Succinct Nearest Neighbor Search [☆]

Eric Sadit Tellez[a], Edgar Chavez[b], Gonzalo Navarro[c,1]

[a]*División de Estudios de Posgrado, Facultad de Ingeniería Eléctrica, Universidad Michoacana de San Nicolas de Hidalgo, Mexico.*
[b]*Facultad de Ciencias Físico-Matemáticas, Universidad Michoacana de San Nicolas de Hidalgo, Mexico.*
[c]*Departament of Computer Science, University of Chile, Chile.*

Abstract

An efficient and universal similarity search solution is a holy grail for multimedia information retrieval. Most similarity indexes work by mapping the original multimedia objects into simpler representations, which are then searched by proximity using a suitable distance function.

In this paper we introduce a novel representation of the objects as sets of integers, with a distance that is computed using set operations. This allows us to use compressed inverted indexes, which have become a mature technology that excels in this type of search. Such indexes allow processing queries in main memory even for very large databases, so that the disk is accessed only to verify a few candidates and present the results.

We show that our technique achieves very good speed/compression/recall tradeoffs. As an example, to reach 92% recall in 30-nearest neighbor searches, an index using 1 to 2.5 bytes per object inspects less than 0.6% of the actual objects. Furthermore, the ratio between the distances to the actual nearest neighbors and those reported by our algorithm is very close to 1.

[☆]A preliminary version of this article appeared in *Proc. SISAP 2011.*

Email addresses: `sadit@lsc.fie.umich.mx` (Eric Sadit Tellez), `elchavez@fismat.fie.umich.mx` (Edgar Chavez), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro)

[1]Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

1. Introduction

A metric space is a pair (U, d) with U a universe of objects and $d(\cdot, \cdot)$ a distance function $d : U \times U \to \Re$ holding the metric axioms: For all $x, y, z \in U$, $d(x, y) \geq 0$ and $d(x, y) = 0 \iff x = y$, $d(x, y) = d(y, x)$, and $d(x, z) + d(z, y) \geq d(x, y)$. These properties are known as strict positiveness, symmetry, and the triangle inequality, respectively. A finite subset of the metric space is the database $S \subseteq U$. Problems of diverse domains, from pattern recognition, classification and statistics, to multimedia retrieval, can be formulated with the abstract framework described above. Two queries of most interest are *range* queries $(q, r)_S$, defined as the objects of S at distance at most r from q, and ℓ-Nearest-Neighbor queries ℓNN_S, defined as the ℓ elements of S closest to q. The second type of queries have been argued to be more meaningful for multimedia information retrieval applications, as ℓ is a more intuitive measure for a user unfamiliar with the precise distance function used.

We focus on ℓNN queries in this paper. These have a well-known linear worst case [7, 21, 12, 4] when the database has high intrinsic dimensionality, that is, a concentrated histogram of distances. Under this circumstance, all the traditional techniques suffer from a condition known as the *curse of dimensionality* (*CoD*). This is a serious problem in many multimedia retrieval applications. An alternative to overcome this limitation is to relax the condition of finding the exact ℓNN and use *approximate* techniques. It has been shown that the query time can be dramatically reduced by only slightly relaxing the precision of the solution [16]. Tight approximate solutions are usually as satisfactory as exact ones in multimedia retrieval, since the distance model is already a simplification of a vague retrieval requirement. Furthermore, approximate methods usually work well on similarity spaces where not all the metric axioms hold.

A general approach to solve proximity search problems is to map the objects in the original space onto a simpler data space. While ad-hoc mappings can be defined for specific applications (e.g., feature vectors for images) we are interested in universal mappings that can be defined for arbitrary object types.

For example, *pivot based methods* map objects into coordinate spaces using distances to selected objects (the pivots) [7]. Recently, as described in Section 2, various successful approximate searching methods build on weaker mappings where only the distances to some pivots (usually the closest) are chosen, or only the order of those distances is used. Proximity is hinted by similarity of distances to the chosen pivots or of the orderings.

In this work we push the idea even further by considering only the subset of pivots. Each object in the database will be represented by the set of its nearest pivots, among a relatively large set to choose from, and proximity is hinted by the number of shared pivots. This is at the same time simpler and more efficiently computed than previous measures. The efficiency of the approach comes from using an inverted index to find similar sets of pivots, which is a technology from textual information retrieval [2]. At this respect, we take a further step by representing the inverted index in compressed form.

Our contribution is twofold. First, we describe the *neighborhood approximation* (NAPP), a simple mapping from general similarity spaces to a set of integers (with a proximity semantics). NAPP is an approximate technique with high recall and high quality. The second contribution is to represent the NAPP mapped database in a compact way, giving rise to the *compressed NAPP inverted index*. The combination of these two novelties yields a data structure supporting very fast proximity searches, using little storage space, with high recall and high quality in the answer.

The rest of the paper is organized as follows. Section 2 is a brief review of related work. Section 3 introduces the mapping transformation and Section 4 the underlying data structure for our index. The compression of the index is described in Section 5. Experimental results showing search time, memory usage and recall of our index in real databases are shown in Section 6. Conclusions and future work are discussed in Section 7.

2. Related Work

Exact searching methods are divided into two broad categories, as follows [7, 21, 12, 4].

Compact partition indexes define a partition of the space with purported high locality. One can, for example, use a set of objects (called centers) with a covering radius r_c. Any item u where $d(u, p) \leq r_c$ is represented by the center p. The scheme can be repeated recursively inside each ball using smaller radii at each level.

Pivot based indexes use a set of distinguished objects, called pivots, to compute an implicit contractive mapping with a distance easier to compute than the original $d(\cdot, \cdot)$. Most of the dataset is filtered out on the mapped space, and only a few candidates must be compared to the query on the original space. A pivot based index using sufficient pivots surpasses the performance of a compact partition index, but the necessary number of pivots is too large for high dimensional databases.

Both schemes, however, are ineffective on high dimensional spaces, due to the CoD. This is intrinsic to the problem rather than a limitation of the known data structures [17, 18]. Unfortunately enough, the CoD arises in many applications of interest, including multimedia information retrieval.

To speed up searches in those cases, we can drop the condition of retrieving all the objects relevant to the query, and be satisfied with a good approximation. The quality of the approximation can be measured in several ways. For ℓNN queries, some measures are the *precision* (how many of the reported answers are among the ℓ true nearest neighbors), the *recall* (how many of the true ℓ nearest neighbors are reported), and the ratio between the distances towards the reported objects and those towards the ℓ true nearest neighbors). Note that precision and recall are complementary measures: one usually increases one by decreasing the other. In information retrieval scenarios with a small number of answers (as in ℓNN search), recall is more important because the user has little chance of being aware of relevant results that have been lost, whereas irrelevant results included are evident by examining the set of answers.

A recent survey covers many of the techniques to trade speed for accuracy in approximate searches [16]. Below we review some techniques related to our approach.

2.1. The Permutation Index

An index called the *Permutation Index* (PI) [6] has high precision and recall even for datasets of high intrinsic dimension, and can also be used in more general similarity spaces. The PI drastically reduces the number of candidate objects to be compared directly to the query.

The idea behind the PI is to represent each database object with the permutation of a set of pivots, called the *permutants*, sorted by distance to the object. The distance between objects is hinted by the distance between their respective permutations. More formally, let $R \subseteq U$. For each $u \in S$, we sort R by distance to u, obtaining a permutation π_u of size $\sigma = |R|$. The distance between permutations π_u, π_q is computed using π_u^{-1} and π_q^{-1} treated as vectors, with Minkowski's L_1 or L_2 distances.

Since σ is small, the PI is especially useful for expensive distances. All the permutations of S can be represented with $n\sigma \log \sigma$ bits. The search is carried out by computing n permutation distances plus γ metric space distances, where γ is a parameter controlling the number of candidates to be verified with the distance function. This parameter is used in our index too.

The good precision and recall of the PI is paid with a large number of L_1 or L_2 distances computed among permutations. This cost dominates the savings on metric distance computations, unless the distance function is very expensive. Some authors have tried to reindex the permutation space [10], but it turns out to be of medium to high dimensionality. As a result, the scalability of the technique is limited.

2.2. The Metric Inverted Index

Amato et al. [1] gave a nice algorithmic way to simplify the distance computation between permutations in the *Metric Inverted Index*. For each object, only $K \ll \sigma$ closest permutants are stored. The structure is an inverted index storing, for each permutant p, a posting list with pairs $(objId, pos)$ sorted by *objId*, so that p is among the K permutants closest to *objId*, precisely the *pos*-th one.

The permutation π_q is computed at query time. The permutation π_u, for any $u \in S$, is partially known because at most K references are stored in the inverted file. For the missing references, a constant penalty ω is added to compute the Spearman Footrule distance; one possible choice is $\omega = \frac{\sigma}{2}$. There are two choices for the permutation reconstruction: to take the union or the intersection of the posting lists of the permutants closest to q. The union produces better quality results, while intersection yields faster responses. This index uses at least $nK \log(nK)$ bits for the posting lists.

2.3. The Brief Permutation Index

Another algorithmic solution to compute an approximate distance between permutations is the *Brief Permutation Index* [23]. The main idea is to encode the permutation vectors using fixed-size bit-strings and compare them using the binary Hamming distance. This produces a smaller representation that can be filtered out faster than the original permutations space. Nevertheless the set of candidate objects after filtering with the brief version of the permutations is quite large and this is relevant for high-complexity distances. The advantage against the original algorithm is the reduction of some extra CPU cost and smaller requirements of memory, yielding faster searches for large databases.

The resulting Hamming space encodes each permutant with a single bit using the information about how much it deviates from the original position in the identity permutation. If the permutant is displaced by more than m positions (which is a parameter) the corresponding bit is set to 1, else it is set to 0. The number of bits then matches the number of permutants. A fair choice for m is $\frac{\sigma}{2}$. One observation is that the central positions are assigned mostly 0's because the central permutants have less room for displacement. This is solved using an inline central permutation [23].

Even if computing the Hamming distance is cheaper than computing L_2, a sequential scan can be time-consuming for large databases. The same authors presented later a version indexed with Locality Sensitive Hashing [22]. Unfortunately, the recall dropped as the speed increased.

2.4. The Prefix Permutation Index

The last approach using the permutations idea is the PP-Index [9]. It stores only the prefixes of the permutations and hints the proximity between two objects with the length of their shared prefix (if any). Longer shared prefixes hint higher proximity and shorter length prefixes reflect lower proximity. This strict notion of proximity yields very low recall, allowing only small γ values. This condition is somewhat alleviated by using several permutation sets, several indexes, and tricks like randomly perturbing the query, which end up increasing the number of queries to the index and affecting the main advantage of search speed. The index consists in a compact trie [2] representing the space of permutation prefixes. A plain representation needs $Kn \log \sigma$ bits. The compact trie is usually smaller, and the storage usage depends on the amount of shared prefixes. The first levels in the trie are stored in main memory and the lower levels in secondary memory.

In a concrete example, the PP-Index needs up to eight indexes to achieve perfect recall on the 106 million MPEG7 vectors of the CoPhIR dataset [5].

3. Neighborhood Approximation

All the approaches described in the previous section are variants of the idea of using permutations as object proxies. We will reuse some notation and introduce a new formulation of the technique that is simpler and more powerful.

We call our approach *Neighborhood approximation* (NAPP). In a way NAPP is a generalization of the permutation idea and we believe it captures the features responsible of the high recall of the permutations, while simultaneously allowing a compact representation and fast searching. We will reuse the notation of R, γ, and σ.

3.1. The Core Idea

We choose a set R of *references* from the database, with $|R| = \sigma \ll n$. Each object $u \in U$ is represented by a set of references, called the *neighborhood* of u, defined as $P_u = KNN_R(u)$ (that is, the K closest neighbors of u in R) for a fixed parameter K to be discussed later. We assume P_u to be an ordered set. The default order will be the proximity to u.

Note that sets of references will act as database object proxies, just as permutations do in permutation based indexes. We follow the same path: every object in the database is transformed into its neighborhood representation (a set of references) and to satisfy a query q we first compute its representation P_q. As in other indexes, we will obtain a set of candidate objects in S that need to be checked against the query. The list of candidates in NAPP will be the objects u such that $P_u \cap P_q \neq \emptyset$.

3.2. Retrieval Quality Considerations

Consider two objects $u \in S$ and $q \in U$, and their respective neighborhoods $P_q, P_u \subseteq R$ (see Figure 1). The next two observations follow immediately from the triangle inequality.

Observation 1. *Let $M = P_q \cap P_u$. If $M \neq \emptyset$ then the distance $d(q, u)$ is lower and upper bounded as follows:*

$$\max_{v \in M} |d(q,v) - d(u,v)| \leq d(q,u) \leq \min_{v \in M} d(q,v) + d(v,u)$$

Observation 2. *If $R \subseteq S$ and q^* denotes the nearest neighbor of q in P_q, then $d(q, 1NN_S(q)) \leq d(q, q^*) = d(q, 1NN_R(q))$.*

The upper and lower bounds depend on the selection of R. If R is dense enough then the bounds above become tighter. A rule of thumb is to have as many references as one can handle without slowing down the process of comparing q with the set of references. A final remark is that R should have the same distribution of S, so we should select the references uniformly at random from the database. This is a core heuristic in our approximate index. Figure 1 shows a bad case on the left, and a more realistic case on the right.

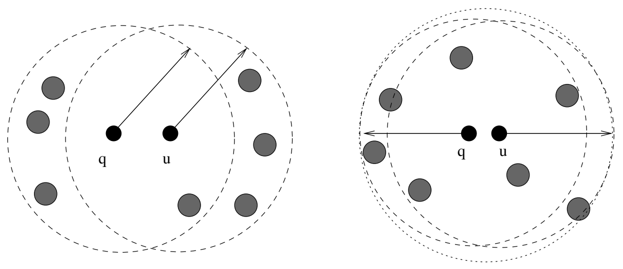


Figure 1: An example showing shared references as proximity predictor. Gray balls are references, dark balls are objects in S.

4. The NAPP Inverted Index

Proximity in the NAPP framework is hinted by the number of shared references, that is, the size of the intersection of two sets. A natural representation to compute this number is an inverted index.

Each element of S and each element of R will be denoted by an integer. Actual objects may reside somewhere else, for example on disk. We use integers as object identifiers, so we abuse notation and call $R = \{1, \cdots, \sigma\}$ and $S = \{1, \cdots, n\}$; it will be clear from context which collection an index i refers to.

We define a list for each reference r, $L[r] = \{s_1, s_2, \cdots, s_k\} \subseteq S$ such that $r \in P_{s_i}$. In other words, $L[r]$ is the list of all elements having reference r among their K nearest neighbors. Algorithm 1 gives the construction.

Algorithm 1 Construction of the NAPP inverted index

```
1: $R$ is the set of references of size $|R| = \sigma$.
2: Let $L[1, \sigma]$ be an array of sorted lists of integers, initially empty.
3: Let $S = \{1, \cdots, n\}$
4: for $i$ from 1 to $n$ do
5:     Compute $P_i[1, K]$, the $K$ nearest neighbors of $i$ in $R$
6:     for $j$ from 1 to $K$ do
7:         $L[P_i[j]] \leftarrow L[P_i[j]] \cup \{i\}$
8:     end for
9: end for
```

Experimentally, we have observed that many objects with a small intersection cardinality (1 or 2) appear in the candidate lists, and very frequently they are not close to the query. It is then natural to impose an additional condition about the minimum size of the intersection, say at least t shared references. The search for the candidates using this restriction is implemented using the *t-threshold* algorithm, a generalization of the set union/intersection problem of K sets, where the solution is a collection of objects appearing in at least t sets. Setting $t = 1$ is equivalent to the set union and $t = K$ is equivalent to the set intersection. We adapted the Barbay-Kenyon algorithm [3] to obtain the candidate list. This is described in Algorithm 2.

To represent R we need $\sigma \log n$ bits. The storage requirements of the plain mapping is $Kn \log \sigma$ bits. Using the inverted index the space cost increases to $Kn \log n$ bits, that is, a total of Kn integers of $\log n$ bits, distributed among the σ sorted lists.

5. Compressing the Inverted Index

The space of our index is Kn integers. For a value like $K = 7$, which is adequate in terms of retrieval quality, this space overhead is larger than that required by other typical data structures for similarity search. In this section we show how the posting lists can be compressed using compact representations for sparse bitmaps, with a very small speed penalty for set union and intersection computations [15, 11, 20].

We must handle σ lists. The s items of a list are distributed in the range $[1, \ldots, n]$, then ideally we can represent that list using $\log \binom{n}{s} = s \log \frac{n}{s} + O(s)$ bits. Using the *sarray* sparse bitmap representation [15] we can represent such an inverted list using $s \log \frac{n}{s} + 2s + o(s)$ bits. As all the s items add up to Kn items overall, the worst case arises when $s = Kn/\sigma$ for each list, where the complete index takes $Kn \log \frac{\sigma}{K} + 2Kn + o(Kn)$ bits. The *sarray* supports constant access to every position of every list.

Algorithm 2 Solve an ℓNN query in the inverted index

```
1: Compute $P_q[1, K] = KNN_R(q)$ by brute force
2: Let $Q[1, K] = L[P_q[1]], \ldots, L[P_q[K]]$ be the lists of references to consider
3: Let $POS[1, K] = 1, \ldots, 1$ be pointers to the current position of the lists in $Q$
4: Let $CND$ be a priority queue to store the set of candidates
5: while $Q.Length \geq t$ do
6:    Ascending sort $Q$ using $Q[i][POS[i]]$ as key for $1 \leq i \leq Q.Length$
7:    if $Q[1][POS[1]] \neq Q[t][POS[t]]$ then
8:       Advance all $POS[i]$ for $1 \leq i < t$ until it holds $Q[i][POS[i]] \geq Q[t][POS[t]]$
9:       Restart the loop
10:   end if
11:   Find the greatest $k \geq t$ such that $Q[k][POS[k]] = Q[t][POS[t]]$
12:   if $k = Q.Length$ then
13:      Increment all $POS[i] \leftarrow POS[i] + 1$ for $1 \leq i \leq Q.Length$
14:   else
15:      Advance all $POS[i]$ for $1 \leq i \leq k$ until $Q[i][POS[i]] \geq Q[k+1][POS[k+1]]$
16:   end if
17:   Append $Q[t][POS[t]]$ to $CND$ with priority $k$ (the intersection size)
18:   Evaluate the distance between $q$ and the $\gamma$ highest-priority candidates in $CND$
19:   Return the $\ell$ closest objects to $q$
20: end while
```

Note 1: Increasing and advancing in POS and Q may reach the end of some lists, in which case the entry must be removed from both POS and Q. This is why we use $Q.Length$ instead K.

Note 2: *Advance* means searching for the desired key in the list. In particular we use doubling search [13] since it makes the algorithm of Barbay-Kenyon instance-optimal in the comparison model [3].

5.1. Inducing Runs in the Index

The plain representation and the *sarray* encoding are enough to host medium to large databases in main memory in a standard desktop computer. In particular, when using the *sarray* the index is compressed to its zero-order entropy and the extension to secondary memory is straightforward.

On the other hand, to handle very large databases or when using devices with scarce memory (such as a mobile phone), better compression is needed. The additional compression is obtained by renumbering objects (represented by integers) so as to obtain proximal integers in the inverted lists $L[r]$. This is done by observing that objects in any given inverted list $L[r]$ share at least the reference r, and hence cannot be much far apart from each other, as described in Observation 1.

After computing $P_u = KNN_R(u)$ for each object $u \in S$, the reordering procedure starts by sorting P_u in increasing order of the reference identifiers, and not by proximity to u. Second, the database is lexicographically sorted by the P_u sequences, using a linear sort for the first levels and a threeway quicksort for deeper levels, similarly to practical suffix array sorting algorithms [19]. Third, the permutation of S induced by this sorting is used to renumber the database S. Finally, the inverted index is created using the permuted mapping. Figure 2 illustrates the process.

The first step creates, inside the inverted lists, ranges of consecutive integers such that the i-th integer plus 1 is equal to the $(i+1)$-th integer. These regions are named *runs*, and are suitable to be encoded with a run-length scheme. For ranges not in a run we aim at having small differences between consecutive elements. Although *sarray* does not profit from runs and small differences, we can reduce space significantly by using an alternative encoding.

Differential encoding. An inverted list encoded with differences is just the list of differences between consecutive entries, as shown in the example of Figure 2. Each difference is encoded using Elias-γ [8] which is a variable-length integer encoding using $2\log x + 1$ bits to encode an integer x. It uses less space than fixed-length binary encoding (which uses $\log n$ bits) if $x \leq \sqrt{n/2}$. We have s integers in the range 1 to n. In the worst case each difference is n/s and we need twice the optimal number of bits, $2s \log \frac{n}{s} + 2s$. This worst case is unlikely, however, as we have induced runs.

Our search algorithm (2) uses doubling search to *advance*, so we need access to the i-th element. In order to parameterize time and storage requirements we add absolute samplings each B entries. Now an access to any item in a list needs to decode at most B Elias-γ codes. There are at most $\frac{Kn}{B}$ samples overall.

The inverted list is represented by the differences. If s is the size of a list then we need $\sum_{i=1}^{s} 2\log(docId_i - docId_{i-1} + 1)$ bits[2] to represent the list using Elias-γ without samples. These variable-length representations are concatenated in a memory area L', and a pointer to L' is set every B positions of the original list L. If these pointers are represented using a *sarray*, we need $(s/B)\log(|L'|/(s/B)) + 2s/B + o(s/B)$ bits for them. In the worst case $s = \frac{Kn}{\sigma}$ and $|L'| = 2s \log \frac{n}{s} + 2s$, and the space for the samples adds up to $(Kn/B)(\log(B(\log \frac{\sigma}{K} + 1)) + O(1))$.

Let us choose $B = \Theta(\log \frac{Kn}{\sigma})$, so accessing any integer costs $O(\log \frac{Kn}{\sigma})$ time. Then the space for the samples becomes $O(Kn \log \frac{\sigma}{K} / \log \frac{Kn}{\sigma}) + o(Kn)$ bits. If $\log \frac{\sigma}{K} = o(\log n)$, the time is $O(\log n)$ and the space is $o(Kn)$ bits.

Differential + Run-Length encoding. The differential encoding of the inverted index is suboptimal when long runs arise, as their lengths are essentially represented in unary. A better option is to encode the run lengths using Elias-γ codes. As in the differential encoding, we use regular samplings to get fast access to the i-th integer. Figure 2 shows an example of run-length encoding of the inverted index, where only differences of 1's are run-length encoded as a tuple (1, length). Since we always decode from left to right it is simple to mix differences with run-length encodings. If an absolute sample falls inside a sample, the run is cut. This is suboptimal in space, but allows decompression without binary searching to locate the actual position.

[2] We define $docId_0 = 0$.

A natural optimization for long runs is introduced as follows: if the j-th and the $(j+1)$-th absolute samples belong to the same run we say that the range is *filled*, and no representation of the data inside the range is necessary; just the samples are stored.

Notice that if the i-th integer lies in a filled range, it is decoded in constant time. Similarly, even when the worst case requires B decompressions of items, the average time is way smaller because when we find a run we advance many positions in constant time.

Finally, note that our lexicographic sorting by P_u ensures that some runs will arise: all the objects sharing the first element in P_u have received contiguous identifiers. As a result, the inverted list of each $r \in R$ contains at least one run, and the sum of all those runs is n. Therefore, run-length encoding saves us at least n bits, which are compressed to at most $O(\sigma \log \frac{n}{\sigma})$ bits.

6. Experimental Results

We give experimental results for four representative spaces.

Documents. A collection of $25,157$ short news articles in the TFIDF format from Wall Street Journal $1987-1989$ files from TREC-3 collection. We use the angle between vectors as the distance measure, which is equivalent to the popular cosine similarity measure. The dataset is available at the SISAP library (`http://www.sisap.org`). We extracted 100 random documents from the collection as queries; these documents were not indexed. Each query searches for 30*NN*. TFIDF documents are vectors of thousands of coordinates. We choose this space because of its high intrinsic dimensionality; the histogram of distances is shown in Figure 3(a). As a reference, a sequential scan comparing the query with every object needs 0.23 seconds in our machine.

CoPhIR MPEG7 Vectors. A subset of 10 million 208-dimensional vectors from the CoPhIR database [5]. We use the L_1 distance. Vectors are a linear combination of five different MPEG7 features [5]. The first 200 vectors from the

Id	P_u	
	Orig.	Num. Sort
1	312	123
2	321	123
3	123	123
4	421	124
5	521	125
6	431	134
7	513	135
8	531	135
9	154	145
10	541	145
11	514	145
12	145	145
13	235	235
14	532	235
15	423	245
16	245	245
17	254	245
18	542	245
19	345	345
20	354	345
21	543	345

Inverted index

```
1 -> 1,2,3,4,5,6,7,8,9,10,11,12
2 -> 1,2,3,4,5,13,14,15,16,17,18
3 -> 1,2,3,6,7,8,13,14
4 -> 4,6,9,10,11,12,15,16,17,18,
     19,20,21
5 -> 7,8,9,10,11,12,13,14,15,16,
     17,18,19,20,21
```

Inverted index with differences

```
1 -> 1,1,1,1,1,1,1,1,1,1,1,1
2 -> 1,1,1,1,1,8,1,1,1,1,1
3 -> 1,1,1,3,1,1,5,1
4 -> 4,2,3,1,1,1,3,1,1,1,1,1,1
5 -> 7,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Inverted index with differences + Run-Length

```
1 -> (1,12)
2 -> (1,5),8,(1,5)
3 -> (1,3),3,(1,2),(1,1)
4 -> 4,2,3,(1,3),3,(1,6)
5 -> 7,(1,14)
```

Figure 2: Example of the induction of runs for plain, differences and run-length encoding of lists. Here $\sigma = 5$, $n = 21$.

database are queries; each query searches for $30NN$. Figure 3(b) shows the histogram of distances. A sequential scan takes 47.3 seconds on our machine.

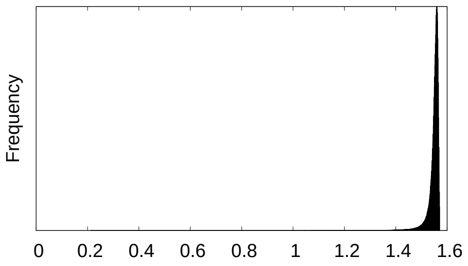
Color histograms. A set of $112,544$ color histograms (112-dimensional vectors) from an image database from the SISAP testbed. We chose 200 histogram vectors at random and applied a perturbation of ± 0.5 on one random coordinate for each of them. The searches look for $30NN$ under distance L_2. Figure 3(c) shows the histogram of distances. A sequential scan requires 0.27 seconds on our testing hardware.

Synthetic datasets. In order to explore the behavior of the compressed NAPP inverted index parametrized by the dimensionality of the datasets, we generate six vector databases of 1 million objects each, and six different dimensionalities, with L_2 distance. Each vector was generated randomly in the unitary hypercube. More details will given in Section 6.3.

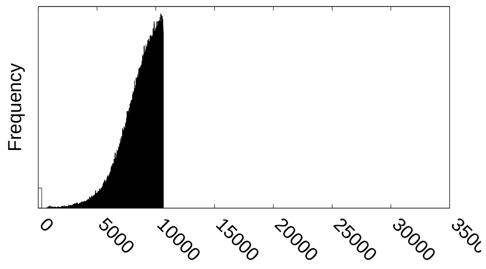
We use 30 nearest neighbors because it is a common value as an output size in both textual and multimedia information retrieval systems.

Implementation notes and test conditions. All the algorithms were written in C# , with the Mono framework (`http://www.mono-project.org`). Algorithms and indexes are available as open source software in the *natix* project (`http://www.natix.org`). The experiments were executed on a 16 core Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running CentOS Linux. The entire database and indexes were maintained in main memory and without exploiting any parallel capabilities of the workstation.

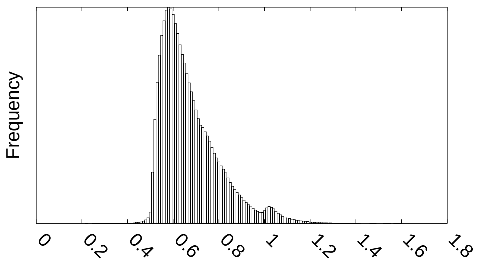
All our experiments were performed fixing $K = 7$ and with several σ values. The selection of K affects the required space, the search time, and the quality of the answer. We observed experimentally that $K = 7$ is a good tradeoff between space, time and recall. The support of this choice is based on the assumption that the histograms of distances seen from random objects are similar. Thus, fixing K to a constant will fix the covering radii of the references to a constant value, equivalent to a constant percentile of objects being covered. This



(a) Document space with angle distance.



(b) CoPhIR space with L_1 distance.



(c) Color histogram with L_2 distance.

Figure 3: Histograms of our real-life test databases using the chosen query sets.

simplification reduces the complexity of fine tuning the parameters of the index. Experimental results validating this choice are presented in previous work [9, 1, 24]. The particular value of $K = 7$ is not optimal for all datasets, but for simplicity we used this fixed value, which in particular shows the robustness of the methods. Moreover, using this fixed value enhances the recall quality of some of the state of the art methods, yet it affects both their time and space performances.

6.1. General performance

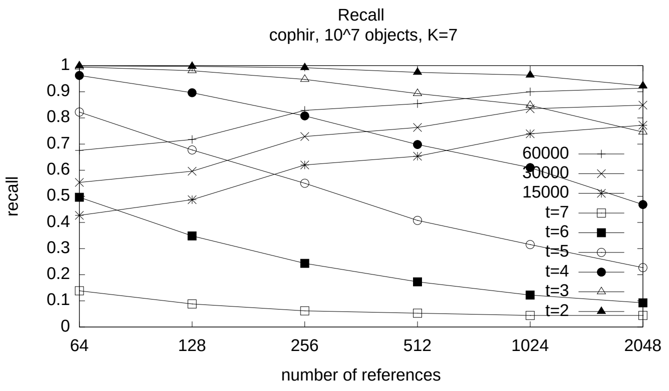
In this section we analyze the recall, total time, and the percentage of the reviewed database (i.e., objects directly compared to the query) in the CoPhIR database. Experimental results are shown for two types of queries: t-threshold queries (for $t = 2$ to 7), and 1-threshold with a fixed number of compared objects (60,000; 30,000; and 15,000).

Our primary quality measure is the recall. Since our queries are $30NN$, the recall is just the number of true $30NN$ elements returned, divided by 30. This measure ignores how close the non-relevant objects are from the true $30NN$. In the next section we discuss this point.

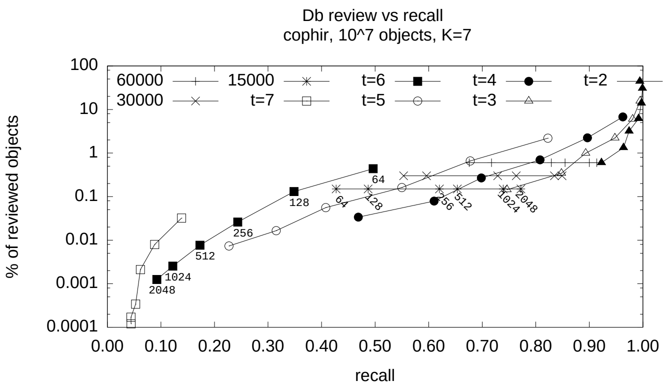
Figure 4(a) shows how the recall evolves with the number of references. Methods based on t-threshold show a decreasing recall as a function of t: a smaller t gives better recall. Smaller σ values also give better recall, but at the cost of more distance computations and time, see Figures 4(b) and 4(c), respectively. Notice that, in both figures, the points in each curve are produced by indexes with different σ values. Then, for $t > 1$, σ decreases as the recall increases, whereas for $t = 1$ it is the opposite. We put labels with the σ values on selected points of the curves to increase readability.

Larger σ values yield faster indexes. The speedup is produced because the Kn objects are split into more inverted lists. We note that the distribution of lengths (of inverted lists) is not Zipfian as it is in text inverted indexes for natural languages [2].

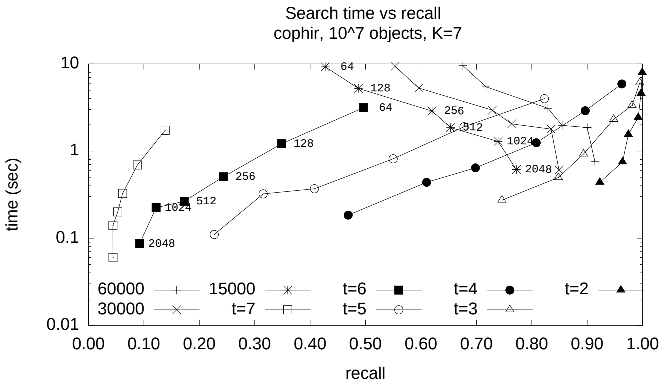
All these parameters induce tradeoffs that can be used to effectively tune



(a) Recall.



(b) Percentage of the database compared to the query.



(c) Search times.

Figure 4: CoPhIR recall and performance

real applications. For example, for $t = 2$ and $\sigma = 2048$, the index achieves a recall of 0.92, comparing 0.6% of the database in about 0.4 seconds.

Larger t values produce faster searches, since the algorithm skips parts of the input lists, due to *advance* commands in Algorithm 2. Fixing γ, the number of elements to be verified, restricts the percentage of verified elements of the database and hence bounds the total time. See lines "15000", "30000" and "60000" of Figure 4. In this case $t = 1$ and the t-threshold algorithm is equivalent to set *union* (being linear in the number of items in the input lists). Notice that, under this configuration, the performance is dominated by the operations on *CND*, the priority queue of Algorithm 2. Based on Figure 4(c), this strategy (lines named "15000", "30000" and "60000") is useful to control the search time, yet it needs to compute the entire set *union*.

Naturally, a hybrid configuration achieves better control of the performance and quality, that is, the combination of t-threshold and fixed γ. For example, for $\sigma \geq 1024$, pure t-threshold configurations yield better times than just fixing the cardinality, see Figure 4(c). The opposite holds for $\sigma < 1024$.

Comparison with previous work. We compared our work with four indexes using the permutations approach [6, 9, 1, 23], described in Section 2. Since we do not have the actual implementations of all the indexes being compared, we fix our attention on the recall, disregarding the time, and also fixing the number of references. A thorough comparison is still needed to have a fair overview of the tradeoff between speed, recall and space usage for all this indexes. The PP-index [9] was run as a single instance, and without query expansion.

Figure 5(a) shows the recall ratios for the CoPhIR dataset. Here the permutation index is very effective, achieving perfect recall even for small σ. Unfortunately, it is very slow since it cannot be indexed. The brief index follows the same behavior, but it is slower. The metric inverted index and the NAPP inverted index improve the performance as σ grows. The PP-Index remains unaffected for small σ, but it is affected negatively for $\sigma > 512$.

For the document and color spaces, Figures 5(b) and 5(c), respectively, show similar behavior. The permutation index rapidly reaches a steady recall value for colors, while the recall grows slowly as σ increases for documents. The metric inverted file improves as σ increases. Finally, the NAPP inverted index displays the sharpest increase on the recall as σ grows.

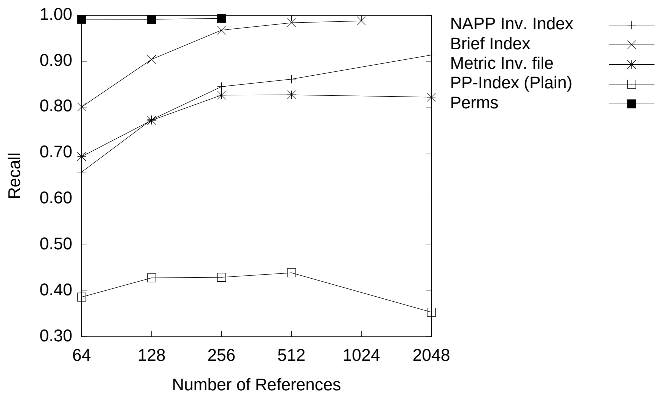
The recall proportional to σ is a very interesting property since R can be easily maintained in memory, while the entire database can reside on disk. Note also that, in the comparison, we are considering the same number of references for the distinct methods, but these may require distinct space per reference. The NAPP index requires the least information per reference (no ordering and no distance values), so taking the number of references as a uniform measure is conservative. We explore further our space usage in Section 6.2.

Proximity ratio as a measure of retrieval quality. We have used the recall as a measure of quality, that is, we have penalized only the false negatives. In multimedia information retrieval applications, especially when some relevance feedback is expected from the user, we want to measure how close the reported *non-relevant* objects (the false positives) are from the relevant ones. To this end we show statistics of the ratio between the covering radius of the 30-th nearest neighbor and the distance given by NAPP in Table 1. Note that large σ values yield results that are very close to the real answers. This supports Observation 1, which bounds the distance to the query, not the recall.

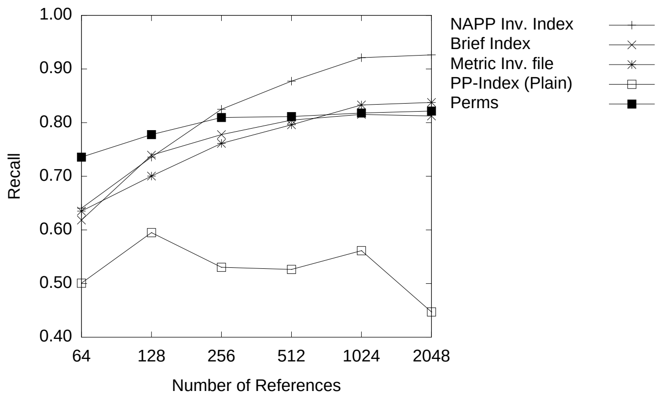
The same statistics are given for the database of documents and colors, respectively, in Tables 2 and 3. Note that these databases have worse ratio, probably because of their higher intrinsic dimensionality (recall Figure 3). Still, in all the experiments, our false positives are nevertheless very close to the exact results.

6.2. The Compressed NAPP Inverted Index

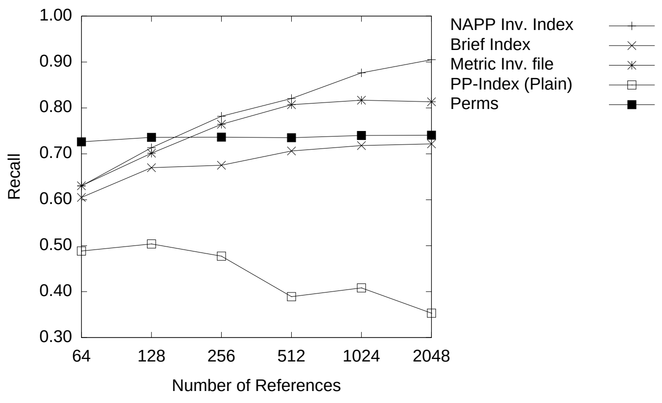
The plain inverted NAPP index uses a fixed number of bits in the inverted lists. For example, using 32-bit integers, the index for CoPhIR uses 267 megabytes, that is, 224 bits per database object. Our compressed representation uses from 10 to 80 bits per object for CoPhIR, and 15 to 80 bits in



(a) CoPhIR, reviewing at most 60,000 candidates.



(b) Documents, reviewing at most 1,000 candidates.



(c) Color histograms, reviewing at most 3,000 candidates.

Figure 5: Recall behavior of the NAPP index and previous work.

σ	γ	max-ratio			
		mean	stdev	min	max
64	15000	1.06	0.04	1.00	1.25
128	15000	1.05	0.03	1.00	1.25
256	15000	1.03	0.03	1.00	1.27
512	15000	1.02	0.02	1.00	1.12
1024	15000	1.02	0.01	1.00	1.06
2048	15000	1.01	0.01	1.00	1.10
64	30000	1.04	0.03	1.00	1.19
128	30000	1.03	0.02	1.00	1.16
256	30000	1.02	0.02	1.00	1.26
512	30000	1.01	0.01	1.00	1.11
1024	30000	1.01	0.01	1.00	1.04
2048	30000	1.01	0.01	1.00	1.07
64	60000	1.02	0.02	1.00	1.12
128	60000	1.02	0.02	1.00	1.09
256	60000	1.01	0.02	1.00	1.25
512	60000	1.01	0.01	1.00	1.07
1024	60000	1.01	0.01	1.00	1.04
2048	60000	1.00	0.01	1.00	1.06

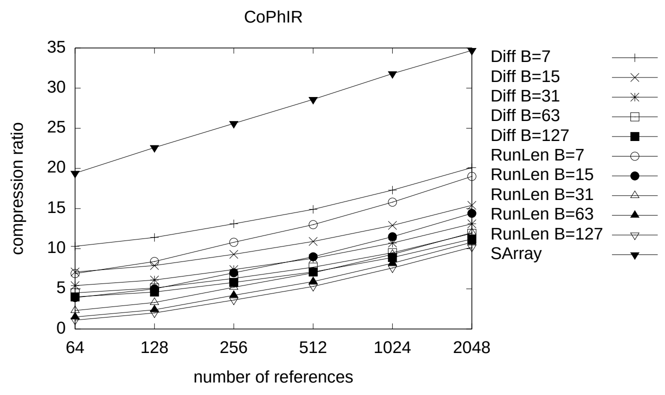
Table 1: Statistics of the covering radius (30-th nearest neighbor) of the CoPhIR database.

σ	γ	max-ratio			
		mean	stddev	min	max
64	100	1.14	0.17	1.00	2.01
128	100	1.11	0.14	1.00	1.87
256	100	1.10	0.17	1.00	2.27
512	100	1.05	0.07	1.00	1.58
1024	100	1.03	0.06	1.00	1.51
2048	100	1.02	0.02	1.00	1.10
64	500	1.08	0.14	1.00	1.86
128	500	1.05	0.10	1.00	1.80
256	500	1.03	0.09	1.00	1.77
512	500	1.01	0.02	1.00	1.12
1024	500	1.01	0.03	1.00	1.22
2048	500	1.00	0.01	1.00	1.07
64	1000	1.05	0.08	1.00	1.62
128	1000	1.02	0.03	1.00	1.14
256	1000	1.01	0.03	1.00	1.14
512	1000	1.01	0.02	1.00	1.12
1024	1000	1.00	0.01	1.00	1.06
2048	1000	1.00	0.01	1.00	1.07

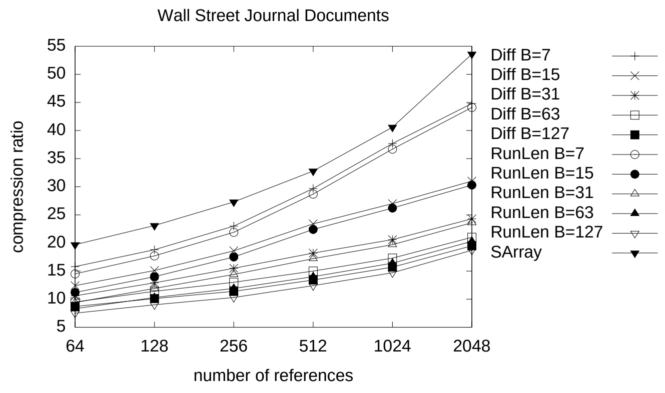
Table 2: Radius statistics for the database of documents.

the space of documents. For the space of colors we found a behavior similar to CoPhIR, yet requiring some additional bits per object.

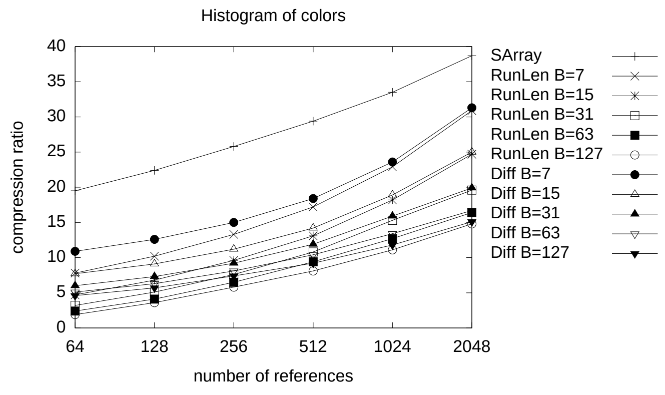
Our experiments confirm that the number of induced runs is large: the smallest index is the run-length based one and the largest compressed index is the *sarray*, as shown in Figure 6. Note that even the space gain of *sarray* is considerable compared to the plain index. It can be seen that σ is also a crucial parameter for compression: small σ values produce a small index, yet it needs to review larger portions of the database.



(a) CoPhIR, 10^7 objects; the plain index uses 267 megabytes



(b) Documents, 25,057 objects; the plain index uses 0.67 megabytes



(c) Colors, 112,682 objects; the plain index uses 3.1 megabytes

Figure 6: Compression ratio as a percentage of the plain inverted index.

σ	γ	max-ratio			
		mean	stddev	min	max
1000	64	1.05	0.09	1.00	1.70
1000	128	1.03	0.06	1.00	1.28
1000	256	1.03	0.06	1.00	1.25
1000	512	1.02	0.05	1.00	1.24
1000	1024	1.01	0.04	1.00	1.25
1000	2048	1.00	0.02	1.00	1.15
2000	64	1.04	0.09	1.00	1.69
2000	128	1.03	0.06	1.00	1.26
2000	256	1.02	0.05	1.00	1.25
2000	512	1.02	0.04	1.00	1.24
2000	1024	1.01	0.03	1.00	1.25
2000	2048	1.00	0.02	1.00	1.14
3000	64	1.04	0.08	1.00	1.63
3000	128	1.02	0.05	1.00	1.25
3000	256	1.02	0.05	1.00	1.25
3000	512	1.01	0.04	1.00	1.24
3000	1024	1.01	0.03	1.00	1.25
3000	2048	1.00	0.02	1.00	1.14

Table 3: Radius statistics for the color histograms database.

6.2.1. Time performance of the compressed index

In these experiments, shown in Table 4, all the compressed indexes were produced with induced runs. For the *plain* index we show the two encodings, with and without induced runs, because this affects the retrieval speed. For example, for the CoPhIR index the plain index working with the induced runs is about 2.5 times faster than the original one. This is not surprising since the runs allow faster skipping when carrying out the intersections. For *differences* and *run-length* encodings, the parameter B (Section 6.2) drives the tradeoff between time and compression. Run-length and differences are still interesting methods since they achieve low compression ratios, as shown in Figure 6. Moreover, for

type of		search time (sec)		
encoding	B	CoPhIR	documents	colors
Differences	7	2.57	0.020	0.0057
Differences	15	3.34	0.020	0.0058
Differences	31	4.81	0.022	0.0061
Differences	63	7.69	0.025	0.0066
Differences	127	13.50	0.028	0.0075
Run-Length	7	2.57	0.019	0.0054
Run-Length	15	2.73	0.019	0.0055
Run-Length	31	2.75	0.019	0.0056
Run-Length	63	2.71	0.019	0.0054
Run-Length	127	2.64	0.020	0.0055
sarray	-	0.34	0.031	0.0056
plain (*w/runs*)	-	0.17	0.024	0.011
plain (*original*)	-	0.42	0.029	0.014

Table 4: Average search time on the compressed NAPP inverted index and the plain version. Indexes were configured to use $\sigma = 2048$ and $(t = 2)$-threshold search. Indexes for CoPhIR use $\gamma = 15000$, and $\gamma = 1000$ for documents and colors.

the CoPhIR dataset, the run-length based indexes are just four times slower than the NAPP inverted index (without runs).

This tradeoff is significant for the CoPhIR database, where the search time increases several times as compared with the plain representation. The *sarray* structure is quite fast (faster than *plain original*) and still compresses significantly. This is explained because the *sarray* gives constant time access to the i-th element [15].

Contrasting with the CoPhIR results, compressed indexes for the documents database are as fast as the plain representation, and even faster for some configurations (i.e., for *sarray*). Even more, for the smaller colors dataset, all compressed indexes are twice as fast as the original index, even surpassing the plain index with runs. Remarkably, the run-length representation uses almost constant time as B grows.

dim	mean	stddev	min	max
4	1.00	0.00	1.00	1.00
8	1.00	0.00	1.00	1.00
12	1.00	0.02	1.00	1.21
16	1.00	0.02	1.00	1.19
20	1.01	0.04	1.00	1.24
24	1.02	0.04	1.00	1.26

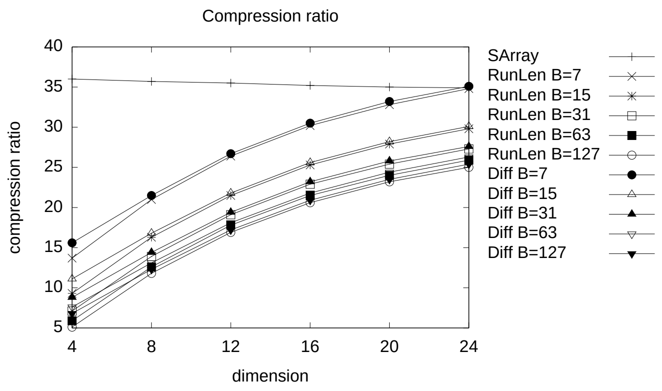
Table 5: Proximity ratio as affected by the dimensionality. The NAPP uses a threshold of 2.

6.3. The dimensionality effect

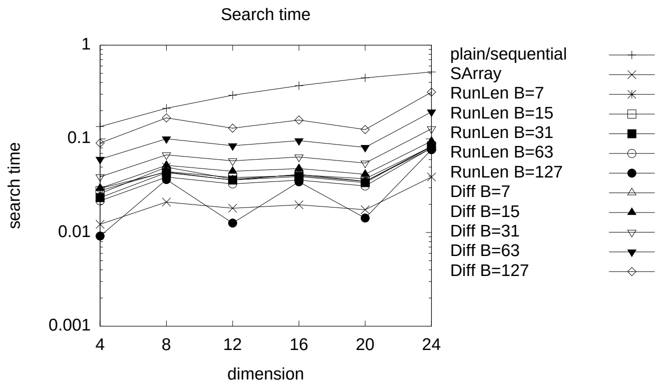
For this experiment, we produce databases in the real unitary hypercube of dimensions 4, 8, 12, 16, 20, and 24, and select coordinates using a uniform distribution. Each dataset contains one million vectors, $n = 10^6$. All queries are allowed to compare 1,000 objects, and all indexes are built using 2,048 references and $K = 7$. Under this configuration, the number of compared objects is fixed to 0.3% of the database, in terms of distance computations. We performed $30NN$ queries for new randomly generated vectors.

We can see the effect of the dimensionality on the compression ratio in Figure 7(a). As expected, the compression capabilities are significantly reduced as the dimension increases. About search quality, we remark that only 0.3% of the database is compared to the query (and thus the search time is essentially constant), yet we achieve close to perfect recall for all configurations. The search time is depicted in Figure 7(b). We can observe the speedup (up to two orders of magnitude) against sequential search, which is the only choice for exact searches on high dimensional datasets (dimension over 20, in practice).

The behavior of the proximity ratio of the nearest neighbor is shown in Table 5. The mean is quite small, ranging from 1 to 1.02, with a very small standard deviation. The maximum ratio is also small, yet it clearly increases with the dimension.



(a) The effect of the dimension on the compression ratio.



(b) Searching time on increasing dimensionality

Figure 7: Behavior of the NAPP compressed index as the dimension increases, on 10^6 random vectors.

Random uniform data is free of clusters. Even on this setup our index shows a good tradeoff among memory, space, speed, recall, and proximity ratio. To put this in perspective, notice that even if the document dataset has intrinsic dimensionality higher than 24, it yields better compression because its do form clusters.

7. Conclusions and Future Work

We introduced a novel approximate index for general metric and similarity spaces called the NAPP inverted index. Our index is capable of achieving high recall in sub-second queries, even for large databases. The plain index uses a few integers per object and the compressed versions use a few bits per object, with a small penalty in search speed for large databases, and a small speedup for small ones. The compression allows one to efficiently use higher hierarchies of memory (RAM and caches). From another perspective, medium-sized indexes (a few millions of objects) can fit in small computers with limited resources and mobile devices, bringing proximity search to these popular computing instruments.

The quality achieved for our index was measured in two senses: recall and proximity ratio. In both of them, the NAPP inverted index is a very competitive option when compared with traditional solutions.

We introduced a novel technique able to induce runs in the inverted index, usable in at least two scenarios: for speeding up a plain index, and for inducing compression in compressed indexes. The *sarray* index produces a fast compressed version and can be used with or without induced runs. Differential encoding plus run-length compression achieves high compression rates and at the same time very fast indexes.

We measure the behavior of our techniques in three real world datasets, and a set of six synthetic databases. Experiments on these datasets study different aspects of the NAPP index. Real world databases show the performance (time, compression, and retrieval quality) of what can be found on real applications. Synthetic databases explore the effect of the dimensionality in our index, that is one of the greatest challenges of metric indexes. We also analyzed the performance in terms of space usage, a problem usually ignored in the literature. In all cases, both our plain and compressed NAPP indexes display an exceptionally good tradeoff between memory, time, recall and proximity ratios, making them excellent options for several real world applications.

Nevertheless, the time performance our method is tightly linked to the underlying t-threshold algorithm, which is primarily designed for uncompressed inverted indexes. The design of faster ad-hoc algorithms for the t-threshold problem on compressed inverted lists, and the optimization and scalability of the technique using parallel and distributed techniques, remain as open problems.

Another challenge is how to efficiently support dynamism on the NAPP inverted index, that is, how to support insertions and deletions of objects and refrences. Probably this can be addressed with a variation of dynamic compressed bitmaps [14]. Maintaining set R dynamically also requires new efficient algorithms to locate objects affected by the inserted reference.

Construction time is dominated by the σn distances computed, hence the preprocessing step is linear on σ for a fixed n and can be large. For example, for CoPhIR, it ranges from 49 minutes to 32 hours, for $\sigma = 64$ and 2048 respectively. For the documents database, it requires 14.45 seconds using 64 references, and up to 9 minutes for $\sigma = 2048$. A simple scheme to speed up the construction is to index the references and then solve KNN searches over R, speeding both search and construction times. In particular, we may use a NAPP inverted index to index R. Notice that using a larger R set yields a faster index; the sketched boosting technique may allow a significant increase in the number of references.

References

[1] G. Amato, P. Savino, Approximate similarity search in metric spaces using inverted files, in: InfoScale '08: Proceedings of the 3rd international

conference on Scalable information systems, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2008, pp. 1–10.

[2] R.A. Baeza-Yates, B.A. Ribeiro-Neto, Modern Information Retrieval, ACM Press / Addison-Wesley, 1999.

[3] J. Barbay, C. Kenyon, Adaptive intersection and t-threshold problems, in: Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM-SIAM, ACM, 2002, pp. 390–399.

[4] C. Böhm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases, ACM Computing Surveys 33 (2001) 322–373.

[5] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, F. Rabitti, Cophir: a test collection for content-based image retrieval, CoRR abs/0905.4627v2 (2009).

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

[7] E. Chávez, G. Navarro, R. Baeza-Yates, J.L. Marroquín, Searching in metric spaces, ACM Comput. Surv. 33 (2001) 273–321.

[8] P. Elias, Universal codeword sets and representations of the integers, Information Theory, IEEE Transactions on 21 (1975) 194 – 203.

[9] A. Esuli, 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), Boston, USA, pp. 17–24.

[10] K. Figueroa, K. Fredriksson, Speeding up permutation based indexing with indexing, in: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications, IEEE Computer Society, pp. 107–114.

[11] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA), CTI Press and Ellinika Grammata, Greece, 2005, pp. 27–38.

[12] G.R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces (survey article), ACM Trans. Database Syst. 28 (2003) 517–580.

[13] D.E. Knuth, The Art of Computer Programming, Volume III: Sorting and Searching, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2nd ed edition, 1998.

[14] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Transactions on Algorithms (TALG) 4 (2008) article 32. 38 pages.

[15] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007, SIAM, New Orleans, Louisiana, USA, 2007.

[16] M. Patella, P. Ciaccia, Approximate similarity search: A multi-faceted problem, Journal of Discrete Algorithms 7 (2009) 36–48.

[17] V. Pestov, Intrinsic dimension of a dataset: what properties does one expect?, in: Proc. 20th Int. Joint Conf. on Neural Networks, Orlando, FL, 2007, pp. 1775–1780.

[18] V. Pestov, An axiomatic approach to intrinsic dimension of a dataset, Neural Networks 21 (2008) 204–213.

[19] S.J. Puglisi, W.F. Smyth, A.H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2007).

[20] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM/SIAM, San Francisco, CA, USA, 2002, pp. 233–242.

[21] H. Samet, Foundations of Multidimensional and Metric Data Structures, The morgan Kaufman Series in Computer Graphics and Geometic Modeling, Morgan Kaufmann Publishers, University of Maryland at College Park, 1 edition, 2006.

[22] E.S. Tellez, E. Chavez, On locality sensitive hashing in metric spaces, in: Proceedings of the Third International Conference on SImilarity Search and APplications, SISAP 2010, ACM, New York, NY, USA, 2010, pp. 67–74.

[23] E.S. Tellez, E. Chavez, A. Camarena-Ibarrola, A brief index for proximity searching, in: Proceedings of 14th Iberoamerican Congress on Pattern Recognition CIARP 2009, Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, 2009, pp. 529–536.

[24] E.S. Tellez, E. Chavez, M. Graff, Scalable pattern search analysis, in: To appear in the Third mexican congress on Pattern Recognition, MCPR 2011, Springer Verlag, Lecture Notes in Computer Science, 2011.