Solving Similarity Joins and Range Queries in Metric Spaces with the List of Twin Clusters[☆]

Rodrigo Paredes[a,*,1], Nora Reyes[b,1,2]

[a] *Yahoo! Research Latin America, Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*
[b] *Depto. de Informática, Universidad Nacional de San Luis. Ejército de los Andes 950, San Luis, Argentina.*

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 solve two variants of the similarity join problem: (1) *range joins*: Given two sets of objects and a distance threshold r, find all the object pairs (one from each set) at distance at most r; and (2) *k-closest pair joins*: Find the k closest object pairs (one from each set). For this sake, we devise a new metric index, coined *List of Twin Clusters* (LTC), which indexes both sets jointly, instead of the natural approach of indexing one or both sets independently. Finally, we show how to use the LTC in order to solve classical range queries. Our results show significant speedups over the basic quadratic-time naive alternative for both join variants, and that the LTC is competitive with the original *list of clusters* when solving range queries. Furthermore, we show that our technique has a great potential for improvements.

Key words: General metric spaces, Indexing methods, Similarity joins

1. Introduction

Proximity or *similarity searching* is the problem of, given a data set and a similarity criterion, finding the elements from the set that are close to a given query. This is a natural extension of the classical problem of exact searching. It has a vast number of applications. Some examples are:

- *Non-traditional databases.* New so-called *multimedia* data types such as images, audio and video cannot be meaningfully queried in the classical sense. In multimedia applications, all the queries ask for objects *similar* to a given one, whereas comparison for exact equality is very rare. In fact, no application will be interested in finding an audio segment exactly equal to a given one, or in retrieving an image pixelwise equal to the query image (as the probability that two different images are pixelwise equal is negligible unless they are digital copies of the same source). Some example applications are image, audio or video databases, face recognition, fingerprint matching, voice recognition, medical databases, and so on.

- *Text retrieval.* Huge text databases with low quality control have emerged (the Web being the most prominent example), and typing, spelling or OCR (optical character recognition) errors are commonplace in both the text and the queries. Documents containing a misspelled word are no longer retrievable by a correctly written query

[☆]Early parts of this work appeared in SISAP'08.
[*]Corresponding author. Telephone: (+562) 978 0678. Fax: (+562) 689 5531.
Email addresses: `raparede@dcc.uchile.cl` (Rodrigo Paredes), `nreyes@unsl.edu.ar` (Nora Reyes)
[1]Supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; and AECI Project A/8065/07, Spain.
[2]Supported in part by Fondecyt Grant 1-080019.

or vice versa. Thus, many text search engines aim to find text passages containing close variants of the query words. There exist several models of similarity among words (variants of the "edit distance" [29, 39]) which capture very well those kinds of errors. Another related application is spelling checkers, where we look for close variants of a misspelled word in a dictionary.

- *Information retrieval.* Although not considered as a multimedia data type, unstructured text retrieval poses problems similar to multimedia retrieval. This is because textual documents are in general not structured to easily provide the desired information. Although text documents may be searched for strings that are present or not, in many cases it is more useful to search them for semantic concepts of interest. The problem is basically solved by retrieving documents similar to a given query [45, 5], where the query can be a small set of words or even another document. Some similarity approaches are based on mapping a document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension. Similarity functions are then defined on that space. Notice, however, that as the vocabulary can be arbitrarily large, the dimensionality of this space is usually very high (thousands of coordinates).

- *Computational biology.* DNA and protein sequences are basic objects of study in molecular biology. They can be modeled as strings (symbol sequences), and in this case many biological quests translate into finding local or global similarities between such sequences in order to detect homologous regions that permit predicting functionality, structure or evolutionary distance. An exact match is unlikely to occur because of measurement errors, minor differences in genetic streams with similar functionality, and evolution. The measure of similarity used is related to the probability of mutations such as reversals of pieces of the sequences and other rearrangements (global similarity), or variants of edit distance (local similarity).

- There are many other applications, such as *machine learning and classification*, where a new element must be classified according to its closest existing element; *image quantization and compression*, where only some vectors can be represented and those that cannot must be coded as their closest representable point; *function prediction*, where we want to search for the most similar behavior of a function in the past so as to predict its probable future behavior; and so on.

All those applications have some common characteristics, captured under the *metric space model* [13, 26, 46, 51]. There is a universe $\mathbb{X}$ of *objects*, and a nonnegative *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). The distance function gives us a dissimilarity criterion to compare objects from the database. Thus, the smaller the distance between two objects, the more "similar" they are. This distance satisfies the following properties that make $(\mathbb{X}, d)$ a *metric space*:

$\forall\ x, y \in \mathbb{X}$,	$x \neq y \Rightarrow d(x, y) > 0$	strict positiveness,
$\forall\ x, y \in \mathbb{X}$,	$d(x, y) = d(y, x)$	symmetry,
$\forall\ x \in \mathbb{X}$,	$d(x, x) = 0$	reflexivity,
$\forall\ x, y, z \in \mathbb{X}$,	$d(x, z) \leq d(x, y) + d(y, z)$	triangle inequality.

These properties hold for many reasonable similarity functions.

Typically, we have a finite *database* or *dataset* $\mathbb{U} \subset \mathbb{X}$, which is a subset of the universe of objects and can be preprocessed to build an *index*. Later, 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:

Range query (q, r)**:** Retrieve all the elements in $\mathbb{U}$ which are within distance r to q. That is, $(q, r) = \{x \in \mathbb{U}, d(x, q) \leq r\}$.

k-**Nearest neighbor query** $NN_k(q)$**:** Retrieve the k elements from $\mathbb{U}$ closest to q. That is, $NN_k(q)$ such that $\forall x \in NN_k(q)$, $y \in \mathbb{U} \setminus NN_k(q)$, $d(q, x) \leq d(q, y)$, and $|NN_k(q)| = k$.

Given the database $\mathbb{U}$, these similarity queries can be trivially answered by performing $|\mathbb{U}|$ distance evaluations. However, as the distance is assumed to be expensive to compute (think, for instance, in comparing two fingerprints), 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 ultimate goal is to structure the database so as to compute many 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; that is, given two datasets, finding pairs of objects (one from each set) satisfying some similarity predicate. If both datasets coincide, we talk about the *similarity self join*. To illustrate this 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 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. Despite the great attention that this primitive has received in traditional and even multidimensional databases [9, 30, 6, 28] little has been done for general metric databases [19, 20].

In this work, we start by considering a variant of similarity join, the *range join*[3]: Given two datasets $A, B \subset \mathbb{X}$ and a distance threshold $r \geq 0$, find all the object pairs at distance at most r. Formally, given two finite datasets $A = \{a_1, \ldots, a_{|A|}\}$ and $B = \{b_1, \ldots, b_{|B|}\}$, the range join $A \bowtie_r B$ is the set of pairs

$$A \bowtie_r B = \{(a, b),\ a \in A, b \in B, d(a, b) \leq r\}. \tag{1}$$

The range join essentially translates into solving several range queries, where queries come from one set and objects relevant for each query come from the other. Thus, a natural approach to compute $A \bowtie_r B$ consists in indexing one set and later solving range queries for each element from the other. Moreover, following this approach we can also try indexing both sets independently in order to speed up 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* (LC) [12], we devise a new metric index, coined *list of twin clusters* (LTC).

Next, we show how to use the LTC in order to compute another variant, the *k-closest pair join*: Given two datasets A and B, find the k closest object pairs. Formally, the k-closest pair join $A \bowtie_k B$ is a k-element set of pairs where for all pairs $(a, b) \in A \bowtie_k B$, $a \in A$, $b \in B$ and for all pairs $(u, v) \in ((A \times B) \setminus (A \bowtie_k B))$, $u \in A$, $v \in B$, then $d(a, b) \leq d(u, v)$. In case of ties we choose any k-element set of pairs that satisfies the condition.

Finally, we show how to use the LTC in order to solve basic range queries for objects $q \in \mathbb{X}$ retrieving relevant objects from $(A \cup B)$. That is, use the LTC not only as an index to solve similarity joins but also as an index to solve the basic similarity primitives.

Afterwards we carry out an experimental evaluation of the LTC approach in order to verify that both similarity join variants significantly improve upon the basic quadratic-time naive alternative, and also that the LTC is competitive with the classical LC when solving range queries. Furthermore, we show that our technique has a great potential for improvements.

This paper is organized as follows. In the next section we review related work both in the list of clusters and similarity joins. Then, in Section 3 we describe the LTC, its basic operations and its construction; and in Section 4, how to use it in order to compute range joins, k-closest pair joins, and general range queries. Experimental results are shown in Section 5. Finally, in Section 6 we draw our conclusions and future work directions. An early version of this work appeared in [43].

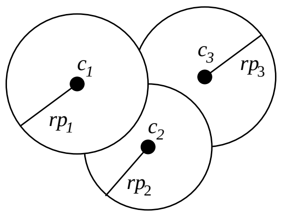
2. Related work

2.1. *List of clusters*

Let us briefly recall what a list of clusters [12] is. 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, whose value depends on whether the number of objects in the bucket is fixed or not. 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 $E \leftarrow E \setminus I$. Next, the process is repeated recursively inside E. The construction process returns a list of triples (c, rp, I) (center, radius, bucket), as shown in Fig. 1(a).

[3]Even though other authors have named this operation similarity join, we have called it range join to differentiate it from the other join variant, the k-closest pair join.



$$\begin{tikzcd}(c_1, rp_1, I_1) \arrow[r, "E_1"] & (c_2, rp_2, I_2) \arrow[r, "E_2"] & (c_3, rp_3, I_3)\end{tikzcd}$$

(a) A list of cluster.

$$\begin{tikzcd}
(c_1, r, I_1) \arrow[r, "E_1"] \arrow[d] & (c_2, r, I_2) \arrow[r, "E_2"] \arrow[dr] & (c_3, r, I_3) \\
(c_{1,1}, r, I_{1,1}) \arrow[r, "E_{1,1}"] & (c_{1,2}, r, I_{1,2}) \arrow[d] & (c_{2,1}, r, I_{2,1}) \\
 & (c_{1,2,1}, r, I_{1,2,1}) &
\end{tikzcd}$$

(b) A recursive list of clusters.

Figure 1: In (a), the clusters obtained when the centers are chosen in this order: c_1, c_2 and c_3, and the resulting list of clusters. In (b), a recursive list of cluster.

This data structure is asymmetric, because the first center chosen has preference over the next ones in case of overlapping balls (see Fig. 1(a)). 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 [12], the authors consider many alternatives to select both the zone radii and the next center in the list. They have experimentally shown 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 $O\left(n^2/m\right)$, where m is the size of each zone.

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 exhaustively search 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 $\alpha \in (0.5, 1)$ [12].

Recently, M. Mamede proposed the *recursive list of clusters* (RLC) [33], which can be seen as a dynamic version of the LC. The RLC is composed by clusters of fixed radius, so the number of objects of each cluster can differ. In fact, it can be experimentally verified that first clusters are very densely populated, whereas last ones often contain only the center. The RLC's construction algorithm is very similar to the LC's. Each RLC node is a triple (c, r, I) (center, fixed radius, bucket). Once we select the set I of objects for the current node, we continue adding nodes to the RLC by processing the rest of the objects. The difference comes from the following key fact. If the size of the internal bucket is big enough, say, greater than some small constant m, we recursively build a RLC for the elements of I. Fig. 1(b) shows a RLC. This algorithm takes $O(n \log_\beta n)$ distance computations to construct a RLC of n objects for some $\beta \in (1, 2)$, which is better than the one for LC. The search algorithm has to be slightly modified to support the fact that the bucket I can be a set of at most m elements, or a RLC itself. 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 datasets $A, B \subset \mathbb{X}$, the naive approach to compute the similarity joins $A \bowtie_r B$ or $A \bowtie_k B$ 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 spaces, there are some alternatives [9, 30, 6, 28]. In [9], the authors solve range joins in $\mathbb{R}^2$ or $\mathbb{R}^3$ by indexing both datasets A and B with two R-trees [23], and then traverse both indices simultaneously to find the set of pairs of objects matching each other. In [30], the authors used the hash-join idea to compute spatial joins (that is, for low dimension vector spaces). The idea consists in performing the similarity join computation in two phases. In the first phase, both datasets are partitioned into buckets with similar spatial decomposition (however, each object can be mapped into multiple buckets), and in the second phase, the buckets are joined in order to produce the outcome. The buckets are built by using a variant of the R-tree called *seeded tree*, which is studied in detail in [31]. In [6], the authors present the *EGO-join* strategy. It divides the space with an ϵ-grid, a lattice of hypercubes whose edges have size ϵ, and uses different methodologies to traverse the grid. They show results for dimensions $D \leq 16$. In [28], the 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 metric spaces, as they use coordinate information that is not necessarily available in general metric spaces.

In metric spaces, a natural approach to solve this problem consists in indexing one or both sets independently (by using any from the plethora of metric indices [3, 4, 7, 8, 10, 11, 12, 14, 15, 17, 21, 27, 36, 37, 38, 40, 41, 44, 47, 48, 49, 50], most of them compiled in [13, 26, 46, 51]), and then solving range queries for all the involved elements over the indexed sets. In fact, this is the strategy proposed in [19], where the authors use the *D-index* [18] 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 [20].

With respect to the *k*-closest pair join, in the case of multidimensional vector spaces, there are some approaches that rely on the coordinate information to compute approximated results [32, 1, 2]. However, as far as we know, there is no previous attempt to compute the join $A \bowtie_k B$ in the metric space context.

Finally, in [42], the authors 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 $\mathbb{U}$.

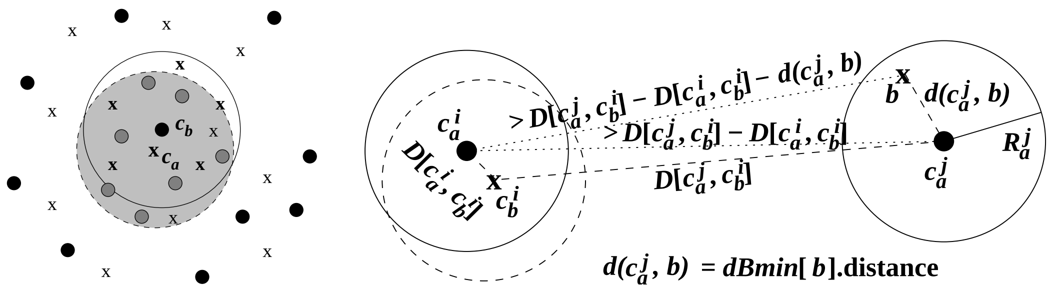
3. List of twin clusters

The basic idea of our proposal to solve the similarity join problem 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 have devised 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* [12]. In spite of their experimental results, we have chosen to use clusters with fixed radius. Note that, had we 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 lists of overlapping clusters, which we call twin clusters (see Fig. 2(a)). Each cluster is a triple (center, effective radius, internal bucket). Following the LC idea, we have chosen that every object being a center is not included in its twin bucket. So, when solving range queries, most of the relevant objects would belong to the twin cluster of the object we are querying for. We have also considered additional structures in order to speed up the whole process. The LTC's data structures are:

1. Two lists of twin clusters $\mathcal{CA}$ and $\mathcal{CB}$. Cluster centers of $\mathcal{CA}$ (resp. $\mathcal{CB}$) belong to dataset A (resp. B) and objects in its inner buckets belong to dataset B (resp. A).
2. A matrix D with the distances computed from all the centers from dataset A towards all the centers from 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.


c_b
c_a
c_a^i
$D[c_a^i, c_b^i]$
c_b^i
$> D[c_a^j, c_b^i] - D[c_a^i, c_b^i] - d(c_a^j, b)$
$> D[c_a^j, c_b^i] - D[c_a^i, c_b^i]$
$D[c_a^j, c_b^i]$
b
$d(c_a^j, b)$
R_a^j
c_a^j
$d(c_a^j, b)$ = $dBmin[b]$.distance

(a) The twin clusters. (b) Solving the cluster center c_a^i.

Figure 2: In (a), the twin clusters overlap each other. In (b), using the stored distances to solving the cluster center c_a^i.

We compute both similarity join variants by solving range queries for objects from one dataset retrieving relevant objects from the other. 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. From now on, r denotes the similarity join radius, and R the radius used to index both datasets A and B jointly with the LTC.

3.1. *Solving range queries with the LTC index*

We have to solve range queries for 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.

To understand the concept of non-indexed objects, we have to take into account that when constructing the LTC not all the objects get inside any of the twin clusters. This is because the LTC construction finishes when one of the datasets gets empty, as will be explained in Section 3.2. So, all the objects remaining in the other dataset turn into the set of *non-indexed objects*. These objects are not fully indexed in the LTC. In fact, we only store the distances from them towards the closest and furthest centers. (Later, we use these distances in order to try to avoid further computation when solving similarity joins and range queries.)

3.1.1. *Solving cluster centers*

Let (c_a^i, R_a^i, I_a^i) denote the i-th cluster of CA, and c_b^i the c_a^i's twin center. After constructing the LTC, each center $c_a^i \in A$ has been compared with all the objects $b \in B$ stored both inside its own internal bucket I_a^i and inside the buckets of following centers. So, if the similarity join radius is lower than or equal to the LTC construction radius (that is, if $r \leq R$), in order to solve the range query for c_a^i, we need to verify whether the following objects are relevant: (1) its twin center, (2) objects inside their own internal bucket, and (3) objects in the buckets of previous clusters.

Otherwise, as $r > R$, we would need to review not only regular objects but also cluster centers of *all* the clusters in the list CA to finish the range query for c_a^i.

When reviewing the previous clusters, we can avoid some distance computations using the LTC and the triangle inequality, see Fig. 2(b). We have to check the previous clusters (c_a^j, R_a^j, I_a^j), $j < i$, only if $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| \leq R_a^j + r$; else, the cluster (c_a^j, R_a^j, I_a^j) is not relevant for c_a^i. Inside a relevant cluster, we can still use the triangle inequality to avoid a direct comparison. Given an object b in the bucket I_a^j, if $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| - dBmin[b]$.distance $> r$, then b is not relevant. Fig. 3 depicts the algorithm.

```
rqCenter (Integer $i$, Radius $r$)
1.  If $D[c_a^i, c_b^i] \leq r$ Then Report $(c_a^i, c_b^i)$ // twin center
2.  For each $b \in I_a^i$ Do // own cluster
3.      If $dBmin[b].distance \leq r$ Then Report $(c_a^i, b)$
4.  For each $(c_a^j, R_a^j, I_a^j) \in CA$,   $j \leftarrow 1$ to $i-1$ Do // previous clusters
5.      If $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| \leq R_a^j + r$ Then
6.          For each $b \in I_a^j$ Do // internal bucket
7.              If $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| - dBmin[b].distance \leq r$ AND $d(c_a^i, b) \leq r$
8.                  Then Report $(c_a^i, b)$
```

Figure 3: Range query for cluster centers.

3.1.2. *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, R_b^i, I_b^i) the object a belongs to. So, we verify if c_b^i is relevant. Let (c_a^i, R_a^i, I_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 check the objects within I_a^i. Next, we check other clusters (c_a^j, R_a^j, I_a^j) in CA, and their respective twin centers. When $r < R$, it is enough to check regular objects of previous clusters (as cluster centers are necessarily further than r), otherwise, we would also need to check the cluster centers.

We use $|D[c_a^j, c_b^i] - dAmin[a]$.distance$|$ to lower bound the distance between a and c_a^j. So, we can lower bound the distance between a and c_b^j (the twin center of c_a^j) with $|D[c_a^j, c_b^i] - dAmin[a]$.distance$| - D[c_a^j, c_b^j]$, by using the generalized triangle inequality. Moreover, if we still need to compute $d(a, c_b^j)$, we use this computation to (hopefully) improve the lower bound of the distance between a and c_a^j. Finally, we check whether the current lower bound allows us to neglect the j-th cluster, that is, we only visit it if the bound is lower than or equal to $R_a^j + r$; else, it does not contain relevant elements. The algorithm is depicted in Fig. 4.

```
rqRegular (Object $a$, Radius $r$)
1.  $(c_b^i, d_1) \leftarrow dAmin[a]$ // obtaining the center $c_b^i$ and the distance
2.  If $d_1 \le r$ Then Report $(a, c_b^i)$ // we check $c_b^i$, its twin cluster is $(c_a^i, R_a^i, I_a^i)$
3.  $d_2 \leftarrow D[c_a^i, c_b^i]$ // $d_2$ is the distance between twin centers
4.  For each $b \in I_a^i$ Do // checking the twin bucket
5.      $d_3 \leftarrow dBmin[b]$.distance, $d_s \leftarrow d_1 + d_2 + d_3$
6.      If $d_s \le r$ Then Report $(a, b)$
7.      Else If $2\max\{d_1, d_2, d_3\} - d_s \le r$ AND $d(a, b) \le r$ Then Report$(a, b)$
    // checking other clusters, $d_1$ has not changed
8.  For each $(c_a^j, R_a^j, I_a^j) \in CA$,  $j \leftarrow 1$ to $|CA|$, $j \ne i$ Do
9.      $d_2 \leftarrow D[c_a^j, c_b^i]$, $lb \leftarrow |d_2 - d_1|$, $d_3 \leftarrow D[c_a^j, c_b^j]$ // $c_b^j$ is $c_a^j$'s twin center
10.     If $lb - d_3 \le r$ Then // first, we check $c_b^j$
11.         $d_4 \leftarrow d(a, c_b^j)$, $lb \leftarrow \max\{lb, |d_4 - d_3|\}$ // and update $lb$ if we can
12.         If $d_4 \le r$ Then Report $(a, c_b^j)$
13.     If $lb \le R_a^j + r$ Then // next, we check objects in bucket $I_a^j$
14.         For each $b \in I_a^j$ Do
15.             If $lb - dBmin[b]$.distance $\le r$ AND $d(a, b) \le r$ Then Report$(a, b)$
```

Figure 4: Range query for regular objects.

In the pseudocode of Fig. 4 we do not care whether the cluster is previous or not, so when $r > R$ **rqRegular** does not change.

3.1.3. Solving non-indexed objects

We need to check all the clusters in CA and their twin centers. As in 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. Fig. 5 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. When $r > R$ **rqNonIndexedA/B** does not change.

```
rqNonIndexedA (Object $a$, Radius $r$)
1.  $(c_b^{min}, d^{min}) \leftarrow dAmin[a]$, $(c_b^{max}, d^{max}) \leftarrow dAmax[a]$
2.  For each $(c_a, R_a, I_a) \in CA$ Do // checking all the clusters
3.      $d_1 \leftarrow D[c_a, c_b^{min}]$, $d_2 \leftarrow D[c_a, c_b^{max}]$, $lb \leftarrow \max\{|d_1 - d^{min}|, |d_2 - d^{max}|\}$
4.      $d_3 \leftarrow D[c_a, c_b]$ // $c_b$ is twin center of $c_a$
5.      If $lb - d_3 \le r$ Then // first, we check $c_b$
6.          $d_4 \leftarrow d(a, c_b)$, $lb \leftarrow \max\{lb, |d_4 - d_3|\}$ // and update $lb$ if we can
7.          If $d_4 \le r$ Then Report $(a, c_b)$
8.      If $lb \le R_a + r$ Then // the cluster could be relevant
9.          For each $b \in I_a$ Do // next, we check the bucket $I_a$
10.             $d_3 \leftarrow dBmin[b]$.distance
11.             $lb_1 \leftarrow 2\max\{d_1, d_3, d^{min}\} - d_1 - d_3 - d^{min}$
12.             $lb_2 \leftarrow 2\max\{d_2, d_3, d^{max}\} - d_2 - d_3 - d^{max}$
13.             If $\max\{lb_1, lb_2\} \le r$ AND $d(a, b) \le r$ Then Report$(a, b)$
```

Figure 5: 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, we remove c_b from I_a, and add to its internal bucket I_b all the elements $a \in A$ such that $d(a, c_b) \leq R$. (Fig. 2(a) illustrates the concept of twin clusters.) For 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 a regular object $a \in A$ (resp $b \in B$), array *dAmin* (resp. *dBmin*) stores its respective center $c_b \in B$ (resp. $c_a \in A$) and the distance from the object to that center.

Note also that we have to store and maintain the matrix D in order to filter out elements when actually performing similarity joins and range queries. 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*). We do this in the auxiliary triple *nonIndexed* (*label*, *set*, *array*). If the dataset B gets empty, then *nonIndexed* $\leftarrow$ ("A", A, *dAmax*), discarding array *dBmax*; otherwise, we discard array *dAmax*, so *nonIndexed* $\leftarrow$ ("B", B, *dBmax*). Fig. 6 depicts the construction algorithm.

According to the analysis performed in [12], the cost of constructing the LTC is $O((\max\{|A|, |B|\})^2/p^*)$, where p^* is the expected bucket size.

4. Using the LTC index

As there is an underlying symmetry in the join computation, we assume, without loss of generality, that we are computing range queries for elements in A, and $|A| \geq |B|$. (Otherwise, we swap the datasets.) In Section 4.1, we give the LTC-join algorithm for computing range joins $A \bowtie_r B$. Next, in Section 4.2, we compute k-closest pair joins $A \bowtie_k B$ by simulating them as range joins with decreasing radius. Finally, in Section 4.3, we show how to solve basic range queries using the LTC. These three sections assume that given the datasets A and B, and a radius R, we have previously computed the LTC index by calling **LTC**(A, B, R).

4.1. Computing the range join

Given a threshold r we actually compute the range join $A \bowtie_r B$, by traversing both lists CA and CB. For cluster centers (from CA) we call **rqCenter**, and for regular objects (from buckets in CB) we call **rqRegular**. Finally, as all the matching pairs considering non-indexed objects are not yet reported, by using the auxiliary triple *nonIndexed* we determine which dataset non-indexed objects come from, and for all of those objects we call **rqNonIndexedA** or **rqNonIndexedB**, accordingly. The algorithm is depicted in Fig. 7.

4.2. Computing the k-closest pair join

The basic idea is to compute the k-closest pair join $A \bowtie_k B$ as if it were a range join with decreasing radius. For this sake, we need an additional *k-element* priority queue *heap* to store triples of the form (object, object, distance) sorted increasingly by the third component. We initialize *heap* with k triples (NULL, NULL, ∞).

```
LTC (Dataset $A$, Dataset $B$, Radius $R$)
1.  $CA \leftarrow \emptyset, CB \leftarrow \emptyset$ // the lists of twin clusters
2.  For each $a \in A$ Do
3.      $dA[a] \leftarrow 0$ // sum of distances to centers in $B$
        // (closest and furthest center in $B$, distance)
4.      $dAmin[a] \leftarrow (\text{NULL}, \infty), dAmax[a] \leftarrow (\text{NULL}, 0)$
5.  For each $b \in B$ Do
        // (closest and furthest center in $A$, distance)
6.      $dBmin[b] \leftarrow (\text{NULL}, \infty), dBmax[b] \leftarrow (\text{NULL}, 0)$
7.  While $\min(|A|, |B|) > 0$ Do
8.      $c_a \leftarrow \text{argmax}_{a \in A}\{dA\}, A \leftarrow A \setminus \{c_a\}$
9.      $c_b \leftarrow \text{NULL}, d_{c,c} \leftarrow \infty, I_a \leftarrow \emptyset, I_b \leftarrow \emptyset$
10.     For each $b \in B$ Do
11.         $d_{c,b} \leftarrow d(c_a, b)$
12.         If $d_{c,b} \leq R$ Then
13.             $I_a \leftarrow I_a \cup \{(b, d_{c,b})\}, B \leftarrow B \setminus \{b\}$
14.             If $d_{c,b} < d_{c,c}$ Then $d_{c,c} \leftarrow d_{c,b}$, $c_b \leftarrow b$
15.         If $d_{c,b} < dBmin[b]$.distance Then $dBmin[b] \leftarrow (c_a, d_{c,b})$
16.         If $d_{c,b} > dBmax[b]$.distance Then $dBmax[b] \leftarrow (c_a, d_{c,b})$
17.     $I_a \leftarrow I_a \setminus \{(c_b, d_{c,c})\}$ // removing center $c_b$ from bucket $I_a$
18.     For each $a \in A$ Do
19.         $d_{a,c} \leftarrow d(a, c_b)$
20.         If $d_{a,c} \leq R$ Then $I_b \leftarrow I_b \cup \{(a, d_{a,c})\}, A \leftarrow A \setminus \{a\}$
21.         Else $dA[a] \leftarrow dA[a] + d_{a,c}$
22.         If $d_{a,c} < dAmin[a]$.distance Then $dAmin[a] \leftarrow (c_b, d_{a,c})$
23.         If $d_{a,c} > dAmax[a]$.distance Then $dAmax[a] \leftarrow (c_b, d_{a,c})$
24.     $CA \leftarrow CA \cup \{(c_a, \max_{b \in I_a}\{d(c_a, b)\}, I_a)\}$ // (center, effective radius, bucket)
25.     $CB \leftarrow CB \cup \{(c_b, \max_{a \in I_b}\{d(a, c_b)\}, I_b)\}$ // (center, effective radius, bucket)
    // we only conserve the $dXmax$ array for non-indexed objects
26. If $|A| > 0$ Then $nonIndexed \leftarrow$ (“A”, $A$, $dAmax$)
27. Else $nonIndexed \leftarrow$ (“B”, $B$, $dBmax$)
28. For each $c_a \in$ centers($CA$), $c_b \in$ centers($CB$) Do $D[c_a, c_b] \leftarrow d(c_a, c_b)$
    // distances $d(c_a, c_b)$ have already been computed, so we can reuse them
29. Return ($CA, CB, D, dAmin, dBmin, nonIndexed$)
```

Figure 6: LTC construction algorithm.

```
rangeJoin (Radius $r$)
1.  For each $c_a^i \in CA, I_b^i \in CB$,  $i \leftarrow 1$ to $|CA|$ Do
2.      rqCenter($i, r$) // solving the center
3.      For each $a \in I_b^i$ Do rqRegular($a, r$) // solving regular objects
4.  ($label, set, array$) $\leftarrow nonIndexed$
5.  If $label$ = “A” Then For each $a \in set$ Do rqNonIndexedA($a, r$)
6.  Else For each $b \in set$ Do rqNonIndexedB($b, r$)
```

Figure 7: Using the LTC for computing range joins.

Before computing the range queries, we need to reduce the search radius. To do so, we populate the priority queue *heap* with all the distances stored in *dAmin* and *dBmin*. Each time we find a pair (a, b) of objects which are closer than the furthest pair in *heap* (that is, lower than *heap*.**max**().distance), we drop the furthest pair and insert the triple $(a, b, d(a, b))$. If it is necessary, that is, when the maximum distance stored in *heap* is greater than the LTC construction radius R, we continue the reduction of the search radius by using the distances in D. Note that all the k candidate pairs stored in *heap* were found for free in terms of distance computations.

After this preprocessing, we start computing the range queries for objects in A. Special care must be taken to avoid repeating pairs with the ones already present in *heap*. A simple alternative is fixing the initial search radius as $auxR \leftarrow heap.$**max**().distance + ε (with any $\varepsilon > 0$) —that is, slightly larger than the furthest pair we currently have in *heap*— and then emptying *heap*. Next, we re-initialize *heap* with k triples (NULL, NULL, $auxR$). Note that this alternative only requires CPU time but no distance computations. In the pseudo-code we use this alternative for readability, however in the actual implementation we use another alternative which is more efficient[4].

We start by solving the range queries for cluster centers using the radius *heap*.**max**().distance. Once again, each time we find a pair of objects (a, b) closer than *heap*.**max**().distance, we modify *heap* by extracting its maximum and then inserting the triple $(a, b, d(a, b))$. Therefore, we are solving a range query with decreasing radius. We continue with the computation for regular objects and finally for non-indexed ones. When the computation finishes, *heap* stores the k pairs of the result. Fig. 8 depicts the algorithm. As cluster centers usually require less work to compute their range queries than regular or non-indexed objects, starting the k-closest pair join computation with them should help to reduce the search radius fast.

Note that, after reviewing all distances stored in the LTC-index, it is possible that the current join radius could be greater than the indexing radius. In this case, we have to process the cluster centers and regular objects using the variants developed for this particular purpose.

4.3. Solving range queries with the LTC

The lists CA and CB can be seen as a classical list of clusters for the datasets B and A, respectively. So, we derive a range query algorithm based on the LTC, which traverse both lists simultaneously. Note that, in this case, we cannot directly use the LC range query stop criterion (that is, when the query ball is fully contained by the current bucket). Instead, we add boolean variables to control whether it is necessary to search the lists.

Also, using distances in matrix D we compute lower and upper bounds on the distances between the query and the centers. To do that, during the computation, we maintain two sets of distances DA and DB which store the distances computed from the query to centers of lists CA and CB, respectively. Therefore, using all the computed distances stored in DB, the distance $d(q, c_a)$ is lower bounded by $\max_{(c_b, d(q,c_b)) \in DB}\{|d(q, c_b) - D[c_a, c_b]|\}$. Likewise, it is upper bounded by $\min_{(c_b, d(q,c_b)) \in DB}\{d(q, c_b) + D[c_a, c_b]\}$. Symmetrically, we upper and lower bound the distance $d(q, c_b)$. So, we modify the LC range query algorithm according to these bounds.

Finally, we need to check the non-indexed objects. For this sake, once again we use the triple *nonIndexed* (*label*, *set*, *array*). So, for the (non-indexed) objects stored in *set* we can compute the lower bounds of the distances from them towards the query by using arrays *dAmin* or *dBmin* according to which dataset they came from (that is, according to *label*). Also, we use *array* in order to improve the lower bound. Recall that, *array* stores either arrays *dAmax* or *dBmax*, depending on *label*. Fig. 9 depicts the algorithm.

5. Experimental evaluation

We have selected four pairs of real-world datasets from three kinds of metric spaces, namely, face images, strings and documents (the two latter are of interest of Information Retrieval applications [5]). The results on these datasets are representative of other metric spaces and datasets we have tested. A detailed description of the datasets follows.

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

The whole set has four face images from 254 people, thus we have divided it into two subsets: one of them with three face images per person (FACES762 for short, because it has 762 face images) and the other with the fourth one (FACES254 for short).

[4]For instance, in order to avoid the constant ε it is enough to replace the "<" sign for "≤" in line 1 of auxiliary procedure **checkMax** (Fig. 8). Although this modification works well with continuous distances, it fails to discard enough values in the discrete case.

```
kClosestPairJoin (Integer $k$)
1.  PriorityQueue $heap \leftarrow \emptyset$ // sorted by increasing distance (third component)
2.  For $i \leftarrow 1$ to $k$ Do $heap$.insert(NULL, NULL, $\infty$)
    // using distances in $dAmin$, $dBmin$, and $D$ to reduce the search radius
3.  For each $a \in A$ Do $(c_b, dist) \leftarrow dAmin[a]$, checkMax($heap, a, c_b, dist$)
4.  For each $b \in B$ Do $(c_a, dist) \leftarrow dBmin[b]$, checkMax($heap, c_a, b, dist$)
5.  If $heap$.max().distance $> R$ Then
6.      For each $c_a^i \in CA, c_b^j \in CB,\ \ i, j \leftarrow 1$ to $|CA|$ Do
7.          checkMax($heap, c_a^i, c_b^j, D[c_a^i, c_b^j]$)
8.  $auxR \leftarrow heap$.max().distance $+ \varepsilon$ // fixing the initial search radius
9.  $heap \leftarrow \emptyset$, For $i \leftarrow 1$ to $k$ Do $heap$.insert(NULL, NULL, $auxR$) // resetting $heap$
10. For each $c_a^i \leftarrow CA,\ \ i \leftarrow 1$ to $|CA|$ Do // reviewing centers
11.     $foundSet \leftarrow$ rqCenter($i, heap$.max().distance)
12.     For each $(b, dist) \in foundSet$ Do checkMax($heap, c_a^i, b, dist$)
13. For each $I_b^i \leftarrow CB,\ \ i \leftarrow 1$ to $|CB|$ Do // reviewing regular objects
14.     For each $a \in I_b^i$ Do
15.         $foundSet \leftarrow$ rqRegular($a, heap$.max().distance)
16.         For each $(b, dist) \in foundSet$ Do checkMax($heap, a, b, dist$)
17. $(label, set, array) \leftarrow nonIndexed$ // reviewing non-indexed objects
18. If $label$ = “A” Then For each $a \in set$ Do
19.     $foundSet \leftarrow$ rqNonIndexedA($a, heap$.max().distance)
20.     For each $(b, dist) \in foundSet$ Do checkMax($heap, a, b, dist$)
21. Else For each $b \in set$ Do
22.     $foundSet \leftarrow$ rqNonIndexedB($b, heap$.max().distance)
23.     For each $(a, dist) \in foundSet$ Do checkMax($heap, a, b, dist$)
```

```
checkMax (PriorityQueue $heap$, Object $a$, Object $b$, Distance $dist$)
1.  If $dist < heap$.max().distance Then
2.      $heap$.extractMax(), $heap$.insert($a, b, dist$) // reducing search radius
```

Figure 8: Using the LTC for computing *k*-closest pair joins.

Strings: a dictionary of words, where the distance is the *edit distance* [29], 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 have considered two pairs of datasets: a subset of 69,069 English words with a subset of 89,061 Spanish words and the same subset of English words with a subset of 494,048 vocabulary terms from a collection of documents.

Documents: a set of 2,957 documents, each of them of approximately 500 KB, obtained by splitting the original 1,265 documents from the TREC-3 collection [24], so that sub-documents obtained from the same original document have an overlap of about 50%. We synthesize the vectors representing the sub-documents by using the program machinery provided in the *Metric Spaces Library* (`http://sisap.org/?f=library`) [22]. As usual, we use the *cosine distance* [45] to compare two documents.

We divide the dataset in two subsets, one of them with 1,846 documents (DOCS1846 for short), and the other with 1,111 documents (DOCS1111 for short).

As we mentioned previously, we work with two particular similarity joins: range joins $A \bowtie_r B$ and k-closest pair joins $A \bowtie_k B$. In the join experiments, we built the index using all the objects considered for each dataset. All our results are averaged over 10 index constructions using different permutations of the datasets. In the range query

```
rangeQuery (Object q, Radius r)
1.  $stop_A \leftarrow$ FALSE , $stop_B \leftarrow$ FALSE
2.  For each $(c_a, R_a, I_a) \in CA, (c_b, R_b, I_b) \in CB,\ \ i \leftarrow 1$ to $|CA|$ Do
3.      If $stop_A$ AND $stop_B$ Then Break
4.      $lbA \leftarrow \max_{(c_b, d(q,c_b)) \in DB}\{|d(q,c_b) - D[c_a,c_b]|\}$ // lower bounding $d(q,c_a)$
5.      $ubA \leftarrow \min_{(c_b, d(q,c_b)) \in DB}\{d(q,c_b) + D[c_a,c_b]\}$ // upper bounding $d(q,c_a)$
6.      $lbB \leftarrow \max_{(c_a, d(q,c_a)) \in DA}\{|d(q,c_a) - D[c_a,c_b]|\}$ // lower bounding $d(q,c_b)$
7.      $ubB \leftarrow \min_{(c_a, d(q,c_a)) \in DA}\{d(q,c_a) + D[c_a,c_b]\}$ // upper bounding $d(q,c_b)$
8.      If $stop_A$ = FALSE AND $lbA \leq R_a + r$ Then
9.          $dqa \leftarrow d(q,c_a), DA \leftarrow DA \cup \{(c_a, dqa)\}, ubA \leftarrow dqa$
10.         If $dqa \leq r$ Then Report $c_a$
11.         If $dqa \leq R_a + r$ Then
12.             For each $b \in I_a$ Do If $d(q,b) \leq r$ Then Report $b$
13.     If $stop_B$ = FALSE AND $lbB \leq R_b + r$ Then
14.         $dqb \leftarrow d(q,c_b), DB \leftarrow DB \cup \{(c_b, dqb)\}, ubB \leftarrow dqb$
15.         If $dqb \leq r$ Then Report $c_b$
16.         If $dqb \leq R_b + r$ Then
17.             For each $a \in I_b$ Do If $d(q,a) \leq r$ Then Report $a$
18.     If $ubA \leq R_a - r$ Then $stop_A \leftarrow$ TRUE
19.     If $ubB \leq R_b - r$ Then $stop_B \leftarrow$ TRUE
20. $(label, set, array) \leftarrow nonIndexed$
21. For each $o \in set$ Do // reviewing non-indexed objects
22.     $lb \leftarrow 0$ // we try to lower bound $d(q,o)$ with the distances stored in $DA$ and $DB$
23.     If $label$ = “A” Then
24.         $(c_b^{min}, d^{min}) \leftarrow dAmin[o], (c_b^{max}, d^{max}) \leftarrow array[o]$
25.         If $(c_b^{min}, dqb) \in DB$ Then $lb \leftarrow \max\{lb, |dqb - d^{min}|\}$
26.         If $(c_b^{max}, dqb) \in DB$ Then $lb \leftarrow \max\{lb, |dqb - d^{max}|\}$
27.     Else
28.         $(c_a^{min}, d^{min}) \leftarrow dBmin[o], (c_a^{max}, d^{max}) \leftarrow array[o]$
29.         If $(c_a^{min}, dqa) \in DA$ Then $lb \leftarrow \max\{lb, |dqa - d^{min}|\}$
30.         If $(c_a^{max}, dqa) \in DA$ Then $lb \leftarrow \max\{lb, |dqa - d^{max}|\}$
31.     If $(lb \leq r)$ AND $d(q,o) \leq r$ Then Report $o$
```

Figure 9: Using the LTC for computing range queries.

experiments, we built the index with 90% of the objects from both datasets, so we use the remain 10% for the range queries, and we report the average computed over all those queries.

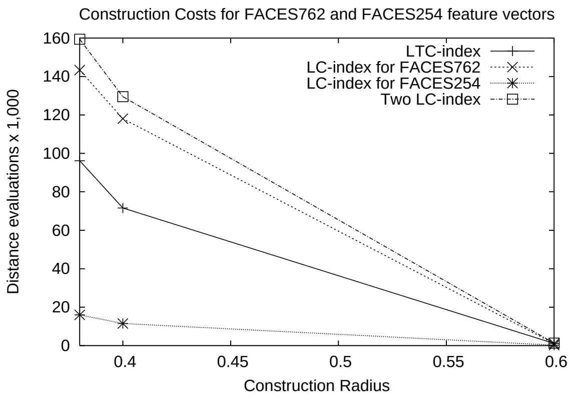
5.1. LTC construction

We start the experimental evaluation by verifying that the cost of constructing the LTC-index for each pair of datasets is similar to the one needed to index the larger dataset with a basic LC. We have tested several values for the construction radius R. For face images, we show construction results when indexing with radii R 0.38, 0.40, 0.60, and 0.80; for strings, radii 3 to 6; and for documents, radii 0.38, 0.40, and 0.60. Fig. 10 shows the results.

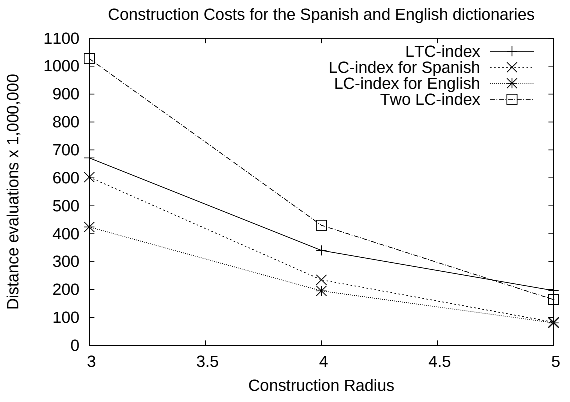
From now on, the join and range query costs do not include the cost of building the LTC and LC indices, as we consider that they would be amortized among many computations of similarity joins and range queries.

5.2. Range joins

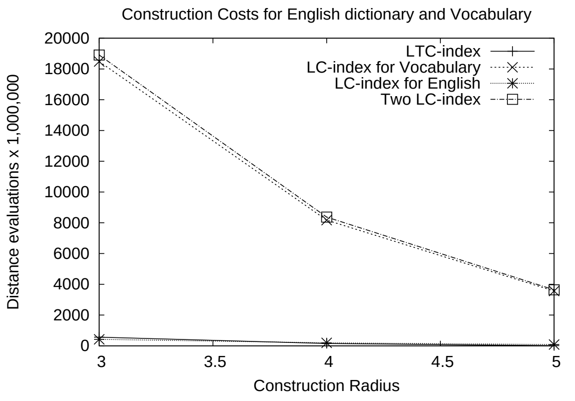
In these experiments we have used the following parameters. For the face images, we have considered thresholds that retrieve on average 1, 5, or 10 relevant images from FACES762 per range query, when queries came from FACES254. This corresponds to radii r equal to 0.2702, 0.3567, and 0.3768, respectively. For the strings, we have



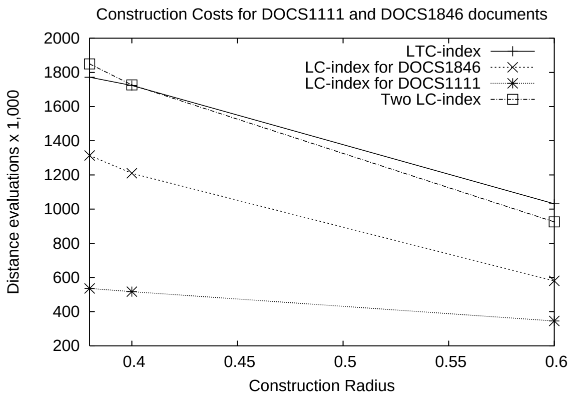
(a) Face images



(b) Spanish and English dictionaries



(c) The vocabulary and the English dictionary



(d) Documents from TREC-3

Figure 10: Constructing the LTC varying the building radius, for the face image datasets (a), Spanish and English dictionaries (b), the English dictionary and the vocabulary (c), and documents (d).

used radii r equal to 1, 2, and 3, as the edit distance is discrete. In the joins between dictionaries this retrieves 0.05, 1.5, and 26 Spanish words per English word on average, respectively. In the joins between the English dictionary and the vocabulary this retrieves 7.9, 137, and 1,593 vocabulary terms per English word on average, respectively. For the documents space, we have used thresholds retrieving on average 2, 3, or 30 relevant documents from DOCS1846 per range query, when we make queries from DOCS1111. So, the values of radii r for the document space are 0.25, 0.265, and 0.38, respectively.

If we have one dataset indexed, 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 dataset. 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 dataset. We have called this join algorithm LC-range join. Furthermore, if we have both datasets indexed, although we could apply the same trivial solution (that is, ignoring one of the indices), we can avoid more distance calculations by using all the information we have from both indices. In order to compare our proposal with an example of this kind of algorithm, we have considered indexing both datasets using a LC and then applying a join algorithm that uses all the information from both indices to improve the join cost. We have named it as LC2-range join, and it is depicted in Fig. 11.

Because we need to fix the construction radius before building the LC and LTC indices, in each case we have considered different radii and we have chosen the one which obtains the best join cost for each alternative. We have tested several cases where the construction radius R is greater than or equal to the largest join radius r used in $A \bowtie_r B$. We have also included a brief test in order to evaluate the performance when the join radius is greater than the indexing one, that is, when $r > R$.

```
rangeJoinLC2 (List $L_1$, List $L_2$, Radius $r$)
1.  For each $(c_i, r_{c_i}, I_{c_i}) \in L_1$ Do
2.    For each $(c_j, r_{c_j}, I_{c_j}) \in L_2$ Do
3.      $d_{cc} \leftarrow d(c_i, c_j)$, $d_s \leftarrow d_{cc} + r_{c_i} + r_{c_j}$
4.      If $d_{cc} \le r$ Then Report $(c_i, c_j)$
5.      If $2\max\{d_{cc}, r_{c_i}, r_{c_j}\} - d_s \le r$ Then // generalized triangle inequality
6.        For each $y \in I_{c_j}$ Do // $d(c_j, y)$ is stored in $L_2$
7.          If $|d_{cc} - d(c_j, y)| \le r$ Then // checking $y$ with the center $c_i$
8.            $d_y \leftarrow d(c_i, y)$, If $d_y \le r$ Then Report $(c_i, y)$
9.          For each $x \in I_{c_i}$ Do // $d(c_i, x)$ is stored in $L_1$, checking pairs $(x, y)$
10.           $d_s \leftarrow d_{cc} + d(c_i, x) + d(c_j, y)$
11.           $lb \leftarrow 2\max\{d_{cc}, d(c_i, x), d(c_j, y)\} - d_s$
12.           If $d_y$ were calculated Then $lb \leftarrow \max\{lb, |d_y - d(c_i, x)|\}$
13.           If $lb \le r$ AND $d(x, y) \le r$ Then Report $(x, y)$
14.       For each $x \in I_{c_i}$ Do // we check all $x \in I_{c_i}$ with the center $c_j$
15.         If $|d_{cc} - d(c_i, x)| \le r$ AND $d(x, c_j) \le r$ Then Report $(x, c_j)$
16.     If $d_{cc} + r_{c_i} + r \le r_{c_j}$ Then
17.       Break // stop searching $(c_i, r_{c_i}, I_{c_i})$ on $L_2$
```

Figure 11: LC2-range join for two Lists of Clusters.

Fig. 12 illustrates the performance of the LTC-range join considering the different radii in all the pairs of datasets. We have shown the number of object pairs retrieved.

As it can be noticed, the best result is obtained when the building radius R is the closest to the greatest value of r considered in each case. The LC2-range join has a similar behavior, but in the case of LC-range join, the best radius can vary a little; in fact, for the range join between both dictionaries, it is 4, and for documents, it is 0.60.

Fig. 13 depicts a comparison among the three range join algorithms (without construction costs) for the four pairs of datasets, using the best value of the building radius R experimentally determined for each range join algorithm. Once again, we have shown the number of object pairs retrieved. We can observe that the LTC-range join algorithm largely outperforms the other range join algorithms considered in three of the pairs of datasets used. For the range join between the English dictionary and the vocabulary, LC-range join and LC2-range join beat us, despite the LTC-range join's significant improvement over the Nested Loop join in all thresholds used.

We suspect that this non-intuitive behavior showing that the simplest algorithm, LC-range join, outperforms our LTC-range join between the vocabulary and the English dictionary can be explained by taking into account the number of non-indexed objects. In this case 39% of the vocabulary terms are not indexed, while in the others, where the LTC-range join is the best method, the percentage of non-indexed objects is lower. For instance, in the experiment of face images, only 2% of the faces are not indexed; in the experiment of Spanish and English dictionaries non-indexed words represent 23% of the dataset, and for documents the percentage of non-indexed documents is 20%.

Also, we have split the join cost in three parts (for centers, regular objects and non-indexed objects) in order to illustrate this non-intuitive result. The values are shown in Table 1. As can be seen, in a favorable case, like face images, most of the work is performed among regular objects. Instead, in the join between the vocabulary and the English dictionary, most of the work is performed when solving non-Indexed objects.

Fig. 14 depicts a brief comparison among the three range join algorithms (without construction costs) when the join radius is greater than the indexing one, that is, when $r > R$. In the plots we show results for the face images and the document datasets. Once again, we have shown the number of object pairs retrieved. As it is expected, we observe a performance degradation in our LTC-based range join (which is also seen both in LC-join and LC2-join), yet it remains as the best range join alternative.

Finally, Table 2 gives the performance ratios of distance computations for the four pairs of datasets. The values are computed according to this formula: $\frac{\text{join}-\text{LTC-range join}}{\text{join}} \cdot 100\%$.

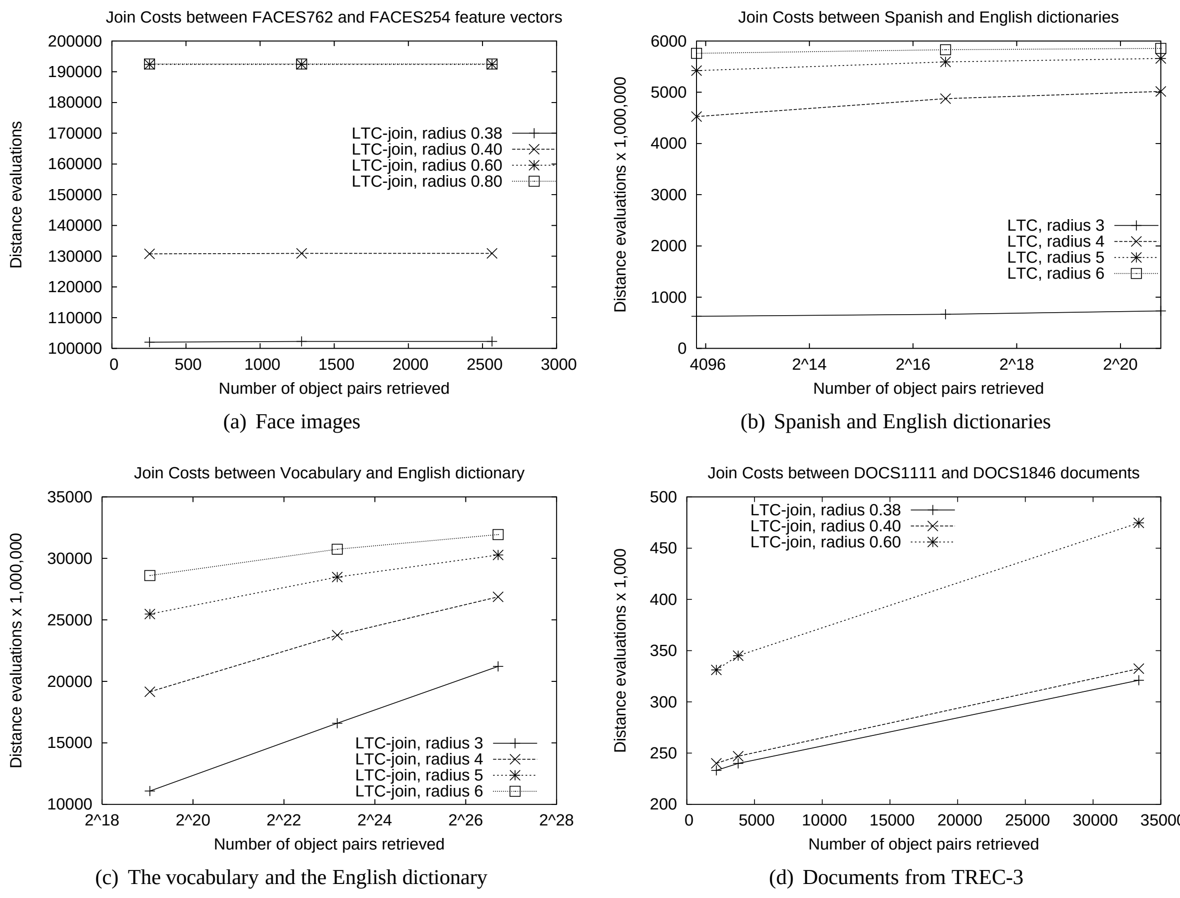


Figure 12: Comparison among the different radii considered for the LTC index construction, for the face image datasets (a), Spanish and English dictionaries (b), the English dictionary and the vocabulary (c), and documents (d). Note the logscales.

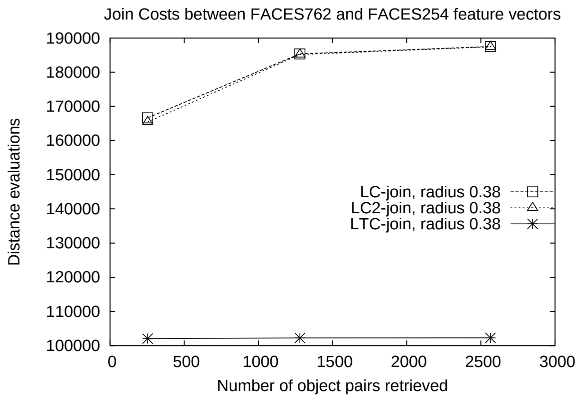
5.3. *k*-Closest pair join

In this case, we can only compare the performance of the LTC-based *k*-closest pair join $A \bowtie_k B$ with the LTC-range join, as we do not have any other alternative for metric spaces. (As far as we know, there is no previous attempt to solve this variant.) Fig. 15 shows the results when retrieving the 10, 100 and 1,000 closest pairs for the four pair of datasets. As it can be seen, the performance of the $A \bowtie_k B$ is similar to the one of the equivalent range join. This reveals that the strategy used to reduce the search radius operates effectively. It is interesting to note that the *k*-closest pair join performance in the string space is slightly better than the range join one. This is because the edit distance is discrete and there are thousands of word pairs at distance 1. So, the heap of *k* candidate pairs is promptly filled with pairs at distance 1 and subsequent range queries use radius 0.

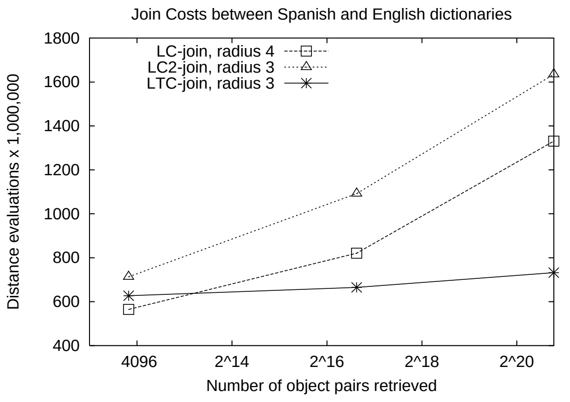
5.4. Range queries

We have computed range queries using the same join radii of Section 5.2. That is, in the face image space we have used radii r equal to 0.2702, 0.3567, and 0.3768, retrieving 0.7, 4.5 and 9.1 images, respectively. For the strings we have used radii r equal to 1, 2, and 3, recovering 2, 25, and 229 words from both English and Spanish dictionaries; and 7, 161, and 2,025 words from the English dictionary and the vocabulary, respectively. For the documents space, we have used radii 0.25, 0.265, and 0.38, retrieving 3, 5, and 47 documents, respectively. These results were averaged over the whole subsets of queries (that is, 10% of the union of both datasets).

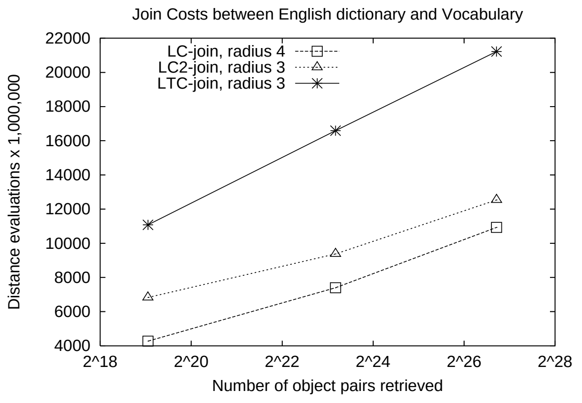
The plots of Fig. 16 show a comparison of the LTC-based range query algorithm with respect to (i) index the union of both datasets with a single LC, and (ii) index each dataset with a LC. Alternative (i) implies adding a new index



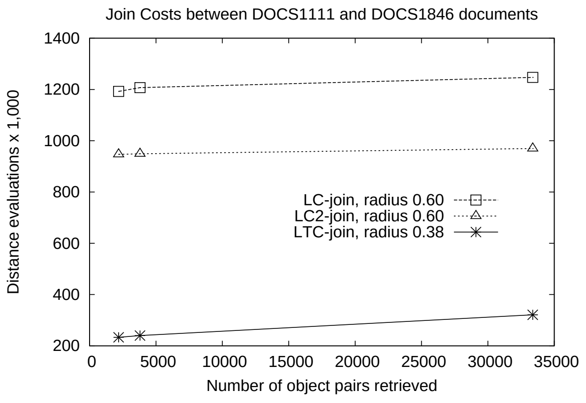
(a) Face images



(b) Spanish and English dictionaries

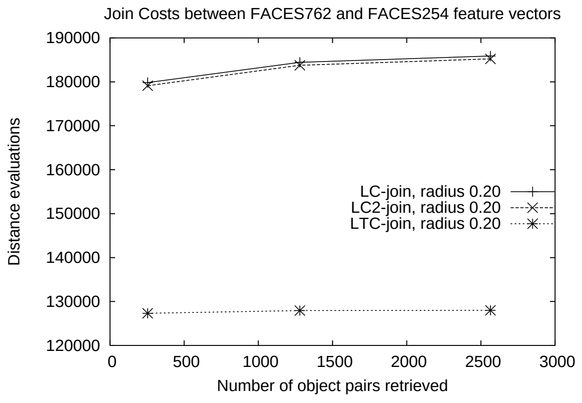


(c) The vocabulary and the English dictionary

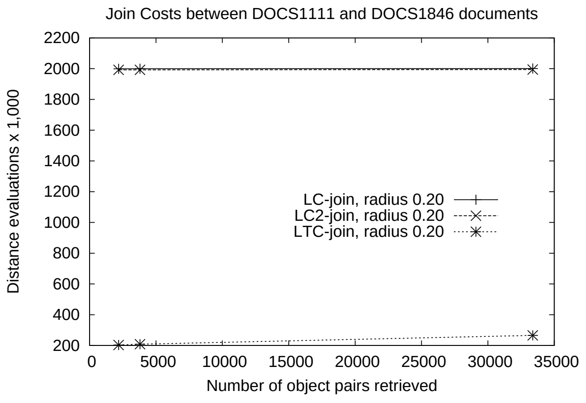


(d) Documents from TREC-3

Figure 13: Comparison among all the range join algorithms considered, using in each case the best value experimentally determined for the building radius of the index. For face image databases (a), Spanish and English dictionaries (b), the English dictionary and the vocabulary (c), and documents (d). Note the logscales.



(a) Face images



(b) Documents from TREC-3

Figure 14: Comparison among all the range join algorithms considered using a join radius greater than the indexing radius, that is, $R < r$. For face image databases (a), and documents (b).

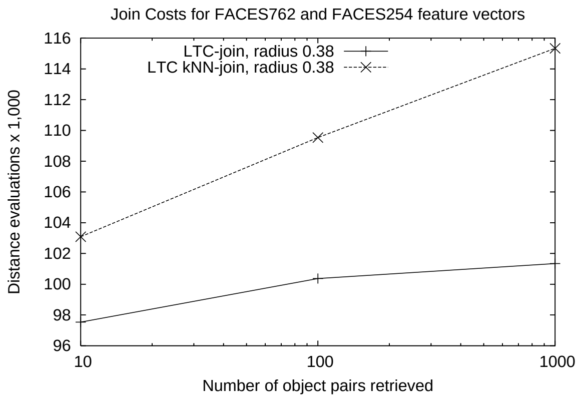
Table 1: Fraction of the total join cost performed by centers, regular objects and non-indexed objects.

(a) Join between face image datasets.

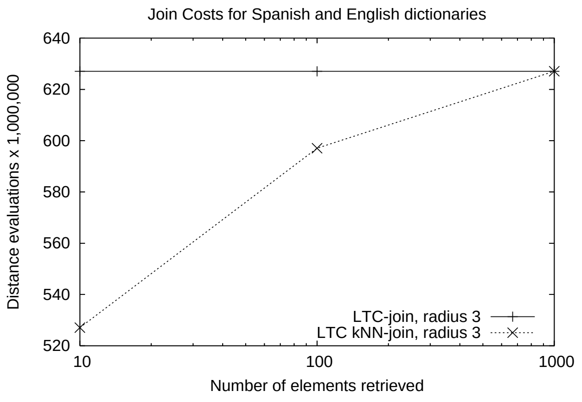
	$r = 0.2702$	$r = 0.3567$	$r = 0.3768$
centers	1.8%	1.8%	1.8%
regular objects	83.9%	84.0%	84.0%
non-Indexed objects	14.3%	14.2%	14.2%

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

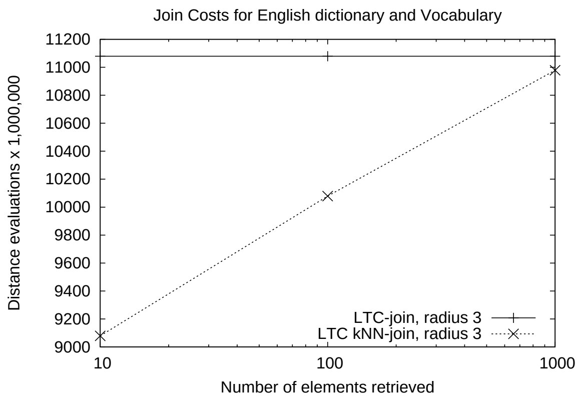
	$r = 1$	$r = 2$	$r = 3$
centers	0.3%	0.7%	1.4%
regular objects	27.8%	29.6%	30.8%
non-Indexed objects	71.9%	69.7%	67.8%



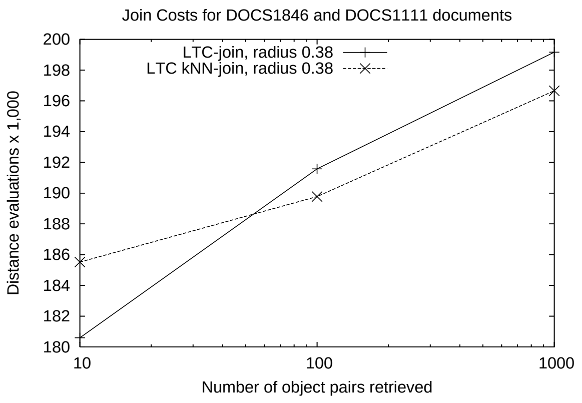
(a) Face images



(b) Spanish and English dictionaries



(c) The vocabulary and the English dictionary



(d) Documents from TREC-3

Figure 15: Comparison between the *k*-closest pair join and the equivalent range join. For face image databases (a) Spanish and English dictionaries (b), the English dictionary and the vocabulary (c), and documents (d). Note the logscales.

in order to support range queries, while alternative (ii) is equivalent to our approach, in the sense that it reuses the indices in order to cope with similarity joins and the classic similarity primitive.

In the comparison, our LTC-based range query algorithm shows good performance when compared with the basic LC approach. This can be explained when we consider that we store more information in the LTC-index than in

Table 2: Performance ratio of the LTC-range 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-range join	LC2-range join	Nested Loop
0.2702	38%	38%	47%
0.3567	44%	44%	47%
0.3768	45%	45%	47%

(b) Join between Spanish and English dictionaries.

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

(d) Join between the DOCS1846 and DOCS1111.

Threshold	LC-range join	LC2-range join	Nested Loop
0.25	80%	75%	88%
0.265	80%	74%	88%
0.38	74%	67%	84%

the basic LC. In fact, the matrix of distances between centers allows us to effectively reduce the number of distance computation performed in three cases. With respect to the contestants, it can be seen that it is systematically better to have a single LC indexing the union of the datasets than two LC indexing each dataset independently. Finally, only in the face image spaces (see Fig. 16(a)) using a single LC is slightly faster than our LTC-based range queries.

6. Conclusions

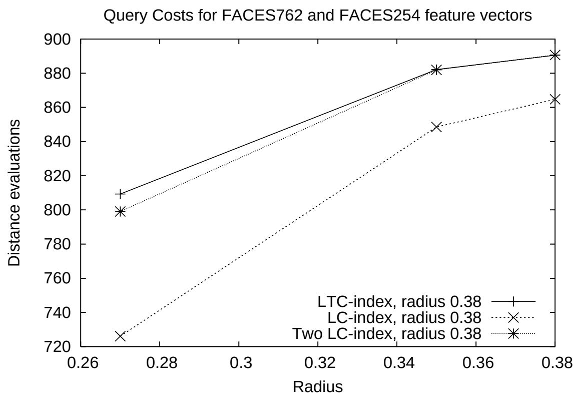
In this work we have shown a new approach for computing similarity joins between two datasets which consists in indexing both datasets jointly. For this sake, we have proposed a new metric index, coined *list of twin clusters* (LTC). We have experimentally verified that the cost of constructing the LTC index is similar to that of constructing a single LC in order to index the larger dataset.

Based on the LTC index we have solved two kinds of similarity joins: (1) *range joins* $A \bowtie_r B$: Given distance threshold r, find all the object pairs (one from each set) at distance at most r; and (2) *k-closest pair joins* $A \bowtie_k B$: Find the k closest object pairs (one from each set). The results of the experimental evaluation of the range join not only show significant speedups over the basic quadratic-time naive alternative but also over other two range join algorithms, LC-range join and LC2-range join, for three of the pairs of datasets considered.

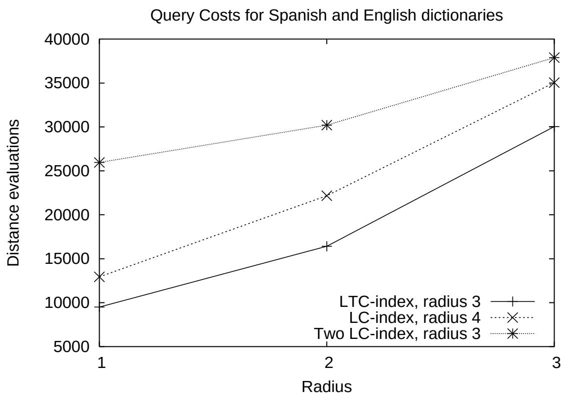
With respect to the k-closest pair join, the results of the experimental evaluation show that it is rather efficient, as it requires a work similar to the one performed by the equivalent range join over the LTC index. This resembles the performance of range-optimal k-nearest neighbor search algorithms [25].

Finally, we have shown that the LTC-based range query algorithm is competitive with, and in some cases better than, the LC search algorithm.

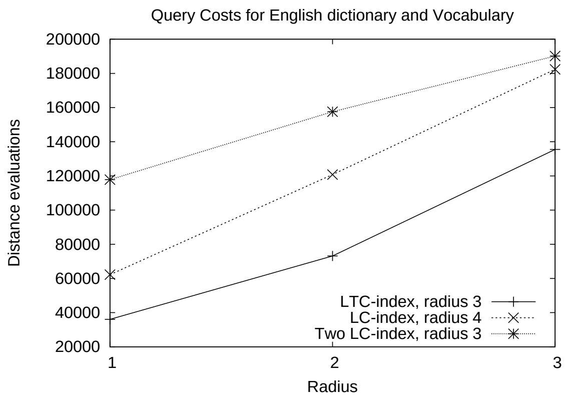
Our new LTC index stands out as a practical and efficient data structure to solve two particular cases of similarity joins, such as $A \bowtie_r B$ and $A \bowtie_k B$, and as an index to speed up classical range queries. The LTC can be used for pairs of databases in any metric space and therefore it has a wide range of applications.



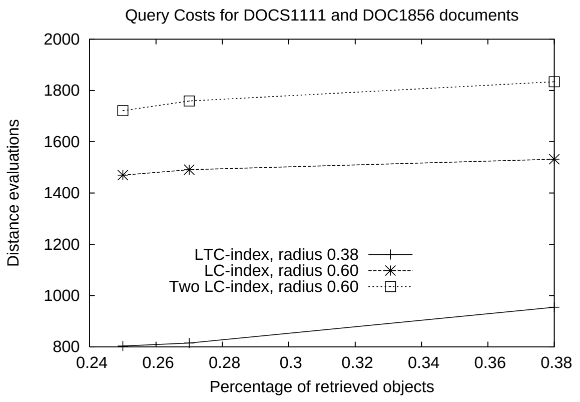
(a) Face images



(b) Spanish and English dictionaries



(c) The vocabulary and the English dictionary



(d) Documents from TREC-3

Figure 16: Computing range queries over the LTC varying radius, for the face image datasets (a), Spanish and English dictionaries (b), the English dictionary and the vocabulary (c), and documents (d).

Several lines of future work on similarity joins indices and algorithms remain open:

- The similarity self join: although in this case there is no reason to build a LTC index, we plan to create another variant of LC specially designed for this kind of join.
- Optimization of LTC by evaluating internal distances: at construction time of the LTC index and when we evaluate 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 by choosing better centers we will be able to significantly reduce the memory needed for the matrix of distances among centers.
- Different kinds of joins: we are developing algorithms to solve other kinds of similarity joins over the LTC index or its variants. For instance, computing the *k*-nearest neighbors for each object in one dataset and retrieving relevant objects from the other.
- There are cases where we could be interested in computing range queries on either dataset *A* or *B* but not both, so we also plan to develop strategies to solve this kind of range queries.

- When using clusters of fixed radius, we experimentally observe that the first clusters are much more populated than the following ones. Moreover, we can also include the study of dynamic LTCs. Therefore, we have also considered developing a version of the LTC similar to Mamedes's *recursive list of clusters* [33].
- Since in some cases many non-indexed objects exist, and this harms the performance of the LTC-range join, we have also considered researching on alternatives to manage the non-indexed objects.
- Developing parallel variants for the LTC index is another interesting line of research, aiming at reducing the computation time (there are already some parallel versions for other metric indices [16, 34, 35]).

Acknowledgments

We wish to thank the helpful and constructive comments from the anonymous referees, and Mauricio Marin, who asked us how to solve range queries using the LTC index.

References

[1] F. Angiulli and C. Pizzuti. Approximate *k*-closest-pairs with space filling curves. In *Proc. 4th Intl. Conf. on Data Warehousing and Knowledge Discovery (DaWaK'02)*, pages 124–134, London, UK, 2002. Springer-Verlag.

[2] F. Angiulli and C. Pizzuti. An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data & Knowledge Engineering (DKE)*, 53(3):263–281, 2005.

[3] R. A. Baeza-Yates. Searching: An algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, New York, 1997.

[4] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212. Springer-Verlag, 1994.

[5] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[6] 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.

[7] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. 1997 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'97)*, pages 357–368. ACM, 1997.

[8] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conf. on Very Large Databases (VLDB'95)*, pages 574–584. Morgan Kaufmann, 1995.

[9] 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.

[10] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM (CACM)*, 16(4):230–236, 1973.

[11] E. Chávez, J. L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

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

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

[14] T. Chiueh. Content-based image indexing. In *Proc. 20th Conf. on Very Large Databases (VLDB'94)*, pages 582–593, 1994.

[15] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd Conf. on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[16] G.V. Costa and M. Marin. Distributed sparse spatial selection indexes. In *Proc. 16th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing (PDP'08)*, pages 440–444. IEEE Computer Society Press, 2008.

[17] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Information Systems (IS)*, 12(2):171–175, 1987.

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

[19] 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.

[20] 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.

[21] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the least cost for proximity searching in metric spaces. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA'06)*, LNCS 4007, pages 279–290. Springer, 2006.

[22] K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[23] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'84)*, pages 47–57, 1984.

[24] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conf. (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[25] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Dept. of Comp. Sci. Univ. of Maryland, Nov 2000.

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

[27] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. on Software Engineering (TSE)*, 9(5), 1983.

[28] 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.

[29] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[30] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. 1996 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pages 247–258. ACM Press, 1996.

[31] M.-L. Lo and C. V. Ravishankar. The design and implementation of seeded trees: An efficient method for spatial joins. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):136–152, 1998.

[32] M. Lopez and S. Liao. Finding k-closest-pairs efficiently for high dimensional data. In *Proc. 12th Canadian Conf. on Computational Geometry (CCCG'00)*, pages 197–204, 2000.

[33] 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.

[34] M. Marin, G.V. Costa, and C. Bonacic. A search engine index for multimedia content. In *Proc. 14th European Conf. on Parallel and Distributed Computing (EuroPar'08)*, LNCS 5168, pages 866–875, 2008.

[35] M. Marin and N. Reyes. Efficient parallelization of spatial approximation trees. In *Proc. 5th Intl. Conf. on Computational Science (ICCS'05)*, LNCS 3514, pages 1003–1010, 2005.

[36] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters (PRL)*, 15:9–17, 1994.

[37] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[38] G. Navarro, R. Paredes, and E. Chávez. *t*-spanners for metric space searching. *Data & Knowledge Engineering (DKE)*, 63(3):818–852, 2007.

[39] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[40] G. Navarro and N. Reyes. Dynamic spatial approximation trees. *ACM Journal of Experimental Algorithmics (JEA)*, 12: article 1.5, 68 pages, 2008.

[41] R. Paredes and E. Chávez. Using the *k*-nearest neighbor graph for proximity searching in metric spaces. In *Proc. 12th Intl. Symp. on String Processing and Information Retrieval (SPIRE'05)*, LNCS 3772, pages 127–138. Springer, 2005.

[42] 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.

[43] R. Paredes and N. Reyes. List of twin clusters: a data structure for similarity joins in metric spaces. In *Proc. 1st Intl. Workshop on Similarity Search and Applications (SISAP'08)*, pages 131–138. IEEE Computer Society Press, 2008.

[44] O. Pedreira and N. R. Brisaboa. Spatial selection of sparse pivots for similarity search in metric spaces. In *33rd Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, LNCS 4362, pages 434–445. Springer, 2007.

[45] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Comm. of the ACM (CACM)*, 18(11):613–620, 1975.

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

[47] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters (IPL)*, 40(4):175–179, 1991.

[48] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters (PRL)*, 4:145–157, 1986.

[49] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA'93)*, pages 311–321. SIAM Press, 1993.

[50] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, 1998. In *6th DIMACS Implementation Challenge: Near Neighbor Searches Workshop, ALENEX'99*.

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