List of Clustered Permutations in Secondary Memory for Proximity Searching [*]

Patricia B. Roggero, Nora S. Reyes
Departamento de Informática, Universidad Nacional de San Luis
San Luis, Argentina
and
Karina M. Figueroa
Facultad de Ciencias Físico-Matemáticas, Universidad Michoacana
Morelia, México
and
Rodrigo A. Paredes
Departamento de Ciencias de la Computación, Universidad de Talca
Curicó, Chile

ABSTRACT

Similarity search is a difficult problem and various indexing schemas have been defined to process similarity queries efficiently in many applications, including multimedia databases and other repositories handling complex objects. Metric indices support efficient similarity searches, but most of them are designed for main memory. Thus, they can handle only small datasets, suffering serious performance degradations when the objects reside on disk. Most real-life database applications require indices able to work on secondary memory.

Among a plethora of indices, the List of Clustered Permutations (LCP) has shown to be competitive in main memory.We introduce a secondary-memory variant of the LCP, which maintains the low number of distance evaluations when comparing the permutations themselves, and also needs a low number of I/O operations at construction and searching.

Keywords: metric spaces, permutation-based algorithm, list of clusters, secondary memory

1. INTRODUCTION

"Proximity" or "similarity" searching is the problem of looking for objects in a dataset, that are "close" or "similar enough" to a given query object, under a certain (expensive to compute) distance. Similarity search has become a very important operation in applications that deal with unstructured data sources. For example, multimedia databases that manage objects without any kind of structure, such as images, fingerprints or audio clips. This approximation has applications in a vast number of fields. These problems can be mapped into a *metric space model* [3]. That is, there is a universe $\mathbb{X}$ of objects, and a non negative real valued distance function $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+ \cup \{0\}$ defined among them. This distance satisfies the three axioms that make the set a *metric space*: *strict positiveness* ($d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($d(x, y) = d(y, x)$), and *triangle inequality* ($d(x, z) \leq d(x, y) + d(y, z)$).

The smaller the distance between two objects, the more "similar" they are. We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, $|\mathbb{U}| = n$, which is a subset of the universe and can be preprocessed to build an index. Later, given a new object from the universe (a *query* $q \in \mathbb{X}$), we must retrieve all similar elements found in the database. There are two typical similarity queries:

- *Range query* (q, r)*:* retrieve all elements within distance r to q in $\mathbb{U}$.
- k*-Nearest neighbor query (*k*-NN):* retrieve the k closest elements to q in $\mathbb{U}$.

Our focus is on *approximate proximity searching*, where accuracy can be traded off for efficiency, as

[*] Partially funded by Fondecyt grant 1131044, Chile.

opposed to exact similarity search algorithms. However, there are generic techniques to convert any exact algorithm into approximate by using a form of aggressive pruning, as described for example, in [6].

For general metric spaces, there exist a number of methods to preprocess the database in order to reduce the number of distance evaluations [3, 14, 13]. In general metric spaces, the (black-box) distance function is the only way to distinguish between objects, and usually, the function of distance is expensive to calculate (in time and/or resources), compared to the CPU time to traverse the index and decide which elements are relevant. However, when the index is located in secondary memory the I/O operations are also very significant [12]. Therefore, the goal of similarity search algorithms for metric spaces in secondary memory is to solve queries minimizing the number of distances evaluations and I/O operations.

Since this kind of datasets lacks of total order to avoid a full linear scan, it is necessary to preprocess the database to build an index which allows answering queries with less effort. The *List of Clusters* (LC) [2] is one of the most efficient algorithms in high dimensional spaces (difficult spaces), however it takes $O(n^2)$ distance calculations to build the index. In other hand, the *Permutation Based Algorithm* (PBA) [4, 5, 9] is an approximate method that has been showed unbeatable in practice, but it only works well in high dimensions, as the authors claim. Once the index is built by calculating the "permutation" of each database object, during searches we have to calculate the permutation of the query object q and compare it with all permutations of database objects, to compute the order to review permutations. This takes at least $O(|\mathbb{P}|)$ distance calculations, where $|\mathbb{P}|$ is the permutation size, and $O(n)$ evaluations of the "permutation distance". There have been several proposals to avoid the sequential scan in PBA, however all of them lost accuracy regarding the original technique [7, 10]. In [9], a combination of the main ideas of LC and PBA is presented, designing a new metric index to answer approximate similarity search. This new index, called as *List of Clustered Permutations* (LCP), achieves a good search performance and beats both LC and PBA.

However, when we want to answer approximate similarity queries on large volumes of data, working in secondary memory and considering distance and I/O costs is necessary. The I/O time is composed of the number of disk pages read and written; we call B the size of the disk page in bytes. Given a dataset of $|\mathbb{U}| = n$ objects of total size N bytes and disk page size B, queries can be trivially answered by performing n distance evaluations and N/B I/Os. The goal of a secondary-memory index is to preprocess the dataset so as to answer queries with as few distance evaluations and I/Os as possible.

Therefore, in this article we use the idea of LCP [9], built on LC and PBA, but considering the index have to be located in secondary memory. So, the idea is to keep each cluster of the list on a disk page in secondary memory. Hence, the cluster size must consider the disk page size. Besides, in order to accelerate searches the information of centers (permutants) and some few more data are also maintained in main memory.

The rest of this paper is organized as follows. In Section 2 we describe the previous works and some basic concepts. Next, in Section 3 we detail the *List of Clustered Permutations* (LCP) and in Section 4 we present our secondary-memory variant of LCP. In Section 5 we show the experimental evaluation of our proposal. Finally, we draw some conclusions and future work directions in Section 6.

2. PREVIOUS WORKS

In order to introduce our secondary-memory index, we describe briefly the main aspects of the previous works used as basis.

2.1. Permutation-Based Algorithm

In [4, 5] the authors introduce the permutation based algorithm (PBA), a novel technique that shows a different way to sort the space. At preprocessing time, a subset of objects $\mathbb{P} = \{p_1, p_2, .., p_{\|\mathbb{P}\|}\} \subseteq \mathbb{U}$ is selected out of the database, which are called the *permutants*. Each object $u \in \mathbb{U}$, computes its distance to all the permutants (i.e., computes $d(u,p)$ for all $p \in \mathbb{P}$) and sorts them increasingly by distance. Then, for each object $u \in \mathbb{U}$, just the order of the permutants (not the distances) is stored in the index.

If we define Π_u as the permutation of $(1, \ldots, \mathbb{P})$ for the object u, so $\Pi_u(i)$ is the i-th cell in the u's permutation and $p_{\Pi_u(i)}$ denotes the i-th permutant. For example, if $\Pi_u(i) = (5, 2, 1, 3, 4)$ then $p_{\Pi_u(2)} = p_2$. Within the permutation, for all $1 \leq i < \|\mathbb{P}\|$ it holds either $d(p_{\Pi_u(i)}, u) < d(p_{\Pi_u(i+1)}, u)$ or, if there is a tie $(d(p_{\Pi_u(i)}, u) = d(p_{\Pi_u(i+1)}, u))$, then the permutant with the lowest index appears first in

Π_u. We call the i-th permutant $\Pi_u(i)$, the *inverse permutation* Π_u^{-1}, and the position of i-th permutant $\Pi_u^{-1}(p_i)$. The set of all the permutations stored in the index needs just $O(n|\mathbb{P}|)$ memory cells.

During searches, we compute the distance from the query $q \in \mathbb{X}$ to all the permutants in $\mathbb{P}$ and obtain the query permutation Π_q. Next, Π_q is compared with all the permutations stored in the index, which takes $O(n)$ permutation distances.

Authors in [4] claim that the order induced by the permutation of the query object q (Π_q) is extremely promising and reviewing a small fraction of the dataset is enough to get a good answer.

The *permutation distance* is calculated as follows: let Π_u and Π_q permutations of $(1, \ldots, |\mathbb{P}|)$. We compute how different is a permutation from the other one using *Spearman Rho* (S_ρ) metric. In [8] the S_ρ distance is defined as:

$$S_\rho(\Pi_u, \Pi_q) = \sqrt{\sum_{1 \leq i \leq \mathbb{P}} (\Pi_u^{-1}(i) - \Pi_q^{-1}(i))^2}$$

2.2. List of Clusters

There are many indices for metric spaces [3, 14, 13, 1]. One of the most economical in space used and rather efficient is the *List of Clusters* (LC) [2], because it needs $O(n)$ space and has an excellent search performance in high dimension. Regretably, its construction requires $O(n^2)$ distance evaluations, which is very expensive. The LC index is built recursively. The LC has two variants, one with a fixed cluster size and the other with a fixed cluster radius. We describe here the variant of fixed cluster size that sets the maximum number of elements that fits in a disk page included into a cluster, because the LCP is based on it [9], has better search performance [2], and it is more convenient to secondary memory.

Firstly, a center c is selected from the database and a bucket size b is given. c chooses its b-closest elements of the database and build the subset I, which is the answer of a b-nearest neighbor query of c in $\mathbb{U}$. Let cr_c be the distance from c to its farthest neighbor in I. The tuple (c, I, cr_c) is called a *cluster*. This process is recursively repeated with the rest of the non-clustered objects; in this case with $\mathbb{U} \setminus (I \cup \{c\})$. Finally, we have a set of centers $\mathbb{C}$ with their cluster elements and their covering radii, organized as a list.

To answer queries, the query object q is compared with all the cluster centers $\mathbb{C}$. During a range search (q, r), for each cluster with center c_i, if the distance from its c_i to the query q is larger than its covering radius cr_{c_i} plus the query radius r we can discard its whole bucket, otherwise we review it exhaustively. Formally, if $d(q, c_i) > cr_{c_i} + r$ the cluster of c_i can be completely discarded.

3. LIST OF CLUSTERED PERMUTATIONS

As it is aforementioned, LC is a good search index but is costly to build and PBA gives a way to answer approximate similarity queries, while trades accuracy or determinism for faster searches. The simplest way to reduce the construction time of LC is to avoid distance computations. For this sake, we have two possibilities: a bigger bucket size, or using another, cheaper, way to build the index. Follow the second possibility, in [9], we propose to combine the PBA with the LC. We choose a set of permutants, where each one within this set has a double role, as permutant and as a cluster center. Besides, only the cluster centers store their permutation. This index is called *List of Clustered Permutations* (LCP) [9].

As we mention previously, when we solve a similarity query q with the standard PBA, we need to spend $|\mathbb{P}|$ evaluations of the distance d to compute the query permutation Π_q, n evaluations of the permutation distance S_ρ to compute the order induced by Π_q on $\mathbb{U}$, and $O(fn)$ distance evaluations (of d) to compare q with the fraction f of the dataset objects that are the most promising to be relevant for the query. With the LCP index, only $|\mathbb{P}|$ ($\ll n$) evaluations of the permutation distance S_ρ are needed to compare Π_q with the permutation of each cluster center. Then, some distances evaluations are needed to review non-discarded clusters. In [9] the authors shown experiments that verify the improvement of LCP over the traditional LC.

The building process of the index is done as follows: a set $\mathbb{P} = \{c_1, \ldots, c_{|\mathbb{P}|}\}$ of centers (permutants) is randomly selected, and for each database object $u \in \mathbb{U}$, $d(u, c_i), \forall c_i \in \mathbb{P}$ is calculated. Hence, we can compute the permutations for all the objects u in the dataset $\mathbb{U}$. Then, the first center is chosen and grouped its $b = \frac{n}{|\mathbb{P}|} - 1$ most similar objects according to the permutation distance S_ρ (excluding all the cluster centers, so that no center can be inside the bucket of another center). The process continues it-

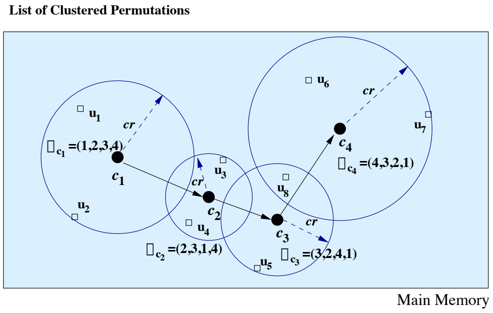

Figure 1: An example of LCP.

eratively with the rest of elements in $\mathbb{P}$ until every element in $\mathbb{U} \setminus \mathbb{P}$ is clustered. Every center c_i maintains its covering radius cr_{c_i} (that is, the distance to the farthest object in its bucket), its bucket (its subset of elements) and its permutation. All the permutations of elements in $\mathbb{U} \setminus \mathbb{P}$ are discarded; that is, the permutations of all the objects within a bucket will not be stored.

Figure 1 shows an example of LCP index for a little set of points in $\mathbb{R}^2$, where the set $\mathbb{P} = \{c_1, c_2, c_3, c_4\}$ of centers (permutants) is selected and $b = 2$ (only two points belong to each cluster). We also show the covering radii and the permutations of centers.

Therefore, the space used for the index is $n + \mathbb{P}^2$ cells, and the construction time is $O(n|\mathbb{P}|)$ evaluations of both the space distance d and the permutation distance S_ρ. It can be noticed that the whole LCP index can be packed using only $(n + \mathbb{P}^2)log_2|\mathbb{P}|$ bits.

As it is mentioned, the standard LC discards clusters during a range search (q, r) by using the covering radii criterion. Let $d(q, c)$ be the distance between the query q and the center of the cluster c and cr_c the covering radius of center c. So, if $d(q, c) > r + cr_c$ the cluster whose center is c can be discarded.

Since the centers of LCP have permutations, a heuristic method can be introduced to discard clusters, modifying the criteria explained in [2]. In [9] authors mention that their preliminary experimental results have shown that if an object (for instance, a cluster center), and its permutation have (just) one permutant that moved far away with respect to its position inside query permutation, then this object is not relevant, so it can be discarded (and also its bucket). For example, if the permutation of the query is $(1, 2, 3, 4)$ and the permutation of the center is $(4, 1, 2, 3)$, even though most of both permutations are similar, the position shifting of permutant 4 suggests that the object can be discarded.

Basically, it is necessary to know how much could a permutant move away inside the permutation of an object. So, by using the query permutation Π_q and the range query radius r, it can be estimated how far a permutant could shift. To do that, for a pair of permutants c_i, c_j , where c_i is closer to the query q than c_j, and $d(c_j, q) - d(c_i, q) \leq r$, the method does not discard an object whose permutation has an inversion of these permutants; this is, it does not discard an object that is closer to c_j than to c_i. But, if the distance difference is larger, although permutant inversion is possible there as a big chance that the object were irrelevant, so the object can be discarded. During query process, a cluster center (and its bucket) is discarded when a permutant shifts more than tolerated.

4. LCP IN SECONDARY MEMORY

In order to obtain an efficient variant of LCP for secondary memory, we have to consider some important aspects of using a disk as storage. An I/O operation on disk involves three main times: the time of head positioning, lattency, and transfer time. The transfer unit of a disk is called a *disk page*. Therefore, a way of reducing times is to read/write few disk pages. One key aspect for this objective is to use as few disk pages as possible, and other is to read/write disk pages when it is strictly necessary.

Therefore, our proposal consider the design of a secondary-memory variant of LCP [9] that occupies the smallest possible amount of disk pages and it only reads/writes a disk page when it is actually necessary. Hence, we take advantage of main memory to store some information of the index and we set the size of the clusters as how many database elements fits in a disk page of size B. We called our variant as *LCP**.

The build process of LCP* is almost the same used for LCP. As we mentioned, we set the cluster size as a function of disk page size and the size of the representation of an object. Therefore, at this point LCP* is different from LCP, because b is a fixed value defined mainly by B and not as a function of the number of centers selected. On the other hand, the number of centers needed is determined as a function of the resulting size b of a cluster; that is, $|\mathbb{P}| = \frac{n}{b+1}$. Thereby, we force that each cluster fits completely in a disk page and, because of that, when we need to review the elements of a cluster we only have to read only one disk page. In each cluster only the real objects are stored. As LCP does, all the permutations of elements in $\mathbb{U} \setminus \mathbb{P}$ are discarded. Furthermore, taking advantage of main memory storage, we replicate some information of LCP* in main memory, in order to avoid reading unnecessarily a page (cluster) only to compare the query object q with the center c of a cluster and then determine its cluster is non relevant. Hence, we maintain in main memory the list of selected centers $\mathbb{P}$. For each center $c \in \mathbb{P}$ we store its covering radius, its permutation, the actual number of elements in its cluster, and the number of disk page where is stored its cluster.

Then, when we process a range query (q, r), we can determine without reading any disk page the set of candidate clusters that can be relevant to the query. This stage needs $|\mathbb{P}|$ distance computations to obtain the permutation Π_q in addition to $|\mathbb{P}|$ calculations of S_ρ distance to compare Π_q with the permutation of each center and determine which clusters have to be reviewed. Next, in order to optimize the necessary time to retrieve all the candidate clusters, we sort the number of disk pages that will be read, because it is cheaper to read disk pages in a sequential way. Then, we order the elements retrieved from the clusters read and compare q with the ordered set, as LCP does.

Figure 2 depicts the same example of Figure 1, but simplified because we want to show mainly the two parts of our index. As in LCP, we can use the parameter f to limit the fraction of more promising database objects that will be compared, via d, with the query q. Besides, if it appears as necessary we can add another parameter s to LCP* that limits the number of disk pages that we will read. In this case, among the list of candidate clusters we select the s more promising. Therefore, it is possible to trade accuracy with distance evaluations and I/O operations as we need, and limit to f the number of distance evaluations and/or to s the number of I/O operations.

5. EXPERIMENTAL RESULTS

In order to evaluate the performance of our LCP*, we select a sample of different metric spaces from SISAP [11]: sets of synthetic vectors on the unitary cube and a real-life database. For lack of space, we only show the results obtained with the synthetic metric spaces. Since our LCP* is an approximated method, we can relax the discarding criteria by accepting bigger shifts. We tabulate these results.

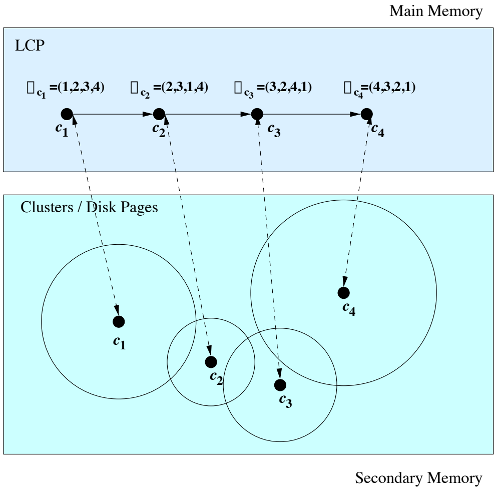

Figure 2: A simplified example of LCP*.

Synthetic Databases: In these experiments we used synthetic databases with vectors uniformly distributed on the unitary cube. We use 100,000 points in different dimensions (5, 10, 15, and 20) under Euclidean distance. As we can precisely control the dimensionality of the space, we use these experiments to show how much the predictive power of our technique varies with the dimensionality. We call these spaces as C5, C10, C15, and C20.

In all cases, we build the index with the 90% of the database elements and we use the remaining 10%, randomly selected, as queries. So, the elements used as query objetcs are not in the index. We average the search costs of all these queries. We also evaluate the effect of using page sizes of 4KB and 8KB, that produce different clusters sizes and number of centers.

Figure 3 illustrates the search costs of query for the synthetic spaces, measured in distance evaluations (left) and number of pages read (right), as the shifting criterion is relaxed. We also evaluate the effect of page sizes for the search performance: 4KB (up) and 8KB (down). As it can be noticed, the number of distance evaluations grows as dimension increases, but sublinearly. Besides, the number of pages read is very low, between 1 an 5 for all cases. Surprisingly, the number of pages read does not decrease as page size increases. This odd behavior can be because as page size increases cluster sizes grows, but the clusters are so big that they can not be discarded easily.

As it is aforementioned, an approximate similarity searching can obtain an inexact answer. That is,

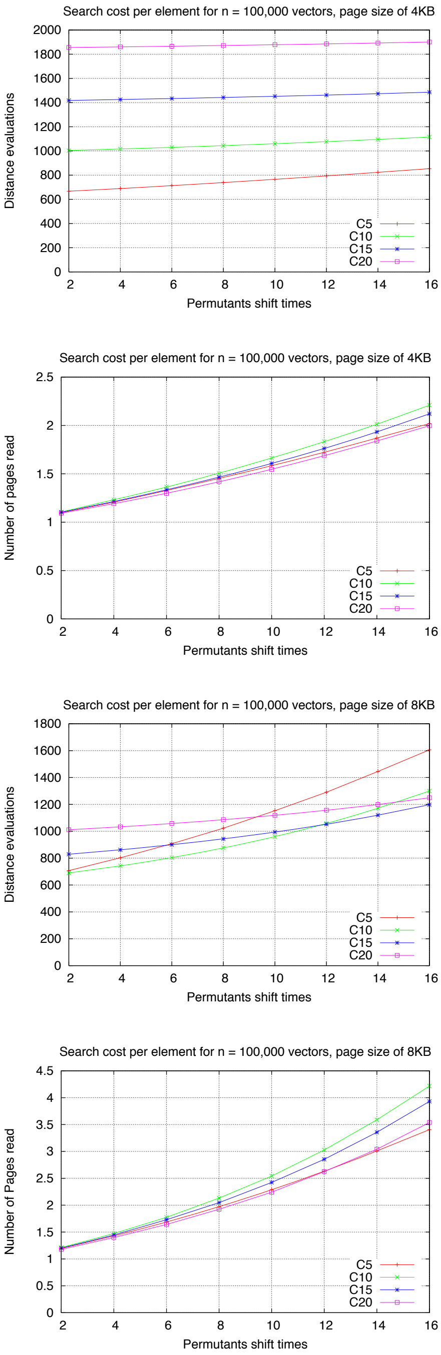

Figure 3: LCP* Search costs for the synthetic spaces, considering page sizes of 4KB and 8KB.

if a 1-NN query of an element $q \in \mathbb{U}$ is posed to the index, it answers with the closest element from $\mathbb{U}$ between only the elements that are actually compared with q. However, as we want to save as many distance calculations as we can, q will not be compared against many potentially relevant elements. If the exact answer of 1-NN$(q) = \{x_1\}$, it determines the radius $r_1 = d(x_1, q)$ needed to enclose x_1 from q. An approximate answer of 1-NN(q) could obtain an element z whose $d(q, z) > r_1$.

Recall is a measure commonly used to evaluate the retrieval effectiveness of a method. It is defined as the ratio of relevant elements retrieved for a given query over the number of relevant elements for that query in the database. This measure take on values between 0 and 1. So, for each query element q the exact 1-NN$(q) = Rel(q)$ is determined with traditional LC. The approximate-1-NN$(q) = Retr(q)$ is answered with LCP* index, let be the set $Retr(q) = \{y_1\}$. It can be noticed that the approximate search will also return one element in this case, so $|Retr(q)| = |Rel(q)| = 1$. Thus, we determine the number of elements obtained which are relevant by verifying if $d(q, y_1) = r_1$. We use recall to analyze the retrieval effectiveness of our proposal in 1-NN queries.

Figure 4 illustrates the recall obtained for the synthetic spaces, as the shifting criterion is relaxed. We evaluate the effect of page sizes for this measure: 4KB (left) and 8KB (right). For this matter, better results are obtained with 8KB than with 4KB. For the other hand, recall decreases as dimension grows with both sizes. For example, for C5 space and 8KB of disk page size, we can obtain a recall of almost 80% only evaluating approximately 1600 distances and just reading 3.5 disk pages.

6. CONCLUSIONS

We have presented a new index for approximate similarity search in secondary memory. The LCP* structure extend an in-memory approximate data structure LCP [9], that offers a good balance between construction and search time. The secondary-memory version also supports approximate searches by calculating few distances and reading very few disk pages. So, we have obtained a more practical index, because it maintain the good characteristics of LCP, but it can be applicated on massive datasets that require secondary memory storage.

As future works we have to analyze if there is a

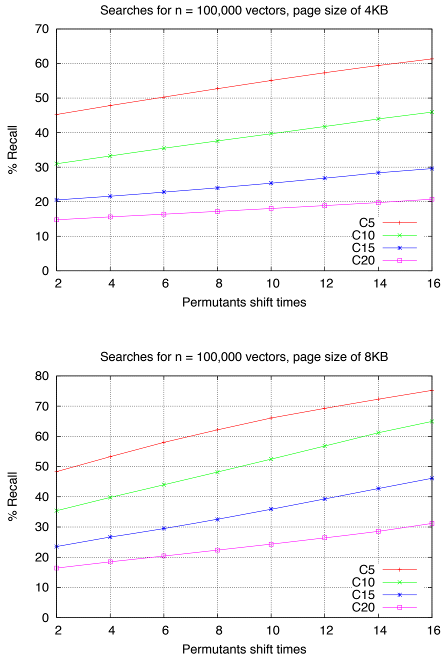

Figure 4: Recall of LCP* for the synthetic spaces, considering page sizes of 4KB and 8KB.

best disk page size for each metric space and to validate our results over larger real-life databases. We also have to check how the performance is affected when we limit the number of disk pages read and the number of distance calculations, by using the parameters f and s. As it is mentioned in [9], we also want to explore the possibility of using short permutations for objects inside the clusters. This is supported by the facts that the beginning of the permutation is the most important data portion to process and that we can trade space in order to improve the recall results.

7. REFERENCES

[1] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comp. Surveys*, 33(3):322–373, Sept. 2001.

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

[3] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Comp. Surveys*, 33(3):273–321, Sept. 2001.

[4] E. Chávez, K. Figueroa, and G. Navarro. Proximity searching in high dimensional spaces with a proximity preserving order. In *Proc. of MICAI 2005, Mexico*, 405–414, 2005.

[5] E. Chavez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1647–1658, 2008.

[6] E. Chávez and G. Navarro. Probabilistic proximity search: fighting the curse of dimensionality in metric spaces. *Inf. Process. Lett.*, 85(1):39–46, Jan. 2003.

[7] A. Esuli. Mipai: Using the pp-index to build an efficient and scalable similarity search system. In *Proc. of SISAP 2009, 29-30 Aug. 2009, Czech Republic*, 146–148, 2009.

[8] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, 28–36, USA, 2003.

[9] K. Figueroa and R. Paredes. List of clustered permutations for proximity searching. In *Proc. of SISAP 2013*, LNCS 8199, 50–58. Springer, 2013.

[10] K. Figueroa, R. Paredes, and R. Rangel. Efficient group of permutants for proximity searching. In *Proc. of MCPR 2011*, LNCS 6718, 42–49. Springer, 2011.

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

[12] G. Navarro and N. Reyes. Dynamic list of clusters in secondary memory. In *Proc. of SISAP 2014*, LNCS 8821, 94–105. Springer Int. Publishing, 2014.

[13] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., USA, 2005.

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