Indexing and Search for Fast Music Identification*

Guang-Ho Cha

Seoul National University of Science and Technology
Seoul 139-743, Republic of Korea
ghcha@snut.ac.kr

Abstract. In this paper, we present a new technique for indexing and search in a database that stores songs. A song is represented by a high dimensional binary vector using the audio fingerprinting technique. Audio fingerprinting extracts from a song a fingerprint which is a content-based compact signature that summarizes an audio recording. A song can be recognized by matching an extracted fingerprint to a database of known audio fingerprints. In this paper, we are given a high dimensional binary fingerprint database of songs and focus our attention on the problem of effective and efficient database search. However, the nature of high dimensionality and binary space makes many modern search algorithms inapplicable. The high dimensionality of fingerprints suffers from the curse of dimensionality, i.e., as the dimension increases, the search performance decreases exponentially. In order to tackle this problem, we propose a new search algorithm based on inverted indexing, the multiple sub-fingerprint match principle, the offset match principle, and the early termination strategy. We evaluate our technique using a database of 2,000 songs containing approximately 4,000,000 sub-fingerprints and the experimental result shows encouraging performance.

1 Introduction

Large digital music libraries are becoming popular on the Internet and consumer computer systems, and with their growth our ability to automatically analyze and interpret their content has become increasingly important. The ability to find acoustically similar, or even duplicate, songs within a large music database is a particularly important task with numerous potential applications. For example, the artist and title of a song could be retrieved given a short clip recorded from a radio broadcast or perhaps even sung into a microphone. Broadcast monitoring is also the most well known application for audio fingerprinting. It refers to the automatic playlist generation of radio, TV or Web broadcasts for, among others, purposes of royalty collection, program verification and advertisement verification.

Due to the rich feature set of digital audio, a central task in this process is that of extracting a representative audio fingerprint that describes the acoustic content of each song. Fingerprints are short summaries of multimedia content. Similar to a human fingerprint that has been used for identifying an individual, an audio fingerprint

* This study was financially supported by Seoul National University of Science and Technology.

is used for recognizing an audio clip. We hope to extract from each song a feature vector that is both highly discriminative between different songs and robust to common distortions that may be present in different copies of the same source song.

In this paper, we adopt an audio fingerprinting technique [1] based on the normalized spectral subband moments in which the fingerprint is extracted from the 16 critical bands between 300 and 5300 Hz. The fingerprint matching is performed using the fingerprint from 20-second music clips that are represented by about 200 subsequent 16-bit sub-fingerprints.

Given this fingerprint representation, the focus of our work has been to develop an efficient search method for song retrieval. This problem can be characterized as a nearest neighbor search in a very high dimensional (i.e., 3200 (= 200 × 16)) data space.

High dimensional nearest neighbor search (NNS) is a very well studied problem: given a set P of points in a high dimensional space, construct a data structure which, given any query point q, finds the point p closest to q under a defined distance metric. The NNS problem has been extensively studied for the past two decades. The results, however, are far from satisfactory, especially in high dimensional spaces [2]. Most NNS techniques generally create a tree-style index structure, the leaf nodes represent the known data, and searching becomes a traversal of the tree. Specific algorithms differ in how this index tree is constructed and traversed. However, most tree structures succumb to the curse of dimensionality, that is, while they work reasonably well in a 2 or 3 dimensional space, as the dimensionality of the data increases, the query time and data storage would exhibit an exponential increase, thereby doing no better than the brute-force sequential search [2].

Recent work [3, 4, 5] appears to acknowledge the fact that a perfect search that guarantees to find exact nearest neighbors in a high dimensional space is not feasible. However, this work has not been extended to the fingerprinting system which deals with the binary vectors and the Hamming distance metric. That is, in the fingerprinting system, the bit error rate between two binary vectors is used as a distance metric rather than Euclidean distance. Moreover, the big limitation of their work [3, 4, 5] is that it often needs to search a significant percentage of the database.

In this paper, we adopt the inverted file as the underlying index structure and develop not only the technique to apply the inverted file indexing to high dimensioal binary fingerprint databases but also the efficient search algorithm for fast song retrieval. Though our work focuses on searching songs based on audio fingerprints, the devised technique is generally applicable to other high dimensional binary vector search domains.

2 Related Work

Haitsma and Kalker's fingerprinting [6, 7, 8] is the seminal work on fingerprinting. It extracts 32-bit sub-fingerprints for every interval of 11.6 milliseconds in a song and the concatenation of 32-bit sub-fingerprints extracted constitutes the fingerprint of the song. Each sub-fingerprint is derived by taking the Fourier transform of 5/256 second overlapping intervals from a song, thresholding these values and subsequently compu-

ting a 32-bit hash value. They proposed an indexing scheme that constructs a lookup table (LUT) for all possible 32-bit sub-fingerprints and lets the entries in the LUT point to the song(s) and the positions within that song where the respective sub-fingerprint value occurs. Since a sub-fingerprint value can occur at multiple positions in multiple songs, the song pointers are stored in a linked list. Thus one sub-fingerprint value can generate multiple pointers to songs and positions within the songs. By inspecting the LUT for each of the 256 extracted sub-fingerprints a list of candidate songs and positions is generated. Then the query fingerprint block (256 sub-fingerprints) is compared to the positions in the database where the query sub-fingerprint is located.

The first problem of Haitsma-Kalker's method is that the 32-bit LUT containing 2^{32} entries is too large to be resident in memory. Furthermore, LUT is very sparsely filled because only a limited number of songs reside in comparison with the size of LUT. By adopting inverted file indexing we resolve this problem.

In an inverted file index, a list of all *indexing terms* is maintained in the search structure called a *vocabulary*, and the vocabulary is usually implemented by the B-trees. However, in our approach, we employ a hash table instead of the B-trees as the vocabulay in order to accomplish the lookup time of $O(1)$ rather than $O(\log n)$. Contrary to large text databases that widely use the inverted file index where the number of query terms is a few, in fingerprint querying, the number of query terms (i.e, query sub-fingerprints) is several hundreds when we assume the duration of the query song clip is several seconds. Therefor the lookup time of $O(1)$ is crucial.

The second problem of Haitsma-Kalker's method is that it makes the assumption that under *mild* signal degradations at least one of the computed sub-fingerprints is error-free. However, it is acknowledged in the paper that the mild degradation may be a too strong assumption in practice. Heavy signal degradation may be common when we consider transmission over mobile phones or other lossy compressed sources. For example, a user can hear and record a part of a song from the radio in his or her car and then transmit a fingerprint of the song to a music retrieval system using his or her mobile phone to know about the song. Considered those situations, we cannot assume that there exists a subset of the fingerprint that can be matched perfectly.

In order to avoid the above perfect matching problem, if we allow the number of error bits in a sub-fingerprint up to n, then the search time to find matching fingerprints in a database is probably unacceptable. For example, if we cope with the situation that the minimum number of erroneous bits per 32-bit sub-fingerprint is 3, then the number of fingerprint comparisons increases with a factor of 5488 ($= {}_{32}C_1 + {}_{32}C_2 + {}_{32}C_3$) which leads to unacceptable search times. In our approach, we break this dilemma by increasing the number of pointers to a song s by the number of sub-fingerprints with up to the permitted number, say n, of erroneous bits. In other words, in addition to each original 16-bit sub-fingerprint in our system, we generate more $\sum_{i=1}^{n} \binom{16}{i}$ 16-bit sub-fingerprints with up to n toggled bits and use them also as sub-fingerprints for song s. This means that the number of fingerprint comparisons does not increase during the search for song matching even if we cope with the situations that the number of erroneous bits per sub-fingerprint is up to n. By applying this duplication of sub-fingerprints with n toggled bits from original sub-fingerprints to indexing, we are able to resolve the second limitation of Haitsma-Kalker's method.

The third problem of Haitsma-Kalker's method is that their search algorithm is built on the "single match principle", i.e., if two fingerprints are similar, they would have a relatively high chance of having at least one "matching" identical sub-fingerprint, and therefore their method fetches the full song fingerprint from a database and compares it with the query fingerprint as soon as it finds a single sub-fingerprint matched to a certain query sub-fingerprint. They ignore the multiple occurrences of matching. However, in fact, multiple occurrences of sub-fingerprint matching are common and many candidate songs with multiple occurrences of matching are found during the search. Therefore, if the multiple occurrences of matching are not considered in the query evaluation, the search wastes much time to inspect the candidate songs which are eventually judged to be incorrect even though they have several matchings to the query sub-fingerprints. We tackle this problem and improve the search performance by introducing the "multiple sub-fingerprints match principle".

We also introduce the "offset match principle" in the search. It means that if two fingerprints are similar and there are multiple occurrences of sub-fingerprint matching between them, they may share the same relative offsets among the occurrence positions of matching. This offset matching principle improves the search performance greatly by excluding the candidates that do not share the same relative offsets of matching occurrence positions with the query fingerprint. This reduces the number of random database accesses remarkably.

Miller et al. [9, 10] assumed the fingerprint representation of 256 32-bit sub-fingerprints of Haitsma and Kalker [6, 7, 8] and proposed the 256-ary tree to guide the fingerprint search. Each 8192(= 32 × 256)-bit fingerprint is represented as 1024 8-bit bytes. The value of each consecutive byte in the fingerprint determines which of the 256 possible children to descend. A path from the root node to a leaf defines a fingerprint. The search begins by comparing the first byte of the query with the children of the root node. For each child node, it calculates the cumulative number of bit errors seen so far. This is simply the sum of the parent errors and the Hamming distance between the 8-bit value represented by the child and the corresponding 8-bit value in the query. Then a test is applied to each child, in order of increasing error, to determine whether to search that child. If the best error rate seen so far is greater than the threshold, then the child is searched. The search continues recursively and when a leaf node is reached, the error rate associated with the retrieved fingerprint is compared to the best error rate seen so far. If it is less, then it updates the best error rate to this new value and assigns this fingerprint as the best candidate nearest neighbor so far.

The first problem of Miller et al.'s method is that the size of the 256-ary tree is too large and the depth of the tree is also too deep to be practical in the disk-based database search. According to their experimental results [9, 10], they search an average of 419,380 nodes, which is 2.53% of the nodes in the index tree that stores about 12,000,000 sub-fingerprints.

Moreover, they assume that each song is also represented by a fingerprint with 8192 bits, i.e., the same number of bits as the query fingerprint. It means that the length of each song in a database is assumed to be the same as that of the query song. It makes the indexing and search problem simpler. But actually an individual song with an average length of 4 minutes has approximately 10,000 sub-fingerprints in Haitsma-Kalker's method. Therefore, it is not practical to model a song with only a 8192-bit fingerprint and thus this mechanism is not feasible to apply to real applications.

The third problem is that Miller et al.'s 256-ary tree uses a probabilistic method based on a binomial distribution to estimate the bit error rate (BER) in each tree node. This BER is used to determine whether to search that node. However, it is difficult to predict the exact BER in advance, and therefore, the correct rate to find the most similar fingerprint is at most 85% in Miller et al.'s method [9, 10]. In order to increase the correct rate, the expected BER in each node should be determined more conservatively, and in that case, the search performance may degenerate to be worse than that of the brute-force sequential search. Therefore, it is difficult to reduce the search space to find the nearest neighbor in a high-dimensional space using the *k*-ary tree. Furthermore, if the *k*-ary tree tries to reduce the search space more, the error rate increases inevitably. It means that the *k*-ary tree approach cannot overcome the curse of dimensionality problem.

Keeping those limitations in mind, in this paper, we propose a new indexing and search algorithm that resolves the limitations of Haitsma-Kalker's method and Miller et al.'s *k*-ary tree method.

3 Indexing and Search Algorithm

We now describe a new indexing scheme and a new search algorithm for song databases implemented with audio fingerprints. The underlying structure of our indexing is based on the *inverted file* that has been widely used for text query evaluation such as Google [11]. Searching a multimedia database for a multimedia object (video clip or song) is similar to searching a text database for documents. A single multimedia object is represented by multiple atomic units (e.g. subsequent 16-bit sub-fingerprints in our case) and it can be found using the atomic units. Similarly, Documents in a text database are represented by multiple keywords and they are found by keyword matching. This is the reason why we adopt the inverted file as the underlying index structure in our work. However, there are also many differences between them and they make the fingerprint search problem more difficult.

- In the fingerprint search, the query fingerprint may not match any fingerprint in a database because the fingerprint extracted from a query song may have bit errors due to distortions to the query song. In other words, contrary to the text database search where only exact maching is supported, the fingerprint search should identify the correct song in a database even though there is a severe signal degradation of the query song. This means that the fingerprint search must support *imperfect matching* or *similarity search*.
- Individual bits in a fingerprint have their own meaning, and therefore, the Hamming distance between two fingerprints is used as a dissimilarity metric. This means that the search problem is: given a 3200 (= 16 × 200)-bit vector with errors, how to find the vector most similar to that in the 3200-dimensional binary space. However, the high-dimensional similarity search is known to suffer from the *dimensionality curse*. As aforementioned, the indexing method such as the *k*-ary tree approach cannot avoid the the dimensionality curse problem.

- The query fingerprint is assumed to be 200 16-bit sub-fingerprints in our work. However, assuming that the duration of the average song is 4 minutes, then the number of sub-fingerprints in a song is approximately 2,000 in our system. This difference of lengths between the query song and songs in a database makes the search problem more difficult.

We resolve the problems explained above by adapting the inverted file index suitably to our high dimensional binary database and creating the search algorithm with several sophisticated strategies.

3.1 Index Struture

An inverted file index works by maintaining a list of all sub-fingerprints in a collection, called a *vocabulary*. For each sub-fingerprint in the vocabulary, the index contains an *inverted list*, which records an identifier for all songs in which that sub-fingerprint exists. Additionally, the index contains further information about the existence of the sub-fingerprint in a song, such as the number of occurrences and the positions of those occurrences within the song.

Specifically, the vocabulary stores the following for each distinct 16-bit sub-fingerprint *t*,

- a count f_t of the songs containing *t*,
- the identifiers *s* of songs containing *t*, and
- the pointers to the starts of the corresponding *inverted lists*.

Each inverted list stores the following for the corresponding sub-fingerprint *t*,

- the frequency $f_{s,t}$ of sub-fingerprint *t* in song *s*, and
- the positions p_s within song *s*, where sub-fingerprint *t* is located.

t	f_t	*s*	Inverted list for *t*
1100010010001010	1	<5>	5(2: 24, 45)
0100000111001011	2	<4, 34>	4(1: 8), 34(3:78, 90, 234)
1100110001100001	1	<77>	77(1: 18)
⋮	⋮	⋮	⋮
1010111010101001	3	<102, 981, 1201>	102(1: 62), 981(2: 12, 90), 1201(2: 99, 187)

Fig. 1. Inverted file index. The entry for each sub-fingerprint *t* is composed of the frequency f_t, song identifiers *s*, and a list of triplets, each consisting of a song identifier *s*, a song frequency $f_{s,t}$, and the positions p_s within song *s*, where sub-fingerprint *t* is located.

Then the inverted lists are represented as sequences of <s, $f_{s,t}$, p_s > triplets. These components provide all information required for query evaluation. A complete inverted file index is shown in Fig. 1.

The vocabulary of our indexing scheme is maintained as a hash table instead of the B-trees in order to achieve the lookup time approaching $O(1)$ rather than $O(\log n)$. Typically, the vocabulary is a fast and compact structure that can be stored entirely in main memory.

For each sub-fingerprint in the vocabulary, the index contains an inverted list. The inverted lists are usually too large to be stored in memory, so a vocabulary lookup returns a pointer to the location of the inverted list on disk. We store each inverted list contiguously in a disk rather than construct it as a sequence of disk blocks that are linked. This contiguity has a range of implications. First, it means that a list can be read or written in a single operation. Accessing a sequence of blocks scattered across a disk would impose significant costs on query evaluation. Second, no additional space is required for next-block pointers. Third, index update procedures must manage variable-length fragments that vary in size, however, the benefits of contiguity greatly outweighs these costs.

3.2 Generating More Sub-fingerprints With Up To *n* Toggled Bits

Haitsma and Kalker [6, 7, 8] assumed that there exists at least one sub-fingerprint in a query that can be matched perfectly to the correct song in a database. In their experiments, they insisted that about 17 out of the 256 sub-fingerprints in a query were error-free. However, the more noises or errors in a query, the more likely any hits to the correct song are not found.

In order to avoid situations that matching is failed due to some erroneous bits in the query fingerprint, we generate more $\sum_{i=1}^{n} \binom{16}{i}$ 16-bit fingerprints with up to n toggled bits from each original 16-bit sub-fingerprint for a song and index them in the vocabulary together with the original sub-fingerprint. This means that the amount of space requirement in the index increases with a factor of $\sum_{i=1}^{n} \binom{16}{i}$. However, we can achieve $O(1)$ lookup time to find the pointer to its own inverted lists without $\sum_{i=1}^{n} \binom{16}{i}$ times more fingerprint comparisons. Fig. 2 shows this construction in which there exist up to $\sum_{i=1}^{n} \binom{16}{i} + 1$ pointers to the inverted lists for a song.

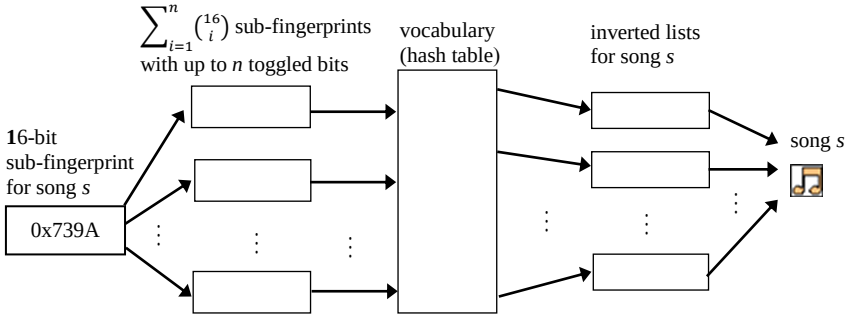

Fig. 2. Inverted file index structure that stores $\sum_{i=1}^{n} \binom{16}{i}$ sub-fingerprints with up to n toggled bits together with the original 16-bit sub-fingerprint for song s in a vocabulary

3.3 Indexed Search

The query evaluation process is completed in two stages. The first stage is a *coarse index search* that uses the inverted file index to identify candidates that are likely to contain song matches. The second stage is a *fine database search* that fetches each candidate's full fingerprint from a song database and then computes the similarity between the query fingerprint and the song's fingerprint fetched. The fine search is more computationally expensive because it requires the random disk access. Therefore, our strategy is to avoid the expensive random disk accesses as possible as we can.

To conduct the coarse index search, we use a ranking structure called *accumulator*. The accumulator records the following:

- the accumulated occurrences of the matched song identifiers (IDs),
- the matching positions both within the query and the matched song IDs, and
- the accumulated number of matchings that have the same relative offset between the matching positions within the query and the retrieved song IDs when there are multiple matchings. We call this *offset-match-count*.

Ultimately, the information what we need is the offset-match-count for every candidate of a search. If a specific song ID has been encountered and its offset-match-count has reached a certain threshold, then we load the full fingerprint from database using the retrieved song ID. The subsequent comparison is based on the Hamming distance between the query fingerprint and the song fingerprint on their matching positions. Computing the Hamming distance involves counting the bits that differ between two binary vectors.

In practical applications, many search candidates that have a single match or even multiple matches with query sub-fingerprints are generated although they are not the correct object what we seek for. Therefore, a significant percentage of the database needs to be searched if the search algorithm loads the full fingerprint of a candidate as soon as it encounters the candidate whose sub-fingerprint matches a certain query sub-fingerprint. In other words, the search strategy such as Haitsma-Kalker's method based on the single match principle is inevitably inefficient in disk-based applications although this problem may be less evident if all data are resident in memory. In fact, the candidate whose offset-match-count has reached a certain threshold (e.g. 3) has the great possibility of being the correct object what we seek for and there are almost no candidates that have their offset-matching-count reaching the threshold while they are not the correct answer. If two fingerprints are similar and there are multiple occurrences of sub-fingerprint matching between them, they may share the same relative offsets among the occurrence positions of matching.

The search algorithm using an inverted file index is illustrated in Fig. 3 and described in Fig. 4. There are six cost components in the fingerprint search, as summarized in Table 1. The first one is to initialize the array *accumulator* that records the accumulated number of matchings that have the same relative offset between the matching positions of the query and the retrieved song IDs. The second one is to compute *n* sub-fingerprints from a query song clip.These two operations are computed very fast. The third one is to retrieve the index information about songs stored in the inverted file index. This I/O operation is fast because the index information is lightweight and several inverted lists can be read in a single operation since they are contiguously

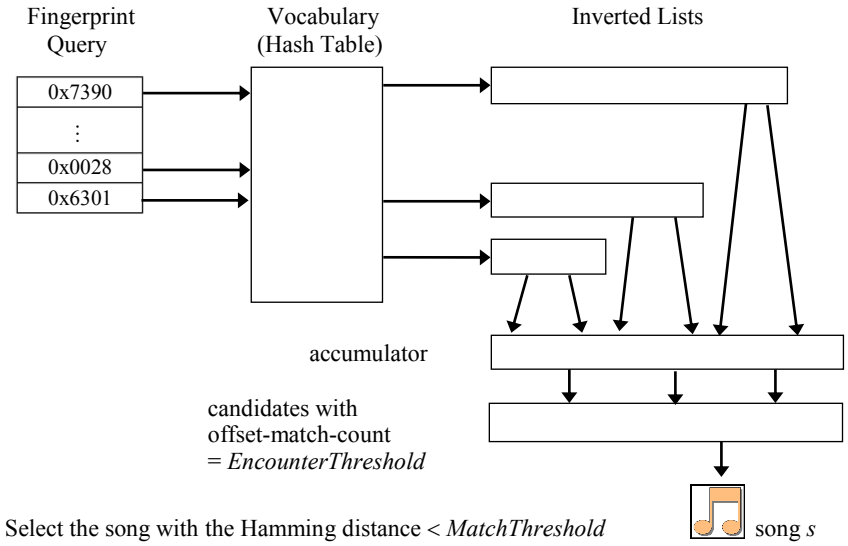

Fig. 3. Using an inverted file index and an accumulator to calculate song's similarity score

[Algorithm] Fingerprint Search
Input: Query song clip Q
Output: Match song P or 'No match'

```
1: Initialize accumulator A[j], 1 ≤ j ≤ |DB|, for each song  j in a database (DB).
2: Compute n sub-fingerprints t1, t2, ..., tn (e.g. n = 200) from a query Q.
3: for i = 1 to n do {
4:         IDs ← InvertedFileIndex[ti ]                    // coarse index search
5:         for j ∈ IDs do {
6:                 Increment occurrences of j in A[j] by 1.
7:                 Compute the offset-matching-count c of j in A[j].
8:                 if c = EncounterThreshold then {
9:                         P ← DB[j]                       // fine database search
10:                        if HammingDistance(P, Q) < MatchThreshold then
11:                                Return P;               // early termination
12:                }
13:        }
14: }
15: Return 'No match'
```

Fig. 4. Indexed computation of similarity between a query Q and a fingerprint database

stored in a disk. The fourth one is to count the accumulated occurrences of the IDs and their offset-matching-counts. This simply involves read/write accesses to the memory

Table 1. Cost Components in Fingerprint Search

No	Components	Computation	Operation	Cost
1	Initialize accumulator	$O(1)$	memory	very fast
2	Compute n sub-fingerprints	$O(1)$	memory	fast
3	Retrieve song IDs	$O(n)$	disk	lightweight
4	Count occurrences	$O(n)$	memory	very fast
5	Load full fingerprint	$O(n)$	disk	heavyweight
6	Compute Hamming Distance	$O(n)$	memory	fast

array. The fifth operation is to fetch the full fingerprints of the candidate songs. This is the most expensive I/O operation that includes the random disk accesses to the candidate song's fingerprint database. The final one is to fully compare the retrieved candidate fingerprint with the query fingerprint and this operation is also computed fast. Therefore, our strategy is to make the best use of the fast operations 3 and 4 while avoiding the most expensive operation 5 and the operation 6.

In the search algorithm, the query sub-fingerprints are generated at a time. Initially each song has a similarity of zero to the query, this is represented by creating the array accumulator A initialized by zero and the counts of song occurrences and offset-matching-count of $A[j]$ are increased by matching of song j and its matching of position offset to those of the query, respectively. Contrary to Haitsma-Kalker's method that fetches the full song fingerprint from a database and compares it with query as soon as it finds a song matched to a query, our search algorithm postpones it until the offset-matching-count of the candidate reaches a certain threshold (*EncounterThreshold* in the algorithm of Fig. 4) in order to avoid the expensive operations 5 and 6 in Table 1 as possible as it can. Based on our experimental results, *EncounterThreshold* = 3 shows a suitable trade-off between speed and accuracy. Without considered the offset matching principle and the multiple matching principle, the search algorithm would be similar to Haitsma-Kalker's method.

Besides the "multiple matching principle" and the "offset matching principle", another contribution of our search algortihm to the speedup is the "early termination strategy" shown in steps 10 and 11 in the algorithm of Fig. 4. The fewer errors in a query, the more likely the match is found at an early stage. Even before the full search of n (e.g., n = 200) sub-fingerprints is completed, if a song's offset-matching-count meets the condition of *EncounterThreshold*, then the the song is fully compared with query and it can be reported as a match if its Hamming distance to the query is less than a certain threshold (*MatchThreshold* in the algorithm of Fig. 4). This early search termination also contributes to the speedup.

4 Experimental Results

In order to evaluate our indexing scheme and search algorithm, we digitized 2,000 songs and computed about 4,000,000 sub-fingerprints from the songs. 1,000 queries were generated by randomly selecting 20-second portions from the song database and playing them through inexpensive speakers attached to a PC. These song snippets were then digitized to be used as queries using an inexpensive microphone. For each query, we know the answer, i.e. correct song, because we know which song each query is derived from. Therefore, we can compute the error rate that could not identify the correct song. In addition, we generated 100 queries from songs not stored in a database to evaluate the performance when a search returns no match.

We compare our method with Haitsma-Kalker's method to assess the performance of ours. In our experiment, we set *EncounterThreshold* to 3 and *MatchThreshold* to 0.2 in the search algorithm of Fig. 4. In addition, we also generated more 16-bit sub-fingerprints with up to 2 toggled bits from each original 16-bit sub-fingerprint for a song and stored them in the index vocabulary together with the original. Therefore, our index maintains 136 ($= {}_{16}C_1 + {}_{16}C_2$) times more index entries compared with other indexing methods such as Haitsma-Kalker's method.

We define the *search size* as the percentage of the songs' full fingerprints retrieved for comparison to the whole database size. The search size in our algorithm is in effect only one, in other words, we retrieve the full fingerprint of a song only if it is the correct song. This is due to the "multiple matching principle" and the "offset matching principle". Of between them, the offset matching principle makes the greatest contribution to the very small search size. On the other hand, Haitsma-Kalker's method could not achieve this performance.

First, we study the performance when a search returns no match, hence has no early termination. We conducted the experiment using the songs with no match in the database. In our algorithm, there is no case to load any candidate song's full fingerprint from the database when a match is not found. This is mainly due to the offset match principle because there is almost not the case that the offsets of match positions of sub-fingerprints are also matched although some sub-fingerprints themselves may be matched accidentally when two songs are not actually similar.

However, in Haitsma-Kalker's case, a significant percentage of the database needs to be searched even though there is no match, i.e. its search size approaches the database size. This inefficiency is caused by their method that retrieves the candidate's full fingerprint from the database as soon as it finds a single sub-fingerprint match.

Table 2 reports the proportions of candidate songs in a database that have any matches to the sub-fingerprints in a query even though the songs do not have actual match to the query fingerprint. It means that Haitsma-Kalker's method searches 85% (number of sub-fingerprint matches = 1 in Table 2) of the database when no match is found because it retrieves the candidate song's full fingerprint as soon as it finds the sub-fingerprint matched to a portion of query but does not succeed in finding actual song match.

Second, let us study the performance when a search returns a match, hence it has early termination. Even in this case, Haitsma-Kalker's method searches a significant percentage ($\approx$ 18%) of the database in our experiment. It is particularly inefficient to retrieve sporadically dispersered data from disk. Fingerprints in a database are actuallly

Table 2. Proportions of candidates that have sub-fingerprint matches to a query

number of sub-fingerprint matches	≥ 20	≥ 10	≥ 8	≥ 6	≥ 4	≥ 2	= 1
proportions of candidates	3.8%	22%	32%	45%	54%	77%	85%

Table 3. Comparison between our algorithm and Haitsma-Kalker's algorithm

Algorithm	False dismissal rate	Search size	Average search time
Our algorithm	0.74%	0.05%	30 msec
Haitsma-Kalker's algorithm	0.95%	18%	3120 msec

retrieved randomly and it is very costly contrary to the access of inverted lists of index. On the other hand, our method retrieves only the correct song's full fingerprint. This is also due to the offset match principle of our algorithm.

Third, let us consider the false dismissal rate in the search. In our algorithm, false dismissal is occurred when offset-match-counts of candidates are less than *Encounter-Threshold*. Although we adopt the offset match principle to avoid the expensive operation of retrieving full fingerprints, the false dismissal rate of our algorithm is less than 1% (Table 3). In effect, the fewer error bits in a query, the more likely the false dismissal error is reduced.

Finally, in the speed test, our method is far faster than Haitsma-Kalker's method. This speedup is achieved since our search algorithm checks only the correct song, while Haitsma-Kalker's method has to load and compare a significant percentage of the database. This is due to employing the strategy of postponing the access of database to fetch the full fingerprint of a song as well as the early termination strategy without considering all sub-fingerprints. The above experimental results show both of the effectiveness and efficiency of our indexing and search strategy. Table 3 summarizes the experimental results for both our algorithm and Haitsma-Kalker's algorithm.

5 Conclusion

In this paper, we propose a new search algorithm based on inverted indexing for efficiently searching a large high dimensional binary database. The indexing method employs the inverted file as the underlying index structure and adopts the strategy of maintaining duplicated fingerprints with toggled bits, so that it reduces the song recognition error rate and achieves the efficient song retrieval.

The search algorithm adopts the "multiple match principle", the "offset match principle", and the "early termination strategy", so that it postpones the fetch of full finger-

prints and therefore it reduces the number of expensive random disk acceses dramatically.

The experimental result shows the performance superiority of our method to Haitsma-Kalker's. This makes our new indexing and search strategy a useful technique for efficient high dimensional binary database searches including the application of song retrieval.

References

1. Seo, J.S. et al.: Audio Fingerprinting Based on Normalized Spectral Subband Moments, *IEEE Signal Processing Letters*, vol. 13, no. 4 (2006) 209-212.
2. Cha, G.-H., Zhu, X., Petkovic, D., and Chung, C.-W.: An efficient indexing method for nearest neighbor searches in high-dirnensional image databases, *IEEE Tr. on Multimedia*, vol. 4, no. 1 (2002) 76–87.
3. Gionis, A., Indyk, P., and Motwani, R.: Similarity Search in High Dimensions Via Hashing, *Proc. VLDB Conf.* (1999) 518–529.
4. Zezula, P., Amato, G., Dohnal, V. and Batko, M.: *Similarity Search: The Metric Space Approach*, Springer (2006)
5. Aggarwal, C.C. and Yu, P. S.: On Indexing High Dimensional Data with Uncertainty, *Proc. SIAM Data Mining Conference*. (2008) 621-631
6. Haitsma, J. and Kalker, T.: A Highly Robust Audio Fingerprinting System With an Efficient Search Strategy, *J. New Music Research*, vol. 32, no. 2 (2003) 211–221.
7. Haitsma, J. and Kalker, T.: Highly Robust Audio Fingerprinting System, *Proc. Int. Symp. on Music Information Retrieval* (2002) 107–115.
8. Oostveen, J., Kalker, T. and Haitsma, J.: Feature Extraction and a Database Strategy for Video Fingerprinting, *Proc. Int'l Conf. on Visual Information Systems*, vol. LNCS 2314 (2002) 117–128.
9. Miller, M.L., Rodriguez, M.C. and Cox, I.J.: Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces, *J. VLSI Signal Processing*, vol. 41 (2005) 285–291.
10. Miller, M.L., Rodriguez, M.C. and Cox, I.J.: Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces, *Proc. IEEE Multimedia Signal Processing Workshop* (2002) 182–185.
11. Brin. S and Page, L.: The anatomy of a large-scale hypertextual Web search engine, *Computer Networks and ISDN Systems*, vol. 30 (1998) 107–117.