PP-Index: Using Permutation Prefixes for Efficient and Scalable Approximate Similarity Search

Andrea Esuli

Istituto di Scienza e Tecnologie dell'Informazione- Consiglio Nazionale delle Ricerche
via Giuseppe Moruzzi, 1- 56124, Pisa - ITALY
andrea.esuli@isti.cnr.it

ABSTRACT

We present the Permutation Prefix Index (PP-Index), an index data structure that allows to perform efficient approximate similarity search.

The PP-Index belongs to the family of the permutation-based indexes, which are based on representing any indexed object with "its view of the surrounding world", i.e., a list of the elements of a set of reference objects sorted by their distance order with respect to the indexed object.

In its basic formulation, the PP-Index is strongly biased toward efficiency, treating effectiveness as a secondary aspect. We show how the effectiveness can easily reach optimal levels just by adopting two "boosting" strategies: multiple index search and multiple query search. Such strategies have nice parallelization properties that allow to distribute the search process in order to keep high efficiency levels.

We study both the efficiency and the effectiveness properties of the PP-Index. We report experiments on collections of sizes up to one hundred million images, represented in a very high-dimensional similarity space based on the combination of five MPEG-7 visual descriptors.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; H.3.3 [**Information Systems**]: Information Storage and Retrieval—*Search process*

General Terms

Algorithms,Experimentation,Performance

Keywords

approximate similarity search, metric space, scalability

1. INTRODUCTION

The similarity search model [12] is a search model in which, given a query q and a collection of objects D, all belonging to a domain $\mathcal{O}$, the objects in D have to be sorted by their similarity to the query, according to a given *distance function* $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+$ (i.e., the closer two objects are, the most similar they are considered). Typically only the k-top ranked objects are returned (k-*NN query*), or those within a maximum distance value r (*range query*).

One of the main research topics on similarity search is the study of the scalability of similarity search methods when applied to high-dimensional similarity spaces.

The well known "curse of dimensionality" [7] is one of the hardest obstacles that researchers have to deal with when working on this topic. Along the years, such obstacle has been attacked by many proposals, using many different approaches. The earliest and most direct approach consisted in trying to improve the data structures used to perform *exact* similarity search. Research moved then toward the exploration of *approximate* similarity search methods, mainly proposing variants of exact methods in which some of the constraints that guarantee the exactness of the results are relaxed, trading effectiveness for efficiency.

Approximate methods [17] that are not derived from exact methods have been also proposed. On this field, the recent research on *permutation-based indexes* (PBI) [1, 6] has shown a promising direction toward scalable data structures for similarity search.

In this work we present the Permutation Prefix Index (PP-Index), an approximate similarity search structure belonging to the family of the permutation-based indexes. We describe the PP-Index data structures and algorithms, and test it on data sets containing up to 100 million objects, distributed on a very high-dimensional similarity space. Experiments show that the PP-Index is a very efficient and scalable data structure both at index time and at search time, and it also allows to obtain very good effectiveness values. The PP-Index has also nice parallelization properties that allow to distribute both the index and the search process in order to further improve efficiency.

2. RELATED WORKS

The PP-Index belongs to the family of the permutation-based indexes, a recent family of data structure for approximate similarity search, which has been independently introduced by Amato and Savino [1] and Chavez et al. [6].

The PP-Index has however a key difference with respect to previously presented PBIs: as we detail in this section, such PBIs use permutations in order to estimate the real distance order of the indexed objects with respect to a query. The PP-Index instead uses the permutation prefixes in order to quickly retrieve a reasonably-sized set of candidate objects,

which are likely to be at close distance to the query object, then leaving to the original distance function the selection of the best elements among the candidates.

For a more detailed review of the most relevant methods for similarity search in metric spaces we point the reader to the book of Zezula et al. [18]. The recent work of Patella and Ciaccia [8] more specifically analyzes and classifies the characteristics of many approximate search methods.

Chávez et al. [6] present an approximate similarity search method based on the intuition of "*predicting the closeness between elements according to how they order their distances towards a distinguished set of anchor objects*".

A set of *reference objects* $R = \{r_0, \ldots, r_{|R|-1}\} \subset \mathcal{O}$ is defined by randomly selecting $|R|$ objects from D. Every object $o_i \in D$ is then represented by Π_{o_i}, consisting of the list of identifiers of reference objects, sorted by their distance with respect to the object o_i. More formally, Π_{o_i} is a *permutation* of $\langle 0, \ldots, |R| - 1 \rangle$ so that, for $0 < i < |R|$ it holds either (i) $d(o_i, r_{\Pi_{o_x}(i-1)}) < d(o_i, r_{\Pi_{o_x}(i)})$, or (ii) $d(o_i, r_{\Pi_{o_x}(i-1)}) = d(o_i, r_{\Pi_{o_x}(i)})$ and $\Pi_{o_x}(i-1) < \Pi_{o_x}(i)$, where $\Pi_{o_x}(x)$ returns the i-th value of Π_{o_x}.

All the permutations for the index objects are stored in main memory. Given a query q, all the indexed permutations are sorted by their similarity with Π_q, using a similarity measure defined on permutations. The real distance d between the query and the objects in the data set is then computed by selecting the objects from the data set following the order of similarity of their permutations, until the requested number of objects is retrieved. An example of similarity measure on permutations is the *Spearman Footrule Distance* [9]:

$$SFD(o_x, o_y) = \Sigma_{r \in R} |P(\Pi_{o_x}, r) - P(\Pi_{o_y}, r)| \qquad (1)$$

where $P(\Pi_{o_x}, r)$ returns the position of the reference object r in the permutation assigned to Π_{o_x}.

Chávez et al. do not discuss the applicability of their method to very large data sets, i.e., when the permutations cannot be all kept in main memory.

Amato and Savino [1], independently of [6], propose an approximate similarity search method based on the intuition of representing the objects in the search space with "*their view of the surrounding world*".

For each object $o_i \in D$, they compute the permutation Π_{o_i} in the same manner as [6]. All the permutations are used to build a set of inverted lists, one for each reference object. The inverted list for a reference object r_i stores the position of such reference object in each of the indexed permutations. The inverted lists are used to rank the indexed objects by their SFD value (equation 1) with respect to a query object q, similarly to [6]. In fact, if full-length permutations are used to represent the indexed objects and the query, the search process is perfectly equivalent to the one of [6]. In [1], the authors propose two optimizations that improve the efficiency of the search process, not affecting the accuracy of the produced ranking. Both optimizations are based on the intuition that the information about the order of the closest reference objects is more relevant than the information about distant ones.

One optimization consists in inserting into the inverted lists only the information related to $\Pi_{o_i}^{k_i}$, i.e., the part of Π_{o_i} including only the first k_i elements of the permutation, thus reducing by a factor $\frac{|R|}{k_i}$ the size of the index. For example, given $|R| = 500$ a value of $k_i = 100$ reduces by five times the number of disk accesses with respect to using full-length permutations, with a negligible loss in accuracy.

Similarly, a value k_s is adopted for the query, in order to select only the first k_s elements of Π_q. Given $|R| = 500$ a value of $k_s = 50$ reduces by ten times the number of disk accesses, also slightly improving the accuracy.

3. THE PP-Index

The PP-Index represents each indexed object with a *very short* permutation prefix.

The PP-Index data structures consists of a *prefix tree* kept in main memory, indexing the permutation prefixes, and a *data storage* kept on disk, from which objects are retrieved by sequential disk accesses.

This configuration of data structures is interestingly similar to the one used by Bawa et al. [4], however, it is relevant to note that our work and [4] are based on completely different approaches to the problem. The latter proposes the LSH-Forest, an improvement to the LSH-Index [11] that is based on using hash keys of variable lenght. These are used to identify a set of candidate objects with hash keys that have a prefix match with the hash key of to the query. Thus the LSH-Forest, like the other LSH-based methods, is based only on probabilistic considerations, while the PP-Index, like the other PBIs, relies on geometrical considerations.

More generally, a key difference between the PBI model and the LSH model is that the hash functions of the LSH model are solely derived from the similarity measures in use, independently of the way the indexed objects are distributed in the similarity space, while in the SPI model the reference objects provide information about this aspect.

The PP-Index is designed to allow very efficient indexing by performing bulk processing of all the objects indexed. Such bulk processing model is based on the intuitive assumption that the data, in the very large collections the PP-Index is designed for, have a relatively static nature. However, it is easy to provide the PP-Index with update capabilities (see Section 3.6).

3.1 Data structures

Given a collection of objects D to be indexed, and the similarity measure d, a PP-Index is built by specifying a set of *reference objects* R, and a *permutation prefix length* l.

Any object $o_i \in D$ is represented by a *permutation prefix* w_{o_i} consisting of the first l elements of the permutation Π_{o_i}, i.e., $w_{o_i} = \Pi_{o_i}^l$. Any object $o_i \in D$ is also associated with a *data block* b_{o_i}. A data block b_{o_i} contains (i) the information required to univocally identify the object o_i and (ii) the essential data used by the function d in order to compute the similarity between the object o_i and any other object in $\mathcal{O}$.

The prefix tree of the PP-Index is built on all the permutation prefixes generated for the indexed objects. The leaf at the end of a path relative to a permutation prefix w keeps the information required to retrieve the data blocks relative to the objects represented by w from the data storage.

The data storage consists of a file in which all the data blocks are sequentially stored. The order of objects (represented by data blocks) in the data storage is the same as the one produced by performing an ordered visit of the prefix tree. This is a key property of the PP-Index, which allows to use the prefix tree to efficiently access the data storage.

Specifically, the leaf of the prefix tree relative to permutation prefix w contains two counter values h_w^{start} and h_w^{end}, and two pointer values p_w^{start} and p_w^{end}. The counter values

```
BuildIndex(D, d, R, l)
 1  prefixTree ← EmptyPrefixTree()
 2  dataStorage ← EmptyDataStorage()
 3  for i ← 0 to |D − 1|
 4  do o_i ← GetObject(D, i)
 5     dataBlock_{o_i} ← GetDataBlock(o_i)
 6     p_{o_i} ← Append(dataBlock_{o_i}, dataStorage)
 7     w_{o_i} ← ComputePrefix(o_i, R, d, l)
 8     h_{o_i} ← i
 9     Insert(w_{o_i}, h_{o_i}, p_{o_i}, prefixTree)
10  L ← ListPointersByOrderedVisit(prefixTree)
11  P ← CreateInvertedList(L)
12  ReorderStorage(dataStorage, P)
13  CorrectLeafValues(prefixTree, dataStorage)
14  index ← NewIndex(d, R, l, prefixTree, dataStorage)
15  return index
```

Figure 1: The BuildIndex function.

indicate the ordinal position in the sequence of data blocks in the data storage of the first and the last data blocks relative to the permutation prefix w. The pointer values indicate instead the byte offset in the data storage where the data blocks are effectively serialized. In case the data blocks have a fixed size s, the p values can be omitted and computed when necessary as $p_w = h_w \cdot s$.

The data storage implementation must allow, given any two pointers p' and p'', to sequentially retrieve all the data blocks included between them.

3.2   Building the index

Figure 1 shows a pseudo-code description of the indexing function for the PP-Index.

The indexing algorithm first initializes an empty prefix tree in main memory, and an empty file on disk, to be used as the data storage.

The algorithm takes in input an object $o_i \in D$, for i from 0 to $|D − 1|$, and appends its data block at the current end position p_{end} of the data storage file. Then the algorithm computes, for the object o_i, the permutation prefix w_{o_i} (ComputePrefix), and inserts w_{o_i} into the prefix tree. The values $h_{o_i} = i$ and $p_{o_i} = p_{end}$ are stored in the leaf of the prefix tree corresponding to permutation prefix w_{o_i}. When more that one value has to be stored in a leaf, a list is created.

Figure 2 shows an example list of permutation prefixes generated for a set of objects and the data structures resulting from the above discussed first phase of data indexing.

The successive phase (ReorderStorage) consists in reordering the data blocks in the data storage to satisfy the order constrains described in the previous section. An ordered visit of the prefix tree is made in order to produce a list L of the h_{o_i} values stored in the leaves. Thus, the h_{o_i} values in the list L are sorted in alphabetical order of the permutation prefixes their relative objects are associated with.

Data blocks in the data storage are reordered following the order of appearance of h_{o_i} values in list L. For example, given a list for $L = \langle 0, 4, 8, 6, 1, 3, 5, 9, 2, 7 \rangle$, the data block relative to object o_7, identified in the list by the value $h_{o_7} = 7$, has to be moved to the last position in the data storage, since h_{o_7} appears in the last position of the list L (see the values in the leaves of the prefix tree in Figure 2).

To efficiently perform the reordering, the list L is *inverted* into a list P. The i-th position of the list P indicates the new position in which the i-th element of the data storage has to be moved. For the above example,

Index characteristics
|D|=10, |R|=6, l=3

Permutation prefixes
w_{o_0}=<1, 3, 2> w_{o_1}=<2, 3, 0>
w_{o_2}=<5, 2, 3> w_{o_3}=<4, 1, 3>
w_{o_4}=<1, 3, 2> w_{o_5}=<4, 1, 3>
w_{o_6}=<1, 3, 4> w_{o_7}=<5, 2, 3>
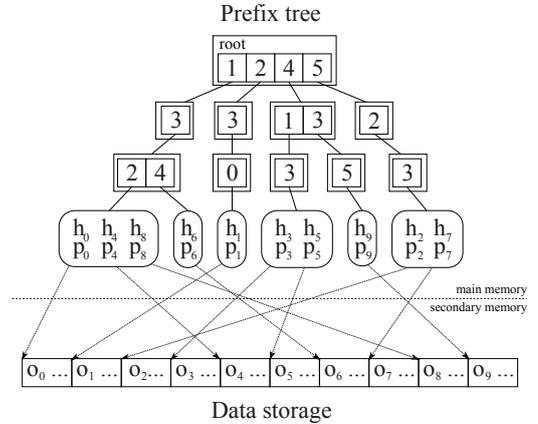w_{o_8}=<1, 3, 2> w_{o_9}=<4, 3, 5>

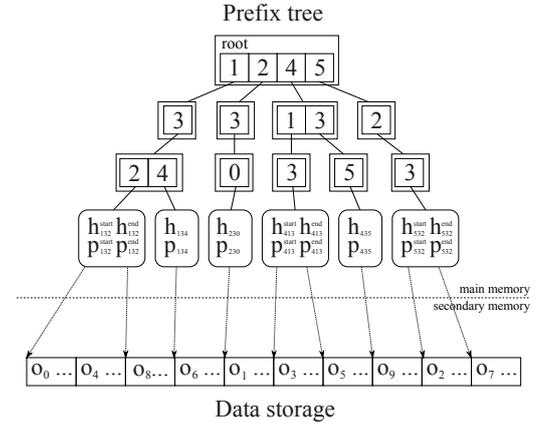**Figure 2: Sample data and partially-built index data structure after the first indexing phase.**

Figure 3: The final index data structure.

$P = \langle 0, 4, 8, 5, 1, 6, 3, 9, 2, 7 \rangle$.

Once P is generated the data storage is reordered accordingly, using an m-way merge sorting method [13]. The advantages of using this method are that it involves only sequential disk accesses, and that it has a small (and configurable) main memory space occupation.

In order to obtain the final index structure, the values in the leaves of the prefix tree have to be updated accordingly to the new data storage (CorrectLeafValues). This is obtained by performing an ordered visit to the prefix tree, the same performed when building the list L, synchronized with a sequential scan of the reordered data storage. The number of elements in the list of a leaf determines the two h^{start} and h^{end} values that replace such list, and also the number of data blocks to be sequentially read from the data storage, in ordered to determine the p^{start} and p^{end} values.

Figure 3 shows the final index data structure.

3.3   Search function

The search function is designed to use the index in order to

```
SEARCH(q, k, z, index)
 1   (p^{start}, p^{end}) ← FINDCANDIDATES(q, index.prefixTree,
 2       index.R, index.d, index.l, z)
 3   resultsHeap ← EMPTYHEAP()
 4   cursor ← p^{start}
 5   while cursor ≤ p^{end}
 6   do dataBlock ← READ(cursor, index.dataStorage)
 7       ADVANCECURSOR(cursor)
 8       distance ← index.d(q, dataBlock.data)
 9       if resultsHeap.size < k
10          then INSERT(resultsHeap, distance, dataBlock.id)
11          else if distance < resultsHeap.top.distance
12              then REPLACETOP(resultsHeap, distance,
13                      dataBlock.id)
14   SORT(resultsHeap)
15   return resultsHeap

FINDCANDIDATES(q, prefixTree, R, d, l, z)
 1   w_q ← COMPUTEPREFIX(q, R, d, l)
 2   for i ← l to 1
 3   do w_q^i ← SUBPREFIX(w_q, i)
 4       node ← SEARCHPATH(w_q^i, prefixTree)
 5       if node ≠ NIL
 6          then minLeaf ← GETMIN(node, prefixTree)
 7              maxLeaf ← GETMAX(node, prefixTree)
 8              if (maxLeaf.h^{end} − minLeaf.h^{start} + 1) ≥ z
 9                  ∨ i = 1
10              then return (minLeaf.p^{start},
11                      maxLeaf.p^{end})
12   return (0, 0)
```

Figure 4: The Search function.

efficiently answer to k nearest neighbor queries. The search strategy consists in searching a subtree of the prefix tree that identifies a specified number of candidate objects all represented by permutation prefixes having a prefix match with the permutation prefix representing the query.

A k-NN query is composed of the query object q, the k value, and the z value, indicating the minimum number of *candidate* objects among which the k nearest neighbors have to be selected.

The FINDCANDIDATES function determines the smallest subtree of the prefix tree having a prefix match with the permutation prefix w_q, i.e., the permutation prefix relative to the query q, and retrieving at least z objects. The function returns two pointers p^{start} and p^{end} to the positions in the data storage of the data blocks of the first and the last candidate objects.

The distance of each candidate object with the query is computed, using the distance function d. A heap is used to keep track of the k objects closest to the query.

3.4   Prefix tree optimizations

In order to reduce the main memory occupation of the prefix tree it is possible to simplify its structure without affecting the quality of results. These are search-specific optimizations, and a non-optimized version of the prefix tree should be saved for other operations (e.g., update and merge).

A first optimization consists in pruning any path reaching a leaf which is composed of only-child, given that this kind of path does not add relevant information to distinguish between different existing groups in the index. Another optimization consists in compressing any path of the prefix tree composed entirely of only-children into a single label [15], thus saving the memory space required to keep the chain of nodes composing the path.

A PP-Index-specific optimization, applicable when the z value is hardcoded into the search function, consists in re-

ducing to a single leaf the subtrees of the prefix tree that points to less than z objects, given that none of such subtrees will be ever selected by the search function.

3.5   Improving the search effectiveness

The "basic" search function described is Section 3.3 is strongly biased toward efficiency, treating effectiveness as a secondary aspect. The PP-Index allows to easily tune effectiveness/efficiency trade-off, and effectiveness can easily reach optimal levels just by adopting the two following "boosting" strategies:

Multiple index: t indexes are built, based on different $R_1 \ldots R_t$ sets of reference objects. This is based on the intuition that different reference object sets produce many differently *shaped* partitions of the similarity space, resulting in a more complete coverage of the area around queries.

A search process the using multiple index strategy can be parallelized by distributing the indexes over multiple machines, or just on different processes/CPUs on the same machine, maintaing almost the same performance of the basic search function, with a negligible overhead for merging the t k-NN results, as far as there are enough hardware resources to support the number of indexes involved in the process.

Multiple query: at search time, p additional permutation prefixes from the query permutation prefix w_q are generated, by swapping the position of some of its elements. The geometric rationale is that a permutation prefix w' differing from another permutation prefix w'' for the swap of two adjacent/near elements identifies an area $V_{w'}$ of the similarity space adjacent/near to $V_{w''}$ allowing to extend the search process to areas of the search space that are likely to contain relevant objects.

The heuristic we adopt in our experiments for swapping permutation prefix elements consists in sorting all the reference objects pairs appearing in the permutation prefix by their difference of distance with respect to the query object. Then the swapped permutation prefixes are generated by first selecting for swap those pairs of identifiers that have the smallest distance difference.

The multiple query strategy can be parallelized by distributing the queries over different processes/CPUs on the machine handling the index structure.

3.6   Update and merge, distributed indexing

The PP-Index data structures allows to very efficiently merge indexes built using the same parameters into a single index. The merge functionality supports three operations:

Supporting update operations: it is easy to add update capabilities to an index by maintaining a few additional data structures. Deleted objects are managed using a vector of their identifiers. Newly inserted or modified objects are stored in an all-in-memory secondary PP-Index used in conjunction with the main index structure. A periodic merge procedure is used when the secondary index reaches a given memory occupation limit.

Indexing very large collection: the main memory occupation of a prefix tree reaches its maximum during the indexing process, when it has to be entirely kept in memory, while during search, thanks to the optimization methods described in Section 3.4, its size can be reduced by orders of magnitude. This issue is solved building a number of smaller partial indexes and then merging them into the final index.

Distributing the indexing process: the indexing process

of smaller indexes can be distributed of different machines, given that the information contained in any smaller index is completely independent of the one contained in the others. Also the merge process can be distributed, if it is performed in a number of steps that involve the creation of intermediate indexes.

The merge process consists in merging the prefix trees of the source indexes into a single prefix tree, i.e., by enumerating, in alphabetical order, all the permutation prefixes contained in the source indexes.

Such enumeration can be easily produced by performing a parallel ordered visit of all the prefix trees being merged. If the prefix trees of the source indexes are saved to the storage using a depth first visit, the merge process requires only a single read of the serialized prefix trees. Obviously, the new prefix tree is directly serialized on disk during the permutation prefix enumeration process.

In the case of an update process, the identifiers of the deleted objects are used to skip deleted objects during the merge process.

The data storages are merged during the permutation prefix enumeration.

4. EXPERIMENTS

4.1 The CoPhIR data set

The CoPhIR[1] [5] data set has been recently developed within the SAPIR project, and it is currently the largest multimedia metadata collection available for research purposes. It consists of a crawl of 106 millions images from the Flickrphoto sharing website.

The information relative to five MPEG-7 visual descriptors [16] have been extracted from each image, resulting in more than 240 gigabytes of XML description data.

We have randomly selected 100 images from the collection as queries and we have run experiments using the first million (1M), ten millions (10M), and 100 millions (100M) images from the data set.

We have run experiments on a linear combination of the five distance functions for the five descriptors, using the weights proposed in [3].

Descriptor	Type	Dimensions	Weight
Scalable Color	L_1	64	2
Color Structure	L_1	64	3
Color Layout	sum of L_2	80	2
Edge Histogram	L_1	62	4
Homogeneous Texture	L_1	12	0.5

Table 1: Details on the five MPEG-7 visual descriptors used in CoPhIR.

4.2 Configurations and evaluation measures

We have explored the effect of using different sized R sets, by running the experiments using three R set sizes consisting of 100, 200, 500, and $1,000$ reference objects. We have adopted a random selection policy of objects from D for the generation of the various R sets, following [6], which reports the random selection policy as a good performer.

In all the experiments we have used a fixed value of $l = 6$.
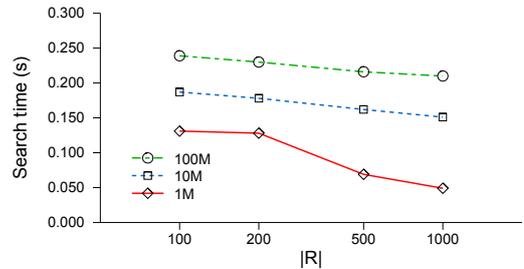
Figure 5: Search time w.r.t. to the size of R, for $z = 1,000$ and $k = 100$ (single index, single query).

We have tested a basic configuration based on the use of a single index and the search function described in Section 3.3, i.e., an efficiency-aimed configuration.

We have then tested the use of multiple indexes , on configurations using 2, 4 and 8 indexes, and also the multiple query search strategy by using a total of 2, 4, and 8 multiple queries. We have also tested the combination of the two strategies.

We have applied the index optimization strategies described in Section 3.4 to all the generated indexes.

On the held-out data we have tested various z values, in this paper we report the results obtained for $z = 1,000$, which has produced a good trade-off in effectiveness/efficiency.

The experiments have been run on a desktop machine running Windows XP Professional, equipped with a Intel Pentium Core 2 Quad 2.4 GHz CPU, a single 1 TB Seagate Barracuda 7,200 rpm SATA disk (with 32 MB cache), and 4 GB RAM. The PP-Index has been implemented in c#. All the experiments have been run in a single-threaded application, with a completely sequential execution of the multiple index/query searches.

We have evaluated the effectiveness of the PP-Index by adopting a *ranking*-based measure and a *distance*-based measure [17], *recall* and *relative distance error*, defined as:

$$Recall(k) = \frac{|D_q^k \cap P_q^k|}{k} \qquad (2)$$

$$RDE(k) = \frac{1}{k}\sum_{i=1}^{k}\frac{d(q, P_q^k(i))}{d(q, D_q^k(i))} - 1 \qquad (3)$$

where D_q is the list of the elements of D sorted by their distance with respect to q, D_q^k is the list of the k closest elements, P_q^k is the list returned by the algorithm, and $L_q^k(i)$ returns the i-th element of the list L.

4.3 Results

$|D|$	indexing time (sec)	prefix tree size		data storage	l'
		full	comp.		
1M	419	7.7 MB	91 kB	349 MB	2.1
10M	4385	53.8 MB	848 kB	3.4 GB	2.7
100M	45664	354.5 MB	6.5 MB	34 GB	3.5

Table 2: Indexing times (with $|R| = 100$), resulting index sizes, and average prefix tree depth l' (after prefix tree compression with $z = 1,000$).

Table 2 reports the indexing times for the various data set sizes ($|R| = 100$), showing the almost perfect linear proportion between indexing time and data set size. With respect to the indexing times we note that: (i) the twelve hours time, required to build the 100M index for the $|R| = 100$, is
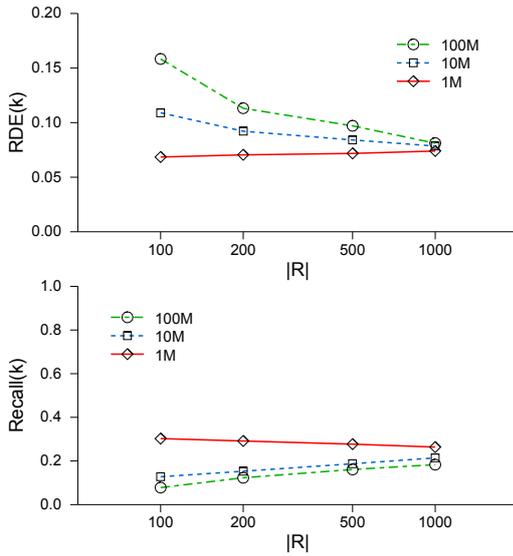
Figure 6: Effectiveness with respect of the size of R set, for $k = 100$ and $z = 1,000$ (single index, single query).

$|R|$	$|D|$		
	1M	10M	100M
100	4,075	5,817	7,941
200	3,320	5,571	7,302
500	1,803	5,065	6,853
1,000	1,091	4,748	6,644

Table 3: Average z' value, for $z = 1,000$.

in line with the fourteen hours we have measured to build a text search index on the descriptions and the comments associated with the indexed images; (ii) this time refers to a completely sequential indexing process, not leveraging on the parallelization possibilities described in Section 3.6; (iii) we have not explored the possibility of using a similarity search data structure in order to answer l-NN query on the R set necessary to build the permutation prefix.

The table also shows the resulting memory occupation of the prefix tree before and after the application of the compression strategies described in Section 3.4. The values shows how such strategies allows to reduce by orders of magnitude the main memory requirement of the PP-Index (at least by a factor fifty in our case) without affecting the quality of the results.

As expected, the disk occupation is perfectly linear with respect to the data set size, given that the disk data storage contains only a sequential serialization of data blocks (375 bytes each one).

The last column of Table 2 reports the average depth of the leaves of the prefix tree, after the compression. The l' values show that the l value is not crucial in the definition of a PP-Index, given that the only requirement is to choose a l value large enough in order to perform a sufficient differentiation of the indexed objects.

The graph of Figure 5 plots the search time with respect to the size of R and the data set size, for $k = 100$ (single index, single query). For the worst case, with $|R| = 100$, we have measured an average 0.239 seconds search time on the 100M index, with an average of less than eight thousands candidates retrieved from the data storage (see Table 3). The search time decreases in a direct proportion the decrease of the z' value, which follows from the more detailed
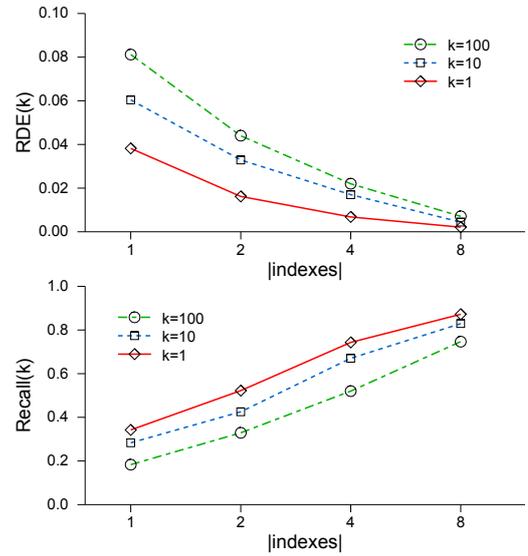

Figure 7: Multiple index search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

partitioning of objects into the permutation prefix space, determined by the increase of $|R|$. Even though the z' value increases as D gets larger, the increase of z' is largely subproportional to the growth of D: when D grows by a factor 100, z' increases at worst by a factor 6.6.

A possible issue with the z' value is that it is not so close the z value, and it has potentially no limits. However, during the experiments the z' value has never been a critical factor with respect to efficiency: we have not observed any extreme case in which $z' >> z$, and the z' value has never required more than a single read operation from disk to retrieve all the candidate objects (e.g., retrieving 10,000 data blocks from the data storage involves reading only 3.7 MB from disk). We leave to future works an investigation of the relations of the z value with the other parameters.

Figure 6 shows the effectiveness of the PP-Index with respect to the size of the R and the data set size, using a single-index/single-query configuration, for $k = 100$.

Effectiveness values improve with the increase of $|R|$ for the 10M and 100M data sets, while the 1M data set shows the inverse tendency. This confirms the intuition that larger data sets requires a richer permutation prefix space (generated by a larger set R) to better distribute their elements, until a limit is reached and objects became too sparse in the permutation prefix space and the effectiveness worsen.

The maximum-efficiency (0.210 seconds answer time) configuration of PP-Index has obtained a 18.3% recall and 8.1% RDE on the 100M data set, for $k = 100$.

Figures 7 and 8 show respectively the effects on effectiveness of the multiple index and multiple query strategies, for three k values.

With respect to the multiple index strategy we have measured a great improvement on both measures reaching a 74% recall (four times better than the single-index case) and a 0.7% RDE (eleven times better) for the eight index case.

For the above mentioned eight index configuration we have measured an average 1.72 seconds search time, for a completely sequential search process. The four index configuration allows to reach a 52% recall (67% for $k = 10$) and just a 2.2% RDE with a sub-second answer time.

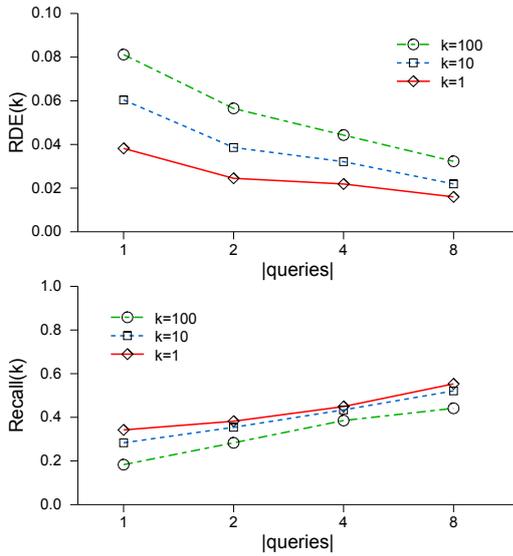It is relevant to note that, given the small memory occu-

Figure 8: Multiple query search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

pation of the compressed prefix tree, we have been able to simultaneously load eight 100M indexes into the memory, thus practically performing search on an 800 million objects index, though with replicated data, on a single computer.

The multiple query strategy also shows relevant improvements, though of minor entity with respect to the multiple index strategy. This is in part motivated by the fact that many of the queries, generated by permuting the elements of the original query permutation prefix, actually resulted in retrieving the same candidates of other queries[2]. On the 100M index, for $|R| = 1,000$, on the average, 1.92 distinct queries to be effective in retrieving candidates for the two queries configuration, 3.18 queries for the four queries configuration, and 5.25 queries for the eight queries configuration.

Figure 9 shows the effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, for $|R| = 1,000$. We have measured an average search time of 12.45 seconds, for a fully sequential search process. This setup produces almost exact results, with a recall $> 97\%$ and a RDE $< 0.01\%$.

We have measured, on the average, a total of 370,000 data blocks retrieved from the data storage among the average 44.5 queries being effectively used to access the data storages for each original query. Although this z' value is relatively high, it just represents the 0.3% of the whole collection. This is a very low value considering, for example, that Lv et al. [14], proposing a multiple query strategy for the LSH-Index, have measured a percentage of distance computations with respect to the data set size, in order to obtain a 96% recall, of 4.4% on a 1.3 million objects data set and of 6.3% on a 2.6 million objects data set.

4.4 Comparison experiments

It is a hard task to run comparative experiments on novel and very large data sets, such as CoPhIR due to many reasons: (i) lack of previous results on the same data set; (ii) lack of a publicly available implementation for many of the methods involved in the comparison; (iii) when an implementation is available, it is typically not designed to scale

---

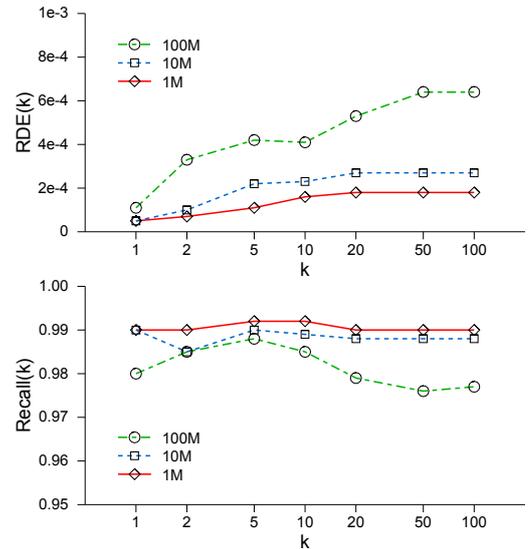[2]Such candidates are read only once from the data storage.

Figure 9: Combined multiple query and multiple index strategies, using eight queries and eight indexes, using $|R| = 1,000$ and $z = 1,000$.

to very large data sets, but just a proof of concept; (iv) moreover, the implementation is usually designed to take in input only a specific data type/format, which makes harder to port the application to different data types.

For this reasons we have currently limited our comparison to replicating two experiments that, among the others, are most closely related to our work: Batko et al. [2], which have run experiments using an early release of the CoPhIR data set, and Amato and Savino [1], whose proposal is the most closely related to our own.

Batko et al. [2] have run experiments on the CoPhIR data set, with data sets size of 10 and 50 millions images[3]. They reports an 80% recall level for 100-NN queries on both collection. For the 50M they test both a 80-CPU infrastructure with 250 GB RAM, keeping all the index data in memory, and a 32-CPU infrastructure with 32 disks storing the index data, obtaining a 0.44 seconds answer time for the memory-based solution and 1.4 seconds for the disk-based one.

The PP-Index could achieve a better performance than [2] by distributing the search process, yet using much less resources than [2]. We have simulated the distribution of the eight indexes on eight distinct computers, each one using two processes executing four queries each, measuring the query answer time as the time of the slowest of the 16 processes plus the time to merge the 16 100-NN intermediate results. We have measured an average 1.02 second answer time to obtain $> 95\%$ recall on the 50M data set.

Amato and Savino [1] test their method on the Corel data set[4]. The data set consists of 50,000 32-dimensions color HSV histograms extracted from the images. The distance function used to compare the histograms is L_1.

Replicating [1], we have selected 50 random objects as queries, and indexed the rest of the collection. Given the small size of the data set, we have set $|R| = 50$. The time required for generating the PP-Index is 4.9 second, with a

---

[3]We suppose they use the same linear combination of visual descriptors of our experiments, given that two authors are also the authors of [3], from which we take our weights.
[4]http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html

disk occupation of 13 MB and a memory occupation of 450 kB. In [1] the index structure generated by setting $k_i = 100$ is estimated to require 20 MB. This value does not include the HSV histograms, which are required to reorder the retrieved objects by the true similarity.

The maximum recall level obtained in [1] for $k = 50$ is about 54%, requiring to read 2.4 MB of data from disk (600 blocks of 4 kB size). The PP-Index, in a single-index/four-query configuration ($z = 500$), obtains a 89.6% recall, requiring to read just 900 kB of data from disk, in four sequential reads. The single-index/single-query configuration obtains a 66% recall.

5. CONCLUSIONS

We have introduced the PP-Index, an approximate similarity search data structure based on the use of short permutation prefixes. We have described the PP-Index data structures and algorithms, including a number of optimization methods and search strategies aimed at improving the scalability of the index, its efficiency, and its effectiveness.

The PP-Index has been designed to take advantage of the relatively static nature one could expect from very large collections. However, as we have described, it is easy to support fast update operations.

We have evaluated the PP-Index on a very large and high-dimensional data set. Results show that it is both efficient and effective in performing similarity search, and it scales well to very large data sets.

We have shown how a limited-resources configuration obtains good effectiveness results in less than a second, and how almost exact results are produced in a relatively short amount of time. The parallel processing capabilities of the PP-Index allow to distribute the search process in order to further improve its efficiency.

The comparison with experimental results published for two closely related method, which are among the top-performers on the task, shows that the PP-Index outperforms the compared methods, both in efficiency and effectiveness. Only one [2] of the works we compare with uses a data set of a size comparable to our largest one. We plan to extend the comparison with some of the competing methods, by porting them on the larger data set sizes.

The PP-Index has been already used to build a performing similarity search system[5] [10].

There are many aspect of our proposal that are worth to be further investigated. For example, the R set is a crucial element of the PP-Index. We plan to study element selection policies alternative to the random policy, e.g., selecting centroids of clusters of D, or the most frequent queries from a query log.

6. REFERENCES

[1] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *INFOSCALE '08: Proceeding of the 3rd International ICST Conference on Scalable Information Systems*, pages 1–10, Vico Equense, Italy, 2008.

[2] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubský, and P. Zezula. Crawling, indexing, and similarity searching images on the web. In *Proceedings of SEDB '08, the 16th Italian Symposium on Advanced Database Systems*, pages 382–389, Mondello, Italy, 2008.

[3] M. Batko, P. Kohoutkova, and P. Zezula. Combining metric features in large collections. *SISAP '08, 1st International Workshop on Similarity Search and Applications*, pages 79–86, 2008.

[4] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 651–660, Chiba, Japan, 2005.

[5] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627, 2009.

[6] E. Chávez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 30(9):1647–1658, 2008.

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

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, Athens, Greece, 1997.

[9] P. Diaconis. Group representation in probability and statistics. *IMS Lecture Series*, 11, 1988.

[10] A. Esuli. MiPai: using the PP-Index to build an efficient and scalable similarity search system. In *SISAP '09, Proceedings of the 2nd International Workshop on Similarity Search and Applications*, Prague, CZ, 2009.

[11] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the 30th ACM symposium on Theory of computing*, pages 604–613, Dallas, USA, 1998.

[12] H. V. Jagadish, A. O. Mendelzon, and T. Milo. Similarity-based queries. In *PODS '95: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 36–45, San Jose, US, 1995.

[13] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, chapter 5.4: External Sorting, pages 248–379. second edition, 1998.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 950–961, Vienna, Austria, 2007.

[15] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[16] MPEG-7. Mpeg-7. multimedia content description interfaces, part3: Visual, 2002. ISO/IEC 15938-3:2002.

[17] M. Patella and P. Ciaccia. The many facets of approximate similarity search. *SISAP '08, 1st International Workshop on Similarity Search and Applications*, pages 10–21, 2008.

[18] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. 2005.

---

[5] http://mipai.esuli.it/