A Compressed Index for Hamming Distances

Francisco Santoyo[1], Edgar Chávez[2], and Eric S. Téllez[3]

[1] School of Electrical Engineering
Universidad Michoacana, México
psantoyo@dep.fie.umich.mx
[2] Department of Computer Science
CICESE, México
elchavez@cicese.mx
[3] INFOTEC
Aguascalientes, México
eric.tellez@infotec.com.mx

Abstract. Some instances of multimedia data can be represented as high dimensional binary vectors under the hamming distance. The standard index used to handle queries is Locality Sensitive Hashing (LSH), reducing approximate queries to a set of exact searches. When the queries are not selective and multiple families of hashing functions are employed, or when the collection is large, LSH indexes should be stored in secondary memory, slowing down the query time.

In this paper we present a compressed LSH index, queryable without decompression and with negligible impact in query speed. This compressed representation enables larger collections to be handled in main memory with the corresponding speedup with respect to fetching data from secondary memory.

We tested the index with a real world example, indexing songs to detect near duplicates. Songs are represented using an entropy based audio-fingerprint (AFP), of independent interest.

The combination of compressed LSH and the AFP enables the retrieval of lossy compressed audio with near perfect recall at bit-rates as low as 32 kbps, packing the representation of 30+ million music tracks of standard length (which is about the total number of unique tracks of music available worldwide) in half a gigabyte of space. A sequential search for matches would take about 15 minutes; while using our compressed index, of size roughly one gigabyte, searching for a song would take a fraction of a second.

Keywords: Audio indexing, Succinct Audio-Fingerprint, Succinct LSH Indexes.

1 Introduction

High dimensional binary vectors under the hamming distance can represent many interesting objects for applications. The standard index used to handle queries in this setup is Locality Sensitive Hashing (LSH), reducing approximate queries to a set of exact searches. When the queries are not selective and multiple families of hashing functions are employed, or when the collection is large, LSH indexes should be stored in secondary memory, slowing down the query time.

A.J. Machado Traina et al. (Eds.): SISAP 2014, LNCS 8821, pp. 113–126, 2014.
DOI: 10.1007/978-3-319-11988-5_11 © Springer International Publishing Switzerland 2014

Compressing the data is an option to avoid overflow to secondary memory as long as the compressed representation is usable without decompressing. In this paper we present a compressed LSH index, which can be queried without decompression and with negligible impact in the query speed. This defer the use of secondary memory for larger collections, implying a non-trivial speedup with respect to a secondary memory index.

We performed a real world test for our algorithms. We selected the problem of indexing music tracks. The total amount of music tracks worldwide is in the order of 30 million. The *Apple iTunes* music-store lists less than 30 million songs in its catalog. Other on-line music-stores like *Amazon MP3* offer a 22 million song catalog to choose from and *Deezer* or *Spotify* just advertise more than 30 million songs. With the increasing number of records, the creation of high performance music search algorithms becomes a basic requirement for any music on demand application. The industry should respond with systems being able to discover, navigate, and recommend music. One basic tool for music retrieval is the simple matching of a track in a collection.

The task of matching whole songs in audio collections has been tackled by fingerprinting the audio, and then comparing the corresponding fingerprints. This method serves multiple purposes, on the one hand the fingerprinting procedure masks subtle differences between audio objects and conflates near duplicates. On the other hand, having a succinct representation of the audio avoids a lengthy comparison in the original domain.

While audio fingerprints (AFP) can be made very robust to ambient noise and other severe degradations, there is a tension between robustness and the memory footprint of the representation [1]. Other commercial approaches use a time-frequency representation of the audio (as in [2] and [3]) with limitations in both, the processing power required to obtain the AFP, and the type of index to be engineered to obtain fast answers.

We did focus on whole-song identification with the only expected degradation transcoding (e.g. lossy compression), which induces very mild distortions to the songs. We will also assume that both the query song and the song in the database have the same length and that they are correctly aligned. This is the case, for example, of an audio labeling service; where the user rips the audio from a CD and wants it automatically labeled.

Our second contribution consists in a lightweight AFP using just a few bits per minute of audio (precisely one bit every two seconds). Every song will produce a string of bits of the same size (the strings are cyclically completed to a fixed size). To compare two songs we use the hamming distance between the corresponding AFPs. With this procedure near duplicates are conflated (they have small Hamming distance) and non corresponding songs have large distances. These two facts allow extremely fast searches with no false positives. The unique combination of speed, precision and small memory footprint is unparalleled in the literature. We can pack about 300 million minutes of audio in about one gigabyte, and query a database of this size in a fraction of a second.

2 Related Work

A variant of the classical KD-tree algorithm which efficiently indexes high-dimensional data by recursive spatial partitioning is presented by McFee and Lanckriet [4]. They perform experiments on the One Million Song Dataset [5] to demonstrate that content-based similarity search can be significantly accelerated by the use of spatial partitioning

structures. However, KD-tree suffers (as any exact spatial and metric index) of the so called *curse of dimensionality* (Samet [6] and Chavez et al. [7]); as any spatial method working with the explicit dimensionality, it becomes suboptimal on high dimensional datasets [7].

The interested reader on a more general point of view of Music Information Retrieval is referred to the surveying works of Lu [8], which provides a comprehensive survey of audio indexing and retrieval techniques; Stober and Nürnberger [9] present a structured view on the last decade of Music Information Retrieval research; and Yan et al. [10] present a rich review of large-scale multimedia analysis techniques.

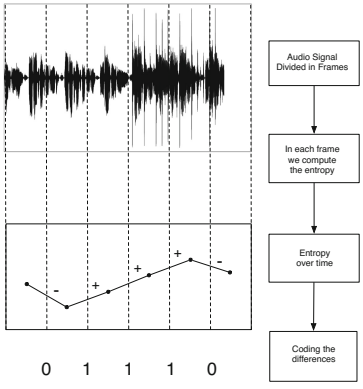
3 Computing the Fingerprint

Our approach is derived from [11]. The signal is framed and for each frame we measure the information content, directly in the time domain. The Information content or self information $I(p_i)$ of a value v_i, depends only on its probability $p_i = P(v_i)$ to occur, the less likely a value to appear, the more information it will bring when it shows up. Therefore, the self information must be a monotonically decreasing function of the probability, usually it is defined as $I(p_i) = \ln(\frac{1}{p_i}) = -\ln(p_i)$

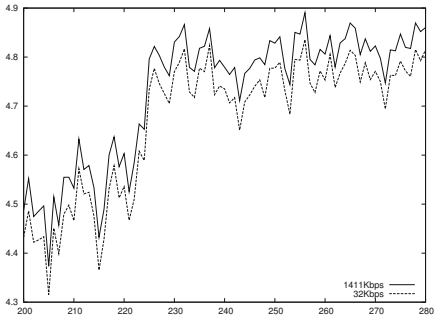
Let $X = \{x_1, x_2, \cdots, x_n\}$ a sequence of values, with f_i denoting the frequency of x_i. The entropy $H(X)$ is the average of all the information contents weighted by their probabilities to occur

$$H(X) = \sum_{i=1}^{m} \frac{f_i}{n} \log(\frac{x_i}{n}) = -\sum_{i=1}^{n} p_i \log(p_i)$$

Transcoded versions of the same song will differ in the amount of information packed, producing a vertical shift. To avoid this shift we keep only the sign of the derivative,



(a) The audio fingerprinting process



(b) The entropy curve for the song *Chilanga Banda - Café Tacuba* for versions @1411Kbps and @32Kbps. The vertical axis shows the magnitude of the entropy as a function of time (horizontal axis).

Fig. 1. The process for obtaining the fingerprint and an example of two versions of the same song

which can be computed measuring only the relative change between frames, with 1 if the change is positive and 0 otherwise, see Figure (1a). The frames can be overlapped to smooth the changes in the sequence, because transcoding induces a time shift. In other words, two versions of a song will be almost aligned. The frame overlapping smooths the time shift, however this increases the size of the fingerprint and also modifies the distance distribution between songs. The optimal frame size and overlap amount can be found experimentally, the optimization goal is to minimize the distance between near duplicates while maximizing the distance between unrelated songs. Figure 1a illustrates the process of obtaining the fingerprint of a song. One of the nice characteristics of this approach is the trivial parallelization in obtaining the fingerprint, fitting well in modern hardware. Furthermore, the operations needed are simple enough to be implemented in low-end processors, such as mobile devices.

We computed the AFP with frames of sizes half, 1 and 2 seconds with overlaps of 0, 50, 75, 90 and 95%. The experiments were performed in a sample of 4000 songs with mp3 encoding at different bit-rates mp3@$\{128, 96, 64, 32\}$ Kbps, also, we denote the original as wav@1411Kbps. We observed two things in this experiment, the first is that overlap increases the distance between near duplicates; the minimum distance (and variance) is obtained with no overlap. The second observation was that the frame size was not critical, changes in the distance were not significant. Due to the later fact, we selected a 2 second frame with no overlap because it gives the smallest memory footprint.

Table 1 shows both the average distance and the standard deviation matrix for a 2-second frame with no overlap. For near duplicates the average distance goes from 1.8% to 4.2% with a small variance.

Table 1. Average normalized distances from one song to all its versions. (μ, ρ)

	@1411Kbps	@128Kbps	@96Kbps	@64Kbps	@32Kbps
@1411Kbps	(0, 0)	(0.018, 0.076)	(0.022, 0.076)	(0.026, 0.076)	(0.054,0.075)
@128Kbps		(0, 0)	(0.008, 0.010)	(0.012, 0.013)	(0.042, 0.028)
@96Kbps			(0, 0)	(0.013, 0.013)	(0.042, 0.028)
@64Kbps				(0, 0)	(0.042, 0.029)
@32Kbps					(0, 0)

Complementarily, for the same setup, the average distance between unrelated songs is one order of magnitude larger as shown in Table 2. The average distance is 42.7%, with standard deviation depending on the bit-rate.

Table 2. Average normalized distances from one song to all the other songs. (μ, ρ)

	@1411Kbps	@128Kbps	@96Kbps	@64Kbps	@32Kbps
@1411Kbps	(0.426, 0.061)	(0.427, 0.061)	(0.427, 0.061)	(0.427, 0.060)	(0.429, 0.058)
@128Kbps		(0.427, 0.061)	(0.427, 0.061)	(0.427, 0.060)	(0.429, 0.058)
@96Kbps			(0.427, 0.061)	(0.427, 0.060)	(0.429, 0.058)
@64Kbps				(0.427, 0.060)	(0.429, 0.058)
@32Kbps					(0.429, 0.058)

Since we do not have overlap, frames are independent of each other which gives us the capability to parallelize the fingerprint computation.

We ended up with a succinct representation of a song, fast to compute, and with nice conflation properties. Near duplicates are one order of magnitude closer than unrelated songs, respectively shown in Tables 1 and 2. This fact avoids the retrieving of false positives when querying by content, as discussed in the next section.

4 Matching Songs

Since we are using the Hamming distance with the AFP, it is natural to use *locality sensitive hashing* (LSH) (Gionis et al. [12]) as the base of our index.

LSH is a fast approximate proximity searching technique giving probabilistic guarantees on the quality of the result. The general idea of an LSH index is to find hashing functions that applied to close objects give the same bucket with high probability. This technique is prone to two types of errors, namely: 1) False positives, when two non related objects fall in the same bucket, and 2) False negatives, when two near duplicates end in a different bucket. Those errors can be alleviated by using more than one LSH function.

In general, the process of finding hashing functions g_i can be tricky; however Hamming spaces are the most studied and hash functions are very simple, they are just random samples of the bit strings.

Definition 1 (Locality Sensitive Hashing, Gionis et al. [12]). *A family of hashing functions $\mathcal{H} = \{g_1, g_2, \cdots, g_h\}$, $g_i : U \to \{0, 1\}$ is called (p_1, p_2, r_1, r_2)-sensitive, if for any p, q:*

— *If $d(p, q) < r_1$ then $Pr[\mathsf{hash}(p) = \mathsf{hash}(q)] > p_1$*
— *If $d(p, q) > r_2$ then $Pr[\mathsf{hash}(p) = \mathsf{hash}(q)] < p_2$*

Where $\mathsf{hash}(u)$ is the concatenation of the output of individual hashing functions g_i, following a fixed order, i.e. $\mathsf{hash}(u) = g_1(u)g_2(u) \cdots g_h(u)$.

Let d_{max} be the maximum possible distance between objects in the metric space; the probability that some g_i computes the same hash for $u, v \in U$ is determined as $Pr[g_i(u) = g_i(v)] = 1 - d(u, v)/d_{max}$.

If hashing functions are selected independently, with replacement, and equally probably to fail, we obtain $Pr[\mathsf{hash}(u) = \mathsf{hash}(q)] = 1 - (d(u, v)/d_{max})^h$. In order to have a sound LSH scheme, the above formula should comply with definition 1.

Other data models and distance functions, like vectors measured with Minkowski norms or sets with Jaccard distances are also studied in the literature, Gionis et al. [12], and Andoni & Indyk [13].

4.1 Normalizing AFP Size

Our AFP is a bit string of variable length, proportional to the length of the song. The LSH based index needs a fixed size representation. Hence each fingerprint is *conceptually* expanded to be of the size of the largest fingerprint on our database, or any fixed

large size if that length is unknown beforehand. Let ℓ_{max} be such length, and ℓ_i be the length of the i-th fingerprint s_i. Let $s[j]$ be the j-th bit in the fingerprint s.

Our database S of size n is denoted as $S = \{\hat{s}_1, \hat{s}_2, \cdots, \hat{s}_n\}$, where $\hat{s}_i[j] = s_i[1+(j \mod \ell_i)]$ for all $1 \leq j \leq \ell_{max}$. The same method is applied to obtain any valid object (e.g. queries $q \in U$), using ℓ_{max} obtained from S. The distance between any two database objects is D (Hamming distance).

A single index would be enough if the length of the objects are not very different, all of them will have the same length using the above normalization. However in a database with disparate lengths, the normalization would be unfair since LSH captures (relatively) very large hashes for smaller fingerprints. In this case the database should be partitioned into several sets, and indexed separately.

4.2 Indexing the AFP Database

We have seen before that all versions of the same song are quite close to each other as shown in Table 1, and complementarily unrelated songs are distant from each other as shown in Table 2. There is a balance between the size of the sample in an LSH index, the implicit searching radius, and the time to retrieve the near duplicates. We cascade a set of indexes of non decreasing selectivity, all of them using different samplings. We apply the hashing in order until reaching the last index, and give up when we reach it.

We compute the minimum and maximum sampling sizes using Table 1 and the size of the AFPs. We can use a single parameter α (where $0 \leq \alpha \leq 1$) establishing what is the selectivity of each index in the cascade, i.e, anything with larger distance than α will be discarded. The maximum α would be fixed to 0.15 because this will capture most of the near duplicates.

The average searching cost will be very low since most queries are solved using only the first indexes, only a fraction of the queries would require all the indexes. The searching steps are described in Algorithm 1.

Algorithm 1: Near-duplicate searching of songs

Input: The query song Q, the index set $\mathcal{L}$ corresponding to the size of Q , the maximum distance α.

Output: The set R of near-duplicate objects of Q.

1: Process Q and obtain its fingerprint q
2: Normalize the audio-fingerprint as the object $\hat{q}$
3: Initialize $R \leftarrow \emptyset$
4: **for all** $I \in \mathcal{L}$ **do**
5: Lookup I to match similar objects $\hat{q}$, put candidates in C
6: Remove from C objects not matching the length of q
7: $R \leftarrow R \cup \{\hat{u} \in C \mid d(q, u) \leq \alpha\}$.
8: **Stop** the iteration if $|R| \geq 1$ (or the minimum desired cardinality)
9: **end for**

5 Compressing LSH

To reduce false positives and false negatives multiple hashing functions are needed; thus, the search index will require more memory and probably will resort to secondary memory for large instances. Our goal is to produce a representation of the LSH index with close to optimal storage.

The general idea is to represent the hashing tables as inverted indexes, and in turn to represent those inverted indexes as an indexed sequence. This type of representation can be compressed and can be queried without decompression.

Let $T = s_1 s_2 \cdots s_n$ be a sequence of symbols on the alphabet Σ of size σ, i.e. $s_i \in \Sigma$. Without loss of generality, let Σ be a set of integers, that is $\Sigma = \{1, 2, \cdots, \sigma\}$. The i-th symbol in T is denoted as T_i.

An index of sequences (IoS) provides three basic operations:

- $\mathsf{Rank}_c(T, pos)$ counts how many c's occurs in T until pos, $c \in \Sigma$.
- $\mathsf{Select}_c(T, r)$ returns the smaller position pos such that $\mathsf{Rank}_c(T, pos) = r$.
- $\mathsf{Access}(T, pos)$ retrieves the symbol stored at the position pos in T, i.e., T_{pos}.

Notice that an IoS replaces T, since we can reconstruct it using Access, but our notation requires to put T in the arguments even when it is not necessarily stored.

5.1 A Brief Survey for Indexes of Sequences

There are several indexes achieving near optimal space bounds, we briefly review some of them. We start establishing the memory costs for any representation.

Memory usage. Let n_c be the number of symbols c in T, then from information theory we can obtain the following formulation, using a fixed code word for each symbol, we require at least $nH_0(T) \leq n \log \sigma$ bits, here, $H_0(T)$ is the order zero empirical entropy of T, i.e., $nH_0(T) = n \sum_{c \in \Sigma} p_c \log \frac{1}{p_c} = \sum_{c \in \Sigma} n_c \log \frac{n_c}{n}$ bits. Here p_c is the probability of occurrence of c in T, empirically, $p_c = n_c/n$.

Binary alphabets (Bitmaps). Most IoS use as building block the binary case, an alphabet of two symbols $c \in \{0, 1\}$ without loss of generality. Consider a bitmap B with n bits, let n_0 and n_1 the number of 0's and 1's respectively in the bitmap.

Gonzalez et al. [14] developed a fast practical approach. It consists on a directory structure of absolute Rank_c samples every $\log^2 n$ bits. This structure solves Rank_c and Select_c in $O(\log n)$ time, and Access in constant time. It stores the plain bitmap using n bits and used $o(n)$ bits to store the absolute samples.

Several indexes achieve near-optimal space for binary alphabets. For example, Raman et al. [15] based on classifying bit blocks and then codifying blocks using tuples (c_i, offset) where c_i describes the class of the block (the number of bits set to 1) and an offset to distinguish a block inside the class c_i. These tuples are cleverly codified such that classes with few members will produce smaller offset's codes. This approach uses $nH_0(B) + O(\log \log n) + o(n)$ bits and solves the three basic operations in constant time; however, the constants are too large in practice. Claude and Navarro [16] improved

the practical, sample based implementation introduced by Gonzales in [14], achieving better performance in practice; however, the space space complexity is similar and can be a waste of resources when $n_1 \ll n$. On the other side, Okanohara and Sadakane [17] presented the *sparse array* (SArray) which achieves $n_1 \log n/n_1 + O(n_1)$ space with $O(1)$ time for Select_1.

In addition to the above, there exists specialized indexes achieving near optimal space. One example is presented in Tellez et al. [18,19] they key idea consist in storing differences with variable length integer codifications along the necessary directory structures to accelerate operations. Tellez introduced Diffset, which is basically the representation of the bitmap as a compressed sorted list with directory structures to provide fast Rank_c, Select_c, and Access performances. Diffset achieves $nH_0(B) + o(n)$ bits. Similarly, Diffset-RL is defined adding run-length compression for large consecutive runs of ones; depending of the distribution it could produce much better compression and times or add n bits in the worst case.

Larger alphabets. For $\sigma > 2$ there are several canonical techniques to index a sequence T of length n as described below.

Grossi et al. [20] introduce the Wavelet Tree (WT), it uses $n \log \sigma + O(\sigma \log n)$ bits solving all operations in $O(\log \sigma)$ time. There exists several variants of the WT. For example, the WT with Huffman shape or with internal bitmaps compressed to nH_0, like surveyed by Navarro and Mäkinen [21]. Very large alphabets are problematic with this scheme since the time complexity of all operations depend on σ.

Golinsky et al. [22] introduce a fast index, robust to large σ. It uses $n \log \sigma + o(n \log \sigma)$ bits, it solves Select_c in constant time, and both Rank_c and Access on $O(\log \log \sigma)$ time. Claude and Navarro [16] show an implementation of Rank_c and Access on $O(\log \sigma)$ time performing better in practice for most instances.

Tellez [18] introduces the Extra Large Bitmap (XLB) family of indexes for large alphabets achieving both compression and fast operations, specially on sequences with low local entropy. The main idea is to codify a sequence using a permutation of $[1 \dots n]$; the trick is to store the inverse in $o(n)$ bits extra, while the direct is represented with a large bitmap that takes advantage of the sparseness of the resulting bitmap. The sequence $T = T_1 T_2 \cdots T_n$ is represented with a bitmap $P[1, \sigma n]$ where the i-th bit is 1 if $T_{i \bmod \sigma} = \frac{i}{\sigma}$, and 0 otherwise. Then, P is a large bitmap, with regions of length n corresponding to each symbol. The basic algorithm solves Rank_c and Select_c on T performing Rank_1 and Select_1 on P. Access is solved using Π^{-1} where $\Pi(i) = \mathsf{Select}_1(P, i) \mod n$. Also, Π^{-1} is stored with the cyclic representation of Munro et al. [23] using $\frac{1}{t} \log n$ bits; it solves Π^{-1} in t time (Select_1 operations on P), where $t \geq 1$. Since all operations are delegated to the P bitmap, the efficiency is tightly linked to P. Since we need to represent a very large bitmap of $n\sigma$ bits with n bits set to 1, then we need an underlying bitmap taking advantage of the sparseness of the represented bitmap.

5.2 The Sequence Representation of LSH

Consider the database $S \subseteq U$, $S = \{u_1, u_2, \cdots, u_n\}$, and a family of hashing functions $\mathcal{H} = \{g_1, g_2, \cdots, g_h\}$, where $h = |\mathcal{H}|$ and $g_i : U \to \{0, 1\}$. A tag of an object

is defined as $\mathsf{tag}(u) = g_1(u)g_2(u)\cdots g_h(u)$. The set of all possible values of $\mathsf{tag}(\cdot)$ is called the alphabet, $\Sigma = \{0, 1, 2, \cdots, \sigma - 1\}$, where $\sigma = |\Sigma| \leq 2^h$. Even when $\mathsf{tag}(u) = \mathsf{hash}(u)$, conceptually tag is an atomic item (indivisible and recognized as a unit), and defines a sequence's symbol. Let us define $T = \mathsf{tag}(u_1)\,\mathsf{tag}(u_2)\cdots\mathsf{tag}(u_n)$. We can store T using $\log\binom{n}{n_1, n_2, \cdots, n_\sigma}$ bits, where n_i is the number of occurrences of the tag i in T.

Recall that high quality results with LSH require several LSH tables, which increase the memory cost and hence the need of a memory efficient representation. The alphabet derived from the LSH representation is large. One option of index is WT, described in the previous section, but the operations for the simulation of LSH make heavy use of the Select_c operation. The performance of WT and most of its variants is poor for our needs. Hence we focus on the Golynski and the XLB approaches. In particular, we use XLB with SArray, Diffset and Diffset-RL; XML-SArray will use $n \log \sigma + o(n)$ bits, while XML-Diffset and XML-DiffsetRL can achieve better compression under particular entries with low local entropy; however, the latter two will introduce a minor term of $O(n \log \log n)$ bits which can impact on sequences with high local entropy since they will be added to the resulting worst case.

5.3 Solving Approximate Nearest Neighbors with T

The abstract data structure for LSH needs access to the buckets. To solve a query, the structure needs to count the number of items in a bucket, and retrieve all items on it. Figure 2b shows a hash table of an example database of 16 objects. Each row is a bucket, represented by some hash value. Figure 2a shows the sequence T of the hash table.

As an abstract data structure T solves similarity queries using the same proximity properties than LSH tables. Algorithm 2 solves the approximate $\mathsf{nn}_{d,S,U}(q)$ queries. The idea is to retrieve all items using $\mathsf{Select}_{\mathsf{tag}(q)}$.

LSH is essentially an indexed table, we can emulated its functionality as follows. i) The number of items with the same hash c is computed with $\mathsf{Rank}_c(T, n)$ (Figure 2c); ii) all items with the same tag c are retrieved as $\mathsf{Select}_c(T, i)$ for $i = 1, 2, \cdots$, $\mathsf{Rank}_c(T, n)$.

Algorithm 2: Searching for the approximate $\mathsf{nn}_{d,S,U}(q)$

Input: The query q, the distance function d, and T.
Output: The approximate nearest neighbor $\mathsf{nn}^*(q)$

1: Let $c = \mathsf{tag}(q)$
2: Let $\mathsf{nn}^*(q) \leftarrow$ undefined
3: **for** $i = 1$ to $\mathsf{Rank}_c(T, n)$ **do**
4: Define p as $\mathsf{Select}_c(T, i)$-th object in T
5: $\mathsf{nn}^*(q) \leftarrow p$ **if** $\mathsf{nn}^*(q)$ is undefined or p is closer to q than the previous $\mathsf{nn}^*(q)$
6: **end for**

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{tag}(u_i)$	8	7	4	6	2	1	3	8	8	4	0	1	1	1	5	9

(a) The sequence T representing the LSH table of Figure 2b

hash	tag	occurrences list
0000	0	→ 11
0001	1	→ 6, 12, 13, 14
0010	2	→ 5
0011	3	→ 7
0100	4	→ 3, 10
0101	5	→ 15
0110	6	→ 4
0111	7	→ 2
1000	8	→ 1, 8, 9
1001	9	→ 16

(b) An example of the LSH hash table representation

	$\mathsf{Select}_c(T,i)$			
	1	2	3	4
$\mathsf{Rank}_0(T,n)=1$	11			
$\mathsf{Rank}_1(T,n)=4$	6	12	13	14
$\mathsf{Rank}_2(T,n)=1$	5			
$\mathsf{Rank}_3(T,n)=1$	7			
$\mathsf{Rank}_4(T,n)=2$	3	10		
$\mathsf{Rank}_5(T,n)=1$	15			
$\mathsf{Rank}_6(T,n)=1$	4			
$\mathsf{Rank}_7(T,n)=1$	2			
$\mathsf{Rank}_8(T,n)=3$	1	8	9	
$\mathsf{Rank}_9(T,n)=1$	16			

(c) Reconstructing the LSH table

Fig. 2. An example of the LSH sequence representation LSH, and its operations

6 Experimental Results

All experiments were performed in a 16 core Intel Xeon 2.40 GHz workstation with 32GiB of RAM, running CentOS. All tasks were restricted to run into a single core, we did not exploited the parallel capabilities of our workstation.

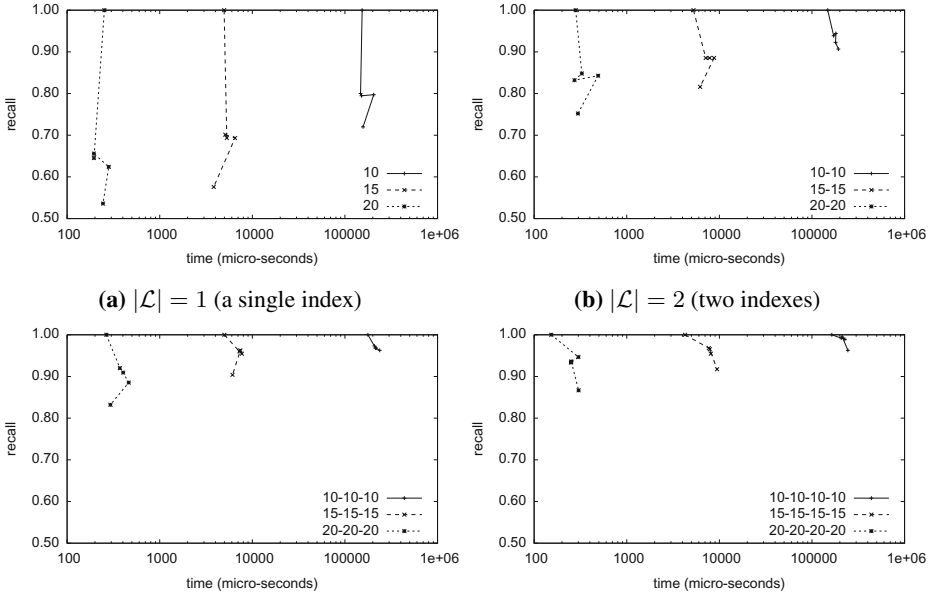
6.1 Case of Study

For the real world example, we collected 3.7 million songs (about 1.5 Tb) and fingerprinted them with the techniques described, using 1 bit every two seconds of music. A collection of this size is necessarily diverse. Our database of fingerprints requires 54 MiB (15 bytes per song), that is always maintained in main memory. Proximity between fingerprints is measured with the Hamming distance. Each fingerprint requires 0.31 seconds in average to be computed (reading PCM files with 16 bits per sample, and 44100 Hz, 1411kbps). We randomly selected 400 songs from the songs database and compressed them to @128kbps, @96kbps, @64kbps, and @32kbps. These versions of the song are similar to the versions in personal music libraries.

Table 3 contains the average size of the LSH indexes for our setups (over our 3.7 million song audio-fingerprint database). It is interesting to notice that the compression ratio (smaller is better) decreases as h does. Using the compact sequence representation of LSH we expect to use from 54% to 77% of the original space of an LSH index. This improvement is important because we are trying to maintain our data structures in the higher places of the memory hierarchy, asymptotically obtaining faster indexes.

Table 3. Average memory requirements

method	h		
	10	15	20
LSH	14.0 MiB	14.3 MiB	22.0 MiB
compressed LSH	7.6 MiB	9.9 MiB	17.0 MiB
compression ratio	0.54	0.69	0.77



(a) $|\mathcal{L}| = 1$ (a single index)

(b) $|\mathcal{L}| = 2$ (two indexes)

(c) $|\mathcal{L}| = 3$ (three indexes)

(d) $|\mathcal{L}| = 4$ (four indexes)

Fig. 3. Recall vs. time to retrieve the near-duplicated using different LSH families and several indexes. The points in the curves are generated searching for different versions of the songs, i.e., perfect recall is achieved for 1411kbps, and from it each point matches with versions with decreasing quality 128, 96, 64, and 32 kbps.

The *recall* increases as h decreases. However, as h decreases we expect higher searching times since the database is partitioned in fewer buckets. The searching and recall compromise is shown in Figure 3. For instance, Figure 3a shows the performance of a single index. Decreasing h increases the searching time exponentially, while the recall is only moderately improved. An option to improve the performance is using more than one index, a set of them $\mathcal{L}$ (as described in Algorithm 1). Using two indexes, Figure 3b, performs better than reducing h, at the cost of using twice the memory. The improvements with three and four indexes, Figures 3c and 3d respectively, are more notorious. Note that the average searching time is smaller than just multiplying the searching time for the number of indexes, this is because we only advance to the next index if the

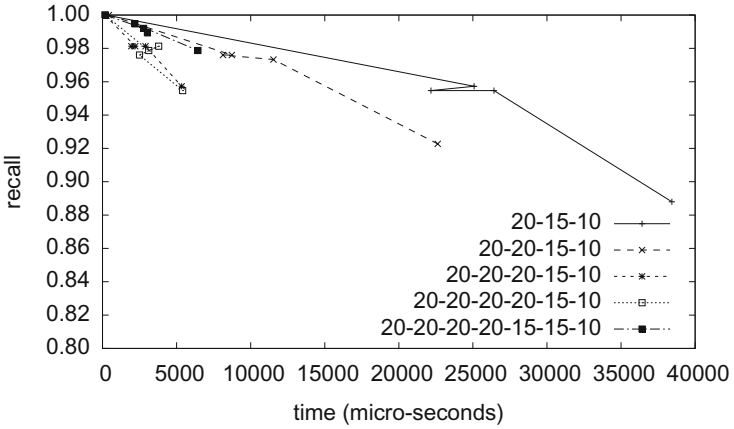


Fig. 4. Our mixed setups that optimize both recall and time performances. The points in the curves are generated searching for different versions of the songs, i.e., perfect recall is achieved for 1411kbps, and from it each point matches with versions with decreasing quality 128, 96, 64, and 32 kbps.

Table 4. Average memory requirements for our mixed setups

setup	memory		
	compressed LSH	LSH	compression ratio
20, 15, 10	34.5 MiB	50.3 MiB	0.686
20, 20, 15, 10	51.5 MiB	72.3 MiB	0.712
20, 20, 20, 15, 10	68.5 MiB	94.3 MiB	0.726
20, 20, 20, 20, 15, 10	85.0 MiB	116.3 MiB	0.735
20, 20, 20, 20, 15, 15, 10	95.4 MiB	130.6 MiB	0.730

current one fails to retrieve the near-duplicates (Algorithm 1). It is clear that the better time-recall tradeoff is found for several indexes with large h values, also the memory cost is reduced using the compressed LSH index (Table 3).

As seen on Figure 3d, we obtain at least 85% of recall for all versions when $h = 20$. However we always obtain at least 90% for $h = 15$, and 95% for $h = 10$. Based on the above facts, we tuned the strategy improving the recall with a small impact in both average searching time and memory cost. The idea is to filter by solving most queries very fast, with only a few queries passing the filter and using a more expensive procedure. This is illustrated in Figure 4. Here we can see that most configurations perform better than 95% of recall with a moderate searching time. For example those instances with at least 5 indexes achieve more than 98% for high quality songs, and less than 7 milliseconds searches. Our setup with seven indexes gives more than 99% of recall and 3 millisecond searches for songs of quality $@64$, $@96$, $@128$ and $@1411$ kbps.

Table 4 shows the cost of storage of the setups of Figure 4. The memory usage is maintained below 100 MiB, since compressed indexes require close to 70% of the uncompressed LSH. Please notice that the compression is important, since it can be central for running a standalone version of the indexes in mobile devices.

7 Conclusions

We presented a compressed index for LSH. The index can be queried without decompression and with negligible impact in the query time. We tested the index for the problem of near duplicate detection in whole-song querying. We made experiments with 3.7 million songs, obtaining near perfect recall and searching times of 5 milliseconds. The index fits well under 100MiB of RAM, and requires only simple operations, easily cacheable. The fingerprint of a 4 minute song is computed in 0.3 seconds in a standard CPU without parallelization. This allows to think in applications running standalone in small devices. Making a simple linear extrapolation of our index, which is a pessimistic assumption, we can fingerprint about 37 million songs in half gigabyte of RAM; being able to query the collection in a fraction of a second. The assumption is pessimistic because the compression ratio and searching times scale sub-linearly.

In future work we will try to estimate with very few parameters the audio quality of a song using our fingerprinting technique. We believe we only need the area under the curve of the time entropy profile. This feature can act as filter in a third party storage and streaming service, for example, the service provider may reject to stream low quality audio found in the users folders.

From the searching point of view we will investigate generalizations of the $\mathcal{L}$ set of indexes using metric indexes, more robust than LSH with higher error rates.

Acknowledgements. We want to thank the anonymous referees who helped us to improve the presentation with insightful observations. This work was partially supported by CONACyT and CICESE grants. The third author was a postdoc in Universidad Michoacana under the CONACyT's grant 179795 (project "Bases de Datos Multimedia Superescalables").

References

1. Chandrasekhar, V., Sharifi, M., Ross, D.: Survey and evaluation of audio fingerprinting schemes for mobile query-by-example applications. In: Proceedings of ISMIR (2011)
2. LTD, S.: (2006), `http://www.shazam.com/`
3. SoundHound (2008), `http://www.soundhound.com/`
4. McFee, B., Lanckriet, G.R.G.: Large-scale music similarity search with spatial trees. In: Klapuri, A., Leider, C. (eds.) ISMIR, pp. 55–60. University of Miami (2011)
5. Bertin-Mahieux, T., Ellis, D.P.W., Whitman, B., Lamere, P.: The million song dataset. In: Klapuri, A., Leider, C. (eds.) ISMIR, University of Miami, pp. 591–596. University of Miami (2011)
6. Samet, H.: Foundations of Multidimensional and Metric Data Structures, 1st edn. The morgan Kaufman Series in Computer Graphics and Geometic Modeling. Morgan Kaufmann Publishers, University of Maryland at College Park (2006)

7. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
8. Lu, G.: Indexing and retrieval of audio: A survey. Multimedia Tools Appl. 15(3), 269–290 (2001)
9. Stober, S., Nürnberger, A.: Adaptive music retrieval - a state of the art. Multimedia Tools and Applications (2012), 'Online First' article
10. Yan, R., Huet, B., Sukthankar, R.: Large-scale multimedia retrieval and mining. IEEE Multimedia 18, 11–13 (2011)
11. Camarena-Ibarrola, A., Chavez, E.: A robust entropy-based audio-fingerprint. In: Proceedings of the International Conference on Multimedia and Expo, ICME 2006, pp. 1729–1732 (2006)
12. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases, VLDB 1999, pp. 518–529. Morgan Kaufmann Publishers Inc., San Francisco (1999)
13. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications ACM 51, 117–122 (2008)
14. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA), pp. 27–38. CTI Press and Ellinika Grammata, Greece (2005)
15. Raman, R., Raman, V., Rao, S.S.: 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), pp. 233–242. ACM/SIAM, San Francisco (2002)
16. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
17. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007. SIAM, New Orleans (2007)
18. Tellez, E.S.: Practical Proximity Searching in Large Metric Databases. PhD thesis, Universidad Michoacana, Morelia, Michoacán, México (July 2012)
19. Tellez, E.S., Chavez, E., Navarro, G.: Succinct nearest neighbor search. Information Systems 38(7), 1019–1030 (2013)
20. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003, pp. 841–850. Society for Industrial and Applied Mathematics, Philadelphia (2003)
21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1) (2007)
22. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, SODA 2006, pp. 368–373. ACM, New York (2006)
23. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 345–356. Springer, Heidelberg (2003)