# Use of 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*
*Tel.: +39-050-3153054*
*andrea.esuli@isti.cnr.it*

**Abstract**

We present the Permutation Prefix Index[1] (PP-Index), an index data structure that supports 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. We show how the effectiveness can easily 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.

*Keywords:* approximate similarity search, metric space, scalability

## 1. Introduction

The similarity search model (Jagadish et al., 1995) 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* $d : \mathcal{O} \times \mathcal{O} \rightarrow \mathbb{R}^+$ (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 $k$-NN query model is the most frequently adopted in similarity search, because of (i) the ability to control the result set dimension, which is a very desirable property of the retrieval process (Patella and Ciaccia, 2009), and (ii) the fact that in most of the similarity search applications in which objects are distributed in a high-dimensional space, the definition of a meaningful distance range $r$ is not obvious.

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" (Chavez et al., 2001) is one of the hardest obstacles that researchers have to deal with when working on this topic. Over 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 (Patella and Ciaccia, 2009) that are not derived from exact methods have been also proposed. One successful contribution of this kind is the *Local Similarity Hashing* (LSH) model proposed

[1] This work is a revised and extended version of Esuli (2009b), presented at the 2009 LSDS-IR Workshop, held in Boston.

by Indyk and Motwani (1998), followed in the years by a number of interesting extensions (Bawa et al., 2005; Lv et al., 2007). More recently, the recent research on *permutation-based indexes* (PBI) (Amato and Savino, 2008; Chávez et al., 2008; Skala, 2009) has shown another 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 PBIs. We test the PP-Index 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 obtains high effectiveness values. The PP-Index has also nice parallelization properties that support the distributed execution of both the index and the search processes in order to further improve efficiency.

### *1.1. Outline*

Section 2 describes the works that are most closely related to PP-Index, also introducing some of the concepts at the base of our work. Section 3 describes the PP-Index. Experiments and results are reported in Section 4. Section 5 concludes.

## 2. Related works

The PP-Index belongs to the family of the permutation-based indexes, a recent family of data structures for approximate similarity search, which has been independently introduced by Amato and Savino (2008) and Chávez et al. (2008). The following sections describe such works and also introduce more formally the concepts at the base of PBIs. 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. (2005). The recent work of Patella and Ciaccia (2009) more specifically analyzes and classifies the characteristics of many approximate search methods.

### *2.1. Ordering permutations*

Chávez et al. (2008) 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 $\Pi_{o_i}$, consisting of the list of identifiers of reference objects, sorted by their distance with respect to the object $o_i$. More formally, $\Pi_{o_i}$ 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 $\Pi_{o_x}$.

Figure 1 shows the permutation-based partitions of the two-dimensional space generated by a set of randomly picked reference points, using the Euclidean distance as the similarity measure.

Given a query $q$, all the indexed permutations are sorted by their similarity with $\Pi_q$, using a similarity measure defined on permutations, e.g., the *Spearman Footrule Distance* (Diaconis, 1988):

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

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

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.

Chávez et al. (2008) hold all the full length permutations into the main memory and the authors do not discuss the applicability of the method to very large data sets, i.e., when the permutations cannot be all kept in main memory.

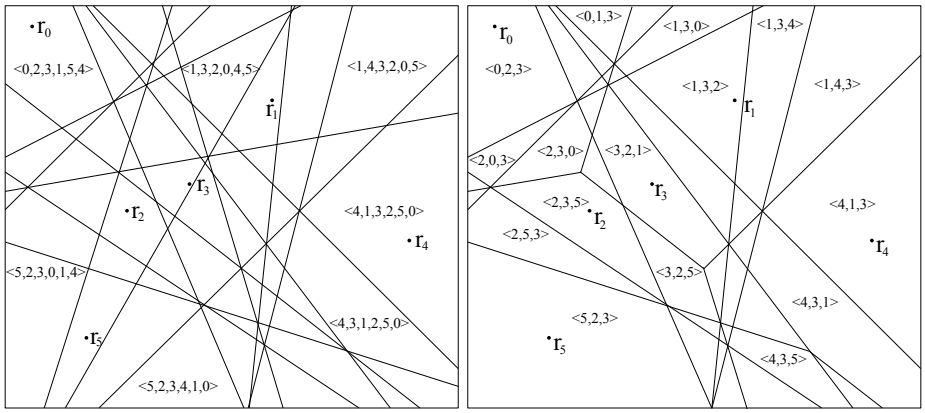


Figure 1: Permutations generated by a set of reference points on a plane, using the Euclidean distance as the similarity measure, and the relative space partitions generated by choosing a permutation prefix length $l = 3$.

### *2.2. Metric Inverted File*

Amato and Savino (2008), independently of Chávez et al. (2008), propose the *Metric Inverted File*, 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 $\Pi_{o_i}$ in the same manner as Chávez et al. (2008). 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 Chávez et al. (2008). 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 Chávez et al. (2008). Amato and Savino (2008) propose an efficiency optimization based on indexing in the inverted lists only the information related to $\Pi_{o_i}^{k_i}$, i.e., the part of $\Pi_{o_i}$ including only the first $k_i$ elements of the permutation, thus reducing by a factor $\frac{|R|}{k_i}$ the size of the index. A similar processing is applied to queries, by using a $k_s$ prefix length. The intuition, confirmed by their experiments, is that the information about the order of the closest reference objects is more relevant than the information about distant ones.

### *2.3. Theoretical properties of permutation-based space*

Skala (2009) presents a study on the space-partitioning properties of permutation-based indexes. The main hypothesis on which permutation-based indexes are based is that a relatively small number of reference objects can produce a very high number of permutations, following the exponential growth properties of permutations. However, the permutations used in similarity search problems are connected to regions of a similarity space, and not all permutations are deemed to exist in such space.

Skala proves a number of theoretical results that provide bounds on the number of possible permutations with respect to the number of reference objects, the distance measure ($L_1$, $L_2$, $L_{\inf}$), and the dimensionality of the similarity space. Although Skala's work does not directly affect our work, we regard it as a first interesting investigation toward a better theoretically-founded model for permutation-based indexes.

### *2.4. M-Index*

Novak and Batko (2009) propose the M-Index, a similarity search data structure that uses a *universal mapping schema* to map the elements of a metric space $\mathcal{U}$ to a numeric domain. The mapping method is based on the use of *pivots*, i.e., reference points, and a normalized metric distance function $d : \mathcal{U} \times \mathcal{U} \rightarrow [0, 1)$. The output of the mapping function for an object $o$ is directly related to the permutation describing the pivots sorted by their relative distance with respect to $o$.

M-Index provides both exact search, that exploits the various metric-based principles of data partitioning, pruning and filtering, an approximate search, that uses a search method similar to multi-probe LSH (Lv et al., 2007). Novak et al. (2010) prove that the M-Index satisfies all the properties required by LSH methods (Indyk and Motwani, 1998).

### *2.5. LSH methods*

The configuration of data structures of the PP-Index is interestingly similar to the one proposed by Bawa et al. (2005) for the LSH-Forest, however, it is relevant to note that two works are based on completely different approaches to the problem.

The LSH-Forest is an improvement of the LSH-Index (Indyk and Motwani, 1998). The LSH-Index is based on the notion of *locality-sensitive hash function family* $\mathcal{H}$.

A family $\mathcal{H}$ of functions from a domain $\mathcal{O}$ to a range $U$ is called $(r, \epsilon, p_1, p_2)$-sensitive, with $r, \epsilon > 0$, $p_1 > p_2 > 0$, if for any $p, q \in \mathcal{O}$:

- if $d(p, q) \leq r$ then $\mathbb{P}[h(p) = h(q)] \geq p_1$
- if $d(p, q) > r(1 + \epsilon)$ then $\mathbb{P}[h(p) = h(q)] \leq p_2$

for any hashing function $h$ randomly selected from $\mathcal{H}$.

Intuitively, two objects hashed by a LSH function have a higher probability (at least $p_1$) to "collide" if they are closer than $r$, and a lower probability (at most $p_2$) if they are more distant than $r(1 + \epsilon)$.

The LSH-Index uses $j$ randomly chosen functions $h_i \in \mathcal{H}$ to define a hash function $g(x) = (h_1(x)$ $h_2(x) \dots h_j(x))$. Thus, if two distant objects have a probability $p_2$ to collide for a single $h_i$ function, such probability is significantly lowered to $p_2^j$ by using the $g$ function. In order to maintain a relatively high probability of producing a collision between nearby objects, $t$ different hash tables are built, based on randomly generated $g_1 \dots g_t$ functions.

Given a query object $q$, the various $g_x(q)$ hashes are computed and all the objects in $D$ that have at least a matching hash are considered for the computation of the real distance with the query.

Bawa et al. (2005) extend the LSH-Index by proposing the LSH-Forest, in which the hash keys are indexed by a prefix tree (LSH-Tree). Many LSH-Tree are created, each one based on a different hash function, forming the LSH-Forest. Given a query, the length of the hash key is *shortened*, from the original length of $j$ elements, until at least $z$ candidate objects in the hash table have a prefix match with $g_x(q)$.

An efficient implementation of an LSH-Tree is almost equivalent to the one used for the PP-Index (described in Section 3). However, there are relevant differences to be noted.

The LSH-Forest, like the other LSH methods, is based only on probabilistic considerations, while the PP-Index, like the other PBI methods, relies on geometrical considerations. Shortening the prefix match on permutation prefixes in the PP-Index means releasing some geometrical constraints and retrieving more candidates, but that added candidates are still guaranteed to be relatively close to the query, given that the remaining matching reference objects in the prefix are those which are closer to both the query and each candidate. In the LSH-Forest a shorter prefix match results only in a decreased probability of candidates being related to the query, with the increased probability of inclusion of very distant candidates.

The hash functions of the LSH methods 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 PBI methods the reference objects provide information about this aspect.

A relevant point in favor to PBI methods is their *black box* use of the distance function. This, for example, makes possible to use a distance function that is a linear combination of other distance functions[2], even of different kind[3]. The locality-sensitive hash function family $\mathcal{H}$ of LSH methods relies instead on processing the actual data representing the indexed objects, e.g., sampling bits from the vector of features, and it is not possible to handle non-basic distance functions, e.g., there is no LSH function family for non-trivial linear combination of metrics.

[2]Note that a linear combination of metric distances is still a metric distance.

[3]In our experiments we combine $L_1$ and $L_2$ metrics of different dimensionality

## 3. The PP-Index

Esuli (2009b) describes the PP-Index giving relevance to the algorithmic/technical details necessary to realize an efficient implementation of the PP-Index. This paper describes it at a higher level of abstraction, devoting more space to the comparison with similar data structures.

### *3.1. Data structures and basic search function*

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* $l$. The PP-Index represents each indexed object $o_i$ with a *very short* permutation prefix $w_{o_i} = \Pi^l_{o_i}$, with $l \ll |R|$.

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 $|R| - 1$.

Given a $k$-NN query for an object $q \in \mathcal{O}$, the basic search function of the PP-Index consist of computing the permutation prefix $w_q$ 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, resulting in 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 $w_{o_i}$. 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 $w$. The prefix tree 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 (Dean and Ghemawat, 2008). In the MapReduce model, the indexing process is composed by:

- a *Map* function that generates the permutation prefix for each object being indexed (the generation of a permutation prefix for an object is completely independent from any other object in the collection);
- a *Reduce* function that sorts and merges the data blocks with respect to the associated permutation prefix, e.g., using an $m$-way merge sorting method (Knuth, 1998).

Figure 2 shows an example list of permutation prefixes generated for a set of objects and the resulting PP-Index data structures.

The key difference between the PP-Index and previously presented PBI methods is that the PBI methods of Amato and Savino (2008) and Chávez et al. (2008) 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.

### *3.2. Improving the search effectiveness*

The "basic" search function described in Section 3.1 is strongly biased toward efficiency, treating effectiveness as a secondary aspect. With the PP-Index it is possible to tune effectiveness/efficiency trade-off, and effectiveness can easily reach optimal levels just by adopting the two following "boosting" strategies:

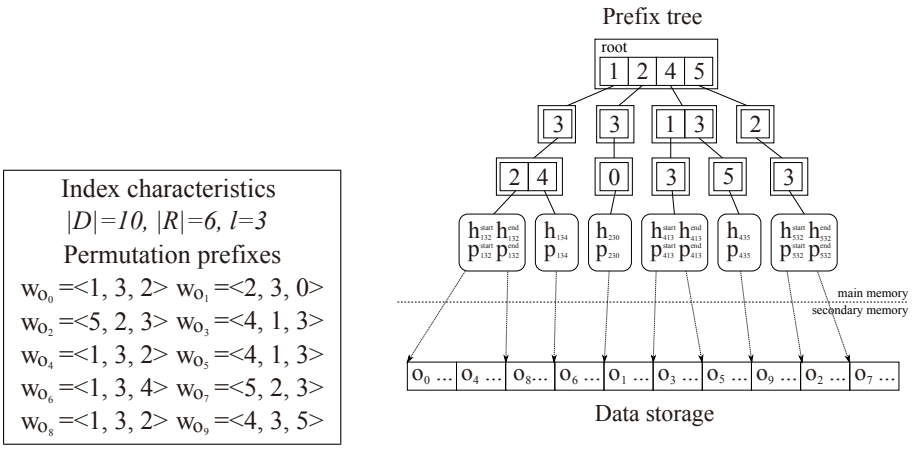


Figure 2: Sample data and resulting PP-Index data structures.

*Multiple index*: $t$ indexes are built, based on different $R_1 \dots R_t$ 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.

The multiple index strategy obviously results in a lot of data replication. This could be considered a waste of disk space when implemented on a single machine and on a relatively small sized collection. However, it has to be considered that PP-Index is designed for very large collections, in which a single data storage could potentially require the entire disk space of a single machine. In this case data replication among multiple machines could be a positive plus, providing data redundancy and fault tolerance.

A search process using the multiple index strategy can be parallelized by distributing the indexes over multiple machines, or just on different processes/CPUs on the same machine, thus maintaining 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 $V_{w''}$. Performing a search additional "swapped" permutation prefixes results in extending the search process to areas of the search space that are likely to contain relevant objects. For example, the permutation prefixes $\langle 2, 3, 5 \rangle$ and $\langle 3, 2, 5 \rangle$, which differ by the swap of the identifiers 2 and 3, identify two adjacent regions in Figure 1.

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.

## 4. Experiments

The experiment section is divided into two parts. The first part extends the results already presented in Esuli (2009b), which are focused on testing the effectiveness and efficiency of the PP-Index on a very large and high dimensional collection. The second part reports on various comparison experiments with other methods.

| 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, and the weights used in the linear combination. The "Dim." column refers to the dimensionality of visual descriptors adopted by the CoPhIR data set.

### *4.1. The CoPhIR data set*

The CoPhIR[4] (Bolettieri et al., 2009) data set has been recently developed within the SAPIR project[5], and it is currently the largest multimedia metadata collection available for research purposes. It consists of a crawl of 106 millions images from the Flickr[6] photo sharing website.

The information relative to five MPEG-7 visual descriptors (MPEG-7, 2002) have been extracted from each image, resulting in more than 240 gigabytes of XML description data. As detailed in Bolettieri et al. (2009), the cost of extracting the MPEG-7 features from a single image is about four seconds, resulting in about twelve years to extract the features for all the images in the CoPhIR data set[7].

We have randomly selected 100 images from the data set as queries, and excluded them from indexing. We have run experiments on three index sizes, selecting for indexing the first million (1M), ten millions (10M), or 100 millions (100M) images from the data set. We have used the remaining six million images, never included in any index, 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 Batko et al. (2008b), listed in Table 1.

### *4.2. Configurations*

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 Chávez et al. (2008), 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 3, i.e., an efficiency-aimed configuration. We have tested the relation of the parameter $z$ with the other parameters of the problems, i.e., $|R|$ and the dataset size.

We have tested the use of multiple indexes (see Section 3.2), 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.

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#[8]. All the experiments have been run in a single-threaded application, with a completely sequential execution of the multiple index/query searches.

[4] `http://cophir.isti.cnr.it/`

[5] `http://www.sapir.eu`

[6] `http://www.flickr.com`

[7] A grid infrastructure composed of 73 nodes has been in used to perform image download and features extraction, which has been able to process about half million images a day.

[8] The implementation is able to run also on Linux+Mono.

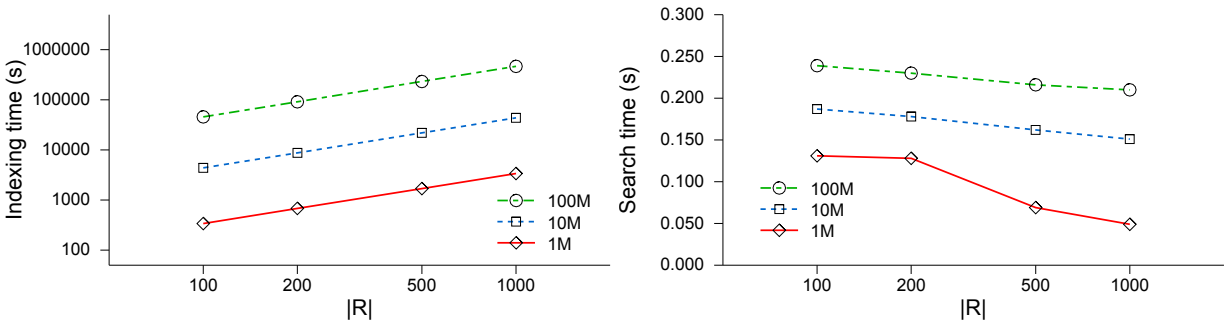


Figure 3: Indexing time and search time w.r.t. to the size of $R$ and the data set size (search with $z = 1,000$ and $k = 100$, single index, single query).

| $\|D\|$ | indexing time (sec) | prefix tree size | | data storage | $l'$ |
|---|---|---|---|---|---|
| | | full | comp. | | |
| 1M | 419 | 7.7 MB | 91 kB | 0.34 GB | 2.1 |
| 10M | 4385 | 53.8 MB | 848 kB | 3.4 GB | 2.7 |
| 100M | 45664 | 354.5 MB | 6537 kB | 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 $z = 1,000$), for the various data set sizes.

### *4.3. Evaluation measures*

We have evaluated the effectiveness of the PP-Index by adopting a *ranking*-based measure and a *distance*-based measure, *recall* and *relative distance error*, which are defined as (Patella and Ciaccia, 2009):

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

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

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 $L$.

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.

### *4.4. Results*

Figure 3 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.

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:

- the twelve hours time, required to build the 100M index for the $|R| = 100$, is comparable with the fourteen hours we have measured to build a text search index on the descriptions and the comments associated with the indexed images;
- the 120 hours time, required to build the 100M index for the $|R| = 1000$ is a relatively long time, but it is still a very fast performance considering that it is about 900 times faster than the time required to extract the visual descriptor from the images (see Section 4.1);

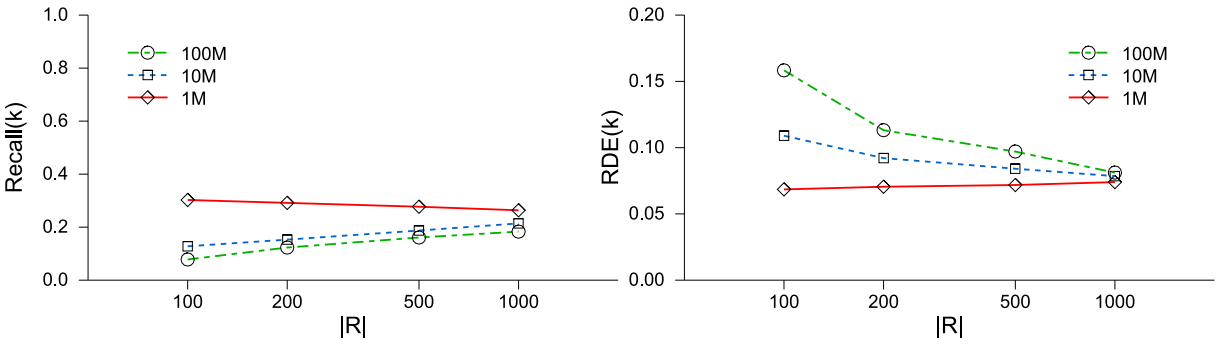


Figure 4: Effectiveness with respect of the size of $R$ set, for $k = 100$ and $z = 1,000$ (single index, single query).

- this time refers to a completely sequential indexing process, not leveraging on the parallelization possibilities mentioned in Section 3.1;
- we have not explored the possibility of using a similarity search data structure in order to answer $l$-NN queries on the $R$ set necessary to build the permutation prefix.

This is a worst-case configuration, involving all the five descriptors. Among the descriptors, the distance function associated with the HT descriptor is the one that has the highest computational cost, requiring about the 40% of the total time. The rest of the time is almost equally distributed among the other descriptors. In fact, building a 100M index with $|R| = 1,000$ for one of the non-HT descriptors requires about eighteen hours.

The table also shows the resulting memory occupation of the prefix tree before and after the application of the compression strategies described in Esuli (2009b) (e.g., pruning of subtrees pointing to less than $z$ objects). The values show how the optimization produce a reduction by orders of magnitude of the main memory space occupation (at least by a factor fifty in our case) without affecting the quality of the results, making the PP-Index a truly effective solution for working on very large data sets.

As expected, the disk occupation is perfectly linear with respect to the data set size, given that the data store on disk contains only a sequential serialization of data blocks[9].

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. For example, when $z = 1,000$, the choice of the value $l = 6$ has been sufficient to create a prefix tree capable of distributing the indexed objects in the permutation prefix space in groups composed of less than $1,000$ objects.

The graph of Figure 3 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 ($z' \geq z$ is the number of candidates actually retrieved from the data storage, see Section 3.1), which follows from the more detailed partitioning of objects into the permutation prefix space, determined by the increase of $|R|$.

Figure 4 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.

[9]The serialization of a data block on disk requires 375 bytes.

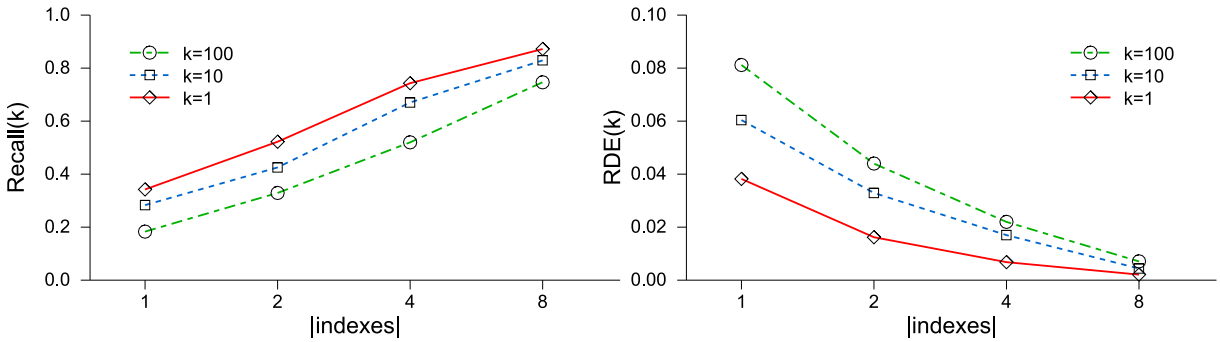


Figure 5: Effectiveness of the multiple index search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

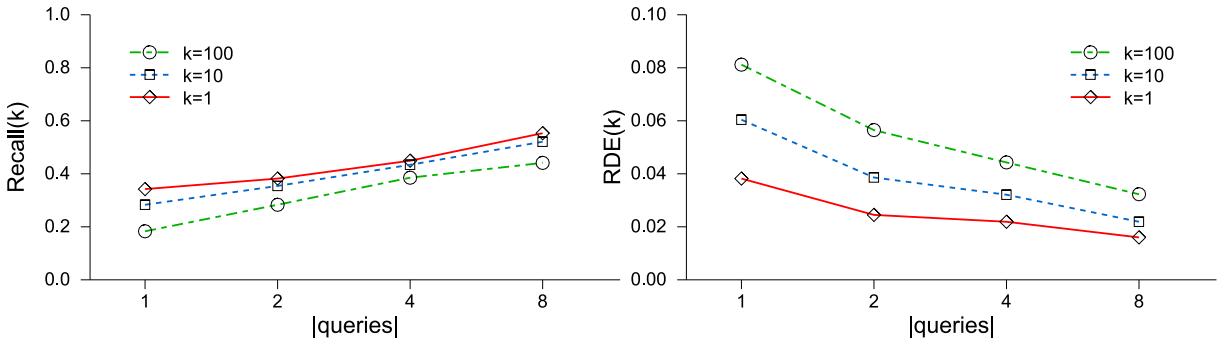


Figure 6: Effectiveness of the multiple query search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

For $|R| = 1,000$ we observe almost the same relative error ($\sim 8\%$), for the three data set sizes. With respect to recall it is interesting to note the 100M curve is very close to the 10M curve, with an average difference of 3%.

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 $k = 100$. Effectiveness values improve for smaller $k$ values, as it is shown in the Figures 5 and 6.

On the single index/single query configuration, we have studied the impact of the $z$ on search effectiveness and the relation between the $z$ value and the $z'$ value, by running experiments using different $z$ values in combination with different $|R|$ values.

Table 3 reports the $z'$ values measured when using $z$ values of 100, 1,000 and 10,000 on the 1M, 10M and 100M indexes, and various sizes of the $R$ set (100, 200, 500, and 1,000). The average $z'$ values are all substantially larger than the relative $z$ value, however we have never observed extremely high $z'$ values, e.g., the highest $z'$ value for a query is 187,032 on the $z = 10,000$, $|R| = 1,000$, 100M configuration. For $z = 1,000$, the experiments the $z'$ value has never been a critical factor with respect to efficiency, and no query has 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).

From the values in Table 3 two trends clearly emerge:

- the $z'$ value increases as $D$ gets larger. The increase of $z'$ is largely sub-proportional to the growth of $D$. For example, when $D$ grows by a factor 100, $z'$ increases at worst by a factor 6.6, when $z = 1,000$.
- the $z'$ value gets closer to the $z$ value as $R$ gets larger. This is related to the more detailed partitioning of the similarity space that a larger $R$ set generates.

Figure 7 shows the recall and RDE values obtained for the above mentioned configurations, limited to the 100M index, for $k = 100$. As one might expect, a larger $z$ value results in better recall and RDE values.

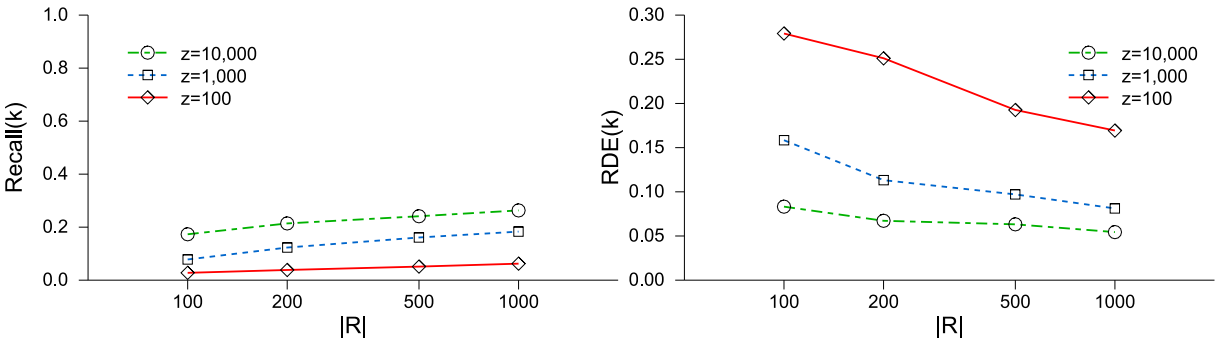


Figure 7: Effectiveness with respect of the $z$ an $|R|$ values, for $k = 100$, on the 100M index (single index, single query).

| | $z = 100$ | | | $z = 1,000$ | | | $z = 10,000$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\|D\|$ | | | $\|D\|$ | | | $\|D\|$ | | |
| $\|R\|$ | 1M | 10M | 100M | 1M | 10M | 100M | 1M | 10M | 100M |
| 100 | 320 | 1,230 | 1,926 | 4,075 | 5,817 | 7,941 | 15,300 | 50,231 | 80,890 |
| 200 | 256 | 1,078 | 1,653 | 3,320 | 5,571 | 7,302 | 13,120 | 47,342 | 73,332 |
| 500 | 194 | 723 | 1,295 | 1,803 | 5,065 | 6,853 | 11,921 | 35,321 | 61.231 |
| 1,000 | 130 | 624 | 1,032 | 1,091 | 4,748 | 6,644 | 11,203 | 31,201 | 54,302 |

Table 3: Average $z'$, i.e., average number of retrieved candidate objects for a query, value measured for various values of $z$, $R$, and different index sizes.

However, a ten-fold increase of the $z$ value results in a limited increase of recall, specially between the $1,000$ and the $10,000$ values. Other search strategies, such as using multiple queries, obtain a larger increase of recall with respect to the increase of number of candidates been retrieved from the index. Thus, the use of large $z$ values is not a viable strategy to improve effectiveness in an efficient way. Among the three tested values, $z = 1,000$ seems to achieve the best overall effectiveness/efficiency trade-off on the various test configurations.

Figures 5 and 6 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. Just by adding a single index to the original one we have measured a factor two improvement of both measures.

The search cost for the multiple index strategy is exactly proportional to the number of explored indexes. For the eight index configuration we have measured an average 1.72 seconds search time, for a completely sequential search process. The four index configuration reaches 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 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 relevance 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[10]. On the 100M index, when $|R| = 1,000$, only 1.92 distinct queries (on average) are effectively used to retrieve candidates for the two queries configuration. The average number of effective queries is 3.18 for the four queries configuration, and 5.25 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

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

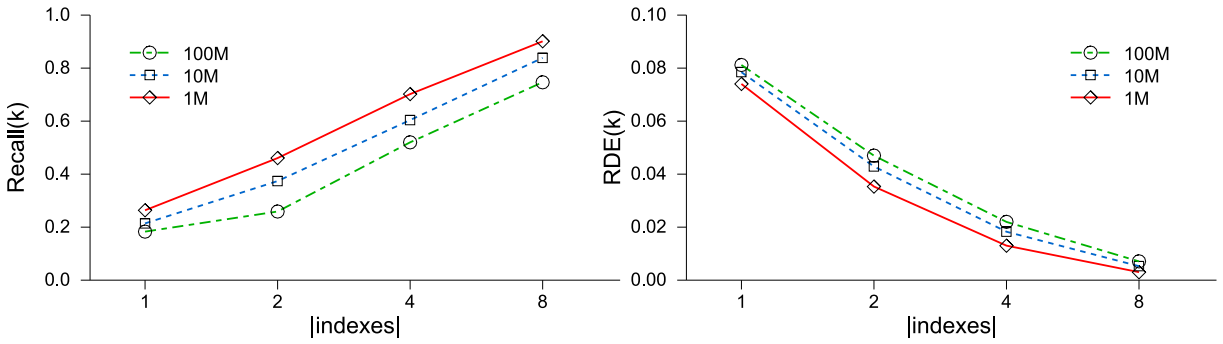


Figure 8: Effectiveness of the multiple index search strategy on various index sizes, using $k = 100$, $|R| = 1,000$, and $z = 1,000$.

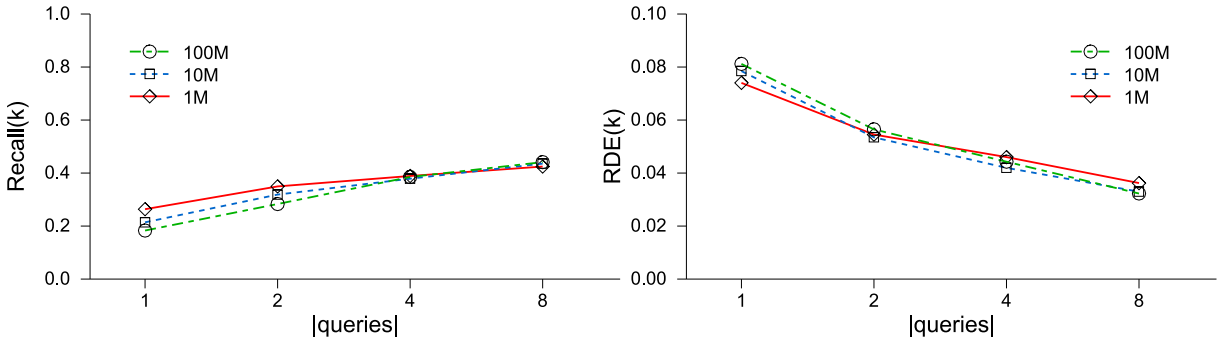


Figure 9: Effectiveness of the multiple query search strategy on various index sizes, using $k = 100$, $|R| = 1,000$, and $z = 1,000$.

queries produce a disk access to sequentially read candidates from data store.

Figure 10 and 11 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 $< 0.01\%$.

In this latter case 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. For the four index/single query case the average percentage of objects being accessed is 0.03%. These are very low values considering, for example, that Lv et al. (2007), proposing a similar 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.

### *4.5. 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:

- lack of previous results on the same data set;
- lack of a publicly available implementation for many of the methods involved in the comparison;
- difficulty and high cost, both in human and hardware resources, to develop and run a precise implementation of the proposed methods, and also to optimize each of them for the new data type;

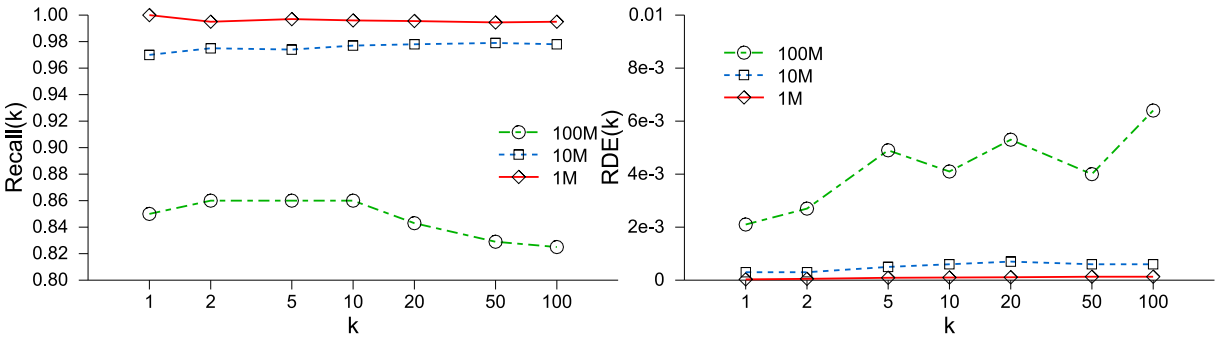


Figure 10: Effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, on various data set sizes, using $|R| = 100$, and $z = 1,000$.

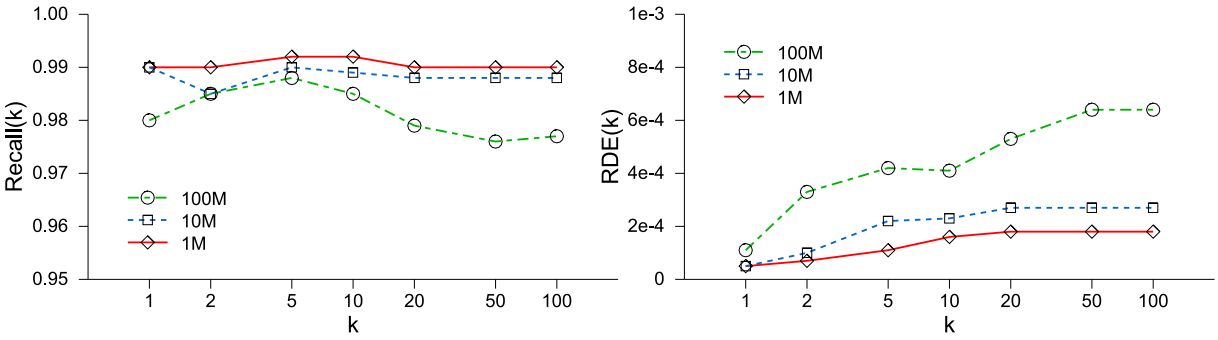


Figure 11: Combined multiple query and multiple index strategies, using eight queries and eight indexes, using $|R| = 1,000$ and $z = 1,000$.

- when an implementation is available, it is typically a not an "industrial strength" version, i.e., it is not designed to scale to very large data sets but just a proof of concept;
- moreover, the implementation is usually designed to take in input only a specific data type/format, which makes it difficult to port the application to different data types.

For this reasons, in order to present a broad comparison with related methods, we have adopted a different experimental setup for each compared method, adapting the experimental setup to the available material, e.g., the availability of software implementations or results of previous experiments on publicly available data sets.

### *4.6. LSH-Forest, comparison on the CoPhIR data set*

Given the strict similarity between the data structures of the PP-Index and those of the LSH-Forest it has been possible for us to produce a reliable implementation the LSH-Forest (Bawa et al., 2005). As pointed out in Section 2.5, there is currently no evident way in the LSH model to define a LSH family $\mathcal{H}$ for a weighted linear combination of metric distances of different nature, we have thus limited our comparison to the *Color Structure* visual descriptor of each CoPhIR image (see Table 1).

The Color Structure visual descriptor uses a $L_1$ norm distance function applied to a 64 dimensions space. Following Datar et al. (2004), we have generated the hashing functions by randomly sampling from a family of hash function of the form $h(\mathbf{v}) = \left\lfloor \frac{\mathbf{a}\cdot\mathbf{v}+b}{r} \right\rfloor$, where $\mathbf{v}$ is the vector with the values of visual descriptor, $\mathbf{a}$ a vector of randomly sampled values from a Cauchy distribution (which is 1-stable), $r = 5$ (optimized with experiments on held out data), and $b$ is randomly sampled with uniform distribution from $[0, r]$.

With respect to LSH-Forest parameters we set the maximum depth $l = 6$ (the same value used for the PP-Index), thus each hashing function is composed by six hashing function, $g(v) = h_1(v) \dots h_6(v)$. Each $h(v)$ function returns an integer number, which is indexed by the nodes of the prefix tree.

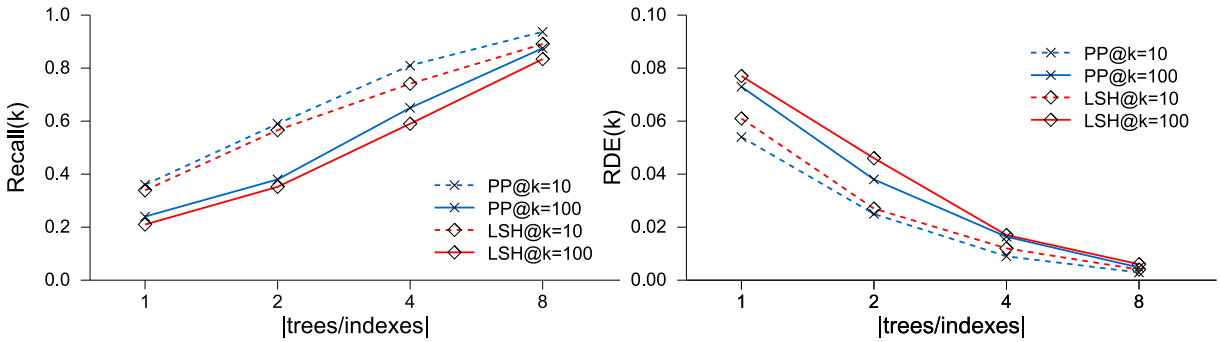


Figure 12: Comparison between PP-Index and the LSH-Forest, varying $k$, on the 10M data set.

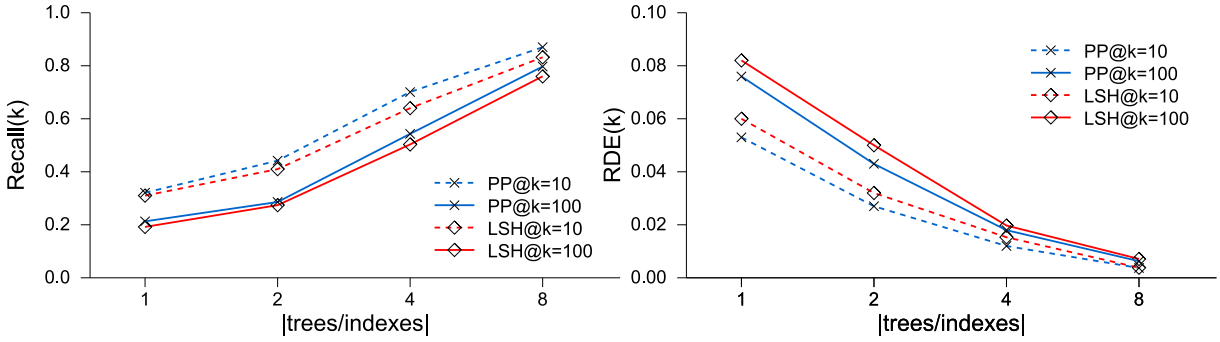


Figure 13: Comparison between PP-Index and the LSH-Forest, varying $k$, on the 100M data set.

We compared the LSH-Forest and the PP-Index by varying the number of LSH-Trees/PP-Indexes from 1 to 8, setting $z = 1,000$ (and $|R| = 100$, for the PP-Index).

Figures 12 and 13 show the comparison of the recall and RDE, averaged on 100 random queries, for various data set sizes, varying the $k$ value, respectively on the 10M and the 100M data sets. The PP-Index obtained, on average, a 6.68% better recall and a 15.98% better RDE with respect to the LSH-Forest, obtaining a maximum recall on the 100M data set, $k = 100$, of 79.7% (76.0% for the LSH-Forest).

These values, specially the RDE improvement, indicate how the criterium of closeness to the reference objects adopted by the PP-Index prefixes produces a better grouping of similar objects, improving the chance of finding the true $k$ most similar objects and also improving the overall similarity of the retrieved objects for those retrieved objects that do not belong to the set of the true $k$ most similar objects.

The PP-Index selected on average 1.12% less candidates than the LSH-Forest, however the high variance in this value ($\pm 6.32$) does not allow us to consider this value as statistically relevant.

### *4.7. M-Chord, comparison on the CoPhIR data set*

Batko et al. (2008a) have run experiments using an early release of the CoPhIR data set, with data set sizes of 1, 10 and 50 millions images. As the similarity function we suppose they use the same linear combination of visual descriptors of our experiments, given that two authors are also the authors of Batko et al. (2008b), from which we take our weights. The similarity search method they use is M-Chord, a P2P data structure for metric-based similarity search. M-Chord uses a one-dimensional P2P protocol in order to distribute the indexed objects among peers. Differently from multiple index PP-Index, the indexed data is only distributed and not replicated among peers, thus requiring less space resources. In M-Chord, multiple queries can be simultaneously processed by the peers, as long as the search process for the queries involves different peers.

In Batko et al. (2008a) the network of peers is deployed over various cluster configurations with variable resources. For the 10M data set they use a 16-CPU infrastructure with 64 GB RAM, keeping all the index

data in memory, obtaining a 0.44 seconds answer time, with $k = 50$. For the 50M data set 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 0.45 seconds answer time for the memory-based solution and 1.4 seconds for the disk-based one. They report an 80% recall level for 50-NN queries on both collection.

Given the very different nature of the two approaches, it has been difficult to set up an experimental comparison based on the use of similar data structure-specific parameters, thus, thank also to the hardware details reported in Batko et al. (2008a), we chose to compare the methods based on the hardware resources they use.

We have distributed the eight PP-Indexes on 16 CPUs, i.e., eight distinct computers (a total of 32 GB RAM) each one with dual core processors, executing four queries on each CPU (the pair of CPUs on the same computer accessed the same PP-Index stored on a single disk), measuring the query answer time as the time of the slowest of the 16 processes plus the time to merge the sixteen 50-NN intermediate results. This configuration, which uses about half of the resources of the disk-based M-Chord configuration on the 50M data set, has obtained, on 100 random queries, an average 1.02 second answer time and a 95.3% recall on the 50M data set. Table 4 summarizes the results of the comparison.

| | | search time (seconds) | | | recall | | | RDE | | |
|---|---|---|---|---|---|---|---|---|---|---|
| method | CPU | 1M | 10M | 50M | 1M | 10M | 50M | 1M | 10M | 50M |
| M-Chord (exact results) | 16 | .45 | 1.70 | - | 100.0% | 100.0% | 100.0% | 0.00% | 0.00% | 0.00% |
| M-Chord with approximation | 16 | - | .44 | - | - | >80.0% | - | - | - | - |
| M-Chord with approximation | 80 | - | - | .45 | - | - | ≈80.0% | - | - | - |
| M-Chord with approximation and disk | 32 | - | .87 | 1.40 | - | - | ≈80.0% | - | - | - |
| PP-Index | 16 | .37 | .77 | 1.02 | 99.2% | 97.7% | 95.3% | 0.02% | 0.03% | 0.07% |

Table 4: Comparison between PP-Index and the M-Chord, with $k = 50$, varying the data set size. M-Chord values are reported from Batko et al. (2008a). Recall values for M-Chord have been derived from Figures 2 and 3 of Batko et al. (2008a).

### *4.8. Metric Inverted File, comparison on the Corel data set*

In order to compare the PP-Index with Metric Inverted File we decided to replicate the experiments of Amato and Savino (2008) on the Corel data set, which is publicly available from the UCI Knowledge Discovery in Database Archive[11].

The data set consists of $50,000$ 32-dimensions color HSV histograms extracted from the images. The HSV color space is divided into 32 subspaces (using 8 ranges of hue and 4 ranges of saturation). The value in each dimension in the HSV histogram of an image is the density of each color in the entire image. The distance function used to compare the histograms is $L_1$.

Replicating Amato and Savino (2008), we have selected 50 random objects as queries, and indexed the rest of the collection. Given the small size of the data set, the PP-Index has been configured with $|R| = 50$, with reference objects randomly selected from the indexed objects, and $l = 6$. As summarized in Table 5, in Amato and Savino (2008) the Metric Inverted File index structure is reported to require 20 MB. This value does not include the HSV histograms, which are required if one wants to reorder the retrieved objects by their true similarity. The time required for generating the PP-Index is 4.9 second, with a disk occupation of 13 MB (including HSV histograms) and a memory occupation of 450 kB.

The maximum recall level obtained in Amato and Savino (2008) for $k = 50$ is 54%, requiring to read 2.4 MB of inverted list data from disk (600 blocks of 4 kB size). The PP-Index, in a single-index/single-query configuration ($z = 500$) obtains a 66% recall ($k = 50$), requiring to read just 230 kB of data from disk, always with a single sequential read access to disk. Average search time is 0.01 seconds. In a single-index/four-query configuration ($z = 500$), the PP-Index obtains a 89.6% recall ($k = 50$), reading about 575 kB of data from disk, requiring an average of 2.5 sequential reads from disk. Average search time is 0.02 seconds.

Figure 14 compares the recall levels obtained by the Metric Inverted File and the two PP-Index configuration with respect to various $k$ values. It is relevant to note the different trend in recall of the PP-Index

[11] `http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html`

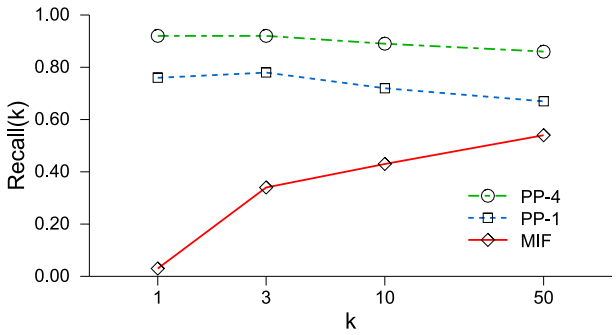


Figure 14: Comparison between PP-Index and the Metric Inverted File, on the Corel Data Set. Recall values for Metric Inverted File are taken from Amato and Savino (2008).

| method | # indexes | index size on disk | indexing time | multiple queries | search time (secs) | data read from disk | | | sequential disk read accesses |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | disk blocks | z' | bytes | |
| MIF | 1 | 20 MB (w/o HSV data) | ≈1 min | no | - | 600 ($k_i = 100$, $k_s = 50$) | - | 2.4 MB | 50 |
| PP-Index | 1 | 13 MB (w/ HSV data) | 4.9 sec | no | 0.01 | - | 898 | 230 kB | 1 |
| PP-Index | 1 | 13 MB (w/ HSV data) | 4.9 sec | 4 | 0.02 | - | 2246 | 575 kB | 2.5 |

Table 5: Comparison between PP-Index and the Metric Inverted File, on the Corel Data Set. Recall values for Metric Inverted File are taken from Amato and Savino (2008).

with respect to the Metric Inverted File. The recall of the PP-Index improves as $k$ gets smaller, this is motivated by the fact that the set of $z'$ candidates does not change for the various $k$ values and thus it is slightly more probable to find the 1-NN object from such set than all the 50-NN objects. The recall of the Metric Inverted File instead gets worse as $k$ gets smaller, this is motivated by the fact that the ranking by similarity of the retrieved objects is done only by the estimated SFD (see Section 2.2) not reranking object by their real distance. The estimated SFD is a relatively coarse-grained measure and thus it is very hard to correctly identify the $k$ closest objects, from the set of objects retrieved by scanning the inverted lists, when $k$ is very small.

Using real distances in the Metric Inverted File would have improved the results but also would have required to read also the HSV vectors from disk in order to compute the real distances, increasing the amount of data read from disk and the also sparseness of disk accesses, given that the Metric Inverted File does not provide any policy for the organization and the optimization of access to the data representing the indexed objects.

## 5. Conclusions

We have presented the PP-Index, an approximate similarity search data structure based on the use of short permutation prefixes.

In its basic formulation, the PP-Index is strongly biased toward efficiency. We have shown how the effectiveness can easily reach optimal levels just by adopting two "boosting" strategies: multiple index search and multiple query search, which both have nice parallelization properties.

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 makes possible to distribute the search process in order to further improve its efficiency.

The implementation of the multiple index strategy we have adopted in this work consists of building each index independently of the others, with its own data storage, thus replicating $t$ times on disk each data block. An alternative implementation could have one main index with its own data storage, and other

$t-1$ indexes with data storages just containing pointers to the main data storage. The first solution is designed to preserve the maximum effectiveness while the second solution trades disk space for efficiency. If implemented naively, the second solution could result in an uncontrolled number of random accesses to the data storage. In future work we will investigate the possibility to build a "smart" plan of accesses to the data storage once all pointers, resulting from searching the $t$ prefix trees, have been collected, in order to reduce the sparseness of accesses.

The comparison with experimental results published for some related methods, which are among the top-performers on the task, shows that the PP-Index compares well against them, specially in the ability of accessing a very small part of the entire collection to produce almost exact results.

We have compared the PP-Index with the LSH-Forest on the entire CoPhIR data set, showing how the locality information encoded into the PP-Index prefixes results in the selection of a set of candidates that are more related to the query, as reflected by the improvement in the RDE values.

Only one (Batko et al., 2008a) of the previous works we compare with used in the original version a data set of a size comparable to our largest one. We plan to extend the comparison with the other competing methods (e.g., Novak and Batko (2009)) which have shown to scale to very large data set sizes.

The PP-Index has been already used to build a performing similarity search system[12] (Esuli, 2009a). By mining the query log of the service we plan to investigate alternative policies to the random selection for the definition of the $R$ set, e.g., clustering queries and selecting the most representative ones of each cluster.

Among the many other possible directions of investigation we mention the formulation of alternative multiple query generation methods, and extending the experiments to different types of content, e.g., textual data.

## 6. Acknowledgments

I would like to thank Giuseppe Amato, for the useful discussions on the topic, Fabrizio Falchi, for the support on the MPEG-7-related issues, and Fabrizio Sebastiani, for his support to this research activity.

## 7. References

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

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

Batko, M., Kohoutkova, P., Zezula, P., 2008b. Combining metric features in large collections. In: Proceedings of the First International Workshop on Similarity Search and Applications. SISAP '08. Cancún, Mexico, pp. 79–86.

Bawa, M., Condie, T., Ganesan, P., 2005. LSH Forest: self-tuning indexes for similarity search. In: Proceedings of the 14th international conference on World Wide Web. WWW '05. Chiba, Japan, pp. 651–660.

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

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

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

Datar, M., Immorlica, N., Indyk, P., Mirrokni, V. S., 2004. Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on Computational geometry. SCG '04. ACM, New York, NY, USA, pp. 253–262.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Diaconis, P., 1988. Group representation in probability and statistics. IMS Lecture Series 11.

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

Esuli, A., 2009b. 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-IR '09. pp. 17–24.

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

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

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

Knuth, D., 1998. The Art of Computer Programming, 2nd Edition. Vol. 3: Sorting and Searching. Ch. 5.4: External Sorting, pp. 248–379.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K., 2007. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd International Conference Very Large Data Bases. VLDB '07. Vienna, Austria, pp. 950–961.

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

Novak, D., Batko, M., 2009. Metric index: An efficient and scalable solution for similarity search. In: Proceedings of the Second International Workshop on Similarity Search and Applications. SISAP '09. IEEE Computer Society, Washington, DC, USA, pp. 65–73.

Novak, D., Kyselak, M., Zezula, P., 2010. On locality-sensitive indexing in generic metric spaces. In: Proceedings of the Third International Conference on Similarity Search and Applications. SISAP '10. ACM, New York, NY, USA, pp. 59–66.

Patella, M., Ciaccia, P., 2009. Approximate similarity search: A multi-faceted problem. J. Discrete Algorithms 7 (1), 36–48.

Skala, M., 2009. Counting distance permutations. J. Discrete Algorithms 7 (1), 49–61.

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