# List of Twin Clusters: a Data Structure for Similarity Joins in Metric Spaces [*]

Rodrigo Paredes
*Depto. de Ciencias de la Computación*
*Universidad de Chile*
*raparede@dcc.uchile.cl*

Nora Reyes
*Depto. de Informática*
*Universidad Nacional de San Luis*
*nreyes@unsl.edu.ar*

## Abstract

*The metric space model abstracts many proximity or similarity problems, where the most frequently considered primitives are range and $k$-nearest neighbor search, leaving out the* similarity join*, an extremely important primitive. In fact, despite the great attention that this primitive has received in traditional and even multidimensional databases, little has been done for general metric databases.*

*We consider a particular type of similarity join: Given two sets of objects and a distance threshold $r$, find all the object pairs (one from each set) at distance at most $r$. For this sake, we devise a new metric index, coined* List of Twin Clusters*, which indexes both sets jointly (instead of the natural approach of indexing one or both sets independently). Our results show significant speedups over the basic quadratic-time naive alternative. Furthermore, we show that our technique can be easily extended to other similarity join variants, e.g., finding the $k$-closest pairs.*

## 1. Introduction

*Proximity* or *similarity searching* is the problem of, given a data set and a similarity criterion, finding elements of the set that are *close* or *similar* to a given query. This is a natural extension of the classical problem of exact searching. It is motivated by data types that cannot be queried by exact matching, such as multimedia databases containing images, audio, video, documents, and so on. In this new framework the exact comparison is just a type of query, while close or similar objects can be queried as well. There exist several computer applications where the concept of similarity retrieval is of interest. Some examples are machine learning and classification, image quantization and compression, text retrieval, computational biology and function prediction.

Proximity/similarity queries can be formalized using the metric space model [4, 8, 12, 13]. There is a universe $\mathbb{X}$ of *objects* and a *distance function* $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+ \cup \{0\}$ defined among them. Objects in $\mathbb{X}$ do not necessarily have coordinates (for instance, strings and images). In order to compare any two different objects we can (only) use the distance function $d$, which is a measure of object dissimilarity. Therefore, the smaller the distance between two objects, the more "similar" they are.

The distance satisfies the following properties that make $(\mathbb{X}, d)$ a *metric space*: $d(x, y) \geq 0$ (positiveness), $d(x, y) = d(y, x)$ (symmetry), $d(x, x) = 0$ (reflexivity) and $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

The typical scenario of the metric space search problem considers that there is a finite *database* or *dataset* $\mathbb{U} \subset \mathbb{X}$. Then, given a new object $q \in \mathbb{X}$, a proximity query consists in retrieving objects from $\mathbb{U}$ relevant to $q$. There are two basic proximity queries or primitives. The first is the *range query* $(q, r)$, which retrieves all the elements in $\mathbb{U}$ which are within distance $r$ to $q$. The second is the *$k$-nearest neighbor query* $NN_k(q)$, which retrieves the $k$ closest elements in $\mathbb{U}$ to $q$. These similarity queries can be trivially answered by performing $|\mathbb{U}|$ distance evaluations. Yet, as the distance is assumed to be expensive to compute, it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations and even I/O time. Thus, the goal is to structure the database so as to compute much fewer distances when solving proximity queries.

Naturally, we can consider other proximity operations. In fact, in this paper we focus on the *similarity join* primitive. To illustrate the concept, let us consider a headhunting recruitment agency. On the one hand, the agency has a dataset of resumes and profiles of many people looking for a job. On the other hand, the agency has a dataset of job profiles sought by several companies looking for employees. What the agency has to do is to to find *all the person-company pairs which share a similar profile*. Similarity joins have other applications such as data mining, data cleaning and data integration, to name a few.

During this work, we consider a particular type of similarity join: Given two sets of objects $A, B \subset \mathbb{X}$ and a dis-

[*] Supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; and AECI A/8065/07.

tance threshold $r \geq 0$, find all the object pairs (one from each set) at distance at most $r$. Formally, the similarity join $A \bowtie_r B$ between two finite sets $A = \left\{a_1, \ldots, a_{|A|}\right\}$ and $B = \left\{b_1, \ldots, b_{|B|}\right\}$ is the set of pairs

$$A \bowtie_r B = \{(a_i, b_j),\ a_i \in A, b_j \in B, d(a_i, b_j) \leq r\}.$$

If both sets coincide, we talk about the *similarity self join*.

The similarity join $A \bowtie_r B$ essentially translates into solving several range queries, where queries come from one set and objects relevant for each query come from the other. So, given the sets $A$ and $B$, a natural approach to compute $A \bowtie_r B$ consists in indexing one set and then solving range queries for each element from the other set. Moreover, following this approach we can also try indexing both sets independently in order to speedup the whole process.

Instead, we propose to *index both sets jointly*, which, to the best of our knowledge, is the first attempt following this simple idea. For this sake, based on Chávez and Navarro's *list of clusters* [3], we devise a new metric index, coined *list of twin clusters*. We carry out some experiments which show significant speedups over the basic quadratic-time naive alternative. Furthermore, we show that our technique can be easily extended to other variants of similarity join, for instance, finding the $k$-closest pairs.

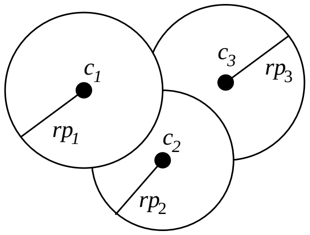
## 2. Related work

### 2.1. List of clusters

Let us briefly recall what a list of clusters (LC) is [3]. The LC splits the space into zones. Each zone has a center $c$ and stores both its radius $rp$ and the bucket $I$ of internal objects, that is, the objects inside the zone.

We start by initializing the set $E$ of external objects to $\mathbb{U}$. Then, we take a center $c \in E$ and a radius $rp$. The *center ball* of $(c, rp)$ is defined as $(c, rp) = \{x \in \mathbb{X},\ d(c, x) \leq rp\}$. Thus, the bucket $I$ of internal objects is defined as $I = E \cap (c, rp)$ and the set $E$ is updated to $E = E - I$. Next, the process is repeated recursively inside $E$. The construction process returns a list of triples $(c, rp, I)$, as shown in Figure 1.

This data structure is asymmetric, because the first center chosen has preference over the next ones in case of overlapping balls (see Figure 1). All the elements inside the ball of the first center ($c_1$ in the figure) are stored in the first bucket ($I_1$ in the figure), despite that they may also lie inside buckets of subsequent centers ($c_2$ and $c_3$ in the figure). In [3], authors consider many alternatives to select both the next center in the list and the zone radii. They have shown experimentally that the best performance is achieved when the zone has a fixed number of elements, so $rp$ is the covering radius of $c$ —that is, the distance from $c$ towards the furthest element in its zone—, and the next center is selected as the element maximizing the sum of distances to centers previously chosen. The brute force algorithm for constructing the list takes $O\big(n^2/m\big)$, where $m$ is the size of each zone.



$$(c_1, rp_1, I_1) \xrightarrow{E_1} (c_2, rp_2, I_2) \xrightarrow{E_2} (c_3, rp_3, I_3)$$

**Figure 1. The clusters obtained when the centers are chosen in this order: $c_1$, $c_2$ and $c_3$, and the resulting list of clusters.**

For a range query $(q, r)$ the list is visited zone by zone. We first compute the distance from $q$ to the center $c$, and report $c$ if $d(q, c) \leq r$. Then, if $d(q, c) - rp \leq r$ we search exhaustively the internal bucket $I$. Because of the asymmetry of the structure, $E$ (the rest of the list) is processed only if $rp - d(q, c) < r$. The search cost has a form close to $O(n^\alpha)$ for some $\alpha \in (0.5, 1)$ [3].

Recently, M. Mamede proposes the *recursive list of clusters* (RLC) [10], which can be seen as a dynamic version of the LC. The construction of a RLC of $n$ objects takes $O(n \log_\beta n)$ distance computations, for some $\beta \in (1, 2)$. Experimental results show that the RLC's search performance slightly improves upon the LC's in uniformly distributed vector spaces in $\mathbb{R}^D$, for $D \leq 12$.

### 2.2. Similarity joins

Given two sets of elements $A, B \subset \mathbb{X}$, the naive approach to compute the similarity join $A \bowtie_r B$ uses $|A| \cdot |B|$ distances computations between all the pairs of objects. This is usually called the *Nested Loop*.

In the case of multidimensional vector spaces $\mathbb{R}^D$, an important subclass of metric space, there are some alternatives [2, 1, 9]. In [2], authors solve similarity joins in $\mathbb{R}^2$ or $\mathbb{R}^3$ by indexing both datasets $A$ and $B$ with two R-trees, and then traverse both indices simultaneously to find the set of pairs of objects matching each other. In [1], authors present the *EGO-join* strategy. It divides the space with an $\epsilon$-grid, a lattice of hypercubes whose edges have size $\epsilon$, and uses different methodologies to traverse the grid. They show results for dimensions $D \leq 16$. In [9], authors give the *Grid-join* and the *EGO$^*$-join*, whose performances are at least one order of magnitude better than that of EGO-join in low dimension spaces. However, none of these alternatives is suitable for general metric spaces as they use coordinate information (unavailable in some metric spaces).

In metric spaces, a natural approach to solve this problem consists in indexing one or both sets independently (by using any metric index [4, 8, 12, 13]) and then solving range queries for all the involved elements over the indexed sets. In fact, this is the strategy proposed in [6], where authors use the D-index [5] in order to solve similarity self joins. Later, they present the eD-index, an extension of the D-index, and study its application to similarity self joins [7].

Finally, in [11], we give subquadratic algorithms to construct the $k$-nearest neighbor graph of a set $\mathbb{U}$, which can be seen as a variant of self similar join where we look for the $k$-nearest neighbors of each object in $\mathbb{U}$.

# 3. List of twin clusters

The basic idea of our proposal to solve the similarity join $A \bowtie_r B$ is to index the datasets $A$ and $B$ jointly in a single data structure. This is because we want to combine objects from different sets, and not to perform distance computations between objects of the same set.

We devise the *list of twin clusters* (LTC), a new metric index specially focused on the similarity join problem. As the name suggests, the LTC is based on Chávez and Navarro's *list of clusters* [3]. In spite of their experimental results, we have chosen to use clusters with fixed radius. Note that, if we had used the option of fixed size clusters, we would have obtained clusters of very different radii, especially in the case when the dataset sizes differ considerably.

Essentially, our data structure considers two list of overlapping clusters, which we call twin clusters (see Figure 2). So, when solving range queries, most of relevant objects would belong to the twin cluster of the object we are querying for. We also consider additional structures in order to speed up the whole process. The LTC's data structures are:

1. Two list of twin clusters $CA$ and $CB$. Cluster centers of $CA$ (resp. $CB$) belong to dataset $A$ (resp. $B$) and objects in its inner buckets belong to dataset $B$ (resp. $A$). Each cluster is a triple (center, internal bucket, radius).

2. A matrix $D_{c_a,c_b}$ with the distances computed from the centers of dataset $A$ towards the centers of dataset $B$.

3. Four arrays $dAmax, dAmin, dBmax$ and $dBmin$ storing the cluster identifier and the maximum or minimum distance for each object from a dataset towards all the cluster centers from the other dataset.

In order to compute the similarity join $A \bowtie_r B$ it suffices with solving range queries for objects from one dataset retrieving relevant objects from the other. Thus, without loss of generality, let us suppose that we are computing range queries for elements in $A$, and $|A| \geq |B|$.

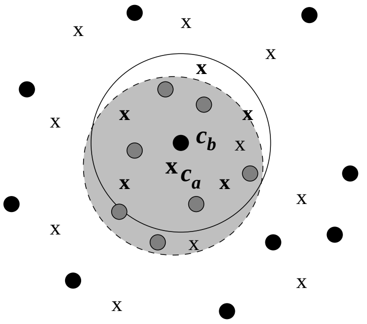


**Figure 2. The twin clusters.**

In Section 3.1, we show how to solve range queries using the LTC's structures. Next, in Section 3.2, we give the LTC construction algorithm. Finally, in Section 3.3, we give the LTC-join algorithm for computing $A \bowtie_r B$.

## 3.1. Solving range queries with the LTC index

We have to process three kinds of objects: *cluster centers*; *regular objects*, the ones indexed in any internal bucket; and *non-indexed objects*, the ones which are neither cluster centers nor regular ones.

**Solving cluster centers.** Let $\left(c_a^i, I_a^i, R_a^i\right)$ denote the $i$-th cluster from $CA$, and $c_b^i$ the $c_a^i$'s twin center. After constructing the LTC, each center $c_a^i \in A$ will has computed the distance towards all the objects $b \in B$ stored both inside its own internal bucket $\left(I_a^i\right)$ and inside the internal buckets of following centers. So, if the similarity join radius $r$ is lower than or equal to the LTC radius $R$ (that is, if $r \leq R$), in order to finish the range query for $c_a^i$, it is enough to verify whether there are relevant objects in the internal buckets $I_a^j$ of previous clusters, that is, for $j < i$.

Otherwise, as $r > R$, we need to review all the clusters in the list $CA$ to finish the range query for $c_a^i$. However, this can be seen as a particular case of solving a regular object, which we discuss soon.

Note that solving the cluster center $c_a^i$ is equivalent to solve a range query $\left(c_a^i, r\right)$ in the previous clusters of $CA$. Yet, we can avoid some distance computations using the LTC and the triangle inequality. We have to check the previous clusters $j < i$ in $CA$ only if $\left|d\left(c_a^j, c_b^i\right) - d\left(c_a^i, c_b^i\right)\right| \leq R_a^j + r$, where $R_a^j$ denotes the effective radius of the $j$-th cluster in $CA$. Otherwise, it is not relevant for $c_a^i$. Figure 3 illustrates. Inside a relevant cluster, we can still use the triangle inequality to avoid a direct comparison. Given an object $b$ in the cluster $\left(c_a^j, I_a^j, R_a^j\right)$, if $\left|d\left(c_a^j, c_b^i\right) - d\left(c_a^i, c_b^i\right)\right| - d\left(c_a^j, b\right) > r$, then $b$ is not relevant. Figure 4 depicts the algorithm. The distances $d(c_a^j, c_b^i)$ and $d(c_a^i, c_b^i)$ come from matrix $D_{c_a,c_b}$, and distances $d(c_a^i, b)$ and $d(c_a^j, b)$ from array $dBmin$.

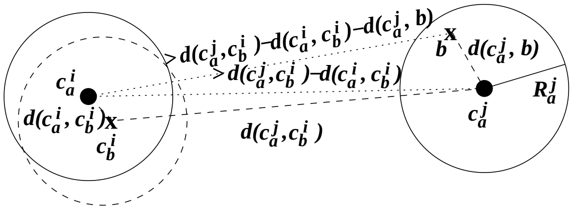

$>d(c_a^j, c_b^i) - d(c_a^i, c_b^i) - d(c_a^j, b)$
$>d(c_a^j, c_b^i) - d(c_a^i, c_b^i)$
$c_a^i$
$d(c_a^i, c_b^i)$
$x$
$c_b^i$
$d(c_a^j, c_b^i)$
$x$
$b$
$d(c_a^j, b)$
$R_a^j$
$c_a^j$

**Figure 3. Solving the cluster center $c_a^i$.**

```
rqCenter (Integer $i$, Radius $r$)
For each $b \in I_a^i$ Do // own cluster
    If $dBmin[b].distance \leq r$ Then Report $(c_a^i, b)$
For each $(c_a^j, I_a^j, R_a^j) \in CA, j = 1$ to $i - 1$ Do
    $d_1 \leftarrow D_{c_a,c_b}(c_a^i, c_b^i)$, $d_2 \leftarrow D_{c_a,c_b}(c_a^j, c_b^i)$
    If $|d_2 - d_1| \leq R_a^j + r$ Then For each $b \in I_a^j$ Do
        $lb \leftarrow |d_2 - d_1| - dBmin[b].distance$
        If $lb \leq r$ AND $d(c_a^i, b) \leq r$ Then Report $(c_a^i, b)$
```

**Figure 4. Range query for cluster centers.**

**Solving regular objects.** Assume that we are solving the range query for a regular object $a \in A$. Using the array $dAmin$, we determine which cluster $(c_b^i, I_b^i, R_b^i)$ object $a$ belongs to. Let $(c_a^i, I_a^i, R_a^i)$ be $c_b^i$'s twin cluster. Due to the LTC construction algorithm, it is likely that many objects relevant to $a$ belong to the twin internal bucket $I_a^i$. Thus, we start by checking $I_a^i$. As usual, we try to avoid distance computations by using the generalized triangle inequality.

Next, we check other clusters $(c_a^j, I_a^j, R_a^j)$ in $CA$. Once again, we check whether the distances between the centers allow us to neglect the cluster, that is, we only visit the $j$-th cluster if $|D_{c_a,c_b}(c_a^j, c_b^i) - d(a, c_b^i)| \leq R_a^j + r$. Otherwise, the $j$-th cluster does not contain relevant elements. We also use the triangle inequality to obtain a lower bound of the distances. The algorithm is depicted in Figure 5.

When $r > R$, **rqRegular** does not change. However, we process cluster centers with a variant of **rqRegular** which initializes the distance between an object to a center ($d_1$) to zero, as we are computing the range query for a center.

**Solving non-indexed objects.** As like the previous algorithms, we use distances between centers, distances to the closest and furthest center and the triangle inequality to lower bound distances, avoiding direct comparisons if we can. Figure 6 depicts the algorithm when non-indexed objects come from dataset $A$, where we only use the arrays $dAmax$ and $dAmin$. If they come from dataset $B$ we use arrays $dBmax$ and $dBmin$ and the situation is symmetric.

```
rqRegular (Object $a$, Radius $r$)
$(c_b^i, d_1) \leftarrow dAmin[a]$ // $c_b^i$ is the center of object $a$
$d_2 \leftarrow D_{c_a,c_b}(c_a^i, c_b^i)$ // $c_a^i$ is $c_b^i$'s twin
For each $b \in I_a^i$ Do // checking the twin cluster
    $d_3 \leftarrow dBmin[b].distance$, $d_s \leftarrow d_1 + d_2 + d_3$
    If $d_s \leq r$ Then Report $(a, b)$
    Else
        $lb \leftarrow 2\max\{d_1, d_2, d_3\} - d_s$
        If $lb \leq r$ AND $d(a, b) \leq r$ Then Report$(a, b)$
// checking other clusters, $d_1$ has not changed
For each $c_a^j \in CA, j \neq i$ Do
    $d_2 \leftarrow D_{c_a,c_b}(c_a^j, c_b^i)$, $lb \leftarrow |d_2 - d_1|$
    If $lb \leq R_a^j + r$ Then
        // we check objects in bucket $I_a^j$ of $c_a^j$
        For each $b \in I_a^j$ Do
            $lb \leftarrow |d_2 - d_1| - dBmin[b].distance$
            If $lb \leq r$ AND $d(a, b) \leq r$ Then Report$(a, b)$
```

**Figure 5. Range query for regular objects.**

```
rqNonIndexedA (Object $a$, Radius $r$)
$(c_b^{min}, d^{min}) \leftarrow dAmin[a]$
$(c_b^{max}, d^{max}) \leftarrow dAmax[a]$
For each $c_a^i \in CA$ Do // checking all the centers
    $d_1 \leftarrow D_{c_a,c_b}(c_a^i, c_b^{min})$, $d_2 \leftarrow D_{c_a,c_b}(c_a^i, c_b^{max})$
    $lb \leftarrow \max\{|d_1 - d^{min}|, |d_2 - d^{max}|\}$
    If $lb \leq R_a^i + r$ Then For each $b \in I_a^i$ Do
        $d_3 \leftarrow dBmin[b].distance$
        $lb_1 \leftarrow 2\max\{d_1, d_3, d^{min}\} - d_1 - d_3 - d^{min}$
        $lb_2 \leftarrow 2\max\{d_2, d_3, d^{max}\} - d_2 - d_3 - d^{max}$
        If $\max\{lb_1, lb_2\} \leq r$ AND $d(a, b) \leq r$ Then
            Report$(a, b)$
```

**Figure 6. Range query for non-indexed objects.**

### 3.2. Construction of the list of twin clusters

We have assumed that the construction of the LTC index is independent of the radius $r$ of the similarity join. Let $R$ be the nominal radius of each cluster in the LTC. The LTC construction process is as follows.

We start by initializing both lists of clusters $CA$ and $CB$ to empty, and for each object $a \in A$ we initialize $dA[a]$ to zero. We use the array $dA$ to choose cluster centers for the LTC (from the second to the last cluster).

Next, we choose the first center $c_a$ from the dataset A at random and we add to its internal bucket $I_a$ all the elements $b \in B$ such that $d(c_a, b) \leq R$. Then, we use the element $c_b \in I_a$ which minimizes the distance to $c_a$ as the center of the $c_a$'s twin cluster and add to its internal bucket $I_b$ all the elements $a \in A$ such that $d(a, c_b) \leq R$. (Figure 2 illustrates the concept of twin clusters.) For the other objects in $A$ we increase their $dA$ values by $d(c_b, a)$, that is, we update their sum of distances to centers in $B$. Once we process the datasets $A$ and $B$ we add the clusters $(c_a, \max_{b \in I_a}\{d(c_a, b)\}, I_a)$ and $(c_b, \max_{a \in I_b}\{d(a, c_b)\}, I_b)$ (center, effective radius, bucket) into the lists $CA$ and $CB$, respectively. Both centers $c_a$ and $c_b$, and elements inserted into the buckets $I_a$ and $I_b$ are removed from the datasets $A$ and $B$. From now on, we use the element maximizing $dA$ as the new center $c_a$, but we continue using the object $c_b \in I_a$ which minimizes the distance to $c_a$ as the center of the $c_a$'s twin cluster. We continue the process until one of the datasets gets empty.

During the process, we compute the distance to the closest and furthest cluster center for all the objects. For this sake we progressively update arrays $dAmin$, $dAmax$, $dBmin$ and $dBmax$ with the minimum and maximum distances known up to then. Note that for regular objects, $dAmin$ and $dBmin$ store the distance from the object to its respective center and the center identifier. Note also that we have to store and maintain the matrix $D_{c_a,c_b}$ to filter out elements when actually performing the similarity join. As these distances are computed during the LTC construction process, we can reuse them to fill this matrix. At the end, we only keep the maximum distances to cluster centers of non-indexed elements. Thus, if they come from dataset $A$ (resp. $B$), we discard the whole array $dBmax$ (resp. $dAmax$), and the distances for cluster centers and regular objects from $dAmax$ (resp. $dBmax$). Figure 7 depicts the construction algorithm.

According to the analysis performed in [3], the cost of constructing the LTC is $O((\max\{|A|, |B|\})^2/p^*)$, where $p^*$ is the expected bucket size.

### 3.3. Computing the LTC-join

Given the datasets $A$ and $B$, and a radius $R$, we compute the LTC index by calling **LTC**$(A, B, R)$.

Later, given a threshold $r$ we actually compute the join $A \bowtie_r B$. To do so, for each object $a \in A$ which is a cluster center or a regular object we solve the range query $(a, r)$. If $a$ is the $i$-th cluster center, we call **rqCenter**$(i, r)$. If $a$ is a regular object, we call **rqRegular**$(a, r)$. Finally, as all the matching pairs considering non-indexed objects are not yet reported, for each of them we call **rqNonIndexedA** (resp. **rqNonIndexedB**) if they come from dataset $A$ (resp. $B$).

## 4. Experimental results

For the experiments we have selected three different pairs of databases containing real data from two kinds of metric spaces. The results on these databases are representative of other metric spaces and databases we tested.

```
LTC (Dataset $A$, Dataset $B$, Radius $R$)
  $CA \leftarrow \emptyset, CB \leftarrow \emptyset$ // the lists of twin clusters
  For each $a \in A$ Do
      $dA[a] \leftarrow 0$ // sum of distances to centers in $B$
      // (closest and furthest center in $B$, distance)
      $dAmin[a] \leftarrow (\text{NULL}, \infty), dAmax[a] \leftarrow (\text{NULL}, 0)$
  For each $b \in B$ Do
      // (closest and furthest center in $A$, distance)
      $dBmin[b] \leftarrow (\text{NULL}, \infty), dBmax[b] \leftarrow (\text{NULL}, 0)$
  While $\min(|A|, |B|) > 0$ Do
      $c_a \leftarrow \text{argmax}_{a \in A}\{dA\}$, $A \leftarrow A - \{c_a\}$
      $c_b \leftarrow \text{NULL}$, $d_{c,c} \leftarrow \infty$, $I_a \leftarrow \emptyset$, $I_b \leftarrow \emptyset$
      For each $b \in B$ Do
          $d_{c,b} \leftarrow d(c_a, b)$
          If $d_{c,b} \leq R$ Then
              $I_a \leftarrow I_a \cup \{(b, d_{c,b})\}, B \leftarrow B - \{b\}$
              If $d_{c,b} < d_{c,c}$ Then $d_{c,c} \leftarrow d_{c,b}$, $c_b \leftarrow b$
          If $d_{c,b} < dBmin[b].distance$ Then
              $dBmin[b] \leftarrow (c_a, d_{c,b})$
          If $d_{c,b} > dBmax[b].distance$ Then
              $dBmax[b] \leftarrow (c_a, d_{c,b})$
      For each $a \in A$ Do
          $d_{a,c} \leftarrow d(a, c_b)$
          If $d_{a,c} \leq R$ Then
              $I_b \leftarrow I_b \cup \{(a, d_{a,c})\}, A \leftarrow A - \{a\}$
          Else $dA[a] \leftarrow dA[a] + d_{a,c}$
          If $d_{a,c} < dAmin[a].distance$ Then
              $dAmin[a] \leftarrow (c_b, d_{a,c})$
          If $d_{a,c} > dAmax[a].distance$ Then
              $dAmax[a] \leftarrow (c_b, d_{a,c})$
      // (center, effective radius, bucket)
      $CA \leftarrow CA \cup \{(c_a, \max_{b \in I_a}\{d(c_a, b)\}, I_a)\}$
      $CB \leftarrow CB \cup \{(c_b, \max_{a \in I_b}\{d(a, c_b)\}, I_b)\}$
  For each $c_a \in \mathbf{centers}(CA)$, $c_b \in \mathbf{centers}(CB)$ Do
      $D_{c_a,c_b}[c_a, c_b] \leftarrow d(c_a, c_b)$ // this distance has
      // already been computed, so we can reuse it
```

**Figure 7. LTC construction algorithm.**

*Face images:* a set of 1,016 761-dimensional feature vectors from a database of face images. Any quadratic form can be used as a distance, so we chose Euclidean distance as the simplest meaningful alternative.

The set have four face images from 254 people, thus we divide the set in two databases: one of them with three face images for each person (FACES762 for short, because it has 762 face images) and the other database with the fourth one (FACES254 for short).

*Strings:* a dictionary of words, where the distance is the *edit distance*, that is, the minimum number of character insertions, deletions and replacements needed to make two strings equal. This distance is useful in text retrieval to cope with spelling, typing and optical character recognition errors.

For this metric space we consider two pairs of databases: a subset of English words with a subset of Spanish words and the same subset of English words with a subset of the vocabulary of terms from a collection of documents. We use random selected subsets with 69,069 English words and 89,061 Spanish words from dictionaries of English and Spanish respectively, and 494,048 words from the vocabulary.

As we mention previously, we work with a particular similarity join: $A \bowtie_r B$. In all cases, we built the index with 100% of the objects considered for each database. All our results are averaged over 10 index constructions using different permutations of the datasets.

For the face images we have considered thresholds that retrieve on average 1, 5 or 10 relevant images from FACES762 per range query, when queries came from FACES254. This corresponds to radii $r$ equal to $0.27017$, $0.3567$ and $0.3768$, respectively. Due to the edit distance is discrete, for the strings we have used radii $r$ equal to 1, 2 and 3. In the joins between dictionaries this retrieve 0.05, 1.5 and 26 Spanish words per English word on average, respectively. In the joins between English dictionary and the vocabulary this retrieve 7.9, 137 and 1593 vocabulary term per English word on average, respectively.

If we have one database indexed with a metric index, we can trivially obtain the similarity join $A \bowtie_r B$ by executing a range query with threshold $r$ for each element from the other database. Because our join index is based on the LC, we also show the results obtained with this simple join algorithm having a LC built for one database. We named this join algorithm as LC-join.

If we have both databases indexed with metric indices, although we could apply the same trivial solution (that is, ignoring one of the indices), we can do better avoiding some distance calculations by using all the information we have from both indices. In order to compare our proposal with an example from this kind of algorithm, we consider to index both databases with LC with a join algorithm that uses all the information from the indices to improve the join cost. We named it as LC2-join.

Because we need to fix the radius before building the LC and LTC indices, we consider in each case different radii and we choose the radius which obtains better join cost for each alternative. In all cases, the radii considered should be greater than or equal to the largest $r$ used in $A \bowtie_r B$. Therefore, for face images we show radii $0.38$, $0.40$, $0.60$ and $0.80$, and for strings we show radii 3 to 6.

Figure 8 illustrates the behavior of LTC-join considering the different radii, in all pairs of databases. We do not include the construction costs of the LTC and LC indices, as we consider that they would be amortized among several join evaluations between the databases considering different thresholds. Besides, we verify that the construction costs of the LTC and the LC over one of the databases are similar.

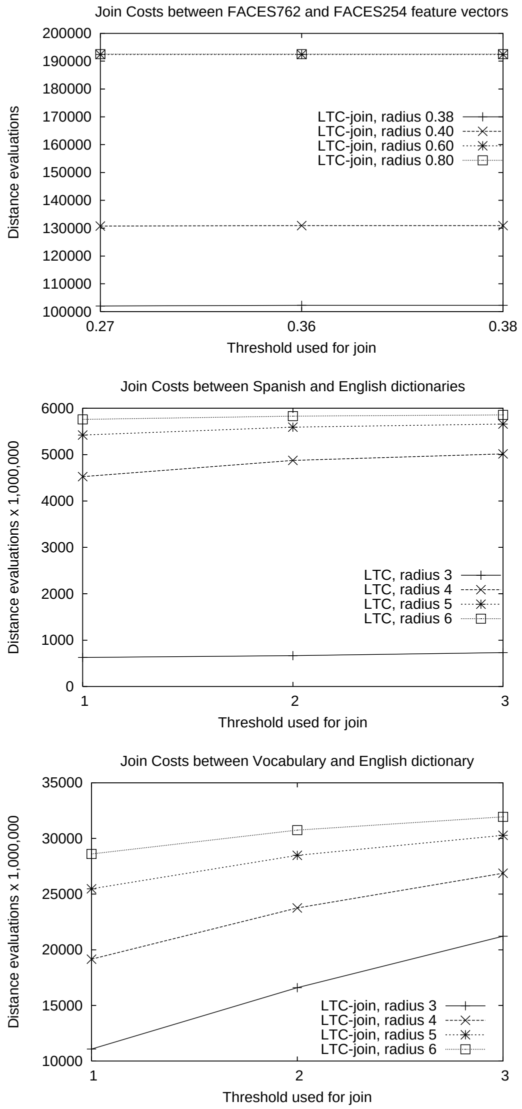


**Figure 8. Comparison among the different radii considered for the construction of the LTC index, for the face image databases (upper), the Spanish and English dictionaries (in the middle), and the English dictionary and the vocabulary (lower).**

The better results are obtained with the building radius $R$ closest to the greatest value of $r$ considered in each case.

Similar behavior have the LC2-join, but in case of LC-join the better radius can vary a little, for example the better radius for the join between both dictionaries is 5.

Figure 9 depicts a comparison among the three similarity join algorithms (without construction costs) for the three pairs of databases, using the better value of the building radius $R$ determined experimentally for each join algorithm.

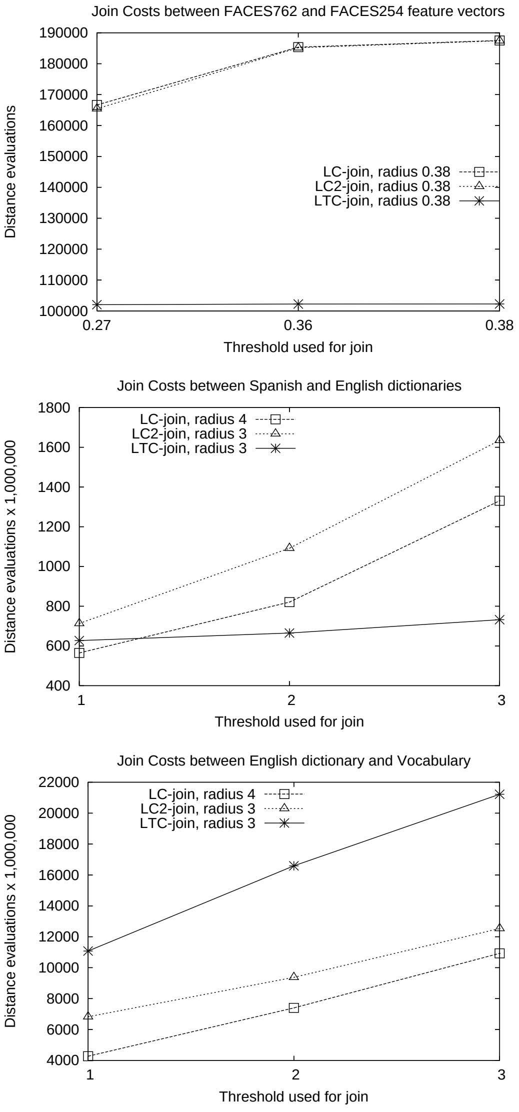


**Figure 9. Comparison among all the join algorithms considered, using in each case the better value experimentally determined for the building radius of the index. For face image databases (upper), the Spanish and English dictionaries (in the middle), and the English dictionary and the vocabulary (lower).**

We can observe that the LTC-join algorithm outperforms largely the other join algorithms considered in two of the pairs of databases used. For the join between the English dictionary and the vocabulary, LC-join and LC2-join beat us, despite LTC-join improves significantly over Nested Loop in all the thresholds used.

We suspect that this non-intuitive behavior showing that the simplest algorithm, LC-join, outperforms our LTC-join between the vocabulary and the English dictionary can be explained by taking into account the amount of non-indexed objects. In this case there is a 39% of non-indexed vocabulary terms, whereas in the others cases where the LTC-join is the best method, the percentage of non-indexed objects is lower. For instance, in the experiment of face images there is just a 2% of non-indexed faces and in the experiment of Spanish and English dictionaries there is a 23% of non-indexed words.

Table 1 gives the performance ratios of distance computations for the three pairs of databases. The values are computed according to this formula: $\frac{\text{join}-\text{LTC-join}}{\text{join}} \cdot 100$.

**Table 1. Performance ratio of the LTC-join for the three databases in all the thresholds used with respect to the other join methods.**

(a) Join between face image databases.

| Threshold | LC-join | LC2-join | Nested Loop |
|---|---|---|---|
| 0.27017 | 38% | 38% | 47% |
| 0.35670 | 44% | 44% | 47% |
| 0.37680 | 45% | 45% | 47% |

(b) Join between the Spanish and English dictionaries.

| Threshold | LC-join | LC2-join | Nested Loop |
|---|---|---|---|
| 1 | -11% | 12% | 89% |
| 2 | 19% | 39% | 88% |
| 3 | 45% | 55% | 87% |

(c) Join between the English dictionary and the vocabulary.

| Threshold | LC-join | LC2-join | Nested Loop |
|---|---|---|---|
| 1 | -159% | -62% | 67% |
| 2 | -124% | -76% | 51% |
| 3 | -94% | -69% | 38% |

## 5. Conclusions

In this work we show a new approach for similarity join algorithms consisting in indexing both datasets jointly. For this sake, we propose a new metric index, coined *list of twin clusters* (LTC). Our results not only show significant speedups over the basic quadratic-time naive alternative but also over the other two join algorithms, like LC-join and LC2-join, for two of the pairs of databases considered.

Our new LTC index stands out as a practical and efficient data structure to solve a particular case of similarity join as $A \bowtie_r B$, that can be used for pairs of databases in any metric space and therefore having a wide range of applications.

As work in progress, already being carried out within our research group, we studying:

- The similarity self join: although in this case it has no sense to build a LTC index, we plan to obtain another variant of LC designed specially for this kind of join.

- Optimization of LTC by evaluating internal distances: at construction time of the LTC index and when we evaluating the similarity join, we do not calculate any distance between elements from the same database. But, we have to analyze if we can improve the join costs if we calculate some internal distances in order to obtain better lower bounds of external distances (that is, distances between elements from both databases).

- The center selection: the best way to choose the twin center of one center is choosing the nearest object in the other database, yet we could study other ways to select a new center from the last twin center in order to represent the real dataset clustering by using the minimum number of cluster centers as possible. Furthermore, we suspect that choosing better centers we can reduce significantly the memory needed for the matrix of distances among centers.

- Different kinds of joins: we are developing algorithms to solve other kinds of similarity join over the LTC index or its variants. For instance, in order to find the $k$-closest pairs of objects from the datasets $A$ and $B$, (one from each set), it is enough to consider an additional $k$-element max-priority-queue $heap$ initialized with $k$ triples $(\text{NULL}, \text{NULL}, \infty)$ sorted by the third component. We start by replacing triples in $heap$ with all the precomputed distances in the LTC index such that they are smaller than the maximum distance stored in $heap$. Next, we continue by computing a similarity join of *decreasing* radius given by the maximum distance $r'$ in $heap$, and each time we find a pair of objects $a \in A$ and $b \in B$ such that $d(a,b) \leq r'$, we modify $heap$ by extracting its maximum and then inserting the triple $(a, b, d(a,b))$. When the computation finishes, $heap$ stores the $k$ pairs of the result.

- When using clusters of fixed radius, we experimentally observe that the first clusters are much more populated than the following ones. Moreover, we also can include the study of dynamic LTCs. Therefore, we also consider developing a version of the LTC similar to Mamedes's *recursive list of clusters* [10].

- Due to in some cases there exist many non-indexed objects, and apparently this harms the performance of the LTC-join, we also consider researching on alternatives to manage the non-indexed objects.

## References

[1] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'01)*, pages 379–388, 2001.

[2] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. 1993 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)*, pages 237–246, 1993.

[3] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[4] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[5] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.

[6] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. Similarity join in metric spaces. In *Proc. 25th European Conf. on IR Research (ECIR'03)*, LNCS 2633, pages 452–467, 2003.

[7] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using eD-index. In *Proc. 14th Intl. Conf. on Database and Expert Systems Applications (DEXA'03)*, LNCS 2736, pages 484–493, 2003.

[8] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[9] D. V. Kalashnikov and S. Prabhakar. Similarity join for low- and high- dimensional data. In *Proc. 8th Intl. Conf. on Database Systems for Advanced Applications (DASFAA'03)*, pages 7–16, 2003.

[10] M. Mamede. Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. In *Proc. 20th Intl. Symp. on Computer and Information Sciences (ISCIS'05)*, LNCS 3733, pages 843–853, 2005.

[11] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of $k$-nearest neighbor graphs in metric spaces. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA'06)*, LNCS 4007, pages 85–97, 2006.

[12] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, New York, 2006.

[13] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.