Approximate Distributed Metric-Space Search

Veronica Gil-Costa
CONICET, University of San Luis, Argentina
Yahoo! Research Latin America

Mauricio Marin
DIINF, University of Santiago de Chile
Yahoo! Research Latin America

ABSTRACT
This paper proposes an approximate search algorithm for metric space search which is suitable for distributed search engines. The desired level of approximation is a parameter that can be dynamically set in accordance with the observed query traffic. At steady state search engine operation, the proposed algorithm calculates exact answers to queries whereas at peak traffic it calculates approximate answers. The search algorithm outperforms previous approaches in memory space usage and running time. Our experimental results show that responding approximate approaches in memory space usage and running time. Our experimental results show that responding approximate answers of very good quality reduces average running time per query in about 40%.

Categories and Subject Descriptors
H.2.8 [Database Applications]: [Image databases]

General Terms
Algorithms, Experimentation, Performance

Keywords
Approximate list of cluster, distributed algorithms.

1. INTRODUCTION
Metric spaces have been shown to be efficient and practical for performing similarity search on very-large collections of complex data objects such as images. Search queries are represented by objects of the same type to those stored in the database where, for example, one is interested in retrieving the top-k objects which are the most similar to a given query. The degree of similarity between two objects is calculated by an application-dependent function called the distance function, which is usually expensive to compute and thereby dominates the running time cost. Provided that the distance function satisfies a few metric-space properties, precomputed distances among objects can be used to index the database in order to reduce the average number of calls to this function during search.

The motivating application for this paper is distributed metric-space search engines where it is critical to efficiently cope with sudden peaks in query traffic. To prevent from saturation, the search engine temporarily responds with approximate answers for selected queries. Typically a large search engine is composed of one or more broker machines and a collection of \( P \) processors forming a distributed memory system. The broker 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. An index data structure devised to speed up query processing is evenly distributed on the \( P \) processors, and parallel search query processing is performed by sending the query to different processors where in each processor the arriving job can be processed by using several processor cores.

Smart index distribution onto processors and metric-space caches are key tools to let metric-space based search engines achieve a very efficient query throughput [12]. Recently an efficient metric-space cache strategy has been proposed in [9]. This approach allows efficient calculation of approximate answers to queries not found in cache. These calculations are performed by re-using object-to-object distances calculated when solving exactly previous queries stored in the cache. When the computed approximation is not good enough the query is calculated in an exact manner by using the distributed index. The cache can be kept in the broker or it can be kept distributed in the \( P \) processors. However, in either case, extra space must be employed to store and fastly retrieve pertinent object-to-object distances.

In this paper we propose a search strategy that avoids the extra memory space and query approximation calculation cost associated with the above cache based approach. The proposed strategy is a more efficient alternative since (1) it is able to fastly calculate a good approximation to any given query and (2) the search algorithm can continue from that point forward to get an exact answer for it. Performance is better since no distance evaluation calculations are wasted and, as opposed to the metric-space cache approach, the precision of approximations can be controlled at will as it is always possible to reach exact results by re-using previous distance calculations.

A number of index data structures have been proposed so far. The one upon which we elaborate this paper is the so-called list of clusters (LC) [4]. The LC is a very simple but efficient index strategy where the order in which the database objects are placed together in each cluster at construction time, is taken into consideration to prune com-
putations at search time. We have found that this feature can be further exploited to efficiently support approximated answers to queries. Namely, in this paper, we propose replacing the original LC search algorithm for a new search algorithm we have devised to let a distributed search engine compute approximate answers to metric-space queries in an on-line and efficient manner.

The remaining of this paper is organized as follows. Section 2 defines metric spaces, presents the LC index data structure and its original search algorithms, and review previous related work. Section 3 presents the proposed approximate search algorithms. Section 4 presents experimental results and Section 5 presents the final conclusions.

2. RELATED PREVIOUS WORK

2.1 Metric spaces

A metric space \((\mathcal{U}, d)\) is composed of a universe of valid objects \(\mathcal{U}\) and a distance function \(d : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}^+\) defined among them. The distance function determines the similarity between two given objects. The goal is, given a set of objects and a query, to retrieve all objects close enough to the query. This function holds several properties: strictly positive\((d(x, y) > 0)\) and if \(d(x, y) = 0\) then \(x = y\), symmetry \((d(x, y) = d(y, x))\), and the triangle inequality \((d(x, z) \leq d(x, y) + d(y, z))\). In this setting a database (instance) \(\mathcal{D}\) is simply a finite collection of objects from \(\mathcal{U}\).

There are two main queries of interest for this paper:

- **range search:** that retrieves all the objects \(u \in \mathcal{D}\) within a radius \(r\) of the query \(q\), that is: \(R_D(q, r) = \{u \in \mathcal{D} : d(q, u) \leq r\}\), and
- **k-nearest neighbors search:** retrieves the set \(kNN_D(q, k) \subseteq \mathcal{D}\) such that \(|kNN_D(q, k)| = k\) and, for every \(u \in kNN_D(q, k)\) and every \(v \in \mathcal{D}\SETMINUS kNN_D(q, k)\), it holds that \(d(q, u) \leq d(q, v)\).

We mainly focus on the efficient solution of k-nearest neighbors queries, since in the context of Search Engines one is more interested in presenting to the user a few relevant results rather than a large number of them.

Some data structures for metric spaces can be classified as cluster based techniques. 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. This technique has been used in the algorithms proposed in [4, 6, 15]. Good surveys on metric spaces can be found in [5, 18, 19].

2.2 The List of Clusters (LC) approach

This particular indexing data structure has been shown to be efficient in high-dimensional metric spaces [4] and it is the most basic building block used at different points in the proposed search engine architecture. In order to build the LC data structure we first choose a “center” \(c\) \(\in \mathcal{D}\) and a radius \(r_c\). The center ball \((c, r_c)\) is the subset of objects of \(\mathcal{D}\) which are at distance at most \(r_c\) from \(c\). We define

\[I_D(c, r_c) = \{u \in \mathcal{D} \SETMINUS \{c\} : d(c, u) \leq r_c\}\]

as the cluster of internal objects which lie inside the center ball \((c, r_c)\), and

\[E_D(c, r_c) = \{u \in \mathcal{D} : d(c, u) > r_c\}\]

as the external objects. The clustering process is recursively applied in \(E\).

There are two simple ways to divide the space: taking a fixed radius for each partition or using a fixed size. To ensure good load balance across processors, we consider partitions with a fixed size of \(k\) objects, thus the radius \(r_c\) is the maximum distance between the center \(c\) and its \(k\)-nearest neighbor. For practical reasons we actually store into a cluster at most \(k\) objects. This is so because during search it is more efficient to have all objects at distance \(r_c\) from a center \(c\) either inside the cluster or outside it, but never some part inside and the remaining one outside of the cluster.

The final LC structure is a list of triples \((c_i, r_{c_i}, I_i)\), that is \((center, radius, cluster)\). A center chosen first during construction has preference over the following ones when the balls are overlapped. All the objects that lie inside the ball of the first center are stored in its cluster \(I_i\), despite that they may also lie inside the clusters of subsequent centers.

During the processing of a search query \(q\) with radius \(r\), the idea is that if the first center is \(c\) and its radius is \(r_c\), we evaluate \(d(q, c)\) and add \(c\) to the result set if it is appropriate. Then, we scan exhaustively the cluster \(I_i\) only if the query ball \((q, r)\) intersects the center ball \((c, r_c)\). Next, we continue with the set \(E\) recursively. However, because of the asymmetry of the data structure, we can stop the search before traversing the whole list of clusters: If the query ball \((q, r)\) is totally and strictly contained in the center ball \((c, r_c)\), we do not need to traverse \(E\) since the construction process ensures that all the objects that are inside the query ball \((q, r)\) have been inserted in \(I_i\).

In [4] different heuristics have been presented to select the centers, and it has been experimentally shown that the best strategy is to choose the next center as the objects that maximizes the sum of distances to previous centers. Thus, in this work we use this heuristic to select the centers. Additionally, to reduce the number of distance evaluations at query time we store \(d(c_i, o_j)\) \(\forall o_j \in I_i\) [14].

2.3 Approximate Metric Space Algorithms

An approximated answer is formed by those objects which are close to the current query but not all of them are the \(k\) closest ones. The idea is that in case the quality of an approximated answer is adequate according to some given measure, the system can return it to the user. Some previous work on approximate similarity search can be found in [1, 3, 11, 16]. The work on [7] presents a permutation prefix index that is able to deliver good approximate answers to queries. However, it does not guarantee exact results and neither the \(k\) closest results for a query. Work related to loosely-coupled distributed systems can be found in [2, 17] for P2P networks, a setting which is completely different to ours as explained in [13].

Strategies that allow to retrieve approximate results using a metric space cache can be found in [9, 8, 10]. These algorithms ensure the retrieval of at least the object which is the closest one to a query \(q\). The remaining ones form part of the approximate answer to \(q\). They show that it is possible to determine whether some of the objects stored in the cache are among the \(k\) nearest neighbors of a query \(q\), which is not currently in cache. Otherwise it would be necessary to resort to the index to determine those objects.

More formally, let \((\mathcal{U}, d)\) be a metric space, let \(\mathcal{D} \subseteq \mathcal{U}\) be a database, let \(\mathcal{C}\) be a metric space cache on \((\mathcal{U}, d)\) and \(\mathcal{D}\), let
$q \in \mathcal{U}$ be a query and let $k \geq 1$. Then $r_q$ denotes the radius of the smallest hyper-sphere centered in $q$ which contains all objects in $kNN_D(q, k)$. We assume that the $k$ nearest neighbour is always unique, i.e., that $|R_D(q, r_q)| = k$. The safe radius $s_q$ of the query $q$ with respect to a query $q_i \in C$ is the radius $r_q$ minus the distance from $q$ to $q_i$, namely $s_q(q_i) = r_q - d(q, q_i)$. It holds that for $k' = |R_C(q, s_q(q_i))|$ (note that $k' \leq k$), $R_C(q, s_q(q_i)) = R_C(q, s_q(q_i)) = kNN_D(q, k')$.

Note that, every cached query $q_i$ gives complete knowledge of the metric space up to distance $r_q$ from $q$. If a query $q$ is inside the hypersphere centered in $q_i$ with radius $r_q$, 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 $q \in \mathcal{U}$ with respect to a query $q_i \in C$ is a positive value, then every object in the range query $R_D(q, s_q(q_i))$ is also in the cache $C$ and thus can be solved over 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 neighbours of $q$ in the whole database $D$. We use this safe radius definition in the next section.

3. APPROXIMATE SEARCH ALGORITHM

To deal efficiently with high query traffic and to reduce the number of distance evaluations, each $P$ distributed node retrieves $M$ approximate results where $k$ of them are the closest local. Those $M$ results are sent to the broker machine to select the top $M$ results. The index is assumed to be evenly distributed among the processors as in [12]. We extend the original LC search algorithms as follows. (1) We first determine the cluster $(c_i, r_{ci})$ containing the query object $q$. We call it main cluster. (2) Then, we compute the safe radius $sf = r_{ci} - d(q, c_i)$. Now we search in the main cluster for similar objects that fall inside the query ball $(q, sf)$. (3) To ensure the $k$ closest objects to the query we visit the clusters built before $c_i$ that overlap the query ball. If we again reach the cluster $c_j$ without the $k$ closest objects, we increase the safe radius by a factor of $\gamma$, we check whether the clusters built before and after $c_j$ intersect the new query ball and we search into those clusters. Previously processed objects are not re-analyze. The algorithm finishes when 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.

Figure 1 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 by the cluster $(c_4, r_4)$. Following the original LC algorithm, we compute the distance between $q$ and all previous centers to $c_4$ (i.e., centers $c_1$, $c_2$ and $c_3$). When we get to $c_4$, we realize that $q$ falls within this cluster. Distances to previous clusters were stored for later use. Then we compute the $sf$ and we check whether some previous clusters intersect $(q, sf)$. In this example, there are no clusters previous to $c_4$ intersecting the query ball so we do not go inside previous clusters. Thus, to get the $k$ nearest objects to $q$ we increase $sf$ by a factor of $\gamma$. The new query ball intersects the cluster $(c_5, r_5)$, so we have to compute the distance $d(q, c_5)$ and obtain the objects falling inside the query ball. Moreover, we have to re-enter into $(c_4, r_4)$ and check for objects falling inside the new query ball. We re-use the pre-computed distances. The second time we increase the query radius (namely $sf$) we have to go inside $(c_1, r_1)$. We also have to re-enter into clusters $c_2$ and $c_3$.

In another example, Figure 2 shows an LC index with three clusters. The identifiers of the centers describe the order of construction of the LC clusters. Figure 2(a) shows that the query ball $(q, sf)$ is completely contained by 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 Figure 2(b) we have to visit $c_1$ and $c_2$ to guarantee the $k$ nearest objects to the query.

The index LC, the query $q$, the total number of results $M$ and the number of exact results, meaning the $k$ closest objects to the query, are the parameters of the Algorithm 1. We perform approximate searches when $k < M$, and we retrieve the exact $k$ nearest objects when $k = M$. We define the variable $out$ to maintain the $M$ results to the query, and previous_centers as the set of centers being allocated before the main cluster that have to be visited to guarantee the $k$ nearest objects. We also set in line 3 the query radius $q.r = \infty$, which will be adjusted to the $k$-th closest object as we process the query. In lines 5-6, we search for the main cluster $(Mcl)$. In line 8 we search inside the main cluster for similar objects to the query. In line 10 we search inside previous clusters and subsequent clusters intersecting the query ball. Finally, in lines 11 and 12 we add approximate results if necessary.

Algorithm 2 shows the steps to perform the first phase of the approximate search algorithm. We compute the distance between the query and the centers $c_i$ until we find the main cluster. Lines 3-5 discard clusters that do not intersect the query ball. In lines 6-9 we check whether the center of the cluster falls within the query ball. In this case the center is added as part of the results and we update the query radius. The query radius is updated once we get $k$ objects in the variable $out$. In line 10 we compute the safe radius. Namely, the first cluster of the LC containing the query object. In lines 11-15 we check whether the current cluster...
Algorithm 1 Approximate search algorithm.

Search(LC, q, M, k)
1: out ← ∅ (Results)
2: previous_centers ← ∅
3: q.r ← ∞ (Query radius)
4: //——— First Phase ————
5: Mcl ← Mcl() (Information of the main cluster)
6: Search_for_Main_Cluster(LC, q, M, k, out, Mcl)
7: //——— Second Phase ————
8: Search_MCluster(Mcl, q, out, previous_centers, k)
9: //——— Third Phase ————
10: Complete_M(LC, q, k, s, previous_centers)
11: if (out.size() ≤ M) then
12: supplement the results with approximate answers
13: end if

Algorithm 2 Search for the main cluster Mcl.

Search_for_Main_Cluster(LC, q, M, k, out, Mcl)
1: for (each c ∈ LC) do
2: d ← d(q, ci)
3: if (out.size() = k AND d − q.r > rci) then
4: continue
5: end if
6: if (d < q.r) then
7: out.insert(ci)
8: update q.r
9: end if
10: r ← (rci − d)
11: if (r > 0) then
12: Mcl ← ci, Mcl ← d, update q.r
13: break
14: end if
15: end for

Algorithm 3 Search inside the Mcl cluster.

Search_MCluster(LC, q, M, k, out, previous_centers, Mcl)
1: dqc ← Mcl
2: for (each ci ∈ Cluster_Mcl) do
3: dco ← d(Mcl, ci) (pre-computed distance)
4: if (dco < dqc − Mcl OR dco > dqc + Mcl) then
5: continue
6: end if
7: d ← d(q, ci)
8: if (d < q.r) then
9: out.insert(ci)
10: end if
11: Add ci to approximated results
12: end if
13: end for
14: for (each ci ∈ LC AND ci < Mcl) do
15: if ((ci, rci) ∩ (q, q.r) <> ∅) then
16: previous_centers.insert(ci)
17: end if
18: end for

Algorithm 4 Third phase of the algorithm.

Complete_M(LC, q, M, k, out, previous_centers, Mcl)
1: subsequent_centers ← ∅
2: while (1) do
3: for (each ci ∈ previous_centers ∪ subsequent_centers ∪ Mcl) do
4: d ← d(q, ci) (pre-computed distance)
5: if (d − q.r > rci) then
6: continue
7: end if
8: Search_Cluster(ci, rci, d, q, out)
9: end for
10: if (out.size() ≥ k) then
11: sort out by distances
12: return
13: end if
14: else
15: q.r ← q.r + q.r · γ (modifies the query radius)
16: previous_centers ← ∅
17: for (each ci ∈ LC AND ci < Mcl) do
18: previous_centers ← centers intersecting q.r
19: end for
20: subsequent_centers ← ∅
21: for (each ci ∈ LC AND ci > Mcl) do
22: subsequent_centers ← post-Mcl centers
23: end for
24: end if
25: end while

4. EXPERIMENTAL RESULTS

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 (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 resorted to an actual AOL user 1-term query log and associated popular images with popular terms found in the log.

In all experiments we set \( Q = 10,000 \) the number of queries to be solved. The distance between two images was evaluated with weighted sum of the distances between each of the five MPEG-7 descriptors used. We used the Euclidean distance over each MPEG-7 descriptor.

We built two LC indexes. The first one named LC, uses the original construction algorithm with the heuristic proposed in [4] to select the centers of the clusters. The second index named LC-Log, uses information from previous queries to select the centers of the clusters. To this end, we generated a set with the \( k \) nearest objects to queries and then we selected from this set the centers of the clusters using the same heuristics presented in [4]. 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 LFU cache upon the LC index. The results are presented normalized to 1. KNN refers to exact \( k \)-NN queries with \( k = M \).

The y-axis of Figure 3(a) shows normalized number of queries. For both indexes most queries require a small safe radius close to 50. Also for most queries, the distance to the \( k \)-th object using \( k = 30 \) is close to 300. Results obtained for \( k = 20, 10 \) and \( k = 5 \) have the same trend. Namely, most \( k \)-th objects have a distance to their queries close to 250 or 300. Only, for \( k = 1 \) most queries report a distance to the \( k \)-th object less than 50.

Figure 3(b) shows the number of distance evaluations performed with different \( \gamma \) values. In this figure we show results for \( k = 1/2M \). The same behavior is observed for \( k = 3/4M \) and \( k = 1/4M \). As expected, the computing cost increases with the \( \gamma \) factor. With a high \( \gamma \) factor we have to perform more distance evaluations due to the query radius grows faster and therefore more objects must be compared against the query. Below 0.1 the results do not significantly improve.

Figure 3(c) shows the number of distance evaluations reported by the standard \( kNN \) algorithm, the standard \( kNN \) algorithm using an average query radius as the initial radius, the LC algorithm proposed in this work using the heuristic [4] and the LC-Log algorithm. The standard \( kNN \) algorithm works as follows. The initial query radius is set to infinite. Then, as we process the LC clusters using the original LC algorithm, the query radius is adjusted to the distance to the \( k \)-th object. The second algorithm (Average Radius) works the same way but the initial radius is set to the average query radius registered by a set of previous queries.

In Figure 3(c) the x-axis shows the \( M \)-values used. For \( k = M \) the proposed algorithm with both indexes (LC and LC-Log) outperforms the standard \( kNN \) search algorithm by 25% on average. The LC-Log presents a 5% better performance than the LC algorithm. For \( K = 1/4M \) and \( M = 10 \) the proposal is almost 40% better than the \( k \)-NN algorithm.

Figure 4(a) shows the quality of query results measured as the fraction of actual \( k \)-NN objects between the approximate \( M \) results. We use the precision formula (relevant objects \( \cap \) retrieved objects)/retrieved objects. For a small \( \gamma = 0.1 \) the quality of results for \( k = 1/2M \) and \( k = 1/4M \) are lower, because we have a smaller set of approximate objects from where to select the remaining \( M - k \) objects. From more detailed analysis we have found that these objects tend to have a low quality for the query.

Figure 4(b) shows the average number of clusters visited by all queries in order to retrieve the \( k \) closest objects to the query. With a \( \gamma = 0.5 \) the LC index requires to visit more clusters than the LC-Log index. This last one reports an improvement of almost 30% over the standard LC index. Using lower a \( \gamma = 0.1 \) the number of clusters visited by all queries is reduced because it allows to prune better the clusters to be visited. Moreover, the difference between the LC and LC-Log indexes is almost 15% (this figure is not included for lack of space). These results indicate which index is more convenient in terms of cache memory management, because by visiting fewer clusters we have to perform fewer replacement operations in the O.S. system cache memory.

Figure 4(c) compares the performance of the RCache, the QCache algorithms [9] running on the broker machine and our LC-Log algorithm running over one machine to present a fair comparison. In this experiment we set the cache size to 1%, 3% and 6% of the number of queries processed. 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 approximate metric index used by these algorithms also grows. Thus, a larger metric index requires more distance evaluations. With a cache size of 6% the LC-Log algorithm reduces by 60% the number of distance evaluations. Note that the LC-Log algorithm is independent of the cache size.

Figure 5 compares the performance of the algorithms with
up to 64 processors. The running time confirms the values obtained in previous figures. The LC-Log algorithm is about 40% better than the standard \( k \)-NN algorithm. Moreover, this figure reflects the advantage of the LC-Log algorithm of visiting fewer clusters than the LC algorithm.

5. CONCLUSIONS AND FUTURE WORK

We have presented an approximate algorithm for a distributed metric space search engine using the LC index. This algorithm avoids bottlenecks caused by expensive operations in the broker machine. It retrieves approximate results to queries ensuring that at least \( k \) of them are exact results. Experiments shows that our algorithms outperform the standard \( k \)-NN algorithm by 23\% in average when \( k = M \) and almost 40\% for a smaller \( k \) value.

For the near future, we plan to improve the performance of the proposed algorithm by devising a strategy able to automatically set the value of \( \gamma \) in accordance with the type of query and the level of query traffic.

6. REFERENCES


