Towards Faster Similarity Search by Dynamic Reordering of Streamed Queries

Filip Nalepa(✉), Michal Batko, and Pavel Zezula

Faculty of Informatics, Masaryk University, Brno, Czech Republic
f.nalepa@gmail.com

Abstract. Current era of digital data explosion calls for employment of content-based similarity search techniques, since traditional searchable metadata like annotations are not always available. In our work, we focus on a scenario where the similarity search is used in the context of stream processing, which is one of the suitable approaches to deal with huge amounts of data. Our goal is to maximize the throughput of processed queries while a slight delay is acceptable. We propose a technique that dynamically reorders the queries coming from the stream in order to use our caching mechanism in huge data spaces more effectively. We were able to achieve significantly higher throughput compared to the baseline when no reordering and no caching were used. Moreover, our proposal does not incur any additional precision loss of the similarity search, as opposed to some other caching techniques. In addition to the throughput maximization, we also study the potential of trading off the throughput for low delays (waiting times). The proposed technique allows to be parameterized by the amount of the throughput that can be sacrificed.

Keywords: Stream processing · Similarity search

1 Introduction

Huge amounts of unstructured data are being produced nowadays resulting from the current digital media explosion. Many tasks targeting at processing such data involve, in some form, searching in the data. Traditional search techniques based on exact match of data attributes cannot be often applied to such data types. Instead, content-based search that treats the data by similarity can be an appropriate option. Such search then usually uses *k-nearest-neighbors (kNN) queries*, which retrieve the k objects that are the most similar to a given query object. The level of similarity is measured by a metric distance function.

Due to the nature of the data and applications that use them, it can be desired to deal with the data as with a potentially infinite stream that is continuously being created. For example, consider a text search-engine crawler that

A. Hameurlain et al. (Eds.): TLDKS XXXVIII, LNCS 11250, pp. 61–88, 2018.
https://doi.org/10.1007/978-3-662-58384-5_3

gathers images from the web and needs to continuously annotate them by textual descriptions according to the image content. As another example, a news notification system needs to compare the newly published articles to the profiles of all the subscribed users to find out who should be notified.

A subtask of these applications is processing the streamed data items by some form of content-based searching. An important characteristic of the applications is that the data do not need to be processed immediately as in interactive applications, but some delay is acceptable. The performance of these applications is mostly determined by the number of processed data items in a given time interval; that is, the throughput is the most important property. The individual query search times can be improved by applying some similarity indexing techniques that have efficient algorithms based on the metric model of similarity [20]. As opposed to interactive applications focusing on the single query optimization, in our scenario, we can afford to postpone processing of some queries if the overall throughput of the system is improved.

I/O costs typically have a significant effect on the performance of similarity search techniques. In our work, we exploit the fact that if a sequence of queries is processed in an appropriate order, we can achieve considerably lower I/O costs and overall processing times than if the order of the queries is random. This is possible if two similar queries need to access similar data of the search index, which is a common property of the indexes. By obtaining an appropriate ordering of queries, the accessed data can be cached in the main memory and reused for evaluation of similar queries lowering down the I/O costs.

The first contribution of the paper is a technique to dynamically reorder the incoming queries, which allows to achieve a significant improvement of the throughput according to our experiments. One of the features of the approach is that it does not influence the quality of query results as other approximation techniques do.

In the paper, we also present a way to trade off the throughput for the delays of the processed queries. In other words, we balance the number of queries processed per a time unit and the waiting times of individual queries. We show how the proposed technique for the throughput maximization can be modified to increase the number of queries processed until a given time limit while maintaining sufficient throughput to be able to keep up with the rate of incoming new query objects.

The presented approach is built upon our previous work [14]. In this paper, we present more effective reordering technique, which allows to achieve even higher throughput. We add formalization of the approach to the query reordering as a problem of traversing a graph. The throughput-delay trade-off is another new contribution of this paper.

The proposed technique is implemented as an extension of the M-Index [15] used for indexing metric-space data, and its performance is compared to the basic version of the M-Index. We used the Profimedia dataset of images [6] represented as high-dimensional vectors (4,096 and 256 dimensions) and measured the throughput and other throughput-related properties of kNN queries. We

were able to significantly improve the throughput compared to the basic version of the M-Index. We also experimented with balancing the throughput and the delays of the processed queries.

The rest of the paper is organized as follows. First, we formally define our problem in Sect. 2. In Sect. 3, we present related work on caching and query reordering in similarity search. A deep look into the proposed technique is presented in Sect. 4. It explains details of the caching system and the principles of dynamic query reordering based on underlying graph model. The experimental evaluation of our approach can be found in Sect. 5. We show modifications of the proposed technique to tradeoff throughput for delays in Sect. 6. Our results are summarized in Sect. 7.

2 Problem Definition and Objectives

Suppose there is a set of complex objects D (e.g., images represented as high-dimensional vectors) and a large database $X \subseteq D$ ($|X| \geq 10^6$). Let $s = (d_1, d_2, \ldots)$ be a stream, that is, a potentially infinite sequence of data items. Each item of the stream is a pair $d_i = (q_i, t_i)$ where $q_i \in D$ is a query object and t_i is the time when it was created (became available). We suppose the data items of the stream are ordered from the oldest ones; that is, it holds that $t_i \leq t_{i+1}$ for each i and $t_1 = 0$.

There is a defined metric space (D, md), which is a universal model of similarity [20]. md is a total distance function $md : D \times D \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. The distance function satisfies these postulates for all $o, p, q \in D$:

- $md(o, p) \geq 0$ (non-negativity),
- $md(o, p) = 0 \iff o = p$ (identity),
- $md(o, p) = md(p, o)$ (symmetry),
- $md(o, q) \leq md(o, p) + md(p, q)$ (triangle inequality).

The distance between any two objects from D corresponds to the level of their dissimilarity; the higher the distance, the higher the dissimilarity.

For each query object q_i in the stream s, a k-nearest-neighbors query $NN(q_i, k)$ is executed, which returns the k nearest objects from the database X to the query object q_i according to the distance function md. We consider the scenario when X is stored on a disk and a subset of its data needs to be accessed in order to evaluate a query. We also suppose I/O operations constitute a significant cost considering all the needed data have to be read from the disk during the query evaluation. Specifically, we target the situations when the time to evaluate a query by this approach is higher than the average time gap between two subsequent data items in the stream.

It is allowed to change the order of the processed query objects. More precisely, at the time t, any query object q_i, where (q_i, t_i) is a data item of the stream and $t_i \leq t$, can be processed.

The goal is to process the query objects of the stream so that the given criteria are optimized. There can be various criteria that can be a subject to

optimization depending on a specific application. The criterion we focus on in this paper is throughput maximization. So the goal is to maximize the number of processed query objects of a given stream until a given time T. Alternatively, the criterion can be defined as the minimization of the number of unprocessed query objects at the time T, that is, the number of (q_i, t_i) from the stream s where $t_i \leq T$ and q_i is not processed.

We propose a technique that can be used to improve the throughput of similarity search processing. The technique is based on reordering of the query objects in the stream combined with a caching of the data that were accessed during the evaluation of previous queries.

3 Related Work

The usage of a caching mechanism in similarity search has been proposed in several papers to reduce the time spent by I/O operations. Unlike traditional caching, which is based on exact matches only (e.g., the one exploited by web search engines [9]), the similarity caching also has to manage similar matches.

Existing caching techniques used to speed up processing of a stream of similarity search queries assume that the queries are appropriately ordered. In particular, they assume that similar queries are placed nearby in the stream. This ensures that the cached values can be actually used before they are overwritten by different queries. However, such a characteristic applies to specific scenarios only, e.g., when there exist some popular objects that are frequently searched. In our approach, we do not consider any specific ordering of queries within a stream. Instead, we reorder the queries so that we obtain the sequences that are desired.

In [10], the authors deal with kNN queries to search for similar images in a metric space. They build their approach on the assumption that there exists a set of popular images, which are queried by users significantly more often than the other images. They propose an approach where the result sets of individual kNN queries are stored in a cache and they are reused to produce approximate results of subsequent queries.

The concept of caching in similarity search is also used in [16], where it is applied to contextual advertising systems. If there is a cache miss for a kNN query q, then a larger set of objects than are actually needed is retrieved from the disk and stored in the cache. When a similar query to the cached query q comes to the system, the cached values of q are explored to obtain results for the new query. In this way, an approximate answer is returned.

Static/Dynamic caching is presented in [19]. The cache consists of two parts. The static part stores queries (along with their results) that remain popular over time. The dynamic cache keeps queries that are popular for a short period of time. A combination of both is used to speedup the evaluation of queries. Several strategies are proposed to select the suitable queries to be stored in the cache based on analysis of past queries.

The paper [5] presents an index structure that serves as a cache and as an index at the same time. The index is built and reorganized dynamically as new

queries are evaluated. The advantage of the proposed approach is that if the cache is not able to provide an answer, the data computed up to that moment are used by the index.

Caching of data partitions loaded from a disk during query evaluation is presented in [8] complemented with caching previous answers, which serves to set initial search radius for similar kNN queries.

The authors of the paper [18] target the situations when the distance computation itself is an expensive operation. They propose D-cache, which stores distances computed during previous queries to spare some distance computations of subsequent queries. The Snake Table presented in [2] uses a cache of distances to avoid some distance computations when processing streams of queries with snake distribution (i.e., consecutive query objects are similar). Since the Snake Table needs a space proportional to the size of the dataset, it is suitable for medium-sized databases.

Another way to improve the throughput of a stream of kNN queries, is to reorder the queries. In [17], the authors optimize nearest neighbor search for videos when each video is represented by a sequence of high-dimensional vectors. Given a query video containing n vectors, a search for each vector is performed and the overall similarity is computed at last. The authors make use of the fact that vectors of subsequent video frames are similar with respect to the Euclidean distance and they propose dynamic query ordering for advanced optimization of both I/O and CPU costs. They make an observation that the candidates of a previous query may help to further reduce the candidates shared with a subsequent query. The algorithm aims at progressively finding a query order such that the common candidates among queries are fully utilized to maximally reduce the total number of candidates. Also, this approach builds on the assumption that the subsequent kNN queries in the stream are similar to each other. Moreover, the ordering technique is designed for low-dimensional vector spaces (tested on 32 dimensions) and for a short sequence of queries (tested on sequences of the length of 60 queries). In our work, we target more complex metric spaces (e.g., high-dimensional vector spaces with thousands of dimensions), in which case it is highly improbable that similar queries can be found in such short sequences.

4 Principles of the Proposed Approach

To speed up a similarity query evaluation, a metric index is often used. In our approach, we consider a generic metric index that uses data partitioning $P = \{p_1, \ldots, p_n\}$ where $p_i \subseteq X$, $p_i \cap p_j = \emptyset$ for any two partitions p_i, p_j, and $\bigcup_{i=1}^{n} p_i = X$. When processing a query object q, a subset of the partitions $I(q) \subseteq P$ needs to be accessed. The partitions are typically stored on a disk [20]. A frequent bottleneck of similarity search techniques is the reading of the partitions from the disk during a query evaluation. Our solution aims to decrease the number of disk accesses, which consequently decreases the time to process the queries.

The aim of data partitioning methods is to generate the partitions in such a way that any two objects p, q of a partition are similar to each other; that is,

$md(p, q)$ is small. When a query is to be evaluated, a score is assigned to each partition according to estimated distances of the query to the objects of the partition. Based on the scores, only the objects of the most promising partitions are directly examined. The definition of the score is dependent on a specific metric index.

Let $I(q) \subseteq P$ be the set of data partitions accessed during the evaluation of the query object q. Taking into account the properties of data partitioning methods, the following holds:

$$md(q_1, q_2) \leq \epsilon \implies |I(q_1) \ominus I(q_2)| \leq \delta \tag{1}$$

where ϵ is a small non-negative real number; δ is a small non-negative integer. That means if two query objects are very similar to each other (their distance is at most ϵ), the sets of accessed data partitions are also very similar (the number of elements in their symmetric difference is at most δ). This property can be used to speed up the processing of query objects q_1 and q_2. First, q_1 is evaluated and the accessed data partitions are kept in the main memory cache. When q_2 is being evaluated, the data partitions stored in the cache can be reused to avoid expensive disk accesses.

However, the caching itself is not typically enough for the speedup. Suppose there is a database of millions of objects indexed in a metric space. In practice, the metric space is often defined as a vector space of a high number of dimensions. Due to such a huge search space, there is very low probability that two subsequent query objects in the stream are similar enough to access overlapping sets of data partitions during their evaluation. For the cache to be sufficiently utilized, the query objects in the stream need to be reordered so that sequences of similar query objects are obtained.

To sum it up, our approach consists of two parts. The first one is the in-memory caching of recently loaded data partitions and reusing them for evaluation of subsequent queries. The second one is the query object reordering allowing to process sequences of similar query objects to maximize the cache utilization.

In Sect. 4.1, we describe the architecture of the system used for processing a stream of query objects. Details of the caching system are presented in Sect. 4.2. In Sect. 4.3, we model the problem of query object ordering from the perspective of graphs and we define a query graph with query objects as the vertices. Section 4.4 discusses ways to construct the query graph; we present principals of traversing the graph in order to maximize the throughput in Sect. 4.3; a specific algorithm is provided in Sect. 4.6.

4.1 Architecture

In this section, we describe the architecture of the whole system. The schema of the architecture is depicted in Fig. 1.

Let us have a stream $((q_1, t_1), (q_2, t_2), \ldots)$. A query object q_i arrives at the application at the time t_i and it is inserted into a component called a buffer. The buffer

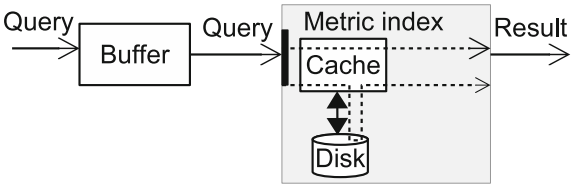


Fig. 1. Architecture

is used to temporarily store the incoming query objects that are waiting for processing. This is the component where the query reordering takes place.

Another part of the architecture is the metric index, which takes care of the query object evaluation. It contains a disk where the database of objects is stored and a main memory cache is used to store the recently loaded data partitions from the disk.

When the metric index is ready for processing another query, a query object is picked from the buffer according to a strategy described in following parts of this paper. During processing of the query, the metric index looks into the cache to possibly use any data partitions obtained while evaluating recent queries. If the data are not in the cache, they are loaded from the disk.

4.2 Cache

The cache is defined as a set of partitions $cache = \{p_1, \ldots, p_m\} \subseteq P$. The size of the cache is limited by the number of objects within the cached partitions: $\sum_{p \in cache} |p| \leq cacheLimit$.

To measure the utility of the cache during evaluation of a given query object q, we define the following function.

$$cacheUtility(q, cache) = \frac{|I(q) \cap cache|}{|I(q)|} \tag{2}$$

where *cache* represents the content of the cache and $I(q) \subseteq P$ is the set of partitions accessed during the evaluation of q. We suppose $I(q)$ is a non-empty set.

To keep track of the content of the cache, we define the function $updateCache(q, cache)$ returning the content of the cache after processing the query object q where *cache* represents the content of the cache before executing q. In our implementation, we use the *least recently used* policy [12]. In particular, the partitions with the oldest last access time are discarded and replaced with the new partitions of the last query while obeying the *cacheLimit*.

The $queryTime(q, cacheUtility)$ represents the time to process the given query object q using the given cache utility. The desired property of the function is that the time should be decreasing with increasing cache utility due to a lower I/O cost.

$$cu_1 \leq cu_2 \iff queryTime(q, cu_1) \geq queryTime(q, cu_2) \tag{3}$$

where cu_1, cu_2 are cache utilities.

4.3 Query Ordering

As it was stated before, a key to a high cache utilization is to process the query objects in an appropriate order.

The problem of query object ordering can be modeled from the perspective of graphs. Let $s = ((q_1, t_1), (q_2, t_2), \ldots)$ be the stream to be processed. Let us consider just the portion of the stream available at the time t: $((q_1, t_1), \ldots, (q_k, t_k))$ so that $t_k \leq t$ and $t_{k+1} > t$. We define the undirected *query graph* $G_t = (V, E)$ at the time t in the following way. The set of vertices is comprised of the subsequence items $V = \{(q_1, t_1), \ldots, (q_k, t_k))\}$. In other words, each query object of the stream subsequence represents a vertex in the query graph.

The graph is complete; that is, there is an edge between every pair of vertices (q_i, t_i) and (q_j, t_j) where $i \neq j$. A value is associated with each edge denoting an upper bound of the query time to process q_i right after q_j or q_j right after q_i. We denote the value assigned to the edge (referred to as the *edge value* in the rest of the paper) between the vertices (q_i, t_i) and (q_j, t_j) as $e((q_i, t_i), (q_j, t_j))$.

The edge value between the vertices (q_i, t_i) and (q_j, t_j) is determined as follows:

$$e((q_i, t_i), (q_j, t_j)) = max\{queryTime(q_i, cu_i), queryTime(q_j, cu_j)\} \quad (4)$$

where $cu_i = cacheUtility(q_i, I(q_j))$ and $cu_j = cacheUtility(q_j, I(q_i))$. That is, the upper bound of the query time is the maximum of the query times when q_i is processed right after q_j or q_j right after q_i, while the cache contains only the partitions needed by one previous query. If the cache contains more partitions rather than those needed by the previous query, the cache utility does not decrease and therefore, the query time does not increase according to Formula 3.

The query graph G_t is defined at each time t where t is a non-negative integer. The graph continuously grows by adding new vertices and edges as new query objects become available in the stream. G_t is a subgraph of G_k where $k > t$; $G_{t+1} = G_t$ if and only if there does not exist an item $(q_i, t + 1)$ in the stream; that is, no new item becomes available at the time $t + 1$. An example of the graph evolution can be seen in Fig. 2.

Our objective is to find such an ordering in which the query objects are processed so that the throughput is maximized. Regarding the query graph, we want to find an acyclic path (i.e., a sequence of the vertices) that determines

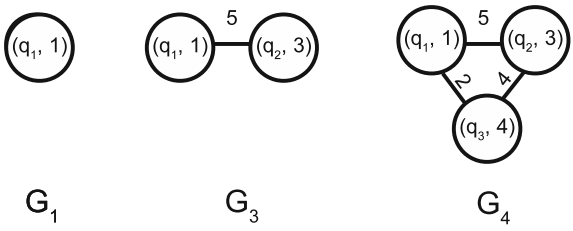


Fig. 2. Query graph evolution example

the ordering of the query objects. Formally, given the time limit T, the task is to find the longest path $((q_{i_1}, t_{i_1}), \ldots, (q_{i_k}, t_{i_k}))$ in G_T so that $start_k < T$ where $start_k$ is the time when the last query object q_{i_k} starts to be evaluated. The length of the path is measured as the number of vertices, that is, the number of processed query objects.

A query object is processed as soon as the preceding query object is processed or as soon as it becomes available in the stream, whichever occurs later. This strategy is applied in order to improve the throughput.

Algorithm 1 describes the generic processing of a stream of query objects. The input is the stream of query objects arriving continuously to the application and the time limit T. The algorithm repeatedly calls the *addNewQueryObjects* function to add newly arrived query objects to the query graph and the *getQueryToProcess* function that returns a query object that is to be processed next according to the query graph and according to the path of query objects generated so far. The returned query object is processed and added to the path. The loop finishes when the time limit T is exceeded.

Algorithm 1. Generic algorithm

Input: stream of query objects $stream = (q_1, q_2, \ldots)$ and the time limit T
Output: *path*: the order in which the stream objects are processed

```
function processStream(stream, T)
    path ← ()
    G ← empty graph
    while T ≤ now do
        G.addNewQueryObjects(stream)
        q ← getQueryToProcess(G, path)
        if q ≠ null then
            process(q)
            path.add(q)
    return path
```

4.4 Query Graph Construction

To process the stream in the described manner, the query graphs have to be generated from the raw stream of query objects. In particular, the edge values have to be assigned. According to the definition, they should denote an upper bound of the time to process a query object at one side of the edge right after processing the query object on the other side. The question is how to predict the query times. Usually, it is not possible to obtain the precise query times in advance. In practice, we are likely to end up with a method for computing the expected times to execute the queries.

The query time greatly depends on the cache utility (see Formula 3), that is, on the number of data partitions loaded from the disk during query evaluation. The problem is that it is not typically possible to precisely determine the set

of needed data partitions without the actual query evaluation. The underlying metric index usually works with a priority queue of data partitions, which is being updated dynamically as the individual data partitions are examined [20]. Therefore, the approach based on precise computation of the cache utility is not usable in practice.

Data partitions that are needed during the processing of a query are generally determined by their distance to the query object in the metric space. So the size of the intersection of data partitions needed during processing of two query objects is influenced by the metric distance of the query objects. Pairs of close query objects are assumed to share more data partitions than pairs of distant ones.

Therefore, a straightforward way to approximate edge values of the query graph can be based on the metric distances between query objects. When a new query object arrives, the distances to all the other query objects of the graph are computed. From the practical point of view, it is actually not necessary to assign the edge values to already processed query objects (except for the last one) because they are not used to generate the path (the details are described later). Despite this practical consequence, the edge value assignment can be very time consuming if there are a lot of query objects (e.g., tens of thousands as in our experiments) that are still waiting for their processing. This is because the metric distance computation can be a computationally intensive operation. If the computational complexity of adding a new query object to the graph is measured by the number of metric distance computations, the complexity is linear with respect to the number of unprocessed query objects already in the graph.

Instead, we use a pivot based technique to estimate the metric distances and the query times [7]. In particular, let there be a fixed set of objects in the metric space; we will denote them as pivots. When a new query object q is to be added to the graph, distances between the query object q and all the pivots are computed. The pivots are ordered from the nearest to the farthest one, which defines a permutation of the pivots. This pivot permutation is stored for each query object. The edge value between two query objects is determined according to the length of the common prefix of their pivot permutations, the longer the common prefix, the lower the edge value. The length of the common prefix of two permutations $(p_{i_1}, p_{i_2}, \ldots, p_{i_m})$ and $(p_{j_1}, p_{j_2}, \ldots, p_{j_m})$ is determined as the maximum number k $(\leq m)$ such that $i_r = j_r$ for all $1 \leq r \leq k$. The pivot based technique was also shown to work well in high-dimensional vector spaces [15], which is essential for our scenario.

We can either stay with such relative edge values, or absolute edge values can be assigned based on empirical measurements. Specifically, we can construct a function that for a length of a common prefix of two query objects assigns an average query time $estimatedQueryTime(prefLength) = qt_{pl}$. Since there is a fixed set of pivots, the cost to insert a new query object in the query graph is constant. As we consider the scenarios when the rate of incoming query objects is high, the number of pivots is typically much lower than the number of query objects waiting for their processing.

Many strategies for pivot selection have been proposed [1]. According to the comparison of pivot selection techniques for permutation-based indexing provided in [1], there is no universally best pivot selection technique, but rather different techniques are optimal for different purposes. The authors also state that "the random chosen pivots is never a bad idea even if it is also never the smartest decision". In our experiments, we use the M-Index [15] to evaluate the similarity search queries, which is also a permutation-based access method. From the efficiency point of view, it is beneficial to use the same set of pivots for building the query graph and for indexing the dataset, since it helps to save some distance computations. The authors of the M-Index claim that the random pivot selection results in similar performance of query evaluations as other selection methods. Due to these statements, we use a random selection of the pivots to build the query graph.

4.5 Query Path Search

As we are now able to efficiently construct the query graphs, let us focus on the throughput maximization criterion. The goal is to generate an appropriate path in the graph. The path is generated whenever a next query object can be processed. The query object that is to be processed needs to be chosen very efficiently so that it does not impose much overhead. To be able to do that, we apply a greedy approach trying to find a path with minimal average edge values.

Let us first define the term of the density of a subgraph $SG = (V_S, E_S)$, where $|V_S| > 1$: Let $(v_{i_1}, v_{i_2}, \ldots, v_{i_{|V_S|}})$ be the shortest path going through all the vertices V_S; $v_{i_j} \in V_S$ for $1 \leq j \leq |V_S|$ where the length of the path is computed as the sum of the edge values between the subsequent vertices in the path. Then $density(SG) = \frac{|V_S|}{\sum_{j=1}^{|V_S|-1} e(v_{i_j}, v_{i_{j+1}})}$; that is, the density is determined by the average edge value in the corresponding shortest path.

The proposed greedy approach relies on finding subgraphs of high density. Such a dense subgraph is determined by the existence of a path through all the vertices of the subgraph with a low average edge value. Since new vertices are continuously added to the graph, the density of the subgraphs changes. The search for dense subgraphs is intended to identify the parts of the query graph that are at the current time possible to be processed with high cache utility. The dense subgraph strategy is combined with the nearest-neighbor strategy [4], which is a simple heuristic technique for the traveling salesman problem: Start at an arbitrary vertex. The next vertex to visit is the one that has the lowest-value edge to the current vertex among the unvisited vertices. This step is repeated until all the vertices of the given subgraph are visited. In summary, the greedy approach repeatedly finds a dense subgraph and processes all the query objects in the subgraph in the nearest-neighbor manner.

This gets us to another problem to solve: how to efficiently identify dense subgraphs. First, we construct a hierarchical clustering of the vertices. Dense subgraphs are then found by exploring individual clusters rather than the whole graph.

Let $G_t = (V, E)$ be the query graph at the time t. The set of clusters $C_t = \{c_{L_1}, c_{L_2}, \ldots, c_{L_k}\}$ is a decomposition of the set of vertices V so that $L_i = max\{e(v_a, v_b) \mid v_a, v_b \in c_{L_i}\}$ for $1 \leq i \leq k$. It means that all the vertices are decomposed into disjoint groups (clusters). Each cluster c_{L_i} defines an upper limit L_i on the edge value between any two vertices, that is, a query time limit. Note that there can be multiple valid decompositions given a set of clusters with their distance limits. This forms the lowest level of the hierarchical clustering. Another level of clusters is constructed by a decomposition of the set of clusters on a lower level and by assigning query time limits to the new clusters so that the limits are obeyed. Eventually, the *tree of clusters* T_t is generated. The vertices of the original query graph (i.e., the query objects) are added as leaf nodes to the corresponding bottom clusters.

For the decomposition of a set of vertices and subsequently clusters, we reuse the pivot based technique that is applied for the query graph construction in Sect. 4.4. Each internal node on the level k of the tree is of the form $n = ((p_{i_1}, \ldots, p_{i_{k-1}}), L_n)$ where $(p_{i_1}, \ldots, p_{i_{k-1}})$ is the pivot permutation prefix shared by all the descending nodes; L_n is the upper limit on the edge value between any two descending leaf nodes. Each leaf node is of the form (q, t), where q denotes the query object; t denotes its time of arrival.

An example can be seen in Fig. 3. Suppose there is the stream $((q_1, 1), (q_2, 3), (q_3, 4), (q_4, 7), (q_5, 11), \ldots)$. Let p_1, p_2, p_3 be the pivots. Let the pivot permutations for the first four items of the stream be (p_1, p_2, p_3), (p_3, p_1, p_2), (p_1, p_3, p_2) and (p_1, p_2, p_3) in the respective order. Let the estimated query times for individual common prefix lengths be 8, 2, 1 for the prefix lengths 0, 1, 2, respectively (see the function *estimatedQueryTime* in Sect. 4.4). On the left side of the figure, there is the query graph G_7 with the query times on the edges. On the right side, there is the generated tree. Individual levels of the tree correspond to the length of the common pivot permutation prefix of the descending leaf nodes. The upper bound of the query time of q_i processed right after q_j is determined by the query time limit of their lowest common parent node. For example, the maximum query time limit of q_3 when processed right after q_4 is determined by the lowest common parent $((p_1), 2)$.

Contrary to the generic Algorithm 1, we actually work with the tree of clusters in the implementation of the proposed approach. However, the structure of the algorithm does not change; that is, there is a loop in which new query objects are added to the tree and a next-to-process query object is selected and processed.

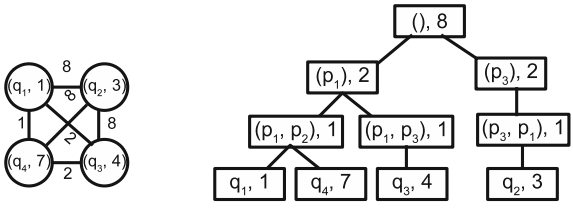


Fig. 3. Query graph and corresponding tree of clusters

A pseudo code of the function inserting a new query object into the tree of clusters is shown in Algorithm 2, which applies the pivot based technique. In the next section, we will describe details of the function that traverses the tree and selects a next query object to be processed.

Algorithm 2. Query object insertion algorithm

Input: tree of clusters *tree*, a query object q to insert and a set of pivots P
  function addNewQueryObject(*tree*, q)
    $pp \leftarrow computePivotPermutation(q, P)$
    $tree.addQueryObject(q, pp)$

4.6 Tree Traversal

Let us describe how the tree of clusters T_t is traversed in accordance with the dense subgraph and nearest-neighbor heuristics. The depth-first search is applied. Suppose q is the last processed query object so far. First, we look for a set of near neighbors by finding the lowest nonempty parent p of q. A node is considered nonempty if and only if it has descending unprocessed leaf nodes (query objects). After that, a child of p is selected based on some given strategy. Recursively, a grandchild and other descendants are selected until a leaf is reached and processed. This is captured in the pseudo code in Algorithm 3.

Algorithm 3. Tree traversal algorithm

Input: tree of clusters *tree* and path *path* representing the order of already processed query objects
Output: the next query object to be processed
  function getQueryToProcess(*tree*, *path*)
    $lastQO \leftarrow path.lastQueryObject$
    $node \leftarrow tree.findLowestNonEmptyParent(lastQO)$
    while $node \neq null$ and *node* is not a leaf **do**
      $node \leftarrow tree.selectChild(node)$
    return *node*

A general strategy for selecting a particular child c (i.e., the subtree with the root c) is to identify dense subtrees (dense clusters) in order to achieve high throughput. A possible way is to select the subtree that minimizes the average query time of processing all its leaf nodes. This way we select the subtree with the largest immediate contribution to the throughput.

Let us illustrate the algorithm with an example. Consider the tree in Fig. 3. Suppose that the currently processed query object is q_2; the other query objects have not been processed yet. Let us find the lowest nonempty parent of $(q_2, 3)$, which is the root $((), 8)$. There is only one possible child to select, which is

$((p_1), 2)$. Now, there are two possible children of $((p_1), 2)$. In order to select one child or the other, the average query time is estimated for each of the two subtrees as follows. The subtree of $((p_1, p_3), 1)$ contains just one leaf node $(q_3, 4)$ and the maximum query time is 8, since the lowest common parent of the last query q_2 and q_3 is $((), 8)$. The maximum time to process the first leaf of the subtree $((p_1, p_2), 1)$ is 8. The second leaf needs at most 1 time unit for processing, since the lowest common parent of $(q_1, 1)$ and $(q_4, 7)$ is $((p_1, p_2), 1)$. Therefore, the $((p_1, p_2), 1)$ subtree is selected, since it minimizes the average query time. Subsequently both its leaves are processed.

Let us formalize the estimation of the average query time. Let $n = (pref_n, qt_n)$ be a nonempty internal node of the tree and $m = (pref_m, qt_m)$ be the lowest common parent of n and of the last processed leaf node. We suppose the leaf node is not a descendant of n.

The maximum time to process all the leaf nodes of the subtree n equals

$$qt_m + childrenQueryTime(n)$$

where qt_m is the maximum query time of the first query object q of n. $childrenQueryTime(n)$ is the maximum time to process all the other descending leaf nodes:

$$childrenQueryTime(n) =$$

$$= max\{|n.children| - 1, 0\} \cdot qt_n + \sum_{c \in n.children} childrenQueryTime(c)$$

where $n.children$ is the set of nonempty direct children of n; qt_n is the maximum time to process a leaf node of n when the lowest common parent of the last and the next processed leaf is n. $childrenQueryTime$ is applied recursively for individual children of n.

The described strategy when the subtree minimizing the average query time is selected possesses two possible caveats. It takes only the current state of the tree to make the decision without considering evolution of the tree. The other caveat is the possibility of starvation because some subtrees may never be chosen for processing.

Let us explain the evolution of the tree of clusters caveat. Let p be a subtree that at a time t contains a set of leaf nodes C_t that are all unprocessed and at a time u it contains a set of leaf nodes C_u such that $t < u$ and $C_t \subset C_u$. Let us consider two scenarios. In the first one, p is selected as the subtree to be processed at the time t and again at the time u. In the second one, p is selected just at the time u. The set of processed leaf nodes is the same for both the scenarios (C_u), but the query time is lower in the second one, since all the leaves of p are processed in a row achieving higher cache utility. It implies that it pays off to process p as few times as possible to achieve low query times, since the subtree "switches" introduce processing overhead.

For illustration purposes, see the Fig. 3 again and let us consider the subtree $((p_1, p_2), 1)$. If both the children are processed in a row, the maximum query time to process the second one is one, since their lowest common parent is $((p_1, p_2), 1)$.

However, if just $(q_1, 1)$ is processed, then another subtree is processed and $(q_4, 7)$ is processed later. The maximum overall time to process $(q_1, 1)$ and $(q_4, 7)$ is then higher than in the first scenario, since $((p_1, p_2), 1)$ is not the lowest common parent of $(q_4, 7)$ and of the previous leaf node. Thus the cost of processing $(q_4, 7)$ is higher than one.

When also considering the starvation problem, a suitable strategy is to choose the subtree that contains the oldest unprocessed leaf node among the considered subtrees. This way the starvation problem is diminished and also the number of times a subtree is chosen for processing is limited.

In fact, the starvation is not really eliminated. There can be a situation when the processing gets stuck in a single subtree if there are always query objects to be processed in this subtree and it cannot be completely emptied. Such a situation can be solved by setting a time limit during which a single subtree of the root node can be continuously processed. When the time limit is reached, the tree traversal is reset to the root node.

The two presented implementations of the *selectChild* function (minimal average query time and oldest leaf) are experimentally compared later in this paper.

5 Experiments

In this section, we experimentally evaluate the proposed techniques for the throughput maximization. We start by describing the setup of the experiments in Sect. 5.1. The impact of the cache utility on the query time is explored in Sect. 5.2. We show how the buffer size influences the throughput in Sect. 5.3. In Sect. 5.4, we experiment with different rates of incoming query objects in the stream. The cache size impact is evaluated in Sect. 5.5. Different ways of constructing the query graph are experimentally compared in Sect. 5.6.

5.1 Setup of Experiments

We use the M-Index [15] structure to index the metric-space data. It employs practically all known principles of metric space partitioning, pruning, and filtering, thus reaching high search performance. The actual data are separated into partitions, which are stored as separate files on a disk and read into the main memory during query evaluations. To partition the data, M-Index uses a set of pivots. To insert an object into the index, the pivots are sorted based on the distance to the object. In this way, a pivot permutation is obtained, which identifies the data partition to insert the object. During a similarity search, mutual distances between the query object and the pivots are used to reduce the set of data partitions that need to be accessed. The M-Index supports executing approximate kNN queries among other operations. One of the stop conditions of a query evaluation is given by the maximum number of accessed objects (the size of a candidate set). Such a stop condition is used in our experiments.

One of the reasons why we chose to use the M-Index is that it can use the same set of pivots as are used for the query graph construction in Sect. 4.4. This is beneficial for the effectiveness of the query ordering. It also improves efficiency because the distances from a query object to the pivots can be computed just once and used both in the query graph and in the M-Index. Another reason for selecting the M-Index is that it also achieves high search performance in high-dimensional vector spaces.

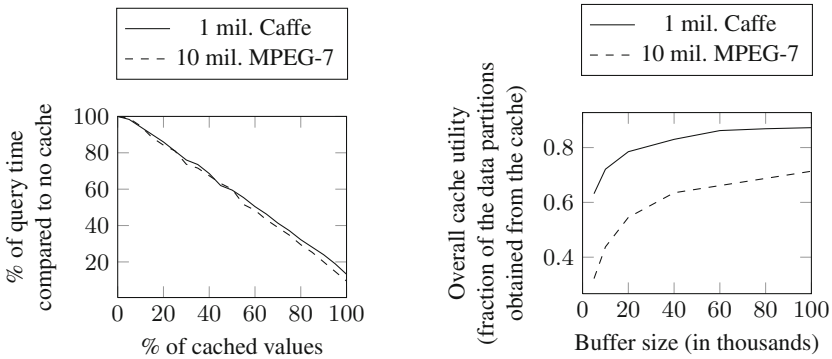
For the experiments, we use the Profimedia dataset of images [6], which is a freely-available large-scale dataset for evaluation of content-based image retrieval systems. We created two different subsets of the images and extracted their visual-feature descriptors. The generated datasets are: 1 million Caffe descriptors [11] (4096 dimensional vectors) and 10 million MPEG-7 descriptors. Separately, we created streams of images represented by corresponding descriptors. During each experiment, images from the respective collection are continuously streamed and stored in the buffer from which they are processed by approximate 10-NN queries. For the approximate kNN queries, we used candidate sets of size 1,000 for the Caffe dataset and size 2,000 for the MPEG-7 dataset. We applied the Euclidean distance and the weighted Euclidean distance as the distance functions for the metric space with Caffe and MPEG-7 descriptors, respectively. For both datasets, we use 160 randomly selected objects as pivots. In the M-Index, this pivot selection strategy was observed to provide similar search performance as other more sophisticated strategies [15].

If not said otherwise, the maximum size of the cache is set to 40,000 descriptors for the Caffe dataset (i.e., 4% of the database); up to 90,000 descriptors are stored for the MPEG-7 dataset (i.e., 0.9% of the database). The least recently used policy is used when inserting to the full cache. In particular, the data partitions with the oldest last access time are discarded and replaced with the new partitions of the current query so that the maximum size of the cache is maintained. To traverse the tree of clusters, the oldest leaf approach is used (see Sect. 4.6) if not stated otherwise.

The tested applications are implemented using Java programming language with the use of the MESSIF library [3] providing an implementation of the M-Index. The experiments were run on Intel Xeon 2.00 GHz with 8 GB RAM. The datasets are stored on a HDD (access time 5 ms, transfer rate 90 MBps). We have run each of the experiments multiple times (at least three) and we have taken the median values where appropriate.

5.2 Cache Utility vs. Query Time

At first, the impact of the cache utility on the query time is explored to validate Formula 3. We ran approximate 10-NN queries for each dataset and we were continuously changing the percentage of the data partitions that could be obtained from the cache. The results are shown in Fig. 4a. The x-axis shows the percentage of the cached values of all the data needed for processing of a particular query; the y-axis represents the percentage of the time to process the query compared to the situation when the cache is not used. It can be observed



(a) Cache utility vs. query time (b) Buffer size vs. cache utility

Fig. 4. Cache utility experiments

that the processing time can be improved dramatically (below 10% of the original time) if the cache is filled with appropriate values, thus the assumption in Formula 3 is valid.

5.3 Buffer Size Impact

In the next group of experiments, we explore the impact of the size of the buffer on the cache utility and the throughput. The buffer size denotes the number of query objects that are waiting for their processing and thus are a subject to reordering. The size of the buffer was fixed during the experiments. At the beginning, the buffer was filled with the query objects from the stream up to the given size; then another query object was loaded whenever a query has been processed to keep the size of the buffer constant. Exactly 100,000 query objects were processed during each experiment. We can observe that with a growing size of the buffer, both the cache utility and the throughput grow because the processed subgraphs (clusters) are denser and the cached values are reused more often, see Figs. 4b and 5. The throughput speedup was computed as the ratio of the processing time using a given buffer size and the processing time when the query objects were processed in their original order without the caching mechanism.

The results when the query objects were processed in their original order (i.e., there is no reordering) are captured by Table 1. Average query times were measured with and without use of the cache. The cache utility was very low, which means the reordering of query objects significantly influences the efficiency.

5.4 Input Frequency Experiments

In the following experiments, we set a fixed frequency f of the incoming query objects of the stream. It means the stream follows the pattern $((q_1, 0), (q_2, f), \ldots, (q_i, (i-1) \cdot f), \ldots)$. We measured the throughput by observing the size of the buffer (i.e., the number of unprocessed query objects). Each experiment was run with a different frequency of streamed query objects.

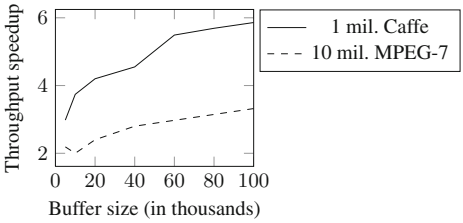


Fig. 5. Buffer size vs. throughput

Table 1. Processing query objects in their original order

Dataset	Cache	Cache utility	Avg. query time [ms]
10 mil. MPEG-7	Yes	0.01	103
10 mil. MPEG-7	No	-	113
1 mil. Caffe	Yes	0.03	65
1 mil. Caffe	No	-	69

The processing was run for two hours for the Caffe dataset and for four hours for the MPEG-7 dataset. The two approaches presented in Sect. 4.6 are compared. In particular, these are the strategies for ordering the subtrees during the depth-first traversal of the tree of clusters, namely, the oldest leaf (OL) approach and the minimal average query time ($MAQT$) approach. The oldest leaf strategy selects the subtree containing the oldest unprocessed query object, while the minimal average query time strategy selects the subtree minimizing the average query time to process all the unprocessed query objects of the subtree.

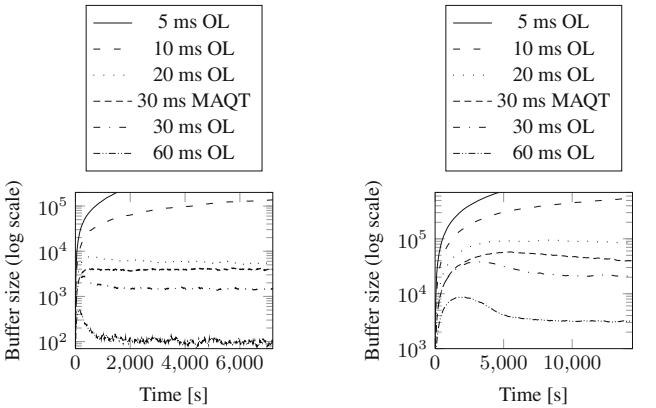
The results are shown in Fig. 6. For the oldest leaf approach, we show the results for input frequencies of 5 ms, 10 ms, 20 ms, 30 ms, and 60 ms. For the $MAQT$ approach, the results are shown just for the 30 ms frequency. It can be observed that the size of the buffer grows at the beginning because the query graph does not contain dense subgraphs, which are essential for the processing speedup. As soon as the subgraphs are dense enough, the processing is more efficient and eventually the buffer size stabilizes once the average query time equals the input frequency. The results are in compliance with the results regarding the buffer size: as the input rate of the streamed items increases, the buffer size also increases to keep up with the incoming query objects. It can be seen the OL approach gives a little better results than the $MAQT$ one. Moreover OL does not have the disadvantage of the starvation.

In all the cases except for the 5 ms and 10 ms frequencies, it was possible to achieve sufficient throughput so that the buffer size was practically stable. It

means that, for example, the average query time was 20 ms for the 20 ms input frequency after the initial phase. The experiments with 5 ms were stopped after reaching the size of the buffer of 200,000 or 700,000 for the Caffe or MPEG-7 dataset, respectively.

Tables 2 and 3 show comparison of delays for individual input frequencies. The delay is the time since a query object enters the buffer until it is processed by the metric index. As expected, the median and the maximum delays are greater for more rapid streams because the buffer sizes are higher and it takes a longer time until a particular query object can be processed. The maximum delay for the *MAQT* approach is very high because it has no starvation prevention. The tables also present the overall cache utility computed as the ratio of the number of data partitions loaded from the cache and of all the data partitions needed during the processing. The cache utility correlates with the size of the buffer as was already seen in Fig. 4b.

Exact distribution of the delays for the experiment with the MPEG-7 dataset and 30 ms input frequency can be observed in Fig. 7. The graph shows the percentage of queries that were processed until a given delay. So for example, we can see that 50% of queries were delayed maximally by 13 min, while about 10% of queries were kept in the buffer for more than 25 min.



(a) 1 mil. Caffe, 2 hour processing (b) 10 mil. MPEG-7, 4 hour processing

Fig. 6. The buffer size evolution in time during fixed input frequency experiments

Table 2. Fixed input frequency statistics for Caffe dataset

Input frequency [ms]	10 OL	20 OL	30 OL	30 MAQT	60 OL
Max delay [*s*]	2721	309	169	7036	59
Median delay [*s*]	740	117	47	38	7
Cache utility	0.87	0.65	0.44	0.48	0.08

Table 3. Fixed input frequency statistics for MPEG-7 dataset

Input frequency [ms]	10 OL	20 OL	30 OL	30 MAQT	60 OL
Max delay [s]	6409	4031	2988	6557	1565
Median delay [s]	2599	1525	894	1050	234
Cache utility	0.92	0.78	0.59	0.64	0.30

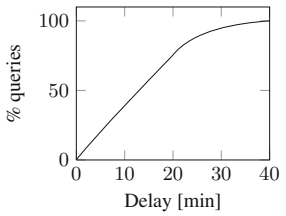


Fig. 7. Delay cumulative function; MPEG-7 dataset; 30 ms input frequency; OL approach; 4 h run time

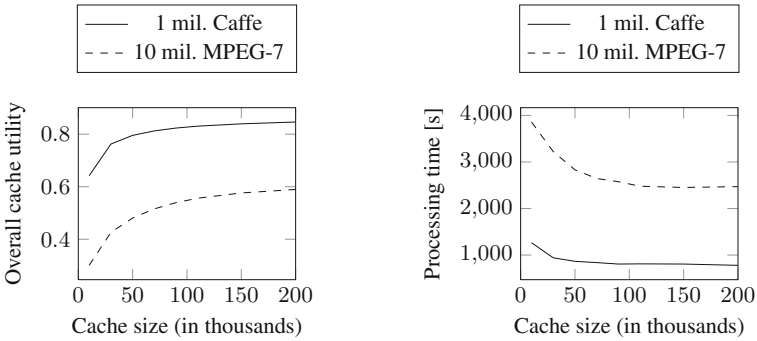
5.5 Cache Size Experiments

The cache utility is likely to increase with an increasing amount of the cached data, since a query can reuse data of multiple previous queries. This expectation is validated by the following experiments. A set of 50,000 queries was processed using a fixed-sized buffer containing 20,000 query objects. The graphs in Fig. 8 depict the results. The cache utility increases with growing cache size; the processing time decreases as the cache size gets larger. We can observe that caching very large amount of data does not bring much improvement in the cache utility and the processing time so for the other experiments, we selected 40,000 and 90,000 objects for the Caffe and the MPEG-7 datasets, respectively, as an appropriate trade-off between the processing time and the storage space.

5.6 Query Graph Construction Experiments

In this section, we compare three approaches to the query graph construction (described in Sect. 4.4). The first one is the pivot based technique we use in the other experiments; that is, the edge values (maximum query times) of the query graph are estimated according to common pivot permutation prefix lengths of query objects. Another approach we consider is the one when the metric distances to all the query objects of the query graph are explicitly computed and the edge values correspond to the distances. The last one uses the knowledge of data partitions needed during evaluation of individual queries and the edge values are determined according to the number of common partitions.

In the experiments, we used the MPEG-7 dataset and a finite stream of 10,000 query objects $((q_1, 1), \ldots, (q_{10000}, 1))$; that is, all the query objects were



(a) Cache size vs. cache utility

(b) Cache size vs. processing time

Fig. 8. Cache size experiments; fixed-sized buffer of 20,000 query objects; 50,000 processed queries

available at once. For each approach, a path in the query graph was found going through all the 10,000 query objects. The query objects were processed in that order and the overall cache utility was acquired.

We considered two cache policies for each approach. In one case, the cache was used to keep only the data partitions that were needed to evaluate one previous query (OPP policy). This is the strategy that is used to set the upper bound of the query time in Sect. 4.3. The other policy is the one that is used for the other experiments; that is, the cache size is limited by 90,000 objects and the least recently used replacement strategy is applied (LRU policy).

For the explicit distance computations technique, the query graph was constructed so that the edge values were set to the metric distances between the query objects. Subsequently a path in the query graph was found using the 2-approximate minimum spanning tree heuristics designed for the traveling salesman problem [13]. Note that the definition of the query graph requires query times to be assigned to the edges. However, for the purposes of the experiments, we worked directly with the metric distances assigned to the edges.

The sum of the edges of the constructed minimum spanning tree was 12,816, which sets the minimal bound on the shortest path. The actual length of the path obtained from the minimum spanning tree was 15,583. We also took the path generated using the pivot based technique and computed its length in terms of the metric distances between query objects, which resulted in the length of 18,059. However, the achieved cache utility for the OPP policy using the pivot based technique was 9% compared to only 3% using the minimum spanning tree path. For the LRU policy, the cache utility using the pivot based technique was 32% compared to only 10% using the minimum spanning tree path. It means the metric distance is not strictly correlated with the cache utility (and the query time). The choice of the query graph construction approach should consider the properties of the used metric index so that the query times are estimated correctly and the query ordering is effective.

For the next approach, the data partitions needed for the evaluation of individual queries were obtained beforehand. The value of the query graph edge $q_i q_j$ was assigned according to the difference of the data partitions needed for processing q_i and q_j; specifically we took $max\{|I(q_i) - I(q_j)|, |I(q_j) - I(q_i)|\}$ where $I(q)$ is the set of the data partitions needed for processing q. The path was found using the minimum spanning tree heuristics, which resulted in the cache utility of 15% for the OPP policy and 29% for LRU. As for OPP, this approach achieves better cache utility than the pivot based approach. If LRU is used, the achieved results are similar to those obtained for the pivot based technique. However, the precise set of needed data partitions is not generally known prior to actual evaluation of the query so this approach is not usable in practice.

A summary of the results is captured by Table 4. We can conclude that from the practical point of view, the pivot based strategy is the most suitable.

Table 4. Query graph construction experiments

Constr. strat.	Cache policy	Cache util.
Pivot based	OPP	0.09
Pivot based	LRU	0.32
Metric dist.	OPP	0.03
Metric dist.	LRU	0.10
Data parts.	OPP	0.15
Data parts.	LRU	0.29

6 Delay Improvement

The proposed approaches so far were focused on the throughput maximization of the query processing and the delays of individual query objects were not targeted for optimization. To recall, the delay is the time since a query object was added into the stream until it was processed. In this section, we show how the throughput can be traded off for improving the delays.

In Fig. 7, which shows the delay cumulative function for the experiment with the MPEG-7 dataset and a fixed input frequency of 30 ms, we can observe that just 5% of the queries were processed with maximally 1 min delay. In this section, we explore the scenario when we want to process more queries until a given delay limit while maintaining sufficient throughput.

Formally, the problem is defined as follows. Given the time limit T, the task is to find a path $((q_{i_1}, t_{i_1}), \ldots, (q_{i_k}, t_{i_k}))$ in the query graph G_T so that $start_k < T$ where $start_j$ is the time when the query object q_{i_j} starts to be evaluated (for more details see Sect. 4.3). Given the delay limit DL, the optimal path maximizes the expression

$$w \cdot |beforeLimit| + |afterLimit|$$

where *beforeLimit* is the set of processed query objects with the delay of maximally *DL*; *afterLimit* is the set of the other processed query objects; $w \geq 1$ is a parameter determining the weight of the query objects processed until the given delay limit.

Such a criterion can also be imagined as weighted throughput. A certain number of points is received for each processed query object. If a query object is processed before the given delay limit, more points are obtained. The goal is to maximize the number of received points in a given time interval.

Let us define *beforeLimit* and *afterLimit* sets formally. We start with the definition of the delay as the time since a query object was added into the stream until it was processed:

$$delay((q_{i_j}, t_{i_j})) = start_j + qt_{i_j} - t_{i_j} \text{ for } j \geq 1$$

where qt_{i_j} is the time to process q_{i_j}.

$$beforeLimit = \{(q_{i_j}, t_{i_j}) \text{ — } delay((q_{i_j}, t_{i_j})) \leq DL\}$$
$$afterLimit = \{(q_{i_j}, t_{i_j}) \text{ — } delay((q_{i_j}, t_{i_j})) > DL\}$$

To address the specified criterion, we modify the strategy for ordering the children subtrees in the depth-first traversal of the tree of clusters (presented in Sect. 4.6). Instead of choosing the subtree containing the oldest query object, a score for each subtree candidate is computed:

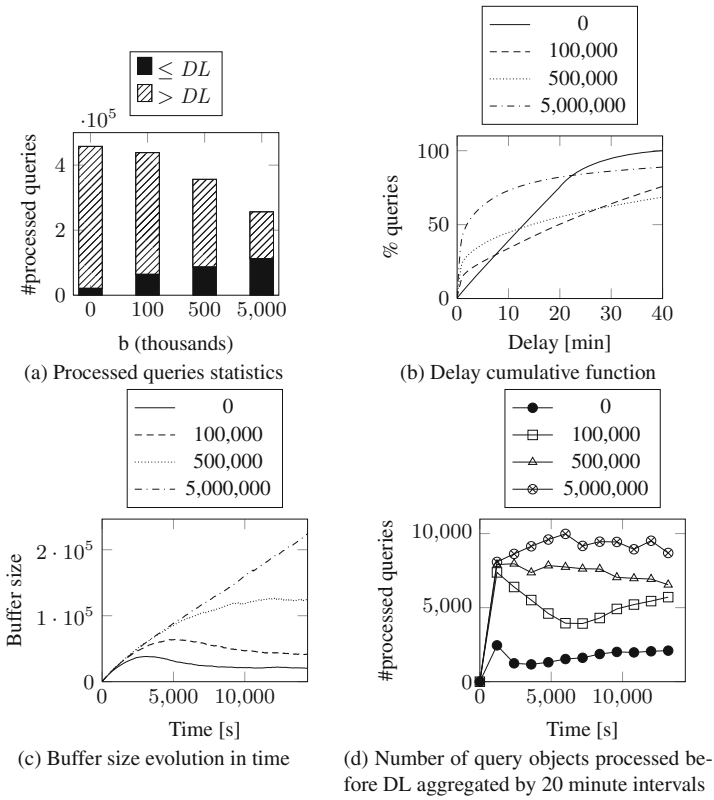
$$a \cdot oldestItemDelay + b \cdot beforeLimitCount$$

where *oldestItemDelay* is the time since the oldest unprocessed query object of the subtree entered the stream; *beforeLimitCount* is the number of unprocessed query objects of the subtree that have been in the stream for at most *DL* time; a and b are the weights influencing the trade-off of the delays and the throughput. When $b = 0$, we get the original oldest leaf approach. As b grows, the subtrees possessing newest query objects are more and more prioritized over the ones containing the oldest leaves. The optimal value of b correlates with the value of the weight w. The influence of the parameters is studied experimentally.

6.1 Delay Improvement Experiments

The following experiments were conducted for different values of b to see how the throughput and the number of queries processed before the given delay limit change. The value of a was fixed to 1. The 10 mil. MPEG-7 dataset was used; the delay limit *DL* was set to 1 min; a new query object entered the buffer every 30 ms. Every experiment was run for 4 h.

The results are depicted in Fig. 9. By increasing the weight of the newest query objects, it is possible to increase the number of query objects with the delay below 1 min. On the other hand, the overall throughput decreases, since subtrees with lower densities are prioritized (see Fig. 9a). The graph in Fig. 9b



(a) Processed queries statistics

(b) Delay cumulative function

(c) Buffer size evolution in time

(d) Number of query objects processed before DL aggregated by 20 minute intervals

Fig. 9. Throughput delay trade-off for various b values; DL = 1 min; MPEG-7 dataset; 30 ms input frequency

depicts the percentage of queries processed until a corresponding delay. The percentage of low delayed queries gets higher with the growing weight b; on the other hand, the percentage of high delayed queries also increases because of the lower throughput. Figure 9c shows the evolution of the buffer size for individual weights. The larger the weight is, the larger the buffer size has to be in order to keep up with the rate of the incoming query objects. When the weight is too high, the buffer grows to very large sizes, since the average query time is very high.

Figure 9d captures the numbers of query objects processed within the delay limit throughout the time. The runs with nonzero b reach approximately the same value after the first 20 min. After that, we can observe the influence of b, since some of the oldest query objects become prioritized.

As can be seen from the experiments, choosing inappropriate values of the parameters a and b can lead to unwanted results as the throughput drops to very low values. In the following, we present a modification of the approach for dynamical setting of the parameters according to the size of the buffer.

The application can be in one of two states. When the first state is active, the throughput maximization is aimed for (THRMAX state). When the other state is active, the number of query objects processed until the given delay limit is maximized (DELLIM state).

The behavior of the application in individual states differs in the setting of the weighting parameters. In the THRMAX state, the oldest leaf approach is applied, that is, $a = 1$, $b = 0$. For the DELLIM state, the subtree containing the largest number of unprocessed query objects that have spent at most DL (the delay limit) in the buffer is selected, that is, $a = 0$, $b = 1$.

The application switches from one state to the other according to the buffer size. In particular, there is a lower and an upper buffer size limit. When the upper limit is exceeded, the system goes to the THRMAX state to maximize the throughput so that the buffer size can be lowered down. When the size of the buffer drops below the lower limit, the DELLIM state is entered to maximize the number of query objects processed before the given delay limit.

Experiments addressing the state switch approach were run with the same settings as the first approach: that is, the MPEG-7 dataset, 1 min delay limit, 30 ms input frequency and four hour run time. Each experiment was run with different limits of the buffer size; the initial state was the DELLIM state. Figure 10 shows how the buffer size evolves in time. We can see that it oscillates between the upper and the lower limit. When the upper limit is hit, the throughput is enhanced by switching to the THRMAX state and the buffer size decreases. When the lower limit is reached, the DELLIM state is entered to increase the number of query objects processed until the given delay limit. However, the throughput is decreased and the buffer enlarges.

Table 5 provides an insight into the approach. Each row corresponds to one experiment with the given lower and upper switch limits of the buffer size. Individual statistics are computed from the second switch when the behavior of the buffer size evolution stabilizes (see Fig. 10) and relevant results can be obtained. In particular, the statistics are computed between the second time the application enters the DELLIM state and the last time the THRMAX state is exited. This ensures that the application was in each state the same number of times.

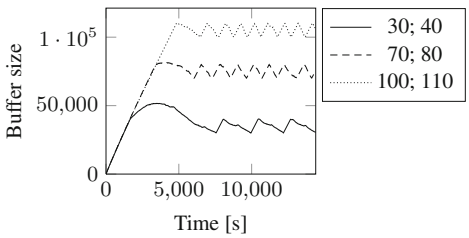


Fig. 10. Buffer size evolution in time for the state switch approach with various switch limits; DL = 1 min; 10 mil. MPEG-7 dataset; 30 ms input frequency; the legend is in the format "lower limit; upper limit" in thousands

Table 5. State switch approach statistics

Lower limit	Upper limit	Time in DS [%]	q in DS [%]	q < DL [%]	q < DL in DS [%]
30,000	40,000	26.4	10.6	13.4	78.4
70,000	80,000	43.9	17.2	18.4	92.1
100,000	110,000	47.5	18.0	18.8	94.3
65,000	85,000	43.6	17.3	18.6	91.5
75,000	80,000	42.6	16.7	17.9	91.8
79,000	80,000	39.8	16.5	17.9	91.0

The column *time in DS* contains the percentage of the time the application spent in the DELLIM state. When a larger buffer size is used, the percentage increases. This is because the average query time is low for large buffer sizes and so the buffer size grows slowly in the DELLIM state and falls fast in the THRMAX state.

The column *q in DS* contains the percentage of queries processed when the application was in the DELLIM state. It correlates with the time spent in the DELLIM state, but it is lower than the percentage of the time spent in the DELLIM state, since the average query time is higher in the DELLIM state than in the THRMAX state.

The column $q < DL$ shows the percentage of queries that were processed until the given delay limit (1 min). Again, this correlates with the time spent in the DELLIM state. To compare the numbers with the oldest leaf approach, which is used for the throughput maximization, only 5% of queries were processed with the delay at most 1 min.

The last column $q < DL$ *in DS* contains the percentage of the "before-limit" queries that were processed while the DELLIM state was active. In other words, all the queries that were processed within the delay limit form the whole (100%); the displayed value in the column is the percentage of those that were processed in the DELLIM state. It can be observed that most of the queries that were processed until the delay limit were in fact processed while the DELLIM state was active.

As it can be seen from the experiments, the presented approach can be successfully used for the throughput delay trade-off.

7 Conclusion

We have presented a novel approach to enhance the throughput of similarity search query processing. The technique is based on dynamic reordering of the incoming query objects combined with in-memory caching of the data partitions used to evaluate previous queries. The representation of the query reordering problem is simplified using a query graph thus allowing a theoretical analysis of the proposed techniques. An appropriate ordering of the queries is continuously created by generating a path in the query graph. The introduced methods are

verified experimentally with positive results. The presented approach allows to achieve significantly better throughput than the baseline approach when the query objects are evaluated in their original order.

In addition to the throughput maximization, we also targeted optimization criterion trading off the throughput for low delays. The presented technique is based on modification of the query ordering strategy proposed for the throughput maximization. It is parameterized according to the amount of the throughput that can be sacrificed.

Acknowledgement. This work was supported by the Czech national research project GA16-18889S.

References

1. Amato, G., Esuli, A., Falchi, F.: A comparison of pivot selection techniques for permutation-based indexing. Inf. Syst. **52**, 176–188 (2015)
2. Barrios, J.M., Bustos, B., Skopal, T.: Analyzing and dynamically indexing the query set. Inf. Syst. **45**, 37–47 (2014)
3. Batko, M., Novak, D., Zezula, P.: MESSIF: metric similarity search implementation framework. In: Thanos, C., Borri, F., Candela, L. (eds.) DELOS 2007. LNCS, vol. 4877, pp. 1–10. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77088-6_1
4. Bellmore, M., Nemhauser, G.L.: The traveling salesman problem: a survey. Oper. Res. **16**(3), 538–558 (1968)
5. Brisaboa, N.R., Cerdeira-Pena, A., Gil-Costa, V., Marin, M., Pedreira, O.: Efficient similarity search by combining indexing and caching strategies. In: Italiano, G.F., Margaria-Steffen, T., Pokorný, J., Quisquater, J.-J., Wattenhofer, R. (eds.) SOFSEM 2015. LNCS, vol. 8939, pp. 486–497. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46078-8_40
6. Budikova, P., Batko, M., Zezula, P.: Evaluation platform for content-based image retrieval systems. In: Gradmann, S., Borri, F., Meghini, C., Schuldt, H. (eds.) TPDL 2011. LNCS, vol. 6966, pp. 130–142. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24469-8_15
7. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Trans. Patt. Anal. Mach. Intell. **30**(9), 1647–1658 (2008)
8. Chung, Y., Su, I., Lee, C., Liu, P.: Multiple k nearest neighbor search. World Wide Web **20**(2), 371–398 (2017)
9. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of web search engines: caching and prefetching query results by exploiting historical usage data. ACM Trans. Inf. Syst. **24**(1), 51–78 (2006)
10. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: Similarity caching in large-scale image retrieval. Inf. Process. Manage. **48**(5), 803–818 (2012)
11. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, MM 2014, Orlando, FL, USA, 03–07 November 2014, pp. 675–678. ACM (2014)
12. Karedla, R., Love, J.S., Wherry, B.G.: Caching strategies to improve disk system performance. IEEE Comput. **27**(3), 38–46 (1994)
13. Laporte, G.: The traveling salesman problem: an overview of exact and approximate algorithms. Eur. J. Oper. Res. **59**(2), 231–247 (1992)

14. Nalepa, F., Batko, M., Zezula, P.: Enhancing similarity search throughput by dynamic query reordering. In: Hartmann, S., Ma, H. (eds.) DEXA 2016. LNCS, vol. 9828, pp. 185–200. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44406-2_14
15. Novak, D., Batko, M., Zezula, P.: Metric index: an efficient and scalable solution for precise and approximate similarity search. Inf. Syst. **36**(4), 721–733 (2011)
16. Pandey, S., Broder, A.Z., Chierichetti, F., Josifovski, V., Kumar, R., Vassilvitskii, S.: Nearest-neighbor caching for content-match applications. In: Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, 20–24 April 2009, pp. 441–450. ACM (2009)
17. Shao, J., Huang, Z., Shen, H.T., Zhou, X., Lim, E., Li, Y.: Batch nearest neighbor search for video retrieval. IEEE Trans. Multimedia **10**(3), 409–420 (2008)
18. 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)
19. Solar, R., Gil-Costa, V., Marín, M.: Evaluation of static/dynamic cache for similarity search engines. In: Freivalds, R.M., Engels, G., Catania, B. (eds.) SOFSEM 2016. LNCS, vol. 9587, pp. 615–627. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49192-8_50
20. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity search - the metric space approach. In: Advances in Database Systems, vol. 32. Kluwer (2006)