Towards High Similarity Search Throughput by Dynamic Query Reordering and Parallel Processing

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 extend our previously published technique that dynamically reorders the incoming queries in order to use our caching mechanism more effectively. The extension lies in adoption of a parallel computing environment which allows us to process multiple queries simultaneously.

Keywords: Stream processing · Similarity search · Parallel processing

1 Introduction

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

Due to the nature of the data and applications which use them, it can be desired to view the data as a potentially infinite stream which is continuously being created. For example, consider a text search-engine crawler that gathers images from the web and needs to continuously annotate them by textual descriptions according to the image content. Another example can be a spam filter that receives incoming emails and compares them to some learned spam knowledge base so that the spam messages can be detected. Finally, consider a news notification system which 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 all these applications is processing the streamed data items by some form of content-based searching. The performance of these applications is mostly determined by the number of processed data items in a given time interval, i.e., the throughput is the most important metric. The individual query search time can be improved by applying some similarity indexing technique, for which there are 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 a slight delay of the single query processing if the overall throughput of the system is improved. Performance of such stream processing applications is studied in [12,13].

I/O costs typically have a significant effect on the performance of similarity search techniques. In our work, we exploit the fact that some orderings of the processed queries can result in considerably lower I/O costs and overall processing times than random orderings. This is based on the assumption that two similar queries need to access similar data of the search index. By obtaining an appropriate ordering of queries, the accessed data can be cached in the main memory and reused for evaluation of similar queries thus lowering down the I/O costs. We have previously published [14] a technique which dynamically reorders the incoming queries that allows to achieve a significant improvement of the throughput.

In this paper, we provide an extension of the technique by adopting it to a parallel computing environment where multiple queries can be processed simultaneously by individual query processors. Due to the nature of our approach, it is very important how the streamed query objects are distributed among the query processors, i.e., which query object goes to which query processor. The main contribution of this work is a proposal of effective and efficient ways in which the query objects are spread among the processors so that high throughput is achieved.

The rest of the paper is organized as follows. First, we present related work on caching and query reordering in similarity search, and parallelization in stream processing. In Sect. 3 we formally define our problem. The originally published technique is summarized in Sect. 4. Its adoption to the parallel computing environment is presented in Sect. 5. Experimental evaluation can be found in Sect. 6.

2 Related Work

The usage of a caching mechanism in similarity search has been proposed in several papers to reduce the amount of I/O operations. In [6], the authors propose caching of similarity search results and reusing them to produce approximate results of similar queries. The concept of caching similarity search results is used also in [16]. The paper focuses on caching policies which incrementally reorganize the cache to ensure that the cached items cover the similarity space efficiently. The Static/Dynamic cache presented in [19] consists of a static part to store queries (along with their results) that remain popular over time and a dynamic part to keep queries that are popular for a short period of time. Authors of [4] present a caching system to obtain quick approximate answers. If the cache cannot provide the answer, the distances computed up to that moment are used to query the index so that the computations are not wasted.

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. Intersections of candidate sets between every pair of queries need to be computed and updated periodically. This approach is designed for a relatively small number of queries (tens). Since we have tens of thousands of queries to be evaluated (as will be seen in Sect. 6), the overhead of such computations is likely to be very high.

The authors of the paper [18] propose D-cache which stores distances computed during previous queries. The Snake Table [2] uses a cache of distances to improve performance of processing streams of queries with snake distribution (i.e., consecutive query objects are similar). In [1], an inverted cache index stores statistics about usefulness of data partitions in order to modify priority queues.

All the aforementioned techniques are designed for interactive applications when queries are evaluated immediately. We focus on scenarios when delays are affordable and the throughput is the main issue which calls for a different type of a solution.

Speaking generally of stream processing, parallelization is a common technique for throughput enhancement. This is typically referred to as an *operator replication* where an operator is a component for processing the stream. Each data item of the stream is sent to one of the replicas where it is subsequently processed. There is a number of issues which need to be dealt with, e.g., determining optimal number of replicas or creating an appropriate strategy for deciding which data item is processed by which replica. These challenges have been widely explored [7,8,10,11]. In our work, we do not aim at enhancing parallel stream processing of general applications. We rather focus on designing schemas of the parallelization applicable to our specific case which is not covered by the general approaches.

In [3], techniques for parallelization of similarity search are studied. The approaches are based on creating distributed metric index structures and parallelization of a single query evaluation. In our work, we use the parallelization to evaluate multiple queries concurrently. In our case, we avoid the overhead related to the distribution of a single query to multiple processing components and the overhead caused by merging partial results into the final answer.

3 Problem Definition and Objectives

Suppose there is a domain of complex objects D (e.g., images) and a large database containing such objects $X \subseteq D$. Let $s = (d_1, d_2, \ldots)$ be a stream, i.e., 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 it was created (entered the application). It holds that $t_i \leq t_{i+1}$ for each i and $t_1 = 0$.

As a universal model of similarity we use the metric space (D, d) [20], where d is a total distance function $d : D \times D \rightarrow R$. The distance between two objects corresponds to the level of their dissimilarity ($d(p, p) = 0$, $d(o, p) \geq 0$).

For each query object q_i in the stream s, a k-nearest neighbors query $NN(q_i, k)$ is executed which returns k nearest objects from the database to the query object. It is allowed to change the order of the processed query objects. More precisely, at 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 throughput is maximized. Specifically, we want to maximize the number of processed query objects of a given stream until a given time T. Alternatively, the criteria can be defined as minimization of the number of unprocessed query objects at the time T, i.e., the number of (q_i, t_i) where $t_i \leq T$ and q_i is not processed.

Our objective is to extend our previously proposed technique enhancing the throughput of the similarity search. The extension lies in parallel processing of multiple queries. In particular, we focus on the ways to distribute the queries among individual query processors to gain maximum effectiveness.

4 Enhancing Throughput with a Single Query Processor

In this paper, we build upon our previous work [14] in which we proposed a technique for enhancing the throughput of the similarity search by dynamically reordering the queries combined with a caching mechanism to lower down the I/O costs during processing. In this section, we summarize this technique.

We consider a generic metric index which uses data partitioning $P = \{p_1, \ldots, p_n\}$ where $p_i \subseteq D$. When evaluating a query, a subset of the partitions $Q \subseteq P$ 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 at decreasing the number of disk accesses and consequently the time to process the queries.

We make use of the following feature of data partitioning methods. If two query objects are very similar to each other, the sets of accessed data partitions are also very similar. 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. Since huge databases are often used in practice, there is a very low probability that two subsequent queries in the stream are similar enough to access similar 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, the 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.

4.1 Query Ordering

The problem of the query object ordering can be viewed as a graph problem. Let $s = ((q_1, t_1), (q_2, t_2), \ldots)$ be the stream to be processed. Let $((q_1, t_1), \ldots, (q_k, t_k))$ be a finite subsequence of s so that $t_k \leq t$ and $t_{k+1} > t$ for a given t, i.e., the query objects which have become available by the time t. We define the *query graph* $G_t = (V, E)$ at the time t. 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, i.e., 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 the query time to process q_i right after q_j or vice versa. The assigned time is based on the extent of the cache utilization.

To formally define the throughput maximization, 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 $start_k < t$. The path represents the ordering of the queries; $start_k$ 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, i.e., the number of processed query objects. This is in fact a variation of the traveling salesman problem (an NP-hard problem). There is an added difficulty since the query graph evolves throughout the time, i.e., G_t is not completely known before the time t.

Since searching for the optimal solution is unfeasible, we apply a greedy approach trying to minimize average edge values (i.e., the query times). For this we proposed a combined heuristics of a dense subgraph and the nearest-neighbor strategy. The intuition is to find a subgraph containing short edges between the vertices and process the corresponding query objects in a nearest-neighbor manner, thus minimizing query times. To implement the heuristics, we build hierarchical clusters of the query objects using a pivoting technique. 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 of the query object q to 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 identifies the cluster where the query object belongs. By taking just a prefix of the permutation, hierarchical clustering is obtained. The length of a common prefix of two query objects is used to approximate the corresponding edge value in the query graph.

4.2 Architecture

This section describes the architecture of the system using the proposed technique. Its schema is depicted in Fig. 1.

Let us have a stream $((q_1, t_1), (q_2, t_2), \ldots)$. A query object q_i enters the application at the time t_i, and it is inserted into a component called *buffer*. The buffer is used to temporarily store the incoming query objects which are awaiting processing. This is the component where the query reordering takes place.

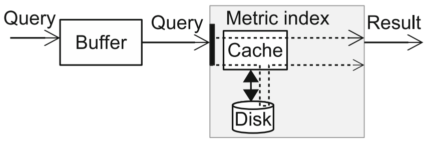


Fig. 1. Architecture

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 which 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 the ordering strategy. During the query processing, the metric index exploits the cache to possibly use any data partitions obtained from evaluating recent queries. If the data are not in the cache, they are loaded from the disk.

5 Parallelization

In general, a way to speed up computer processing is to use parallel computations. In stream processing, the parallelization can be accomplished by creating several instances (replicas) of the processing component. Each data item of the stream is sent to one of the replicas where it is processed. This results in such a scheme where several data items can be processed in parallel. In this section we discuss the applicability of such parallelization method to our case of processing a stream of query objects.

Formally, given the stream $((q_1, t_1), (q_2, t_2), \ldots)$, the number of replicas r and the time limit t, we generate a set of r disjoint paths (i.e., no vertex can be a part of two paths) in the query graph G_t : $\{((q_{i_{11}}, t_{i_{11}}), (q_{i_{12}}, t_{i_{12}}), \ldots, (q_{i_{1k_1}}, t_{i_{1k_1}})), \ldots, ((q_{i_{r1}}, t_{i_{r1}}), (q_{i_{r2}}, t_{i_{r2}}), \ldots, (q_{i_{rk_r}}, t_{i_{rk_r}}))\}$ so that $start_{k_j} < t$ for $1 \leq j \leq r$ where $start_{k_j}$ is the time when the processing of the k_j^{th} query object of the j^{th} replica starts. Each path represents the order of the query objects processed at a particular replica. The goal is to identify such paths which in sum give the largest number of vertices (query objects) to maximize the throughput.

5.1 Parallel Architecture

A generic architecture of the parallel processing system can be seen in Fig. 2. There are several instances (replicas) of the query processor whose task is to evaluate query objects. Each query processor maintains its own cache. During evaluation, the query processor needs to access some data partitions which can be either acquired from the cache or they have to be loaded from a disk. The figure contains just two replicas for simplicity reasons.

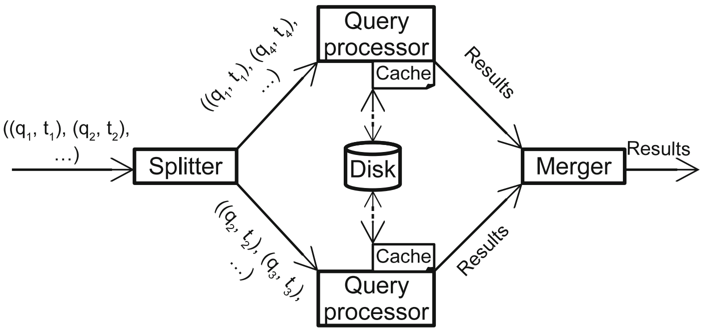


Fig. 2. Generic architecture of parallel processing

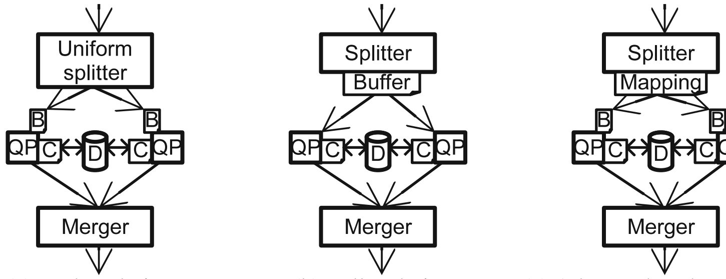
In our experiments, we consider the situation when the database is shared among all the query processors, i.e., they access the same storage space (the disk in the figure). Such a setup allows to quickly add or remove query processors without significant overhead which is advantageous in dynamic environments when it is needed to create or remove replicas on the fly to adapt to load changes.

An important component of the architecture is the splitter which serves as the entry point to the rest of the system. It decides for each query object by which query processor it will be processed. The splitting strategy significantly influences the efficiency of the processing as will be seen in the experiment results. We present three approaches that differ in the functionality of the splitter.

5.2 Push Technique

The first approach is based on a push technique. In this scenario every query processor instance possesses its own buffer of waiting query objects. As soon as a query object arrives at the splitter, it is pushed to the instance of the query processor having the least number of query objects waiting in the buffer. This ensures all the buffers are loaded evenly. The schema is depicted in Fig. 3a. Each query processor continuously evaluates query objects of its own buffer; the same technique for the ordering of the query objects is applied as for the case with a single processor. An advantage of this approach is the low overhead of the splitter, so it can scale well with increasing number of query processors. A disadvantage is that the query graph is not considered for the distribution of query objects among the replicas which can result in ineffective distribution.

Consider two query objects q_1, q_2 needing similar data partitions for their evaluation, i.e., there is a short edge between them in the query graph. If the two query objects are processed by the same replica, the cache may be used to speedup the processing. If each of them is processed by a different replica, the similarity of the query objects cannot be benefit from the cache. This implies that the query objects connected by short edges in the query graph should be processed by the same replica whenever possible.



(a) Push technique (b) Pull technique (c) Advanced push technique

Fig. 3. Replication schemas; QP – query processor, B – buffer, C – cache, D – disk

The dimension of the time plays its role too. Consider two data items (q_1, t_1), (q_2, t_2) where $t_1 \leq t_2$ and a path where q_2 is scheduled for processing at some point after q_1. The bigger the difference between the entry times t_2 and t_1 is, the smaller is the probability that the data partitions needed for processing q_1 are still in the cache when processing q_2 since q_2 may be processed a long time after q_1. This observation is based on the query ordering strategy presented in Sect. 4.1.

5.3 Pull Technique

The disadvantages of the first approach are avoided by the second one which is based on a pull technique. There is one shared buffer for all the query processors which pull query objects from it. The schema can be seen in Fig. 3b. In this case the splitter is in charge of the buffer. A query processor sends a request to the splitter which, according to the ordering strategy, returns a query object from the buffer. The splitter maintains a state for each query processor consisting of the last processed query object by this processor. This is used to enable processing of the query clusters (defined in Sect. 4.1) independently for each query processor. Moreover, the state is used to lock the currently processed cluster. This ensures that a particular cluster cannot be simultaneously processed by multiple query processors. This enables to achieve higher cache utility than in the case of the push technique since all the query objects of the particular cluster which are currently in the buffer are processed by the same replica. Possible disadvantages may occur with a large number of query processors. The splitter then becomes a bottleneck that gets overloaded with requests from the processors. Another possible disadvantage may be when the size of the shared buffer grows over the limits which can fit into the memory.

5.4 Advanced Push Technique

The third approach (advanced push technique) tries to combine advantages of the two previous approaches while mitigating their disadvantages. It is based on a push technique meaning that the splitter actively pushes query objects to the query processors. Each processor possesses its own buffer of waiting query objects. This time the splitter maintains mapping of clusters to individual query processors. This ensures query objects belonging to the same cluster are processed by the same query processor which increases the chances of high cache utility. The schema is depicted in Fig. 3c.

The mapping of the clusters to the replicas works as follows. At first, a predefined cluster hierarchy level $e \geq 1$ is selected. When a query object arrives at the splitter, its pivot permutation is computed, and the prefix of length e is taken representing a cluster. If the cluster is already mapped to a query processor, the query object is sent there for processing. Otherwise the cluster is mapped to the replica having the least number of query objects in the buffer at that moment. The distribution of the streamed query objects among the clusters is not known beforehand and it can change throughout the time. Therefore having such a mapping mechanism may lead to imbalanced load of the replicas. To prevent this, whenever the difference of the maximal and minimal buffer size of all the replicas exceeds a given threshold, remapping of clusters to the replicas is performed to balance the buffer sizes.

The advantage of this technique is that the splitter maintains just the mapping of the clusters to the replicas, and the burden of query ordering is spread among individual replicas. At the same time, the mapping ensures high effectiveness of the distribution of the query objects to the replicas.

6 Experiments

In this section, we provide results of experiments with parallel processing of a stream of query objects using the three approaches presented in the previous section.

6.1 Setup

Let us start with describing the setup of the experiments.

We use the M-Index [15] structure to index the metric-space data. It employs many principles of metric space partitioning, pruning, and filtering, thus reaches very high search performance. The actual data are partitioned into buckets 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 which 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.

The M-Index uses the same set of pivots as are used for the query graph construction. This is beneficial for the effectiveness of the query ordering since the partitioning schema of the metric space used in the M-Index and used for the query graph construction is synergic. This also improves efficiency since 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.

For the experiments, we use the Profimedia dataset of images [5]. We created two different subsets of the images and extracted their visual-feature descriptors. The generated datasets are: 1 million Caffe descriptors [9] (4096 dimensional vectors) and 10 million MPEG-7 descriptors (280 dimensional vectors with complex distance function). Separately, we created streams of images represented by corresponding descriptors. During each experiment, images from the respective collection are continuously streamed to the application and being processed as 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.

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 preserved. This strategy is appropriate since there is a high probability that recently needed partitions will be reused for evaluation of subsequent queries.

All the query processors are run on a single machine in multithreaded environment. For the advanced push technique, the remapping of the subclusters occurs when the ratio of the maximal and minimal buffer size of the replicas exceeds 1.2 (i.e., 20%).

6.2 Evaluation

Fixed Input Frequency. In the first group of experiments, the input frequency of query objects was fixed to 10 ms. This simulates the standard stream processing scenario when the application cannot control the rate of incoming data. The experiments were run with different numbers of query processors: 2, 4 and 8. Each experiment was run for 2 hours for the 1 mil. Caffe dataset and for 4 hours for the 10 mil. MPEG-7 dataset.

The results can be seen in Figs. 4 and 5 depicting the evolution of the buffer sizes during the experiments. For the push techniques, the overall buffer size is taken as a sum of all the buffer sizes at individual replicas. The worst throughput was observed for the simple push technique because of its ineffective distribution of the query objects to the replicas. The other two approaches provide comparable results which are significantly better than those obtained for the simple push approach. Although the advanced push technique distributes the query objects much more effectively than the simple push technique, the pull technique manages to keep the buffer size a little lower most of the time.

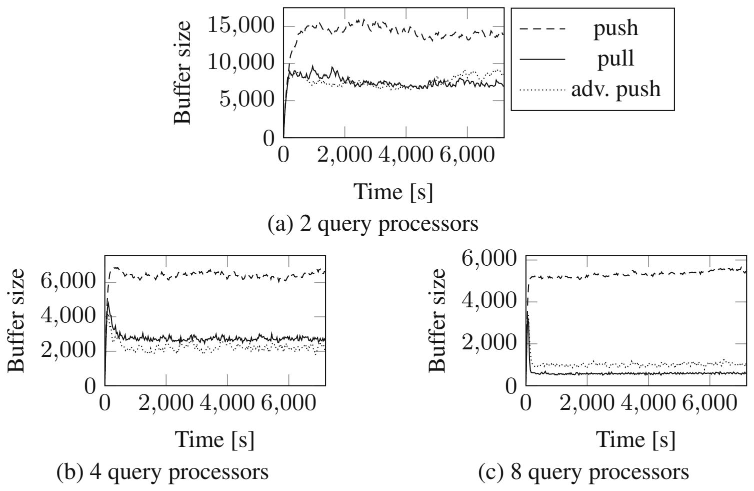


Fig. 4. The buffer size evolution in time during processing of the 1 mil. Caffe dataset

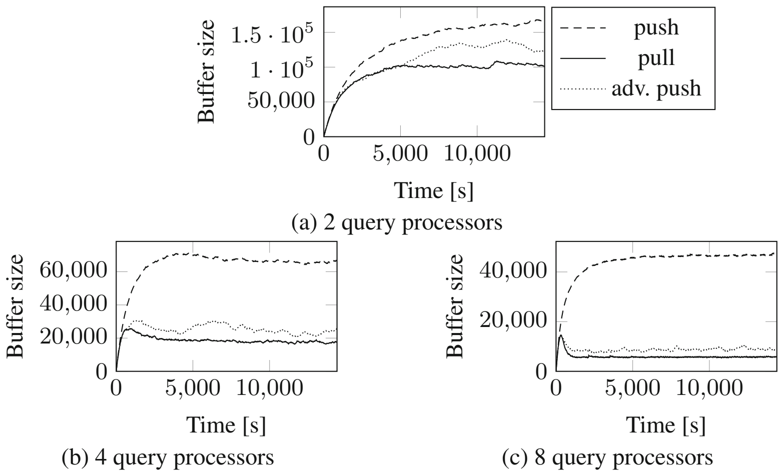


Fig. 5. The buffer size evolution in time during processing of the 10 mil. MPEG-7 dataset

This shows an advantage of the centralized buffer where a better ordering of the query objects can be obtained due to the access to all the buffered query objects rather than just to portions of the data at individual replicas. Another disadvantage of the partial buffers is a fragmentation of clusters among multiple replicas caused by remapping of the clusters to the replicas. On the other hand, if the splitter becomes a bottleneck, the advanced push technique is expected to provide the best performance.

Table 1. Parallelization statistics for the 1 mil. Caffe dataset

	Push			Pull			Adv. push		
# Query processors	2	4	8	2	4	8	2	4	8
Max delay [s]	354	179	90	358	180	90	199	99	81
Median delay [s]	319	140	110	193	100	70	75	23	11
Load difference	1	1	1	0.99	0.99	1	0.98	0.98	0.98

Table 2. Parallelization statistics for the 10 mil. MPEG-7 dataset

	Push			Pull			Adv. push		
# Query processors	2	4	8	2	4	8	2	4	8
Max delay [s]	3356	1440	964	2197	554	327	3131	718	345
Median delay [s]	1300	636	435	917	186	59	1080	255	90
Load difference	0.99	1	0.99	0.99	0.99	0.99	0.99	0.97	0.97

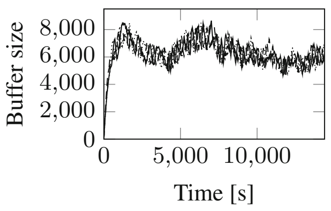


Fig. 6. The buffer size evolution in time per replica for adv. push technique with 4 replicas, 10 mil. MPEG-7 dataset

Tables 1 and 2 present the results considering maximal and median delays (the time since a query object arrives at the application until it is processed). The delays decrease with higher number of query processors. The tables also capture the difference in the number of processed queries by individual query processors computed as $\frac{s}{b}$ where s is the smallest number of queries processed by a single replica; b is the biggest number of queries processed by a single replica. In all the scenarios, the differences in the load of the query processors were negligible.

Due to the nature of the advanced push technique, there can be temporal buffer size imbalances of the individual replicas, and remapping of clusters has to be performed. Figure 6 depicts how the buffer size evolves for individual replicas during the experiment with the MPEG-7 dataset and 4 replicas. Each curve represents the buffer size of one replica. It can be seen the relative difference between the maximal and minimal buffer sizes is kept within the predefined limit of 20%.

Figure 7 shows results of the experiments comparing throughput for high input frequencies (3, 4 and 5 ms). These were conducted for the Caffe dataset using 4 replicas and the pull technique. While the 5 ms input frequency can be coped with, higher frequencies cause the buffer to grow to very large sizes.

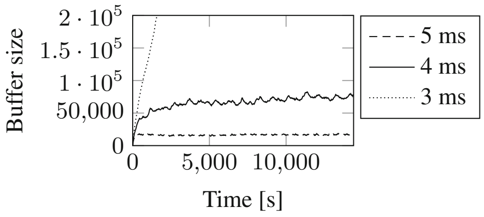


Fig. 7. The buffer size evolution for pull technique with 4 replicas and various input frequencies, 1 mil. Caffe dataset

Table 3. Throughput speedup for parallel processing with no reordering relatively to a single query processor

# Query processors	2	4	8
Caffe	1.90	3.37	5.56
MPEG-7	1.73	3.38	5.83

No Optimizations. In the next experiments, we explore the parallelization impact when no caching and no reordering is used, i.e., the queries are processed in their original order. Whenever a replica is ready for processing, another query object of the stream is pulled and processed. Table 3 shows throughput speedup for different numbers of used query processors. The speedup is computed as the ratio of the average query time (i.e., the time to evaluate a single query object) when just one query processor is used and the average query time when a given number of query processors are used. Since all the replicas access the same database storage, the speedup is not linear. The results serve as the baseline for the following experiments where the optimization techniques are used.

Fixed Buffer Size. Another set of experiments was conducted with a fixed size of the buffer of 10, 000 query objects so that we can compare the throughput of the techniques with a steady state of the buffers. For the push approaches, the buffer size constraint was applied as a limit of the sum of the individual buffer sizes. A next query object was loaded from the stream by the splitter after every processed query object to keep the overall buffer size constant. 100, 000 query objects were evaluated during each experiment. Table 4 captures the throughput speedup (the first three rows). A single query processor with no reordering and no caching is taken as the baseline. The speedup is computed as the ratio of the time needed to process 100, 000 query objects with a single query processor with no reordering and no caching, and the time needed by a given number of query processors with given optimization techniques applied. For the reference, the results contain also the speedup using one query processor with applied reordering and caching (i.e., our original single replica approach). The best speedup can be observed for the pull technique, followed by the advanced push technique. Comparing 8 query processors of the pull technique to a single query

Table 4. Throughput speedup with fixed buffer size relatively to a single query processor with no reordering; the column headers are in the format: # query processors; technique (*PS* – push, *PL* – pull, *AP* – adv. push)

Dataset	Buffer size	1; *PS*	2; *PS*	2; *PL*	2; *AP*	4; *PS*	4; *PL*	4; *AP*	8; *PS*	8; *PL*	8; *AP*
Caffe	10^4	3.7	5.9	6.7	6.5	8	11.2	10.2	8.2	15.6	13
MPEG-7	10^4	2	3.5	4.2	4.2	4.6	6.8	6.4	6.1	10	9
Caffe	$10^4 \cdot$ # query proc.	3.7	6.9	7.8	7.7	11.2	14.1	14.3	15.2	22.8	21
MPEG-7	$10^4 \cdot$ # query proc.	2	4.1	4.9	5.1	6.8	9.7	9.6	9.8	16.3	16.2

processor with applied optimizations, the processing is 4.2 times faster for the Caffe dataset and 5 times faster for the MPEG-7 dataset. One reason of such a rather small factor is sharing the same database storage. Another reason is a small buffer size relatively to the number of query processors which results in small cache utility.

We considered another scenario when, together with the number of query processors, also the buffer size is increased. We used the factor of 10, 000 to set the buffer size; in particular, for 2 query processors, 20, 000 size of the buffer was used, 40, 000 for 4 query processors and 80, 000 for 8 replicas. See the last two rows of Table 4 for the results. For the MPEG-7 dataset, the speedup of the pull technique using 8 query processors compared to one query processor with applied optimizations is 8.1, i.e., better than linear speedup. The buffer size can be observed to be an important aspect of the proposed approaches. (This observation was also made in our previous work for the single processor cases.) Also very small differences between the pull and the advanced push techniques can be noted as the partial buffers of the push technique are of larger sizes than in the previous scenario and so the distribution of query objects among the replicas can be more effective.

To sum up, the distribution strategy of the query objects by the splitter to individual replicas was observed to have a significant impact on the overall throughput of the system. The highest effectiveness of the distribution is achieved by the pull technique. However, it is closely followed by the advanced push technique which eliminates possible scalability issues when a large number of replicas are used. We used an HDD for all the experiments, and different numbers can be expected for an SSD. However, the proposed approaches should still bring improvement due to decompression which has to be carried out when loading the data from the disk.

7 Conclusion

We have presented an extension of the technique for enhancing the throughput of similarity search query processing by dynamic query reordering. The extension lies in the adoption to a parallel environment where multiple queries can be processed simultaneously. We described three approaches to such parallel processing and showed the importance of an appropriate distribution strategy of the query objects to individual query processors.

The best results are achieved with the pull technique when the query ordering is centralized. However, it is closely followed by the advanced push approach when the query objects are intelligently distributed to the query processors and the ordering itself is performed at individual query processors by themselves.

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

References

1. Antol, M., Dohnal, V.: Optimizing query performance with inverted cache in metric spaces. In: Pokorný, J., Ivanović, M., Thalheim, B., Šaloun, P. (eds.) ADBIS 2016. LNCS, vol. 9809, pp. 60–73. Springer, Cham (2016). doi:10.1007/978-3-319-44039-2_5
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., Falchi, F., Zezula, P.: Scalability comparison of peer-to-peer similarity search structures. Future Gener. Comput. Syst. **24**(8), 834–848 (2008)
4. 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). doi:10.1007/978-3-662-46078-8_40
5. 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). doi:10.1007/978-3-642-24469-8_15
6. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: Similarity caching in large-scale image retrieval. Inf. Process. Manag. **48**(5), 803–818 (2012)
7. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. VLDB J. Int. J. Very Large Data Bases **23**(4), 517–539 (2014)
8. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1447–1463 (2014)
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
10. Lakshmanan, G.T., Li, Y., Strom, R.: Placement strategies for internet-scale data stream systems. Int. Comput. IEEE **12**(6), 50–60 (2008)
11. Lakshmanan, G.T., Li, Y., Strom, R.: Placement of replicated tasks for distributed stream processing systems. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pp. 128–139. ACM (2010)
12. Mera, D., Batko, M., Zezula, P.: Towards fast multimedia feature extraction: Hadoop or storm. In: 2014 IEEE International Symposium on Multimedia (ISM), pp. 106–109. IEEE (2014)
13. Nalepa, F., Batko, M., Zezula, P.: Performance analysis of distributed stream processing applications through colored petri nets. In: Kofroň, J., Vojnar, T. (eds.) MEMICS 2015. LNCS, vol. 9548, pp. 93–106. Springer, Cham (2016). doi:10.1007/978-3-319-29817-7_9

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). doi: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., 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, pp. 441–450. ACM (2009)
17. Shao, J., Huang, Z., Shen, H.T., Zhou, X., Lim, E.P., 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). doi:10.1007/978-3-662-49192-8_50
20. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach, vol. 32. Springer, Berlin (2006)