Efficient Similarity Search by Combining Indexing and Caching Strategies*

Nieves R. Brisaboa[1], Ana Cerdeira-Pena[1], Veronica Gil-Costa[3], Mauricio Marin[2], and Oscar Pedreira[1]

[1] Database Lab., Facultade de Informática, Universidade da Coruña, Spain
{brisaboa,acerdeira,opedreira}@udc.es
[2] CITIAPS, DIINF, University of Santiago, Chile
mauricio.marin@usach.cl
[3] DCC, National University of San Luis, Argentina
gvcosta@unsl.edu.ar

Abstract. A critical issue in large scale search engines is to efficiently handle sudden peaks of incoming query traffic. Research in metric spaces has addressed this problem from the point of view of creating caches that provide information to, if possible, exactly/approximately answer a query very quickly without needing to further process an index. However, one of the problems of that approach is that, if the cache is not able to provide an answer, the distances computed up to that moment are wasted, and the search must proceed through the index structure. In this paper we present an index structure that serves a twofold role: that of a cache and an index in the same structure. In this way, if we are not able to provide a quick approximate answer for the query, the distances computed up to that moment are used to query the index. We present an experimental evaluation of the performance obtained with our structure.

1 Introduction

New applications for search engines demand the use of data more complex than plain-text. Metric spaces have proven useful and practical for performing similarity search on very-large collections of complex objects. In this case, queries are objects of the same type of those stored in the database where, for example, one is interested in retrieving the k most similar objects to a given query. The similarity between any two objects is calculated by an application-dependent *distance function*, which is usually expensive to compute. The database is indexed using pre-computed distances to reduce comparisons during the search.

One of the critical issues in large scale search engines is efficiently handling sudden peaks in incoming query traffic. Typically, a large search engine is composed of one or more *front-service* (FS) machines and a collection of P processors

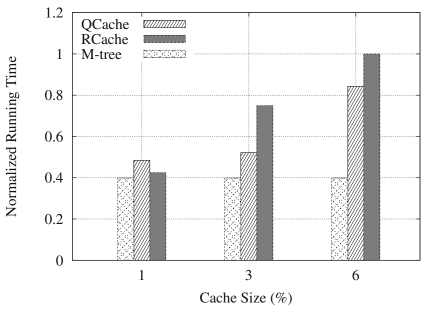
* Partially funded by: MICIN ref. TIN2009-14560-C03-02 (PGE & FEDER), Xunta de Galicia ref. GRC2013/053 (FEDER), and CDTI-MINECO-Axencia Galega de Innovación EXP 00064563/ITC-20133062 for authors in UDC[1]. FONDEF IDeA CA12i10314 for M. Marín. Mincyt-Conicyt CH1204 for V. Gil-Costa.

forming a distributed memory system. The Front-Service is in charge of receiving and sending queries to processors for results calculation. Each processor is seen as a *search node* which is in charge of a fraction of the whole object collection. Efficient search is supported by an index data structure that is distributed onto the P processors and parallel query processing is performed by sending the query to a number of processors. For systems under heavy query traffic it is critical to reduce the number of computations and yet to maintain an efficient throughput (number of queries entirely solved per unit time).

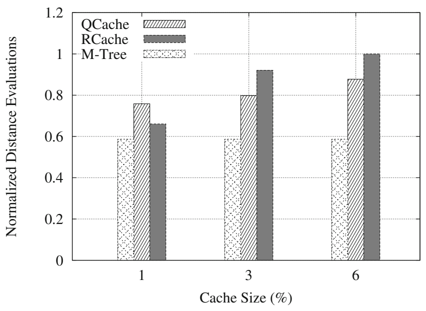
Research in metric-space similarity search has mainly focused on optimizing the execution of single queries. In multiple query settings, where query arrival rate can drastically change in intensity, and query content can become dynamically skewed in unpredictable ways, a relevant question is how we can make current queries benefit from previous query results so that they can be answered with approximate results. The underlying assumption is that approximate answers can be computed with much less computing cycles than regular answers so that servers are able to cope with drastic increase in incoming query traffic.

Caching query results is a feasible solution, and strategies such as QCache and RCache [1] have been proposed. However, they fail to reduce overall computing cycles as they treat independently caching and indexing. Namely, incoming queries that do not benefit from the cache are redirected to the metric index so query answers are computed from scratch. Our experiments show that the cost of accessing the cache in previous strategies can be even more expensive (or similar) than computing the answer for a query from the index itself.

To illustrate the above claim we performed experiments under the setting described in Section 4. We used a query log on the following three cases: (1) each query is sent to a RCache [1] and if the cache fails to produce the top-k results, the query is solved with an M-Tree [2] index, (2) each query is sent to a QCache [1], (a variation of the RCache) and if the cache fails to produce the top-k results, the query is sent to the M-Tree index and, (3) each query is directly solved with the M-Tree index. In each case we computed the running time and the number of distance evaluations required to process the full query log. Figure 1 shows the results normalized to 1. Cache sizes were set to 1%,



(a) Running Time



(b) Distance Evaluations

Fig. 1. Performance achieved by different cache strategies

3% and 6% of the number of queries, which produces cache hit ratios of 15%, 22% and 33% respectively. The results show that the running time and distance evaluations when using caches are a significant part of the search cost.

In this paper we propose a strategy which contains a cache embedded in the index so that computing cycles are not wasted when cache contents are not able to produce good approximate results. In such a case, previous computations for the query are used to continue the traversal of the index in order to produce approximate query results as fast as possible.

The remaining of this paper is organized as follows. Section 2 reviews related work on similarity search and caching. In Section 3 we present a new index structure that combines indexing and caching strategies. Section 4 presents the results obtained in the experimental evaluation of our structure. Finally, Section 5 summarizes the main conclusions from our work.

2 Related Work

A *metric space* (U, d) is composed of a universe of objects U and a *metric*, a function $d : U \times U \to R^+$ that measures the dissimilarity between any two objects and that holds the properties of strictly positiveness ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). The *database* or *collection* of objects is a finite subset $X \subseteq U$, with size $n = |X|$.

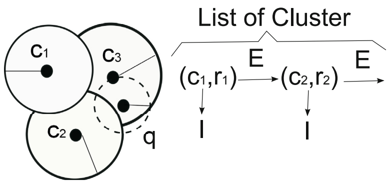
There are two main queries of interest: (a) *range search*, $R_X(q, r)$, that retrieves all the objects $u \in X$ within a search radius r of the query q, and (b) *k-nearest neighbors search*, $kNN_X(q)$, that retrieves the set k most similar objects to q. Given a query $q \in U$, the goal is to retrieve the most similar objects to q with the minimum number of object comparisons.

Many metric index structures have been proposed and studied (see [3,4]). The proposals of this paper make use of one of those structures, the *List of Clusters* (LC) [5], which has shown to outperform well-known alternative metric-space indexes [6]. LC partitions the collection into a set of disjoint clusters as follows. We first choose a *cluster center* $c \in X$ and a radius r_c. The *cluster ball* (c, r_c) contains the subset of elements of X at distance at most r_c from c. We define $I_{X,c,r_c} = \{u \in X - \{c\}, d(c, u) \leq r_c\}$ as the cluster of *internal* elements which lie inside (c, r_c), and $E_{X,c,r_c} = \{u \in X, d(c, u) > r_c\}$ as the *external* elements. The clustering process is recursively applied in E. As shown in [5] a good policy for selecting the next center is to choose the object in the collection that maximizes the sum of distances to previous centers.

Given a query $R_x(q, r)$, q is sequentially compared with the cluster centers of the LC. Given a center c, we exhaustively scan its cluster I (that is, we compare q with the objects $u \in I$) if the query ball (q, r) intersects the cluster ball (c, r_c). The search then continues with the next cluster in LC. At any point of the search, the search stops if the query ball (q, r) is totally and strictly contained in the cluster ball (c, r_c), since the construction process ensures that all the elements that are inside the query ball (q, r) have been inserted in I (as shown in line

```
   Search($LC, q, r$)
1. If $LC$ is empty Then Return
2. Let $LC = (c, r_c, I)$ : $E$
3. Compute the distance $d(c, q)$
4. If $d(c, q) \leq r$ Add $c$ to the set of
     results
5. If $d(c, q) \leq r_c + r$ Then Search $I$
     exhaustively
6. If $d(c, q) > r_c - r$ Then Search($E, q, r$)
```

(a) LC search algorithm



(b) LC example

Fig. 2. The List of Clusters (LC) strategy

6 of Figure 2a). Figure 2b shows three clusters and a query $R_X(q, r)$. In this example, q has to be compared with the objects in the clusters with centers c_2 and c_3, but the cluster with center c_1 is directly discarded.

The selection of effective pivot /cluster centers for metric indexes has been deeply studied. One of the existing proposals for center selection is SSS [7,8] that selects a new object as a center if it is far enough from those already selected. Being M the maximum distance between any two objects in the space, and α a parameter such that $0 \leq \alpha \leq 1$, an object in the collection is selected as a new center if its distance to the previously selected centers is greater than $M \times \alpha$. In [9] it has been shown that the most effective pivots for a given object of the database are the nearest and furthest pivots.

2.1 Parallel Processing for List of Clusters

We assume a parallel architecture in which a *front-service* receives queries and evenly distributes their processing onto the processors. The work in [6] studied various forms of parallelization of the LC strategy concluding that a global indexing strategy called *GG*, which stands for Global Index and Global Centers, achieves the best performance.

The GG strategy builds a LC and distributes it uniformly at random the clusters of the LC onto the processors. Upon reception of a query q, the broker sends it to a circularly selected processor. This processor becomes the *ranker* for that query. It calculates the *query plan*, that is, the list of clusters that intersect (q, r). To this end, it broadcasts the query to all processors and they calculate in parallel a fraction $1/P$ of the query plan. Then they send their n_q/P pieces of the global plan to the ranker, which merges them to get the global plan with clusters sorted in construction order. The then ranker sends the query q and its plan to the processor i containing the first cluster to be visited. This processor i goes directly to the GG clusters that intersect with q, compares q against the objects stored in them, and returns to the ranker those within (q, r). The remaining part of the query plan is passed to the next processor j and so on, till completing the processing of the query.

2.2 Metric Space Cache

A *metric space cache* $\mathcal{C}$ consists of a set of past queries with their respective results. Let $q_i \in \mathcal{C}$, if the query along with all results in $kNN_X(q_i, k)$ are in the cache. Also $o_i \in \mathcal{C}$ denotes that the object $o_i \in X$ is stored in the cache and thus belongs to at least one set $kNN_X(q_i, k)$ associated with a cached query q_i. Let r_q denote the radius of the smallest hyper-sphere centered in q which contains all objects in $kNN_X(q, k)$. The *safe radius* s_q [1,10] of the query q with respect to a query $q_i \in \mathcal{C}$ is the radius r_{q_i} minus the distance from q to q_i, namely $s_q(q_i) = r_{q_i} - d(q, q_i)$. Every cached query q_i gives complete knowledge of the space up to distance r_{q_i} from q_i. If q is inside the hypersphere centered in q_i with radius r_{q_i}, then as long as we restrict ourselves to look inside this hypersphere, we have complete knowledge of the $k' \leq k$ nearest neighbours of q.

Thus, if the safe radius $s_q(q_i)$ of a query $q \in U$ with respect to a query $q_i \in \mathcal{C}$ is a positive value, then every object in the range query $R_X(q, s_q(q_i))$ is also in the cache $\mathcal{C}$ and thus can be solved over the cache with the range query $R_{\mathcal{C}}(q, s_q(q_i))$. Furthermore, the k' objects in $R_{\mathcal{C}}(q, s_q(q_i))$ are also the k' nearest neighbours of q in the whole database X.

In [1,10,11] two different metric-space cache algorithms were presented: RCache (Result Cache) and QCache (Query Cache). RCache uses a hash table $\mathcal{H}$ where, for each query $q_i \in \mathcal{C}$, it stores tuples of the form $(q_i, kNN(q_i, k))$, being the object q_i the hash key. If the query is not in $\mathcal{C}$, then it attempts to give an approximate answer. To search for an approximated answer, RCache uses a metric-space index $\mathcal{M}$ to perform a $kANN_{\mathcal{C}}(q, k)$ search of the k closest objects to q_i which are currently stored in $\mathcal{C}$. QCache builds the metric index $\mathcal{M}$ over the query objects instead of indexing every single object returned by the queries in the cache, as the RCache algorithm does. This reduces by a factor of k the number of indexed objects. The main idea is to search sets of suitable cached queries first and then to use the cached results of those queries to find an approximate answer. According to the experimental results reported in [1,10], the quality of the approximate results returned by both algorithms are comparable.

In [12] a caching metric-space index called D-File was proposed. D-File uses a hash table with entries $[o_1, o_2, d(o_1, o_2)]$, where o_1 and o_2 are the objects identifier and the third component is the computed distance between them. D-File is kept in main memory in order to reduce the number of distance computations performed over a second index like the M-tree [2]. This goal is achieved when the distance $d(o_1, o_2)$ is in the D-File, or by obtaining a lower or upper bound of $d(o_1, o_2)$ and thereby improving the pruning over the M-tree. However, as shown in [13], D-File suffers from a too high internal processing cost because of the hash table. Recently, SnakeTable [13] was specifically proposed for scenarios in which queries are received in streams of very similar queries.

3 Combining Indexing and Caching

One of the problems of previous proposals in caching for searching in metric spaces is the cost of processing the cache in terms of distance computations. If the cache succeeds in providing an answer, the distance computations to process the cache save the distance computations needed to solve the query with an index. However, if the cache fails, those distance computations for processing the cache are wasted and add up to the overall search cost.

In this section we present a variant of the list of clusters that combines indexing and caching policies into the same structure so, if the cache cannot provide an exact or approximate answer for the query, at least the distance computations needed to process the cache are not wasted, since they would be necessary anyway in order to solve the query in the index.

3.1 Index Structure and Construction

The index structure is that of a LC in which cluster centers are not selected among the objects in the database, but among the queries received in search time. Therefore, there is no index built until the system starts receiving queries. This implies that the search cost will be higher for the first queries. To avoid this in a real scenario, we can make use of an additional index at the beginning of the process that would be dropped when the new index structure has stabilized.

Since queries are dynamic, it is not possible to follow the center selection policy of LC. We use SSS [7], that adapts to the dynamic nature of the queries, and guarantees that the centers will be well distributed in the space. Therefore, if the distance from a new query q to existing cluster centers is greater than $M \times \alpha$, q will become a new cluster center and the index will be restructured accordingly. The cluster corresponding to the new cluster center q may be empty if no object in the collection is closer to q than to any other past query used as a cluster center. In this case, the cluster would be removed. As in LC, each object belongs to, and only to, the cluster formed by its closest cluster center.

Reorganizing the index when a new query is selected as a cluster center has a cost in terms of distance computations. However, SSS guarantees that the number of pivots will stabilize at some point, so the cost of that reorganization will be amortized among all processed queries.

The reason for choosing the cluster centers among the queries received by the system is that, in this way, they will better cover the portion of the space defined by the queries, which is not necessarily the same space defined by the objects in the database. This would benefit the algorithms for range and kNN search since the cluster centers would be more similar to future queries.

In addition, we keep additional information in each cluster. Instead of storing only the cluster center, the list of objects belonging to the cluster, and the covering radius, we keep the distances from the center to each of the objects in the cluster, as proposed in [14]. In this way, the cluster centers also play the role of a pivot during the search. As shown in [9], the nearest pivot is the most promising for each object, so using the cluster center as a pivot for each of the objects belonging to that cluster should be the most effective choice. In addition, keeping the distances from the cluster center to each object in the cluster will also make possible to return approximate results to queries sufficiently similar to one of the pivots.

3.2 Index as a Cache for Approximate Search

The main motivation for this index structure is that it can be used as a classical index or as a cache that may help to quickly provide approximate answers to queries when the system is receiving a huge number of queries. When a new query is submitted to the system, the first step consists in comparing the query object with the cluster centers. Since these objects are past queries, they reflect the space defined by the queries, and it is probable that new queries are equal or similar to some of the cluster centers. The higher the number of past queries used as cluster centers, the more the chances that a new query is sufficiently similar to a past query used as a cluster center.

If $d(q, c_i) = 0$ for some c_i, the approximate answer to q will be extracted by just using the information contained in $\mathcal{C}_i$. This may leave some objects out of the answer, but it would not require additional distance computations.

The case in which a new query is exactly equal to a past query used as a cluster center will happen very few times. However, it is still possible to provide an approximate result if q is sufficiently close to some cluster center c_i. Notice that by selecting the cluster centers with SSS, all of them are at least at a distance $M \times \alpha$. Therefore, the covering radius of each cluster will be at most $\frac{M \times \alpha}{2}$. We consider that q is sufficiently similar to the cluster center c_i if:

$$d(q, c_i) \leq \frac{M \times \alpha}{2} \times \rho$$

where $0 \leq \rho \leq 1$. Therefore, ρ determines how close a query has to be to a past query in order to return an approximate answer. Since the structure of the index does not depend on ρ, this parameter can be easily changed during the search phase depending on the processing demands on the system.

3.3 Searching

Algorithm 1 shows the pseudocode of the algorithm for range search. Given $R(q, r)$, and being I the index, the search process proceeds as follows:

- In a first step, the query q is compared with the center of each of the clusters of I (lines 2 and 3). This comparison allows us to determine if q will become a new cluster center or not. If the $d(q, p_i) > M \times \alpha$ for all pivots p_i, q becomes a new cluster center (lines 4 and 5). In this case, it is necessary to restructure the index so the objects that are closer to this new pivot are assigned to its cluster. The procedure $AddNewCluster(I, q)$ carries out this restructuring.

  Notice that it is possible to carry out this reorganization without comparing q with all objects in the database, since the distances from each object x to its cluster center provides us with lower bounds on $d(q, x)$ so it may not be necessary to compute $d(q, x)$.
- After comparing the query q with each of the cluster centers, we can use this information and the index as a cache (lines 6-8). First, if $d(q, c_i) = 0$ for some p_i, we can answer the query with the information contained in the cluster

```
1  Algorithm: RangeSearch $(I, q, r)$
   Data: $I$: Index structure; $q$: query object; $r$: search radius;
2  for $i$ = 1 to I.size do
3      $d[i] = d(q, p_i)$ ;
4  if $\forall i, d[i] \geq M \times \alpha$ then
5      AddNewCluster($I$,$q$);
6  else
7      if $\exists p_i \ / \ d[i] \leq \frac{M\times\alpha}{2} \times \rho$ then
8          ApproximateRangeSearch($I.c_i$, $q$, $r$);
9      else
10         for $i$ = 1 to I.size do
11             if $d[i] - cr_i \leq r$ then
12                 foreach $x_j \in I.\mathcal{C}_i$ do
13                     if $|d[i] - d(c_i, x_j)| \leq r$ then
14                         if $d(q, x_j) < r$ then
15                             $Result \leftarrow Result \cup \{x_j\}$
```

Algorithm 1. Pseudocode for range search with approximate search

formed by c_i. Even if an exact match does not occur, the information of the index allows us to provide an approximate answer. If $d(q, c_i) \leq \frac{M\times\alpha}{2} \times \rho$ for some c_i, we will also build the answer to q from the information contained in the cluster formed by c_i. In that case, we restrict the search to the cluster of c_i. Since the index stores the distances $d(x, c_i)$ for all the objects in the cluster, we can use these distances to obtain lower bounds for $d(q, x)$ and discard some objects x without comparing them with q.

– If cannot provide an approximate answer, the search continues exploiting the rest of information in the index by combining cluster and pivot criteria to discard the objects in the clusters (lines 9-15). Notice that at this point we already have the distance from q to the cluster centers. Those clusters that do not intersect the query ball (q, r) are directly discarded from the result. For those clusters that cannot be completely discarded, we use the distances from the center to the objects x in the cluster to obtain lower bounds on $d(q, x)$. In those cases in which these lower bounds do not allow us to directly discard x, we finally have to compute the distance $d(q, x)$.

With this algorithm, if the structure is not able to provide an approximate answer for a query (thus acting like a cache), the distances computed up to that moment are not wasted, since they are the same needed to search using the index. As in proposals in which the cache is a component separated from the index, in the proposed algorithm it is also possible to activate/deactivate the use of the cache in function of the processing requirements of the system. Actually, the parameter ρ allows us to control the degree to which the cache will be used (and in consequence, the quality of the approximate answers).

Although we have only explained the algorithm for range search, adapting it for the case of kNN search is straightforward. Given a new query q, its list of k nearest neighbors can be initialized with the first objects it is compared with.

Then, the search proceeds as a range search, but updating the range at each step if necessary as the distance from q to its current k^{th} nearest neighbor.

4 Experimental Results

In this section we provide experimental results on different aspects of our structure. We implemented it using the Metric Spaces Library [15]. We used two collections from the library, widely used in the state of the art, namely `English`, a dictionary containing 69,069 words, and `Nasa`, a collection of 40,151 images represented by feature vectors of dimension 20. The edit distance was used for `English`, and the Euclidean distance for `Nasa`.

For each collection, 90% of the objects were used as the database, and the remaining 10% were used as base queries. The query sets were created from the base queries in order to reflect the typical human behavior in real search engines. To do so, the base queries were replicated following the same distribution obtained from a set of real queries obtained from a web search engine. In this way, the queries have a biased distribution that matches that we would obtain in a real system. Following this procedure, we generated files with 10,000 queries for each collection. The query range for `English` was set to 2, and for `Nasa` we used a range that retrieves an average of 0.01% of the collection for each query.

In a first set of experiments, we analyzed how the structure performance evolves when the first queries are submitted to the system. To do this, we ran the 10,000 queries for each collection with an initially empty index. In order to focus on the cost of solving queries and updating the index, the parameter ρ was set to 0 in these experiments when the cache was used, which means that the cache is used only when a new query is exactly equal to a past query. Figures 3 and 4 show for each collection the number of distances needed to answer the range queries with and without using the cache part of the algorithm (`RWC` and `RWOC` respectively), the number of distances needed to update the index with and without cache (`UWC` and `UWOC` respectively), and the sum of these two numbers (`TWOC` and `TWC`). The results are shown in terms of the number of queries received by the system (from 0 to 10,000). As we can see in the results, the cost of solving a query decreases in all cases as the index gets more information. In the case of `Nasa`, which needs a smaller number of centers, we can see how the cost of updating the index quickly decreases as the number of past queries increases.

We conducted experiments to analyze the performance when we use the structure as a cache and an index at the same time, providing approximate but quick answers when possible. In order to leave the cost of updating the index out of the results so it does not interfere with the purpose of these experiments, we ran the 10,000 queries twice for each collection: the first one allows the index to build and stabilize (which has been analyzed in the previous experiments), and the second one allows us to measure search performance when very few changes in the index happen. Figure 5 shows the average number of distance computations to solve range queries and 4-NN queries for different values of the parameter ρ. The results reflect only the performance in the second phase. The figure shows

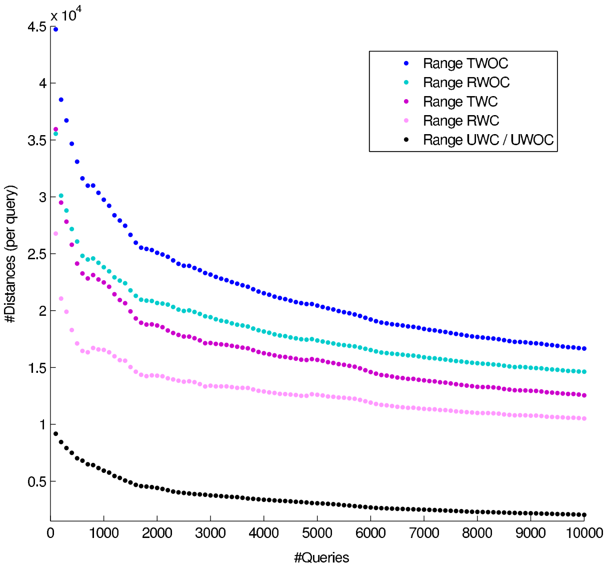


Fig. 3. Search and update cost for range queries in `English`

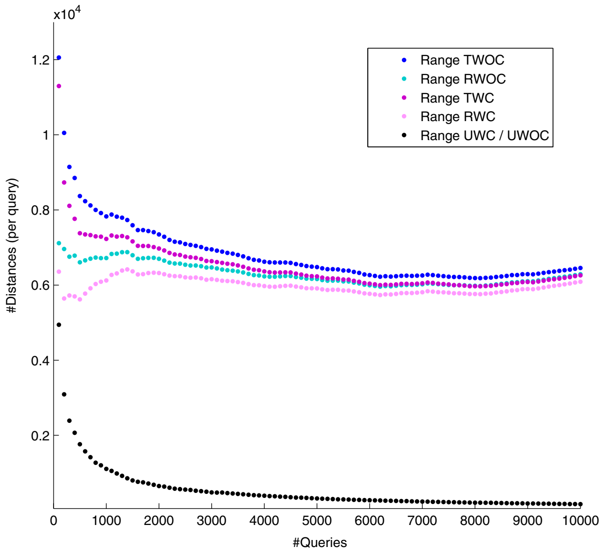
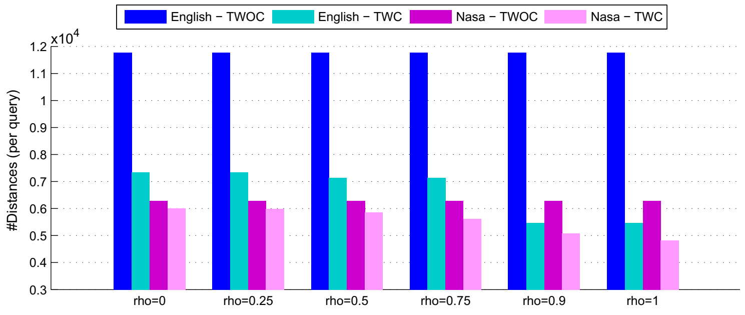
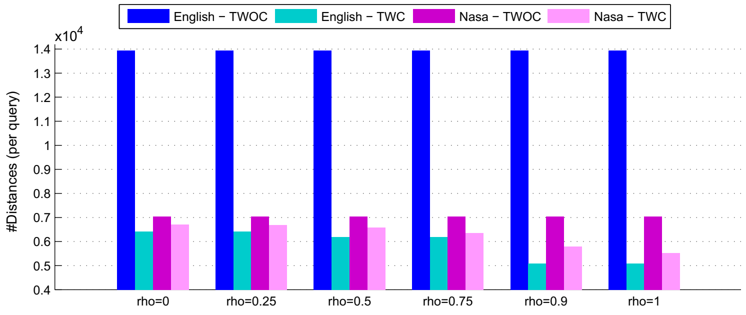


Fig. 4. Search and update cost for range queries in `Nasa`

for each value of ρ the cost of solving the query using the cache (`English-TWC` and `Nasa-TWC`) and without using it (`English-TWOC` and `Nasa-TWOC`). As the value of ρ grows, the requirements for a query to be solved using only information of its closest cluster are lower, which results in less distance computations.



(a) `Range search`



(b) `4-NN`

Fig. 5. Cost for range and 4-NN search using the structure as a cache and as an index

Even for the smallest values of ρ, the improvement with respect to the version that does not use the cache is very significant.

5 Conclusions

We have presented a metric structure for similarity search that allows us to use it as a cache or a classic metric index. One of the main differences with the original List of Clusters structure is that cluster centers are selected among past queries instead of among the objects in the collection, and that they are selected with a criterion that ensures they will be distributed in the space. In addition, each cluster stores the distances from the center to each of its objects. Given a new query, we provide an approximate result if it is equal or very similar to one of the past queries we keep in the index. If we cannot provide an approximate result, the distances computations needed to examine past queries are used to continue the search in the index, and to prune part of the search space.

The main advantage of this metric structure is that it can work as a cache and an index at the same time. Since the cluster centers are past queries, the comparison of each query with them allows us to provide an approximate but quick result. Even if the first step cannot return an approximate answer, those distance computations carried out to that moment are not wasted work, since they would be necessary anyway to query the index. Therefore, the main difference with previous proposals is that the cache is integrated with the index, not requiring any additional processing.

Some lines for future work are still open. We are working on replacement policy to dynamically update the past queries kept in the index as cluster centers. This can be important if the distribution of the queries received by the system are very skewed in certain and short periods of time.

References

1. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: Caching content-based queries for robust and efficient image retrieval. In: Procs. of EDBT, pp. 780–790 (2009)
2. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Procs. of VLDB, pp. 426–435 (1997)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys 33, 273–321 (2001)
4. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity search. The metric space approach. Advances in Database Systems, vol. 32. Springer (2006)
5. Chavez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
6. Gil-Costa, V., Marin, M., Reyes, N.: Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms 7(1), 3–17 (2009)
7. Pedreira, O., Brisaboa, N.R.: Spatial selection of sparse pivots for similarity search in metric spaces. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 434–445. Springer, Heidelberg (2007)
8. Bustos, B., Pedreira, O., Brisaboa, N.: A dynamic pivot selection technique for similarity search in metric spaces. In: Procs. of SISAP, pp. 105–112. IEEE Press (2008)
9. Ares, L.G., Brisaboa, N.R., Esteller, M.F., Pedreira, O., Ángeles, S.: Places: Optimal pivots to minimize the index size for metric access methods. In: Procs. of SISAP, pp. 74–80. IEEE Press (2009)
10. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: A metric cache for similarity search. In: Procs. of LSDS-IR, pp. 43–50 (2008)
11. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: Similarity caching in large-scale image retrieval. Information Processing and Management (2011)
12. Skopal, T., Lokoc, J., Bustos, B.: D-cache: Universal distance cache for metric access methods. Transactions on Knowledge and Data Engineering 99 (2011)
13. Barrios, J., Bustos, B., Skopal, T.: Snake table: A dynamic pivot table for streams of k-nn searches. In: Navarro, G., Pestov, V. (eds.) SISAP 2012. LNCS, vol. 7404, pp. 25–39. Springer, Heidelberg (2012)
14. Marin, M., Gil-Costa, V., Bonacic, C.: A search engine index for multimedia content. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 866–875. Springer, Heidelberg (2008)
15. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007), `http://www.sisap.org/Metric_Space_Library.html`