information retrieval: state of the art and challenges. ACM Trans. Multimedia Comput. Commun. Appl. **2**(1), 1–19 (2006)
15. Moško, J., Lokoč, J., Grošup, T., Čech, P., Skopal, T., Lánský, J.: MLES: multilayer exploration structure for multimedia exploration. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds.) New Trends in Databases and Information Systems. Communications in Computer and Information Science, vol. 539, pp. 135–144. Springer, Switzerland (2015)
16. Novak, D., Batko, M., Zezula, P.: Generic similarity search engine demonstrated by an image retrieval application. In: Proceedings of the 32nd International ACM Conference on Research and Development in Information Retrieval (SIGIR), Boston, MA, USA, p. 840. ACM (2009)
17. Novak, D., Batko, M., Zezula, P.: Metric index: an efficient and scalable solution for precise and approximate similarity search. Inf. Syst. **36**, 721–733 (2011)
18. Oliveira, P.H., Traina Jr., C., Kaster, D.S.: Improving the pruning ability of dynamic metric access methods with local additional pivots and anticipation of information. In: Morzy, T., Valduriez, P., Ladjel, B. (eds.) ADBIS 2015. LNCS, vol. 9282, pp. 18–31. Springer, Heidelberg (2015)
19. Samet, H.: Foundations of Multidimensional And Metric Data Structures. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco (2006)
20. Skopal, T., Lokoc, J., Bustos, B.: D-cache: universal distance cache for metric access methods. IEEE Trans. Knowl. Data Eng. **24**(5), 868–881 (2012)
21. Skopal, T., Hoksza, D.: Improving the performance of M-Tree family by nearest-neighbor graphs. In: Ioannidis, Y., Novikov, B., Rachev, B. (eds.) ADBIS 2007. LNCS, vol. 4690, pp. 172–188. Springer, Heidelberg (2007)
22. Skopal, T., Pokorný, J., Snášel, V.: Nearest neighbours search using the PM-Tree. In: Zhou, L., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 803–815. Springer, Heidelberg (2005)
23. Vilar, J.M.: Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm. Inf. Process. Lett. **56**(5), 265–271 (1995)
24. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, vol. 32. Springer, New York (2005)