PP-Index: Using Permutation Prefixes for Efficient and Scalable Similarity Search
(Extended Abstract)

Andrea Esuli

Istituto di Scienza e Tecnologie dell'Informazione - CNR
`andrea.esuli@isti.cnr.it`

Abstract. The Permutation Prefix Index (PP-Index)[1] is a data structure that allows to perform efficient approximate similarity search. It is a permutation-based index, which is 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 biased toward efficiency. We show how the effectiveness can reach optimal levels just by adopting two "boosting" strategies: multiple index search and multiple query search, which both have nice parallelization properties. We study both the efficiency and the effectiveness properties of the PP-Index, experimenting with collections of sizes up to one hundred million objects, represented in a very high-dimensional similarity space.

1 Introduction

The similarity search model [2] 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). The k-top ranked objects are returned (*k-NN query*), or those within a maximum distance value r (*range query*).

The well known "curse of dimensionality" [3] 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 to the problem 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 [4] that are not derived from exact methods have been also proposed. Recently, the research on *permutation-based indexes* (PBI) [5, 6] has shown a another promising direction toward scalable data structures for similarity search.

---

[1] This is an extended abstract version of [1].

2 The PP-Index

The Permutation Prefix Index (PP-Index) belongs to the family of the *permutation-based indexes* (PBI), independently introduced by Amato and Savino [5], and Chavez et al. [6]. The intuition behind PBIs is that two similar objects have a *similar view of the surrounding world*, i.e., they are likely to see the elements of a set of *reference objects* in the same order of distance, according to a given distance function that models the concept of similarity.

The key difference between the PP-Index and previously presented PBI methods [5, 6] is that the latters use permutations in order to estimate the real distance order of the indexed objects with respect to a query, while the PP-Index uses the permutation prefixes in order to quickly retrieve, in a hash-like manner, a reasonably-sized set of candidate objects that are likely to be at close distance to the query, then leaving to the original distance function the selection of the best elements among the candidates.

More formally, given a set of objects D, from a domain $\mathcal{O}$, and a distance function $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+$, the PP-Index is built by computing a *permutation prefix* Π_o^l for each $o \in D$. Π_o^l is the sequence of the identifiers of the l closest reference objects to o with respect to d, determined on a set of reference objects $R = \{r_0, \ldots, r_{|R|-1}\} \subset \mathcal{O}$, e.g., R could consist of a random sampling of the elements of D.

Data structures: The PP-Index data structures consists of a *prefix tree* kept in main memory, indexing the permutation prefixes and keeping pointers to a *data storage* kept on disk, which stores *data blocks* that contains (i) the information required to univocally identify any object $o_i \in D$ 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 $\mathcal{O}$. Data blocks are sequentially sorted in the data storage by the alphabetical order of their permutation prefixes, where the alphabet is defined by the identifiers of the reference objects, ordered by their natural value, i.e., the first letter of the alphabet is 0 and the last is $|R| - 1$. Figure 1 shows an example list of permutation prefixes generated for a set of objects and the resulting PP-Index.

Basic search function: Given a k-NN query for an object $q \in \mathcal{O}$, the basic search function of the PP-Index consists of computing the permutation prefix Π_p^l and searching for the longest prefix match in the prefix tree whose subtree points to at least z *candidate objects*. Then the k-NN result is computed on such subset of $z' \geq z$ candidate objects, using the distance function d. Given the order of the data blocks in the data storage, the z' objects are all stored in adjacent positions of the data storage, thus allowing an extremely efficient sequential access to data.

Making an analogy with the inverted list indexing model for text, we can consider each indexed object o_i as a document composed by a single *word w_o*, i.e., the permutation prefix Π_o^l. The data storage holds the *posting lists* that, in the PP-Index case, are sequences of data blocks related to objects that are represented by the same permutation prefix Π^l. The prefix three in memory indexes the *lexicon* of permutation prefixes, i.e., all the distinct permutation prefixes extracted

from the indexed objects, in order to optimize the prefix queries performed on the lexicon by the PP-Index search function. From this analogy it is easy to see how the indexing process of the PP-Index can be efficiently parallelized and distributed over multiple processors/machines using, e.g., a MapReduce framework [7] in which the Map function generates the permutation prefixes and the Reduce function sorts and merges the data blocks with respect to the associated permutation prefix.
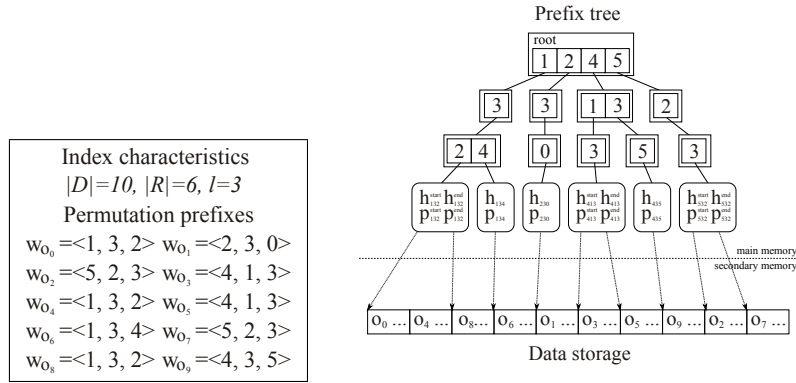

Fig. 1. Sample data and resulting PP-Index data structures.

Improved search function: The above described "basic" search function is strongly biased toward efficiency. The PP-Index allows to tune the effectiveness/efficiency trade-off 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.

Multiple query: at search time, p additional permutation prefixes from the query permutation prefix Π_q^l are generated, by swapping the position of some of its elements. The geometric rationale is that a permutation prefix Π_x^l differing from another permutation prefix Π_y^l for the swap of two adjacent/near elements identifies an area $V_{\Pi_x^l}$ of the similarity space adjacent/near to $V_{\Pi_y^l}$. Performing a search with additional "swapped" permutation prefixes extends the search process to areas of the search space that are likely to contain relevant objects.

In our experiments the swapping heuristic 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.

3   Experiments

3.1   The CoPhIR data set

We have run our experiments on the CoPhIR[2] [8] data set, which consists of a crawl of 106 millions images from the Flickr photo sharing website. Five MPEG-7 visual descriptors are associated to each image.

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 used the remaining 6 million images as held-out data for some preliminary experiments.

We have run experiments on a linear combination of the five distance functions for the five descriptors. As the weights for the linear combination we have adopted those proposed in [9].

3.2   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 the results of [6], which reports the random selection policy as a good performer. In all the experiments we have used a fixed value of $l = 6$. As the results show, the determination of the value l is not a critical choice, as long as it is large enough to produce a detailed partitioning of objects in the permutation prefix space.

We have tested a basic configuration based on the use of a single index and the basic search function described in Section 2, 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, i.e., generating 1, 3, and 7 additional queries from the original query permutation prefix. We have also tested the combination of the two multiple index/multiple query strategies. We have optimized the z value on the held-out data, determining and optimal $z = 1,000$, which has produced a good trade-off in effectiveness/efficiency.

The experiments have been run on a desktop machine, equipped with a 2.4 GHz CPU, a single 1 TB 7,200 rpm SATA disk, and 4 GB RAM. 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 [4], *recall* and *relative distance*
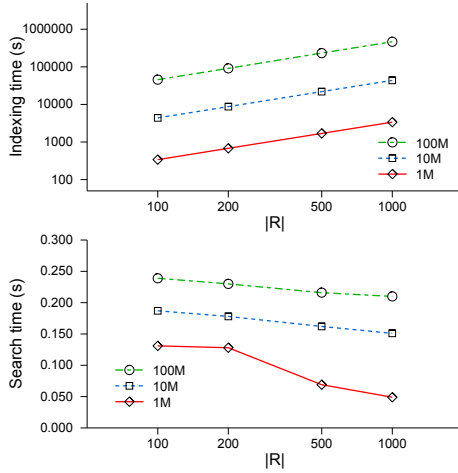
---

[2] http://cophir.isti.cnr.it/

Fig. 2. Indexing time and search time w.r.t. to the size of R and the data set size (search with $z = 1,000$ and $k = 100$, single index, single query).
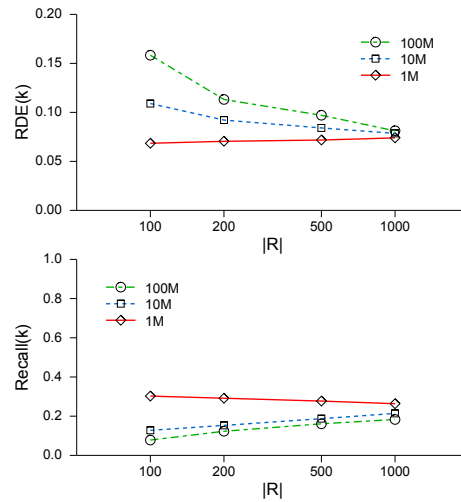

Fig. 3. Effectiveness with respect of the size of R set, for $k = 100$ and $z = 1,000$ (single index, single query).

error, defined as:

$$Recall(k) = \frac{|D_q^k \cap P_q^k|}{k} \qquad (1)$$

$$RDE(k) = \frac{1}{k} \sum_{i=1}^{k} \frac{d(q, P_q^k(i))}{d(q, D_q^k(i))} - 1 \qquad (2)$$

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.

In order to evaluate the efficiency of the PP-Index we have measured: the indexing time, the main memory and disk occupation, the search time, the number of data blocks read from disk, and the sequentiality of disk accesses.

3.3 Results

Figure 2 shows the almost linear proportion of the index time cost with respect to both the size of the reference object set size and the data set size. With respect to the indexing times we note that:

– the twelve hours time, required to build the 100M index for the $|R| = 100$, is 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;
– this time refers to a completely sequential indexing process, not leveraging on the parallelization possibilities of the PP-Index.
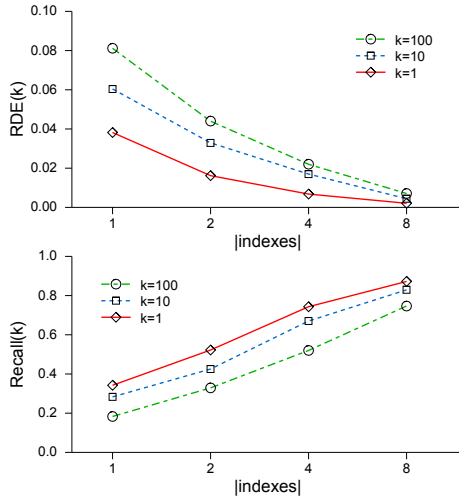
Fig. 4. Effectiveness of the multiple index search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.
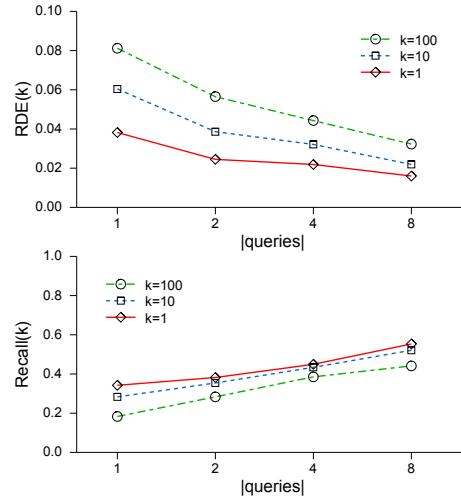
Fig. 5. Effectiveness of the multiple query search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

Figure 2 also 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, $|R| = 100$, we have measured an average 0.239 seconds search time on the 100M index.

We have measured the effectiveness of the PP-Index for various sizes of the R set and the data set size, using a single-index/single-query configuration, and $k = 100$ (Figure 3). Effectiveness values improve with the increase of $|R|$ for the 10M and 100M data sets, while the 1M data set shows the inverse tendency, due to a over-partitioning of the space. For $|R| = 1,000$ we observe almost the same relative error ($\sim 8\%$), for the three data set sizes. The 100M recall curve is very close to the 10M recall curve, with an average difference of 3%.

The most efficient (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 4 and 5 show respectively the effects on effectiveness of the multiple index and multiple query strategies, for three k values. For 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. Just by adding a single index to the original one we have measured a factor two improvement of both measures.

As expected, the search cost for the multiple index strategy is exactly proportional to the number of explored indexes. 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 scores 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 occupation of the compressed
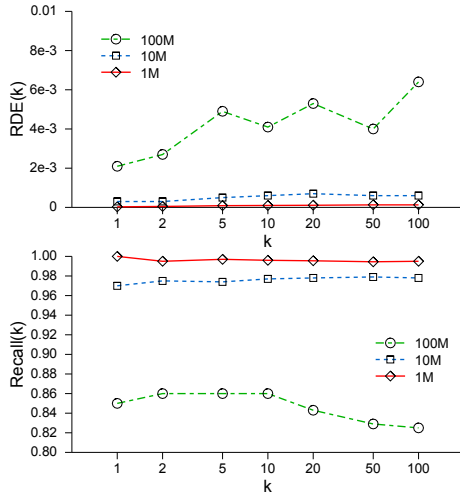
Fig. 6. Effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, on various data set sizes, using $|R| = 100$, and $z = 1,000$.
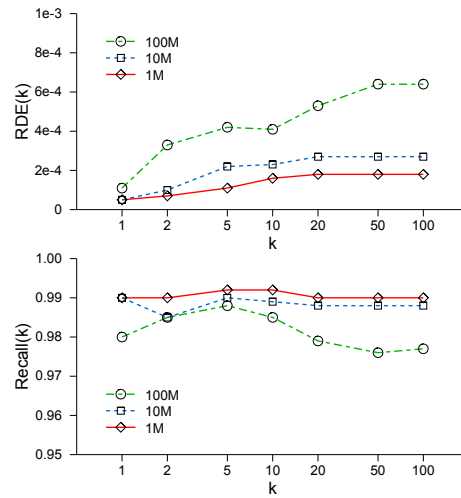
Fig. 7. Combined multiple query and multiple index strategies, using eight queries and eight indexes, using $|R| = 1,000$ and $z = 1,000$.

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[3]. 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. These values directly reflect the average search cost with respect to the single query configuration, i.e., only the effective queries (44.5 on average) produce a disk access to read candidates from data storage.

Figures 6 and Figure 7 show the effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, respectively for $|R| = 100$ and for $|R| = 1,000$. We have measured almost the same search times, for a completely sequential search process, for the two cases (12.45 seconds for $|R| = 1,000$). The case $|R| = 100$ produces a 82.5% recall and a RDE < 0.1% on the 100M index, while the smaller data set sizes it scores almost exact results (recall > 96%). The case $|R| = 1,000$ produces almost exact results, with a recall > 97% and a RDE < 0.01%.

---

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

4   Conclusions

We have presented the PP-Index, an approximate similarity search data structure based on the use of short permutation prefixes. The PP-Index has interesting parallel processing properties both at indexing time and at search time.

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. Moreover, the parallel processing capabilities of the PP-Index allow to distribute the search process in order to further improve its efficiency.

The PP-Index has been already used to build a performing similarity search system[4] [10]. By mining the query log of the service we plan to investigate new policies for the definition of the R set, e.g., clustering queries and selecting the most representative ones of each cluster.

References

1. Esuli, A.: Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. In: Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR09). (2009) 17–24
2. Jagadish, H.V., Mendelzon, A.O., Milo, T.: Similarity-based queries. In: PODS '95: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, San Jose, US (1995) 36–45
3. Chavez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Proximity searching in metrix spaces. ACM Computing Surveys **33**(3) (2001) 273–321
4. Patella, M., Ciaccia, P.: The many facets of approximate similarity search. SISAP '08, 1st Int. Workshop on Similarity Search and Applications (2008) 10–21
5. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: INFOSCALE '08: Proceeding of the 3rd International ICST Conference on Scalable Information Systems, Vico Equense, Italy (2008) 1–10
6. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI) **30**(9) (2008) 1647–1658
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1) (2008) 107–113
8. Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: CoPhIR: a test collection for content-based image retrieval. CoRR **abs/0905.4627** (2009)
9. Batko, M., Kohoutkova, P., Zezula, P.: Combining metric features in large collections. SISAP '08, 1st International Workshop on Similarity Search and Applications (2008) 79–86
10. Esuli, A.: 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)

---

[4] `http://mipai.esuli.it/`