J Supercomput (2018) 74:2006–2034
https://doi.org/10.1007/s11227-017-2207-3

A graph-based cache for large-scale similarity search engines

Veronica Gil-Costa[1,2] · **Mauricio Marin**[3,4] · **Carolina Bonacic**[3,4] · **Roberto Solar**[5]

Published online: 7 December 2017

Abstract Large-scale similarity search engines are complex systems devised to process unstructured data like images and videos. These systems are deployed on clusters of distributed processors communicated through high-speed networks. To process a new query, a distance function is evaluated between the query and the objects stored in the database. This process relays on a metric space index distributed among the processors. In this paper, we propose a cache-based strategy devised to reduce the number of computations required to retrieve the top-k object results for user queries by using pre-computed information. Our proposal executes an approximate similarity search algorithm, which takes advantage of the links between objects stored in the cache memory. Those links form a graph of similarity among pre-computed queries. Compared to the previous methods in the literature, the proposed approach reduces the number of distance evaluations up to 60%.

✉ Veronica Gil-Costa
gvcosta@unsl.edu.ar

Mauricio Marin
mauricio.marin@usach.cl

Carolina Bonacic
cbonacic@usach.cl

Roberto Solar
roberto.solar.gallardo@gmail.com

1 Universidad Nacional de San Luis, San Luis, Argentina

2 CONICET, San Luis, Argentina

3 CeBiB, Centre for Biotechnology and Bioengineering, Santiago, Chile

4 DIINF, Universidad de Santiago de Chile, Santiago, Chile

5 CITIAPS, Universidad de Santiago de Chile, Santiago, Chile

Keywords Approximate similarity search · Metric space cache · Distributed large-scale search engines

1 Introduction

Similarity search in metric spaces has been an efficient approach used to locate user relevant information in large-scale databases of unstructured objects such as text, images, audio, and video [33,58,59]. In contrast to traditional search techniques, metric space similarity search aims to find a set of similar objects for a given user query. Queries are represented by objects of the same type to those stored in the database. In general terms, a metric space consists of a set of objects. To estimate the closeness of objects, their similarity is quantified using a pairwise distance function, which is usually computationally expensive. For complex objects, like fingerprints or biologic components such as DNA and proteins, some authors claim that the distance function is even more expensive than plain disk access [71]. To reduce the number of distance evaluations, a metric index is typically used to prune the search [14,49].

In large-scale search engines where hundreds of queries are processed per second, it is critical to deal efficiently with streams of queries rather than with single queries. These systems are typically deployed on distributed clusters of multicore processors that communicate to each other with high-speed communication networks like the fat-tree [1]. Objects of the database are evenly distributed among processors, usually called search nodes. A set of broker machines are in charge of receiving users queries and distribute them among the search nodes. When a new query arrives, a broker machine sends the query to the search nodes; they process the query by accessing their local indexes and return the answer to the broker machine.

The objects of the database can be indexed by using either the local or global indexing approaches [33,42]. In the local approach, each search node independently builds an index data structure with local objects. Solving a query implies sending it to all the search nodes to get from them their contributions to the final results. In the global approach, instead, a single index is constructed by considering the whole set of database objects to then evenly distribute it onto the processors. The answer to a given query can be completely contained in a small section of the index, and this can potentially reduce the number of processors involved in the solution of each query.

Different techniques have been proposed to enable the realization of efficient metric space search engines. Some examples including efficient storage management with low complexity and high compression techniques [38,55], near-duplicate detection and copy detection approaches [72,73], encryption algorithms [70], and distributed query routing and resource allocation algorithms have been proposed [7,68], among others, that can be applied to improve the performance of metric space search engines.

A critical issue in distributed metric space search engines is to efficiently handle unexpected query bursts. To address this problem, cache memories can be used to store most frequent queries to reduce the traffic directed to the search nodes and to avoid processing expensive distance evaluations by accessing the distributed index. Cache memories are small indexes—smaller than the local index stored in each search node—containing processed queries with its top-k results. There are some caching tools like

Memcached[1] and Redis,[2], but these tools are not designed for metric spaces [12,31]. These are NoSQL databases that use key-value data structures in RAM.

In similarity search engines, it is not trivial to know whether there is a hit cache. That is, to know when the objects retrieved from the cache are the top-k object results for a new query. To alleviate this problem, some approximate metric space algorithms have been proposed [24,25], which guarantee that at least the top-1 object result is retrieved from the cache. However, these approaches can become a bottleneck in large-scale systems where the broker machines have to respond to user requirements in a fraction of a second.

In this paper, we propose a cache strategy for approximate search of objects modeled in a metric space. Our proposal is based on an approximate search algorithm devised for distributed search systems we presented in [32]. It is an iterative algorithm that can be switched to compute approximate or exact answers to metric space queries according to the query traffic rate. In this work, we enable the approximate search algorithm to take advantage of preprocessed queries stored in a cache memory used to create a list of partner objects. The partner objects form a graph that can be traversed to quickly find similar objects to the query and reduce the number of iterations executed by the approximate distributed search algorithm.

The cache implements a very simple but highly efficient data structure called list of clusters (LC) [16], and it can be kept in the broker side or it can be kept distributed in the search nodes. In [43], we have shown that the global index approach is more efficient as the cumulative overheads are smaller than in the local indexing approach. Thus, in this work, the search nodes use a global approach-based index organization. The clusters of the LC are organized into hyper-clusters, which are a kind of hierarchical metadata used to faster prune the index search.

We evaluate our proposal with a query log that emulates the behavior of user submitting queries to a real text search engine. Results show that our strategy reduces by 60% the average number of distance evaluations required to solve queries by delivering approximate object results. Moreover, our strategy is capable of adjusting the level of the quality of the results by specifying the number of exact object results and the number of approximate object results.

The remainder of this paper is organized as follows: Sect. 2 presents the metric space definition and the LC index data structure. In Sect. 3, we review previous related work. Section 4 presents the proposed approximate search algorithm extended with the graph-based cache memory. Section 5 presents experimental results, and Sect. 6 presents the final conclusions and future work.

2 Preliminaries

A metric space (U, d) comprises a universe of objects U and a distance function $d: U \times U \rightarrow R^+ \cup 0$ which fulfills the following properties: $\forall x, y \in U$, (a) strictly positiveness: $x \neq y \Rightarrow d(x, y) > 0$ and $x = y \Rightarrow d(x, y) = 0$; (b) symmetry:

[1] Memcached: https://memcached.org.

[2] Redis: http://redis.io.

$d(x, y) = d(y, x)$; and (c) triangle inequality: $d(x, z) \leq d(x, y) + d(y, z)$. The distance function determines the similarity between two objects. The goal of similarity search is to retrieve all objects close enough to a given query. In this context, a database X is a subset from U. There are two approaches typically used to solve a similarity query:

- Range search $(q, r)_d$: consists of retrieving all objects contained within a ball of center q (query) and radius r, i.e., $RS_X(q, r) = \{\forall x \in X \mid d(q, x) \leq r\}$.
- Nearest neighbor search kNN(q): consists of retrieving the k closest elements in X to a given query $q \in U$. This is, $\mid \text{kNN}(q) \mid = k$, and $\forall x \in \text{kNN}(q), v \in X - \text{kNN}(q), d(x, q) \leq d(v, q)$.

A number of practical index data structures have been proposed in the literature so far [10,13,48]. Their main objective is to reduce the number of distance evaluations computed during the search process. Data structures for indexing metric space databases can be classified into pivot-based, clustering-based, and permutant-based algorithms. Pivot-based algorithms consist of choosing a set of sample objects called pivots. The distances between all objects in the database and the pivots are computed and stored. These pre-computed distances are used to discard database objects by means of the triangle inequality during the query search process [9,11,15,46,53].

Clustering techniques divide the collection of data into groups called clusters such that similar objects fall into the same group. The space is divided into zones as compact as possible. Zones are defined in terms of a reference object called center or centroid [16,19,20,47].

Permutant-based techniques are used for approximate similarity search [4,13]. An approximate answer is formed by those objects, which are close to the current query, but that are not necessarily the k closest ones. Amato and Savino [4] proposed a method using inverted files based on the idea that two objects are similar if they have the same distance to each object of a fixed set. Independently, the authors in [13] presented a similar idea.

2.1 The list of clusters

The list of clusters (LC) is a very simple data structure that has been shown to be efficient in high-dimensional metric spaces [16]. The LC unbalances a tree data structure until it resembles a linked list [65]. This is an asymmetric data structure because clusters are constructed in a sorted-center way, generating a set of overlapped balls. In this work, we use LC because it has shown to be efficient for large-scale similarity search engines [33].

The LC is built as follows. Given a database of objects X, we first choose a center $c \in X$ and a radius r_c. The ball (c, r_c) is the subset of elements of X, which are at distance at most r_c from c. Then, we define $I = \{\forall u \in X \mid d(c, u) \leq r_c\}$ as the bucket of internal objects (all objects which lie within the ball centered at c with radius r_c), and $E = \{\forall u \in X \mid d(c, u) > r_c\}$ as the set of external objects. This process is recursively repeated over E until it is empty.

There are two simple ways to divide the space: using a fixed radius for each partition or using a fixed cluster size. By using a fixed radius, all clusters have the same cov-

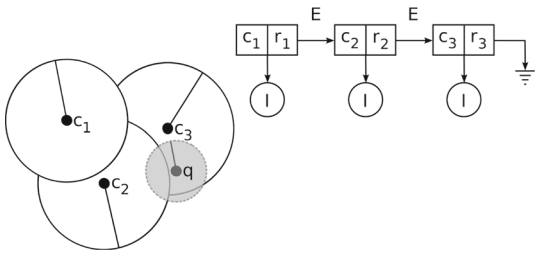

Fig. 1 The influence zone of centers c_1, c_2 and c_3

ering area. However, some clusters store more internal objects than others, producing unbalanced workload among the processors that hold the distributed index. By using clusters of fixed size (same number of internal objects), each cluster covers areas of different size, but the amount of distance evaluation performed within each cluster is the same. To ensure good load balance across processors, we consider partitions with a fixed size of k elements; thus, the radius r_c is the maximum distance between the center c and its k-nearest neighbor.

The final LC structure is a linked list of clusters, in which each cluster is composed of a triplet (c, r_c, I) (center, radius, bucket). During the construction stage, a center chosen first has preference over the following ones when the balls are overlapped. Figure 1 shows an example of this process where the center chosen first has preference over the following ones, generating a set of overlapped balls. All objects that lie within the ball of the first chosen center are stored in its bucket, despite that they may also can be overlapped by other clusters.

In [16], different heuristics have been evaluated to select the centers of the clusters, and it has been experimentally shown that the best strategy is to choose the next center as the one that maximizes the sum of distances to previous centers. Thus, in this work, we use this heuristic to select the centers of the LC. Additionally, to reduce the number of distance evaluations at query time, we keep the distance from the center to all the objects stored in the bucket. That is, inside the cluster with center c_i and bucket I_i, we record the distances $d(c_i, u) \forall u \in I_i$ [44].

In Algorithm 1, we show the steps executed by the search process of the LC. To search for a query q with radius r, we compute the distance between the query and the centers of the LC. In line 1, we traverse the clusters of the LC. In line 2, we check whether $d(q, c) \leq r$. If so, we add c to the set of object results. Also, in line 5, we verify that $d(q, c) \leq r_c + r$, it means that the query ball (q, r) intersects the center ball (c, r_c), and we scan exhaustively the cluster I to search for similar objects to the query. This process continues with the next cluster in the LC. However, if the query ball is completely contained by the cluster (i.e., in line 8, $d(q, c) \leq r_c - r$), we stop the search before traversing the whole index. This is possible because the construction process ensures that all the objects that are inside the query ball (q, r) have been inserted in I.

Figure 1 also shows three overlapped clusters, in which the construction process takes into consideration the following order: c_1, c_2, and c_3, and a query q with radius r. First, the search algorithm computes the distance between q and c_1 and determines there is no intersection ($d(q, c_1) > r_1 + r$). Second, the search algorithm computes

the distance between q and c_2, and since there is intersection ($d(q, c_1) \leq r_1 + r$), we search inside the bucket for similar objects to q. As we keep the pre-computed distance between the objects in the buckets and the center object, we use the triangle inequality to discard nonsimilar objects and reduce the number of distance evaluations. Finally, the search algorithm repeats the same process over the cluster centered at c_3. Note that objects found at the intersection of clusters c_2 and c_3 are stored only into c_2, so the algorithm does not provide duplicate results.

The k-NN search process is similar to the range search described above. However, at the beginning of the search process, the query radius is set to infinite $r = \infty$. We compute the distance between the query and the center of the first cluster $d(q, c_1)$, and we search inside its bucket for similar objects. We build a set of object results sorted from the closest one to the k-NN. The distance between the query and the k-NN object gives the new query radius r. This process is repeated with the following clusters of the LC. When $d(q, x) < r$, we remove the k-NN object, we insert the new object to the set of results, and we update the query radius.

Algorithm 1 Search(LC, q, r)

```
1: for (c, r_c, 𝕀) ∈ LC do
2:    if distance(q, c) ≤ r then
3:       results ← results ∪ c
4:    end if
5:    if distance(q, c) ≤ (r_c + r) then
6:       search exhaustively into 𝕀
7:    end if
8:    if distance(q, c) ≤ (r_c − r) then
9:       break
10:   end if
11: end for
```

3 Related work

In this section, we describe approaches related to the proposal of this article. First, we review approaches devised for approximate search in metric spaces. Second, we review approaches devised for metric space caches.

3.1 Approximate metric space algorithms

Similarity search has shown to be a powerful tool for handling large collections of unstructured data. This approach is very useful in a wide range of applications, such as data mining, pattern recognition, multimedia data retrieval, computer vision, computational biology. Typically, the features of the objects of interest (documents, images, etc.) are represented as vectors in R^d, and a distance metric is used to measure (dis)similarity of objects [36]. Features vectors can vary from hundreds to thousands of dimensions. Therefore, we face up to the so-called curse of dimensionality effect, i.e., the performance of indexing and searching algorithms tends to rapidly decrease when the dimension of features vectors (and the size of database) is increased.

Approximate nearest neighbor (ANN) algorithms try to overcome the following challenges: (a) scalability problems in large-scale databases and (b) the curse of dimensionality effect. An approximate answer consists of objects that are close to the current query, but not all of them are the k closest objects. Usually, the quality of approximate answers is measured with two metrics: (1) *recall*—ratio of the number of relevant retrieved objects to the total of relevant objects and (2) *precision*—ratio of the number of relevant retrieved objects to the total of objects retrieved.

In [28], the authors classify general approximate k-NN approaches into two categories:

- *Retrieved set reduction* Organizes the database in such a way that only a subset of objects are examined during the nearest neighbor search. Some examples of this approach include: region shape refinement [57], perspective-based space transformation [4], probabilistic algorithms [18], locality sensitive hashing [5], etc.
- *Representative size reduction* Organizes the representatives of the database in such a way that only a partial representation of each object is examined during the nearest neighbor search. Typical examples of this approach are dimensionality reduction [21,54] and VA-based indexing [27,66,67].

Ferhatosmanoglu et al. [28] presented a new technique that combines the advantages of the two categories. On the one hand, to reduce the retrieved set of objects, they apply the well-known k-means clustering technique [39,41]. On the other hand, to reduce the representative size of objects, they transform each object by using the Karhunen–Loeve transformation (KLT). Good surveys on approximate nearest neighbors searching algorithms can be found in [69,71].

Arya et al. [6] presented the Balanced Box-Decomposition (BBD) tree. A BBD-tree is a data structure for approximate range search on the basis of the relative error model. The BBD-tree is a binary tree constructed by means of two operations: *split*—by partitioning a cell by an axis-orthogonal hyperplane and *shrink*—by partitioning a cell by a box that lies within the original cell. The presented algorithm aims to find the $(1+\epsilon)$-*approximate nearest neighbor*. Given an error $\epsilon > 0$ and a query q, if $d(q, p) \leq (1+\epsilon) * d(q, p')$, then p is within a factor of $(1+\epsilon)$ of the true nearest neighbor p'.

As described before, Amato and Savino [4] presented a technique for ANN searching based on the idea of: if two objects o_i and o_j are similar, then their view of the surrounding world is similar as well. Therefore, if we take a set of reference objects from the database and we order them depending on their distance to o_i and o_j, then the sorted sets are also similar. Chávez et al. [13] introduced a new probabilistic ANN searching algorithm for metric spaces. The main idea is to predict the proximity between two objects according to how they order their distances toward a set of anchor objects called permutants. Each object is associated with a graph of permutants sorted by their distance to the object. To process a new query, the searching algorithm computes the distances between q and the permutants and builds a signature formed by the lists of permutants sorted by their distance to q. Then, the signature of the query is compared with the signature of the objects in the database. Notice that comparing two signatures is less expensive than computing a distance evaluation [13].

Skala [61] evaluated the number of possible permutations that can occur in a given space. Esuli [22] presented an image retrieval system named MiPai. This system provides visual similarity search and text-based search functionalities. Later, the same author presented in [23] a permutation-based data structure called Permutation Prefix Index (PP-Index). The PP-Index uses permutation prefixes to quickly select a small set of candidate objects that could be close to the query, and next, it is computed the distance between the query and each candidate object to determine the closest ones. Dataset objects are kept on secondary memory. For each permutation object, the PP-Index selects a prefix of fixed size. These prefixes are indexed in a main memory tree-based data structure. The leaves keep the information required to retrieve the disk blocks relative to the objects represented by the permutation prefixes obtained in the path of the tree. Permutant-based indexes have been also used for ranking diversification in [34].

Gennaro et al. [30] proposed a permutation-based image retrieval system for approximate similarity search developed on the basis of the full-text retrieval library Lucene.[3] In [40], is also presented an image retrieval system based on Lucene called LIRe. Novak et al. [51] proposed a distributed index structure for similarity data management called Metric Index (M-Index). This idea was originally presented in [50]. The M-Index is a general mapping mechanism that enables to store data into well-known data structures and still exploit the advantages of metric space properties. They also introduce an approximate search strategy based on permutations. Nevertheless, we have to take into account that permutation-based indexing does not guarantee exact results and neither the k closest objects for a given query as required in this work.

The authors in [2] and later in [3] compared five pivot selection techniques on three permutation-based similarity access methods. The authors conclude that the pivot selection technique should be considered as an integrating and relevant part of any permutation-based access method. Similarly, Figueroa and Paredes [29] proposed to improve the pivot-based index by using less space than the basic permutation technique and also using additional information to discard some objects. Novak and Zezula [52] proposed a mapping of objects from a generic metric space onto main memory codes using pivot spaces to reduce the candidate set size. That is the number of objects not discarded by the triangle inequality.

3.2 Metric space cache

A metric space cache C on a metric space (U, d) and database $X \subseteq U$ consists of a set of past queries together with their respective k results. That is, $q_i \in C$ if the query and its results in $\text{kNN}_X(q_i, k)$ are in the cache. Also $o_i \in C$ denotes that the object $o_i \in X$ is stored in the cache and thus belongs to at least one set $\text{kNN}_X(q_i, k)$ associated with a cached query q_i. More formally:

Definition 1 Let (U, d) be a metric space, let $X \subseteq U$ be a database, let C be a metric space cache on (U, d), let $q \in U$ and $q \in C$ be a query in the metric space cache, and let $k \geq 1$. Then, r_q denotes the radius of the smallest hyper-sphere centered in q,

[3] http://lucene.apache.org.

which contains all objects in $\text{kNN}_X(q, k)$. We assume that the k nearest neighbors are always unique.

The works in [24,25] define the *safe radius* s_q of the query q with respect to a query $q_i \in C$ as the radius r_{q_i} minus the distance from q to q_i, namely $s_q(q_i) = r_{q_i} - d(q, q_i)$. It holds that for $k' = \mid R_C(q, s_q(q_i)) \mid$ (note that $k' \leq k$), $R_C(q, s_q(q_i)) = R_X(q, s_q(q_i)) = \text{kNN}_X(q, k')$.

Note that every query $q_i \in C$ gives complete knowledge of the metric space up to the distance r_{q_i} from q_i. If a query q is inside the ball centered in q_i with radius r_{q_i}, then as long as we restrict ourselves to look inside this ball (q_i, r_{q_i}), we know the $k' \leq k$ nearest neighbors of q. Thus, for a given query $q \in U$ and $q_i \in C$, if the safe radius $s_q(q_i) > 0$, then every object in the range query $R_X(q, s_q(q_i))$ is also in the cache C and we can solve the query q using the objects of the cache with the range query $R_C(q, s_q(q_i))$. Furthermore, the k' objects in $R_C(q, s_q(q_i))$ are also the k' nearest neighbors of q in the database X. We use this safe radius definition in the next section.

The works in [24–26] presented two different metric space cache algorithms, named Result Cache (RCache) and Query Cache (QCache). Both algorithms use a hash table H for storing query objects and their respective results. They have been developed for retrieving approximate results, and their implementation is based on the M-Index. The RCache indexes the object results of the queries stored in H. The QCache indexes the query objects.

Chierichett et al. [17] presented a theoretical analysis of metric space cache. The authors consider that a cache hit occurs when the distance between the query and an object in the cache is lower than a given threshold. This work is also applied to contextual advertising systems [56] by using the locality sensitive hashing (LSH) [36].

A caching metric space index called D-Index is presented in [63]. The D-Index keeps pre-computed distances. The D-Index is implemented by using a hash table with entries $[o_1, o_2, d(o_1, o_2)]$, where o_1 and o_2 are the object identifiers and the third component is the computed distance between the objects. The D-Index is kept in main memory to reduce the number of distance computations performed over a second index like the M-tree [19]. This goal is achieved when $d(o_1, o_2)$ is in the D-Index, or by obtaining a lower or upper bound of $d(o_1, o_2)$ and thereby improving the pruning over the M-tree.

Brisaboa et al. [8] reuses distance evaluations involved in cache miss operations to produce good approximate results. If a cache miss occurs, the distance evaluations performed by the cache lookup procedure are used for traversing the index as fast as possible. Later, Solar et al. [64] proposed a two-level metric space cache. The first level is for static cache, and the second level is for dynamic cache. Different strategies are evaluated to improve the efficiency and the effectiveness of the metric cache.

A cache structure named Snake Table was proposed in [60]. The index is created and updated online, and it is devised for supporting streams of kNN searches. More recently, Matej and Vlastislav [45] presented the Inverted Cache Index technique which records the number of times a given object contributes to the final result of queries. Each object has a memory of its historical accesses. The technique was evaluated on the M-tree and the M-Index.

4 Approximate metric space search algorithm

The proposed approximate search algorithm is designed to achieve the top-k objects by traversing the LC index, and the desired level of approximation is a parameter that can be dynamically set in accordance with the observed query traffic. Under a high query traffic, the search nodes work to retrieve from the globally distributed LC [43] a total of M approximate results, where k of them are the exact closest objects to the query and the remaining $M - k$ ones are approximate results.

The approximate similarity search algorithm works as follows: (1) We first determine the cluster (c_i, r_{c_i}) containing the query object q. We call it the *main cluster*. (2) Then, we compute the safe radius $s_q = r_{c_i} - d(q, c_i)$. Next, we search inside the main cluster for similar objects that lie inside the query ball (q, s_q). (3) To ensure the k closest objects to the query, we visit the clusters built before c_i that overlap the query ball. (4) If we do not find the k closest objects, we increase the safe radius by a factor of γ. (5) We check whether the clusters built before and after c_i intersect the new query ball, and we search for similar objects inside those clusters. Objects processed previously are not considered. Objects not included in the top-k results are placed into a set of approximate results. The algorithm iterates from step 4 until we obtain the k closest objects to q. The remaining $M - k$ objects are selected from the set of objects that have been found to be close enough to q up to this point. Notice that we perform approximate similarity searches when $k < M$, and we retrieve the exact nearest objects when $k = M$.

If we already have the k closest objects to the query but we are missing some of the $M - k$ approximate objects, say $M - k = 2$, our algorithm iterates once more. To this end, we increment the query radius using the γ factor, and we apply the triangle inequality to select the candidate approximate objects. For a set of $A > 2$ candidate objects, our algorithm selects two objects at random no matter their distance to the query object. In this stage, no distance evaluations are performed. Notice that this last step can reduce the quality of results. However, we show in the experimental section that the error introduced in the set of results is negligible in comparison with the improvement in performance.

Figure 2 shows an example of index LC with five clusters $\{(c_1, r_1), (c_2, r_2), (c_3, r_3), (c_4, r_4), (c_5, r_5)\}$ and a query q contained within the cluster (c_4, r_4). We compute the distance between q and the centers preceding to c_4 (i.e., centers c_1, c_2 and c_3). When we get to c_4, we realize that q lies inside this cluster. Distances to previous clusters are stored for later use. Then, we compute the safe radius s_q and we check whether some previous clusters intersect (q, s_q). In this example, there are no clusters before c_4 intersecting the query ball.

To obtain the k nearest objects to q, we increase s_q by a factor of γ. The new query ball intersects the cluster (c_5, r_5), so we have to compute the distance $d(q, c_5)$ and we obtain the new objects falling inside the query ball. Moreover, we have to reenter into (c_4, r_4) and search for objects falling inside the new query ball. We reuse the pre-computed distances. The second time we increase the query radius, we have to go inside (c_1, r_1). We also have to reenter into clusters c_4 and c_5 to ensure the exact top-k object results.

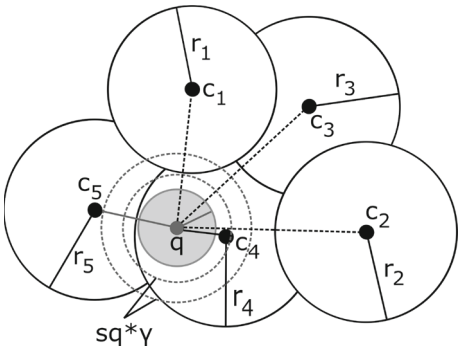

Fig. 2 Approximate similarity search algorithm. The query object is contained by the main cluster c4. The algorithm iterates twice until it finds the exact top-k results

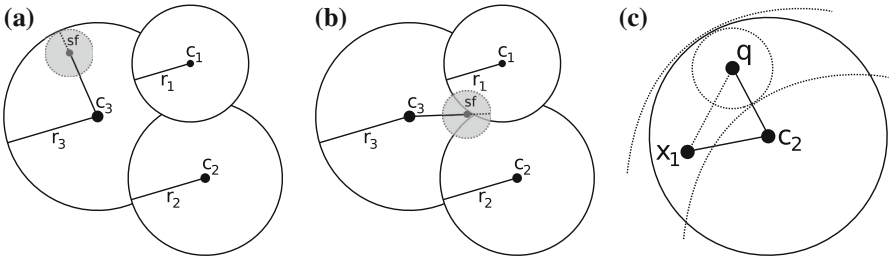

Fig. 3 Approximate similarity search algorithm: **a** the safe radius does not overlap previous clusters, **b** the query ball overlaps two previous clusters, **c** add x_1 into the set of approximate results

Figure 3a, b shows an LC index with three clusters. The identifiers of the centers describe the order of construction of the clusters. Figure 3a shows that the query ball (q, s_q) is completely contained inside the main cluster c_3. If we find k objects inside that ball, we can ensure that they are the k nearest objects to the query due to the construction order. In Fig. 3b, we have to visit c_1 and c_2 to guarantee the k nearest objects to the query. In Fig. 3c, the search algorithm cannot discard the object x_1 using the triangle inequality, but when we compute the distance $d(x_1, q)$, we realize that x_1 is not similar to the query (it is outside the query ball), and therefore it is included in the set of approximate results. The algorithm stops when we obtain the k closest object to the query and the $M - k$ approximate objects.

4.1 Graph-based cache for metric space search

Our graph-based static cache strategy works in tandem with the approximate search algorithm we have devised for the LC index distributed in hyper-clusters [43]. We keep pre-computed queries with their object results organized as follows. After retrieving the Q most frequent queries from a large query log, we select the top-n results for each query and set links among them. The result objects retrieved for each query are linked as partners objects as shown in Fig. 4 at top. We have horizontal and vertical links. The horizontal links connect objects belonging to the exact results of a query q, where the linked objects are the n closest ones to q. Vertical links are used to link

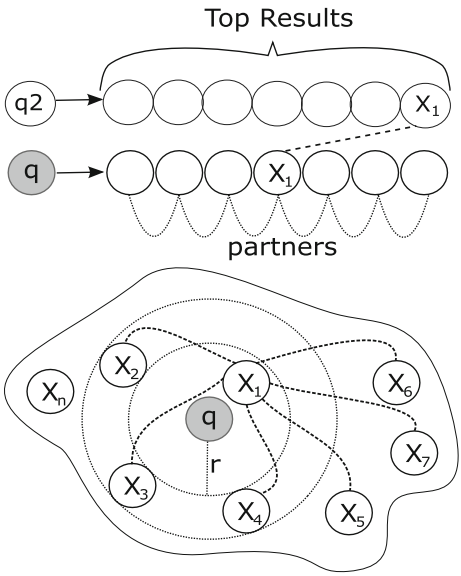

Fig. 4 At top: query results are linked to each other to form the group of partner objects. At bottom: when processing a new query (q, r), only one object x_1 is within the query radius r. The remaining objects can be retrieved through the graph of partners

the same objects reported as results for different queries. For instance, in Fig. 4 at top, the object x_1 belongs to the results of queries q and q_2, and therefore a vertical link is created to connect the two instances of the same object. Experimentally, we have found $n = 10$ to be a good trade-off between space used and performance gain.

The graph of partners is stored in a cache memory, and it is used to process new queries. To this end, we first execute the approximate search algorithm described above to determine the main cluster containing the query object, and we compute the safe radius $s_q > 0$ for the query. If there is a cached object within the query radius, we add this object as part of the results and we search for its partners. We compute the distance between the query object and the partner objects. Then, we add the partner objects to the query result set until we reach the top-k object results and we update the query radius. If there is no object within the query radius, we use by default the γ factor to increase the query radius and the algorithm advances to the next iteration.

The links created among the cached objects form a graph, which can be traversed to quickly find similar objects to the query. Notice that when we find a cached object within the query ball, the query radius is automatically updated without the use of the γ factor. This allows to reduce the number of iterations we have to perform to retrieve the top-k object results.

Figure 4 shows an example of the search process using the partner algorithm. In this example, the object x_1 lies inside the query radius. The object x_1 has a graph of partners. Then, the algorithm computes the distance $d(q, x_i)$ for each object x_i in the list. So, first we obtain $d(q, x_2)$ and we increase the query radius. At this point, we add the objects x_2, x_3, and x_4 as part of the results. If $k = 4$, we add x_5, x_6, and x_7 as approximate results. Clearly, this algorithm helps to reduce the number of iterations required to obtain the exact top-k answers. Nevertheless, the effectiveness can be affected as we could decrease the quality of approximate results. In this example, x_n

is not included as part of the approximate result set, despite it is closer to the query than the remaining objects x_5, x_6, and x_7.

Our proposal is devised to drastically reduce the number of distance evaluations at the cost of losing quality in the results. Moreover, in the cache memory, once we extend the query radius beyond the safe ratio, we cannot guarantee that the objects retrieved are exact results. However, in the next section, we show that the proposed graph-based cache is capable of retrieving good quality results.

In general, during query processing, partner objects are retrieved from the cache and selected to complete the $M - k$ approximate results in a FIFO manner. This approach is simple and fast. On the average, at the front of the respective queue, there is a high probability of finding objects that are closer to the query than objects selected at random from the same queue. The reason is the way the underlying approximate LC algorithm detects partner objects to build the queue. As shown in our experiments section, this produces good quality approximate results for most queries. However, there may exist a few queries for which additional processing is required to improve their approximate results.

It follows that a further refinement of our proposal is to discard partner objects that are detected to be too far from the search query. This at the cost of extra computations and space used in the cache, but without resort to computing (expensive) distance evaluations among partner objects and the search query. To this end, we use the moving average of the search radius of processed queries r_{avg}. For each query q_c in the cache, we keep the distance among the exact object results o_i and the query q_c (i.e., $d(q_c, o_1) \ldots d(q_c, o_M)$). Then, at running time and for a given search query q, if the algorithm detects an object o_i within the current search radius where o_i is also stored in the cache for another query q_c, then the algorithm may discard objects o_j that are linked to o_i in q_c whenever $B(q, o_j) > r_{\text{avg}}$ where B is a lower bound for $d(q, o_j)$, defined as $B(q, o_j) = | \, d(q, o_i) - | \, d(q_c, o_j) - d(q_c, o_i) \, | \, |$.

The experimental results show that the above refinement is convenient for improving the quality of approximate results for individual queries that depart from the observed average quality, with the advantage of slightly reducing distance evaluations but at the cost of increasing the space used in the cache.

Alternatively, our approximate similarity search LC algorithm may be allowed to perform additional iterations, each time complementing with results from the partners cache, until we get a predefined quality upper bound. This implies the computation of additional distance evaluations among objects. In this case, the user may define as an acceptable result quality to be a situation in which the current query search radius becomes a factor ≥ 1 below the average radius of previously solved queries containing M exact results. In this case, the current query search radius is the current M-closest approximate result object.

The experimental results show that the strategy of performing additional iterations is convenient for increasing the probability of including more objects belonging to the exact kNN results in the $M - k$ results. This in turn leads to an improvement in the quality of query results at the cost of executing more distance evaluations per processed query.

In the next section, we evaluate the different trade-offs and compare our proposal against alternative approaches.

5 Experimental results

In this section, we present experimental results to evaluate the performance of our proposal. Experiments were performed using an image collection with 10,000,000 objects crawled from the Flickr system to build the index. Each image contains five MPEG-7 visual descriptors [62] (Scalable Color, Color Structure, Color Layout, Edge Histogram, Homogeneous Texture).

We generated a synthetic query log using information made publicly available by Flickr, where for each image, we know the number of times it has been seen by users, which is a measure of its popularity with respect to the other images in the collection. To simulate a real stream of queries, we employed a Yahoo log that corresponds to a three-month period in 2009; this log contains 2,109,198 distinct queries. The query term space spans a vocabulary of 239,264 different terms. We select queries with one-term and associated popular images with popular terms found in the log. In this way, we simulate the order in which the queries are issued by the metric space search engine users. In the following experiments, we execute a total of $Q = 10{,}000$ queries. The distance between two images is evaluated with the Euclidean distance over each MPEG-7 descriptor.

Experiments were executed on a cluster with $P = 32$ processors with 64-bit Intel Core Quad Q9550 2.83 GHz processor and 4 GB RAM DDR3 1333 MHz each node. To better illustrate the results, we show results normalized to 1. To this end, we divide all quantities by the observed maximum in each case.

We compare our graph-based cache proposal against two approaches for our approximate search algorithm on the LC index. The first one, named LC in the figures shown below, uses the original construction algorithm with the heuristic proposed in [16] to select the centers of the clusters. The second index, named LC-Log in the figures shown below, uses information from previous queries stored in a query log to select the centers of the clusters. To this end, we compute the K nearest objects for each query object present in the query log, and then we select from this set the centers of the clusters by using the same heuristic presented in [16]. Therefore, the second method exploits information about frequently queried objects to build the index. This can be of some benefit as clusters are visited in construction order during query processing. In a way, this simulates a Least Frequently Used (LFU) cache upon the LC index and may be considered as a competitor trying to use a similar strategy than our graph-based cache. In the experiments, we set the static cache to store a 6% of the query log with their top ten result ID objects. Notice that, like the graph-based cache, the LC-Log index can be periodically updated to keep the most popular objects as centers of clusters.

5.1 Performance evaluation

Figure 5 shows the normalized number of distance evaluations (y-axis) performed by the strategies LC and LC-Log operating under different search radius increment γ values (x-axis). A value close to 1 in y-axis represents a greater number of distance evaluations and therefore a worse performance. Each curve of the figure represents

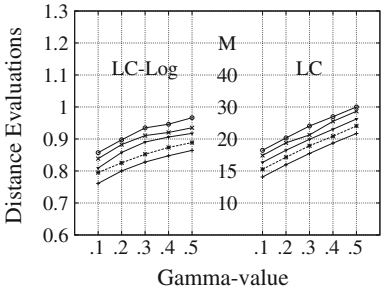

Fig. 5 Normalized number of distance evaluations (y-axis) obtained by our baseline approximate similarity search system deployed on a cluster of $P = 32$ processors and operating under different search radius increment rates (x-axis). The number of object results ranges from $M = 10$ to $M = 40$ where the closest $k = \frac{1}{2}M$ results must be exact ones. There is one curve for each value of M

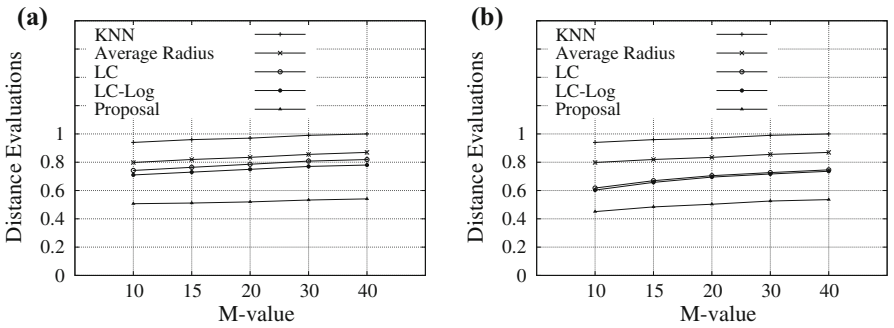

Fig. 6 Normalized number of distance evaluations for different number of retrieved objects M per query. **a** Using $k = M$, **b** using $k = \frac{1}{4}M$ where each query result set contains the k exact closest objects to the query

the value of total number of object results (M) retrieved by each query. For example, the curve with white circle corresponds to $M = 40$. We set $k = \frac{1}{2}M$ as the number of exact object results retrieved by each query. As expected, the computing cost increases with the γ factor. With a high γ factor, we have to perform more distance evaluations because the query radius grows faster and therefore more objects must be compared against the query. Below 0.1, the results do not show significant improvement as the size of the new query radius tends to be similar to the size of the current query radius. Therefore, we require more iterations to find objects nearby the query, but the number of distance evaluations is not significantly affected. The results for LC-Log show that taking into consideration skewing in object retrieval has a positive effect in performance.

Figure 6 shows the number of distance evaluations (y-axis) achieved by (1) the standard kNN algorithm described below, (2) the same standard kNN algorithm but using the average query radius as its initial search radius (named Average Radius), (3) the approximate LC algorithm, (4) the approximate LC-Log algorithm, and (5) the approximate cache of partner algorithm proposed in this work and implemented over

the LC-Log index. The x-axis shows different values for the total number of retrieved objects M per query.

The standard kNN algorithm works as follows. The initial search radius is set to infinite. Then, as we process the LC clusters using the original LC exact search algorithm [16], the search radius is adjusted to be the distance to the current kth object. Objects compared against the query and not included in the set of results are considered as approximate results. The Average Radius algorithm works the same way but the initial radius is set to be the average query radius observed from a set of previous queries.

In Fig. 6a, we show the number of distance evaluations performed when $k = M$. Namely, we do not retrieve approximate object results. In general, as we increase the number of object results M, the number of distance evaluations increases by 5% on average. The average radius kNN algorithm reduces by 10% on average the number of distance evaluations performed by the standard kNN algorithm. Our LC and LC-Log iterative algorithms (with query search radius increment $\gamma = 0.1$) outperform the standard kNN search algorithm by 25% on average. The LC-Log presents better performance than the LC algorithm by 5% at most. However, our graph-based cache proposal working in tandem with the LC-Log algorithm reduces by 45% the number of distance evaluation performed by the kNN algorithm and by 20% on average the number of distance evaluation performed by the LC-Log algorithm alone. These results show that even in the case when we retrieve the exact top-M object results, our proposal can significantly reduce the total number of distance evaluations required to solve queries. In other words, using links to similar objects retrieved from previous queries is useful to avoid computing distance evaluations with dissimilar enough objects. These links allow to quickly jump to clusters storing relevant objects for the queries.

In Fig. 6b, we set $k = \frac{1}{4}M$. In this figure, the results reported by the LC and the LC-Log algorithms are very similar. The difference is 2% at most. Additionally, the LC and the LC-Log algorithms outperform the kNN and the Average Radius algorithm on average by 28 and 17%, respectively. The results also show that our proposal outperforms the kNN baseline algorithm by more than 49% on average. Our proposal performs slightly better in $k = \frac{1}{4}M$ than in $k = M$ where the difference is more significant for small M.

In Fig. 7a, b, we show the number of distance evaluations using search radius increments $\gamma = 0.5$ and $\gamma = 0.1$, respectively. Both figures show the results obtained by LC, LC-Log, and our proposal when we retrieve the $\frac{1}{4}M$, $\frac{1}{2}M$, and $\frac{3}{4}M$ closest objects to the queries for different number of retrieved objects M. As expected, the number of distance evaluations is reduced as we relax the number of exact answers. That is, we compute less distance evaluations to determine the top-k closest object results.

Notice that when we use a smaller value of $\gamma = 0.1$, the LC and the LC-Log indexes have similar performance. This is because the search radius of the query is gradually enlarged avoiding comparing the query against dissimilar enough objects. With a larger γ value, we find more objects in each iteration and therefore the number of distance evaluations tends to be larger. In any case, our proposal outperforms the

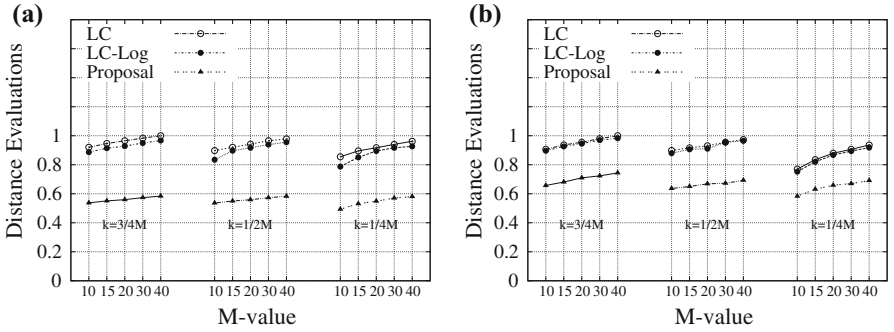

Fig. 7 Normalized number of distance evaluations for different $k < M$, where M is the total number of retrieved objects and k is the number of exact objects that are the closest ones to the query (the remaining $M - k$ objects are approximate results for the query). **a** Using search radius increment $\gamma = 0.5$, **b** using search radius increment $\gamma = 0.1$

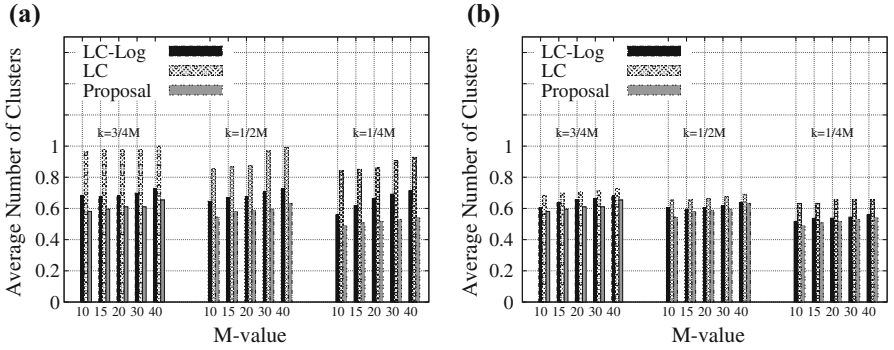

Fig. 8 Normalized average number of LC clusters visited by all queries. **a** Using search radius increment $\gamma = 0.5$, **b** using search radius increment $\gamma = 0.1$

LC and the LC-Log indexes by 20% on average when using $\gamma = 0.1$ and by 30% on average with $\gamma = 0.5$. These results show that our proposal can significantly reduce the number of distance evaluations reported by the other algorithms.

Figure 8a, b shows the average number of clusters visited by all queries in order to retrieve the k closest objects. Both figures show results normalized by the same maximum value obtained with the configuration $\gamma = 0.5$, $k = \frac{3}{4}M$, and $M = 40$. We can see in both figures that the LC index requires to visit more clusters than the LC-Log index. The LC-Log visits 30% fewer clusters than the LC. In Fig. 8b, we show that by using a smaller γ value, the number of clusters visited by all queries is reduced. Thus, as explained before, using a smaller γ value allows to improve the prune when traversing the index. Moreover, the difference between the LC and LC-Log indexes is smaller in this last figure. The LC-Log selects the centers of the clusters from previously processed queries, which helps to have representative objects as centers. Finally, our proposed cache-based algorithm reports the least number of clusters visited. Our algorithm performs a more aggressive pruning, which is directly related to the number of distance evaluations reported in Figs. 6 and 7.

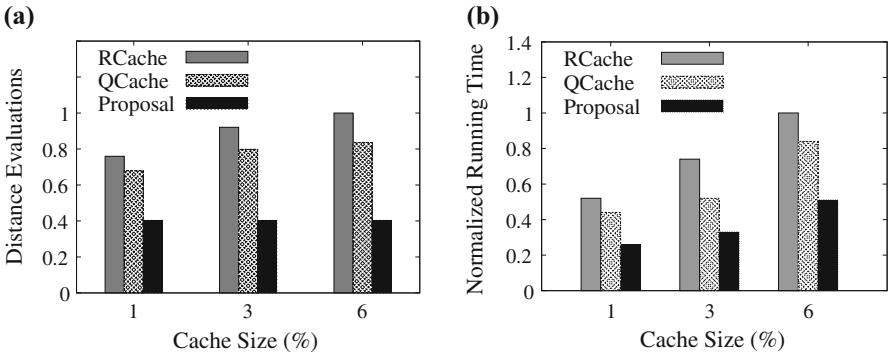

Fig. 9 Evaluation of the QCache and RCache approaches versus our proposal with different cache sizes. **a** Normalized number of distance evaluations, **b** normalized running time

Figure 9a compares the performance of the RCache, the QCache algorithms [25], and our proposed cache-based algorithm running on the broker machine. In this experiment, we set the cache size to be 1, 3, and 6% of the total number of queries and we deploy the graph-based cache in a single processor. We set $k = 1$ and $M = 20$. As we increase the cache size, the number of distance evaluations performed by the RCache and QCache algorithms tends to grow because the metric index used by these algorithms also grows. Thus, a larger metric index requires more distance evaluations. With a cache size of 6%, our proposed algorithm reduces by 60% the number of distance evaluations.

Figure 9b shows the normalized running time reported by the three algorithms. This figure shows that the number of distance evaluations has a direct impact on the running time required to process all queries. In other words, all the algorithms report the same tendency with both metrics. In this figure, we also show that the running time reported by our proposal has a smooth increase with a larger cache size, mainly because of the overhead involved in the iterations and the data management (e.g., list of clusters visited in previous iterations, distance pre-computed, etc.) required to jump from one iteration to another. However, our proposal outperforms the QCache algorithm by 38% on average, and the RCache algorithms by 51% on average.

A limitation of the RCache and QCache strategies is that they do not efficiently scale to fairly large values of k and there is no control about the desired k value (they only ensure $k = 1$). In [24], the authors present experimental results showing that in QCache, considering all cache hits, about 60% of queries retrieve $k = 2$ closest results, 20% of queries retrieve $k = 6$ closest results, and 8% of queries retrieve $k = 20$ closest results. In addition, in the results presented in Fig. 9, the cache hits for cache sizes 1, 3, and 6% are, respectively, 10, 21, 22% for QCache, and 9, 15, 25% for RCache. For instance, this means that for $k = 1$, about 75% of queries cause execution times similar to those presented in Fig. 6 for either kNN or Average Radius on the LC data structure, and about 95% of queries cause similar execution times for $k = 6$. Even considering that running time access to RCache and QCache is negligible, these strategies are not able to achieve the execution times of our proposal in a distributed search engine.

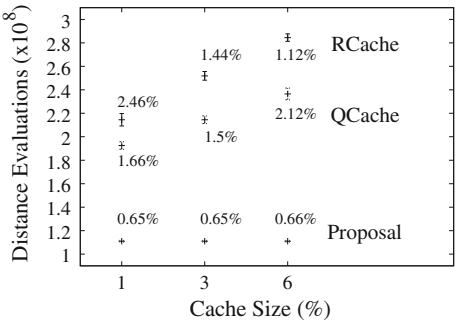

Fig. 10 Average number of distance evaluations reported with different set of queries including variance across different query data sets

We also evaluate the comparative performance of our proposal with five independent set of queries. To this end, we generated each set by selecting 10,000 different queries at random from our query log. In Fig. 10, we show the average number of distance evaluations performed with different cache sizes. That is, for each set of queries, we compute the average number of distance evaluations $\bar{x_1}, \ldots, \bar{x_5}$, respectively. Then, we compute the average of those five values ($\bar{X} = (\sum_{i=1}^{5} \bar{x_i})/5$). We also show the standard deviation (error bars in the figure) of the five values $\bar{x_i}$. Above each error bar shows the percentage of the standard deviation with respect to the average number of distance evaluations $\bar{X}$. The results show that as we change the set of queries, the performance of the algorithms remains stable with a low standard deviation. Our proposal reports a standard deviation for a cache size of 1%, which represents only 0.65% of the average number of distance evaluations. For a cache size of 6%, our proposal reports a similar standard deviation, which represents 0.66% of the average number of distance evaluations. The QCache reports standard deviations representing 1.5% of the average number of distance evaluations and 2.12% of the average number of distance evaluations. Finally, the RCache algorithms reports the highest standard deviation for a cache size of 1%. Therefore, the QCache and the RCache report more fluctuations in the performance when using different set of queries. However, these fluctuations are a small percentage of the total cost of the algorithms.

5.2 Approximation quality

The effectiveness of information retrieval systems is typically measured based on the number of relevant objects retrieved for queries. To this end, in Fig. 11a, b, we show the relative error reported by the LC, the LC-Log, and the proposed algorithms. The relative error on the maximal distance (REM) and the relative error on the sum of distances (RES), both when comparing the exact answers to queries against the approximate answers to the same queries, are defined as [25]:

$$\text{REM} = \frac{\max_{x \in A} d(q, x)}{r_q} - 1.0 \tag{1}$$

$$\text{RES} = \frac{\sum_{x \in A} d(q, x)}{\sum_{y \in \text{kNN}} d(q, y)} - 1.0, \tag{2}$$

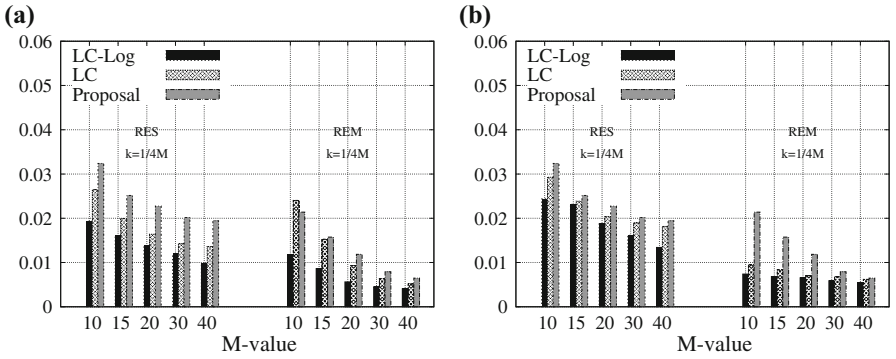

Fig. 11 Average relative error on the sum of distances (RES) and average relative error on the maximal distance (REM). **a** Using query search radius increment $\gamma = 0.5$, **b** using query search radius increment $\gamma = 0.1$

where A is the set of objects retrieved using the approximated algorithms. kNN stands for the exact K closest objects to the query q, and r_q is the query search radius for this query q, which is defined as the distance from the query q to the Kth closest exact answer object. In the results shown below, we calculated the average RES and REM values on the set of queries q executed in each experiment.

Figure 11a shows the results obtained for search radius increment $\gamma = 0.5$, and Fig. 11b shows the results obtained for search radius increment $\gamma = 0.1$. The results clearly show that the average errors measured by RES and REM are well below 4% in all cases. In particular, the REM average values indicate that all of the objects retrieved by the approximate algorithms are almost (4%) as close to the queries as the objects belonging to the exact answers for the same queries. A larger γ value tends to increase the quality of results of the algorithms LC and LC-Log. Therefore, in the figures, the RES values tend to be lower for $\gamma = 0.5$. This effect of the γ value is illustrated in Fig. 12a. In this example, we obtain $k = 3$ exact object results (the black balls within the query search radius) and four approximate results denoted by the letters $\{x, y, z, w\}$. This set of objects are included in the set of approximate object results because they are not discarded by the triangle inequality. But we realize that they are outside the query ball when computing the distance between them and the query. However, if we increase the search radius, we can improve the quality of approximate results by including the set of objects $\{a, b, c\}$ which are closer to q than $\{y, z, w\}$.

For a $\gamma = 0.5$, the average relative error of the maximum distance (REM) tends to be larger than when using $\gamma = 0.1$. We explain this behavior with the example of Fig. 12c. In Fig. 12c, the continuous line represents a small γ value. The dotted line represents a larger γ value. In both cases, we already have the $k = 3$ closest objects to the query. For a small γ value, we report an approximate object named x which had not been discarded by the triangle inequality. The object x drastically increases the query radius. For a larger γ value, we have two approximate candidates named x and w. Because in our example, we need only one approximate result, our algorithm may process first w and then stops. Thus, the maximum distance is larger in this last case. On average, we observed in our experiments that all object results are close enough to

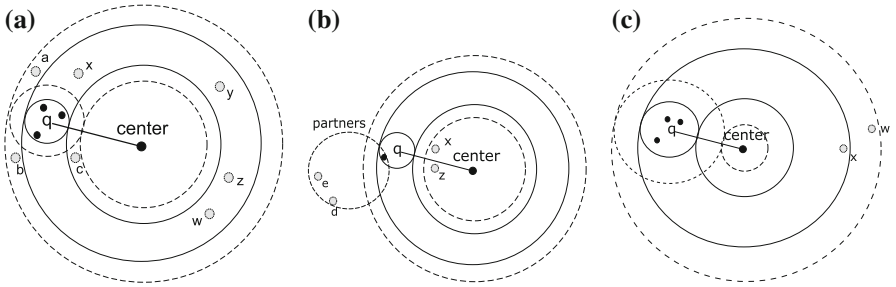

Fig. 12 Analysis of the quality of approximate results. **a** Approximate results can significantly improve their quality with a larger query search radius increment γ value, **b** some partner objects can reduce the quality of approximate results, and **c** a larger query search radius increment γ can increase the relative error on the maximum distance (REM)

the query, but the Mth approximate result object can be farther away, in relative terms, from the exact Mth object. This behavior usually happens when we already have the k-NN objects and we have to select from a small set of approximate objects the elements to complete the M results. Nevertheless, as described at the end of Sect. 4.1, we can easily allow our approximate LC algorithm operating in tandem with the proposed graph-based cache strategy to go for an additional iteration whenever the REM value for the current approximate results for a query is beyond a value, say 0.05 where the value r_q in Eq. 1 may be an average value observed in the current set of queries for which the exact answers have been calculated so far.

Using a larger value of M and a larger value of k, we can reduce both errors since we retrieve more exact answers and we can build a more diverse set of approximate results. Additionally, selecting the centers for the LC data structure from a set of queries already processed (LC-Log) rather than from the entire collection has a better impact on these metrics. This is because the LC-Log data structure construction algorithm selects as centers the objects that were part of previous query exact results. The respective queries are obtained from a query log containing past user queries wherein it is already registered the fact that some queries are more popular than others.

Our proposed cache-based algorithm tends to report higher REM and RES average values than the LC and the LC-Log algorithms operating alone as shown in Fig. 11a, b. Our proposal increases the query radius faster than the LC and the LC-Log algorithms, which explains its more efficient performance but it comes at the cost of reducing the quality of results (we illustrate this behavior in Fig. 12b). Nevertheless, the results in Fig. 11a, b show that the major differences among them occur in cases where the REM and RES average error values of our proposal (with no additional iterations) are already quite low for practical applications (around 2%).

In Fig. 13, we show results for the so-called precision metric, which is a much more demanding performance metric than the RES and REM metrics. The precision metric is computed as $\frac{\text{exact result objects} \cap \text{retrieved objects}}{\text{retrieved objects}}$. Namely, the quality of the query answers is only measured by the ability of the approximate search process to retrieve exact result objects. This metric does not take into account the quality of the approximate result objects present in the remaining $M - k$ result objects. In Fig. 13, we show that when

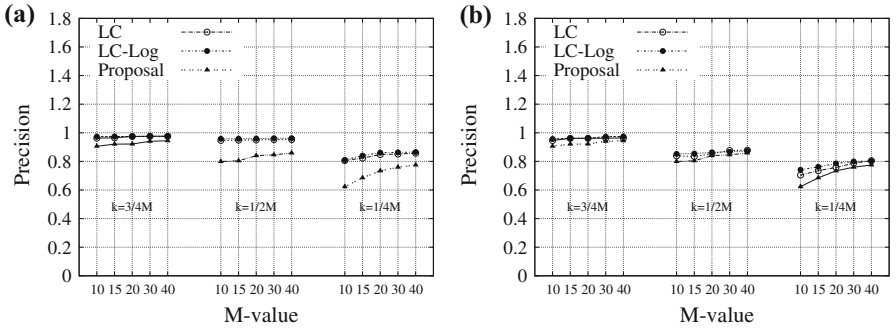

Fig. 13 Precision results obtained by using different query search radius increment γ values. **a** Using $\gamma = 0.5$, **b** using $\gamma = 0.1$

we use a small search radius increment value of $\gamma = 0.1$, the precision of results for $k = \frac{1}{2}M$ and $k = \frac{1}{4}M$ is lower than when we use $\gamma = 0.5$. This is because when using $\gamma = 0.1$, we have a smaller set of current objects from where to select the remaining $M - k$ result objects. On the other hand, and as explained in Fig. 12a, when we use $\gamma = 0.5$, the query radius tends to be larger and so we have a more diverse set of potential result objects.

For the strategy proposed in this paper, namely the graph-based cache operating in tandem with either the LC or the LC-Log approximate algorithms, the results in Fig. 13 show that the influence of the values of the γ factor on the quality of results is lower than the influence of this factor on either the LC or the LC-Log. In other words, the results in Fig. 13 are very similar with $\gamma = 0.5$ and $\gamma = 0.1$. This is because our proposal only uses the γ factor when the query ball is empty.

However, though well above the lower bound k/M, our proposal presents a precision about 20% lower than the LC and LC-Log algorithms operating alone for $M = 10$ and $k = \frac{1}{4}M$ as shown in Fig. 13a. We explain this behavior with the example illustrated in Fig. 12b. In this example, there is only one object inside the query search radius. This object has two partner objects e and d, which are included in the set of approximate results. However, the objects x and z are closer to the query and they are not considered by the cache-based strategy. This situation is not possible when executing the iterative algorithms LC and LC-Log operating alone as the objects x and z are included when we increase the query search radius using the γ increment factor. Overall, the results show that the LC and LC-Log algorithms obtain higher quality of results in terms of the precision metric for small values of k and M, whereas our proposal achieves competitive quality for large values of k and M.

As above said, the precision metric is too demanding (and in fact too pessimistic) as it does not consider the quality of the approximate result objects. Our use case is a metric space search engine. By construction, these systems are devised to retrieve similar objects for user queries during kNN search operations. In this setting, the distance function is usually an imperfect construct devised to capture the user notion of what objects are the closest ones to a given query object. This issue has been discussed in the literature (e.g., [35,37]). In practice, small differences in distance

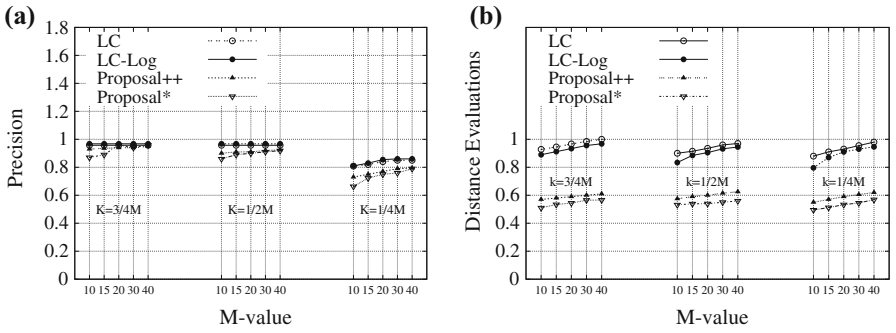

Fig. 14 Results obtained by including one additional iteration in the Proposal algorithm. **a** Precision using $\gamma = 0.5$, **b** respective distance evaluations using $\gamma = 0.5$

among two objects with respect to the query object tend to be irrelevant for users. This argument is what makes the case for the more suitable metrics RES and REM proposed in [25]. In this regard, the above experiments show that as for the metrics RES and REM, our proposal achieves a very competitive quality of results (range 2% as shown in Fig. 11) with the advantage that the overall running time performance is significantly more efficient than the alternative LC/LC-Log alone approximate algorithms (range 20–40% as shown in Fig. 7).

However, even in a case where the quality of the precision metric is a mandatory issue, our proposal can be used with a larger value of k and performance gain will be still relevant. For instance, it can be observed a performance gain of about 13% when looking at the cases Proposal with $k = \frac{1}{2}M$ versus LC/LC-Log with $k = \frac{1}{4}M$ in Fig. 7b, whereas the precision of our proposal is about 8% better than LC/LC-Log precision for the same case as shown in Fig. 13b.

Moreover, in Fig. 14, we show that by executing one additional iteration after finding the closest k exact object results (label Proposal++), the precision of our proposed algorithm is significantly improved. The results for $\gamma = 0.5$ (i.e., the more detrimental case for our proposal) now show a precision lost of at most 6 versus 20% for the case without the additional iteration as compared with the results in Fig. 13a. This also impacts on the performance of the algorithm. However, this impact is small reporting an increment of at most 3.5% in the number of distance evaluations as shown in Fig. 14b as compared with the results presented in Fig. 7a.

In Fig. 14, curves labeled Proposal*, we also present results for our proposal enhanced with the strategy for discarding partner objects found to be too far from the search query (see its description at the end of Sect. 4.1). The results obtained by Proposal* for the precision metric in Fig. 14a are very similar to the results presented for the Proposal algorithm in Fig. 13a. On the other hand, the results in Fig. 14b show that the Proposal* algorithm reduces by 5% (on average) the number of distance evaluations reported by the Proposal++ algorithm. Also, the results in Fig. 14b show that the Proposal* algorithm slightly reduces the number of distance evaluations ($< 5\%$) reported by the Proposal algorithm in Fig. 7a. This is because discarding partner objects too far from the search query actually reduces the growth rate of the

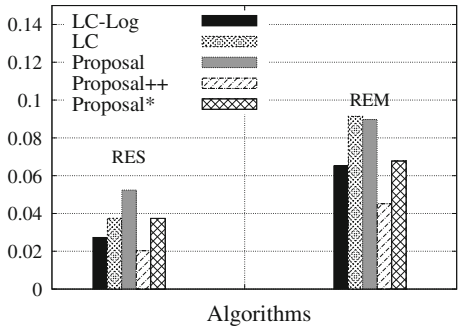

Fig. 15 Average relative error on the sum of distances (RES) and average relative error on the maximal distance (REM) for a subset of queries reporting REM and RES values larger average. Using query search radius increment $\gamma = 0.5$, $M = 10$, and $k = \frac{1}{4}M$

search radius being applied by the approximate search LC algorithm to find more good quality objects that are candidate to become part of the M query results.

Additionally, for the next experiment, we select from our log the queries reporting REM and RES values that are larger than the average. We reexecute the algorithms only for this subset of queries, which represents a 7% of the whole query log. Namely, we calculate the RES and REM metrics by executing the queries that report the worse quality of approximate results. Figure 15 shows the REM and RES average values achieved for $\gamma = 0.5$, $M = 10$, and $k = \frac{1}{4}M$ (i.e., we use values of γ, M, and k representing a demanding case for these metrics). As expected, in this case, both metrics report higher values than the ones obtained when averaging over the whole query log as shown in Fig. 11. For the selected subset of queries, Fig. 15 shows that Proposal++ is able to further reduce the error than the other approximate algorithms, reporting RES = 2% and REM = 4.5%. On the other hand, the Proposal* algorithm presents competitive performance as compared with the LC and the LC-Log algorithms.

In previous figures (Figs. 5, 6), we showed that our proposal reduces the number of distance evaluations for different values of k. On the other hand, in Fig. 13, we show that precision of results is lost and in Fig. 14, we show that the precision can be improved by either increasing k or executing an additional iteration. Now, in Fig. 16, we show that our proposal presents a greater gain than loss regarding these two metrics: distance evaluations versus precision. Figure 16a shows results obtained with $\gamma = 0.5$, and Fig. 16b shows results obtained with $\gamma = 0.1$, where each curve point indicates a different value of M with $k = \frac{1}{4}M$. The results show that the precision reported by our proposal tends to grow faster than the number of distance evaluations. Overall, the increase in precision from one extreme to the another is about two times more efficient in terms of distance evaluation increment in our proposal than in the other approaches. This behavior can be seen clearly when $\gamma = 0.5$. In this case, the precision reported by our proposal improves by almost 20% while the number of distance evaluations grows slower, close to 11%. Also, for a precision close to 80%, our proposal requires 10% less distance evaluations than the other approximate algorithms. When $\gamma = 0.1$, our proposal shows that the precision also improves about 17% and in this case, the number of distance evaluations grows 13%. For a precision close to 77%, our proposal can reduce by 16% the number of distance evaluations reported by the LC and the LC-Log algorithms.

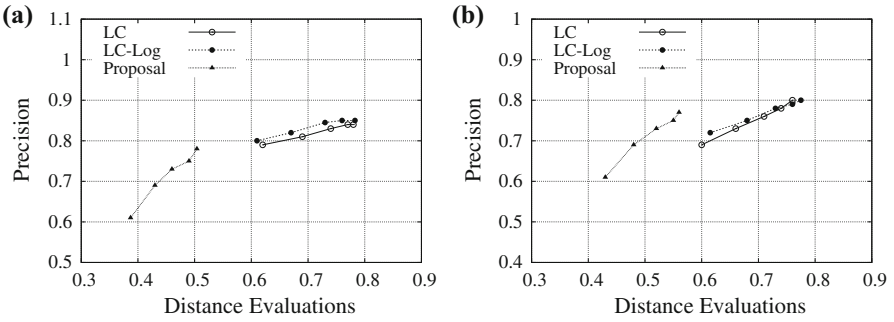

Fig. 16 Precision versus number of distance evaluations when retrieving $k = \frac{1}{4}M$ exact results, where each point in the curves represents a different value of $M = 10, 15, 20, 30, 40$. **a** Using search radius increment $\gamma = 0.5$, **b** using search radius increment $\gamma = 0.1$

Overall, the trade-off between the Proposal* and Proposal++ extensions to the Proposal algorithm is whether we are willing to pay the cost of more space used in the cache or executing more distance evaluations in the underlying LC approximate algorithms respectively. In any case, the experimental results show that the Proposal algorithm (whether extended or not) is an alternative, which is more efficient than previous approaches to approximate and exact kNN metric space search.

6 Conclusions and future work

Similarity search in metric spaces has demonstrated to be an effective paradigm for handling unstructured data (audio, video, documents, etc). However, due to the large amount of data stored in the Web, these systems have to face with new challenges to be able to process thousands of queries per second. In this paper, we have presented an approximate cache-based algorithm for a distributed metric space search engine. This algorithm avoids bottlenecks caused by expensive operations in the search engine processors. It retrieves approximate results for queries, ensuring that at least k of them are exact results from the $M > k$ results calculated for each query. The number of k exact results is a parameter that can be adjusted according to the observed query traffic. As more saturated becomes the search engine processors due to an increasingly high query traffic, the smaller the k value may be at the expense of reporting a larger number of approximate results ($M-k$), but at the gain of a much more efficient running time per processed query. To this end, the proposed cache-based algorithm operating in tandem with our own iterative search algorithm for the LC data structure makes use of exact result objects belonging to previously solved queries. To build this results cache, we create links among the objects forming a graph that can be quickly traversed during the search process to reduce the number of distance evaluations required by the iterative search algorithm.

We have experimentally compared our proposal against the standard kNN algorithm and other approaches for approximate metric space search. The results show that the cache-based algorithm performs an aggressive prune of object-to-object distance

evaluations during the search process, visiting a small number of clusters in the LC data structure and in this way, for instance, outperforming the standard kNN algorithm by 60%. Furthermore, this tendency is maintained by using different k values, which is convenient for setting different values of k when retrieving M result objects for queries in accordance with the observed incoming query traffic. We also evaluated the quality of the approximate query results observing deviations from the exact results of less than 4% for relevant quality metrics in metric space search. Unlike previous metric space cache strategies, our proposal ensures the k closest object results for queries for any value of k with the advantage that among the $M - k$ remaining result objects, there may be also exact results, which improves the overall quality of approximate query answers.

As future work, we plan to study the effects of trending topics in queries in the performance of the system and the quality of approximate results. Since our proposal can support updates in the cache, another interesting related matter is to investigate the elapsed time to update the cache, the time-to-live of frequent queries and fully dynamic caching. In addition, in the present version of our graph-based cache strategy, the vertical links are only set among identical objects participating in the answers for different queries. This can be relaxed to increase the effectiveness of the cache by also linking objects that are close enough to be considered equally similar to the eyes of users.

Acknowledgements This research was supported by the supercomputing infrastructure of the NLHPC Chile, partially funded by CONICYT Basal Funds FB0001, Fondef ID15I10560, and partially funded by PICT 2014 N 2014-01146.

References

1. Al-Fares M, Loukissas A, Vahdat A (2008) A scalable, commodity data center network architecture. SIGCOMM Comput Commun Rev 38(4):63–74
2. Amato G, Esuli A, Falchi F (2013) Pivot selection strategies for permutation-based similarity search. In: SISAP, pp 91–102
3. Amato G, Esuli A, Falchi E (2015) A comparison of pivot selection techniques for permutation-based indexing. J Inf Syst 52(C):176–188
4. Amato G, Savino P (2008) Approximate similarity search in metric spaces using inverted files. In: InfoScale, pp 28:1–28:10
5. Andoni A, Indyk P (2008) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. J Commun ACM 51(1):117–122
6. Arya S, Mount DM, Netanyahu NS, Silverman R, Wu AY (1998) An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J ACM 45(6):891–923
7. Baeza-Yates R, Ribeiro-Neto B (2011) Modern information retrieval, 2nd edn. Addison-Wesley Publishing Company, Reading
8. Brisaboa NR, Cerdeira-Pena A, Gil-Costa V, Marín M, Pedreira O (2015) Efficient similarity search by combining indexing and caching strategies. In: SOFSEM, pp 486–497
9. Burkhard WA, Keller RM (1973) Some approaches to best-match file searching. J Commun ACM 4(16):230–236
10. Bustos B, Navarro G, Chávez E (2003) Pivot selection techniques for proximity searching in metric spaces. J Pattern Recognit Lett 24(14):2357–2366
11. Bustos B, Pedreira O, Brisaboa N (2008) A dynamic pivot selection technique for similarity search. In: SISAP, pp 394–401
12. Cao W, Sahin S, Liu L, Bao X (2016) Evaluation and analysis of in-memory key-value systems. In: BigData, pp 26–33

13. Chávez E, Figueroa K, Navarro G (2008) Effective proximity retrieval by ordering permutations. J Pattern Anal Manag Intell 30:1647–1658
14. Chávez E, Ludueña V, Reyes N, Roggero P (2016) Faster proximity searching with the distal SAT. J Inf Syst 59:15–47
15. Chávez E, Marroquin J, Navarro G (2001) Fixed queries array: a fast and economical data structure for proximity searching. J Multimed Tools Appl 14(2):113–135
16. Chávez E, Navarro G (2005) A compact space decomposition for effective metric indexing. J Pattern Recogn Lett 26(9):1363–1376
17. Chierichetti F, Kumar R, Vassilvitskii S (2009) Similarity caching. In: SIGMOD-SIGACT-SIGART, pp 127–136
18. Ciaccia P, Patella M (2000) PAC nearest neighbor queries: approximate and controlled search in high-dimensional and metric spaces. In: ICDE, pp 244–255
19. Ciaccia P, Patella M, Zezula P (1997) M-tree: an efficient access method for similarity search in metric spaces. In: VLDB, pp 426–435
20. Dehne F, Noltemeier H (1988) Voronoi trees and clustering problems. In: Syntactic and structural, pattern recognition, pp 185–194
21. Egecioglu Ö, Ferhatosmanoglu H, Ogras ÜY (2004) Dimensionality reduction and similarity computation by inner-product approximations. IEEE Trans Knowl Data Eng 16(6):714–726
22. Esuli A (2009) Mipai: using the pp-index to build an efficient and scalable similarity search system. In: SISAP, pp 146–148
23. Esuli A (2010) Pp-index: using permutation prefixes for efficient and scalable similarity search. In: SEBD, pp 318–325
24. Falchi F, Lucchese C, Orlando S, Perego R, Rabitti F (2008) A metric cache for similarity search. In: LSDS-IR, pp 43–50
25. Falchi F, Lucchese C, Orlando S, Perego R, Rabitti F (2009) Caching content-based queries for robust and efficient image retrieval. In: EDBT, pp 780–790
26. Falchi F, Lucchese C, Orlando S, Perego R, Rabitti F (2011) Similarity caching in large-scale image retrieval. J Inf Process Manag 48(5):803–818
27. Faloutsos C, Lin K-I (1995) Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In: SIGMOD, pp 163–174
28. Ferhatosmanoglu H, Tuncel E, Agrawal D, El Abbadi A (2001) Approximate nearest neighbor searching in multimedia databases. In: ICDE, pp 503–511
29. Figueroa K, Paredes R (2015) Boosting the permutation based index for proximity searching. In: MCPR, pp 103–112
30. Gennaro C, Amato G, Bolettieri P, Savino P (2010) An approach to content-based image retrieval based on the lucene search engine library. In: ECDL, pp 55–66
31. Gessert F, Wingerath W, Friedrich S, Ritter N (2017) Nosql database systems: a survey and decision guidance. J Comput Sci R&D 32(3–4):353–365
32. Gil-Costa V, Marin M (2011) Approximate distributed metric-space search. In: LSDS-IR, pp 15–20
33. Gil-Costa V, Marin M, Reyes N (2009) Parallel query processing on distributed clustering indexes. J Discrete Algorithms 7(1):3–17
34. Gil-Costa V, Santos RLT, Macdonald C, Ounis I (2013) Modelling efficient novelty-based search result diversification in metric spaces. J Discrete Algorithms 18:75–88
35. Hersh W, Turpin A, Price S, Chan B, Kramer D, Sacherek L, Olson D (2000) Do batch and user evaluations give the same results? In: SIGIR, pp 17–24
36. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: ACM Symposium on Theory of Computing, pp 604–613
37. Ingwersen P, Järvelin K (2005) The turn: integration of information seeking and retrieval in context (The Information Retrieval Series). Springer, New York Inc, Secaucus
38. Johnston N, Vincent D, Minnen D, Covell M, Singh S, Chinen TT, Hwang SJ, Shor J, Toderici G (2017) Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. CoRR, arXiv:abs/1703.10114
39. Karypis G (2003) Cluto-software for clustering high-dimensional datasets, version 2.1.1. http://glaros.dtc.umn.edu/gkhome/views/cluto
40. Lux M, Chatzichristofis SA (2008) Lire: lucene image retrieval: an extensible java cbir library. In: Conference on Multimedia, pp 1085–1088

41. MacQueen JB (1967) Some methods for classification and analysis of multivariate observations. In: Berkeley Symposium on Mathematical Statistics and Probability, vol 1, pp 281–297
42. Mancini V, Bustos F, Gil-Costa V, Printista AM (2012) Data partitioning evaluation for multimedia systems in hybrid environments. In: 3PGCIC, pp 321–326
43. Marin M, Ferrarotti F, Gil-Costa V (2010) Distributing a metric-space search index onto processors. In: ICPP, pp 13–16
44. Marin M, Gil-Costa V, Uribe R (2008) Hybrid index for metric space databases. In: ICCS, pp 327–336
45. Matej A, Vlastislav D (2016) Optimizing query performance with inverted cache in metric spaces. In: ADBIS, pp 60–73
46. Micó ML, Oncina J, Vidal E (1994) A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. J Pattern Recognit Lett 15(1):9–17
47. Navarro G (2002) Searching in metric spaces by spatial approximation. In: VLDB, pp 28–46
48. Navarro G, Reyes N (2002) Fully dynamic spatial approximation trees. In: SPIRE, pp 254–270
49. Navarro G, Reyes N (2009) Dynamic spatial approximation trees for massive data. In: SISAP, pp 81–88
50. Novak D, Batko M (2009) Metric index: an efficient and scalable solution for similarity search. In: SISAP, pp 65–73
51. Novak D, Batko M, Zezula P (2012) Large-scale similarity data management with distributed metric index. J Inf Process Manag 48(5):855–872
52. Novak D, Zezula P (2016) PPP-codes for large-scale similarity searching. In: Database and expert-systems applications on transactions on large-scale data- and knowledge-centered systems, pp 61–87
53. Pedreira O, Brisaboa NR (2007) Sofsem. In: Theory and practice of computer science, pp 434–445
54. Ogras ÜY, Ferhatosmanoglu H (2003) Dimensionality reduction using magnitude and shape approximations. In: CIKM, pp 99–107
55. Pan Z, Lei J, Zhang Y, Sun X, Kwong S (2016) Fast motion estimation based on content property for low-complexity H.265/HEVC encoder. J IEEE Trans Broadcast 62(3):675–684
56. Pandey S, Broder A, Chierichetti F, Josifovski V, Kumar R, Vassilvitskii S (2009) Nearest-neighbor caching for content-match applications. In: WWW, pp 441–450
57. Pramanik S, Alexander S, Li J (1999) An efficient searching algorithm for approximate nearest neighbor queries in high dimensions. IEEE Multimed Comput Syst 1:865–869
58. Raghavendra S, Nithyashree K, Geeta CM, Buyya R, Venugopal KR, Iyengar SS, Patnaik LM (2016) RSSMSO rapid similarity search on metric space object stored in cloud environment. J Organ Collect Intell 6(3):33–49
59. Ruqeishi K, Koneuay M (2015) Regrouping metric-space search index for search engine size adaptation. In: Similarity search and applications, pp 271–282
60. Saavedra JM, Barrios JM (2015) Sketch based image retrieval using learned keyshapes (LKS). In: British Machine Vision Conference, pp 164.1–164.11
61. Skala M (2009) Counting distance permutations. J Discrete Algorithms 7(1):49–61
62. Skillicorn DB, Hill JMD, McColl WF (2000) Mpeg-7. Multimedia content description interfaces, part 3: visual. Technical Report ISO/IEC 15938-3
63. Skopal T, Lokoc J, Bustos B (2012) D-cache: universal distance cache for metric access methods. J Trans Knowl Data Eng 24(5):868–881
64. Solar R, Gil-Costa V, Marín M (2016) Evaluation of static/dynamic cache for similarity search engines. In: SOFSEM, pp 615–627
65. Sadit Tellez E, Chvez E (2012) The list of clusters revisited. In: Pattern recognition, pp 187–196
66. Wang X, Wang JTL, Lin K-I, Shasha D, Shapiro BA, Zhang K (2000) An index structure for data mining and clustering. J Knowl Inf Syst 2:161–184
67. Weber R, Böhm K (2000) Trading quality for time with nearest neighbor search. In: Extending database technology: advances in database technology, pp 21–35
68. Wei W, Fan X, Song H, Fan X, Yang J (2017) Imperfect information dynamic stackelberg game based resource allocation using hidden Markov for cloud computing. J IEEE Trans Serv Comput PP(99):1–1
69. White D, Jain R (1996) Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California San Diego
70. Xia Z, Wang X, Zhang L, Qin Z, Sun X, Ren K (2016) A privacy-preserving and copy-deterrence content-based image retrieval scheme in cloud computing. J IEEE Trans Inf Forensics Secur 11(11):2594–2608

71. Zezula P, Amato G, Dohnal V, Batko M (2006) Similarity search: the metric space approach, advances in database systems. Springer, Berlin
72. Zhou Z, Wang Y, Wu QMJ, Yang CN, Sun X (2017) Effective and efficient global context verification for image copy detection. J IEEE Trans Inf Forensics Secur 12(1):48–63
73. Zhou Z, Wu QMJ, Huang F, Sun X (2017) Fast and accurate near-duplicate image elimination for visual sensor networks. J Distrib Sens Netw 13(2):1–1