Succinct Nearest Neighbor Search

Eric Sadit Tellez[*]
Universidad Michoacana de San Nicolás de Hidalgo
México
sadit@lsc.fie.umich.mx

Edgar Chávez
Universidad Michoacana de San Nicolás de Hidalgo
México
elchavez@umich.mx

Gonzalo Navarro[†]
Dept. of Computer Science
University of Chile
Chile
gnavarro@dcc.uchile.cl

ABSTRACT

In this paper we present a novel technique for nearest neighbor searching dubbed *neighborhood approximation*. The central idea is to divide the database into compact regions represented by a single object, called the *reference*. To search for nearest neighbors a set of candidate references is first obtained and later enriched with the database objects associated to those references.

This approach can be implemented with an inverted index, which in turn can be represented in a succinct way, spending just a few bits per object. As a consequence it is possible to store the index in main memory, even for relatively large databases.

The speed/compression/recall tradeoff achieved is excellent. To obtain 92% recall in 30-nearest neighbors searches the index reviews less than 0.6% of the database, in time ranging from 0.35 to 2.67 seconds using from 93 to 24 Mbytes for a ten million objects database. The tradeoff comes from using different compression techniques. The uncompressed index requires 0.17 seconds and 267 Mbytes of space.

A quality measure complementary to the recall is the ratio between the covering radius of the actual nearest neighbors and the near neighbors reported by the algorithm. Using this measure our results are within a small constant compared to the exact results.

1. INTRODUCTION

A metric space is a pair (U, d) with U a universe of objects and $d(\cdot,\cdot)$ a distance function $d : U \times U \to \Re$ obeying the metric axioms: For all $x, y, z \in U$ $d(x, y) > 0$ or $d(x, y) = 0 \iff x = y$, $d(x, y) = d(y, x)$, 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 S is the database $S \subseteq U$. Problems of diverse domains, from pattern recognition, classification, statistics, to multimedia indexing 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. We will focus our attention on the second type of queries.

ℓNN searching has a well known linear worst case [7, 21, 12, 4], when the database has high intrinsic dimensionality. Under this circumstance, traditional techniques using the triangle inequality or families of indexes such as *compact partition* indexes and *pivot based* indexes suffer from a condition known as the *curse of dimensionality* (*CoD*) [7, 21, 12, 4]. The problem can be studied from the theoretical point of view using the *concentration of measure* [17, 18]. To overcome this limitation one alternative is to move from exact indexes to approximate indexes, as described in Section 2.

Our contribution is twofold. First we describe the *neighborhood approximation* (NAPP), a simple mapping from general metric spaces to a set of integers (with a proximity semantic). NAPP is an approximate technique with high recall and high quality. The second contribution is to represent the NAPP in a succinct way, giving rise to the *compressed NAPP inverted index*. The combination of these two novelties produces a data structure allowing 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 shows the mapping transformation and Section 4 the underlying data structure for our index. The compression of the index is explained in Section 5.2. Experimental results showing search time, memory usage and recall of our index in real databases are shown in Section 5 and conclusions and future work are discussed in Section 6.

2. RELATED WORK

Exact searching methods are divided into two broad categories as follows:

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 $d(\cdot,\cdot)$. If we count the number of distances computed to answer a query, a pivot based index using many pivots surpasses the performance of a compact partition index, but the optimal number of pivots is too large for high dimensional databases.

Bottom line, both schemes are not effective in high dimensional spaces, due to the CoD. Several books and surveys cover this and

[*]Partially funded by CONACYT grant 161913. Also, part of this work was developed during his stay at the DCC of the University of Chile.

[†]Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile; and by Fondecyt Grant 1-090037, Chile.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SISAP 11, June 30–July 1, 2011, Lipari, Italy. Copyright © 2011 ACM 978-1-4503-0795-6/11/06... $10.00.

other topics for exact proximity searching [7, 21, 12, 4]. They provide excellent introductory and reference material in the area.

To speed up searches, we can drop the condition of retrieving all the objects relevant to the query. A recent survey covers many of the techniques to trade speed for accuracy in approximate searches [16]. Below we review some papers related to our approach.

2.1 Permutation Index

An index called the *Permutation Index* (PI) [6] has high precision and recall even for high intrinsic dimensional datasets and can even be used in similarity spaces (when the distance does not obey the triangle inequality). The PI dramatically reduces the external distance computations, i.e., the number of candidate objects to be compared directly to the query.

The idea behind PI is to represent each database object with the permutation of a set of references, 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, PI is especially useful for expensive distances, such as the Hausdorff distance over the minutia of fingerprints. All the permutations of S can be represented with $n\sigma \log \sigma$ bits. The search is completed 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 PI is paid with a large number of L_1 or L_2 distances in the representation. This internal complexity eats up the advantage of saving distance computations unless the cost of the distance function is very expensive compared with L_1 or L_2. Some authors have tried to reindex this phase hoping that the resulting space has lower intrinsic dimensionality [10]. In general the scalability is limited because the resulting vector space has a medium to high intrinsic dimension, which forbids the use of exact searching methods.

2.2 The Metric Inverted Index

Amato et al [1] gave a nice algorithmic way to simplify the distance computation between permutations with the *Metric Inverted Index*. For each object $K \ll \sigma$ closest references are stored and indexed with an inverted index. The vocabulary is the set of permutants R and each posting list consists of sorted pairs $(objId, pos)$.

The permutation π_q is known at query time and the permutation π_u for $u \in S$ is partially known because in the inverted file at most K references are stored in the posting. For the missing references a constant penalty ω is added to compute the Spearman Footrule; 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. The union produces better quality results, while intersection produces 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 which 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 its shared prefix (if any). Longer shared prefixes hint high proximity and short length prefixes reflect low proximity. This strict notion of proximity yields to very low recalls, allowing only small γ values. This condition is somewhat alleviated by using several permutations 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 data set [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 which is more powerful and simple.

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 effectivity (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 partition the space into a set of *regions*. Each region is represented by an object of the database. An object representing a region will be called a *reference*, and set R, with $|R| = \sigma \ll n$, is the set of all references, representing all regions.

Each object $u \in U$ is represented by a set of objects, 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. We also assume $\sigma \ll n$.

Now that the set of regions will act as database object proxies, just as permutations did in permutation based indexes. We follow

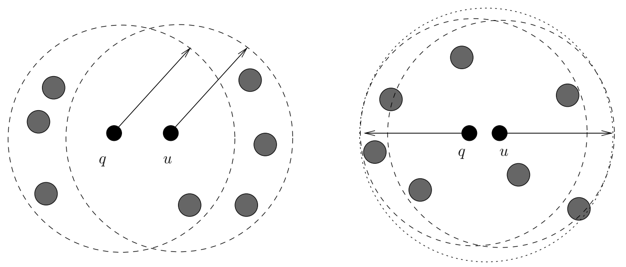


Figure 1: An example showing shared references as proximity predictor. Gray balls are references, dark balls are objects in S.

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 compute its representation P_q. As in any index, we will obtain a set of candidate objects in S which need to be checked against the query. The list of candidates in NAPP will be objects u such that $P_u \cap P_q \neq \emptyset$.

We believe the above framework captures the essence of the permutations approach, yet it is simpler and provides an excellent tradeoff between space usage, speed, recall and retrieval quality.

3.2 Retrieval Quality Considerations

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

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 Observation 2 becomes tighter. A final remark is that R should have the same distribution of S. A rule of thumb will be to have as many references as one can handle without slowing down the process of comparing q with the set of references, and select them uniformly at random from the database.

Figure 1 shows a bad case on the left, and a more realistic case on the right. This is a core heuristic in our approximate index.

4. THE NAPP INVERTED INDEX

Proximity in the NAPP framework is hinted by the number of shared references, i.e., the size of the intersection of two sets, hence the natural choice for an index is an inverted index. Each region in R will be represented by an integer identifier, and hence the representation of each object will be a list of integers.

The size of R, the set of references, should be way smaller than the database S, yet it should reflect the distribution of objects in S. Hence we select $\sigma \ll n$ objects for R uniformly at random. 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 have $R = \{1, \cdots, \sigma\}$ and $S = \{1, \cdots, n\}$; it should be clear from context which collection an index i refers to.

For our computations 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.
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 most objects with a small intersection cardinality (1 or 2) appear very frequently in the candidate list even though they are not always close to the query. It is then natural to impose an additional condition about the minimum size of the intersection. This strategy 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.

Algorithm 2 Solve an *ℓNN* query in the inverted index

```
1: Let $t$ be the minimum allowed cardinality of the intersection, and $\Gamma$ the
   number of desired candidates to be checked with the distance.
2: Let $L[1,\sigma]$ be an array of sorted lists of integers, i.e. the inverted index.
3: Compute $P_q[1,K]$
4: Let $Q$ be the corresponding lists of the regions in $P_q$, computed as
   $Q[1,K] = L[P_q[1]], \ldots, L[P_q[K]]$
5: Let $POS[1,K]$ be an array of pointers to the current position of the
   $i$-th list on $Q$, starting in 1.
6: Let $CND$ be a priority queue to store the set of candidates
7: while $Q.Length \geq t$ do
8:     Ascending sort $Q$ using $Q[i][POS[i]]$ as key for $1 \leq i \leq$
       $Q.Length$; identifiers are permuted to follow the order of $Q$
9:     if $Q[1][POS[1]] \neq Q[t][POS[t]]$ then
10:        Advance all $POS[i]$ for $1 \leq t-1$ such that $POS[i]$ is the
           smallest item such that $Q[i][POS[i]] \geq Q[t][POS[t]]$
11:        Restart the loop
12:    end if
13:    Find the greatest $k \geq t$ such that $Q[k][POS[k]] = Q[t][POS[t]]$,
       then $k$ is the cardinality of the intersection
14:    if $k = Q.Length$ then
15:        Increment all $POS[i] \leftarrow POS[i] + 1$ for $1 \leq i \leq Q.Length$
16:    else
17:        Advance all $POS[i]$ for $1 \leq k$ such that $POS[i]$ is the smallest
           item such that $Q[i][POS[i]] \geq Q[k+1][POS[k+1]]$
18:    end if
19:    Append $Q[t][POS[t]]$ to $CND$
20:    Evaluate the distance between the query and $\Gamma$ candidates (with the
       highest priority from $CND$)
21:    Return the $\ell$ closest objects to the query
22: end while
```

Note 1: Increasing and advancing in POS and Q requires to checked for overflows, in such 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].

To represent R we need $\sigma \log n$ bits, using pointers to S, or σ objects if they are represented explicitly. 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, i.e., a total of Kn integers of $\log n$ bits, distributed among the σ sorted lists.

4.1 Compressing the inverted index

The space of our index is Kn integers. For a typical value like $K = 7$, this is larger than the typical overhead introduced by tree data structures. Nevertheless there is room for improvement by compressing the index using indexed 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 $[1 \cdots n]$; then ideally we can represent that list using $\log \binom{n}{s}$ bits. Using the *sarray* of Okanohara and Sadakane [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.

4.2 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. The procedure starts computing the mapped space, where each object $u \in S$ is represented by P_u, i.e., implemented as an array of integer identifiers of $KNN_R(u)$. Also, P_u is sorted according to the region identifiers, not by proximity to u. Secondly, the database is lexicographically sorted, using a linear sort for the first levels and a threeway quick sort for deeper levels, similarly to practical suffix array sorting algorithms [19]. Thirdly, the permutation of S induced by the sort is used to renumber the database S. Finally, the inverted index is created using the permuted mapping.

The first step creates ranges inside inverted lists 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, $2s \log n/s + 2s$. 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 access to the i-th item in the list needs at most B integer decodings. There are at most $\frac{Kn}{B}$ samples overall and, as usual, the worst case in space will arise when each list contains $\frac{Kn}{B\sigma}$ of them.

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 $\sigma = 5$, $n = 21$.

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[1] 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. We need $(s/B) \log(|L'|/(s/B)) + 2s/B + o(s/B)$ bits for this representation. In the worst case $s = \frac{Kn}{\sigma}$ and $|L'| = 2s \log(n/s) + 2s$, and the space for the samples adds up to $(Kn/B)(\log(B(\log(\sigma/K) + 1)) + O(1)$. This is usually small as $|L'| \ll n$ due to the runs.

Let $B = \Theta(\log \frac{Kn}{\sigma})$. Accessing the i-th integer then costs $O(\log \frac{Kn}{\sigma})$ decodings. Each inverted list needs $\alpha = \frac{Kn}{\sigma} / \Theta(\log \frac{Kn}{\sigma})$ absolute samples, i.e. $\alpha = o(\frac{Kn}{\sigma})$. Each sample needs $\log n$ bits if it is explicitly represented, then representing samples using *sarray* requires $\alpha \log \frac{n}{\alpha} + 2\alpha + o(\alpha)$ bits (for each list).

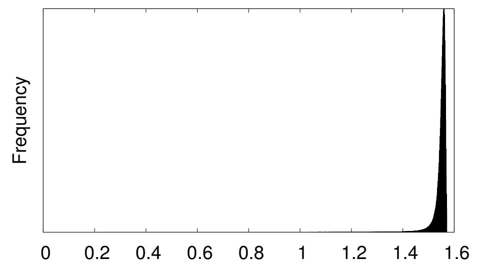
Differential + Run-Length encoding.

Differential encoding of the inverted index represents runs with unary coding, thus for long runs this method is suboptimal. A better option is to encode the length of the run using Elias-γ code. 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.

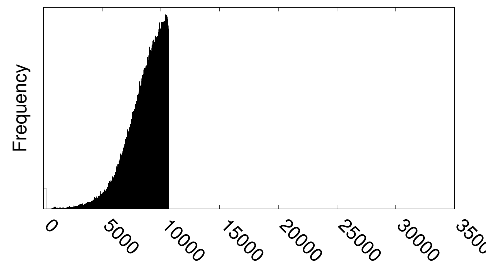
A natural optimization is introduced as follows: if the j-th and the $(j+1)$-th absolute samples are separated by exactly B positions we say that the range is *filled*, and no representation of the data inside the range is necessary; just the sampling data is stored. If the i-th integer lies in a filled range, it is decoded in constant time.

5. EXPERIMENTAL RESULTS

[1]We define $docId_0 = 0$.



(a) Documents are a high dimensional database. The query set is the first 200 documents



(b) 2^{16} randomly selected subspace of CoPhIR 10M. The query set is the first 200 objects

Figure 3: Histograms of our test databases using the selected query sets.

We show experimental results for two representative spaces.

Documents. A collection of 25157 short news articles in the TFIDF format from Wall Street Journal $1987 - 1989$ files from TREC-3 collection. We use the angle of vectors as distance measure. It is available from 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 needs 0.23 seconds.

CoPhIR MPEG7 Vectors. A subset of 10 million 208-dimensional vectors from the CoPhIR database [5]. We use the L1 distance. Vectors are a linear combination of five different MPEG7 features [5]. The first 200 vectors from the database are queries; each query consist in searching 30*NN*. A sequential scan takes 47.3 seconds. Figure 3(b) shows the histogram of distances, as seen from our query set.

We use 30 nearest neighbors because it is a common value as an output in a multimedia information retrieval system.

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 experimentation was executed in a 16 core Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running CentOS Linux. The entire databases and indexes 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 experimental support for this choice is not shown because of space restrictions, but is quite similar to those presented by [9, 1], and recently by [24].

5.1 General performance

In this section we analyze the recall, total time, and the percentage of the reviewed database in the CoPhIR 10M database. Due to space constraints this section do not show all the experiments we have performed. Experimental results are shown for two type of queries: t-threshold queries, and 1-threshold with fixed number of verified objects.

Our primary quality measure is the recall: the ratio between relevant results in S and the relevant objects obtained. Since our queries are 30*NN*, the recall is just the number of true 30*NN* elements returned, divided by 30. This measure ignores how close the non-relevant objects are from the true 30*NN*. 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 function of t; smaller t gives better recall. Smaller σ values (number of references) give better recall, but at the cost of 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, when $t > 1$ the order of σ is descending as the recall increases and for $t = 1$ σ is in ascending order. We put labels in selected curves of the figures to increase readeability.

Larger σ values imply 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 in text inverted indexes for natural languages.

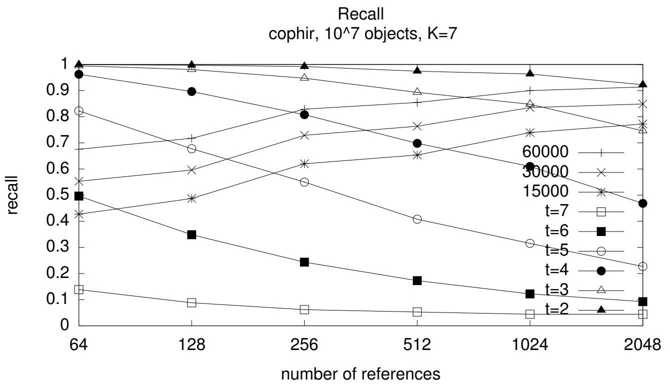
All these parameters induce tradeoffs that can be used to effectively tune real applications. For example, for $t = 2$ and $\sigma = 2048$ the index achieves 0.92 of recall, reviewing 0.6% of the database in about 0.4 seconds.

Large 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 driven by CND, i.e. 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*.

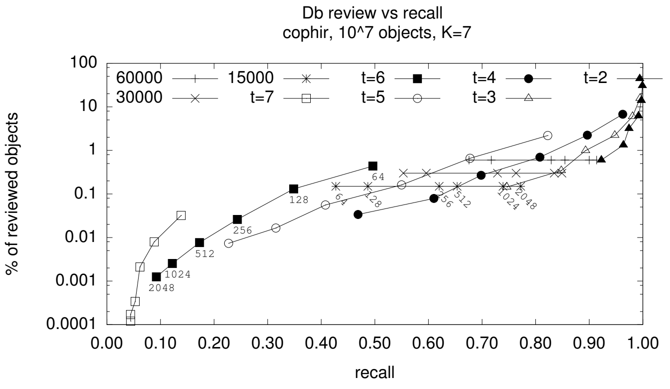
Naturally, a hybrid configuration achieves better control of the performance and quality, i.e. the combination of t-threshold and fixed γ. For example, for $\sigma \geq 1024$ pure t-threshold configurations yields to better times than just fixing the cardinality, see Figure 4(c). The inverse is true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

Comparison with previous work.

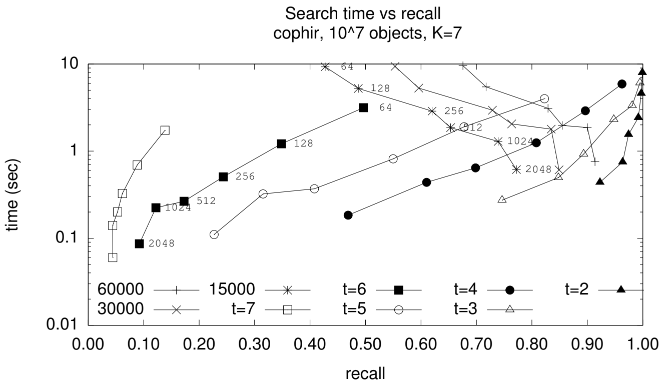
We compared our work with four indexes using the permutations approach [6, 9, 1, 23], as described in Section 1. Since we do not have the actual implementations of all the indexes being compared, and because of space limitations, we carry out a single comparison fixing our attention on the recall, disregarding 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



(a) Recall.



(b) Percentage of database reviewed.



(c) Search times.

Figure 4: CoPhIR 10M recall and performance

single instance, and without query expansion.

For all indexes we fix $\gamma \leq 1000$. Figure 5 shows the results of our experiments, showing that the NAPP inverted index is better when the number of references is large. The comparison is with the database of documents because of its high intrinsic dimensionality.

Proximity Ratio as a Measure of Retrieval Quality.

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 some 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 produce results that are very close to the real answers, supporting Observation 1, which bounds the distance to the query, not the recall. Actual distances for the 30-th nearest neighbor in our query set have the following statistics: mean=3958.16, standard deviation=930.24, minimum=1418, and maximum=6531. The complete histogram of distances is shown in Figure 3(b).

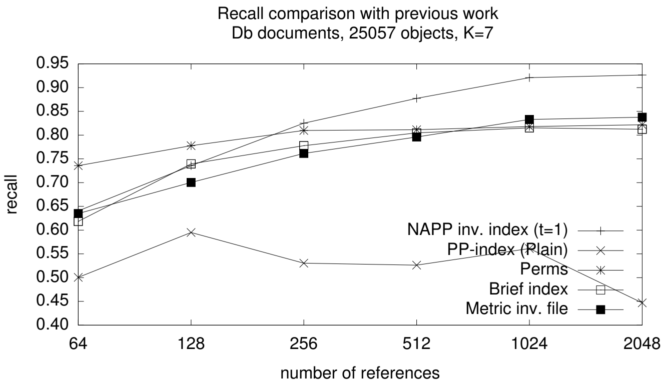


Figure 5: Recall behavior of previous work. We fix all the methods to review at most 1000 candidates.

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 database of CoPhIR.

The same statistics are given for the database of *documents* in Table 2. Notice that this database has worse performance, probably because of the high intrinsic dimensionality. The statistics for the actual distances of the 30-th nearest neighbor are: mean=1.22, standard deviation=0.19, minimum=0.61, and maximum=1.50.

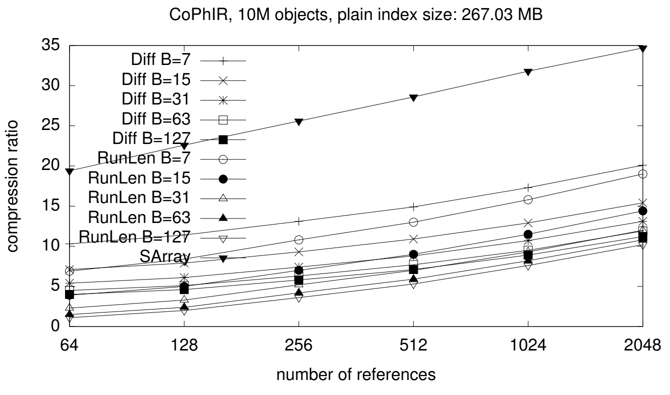
In both experiments, our results are very close to the query, based on the two histograms of distances of Figures 3(a) and 3(b).

5.2 The Compressed NAPP Inverted Index

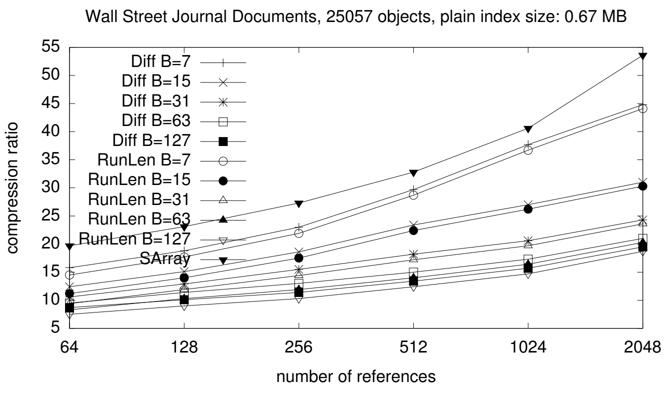
Our plain inverted index uses a fixed number of bits. For example, the index for CoPhIR 10M uses $267MB$, i.e., each object is represented with 224 bits, using integers of 32 bits. The compressed representation uses from 10 to 80 bits per object for *CoPhIR*, and 20 to 80 bits in *documents*. Our experiments confirm

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 the documents.



(a) CoPhIR 10M



(b) Documents

Figure 6: Compressing inverted indexes for our experimental data sets.

that the number of 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. σ is also a crucial parameter for compression. Small σ values produce a small index, yet it needs to review larger portions of the

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

Table 3: The average time necessary to search a query in the compressed NAPP inverted index and the plain version. Indexes were configured using $\sigma = 2048$, $(t = 2)$-threshold search. Indexes for CoPhIR $\gamma = 15000$, and $\gamma = 1000$ for indexes of *documents* database.

database.

5.2.1 Time performance of the compressed index

In the experiment, all compressed indexes were produced with induced runs. For the *plain* index we show the two encodings, with and without induced runs because it 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 t-threshold algorithm is instance optimal. For *differences* and *run-length* encodings, the parameter B (Section 5.2) manages 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, 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* coding is quite fast (faster than *plain original*) and still compress significantly. This can be 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*). Note that small compressed inverted indexes can fit in the CPU caches. This applies to inverted lists involved in the solution of a particular query. Also notice that the distribution of runs produces easier instances for the t-threshold algorithm, taking advantage of the Barbay-Kenyon t-threshold algorithm.

6. CONCLUSIONS AND FUTURE WORK

We introduced a novel approximate index for general metric spaces called NAPP inverted index. Our index is capable of achieving high recall in sub-second queries, even for large metric 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. The compression allows one to efficiently use higher hierarchies of memory (RAM, L2 and L1 cache). From another perspective, medium-sized indexes (a few millions of objects) can fit in small and mobile devices, bringing proximity search to these popular supports.

The quality achieved for our index was measured in two senses: recall and proximity ratio. In both senses 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.

We are working on the creation of faster ad-hoc algorithms for the t-threshold problem for the compressed inverted lists, and in the optimization and scalability of the technique using parallel and distributed techniques.

Another challenge is to efficiently support dynamic operations in the NAPP inverted index for both S and R. Probably this can be addressed with a variation of dynamic compressed bitmaps [14]. Dynamic R sets requires new efficient algorithms to locate objects affected by the inserted reference.

The build 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 10M, 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 building times. In particular, we may use a NAPP inverted index to index R. Notice that using a larger R set produces a faster index; the sketched boosting technique may allow a significant increase in the number of references.

7. REFERENCES

[1] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

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

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

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

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

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

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

[8] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194 – 203, mar 1975.

[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)*, pages 17–24, Boston, USA, 2009.

[10] K. Figueroa and K. Frediksson. Speeding up permutation based indexing with indexing. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 107–114. IEEE Computer Society, 2009.

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

[12] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[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 and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.

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

[16] M. Patella and P. Ciaccia. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36–48, 2009.

[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*, pages 1775–1780, 2007.

[18] V. Pestov. An axiomatic approach to intrinsic dimension of a dataset. *Neural Networks*, 21(2-3):204–213, 2008.

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

[20] R. Raman, V. Raman, and 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)*, pages 233–242, San Francisco, CA, USA, January 2002. ACM/SIAM.

[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 and E. Chavez. On locality sensitive hashing in metric spaces. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP 2010, pages 67–74, New York, NY, USA, 2010. ACM.

[23] E. S. Tellez, E. Chavez, and 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, pages 529–536, Berlin, Heidelberg, November 2009. Springer Verlag.

[24] E. S. Tellez, E. Chavez, and 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.