CHAPTER 1

INTRODUCTION

To develop an application that records audio content of a video using microphone and recognises the video id and its metadata using audio fingerprinting and indexing. The audio content of the video will be recorded for few seconds and the result can be displayed to the user.

People spend about 2 hours average per day on watching video content. Most of the time people are unaware about the video metadata. An average of 2 billion downloads occur for content recognition applications. Content recognition needs to be improvised and new innovations must be brought, such that they provide ease of access. Our application takes a simple problem statement of video recognition and identification that solves most of the real time complications on identifying meta data and provides ease of access for information gain.

1.1 AUDIO FINGERPRINTING

Audio fingerprinting is used to take a short sample of an unknown audio recording and retrieve meta information about the recording. It does this by converting the data rich audio signal into a series of short numerical values (fingerprints) that aim to uniquely identify a musical recording. Audio fingerprinting systems keep large databases of fingerprints for millions of known audio recordings.

To identify an unknown audio recording query, the query's fingerprint is generated and compared to the reference database to find recordings that have identical or similar fingerprints[7]. Unlike symbolic music notation query methods, such as query by-contour and query-by-humming (Ghias et al. 1995), which use symbolic musical information (i.e., knowledge of the instruments and specific notes played in a segment of audio), audio fingerprinting uses lower-level spectral information in a signal to generate a unique identifier of the audio.

An audio fingerprinting system should be able to recognise audios of songs in the same way that a person can. If a person can recognise a song from a short clip of audio comes then an ideal computer system should also be able to recognise the song. This means that a fingerprint algorithm should utilise the perceptual aspects of the audio contained in the file and not just the way that the file itself is encoded digitally. These so-called content-based identification systems (CBIDs) are named as such because they identify matches based on the musical content of the recordings in the corpus rather than the way that the file is stored by the computer. Furthermore, because humans are able to recognise a recording as being the same even in the presence of small changes to the query, such as tempo variations or noise, effective fingerprinting algorithms should also be able to correctly identify queries that have been modified in similar ways. Haitsma et al. (2001) call fingerprinting "robust hashing", because it indicates that a fingerprinting algorithm should generate a fingerprint based on the input that is robust to modifications to the audio that do not dramatically alter the sound[5]. They suggest that one approach to perform this robust hashing is to design an algorithm that approximates the human auditory system.

A robust audio fingerprint is a function that associates to every basic time unit of audio content a short semi-unique bit-sequence that is continuous with respect to content similarity as perceived by the [human auditory system].

In order to match a query to a recording in a reference database, a fingerprinting algorithm must generate identical fingerprints for the reference recording and query. The hashes must be identical even when the query is very short, or when given a recording that sounds similar to a human, but has a different audio signal. If the recording has been made, for example, with a microphone in a noisy room then the fingerprinting algorithm should be able to separate the music in the recording from any additional noise recorded by the microphone.

Reference databases of audio should contain fingerprint codes for the entire duration of audio. This is because it is also useful to identify a song when only a portion of the song is given as a query. This segment could be recorded from any

point in the song, especially if the recording is made in a public place from music that is being played over PA system.

On the other hand, fingerprint lookup systems do not need to store a copy of the audio that is used to create the fingerprint. Fingerprints can be generated and submitted to a database by any person with a copy of the audio. While commercial fingerprinting systems can get fingerprints directly from the music distributors, it is also possible to obtain fingerprints for rare music, out of print music, or music released through independent labels directly from people who have copies of the audio[11].

Audio watermarking is the act of inserting hidden information into an audio stream, that cannot be detected by the human ear. It can be used for many of the same uses as fingerprinting, for example, discovering the copyright status of a song, or providing metadata for a given recording. Additional metadata can be included in a watermark, including both information about the work or artist, or additional information (e.g., news broadcast from a radio station along with a song). Audio watermarking techniques are able to store data in an audio stream at rates of up to 150kbps, providing enough space to include metadata about the currently playing song, or even artwork.

1.2 REQUIREMENT OF FINGER PRINTING ALGORITHM

The primary purpose of an audio fingerprinting system is to identify metadata about a song based on a short segment of the audio signal. Haitsma et al. (2001) and Cano et al. (2005) each describe a list of desirable retrieval criteria that a fingerprinting system should fulfil:

- A system must be able to generate compact fingerprints that can be quickly located in a reference database, with a low error rate. The reference database must able to be updated easily with fingerprints of newly released or digitized audio.

• Fingerprinting systems must be able to identify audio even if it has been altered in different ways. Some common alterations that a system should be able to recover from include ambient noise (for example, if a recording was made in a public place or while driving in a car), and transmission interference (a reduction in quality or increase in noise due to the medium which the audio is transmitted through, for example, FM radio or GSM cell phone networks).

• Recordings should be able to be identified even if the audio has been modified before it was broadcast. For example, frequency equalization, or audio compression may be applied by radio stations or a home stereo system. Resampling of audio results in a speed-up or slowdown of the audio which results in an associated increase or decrease in the pitch of the signal.

• A fingerprinting algorithm should be reliable. It should minimize the number of false positives returned to any query (ideally false positives should be zero). It should be robust enough to detect audio correctly even if the query has been distorted during recording.

• A fingerprinting algorithm should be granular, that is, able to correctly identify a song from just a small segment of audio. This segment could come from any part of the song, not just the beginning

• The fingerprint should be computationally inexpensive to generate, and adding new songs to the database should not result in any perceptible decrease in the speed at which lookups can be performed.

1.3 THE AUDIO FINGERPRINTING PROCESS

Most audio fingerprinting algorithms follow a common sequence of steps when transforming an audio signal to a fingerprint (Figure 1.1) : pre-processing, framing and overlap, transform and analysis, feature extraction, and fingerprint generation (Cano 2007).

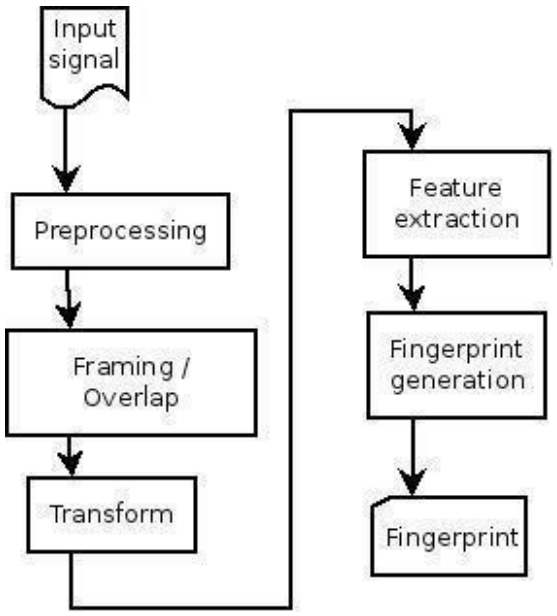

Figure 1.1 Common steps perform in audio fingerprinting algorithms to convert audio to a fingerprint.

The first step, pre-processing, converts all input signals into a common format for analysis by the algorithm. Often, this step involves converting the input to a mono signal and lowering the sampling rate from the standard CD rate of 44,100 Hz. The exact process differs for different algorithms. Next, an algorithm takes the time-series audio signal

The main target of the algorithm for audio fingerprint extraction is the ability to standardize the content of music/audio. A good audio fingerprint algorithm extraction can represent the music/audio from a complex domain of waveform into a new feature-based domain that can support comparing the similarity and difference of two songs/audio from every particular part of its audio fingerprint. In this implementation, we use panako as the audio fingerprint extraction algorithm for the detection of the distorted audio from the original audio file. We inherited the advantages of the panako algorithm for extracting the audio fingerprint and focusing on proposing a new intelligent storage of fingerprints for the computer using multiple GPGPUs and having to accommodate the parallel nearest problem algorithm that can support to handle numerous queries at the same time. Audio fingerprint is a digital vector that

was extracted from the audio/song waveform and able to standardize the content of the audio/song source. Audio fingerprint can easily help for comparing the similarities and differences of songs. In addition, using audio fingerprint for storing can reduce the size of original audio/song with the standard structure. In our system's database, instead of storage the real waveforms of the songs we considered storage of the audio fingerprints and its metainformation for every song/track and converts it into a frequency-domain signal from which more information can be extracted. Framing and overlap determines how many samples to consider when calculating a transform of a time-domain signal. Each frame has a window applied to it to assist in calculations, and the frames are processed in overlapping chunks from the time-series signal. The feature extraction process takes the signal that has been converted into the frequency domain and selects salient features that are used to characterize the audio. Finally, once the features have been chosen and extracted from the signal, they need to be converted into a fingerprint representation that can be stored in a database and compared to unknown query signals.

Due to the rich feature set of digital audios, a central task in this process is that of extracting a representative audio fingerprint that describes the acoustic content of each song. 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 assume a representation of a song developed by Philips in which 3 second interval is represented by 256 subsequent 32-bit sub-fingerprints.

For a full fingerprinting process, the procedure does not end at the fingerprinting algorithm (Figure 1.2). Once the fingerprint has been generated, it must be stored in a reference database. The numerical representation of the fingerprint is usually too unwieldy to be used as an identifier, so a smaller unique identifier is used. This could be as simple as the artist and song name, or a short unique string. A fingerprinting system will provide a lookup service. This lookup operation should be able to take an unknown input query and match the query's fingerprint with a fingerprint that is in the reference database, returning the identifier of the song that the query best matches,

and optionally the location in the song that the query comes from. If the song being looked up is not in the reference database, the fingerprinting system should report that the song is not in the database, rather than giving a wrong answer.

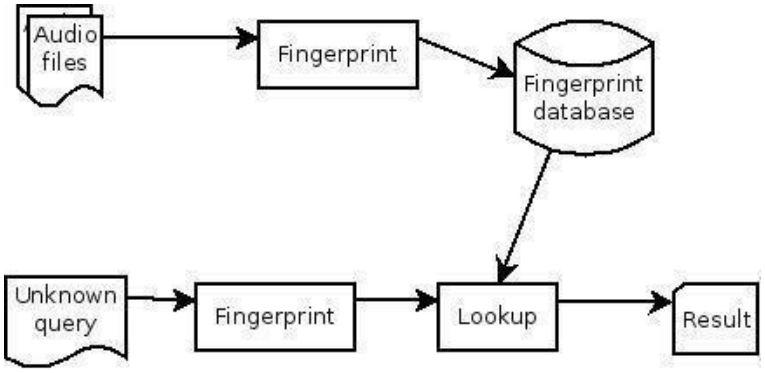

Figure 1.2 Audio fingerprinting process

1.4 FINGER PRINTING ALGORITHM USED IN EVALUATION

There are a large number of fingerprinting algorithms that have been developed. Different algorithms perform the fingerprinting and identification steps differently, and have different strengths in recognising types of music. For the evaluation in this implementation we chose fingerprinting algorithms that use different techniques to generate a fingerprint. We chose these specific algorithms for two reasons. The first reason is because they all use significantly different techniques for generating a fingerprint. The second reason is because each algorithm is freely available to download and run on a server[3]. By running our own version of the server we are able to carefully control the audio that is added to the database and so can tell if the result returned by the algorithm is correct or not. The algorithms are: Echoprint, Chromaprint, based on the algorithm presented by Ke, Hoiem, and Sukthankar (2005), and a landmark hashing algorithm (Ellis 2009), based on (Wang 2003). The algorithms are all used actively in commercial (Echoprint), community (Chromaprint), and research (Landmark) environments. Each fingerprinting algorithm has freely available source code for both the fingerprinting component and the lookup system, which we make use of in the evaluations.

1.5 CONTRIBUTION OF THIS IMPLEMENTATION

This implementation provides a survey of the current state of the art in audio fingerprinting algorithms. It gives an overview of many current algorithms and also provides an in-depth study of fingerprint algorithms that are currently in widespread use commercially and in research projects.

We perform an evaluation of the algorithms, and present statistics on the accuracy of the algorithms for performing fingerprint lookups on a large collection of music. The audio queries are made to the fingerprinting systems are modified to simulate different scenarios that may be encountered when performing a fingerprinting lookup, such as short queries, damaged audio, degraded audio, and audio mixed with a noisy environment. This broad evaluation directly compares different fingerprinting algorithms in identical situations. As part of the evaluation, we have developed an extensible evaluation suite. This suite allows fingerprinting algorithms to be tested repeatedly under the same circumstances and collects results of each evaluation and helps in generating statistics. This evaluation platform can be used by other researchers to test new fingerprinting algorithms in a controlled environment. In fingerprint search, the query fingerprint may not match any fingerprint in a database because the fingerprint extracted from a query may have bit errors due to distortions to a query song. This means that the fingerprint search must support imperfect matching or similarity search.

CHAPTER 2

LITERATURE SURVEY

2.1 INTRODUCTION

2.1.1 Toan Nguyen Mau & Yasushi Inoguchi, Audio fingerprint hierarchy searching strategies on GPGPU massively parallel computer suggested that, The main target of the algorithm for audio fingerprint extraction is the ability to standardize the content of music/audio. A good audio fingerprint algorithm extraction can rep-resent the music/audio from a complex domain of waveform into a new feature-based domain that can support comparing the similarity and difference of two songs/audio from every particular part of its audio fingerprint. In this paper, we use HiFP2.0 as the audio fingerprint extraction algorithm for the detection of the distorted audio from the original audio file. We inherited the advantages of the HiFP2.0 algorithm for extracting the audio fingerprint and focusing on proposing a new intelligent storage of fingerprints for the computer using multiple GPUs and having to accommodate the parallel nearest problem algorithm that can support to handle numerous queries at the same time cluster aware by query and the passing method for audio fingerprint query from one GPU device to another GPU device to get the best accuracy.

Audio fingerprint was developed for representing the audio based on the content of waveform. With the audio fingerprint database, we can easily manage the song/music with high reliability and flexibility. However, with the well-developed Internet of today, the audio data have become bigger and bigger which make the management of audio/music data more difficult. There are two problems that we need to solve when the audio fingerprint database turn into bigdata: the size of the database needs to be sufficient for storing 10 million of audio fingerprint and the strategies for searching the nearest song in acceptable time for thousands of queries at once [Nguyen Mau, T., & Inoguchi, Y. (2016). Audio fingerprint hierarchy searching on massively parallel with multi-gpus using K-modes and lsh. Eighth international

conference on knowledge and systems engineering (KSE) (pp. 49–54). IEEE]. In this research, we propose the methods for storing the audio fingerprint using multiple GPGPU and nearest song searching strategies based on these databases. We also showed that our methods have the significant result for deploying the real system in the future.[2].

2.1.2 Guang-Ho Cha, An Effective and Efficient Indexing Scheme for Audio Fingerprinting suggested that, Large digital music libraries are becoming popular on consumer computer systems, and with their growth our ability to automatically analyse and interpret their content has become increasingly important. The ability to find acoustically similar, or even duplicate, songs within a large audio 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, pro-gram verification and advertisement verification.

Due to the rich feature set of digital audios, a central task in this process is that of extracting a representative audio finger-print that describes the acoustic content of each song. 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 assume a representation of a song developed by Philipin which 3 second interval is represented by 256 subsequent 32-bit sub-fingerprints. Given this fingerprint representation, the focus of our work has been to develop an effective and efficient indexing and search method. This problem can be characterized as a nearest neighbour search in a very high dimensional.

The second problem is that Haitsma and Kalker make the assumption that under 'mild' signal degradations at least one of the computed 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. 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 to n , then the search time to find matching fingerprints in a DB is probably unacceptable.

Miller et al. [6, 7] 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 up-dates the best error rate to this new value and assigns this fingerprint as the best candidate nearest neighbor so far.

first problem of this 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 disk-based database search. Moreover, it assumes that each song is also represented by a fingerprint with 8192 bits, the same number of bits as the query fingerprint. But each actual song with an average length of 5 minutes has approximately 250 million sub-fingerprints. 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.

This indexing scheme employs the inverted file as the underlying structure and adopts the strategies of maintaining duplicated fingerprints with toggled bits,

postponing the fetch of full fingerprint, and the early termination principle. The experiment shows the performance superiority of our method to the Haitsma Kalker method with respect to accuracy and speed. This makes our new indexing scheme a useful technique for audio fingerprinting.[5][6].

2.1.3 Ce Yu, Member, IEEE, Runtao Wang, Jian Xiao and Jizhou Sun High Performance Indexing for Massive Audio Fingerprint Data suggested that, Realtime online audio searching systems require high performance indexing architecture for massive audio fingerprints. To improve the performance of hash table-based indexing for audio fingerprints, this paper designs and evaluates a hybrid data structure which combines linked list with vector to store the values in the hash table to balance the searching performance and the memory usage. To extend the hash table to cluster environment, three distribution patterns are designed and implemented, and experiments show that the content-oriented distribution pattern is better than the key oriented distribution pattern. The proposed serialized data layout of the hash table can further improve the searching performance with less memory usage. All the experiments are executed on practical massive data sets including up to 1,000,000 songs, and the results certificate the improvement of the methods proposed. Audio fingerprints are important for content-based audio searching. By extracting the audio fingerprints (digital features) from the inputted audio fragment and then comparing to existing audio fingerprints in database, the detailed information of that audio fragment can be retrieved. This searching method does not only avoid common problems of traditional keyword-based search, such as lack of tags or wrong tags, but also allows users to perform queries without inputting any keywords.

Audio fingerprints are used in various types of applications and services for consumer devices, of which real-time online music searching engine is an example. Users may search for a specific piece of music by uploading an audio fragment of someone humming it or it being played in background. Several algorithms of extracting and matching audio fingerprints have achieved significant results in

laboratories, and have been applied to products with relatively small datasets. But searching in large-scale data sets effectively and efficiently is still a challenge for both academia and industry.

Like flat text keywords or tags, the audio fingerprint is the special character description of the audio content. The audio fingerprints are generated from overlapped fragments of the audio signals. There are two main workflows in the audio fingerprint system, as illustrated in Fig. The processing of massive audio datasets includes: extracting the features of each audio file, modelling the fingerprints, and storing the fingerprints with proper index. The searching in the fingerprint dataset includes: extracting the features of the inputted audio file, modelling the fingerprint, and searching in the indexed dataset to find matching record(s).

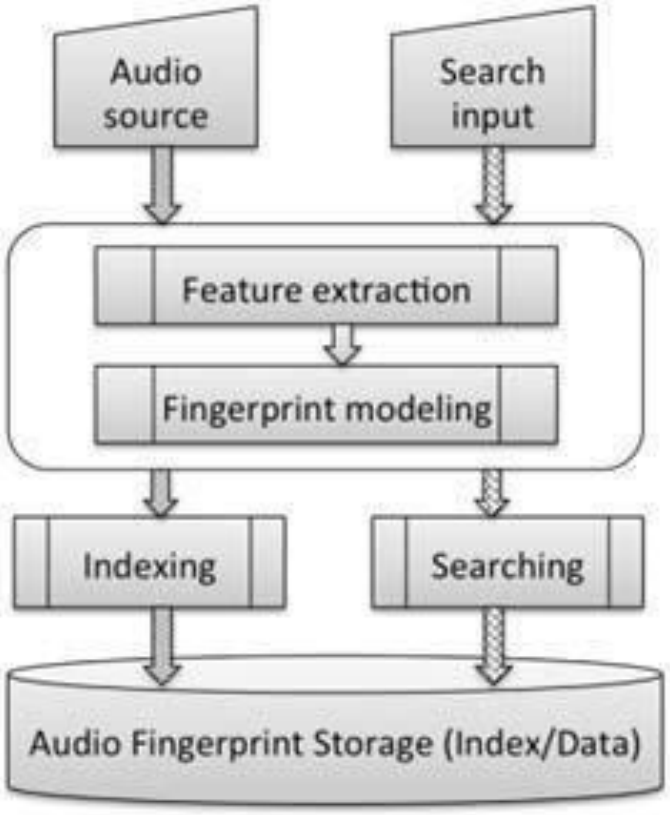

Figure 2.3.1. Workflows in a typical audio fingerprint system.

For online searching applications (i.e. online music search by simply humming a part of a song), real-time response is necessary. The music library will have more than 1,000,000 songs. The size of the audio fingerprint data of such an online audio

searching system are beyond the memory capacity of a single server, and the increasing amount of audio fingerprint data will cause longer response time for queries. Moreover, large number of concurrent queries will get even worse performance. Thus, the backend server environment needs to be extended to a cluster environment to meet the performance requirements.

To extend the audio fingerprint library to multiple nodes in the cluster, the hash table has to be divided into multiple servers. Considering the characters of music audio fingerprint system and the data structure of the LLoV-based hash table, two types of distributed hash table patterns are designed: key-oriented distribution and content-oriented distribution.[4].

2.1.4 Alastiar Porter Evaluating musical fingerprinting systems suggested that, Recorded music is pervasive in our society. It is broadcast over radio waves and on the Internet. It is used as a background to videos, television, and movies. Shops play it in the background while people are shopping. People carry around thousands of songs every day on portable audio players and smartphones, with access to millions more by streaming from the Internet. Personal music players can show the name of the recording that is currently playing, but when listening to music in public environments it can be a challenge to recognise every song that you listen to in a day. It is an impressive skill to be able to listen to a small clip of music and recognise almost immediately the title, composer, or performer of the work. Sometimes though, you may not recognise the song, or it might be familiar and on the tip of your tongue, but you just can't remember the name. You may be in a cafe or shop as a song that you want to know more information about comes on, but it is not introduced or you miss the DJ's introduction. Given the ever increasing amount of music written, recorded, and performed it is virtually impossible for any one person to recognise every song. It seems suitable to delegate such a task to a computer.Computers have been used to develop music-related applications since the time that they were first available in research environments (Downie 2003). Music information retrieval (MIR), a music-specific branch of information retrieval, is concerned with using

computers to store and analyse digital collections of music in all forms (e.g., sheet music, recorded music, metadata about music) and allow queries to obtain analyses and results from them. Audio fingerprinting is a facet of MIR that uses computers to listen to and recognise recordings of songs, much like people can listen to a piece of music and say what the name of the song is. In the audio fingerprinting process, a computer algorithm is used to analyse a corpus of music, identifying features that can be used to uniquely identify musical works in a corpus (an audio "fingerprint", much like fingerprints uniquely identify people). Once a corpus of audio fingerprints has been created, the same algorithm can be used to generate a fingerprint of an unknown clip of audio. The corpus can be searched for fingerprints that are the same as the fingerprint generated by the unknown query in order to retrieve information about the recording.

Audio fingerprinting is used to take a short sample of an unknown audio recording and retrieve metadata about the recording. It does this by converting the data-rich audio signal into a series of short numerical values (or *hashes*) that aim to uniquely identify a musical recording. Audio fingerprinting systems keep large databases of fingerprints for millions of known audio recordings. To identify an unknown audio recording query, the query's fingerprint is generated and compared to the reference database to find recordings that have identical or similar fingerprint hashes. Unlike symbolic music notation query methods, such as query-by-contour and query-by humming (Ghias et al. 1995), which use symbolic musical information (i.e., knowledge of the instruments and specific notes played in a segment of audio), audio fingerprinting uses lower-level spectral information in a signal to generate a unique identifier of the audio.

There are a large number of fingerprinting algorithms that have been developed. Different algorithms perform the fingerprinting and identification steps differently, and have different strengths in recognising types of music. For the evaluation in this implementation we chose an algorithm that uses a technique to generate fingerprints. We chose these specific algorithms for two reasons. The first reason is because they all use significantly different techniques for generating a fingerprint. The second

reason is because each algorithm is freely available to download and run on a server. By running our own version of the server, we are able to carefully control the audio that is added to the database and so can tell if the result returned by the algorithm is correct or not. The algorithms are: Echoprint (Ellis et al. 2011), Chromaprint (Lalinsky´ 2012), based on the algorithm presented by Ke, Hoiem, and Sukthankar (2005), and a landmark hashing algorithm (Ellis 2009), based on (Wang 2003). The algorithms are all used actively in commercial (Echoprint), community (Chromaprint), and research (Landmark) environments. Each fingerprinting algorithm has freely available source code for both the fingerprinting component and the lookup system, which we make use of in the evaluations.[11].

2.1.5 Toan Nguyen Mau & Yasushi Inoguchi Audio fingerprint hierarchy searching strategies on GPGPU massively parallel computer suggested that, The main target of the algorithm for audio fingerprint extraction is the ability to standardize the content of music/audio. A good audio fingerprint algorithm extraction can represent the music/audio from a complex domain of waveform into a new feature-based domain that can support comparing the similarity and difference of two songs/audio from every particular part of its audio fingerprint. In this paper, we use HiFP2.0 as the audio fingerprint extraction algorithm for the detection of the distorted audio from the original audio file. We inherited the advantages of the HiFP2.0 algorithm for extracting the audio fingerprint and focussing on proposing a new intelligent storage of fingerprints for the computer using multiple GPGPUs and having to accommodate the parallel nearest problem algorithm that can support to handle numerous queries at the same time.

The audio fingerprint database F includes a great number of points (also known as audio fingerprint) n: F_1, F_2, F_3,..., F_n. The metainformation for all original audio fingerprints in F is given as well. We implement to search the nearest audio fingerprint in F and use its metainformation for the corresponding query with the goal of showing the metainformation of multiple unknown metainformation queries (audio

fingerprints) F_{q1}, F_{p2}, F_{q3},..., q_T . There are two requirements in this research: the average searching time needs to be below the threshold (1 s) for big database (10million fingerprint database) and the need to support process multiple queries at the same time in parallel processing. There are two main problems when doing this searching system: intelligent distributed storage for multiple devices and strategies for parallel searching among two levels with multiple sub-databases

Audio fingerprint is a digital vector that was extracted from the audio/song waveform and able to standardize the content of the audio/song source. Audio fingerprint can easily help for comparing the similarities and differences of songs. In addition, using audio fingerprint for storing can reduce the size of original audio/song with the standard structure. In our system's database, instead of storage the real waveforms of the songs we considered storage of the audio fingerprints and its metainformation for every song/track.

The authors propose to use two-level hashing to reduce the number of hash construction instead of using many pointers to indicate the address of every bucket. Besides that, partitioning the database can help store the big hash table on several nodes; so the PLSH can work in nodes with small memory. Another strength of PLSH is parallel querying on multiple nodes, each node holds an independent part of the database; so they can search at the same time before sending the result to coordinator. With these advantages, PLSH can speed up the inserting and query stage of Tweet system to 1.5X.[7][2].

2.2 EXISTING SYSTEM

'Shazam' is a free application that helps users figure out the name of a catchy song being played on the radio, television or other place. Users simply hold their iPhone device up to the speaker playing the music and the Shazam application will attempt to identify the tune's album, artist and song title.

Shazam works by analysing the captured sound and seeking a match based on an acoustic fingerprint in a database of more than 11 million songs. A spectrogram of the sound of a violin. The target zone of a song scanned by Shazam. The user tags a

song for 10 seconds and the application creates an audio fingerprint. The working of the shazam application is described in the figure 2.6.

For a fingerprinting algorithm we need a good frequency resolution (like 10.7Hz) to reduce spectrum leakage and have a good idea of the most important notes played inside the song. At the same time, we need to reduce the computation time as far as possible and therefore use the lowest possible window size.

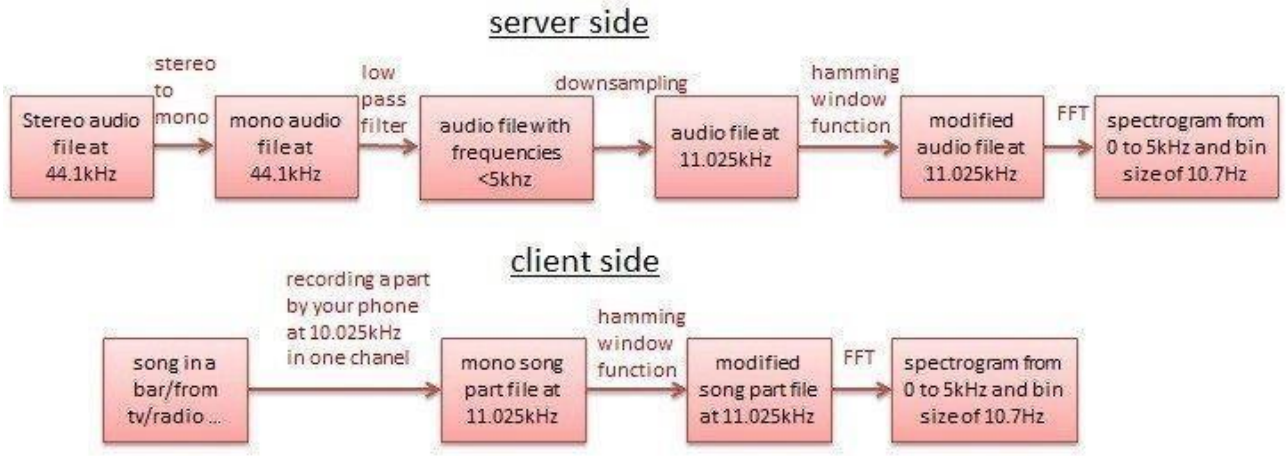

Figure 2.6. background process of shazam.

On the server side (Shazam), the 44.1khz sampled sound (from CD, MP3 or whatever sound format) needs to pass from stereo to mono. It is possible to do that by taking the average of the left speaker and the right one. Before downsampling, Then it need to filter the frequencies above 5kHz to avoid aliasing. Then, the sound can be downsampled at 11.025kHz.

On the client side (phone), the sampling rate of the microphone that records the sound needs to be at 11.025 kHz. Shazam identifies songs based on an audio fingerprint based on a time-frequency graph called a spectrogram. It uses a smartphone or computer's built-in microphone to gather a brief sample of audio being played. Shazam stores a catalogue of audio fingerprints in a database. The user tags a song for 10 seconds and the application creates an audio fingerprint. Shazam works

by analyzing the captured sound and seeking a match based on an acoustic fingerprint in a database of more than 11 million songs. If it finds a match, it sends information such as the artist, song title, and album back to the user. Some implementations of Shazam incorporate relevant links to services such as iTunes, Spotify, YouTube, or Groove Music. If Shazam cannot find a match, it returns a "song not known" dialogue.

CHAPTER 3
SYSTEM SPECIFICATION

3.1 SOFTWARE REQUIREMENTS

3.1.1 Android Studio

Android Studio is Android's official IDE. It is purpose-built for Android to accelerate android application development and help to build the highest-quality apps for every Android device. Android Studio's Instant Run feature pushes code and resource changes to your running app. It intelligently understands the changes and often delivers them without restarting your app or rebuilding your APK, so you can see the effects immediately.

The code editor helps to write better code, work faster, and be more productive by offering advanced code completion, refactoring, and code analysis. As you type, Android Studio provides suggestions in a dropdown list. Simply press Tab to insert the code. The Android Emulator installs and starts apps faster than a real device and allows to prototype and test app on various Android device configurations: phones, tablets, Android Wear, and Android TV devices. User can also simulate a variety of hardware features such as GPS location, network latency, motion sensors, and multitouch input. The Android Emulator installs and starts apps faster than a real device and allows to prototype and test app on various Android device configurations: phones, tablets, Android Wear, and Android TV devices. User can also simulate a variety of hardware features such as GPS location, network latency, motion sensors, and multi-touch input.

3.1.2 ACR Cloud Libraries

ACR cloud is an online service which provides libraries for the content recognition. Which also provides the Application Program Interface as well as the ACR Bucket (database) for the recognition of custom audios. This database can be

used for the storage of custom audios and then convert it in to finger prints. Users can add contents they want to recognize into their own buckets by uploading files directly or sending only fingerprints through console APIs.

ACR technology helps audiences easily retrieve information about the content they watched. For smart TVs and applications with ACR technology embedded the audience can check the name of the song which is played or descriptions of the movie they watched. In addition to that, the identified video and music content can be linked to internet content providers for on-demand viewing, third parties for additional background information, or complementary media.

The fingerprinting algorithm processes the signal of audios and extracts digital features called fingerprints for each audio. Fingerprints are very discriminative so the system can use them to identify the audio they belongs to. Fingerprints are also robust which means they can resist the environment noise and this make it possible to identify recorded audios in rather noisy environments. If the system finds matched fingerprints of the query snippet, it can determine the most like audio in database and gives the position of the snippet in the source audio.

3.1.3 Adobe Photoshop

Photoshop is Adobe's photo editing, image creation and graphic design software. The software provides many image editing features for raster (pixelbased) images as well as vector graphics. It uses a layer-based editing system that enables image creation and altering with multiple overlays that support transparency. Layers can also act as masks or filters, altering underlying colours.

Photoshop's naming scheme was initially based on version numbers. However, in October 2002, following the introduction of Creative Suite branding, each new version of Photoshop was designated with "CS" plus a number. e.g., the eighth major version of Photoshop was Photoshop CS and the ninth major version was Photoshop CS2. Photoshop CS3 through CS6 were also distributed in two different editions: Standard and Extended. In June 2013, with the introduction of Creative Cloud

branding, Photoshop's licensing scheme was changed to that of software as a service rental model and the "CS" suffixes were replaced with "CC". Historically, Photoshop was bundled with additional software such as Adobe ImageReady, Adobe Fireworks, Adobe Bridge, Adobe Device Central and Adobe Camera RAW.

3.2 HARDWARE REQUIREMENTS

Duel core processor and at least of 2GB RAM is needed for the efficient working of the Android studio. Hard Disk of minimum size 160 GB can be used for the storage of the source files. Mobile devices with Android operating system is used to often check the functioning of the application.

CHAPTER 4

IMPLEMENTATION

4.1 PROPOSED SYSTEM

User have to play the audio content of a video from any sources and the meta data about the audio content will be displayed on the screen. This have to be specially build for the TV series and also the custom audios that user can make and upload. The audio content from different sources as well as the the custom uploaded videos will be recognised and the meta data information will be displayed for the user.

This provides better way to identify custom audios using fingerprinting algorithms. It gives a faster details about the unknown audios which we hear from different sources. Audio fingerprints are important for content-based audio searching. By extracting the audio fingerprints (digital features) from the inputted audio fragment and then comparing to existing audio fingerprints in database, the detailed information of that audio fragment can be retrieved. This searching method does not only avoid common problems of traditional keyword-based search, such as lack of tags or wrong tags, but also allows users to perform queries without inputting any keywords.

The audio fingerprints are the compressed description of contents of the audio. The fingerprints are generated by a function which maps the audio wave data to the fingerprint data. Most audio fingerprint matching algorithms are based on the distance calculations such as Hamming distance and Euclidean distance. As for an audio fingerprint system, the search problem is basically matching the input audio fingerprint with billions of archived fingerprints. The efficiency and accuracy of an audio search system depend on both its definition of the audio fingerprint and data storage architecture. Here for the searching API itself use a method to extract the exact data from the database.

4.2 SYSTEM ARCHITECTURE

The architecture diagram of the proposed system is designed and shown in the figure 4.2

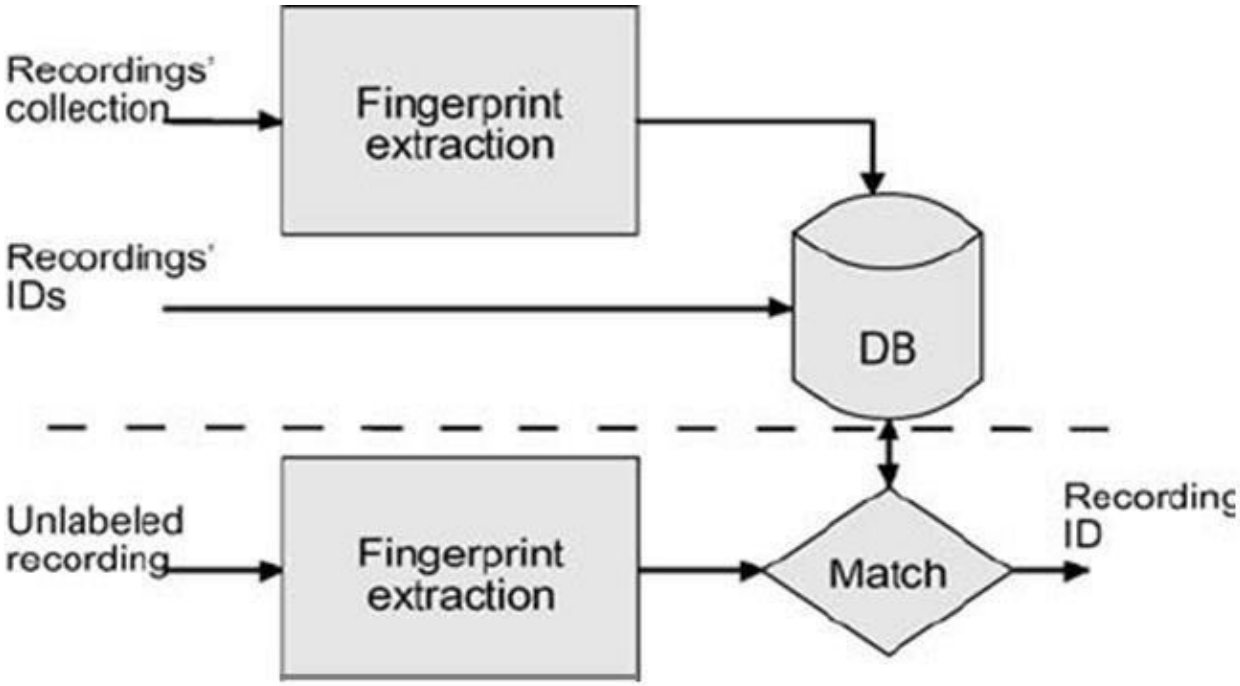

Figure 4.2 System Architecture

Here the YouTube or any other audio or video sources can be taken as Recordings collection. The fingerprint of these recorded collections will be saved on the database. It can be access by using API. And the custom videos are recorded and uploaded in the database further the fingerprints will be generated.

Audio fingerprinting, also named as audio hashing, has been well-known as a powerful technique to perform audio identification and synchronization. It basically involves two major steps: fingerprint (voice pattern) design and matching search. While the first step concerns the derivation of a robust and compact audio signature, the second step usually requires knowledge about database and quick-search algorithms. Though this technique offers a wide range of real-world applications, to the best of the authors' knowledge, a comprehensive survey of existing algorithms appeared more than eight years ago. Thus, in this paper, we present a more up-to-date

review and, for emphasizing on the audio signal processing aspect, we focus our stateof-the-art survey on the fingerprint design step for which various audio features and their tractable statistical models are discussed.

An audio fingerprint is a kind of compact representation of an audio signal. It allows indentifying an unknown audio signal by trying to match its signature against the signature of the entire audio signal stored in a database. This system can be useful in detecting pirated audio signals (watermarking) or recognizing or identifying unknown music broadcast by a radio station, for example. Another purpose of the audio fingerprint is integrity verification.

4.3 LIST OF MODULES

- ACR Library Implementation
- ACR Bucket
- Dependencies
- ACR API
- Flash Screen
- UI design

4.3.1 ACR Library Implementation

An audio fingerprint is a condensed digital summary, deterministically generated from an audio signal, that can be used to identify an audio sample or quickly locate similar items in an audio database. The follow picture gives us an intuitive understanding of audio fingerprint, we can take the black lines and points as fingerprints.

ACR technology helps audiences easily retrieve information about the content they watched. For smart TVs and applications with ACR technology embedded the audience can check the name of the song which is played or descriptions of the movie they watched. In addition to that, the identified video and music content can be linked to internet content providers for on-demand viewing, third parties for additional background information, or complementary media.

The fingerprinting algorithm processes the signal of audios and extracts digital features called fingerprints for each audio. Fingerprints are very discriminative so the system can use them to identify the audio they belongs to. Fingerprints are also robust which means they can resist the environment noise and this make it possible to identify recorded audios in rather noisy environments. If the system finds matched fingerprints of the query snippet, it can determine the most like audio in database and gives the position of the snippet in the source audio.

4.3.2 ACR Bucket

A lot of people want to let their apps recognise their own audio & video contents with Audio fingerprinting technologies to improve user experience and marketing effectiveness. Normally these contents include music, Advertisement, television shows, movies and etc

People can add contents they want to recognise into their own buckets by uploading files directly or sending only fingerprints through console APIs.

Figure 4.3.2 Overview of ACR cloud

The following are the steps to create a ACR Cloud Database

Step 1

Go to dashboard, create a bucket as follow.

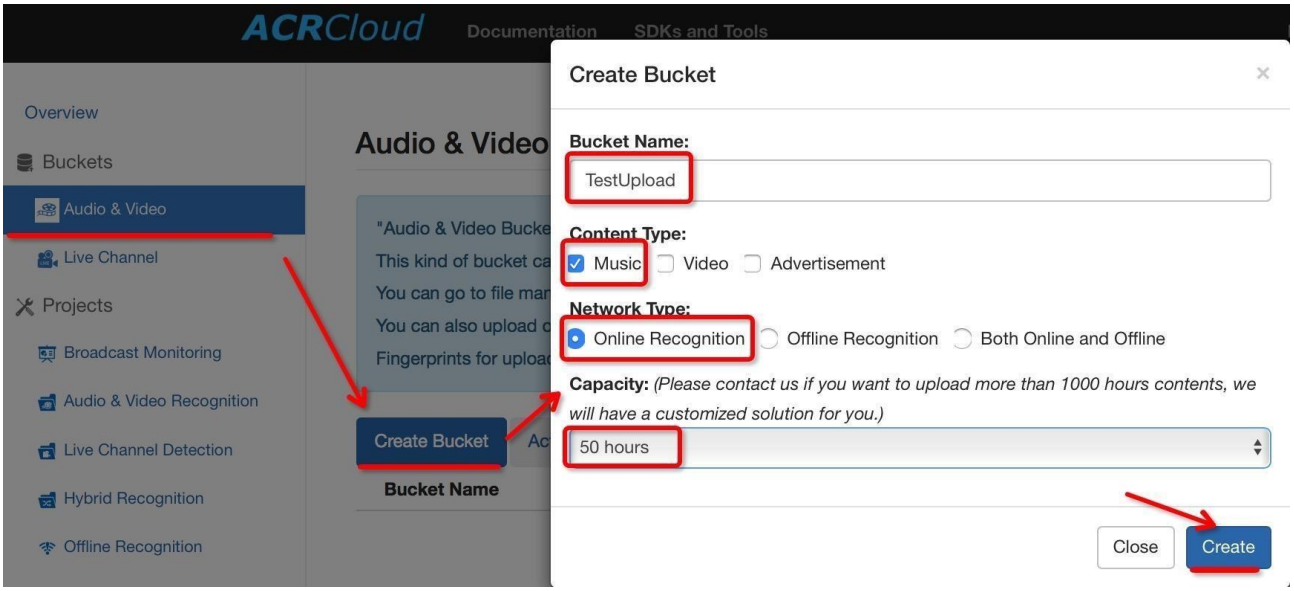

Figure 4.3 Creating Bucket

Step 2

Upload any audio.mp3 into the bucket you just created.

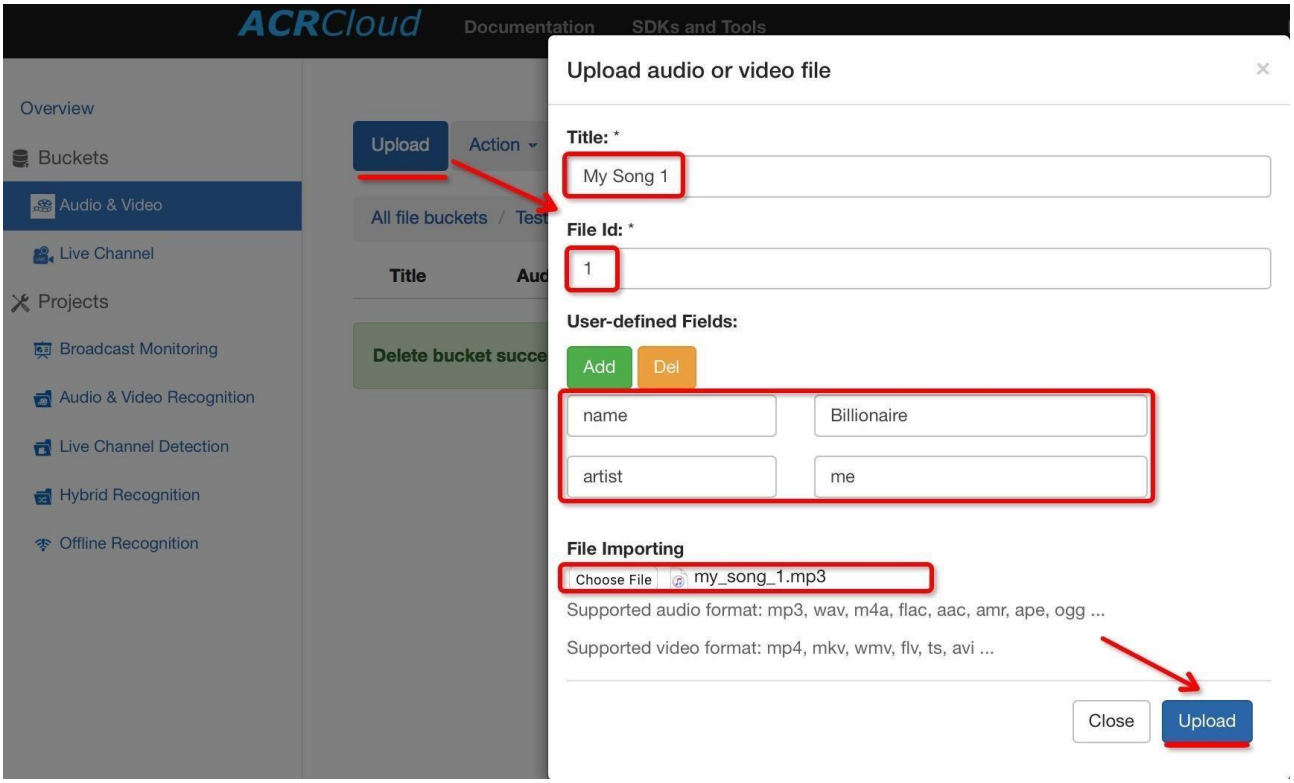

Figure 4.4 Uploading audio to the Bucket

Step 3

Create a project and bind the bucket you just created to it.

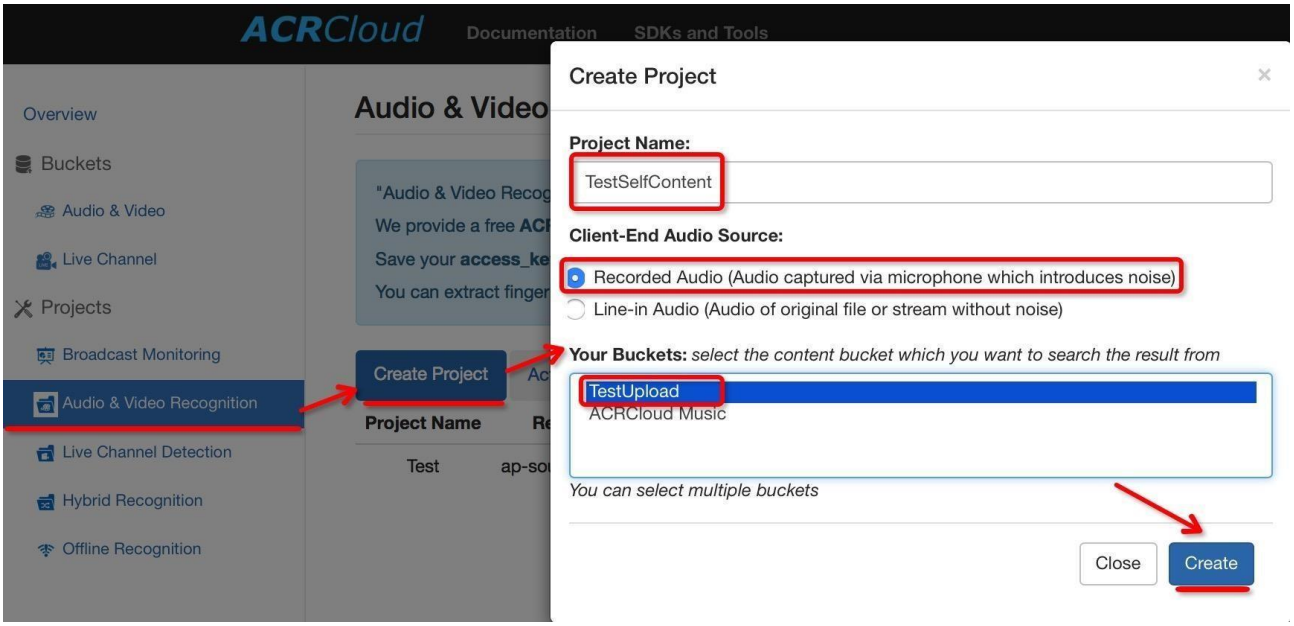

Figure 4.5 Creating a project

Step 4

Save the “host”, “access_key”, “access_secret” . And it can be find in places shown below.

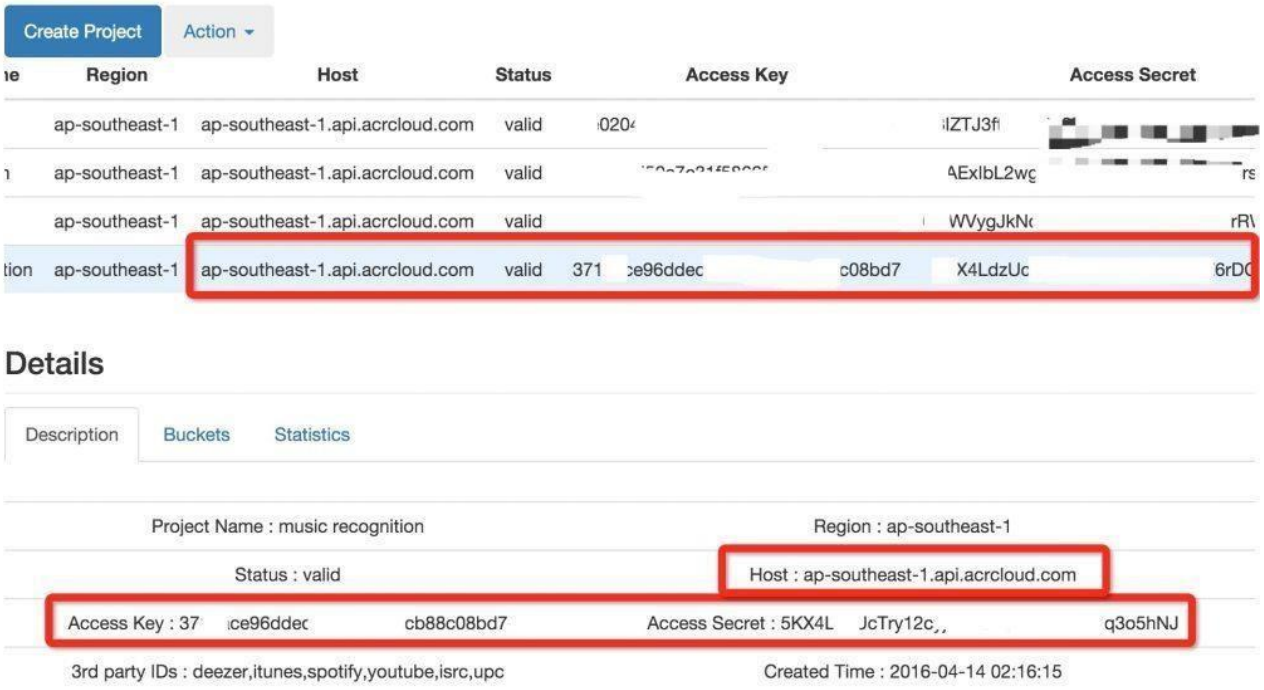

Figure 4.6 ACR Bucket view

4.3.3 Dependencies

The dependencies that are used in the implementation of this application is listed in this section.

```
dependencies {     implementation 'com.wang.avi:library:2.1.3'     implementation
'com.cleveroad:audiovisualization:0.9.1'                          implementation
files('libs/acrcloudandroid-sdk-1.5.7.6.jar')     implementation
'com.github.Mursaat:AnimatedGradientTextView:v0.0.6'              implementation
fileTree(dir: 'libs', include: ['*.jar'])     implementation
'com.android.support:appcompat-v7:28.0.0'                         implementation
'com.android.support.constraint:constraint-layout:1.1.3'     implementation
'com.android.support:design:28.0.0'                               implementation
'com.android.support:support-vector-drawable:28.0.0'     testImplementation
'junit:junit:4.12'                                           androidTestImplementation
'com.android.support.test:runner:1.0.2'                      androidTestImplementation
'com.android.support.test.espresso:espressocore:3.0.2' }
```

4.3.4 ACR Application Program Interface

Audio Fingerprinting (also called Acoustic Fingerprinting) is the kind of most stabled, effective algorithm of ACR and has been widely used in many applications. An audio fingerprint is a condensed digital summary, deterministically generated from an audio signal, that can be used to identify an audio sample or quickly locate similar items in an audio database. The follow picture gives us an intuitive understanding of audio fingerprint, we can take the black lines and points as fingerprints.

Practical uses of audio fingerprinting include identifying songs, melodies, tunes, or advertisements; sound effect library management; and video file identification. Media identification using acoustic fingerprints can be used to monitor the use of specific musical works and performances on radio broadcast, records, CDs and peerto-peer networks. This identification has been used in copyright compliance, licensing, and other monetization schemes.

A robust acoustic fingerprint algorithm must take into account the perceptual characteristics of the audio. If two files sound alike to the human ear, their acoustic

fingerprints should match, even if their binary representations are quite different. Acoustic fingerprints are not bitwise fingerprints, which must be sensitive to any small changes in the data. Acoustic fingerprints are more analogous to human fingerprints where small variations that are insignificant to the features the fingerprint uses are tolerated. One can imagine the case of a smeared human fingerprint impression which can accurately be matched to another fingerprint sample in a reference database; acoustic fingerprints work in a similar way.

Perceptual characteristics often exploited by audio fingerprints include average zero crossing rate, estimated tempo, average spectrum, spectral flatness, prominent tones across a set of frequency bands, and bandwidth.

Most audio compression techniques (AAC, MP3, WMA, Vorbis) will make radical changes to the binary encoding of an audio file, without radically affecting the way it is perceived by the human ear. A robust acoustic fingerprint will allow a recording to be identified after it has gone through such compression, even if the audio quality has been reduced significantly. For use in radio broadcast monitoring, acoustic fingerprints should also be insensitive to analog transmission artifacts.

On the other hand, a good acoustic fingerprint algorithm must be able to identify a particular master recording among all the productions of an artist or group. For use as evidence in a court of law, an acoustic fingerprint method must be forensic in its accuracy.

4.3.5 Flash Screen

The codes used for the implementation of the flash screen is described in the Apendix section. The flash Screen image is designed using adobe photoshop and which is integrated with our application using Android Studio.

Figure 4.7 flash screen

Figure 4.7 illustrates the flash screen which is designed and used in our application.

4.3.6 UI Design

User interface (UI) design is the process of making interfaces in software or computerized devices with a focus on looks or style. Designers aim to create designs users will find easy to use and pleasurable. UI design typically refers to graphical user interfaces but also includes others, such as voice-controlled ones.

Photoshop is used for the UI Design and is implemented using Android Studio. The xml code which is used for the complete User Interface Design is described in this section.

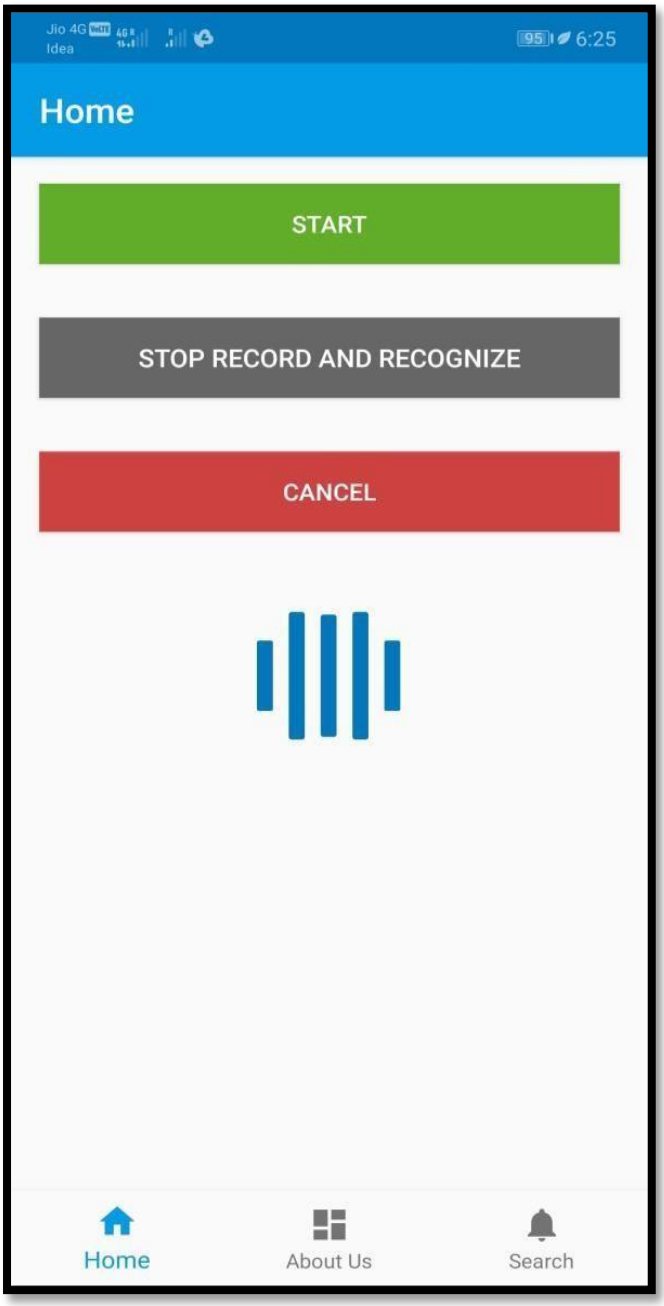

Figure 4.8 UI Design

Xml base code is used for the UI design and which is described in the appendix section.

A Loading animation is added and it will start working when the start button is pressed and will disappear when the audio is identified or the stop button is pressed.

And there are two Navigation activities ‘about us’ and ‘search’. The about us navigation describes the background information including privacy. Using the search navigation it is possible to search the previous audios which recognized with this interface

CHAPTER 5

RESULT AND DISCUSSION

5.1 INTRODUCTION

Audio fingerprinting is best known for its ability to link unlabelled audio to corresponding metadata (e.g. artist and song name), regardless of the audio format. Although there are more applications to audio fingerprinting, such us: Content-based integrity verification or watermarking support, this review focuses primarily on identification. Audio fingerprinting or Content-based audio identification (CBID) systems extract a perceptual digest of a piece of audio content, i.e. the fingerprint and store it in a database. When presented with unlabelled audio, its fingerprint is calculated and matched against those stored in the database. Using fingerprints and matching algorithms, distorted versions of a recording can be identified as the same audio content.

Personal music players can show the name of the recording currently playing. But when listening to music in public environments, it can be a challenge to recognize every song that you listen to in a day.

Imagine the following situation. You're in your car, listening to the radio tape and suddenly you hear a song that catches your attention. It's the best new song you have heard for a long time, but you missed the announcement and don't recognize the artist. Still, you would like to know more about this music. What should you do? You could call the radio station or either your friends, but that's too cumbersome. Would not it be nice if you could push a few buttons on your mobile phone and a few seconds later the phone would respond with the name of the artist, album and the title of the music you're listening to.

One of the first audio fingerprinting systems was developed in the 1990s by E. Wold et al. named Muscle Fish. This audio fingerprinting system was used for the identification of a short sound like a doorbell or applause. The algorithm allows for

the identification of audio, but cannot be used to recognize a musical track based on an excerpt of the track. From the year 2000, several audio fingerprinting systems were released which enabled the recognition of musical tracks based on their content. Examples are Relatable TRM, which was licensed by Napster and an audio fingerprinting system by Philips Research, which description initially was only released in scientific literature. In 2005, this technology of Philips Research was licensed by Grace note which implemented it in their audio recognition products. The original Muscle Fish algorithm was in 2000 acquired by Audible Magic, which further developed the algorithm. They use it for their work in the media recognition and copyright management. The website, MusicBrainz.org, which enables users to manage their music libraries automatically, first used Relatable TRM for the identification of audio, but when they reached the boundaries of scalability of this algorithm, they switched to an audio fingerprinting system by Music-IP.

The working principles is generally a two-step process i.e. submission and lookup.

1. Submission

First, the fingerprint of audio file is made and then the fingerprint is submitted to server.

2. Lookup

The analysis of audio fingerprint is done by server and compares it to other fingerprints.

Fingerprints listed in the database and decides whether it is sufficiently different from known fingerprints as to issue a new ID. Once this step is done, a fingerprint can be calculated for any file and this can be used to look up the corresponding ID. This ID is associated with a given track or recording and metadata can be gathered from there.

5.2 ALGORITHM USED

'Panako' is an acoustic fingerprinting system. Some parts of Panako were inspired by the Robust Landmark- Based Audio Fingerprinting Matlab implementation by Dan

Ellis. The Landmark algorithm uses peaks in the amplitude of the spectrum in each frame to find features to encode as the fingerprint. The system is able to extract fingerprints from an audio stream, and either store those fingerprints in a database, or find a match between the extracted fingerprints and stored fingerprints. Several acoustic fingerprinting algorithms are implemented within Panako. The main algorithm, the Panako algorithm, has the feature that audio queries can be identified reliably and quickly even if they have been speed up, time stretched or pitch shifted with respect to the reference audio.

Analogue physical media such as wax cylinders, wire recordings, magnetic tapes and gramophone records can be digitized at an incorrect or varying playback speed. Even when calibrated mechanical devices are used in a digitization process, the media could already have been recorded at an undesirable speed. To identify duplicates in a digitized archive, a music search algorithm should compensate for changes in replay speed. Next to accidental speed changes, deliberate speed manipulations are sometimes introduced during radio broadcasts: occasionally songs are played a bit faster to fit into a timeslot. During a DJ-set speed changes are almost always present. To correctly identify audio in these cases as well, a music search algorithm robust against pitch shifting, time stretching and speed changes is desired. The Panako algorithm allows such changes while maintaining other desired features as scalability, robustness and reliability.

1. Local maxima are extracted from a constant-Q spectrogram from the query. The local maxima are combined by three to form fingerprints.
2. For each fingerprint a corresponding hash value is calculated.
3. The set of hashes is matched with the hashes stored in the reference database, and each exact match is returned.
4. The matches are iterated while counting how many times each individual audio identifier occurs in the result set.

5. Matches with an audio identifier count lower than a certain threshold is removed, effectively dismissing random chance hits. In practice, there is almost always only one item with a lot of matches, the rest being random chance hits.

6. The residual matches are checked for alignment, both in frequency and time, with the reference fingerprints using the information that is stored along with the hash.

7. A list of audios identifiers is returned ordered by the amount of fingerprints that align both in pitch and frequency.

5.1 Common steps used in audio fingerprint algorithm to convert audios to fingerprints

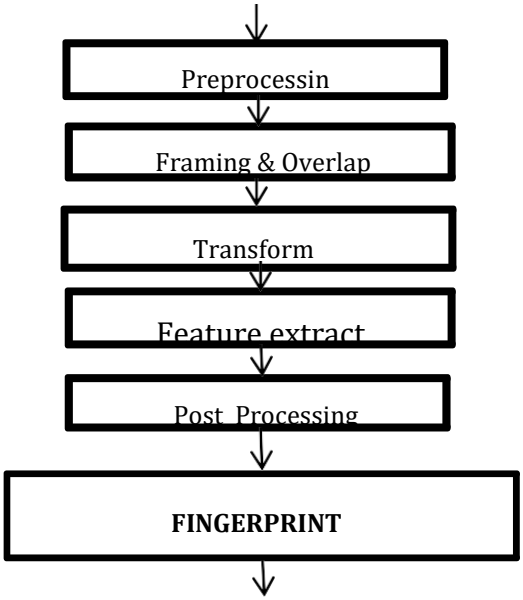

Figure 5.1 Steps used in audio fingerprinting

Given an input signal sample eventually corrupted, its fingerprint allows to quickly retrieve the original file among a database of known audio files when it exists. Since an input audio sample should be identified, audio fingerprints should be composed of elementary keys (called sub-fingerprints) computed continuously along the signal. As stated by Cano et al. most audio fingerprinting systems extract their fingerprint in the same way. The front-end is the part where the feature extraction takes place and forms the basis for the fingerprint extraction. This front-end and fingerprint modeling block which is discussed by Cano et al., is presented in Figure

5.2.1 and described below.

5.2.1 FRONT END

Pre-processing

Pre-processing involves the conversion of the audio data to a standard format . The audio signal is digitized (if necessary) and convert to mono stream. Furthermore, the audio is often resampled to a specific sample rate. This can be the standard CDquality rate of 44.1 kHz, but the audio is often down-sampled to a rate between 4 kHz and 8 kHz. This will lead to the loss of the frequencies above 2 kHz to 4 kHz (Nyquist frequency), but these higher frequencies contain less important information for the recognition of the audio and it drastically improves the efficiency of the transformation algorithm. Finally, normalization can be applied, which will provide the amplitudes of the audio data in a standard range for the framing and overlap stage [5.2.1.1]. Here the down-sampling removes the high frequency information; the high frequency component usually contains less energy and therefore more sensitive to distortion and less stable .

Framing and Overlap

The fingerprint needs to be computed for small excerpts of audio as well as complete songs, the input data is split up into small frames of several milliseconds. Due to this framing, only a short excerpt of the audio will be taken into account. This way the signal appears to remain equal over the length of the frame. Therefore, it is possible to determine describing features of the signal within the frame. To avoid discontinuities at the beginning and ending of the frame, a windowing function is applied. This frame is often extended to the surrounding frames (overlap), because this makes the fingerprint robust against small shifts in alignment of the audio data over the frames. Windowing functions that are used in the selected fingerprinting systems are the Hamming and von Hann windows

Transform

The third stage which is the same for most fingerprinting systems is the transformation stage . The idea behind linear transforms is the transformation of the set of measurements to a new set of features. The redundancy is significantly reduced.

The most common transformation or the most systems apply the Cooley- Tukey implementation of the Fast Fourier Transform (FFT) to facilitate efficient compression, noise removal and subsequent processing. Some other transforms have been proposed - the Discrete Cosine Transform (DCT), the Haar Transform or the Walsh-Hadamard Transform, Richly et al. did a comparison of DFT and the Walsh-Hadamard Transform that revealed that the DFT is generally less sensitive to shifting.

Feature Extract extraction mainly aims at dimensionality reduction in the form of effective descriptions of the underlying signal. Furthermore, by using features that are based on the most robust signal elements it can increase robustness to distortions. Popular features include Mel Frequency Cepstral Coefficients (MFCC), Spectral Flatness Measure (SFM) and Haar features on spectral energies.

Post Processing

This step can be used to normalize the features, to emphasize the temporal evolution of the feature sequence (derivatives) or to represent the data in an efficient form. The order of these steps may be different, repeated, or applied on different time or frequency scales. In conclusion, we can say that each of the before mentioned building blocks aims at one or more of the following goals:

- Dimensionality reduction and compact representation
- Increase robustness to distortion
- Emphasize unique characteristics of the signal
- Match perceptual characteristics

5.2.2 Fingerprint Modelling

The three categories of fingerprint representations are:

Fixed size fingerprint

The size of the fingerprint is independent of the song length. Music is nonstationary; parts with different signal and statistical characteristics are mixed in the final representation. There are three drawbacks of such a system. First, when different parts are mixed together in one model, the discriminating characteristics of such fragments are lost in the modeling procedure; when identifying shorter fragments there is only a partial match with the model derived for the entire system. Second, the timing information and the temporal order of the features is a distinguishing feature of a signal. Third, the fingerprint differences cannot be used to locate the differences between the signals. One of the advantages of losing the temporal information is that the model potentially becomes independent of time scaling distortions.

Constant Rate Fingerprints

Most fingerprinting systems extract features on regular time intervals (frames). Therefore, the fingerprint size is proportional to the song length. The main advantage is that signal characteristics that are changing over time are not mixed in the final fingerprint. Furthermore, the amount of information extracted in a certain time window can be guaranteed. Finally, when comparing the fingerprint of a distorted version to the fingerprint of the original undistorted recording, the fingerprint difference can be used to localize the changes in the distorted version.

Variable Rate Fingerprint

For efficient representation, the rate of fingerprint varies with acoustical events. In this way, the fingerprint only represents those salient characteristics of the underlying acoustic signal. In the Shazam fingerprint, for instance, the spectral peak locations that are most significant in both the frequency and in the temporal dimension represent the fingerprint. This may result in very compact fingerprints. However, one

cannot guarantee the amount of information extracted in a certain time window. The part of the fingerprint that corresponds to a particular time instant is called a subfingerprint. The fingerprint of a song is thus given by a time-series of sub-fingerprints. A number of sub-fingerprints used for identification are called a fingerprint block.

5.2.3 Factors For Good Audio Fingerprinting System

Robestness

This is generally expressed by observing the false negative rate (when the algorithm returns no result, given an available result). In order to achieve high robustness, the fingerprint should be extracted based on perceptual features that do not vary with signal degradation.

Reliability

This is expressed through the false positive rate (when the algorithm returns a result but it is incorrect). This is an important consideration as false positives can cause problems with copyright, royalty distribution, users downloading incorrect songs and so on. Echoprint, Chromaprint and Panako are reliable as all the algorithms' incorrect result display is negligible.

Fingerprint size

This property represents how much storage space is needed for each fingerprint. This can be an important consideration for consumer tagging systems where fingerprints are sent for identification using cellular data networks. Fingerprint length is calculated so as to find the size. Panako stores the hash values but also displays the fingerprint in graphical method so it requires higher space to store fingerprint.

Granularity

This property indicates how many seconds of audio is needed for positive match. This will vary depending on the application, with most applications requiring a relatively small granularity such as a consumer music identification service where

users will normally capture only a small section from the middle of the audio they wish to identify. Echoprint and Panako could display the correct result even in 5 seconds of audio from middle of audio while Chromaprint could not. It could display the result only with longer audio sections from beginning of audio.

Search Speed And Scalability

These are related more to the fingerprint database than to the feature extraction algorithm. They represent how fast fingerprint matching is and its computational cost. The query modifications for the comparative study of algorithms are as follows. Each of the modifications is tested to the three algorithms Echoprint and Panako.

As from the literature review, it is known that the Panako algorithm has the feature that audio queries can be identified reliably and quickly even if they have been speed up, time stretched or pitch shifted with respect to the reference audio.

5.3 HOW THE APPLICATION WORKS

Here audio file from any source is taken as the input of this application. The app tries to record the audio file for minimum one second to maximum of 12 seconds. Then it converts the audio files as fingerprints and this fingerprints are send to the cloud libraries. Indexing occurs and result get displayed on the screen. In case if there Is no matches found it shows an error message.

The main advantage of this application is that this records the audio even in the noisy atmosphere and displays the successful result. The matching of the local audios are done using API with the help of ACR Cloud Libraries. And the custom videos are recognized by the ACR Cloud Bucket which is used as a database to upload the custom audio files. The finger prints of these custom videos are also recognized by using the API

The figure shows the successfully occurred matching and the output

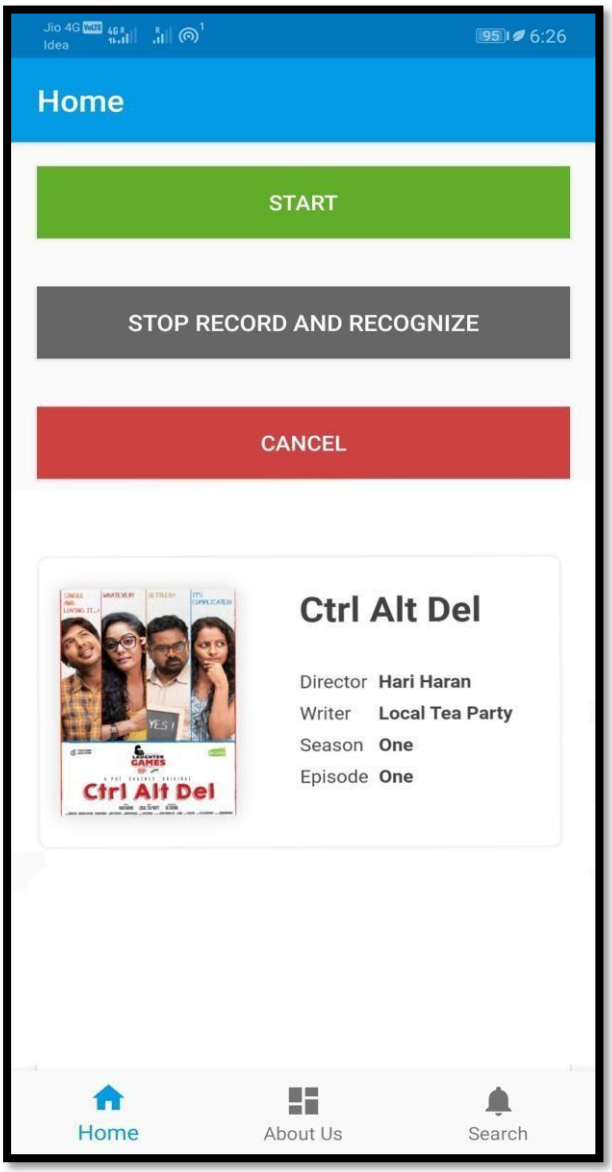

Figure 5.2 Result page

Here the application recognized a custom audio which is uploaded in the ACR Bucket cloud.

CHAPTER 6
CONCLUSION AND FUTURE WORKS

6.1 CONCLUSION

A new mobile interface is created with local and custom audio recognition. API for the content recognition is available in the ACR Cloud. With the help of this API we created a mobile interface that user can recognize the audio files including music, Tv shows and any other custom videos that are uploaded in the ACR Bucket. This interface helps the user to find the meta data about an audio file from any of the sources. One of the main advantages that this Interface provides is, The audio fingerprints of the segments do not necessarily have to be of high quality to be a match. Distortions and interference of the original signal makes matching of the fingerprints less reliable, but (to a certain extent) which takes only less than 10 seconds to recognize it. This provides the name of the audio which described in the Bucket along with its other description including image information.

Within five to ten seconds this application converts the audio file in to fingerprints and match it with both the library collection and the bucket database which is used for the storage of custom audio fingerprints. The custom content as well as its fingerprint need to be uploaded in the bucket database for the matching purpose. After the audio gets converted to fingerprints the API matches it with the millions of fingerprints in the cloud. Which results to the successful recognition.

6.2 FUTURE WORKS

This application is developed for custom audio recognition using audio fingerprint algorithm. We have number of hearing-impaired people living around the world, which is estimated to be over 3 million. This project expansion aims to help them in way that they can notice what is happening around them even in their sleep with the help of IoT environment. As this application provides an Interface for custom video recognition,

both the local and uploaded custom videos can be recognized. The usual sounds that happens every day can be recorded and uploaded to the database. These sounds can be easily identified with this application and make them understand what is happening around with the help of an IOT devise. This feature will be very helpful for hearing impaired people who are living alone. Our project aims to help them to ease their life.

APPENDIX

A.1 SCREENSHOT

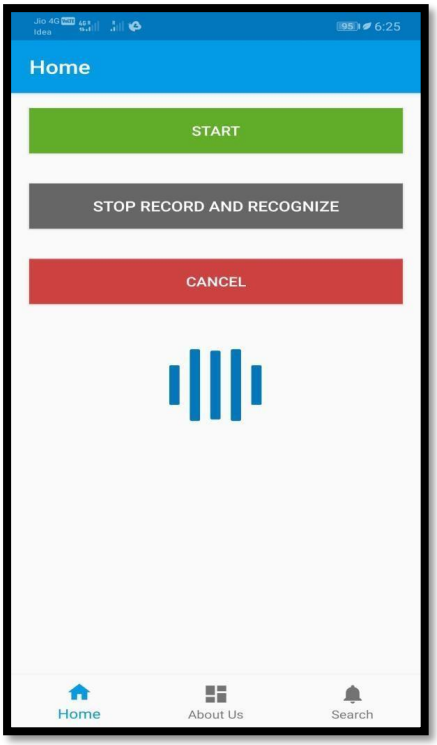

Figure A.1 home screen of the application

This screen shows the main activity content where user can start recognize content.

Figure A.2 Splash screen of the application

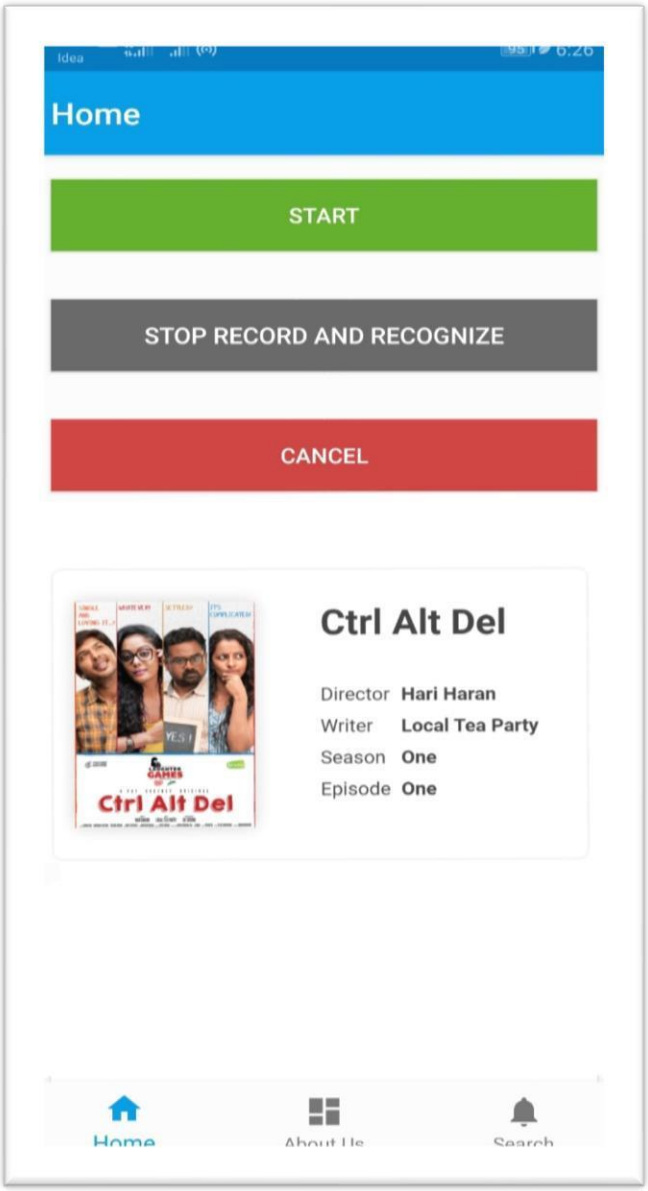

Figure A.3 Final Output after recognizing the series

The output shows the recognised content as a metadata information.

A.2 CODING

JAVA CODE FOR COMPLETE IMPLEMENTATION

```
package com.stardust.mixmatch; import
android.os.Bundle; import android.os.Environment;
import android.support.annotation.NonNull; import
```

```
android.support.design.widget.Bottom
NavigationView; import
android.support.v7.app.ActionBar; import
android.support.v7.app.AppCompatAct
ivity; import
android.support.v7.widget.CardView;
import android.util.Log; import
android.view.Menu; import
android.view.MenuItem; import
android.view.View; import
android.widget.Button; import
android.widget.TextView; import
android.widget.Toast;

import com.acrcloud.rec.sdk.ACRCloudClient; import
com.acrcloud.rec.sdk.ACRCloudConfig; import
com.acrcloud.rec.sdk.IACRCloudListener; import
com.wang.avi.AVLoadingIndicatorView;

import org.json.JSONArray; import
org.json.JSONException;     import
org.json.JSONObject;

import java.io.File;

public class MainActivity extends AppCompatActivity implements
IACRCloudListener {

private BottomNavigationView.OnNavigationItemSelectedListener mOnNavigati =
new BottomNavigationView.OnNavigationItemSelectedListener() {
```

```
@Override public boolean onNavigationItemSelected(@NonNull
MenuItem item) { switch (item.getItemId()) { case R.id.navigation_dashboard:

return true; case
R.id.navigation_home: {

return true; } case
R.id.navigation_notifications:

return true;
} return
false;
} };
private ACRCloudClient mClient; private
ACRCloudConfig mConfig; private
AVLoadingIndicatorView avi;

private TextView  mResult, tv_time;

private boolean mProcessing = false; private boolean
initState = false;

private String path = "";

private long startTime = 0; private long stopTime = 0;
private CardView card; @Override protected void
onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main); ActionBar actionBar
```

```
= getSupportActionBar();
actionBar.setDisplayHomeAsUpEnabled(false);

path = Environment.getExternalStorageDirectory().toString()  +
"/acrcloud/model";

File file = new File(path); if(!file.exists()){
file.mkdirs();
}


mResult = (TextView) findViewById(R.id.result); tv_time
= (TextView) findViewById(R.id.time);
card=(CardView)findViewById(R.id.card);
avi=(AVLoadingIndicatorView)findViewById(R.id.avi); Button
startBtn = (Button) findViewById(R.id.start);
startBtn.setText(getResources().getString(R.string.start));

Button stopBtn = (Button) findViewById(R.id.stop);
stopBtn.setText(getResources().getString(R.string.stop));
findViewById(R.id.stop).setOnClickListener( new View.OnClickListener() {

@Override public void onClick(View
v) { stop(); }  });

Button cancelBtn = (Button) findViewById(R.id.cancel);
cancelBtn.setText(getResources().getString(R.string.cancel));

findViewById(R.id.start).setOnClickListener(new View.OnClickListener() {
```

```
@Override public void
onClick(View arg0) {  start();
avi.smoothToShow();
}
});

findViewById(R.id.cancel).setOnClickListener( new
View.OnClickListener() {

@Override public void onClick(View
v) { cancel(); avi.smoothToHide();
}
});


this.mConfig = new ACRCloudConfig(); this.mConfig.acrcloudListener
= this;

// If you implement IACRCloudResultWithAudioListener and override
"onResult(ACRCloudResult result)", you can get the Audio data.
//this.mConfig.acrcloudResultWithAudioListener = this;

this.mConfig.context = this; this.mConfig.host = "identify-eu-west-
1.acrcloud.com"; this.mConfig.dbPath = path; // offline db path, you can change it
with other path which this app can access.
this.mConfig.accessKey = "3d793ad4ce7a8a6836aa3ab9bc24ec11";
this.mConfig.accessSecret  =  "83Ld6hApwK2rPgrndICFJgBrSLiUfxiqW5cltwuM";
this.mConfig.protocol =
```

```
ACRCloudConfig.ACRCloudNetworkProtocol.PROTOCOL_HTTP;     //
PROTOCOL_HTTPS
this.mConfig.reqMode =
ACRCloudConfig.ACRCloudRecMode.REC_MODE_REMOTE;
//this.mConfig.reqMode =
ACRCloudConfig.ACRCloudRecMode.REC_MODE_LOCAL;
//this.mConfig.reqMode =
ACRCloudConfig.ACRCloudRecMode.REC_MODE_BOTH;

this.mClient = new ACRCloudClient();
// If reqMode is REC_MODE_LOCAL or REC_MODE_BOTH,
// the function initWithConfig is used to load offline db, and it may cost long time.
this.initState = this.mClient.initWithConfig(this.mConfig); if (this.initState) {
this.mClient.startPreRecord(3000); //start prerecord, you can call
"this.mClient.stopPreRecord()" to stop prerecord.
}
}

public void start() { if
(!this.initState) {
Toast.makeText(this, "init error", Toast.LENGTH_SHORT).show(); return;
}

if (!mProcessing) { mProcessing = true; mResult.setText("");
if (this.mClient == null || !this.mClient.startRecognize()) {
mProcessing = false; mResult.setText("start error!");
}                    startTime                    =
System.currentTimeMillis();
}
}
```

```
protected void stop() { if (mProcessing
&& this.mClient != null) {
this.mClient.stopRecordToRecognize();
avi.smoothToHide();
}    mProcessing    =
false;

stopTime = System.currentTimeMillis();
}

protected void cancel() {          mResult.setText("");
if (mProcessing && this.mClient != null) {
mResult.setText(""); mProcessing = false;
this.mClient.cancel(); tv_time.setText(""); avi.hide();
card.setVisibility(View.GONE);
}
}

@Override public boolean
onCreateOptionsMenu(Menu menu) {
getMenuInflater().inflate(R.menu.main,
menu); return true;
}

public void onResult(String result) { if
(this.mClient != null) {
this.mClient.cancel(); mProcessing
= false;
}
```

```
String tres = "\n";

try {
JSONObject j = new JSONObject(result);
JSONObject j1 = j.getJSONObject("status");
int j2 = j1.getInt("code"); if(j2 == 0){
JSONObject metadata = j.getJSONObject("metadata");
//
avi.smoothToHide(); card.setVisibility(View.VISIBLE);
if
(metadata.has("humming")) {
JSONArray hummings = metadata.getJSONArray("humming"); for(int
i=0; i<hummings.length(); i++) { JSONObject tt =
(JSONObject) hummings.get(i);
String title = tt.getString("title");
JSONArray artistt = tt.getJSONArray("artists");
JSONObject art = (JSONObject) artistt.get(0);
String artist = art.getString("name"); tres = tres
+ (i+1) + ".  " + title + "\n";
} }
if (metadata.has("music")) {
JSONArray musics = metadata.getJSONArray("music"); for(int
i=0; i<musics.length(); i++) { JSONObject tt =
(JSONObject) musics.get(i);
String title = tt.getString("title");
JSONArray artistt = tt.getJSONArray("artists");
JSONObject art = (JSONObject) artistt.get(0);
String artist = art.getString("name"); tres = tres + (i+1) + ".  Title:
" + title + "    Artist: " + artist + "\n";
```

```
} }
if (metadata.has("streams")) {
JSON Array musics = metadata.getJSONArray("streams"); for(int
i=0; i<musics.length(); i++) { JSONObject tt =
(JSONObject) musics.get(i);
String title = tt.getString("title");
String channelId = tt.getString("channel_id"); tres = tres + (i+1) + ".  Title: "
+ title + "    Channel Id: " + channelId + "\n";
} }
if (metadata.has("custom_files")) {
JSONArray musics = metadata.getJSONArray("custom_files");
for(int i=0; i<musics.length(); i++) { JSONObject tt =
(JSONObject) musics.get(i); String title = tt.getString("title"); tres
= tres + (i+1) + ".  Title: " + title + "\n";
} } tres = tres + "\n\n" + result;
}else{ tres = result;
}
} catch (JSONException e) { tres
= result;
 e.printStackTrace();
}

mResult.setText(tres);
}

@Override public void
onVolumeChanged(double v) {

}
```

```
@Override protected void
onDestroy() { super.onDestroy();
Log.e("MainActivity", "release");
if (this.mClient != null) {
this.mClient.release(); this.initState
= false; this.mClient
= null;
}
}

@Override public void onPointerCaptureChanged(boolean hasCapture) {

}
}
```

XML CODE FOR UI DESIGN

```
<?xml version="1.0" encoding="utf-8"?>
   <android.support.constraint.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/container" android:layout_width="match_parent"
android:layout_height="match_parent" tools:context=".MainActivity">
<ScrollView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
tools:ignore="MissingConstraints">

<LinearLayout
android:layout_width="fill_parent"
```

```
android:layout_height="wrap_content"
android:orientation="vertical"
android:gravity="center">

<Button android:id="@+id/start"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_margin="16dp"
android:background="#61AD2A"
android:shape="rectangle"
android:text="@string/start"
android:textColor="#ffffffff"
android:textColorHighlight="#2E7D32" />
<Button android:id="@+id/stop"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColorHighlight="@color/color
PrimaryDark"
android:textColor="@color/colorAccent"
android:background="#666666"
android:layout_margin="16dp"
android:textSize="15dp"
android:text="@string/stop" />

<Button android:id="@+id/cancel"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColorHighlight="#DB8484"
android:textColor="@color/colorAccent"
android:background="#E9CE3535"
```

```
android:layout_margin="16dp"
android:text="@string/cancel" />

<com.wang.avi.AVLoadingIndicatorView
android:visibility="gone" android:id="@+id/avi"
android:layout_width="100dp" android:layout_height="100dp"
android:layout_marginTop="20dp"
app:indicatorColor="@color/colorPrimaryDark"
app:indicatorName="LineScalePulseOutIndicator" />
<TextView android:id="@+id/result"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="#000000" />

<TextView android:id="@+id/time"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="#000000" />
<android.support.v7.widget.CardView
android:visibility="gone"
android:id="@+id/card"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_margin="16dp">

<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="vertical">
```

```
<ImageView
android:layout_width="match_parent"
android:layout_height="160dp"
android:scaleType="centerCrop"
/>
<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="vertical"
android:padding="16dp">

<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginBottom="8dp"
android:text="@string/card_title"
android:textColor="#000"
android:textSize="18sp" />

<TextView
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="@string/card_content"
android:textColor="#555" />
</LinearLayout>

<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content">
```

```
<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Save"
android:theme="@style/PrimaryFlatButton"
android:background="@color/colorAccent"
android:textColor="@color/colorPrimary"/>
<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Done"
android:theme="@style/PrimaryFlatButton"
android:background="@color/colorAccent"
android:textColor="@color/colorPrimary"/>
</LinearLayout>
</LinearLayout>

</android.support.v7.widget.CardView>

</LinearLayout>
</ScrollView>

</android.support.constraint.ConstraintLayout>
```

REFERENCES

1. S. Brin and L. Page “The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, vol. 30, pp. 107– 117, 1998.

2. G.-H. Cha, X. Zhu, D. Petkovic, and C.-W. Chung, “An efficient indexing method for nearest neighbor searches in high-dirnensional image databases,” IEEE Tr. Multimedia, vol. 4, no. 1, pp. 76–87, 2002.

3. A. Gionis, P. Indyk, and R. Motwani, “Similarity Search in High Dimensions Via Hashing,” Proc. VLDB Conf., pp. 518–529, 1999.

4. J. Haitsma and T. Kalker, “A Highly Robust Audio Fingerprinting System With an Efficient Search Strategy,” J. New Music Research, vol. 32, no. 2, pp. 211–221, 2003.

5. J. Haitsma and T. Kalker, “A Highly Robust Audio Fingerprinting System,” Proc. Int. Symp. on Music Information Retrieval, pp. 107–115, 2002.

6. M.L. Miller, M.C. Rodriguez, and I.J. Cox, “Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces,” J. VLSI Signal Processing, vol. 41, pp. 285–291, 2005.

7. M.L. Miller, M.C. Rodriguez, and I.J. Cox, “Audio Fingerprinting: Nearest Neighbor Search in High Dimensional Binary Spaces,” Proc. IEEE Multimedia Signal Processing Workshop, pp. 182–185, 2002.

8. J. Oostveen, T. Kalker, and J. Haitsma “Feature Extraction and a Database Strategy for Video Fingerprinting, ” Proc. Int. Conf. on Visual Information Systems, vol. LNCS 2314, pp. 117–128, 2002.

9. P. Zezula, G. Amato, V. Dohnal, and M. Batko, Similarity Search: The Metric Space Approach, Springer, 2006

10. P. Cano, E. Batlle, T. Kalker, and J. Haitsma, “A review of audio fingerprinting,” Journal of VLSI Signal Processing Systems, vo1.41, no.3, pp. 271-284, Nov. 2005.

11. M. Riley, E. Heinen, and J. Ghosh, “A Text-Retrieval Approach to Content-Based Audio Retrieval,” in Proc. The 1st ACM International Conference on Multimedia Information Retrieval, Vancouver, Canada, pp. 105-112, Oct. 2008.

12. M. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney “Contentbased music information retrieval: current directions and future challenges,” Proceedings of the IEEE, vol. 96, no. 4, pp. 668-696, Apr. 2008.

13. R. van Zwol, S. Rüger, M. Sanderson and Y. Mass: “Multimedia information retrieval: new challenges in audio visual search,” Workshop Report of ACM SIGIR Forum, 41(2), pp 77–82, Dec. 2007.

14. W. Son, H. Cho, K. Yoon, and S. Lee, “Sub-fingerprint masking for a robust audio fingerprinting system in a real-noise environment for portable consumer devices,” IEEE Transactions on Consumer Electronics, vol. 56, no. 1, pp. 156-160, Feb. 2010.

15. S. Shirali-Shahreza, H. Abolhassani, and M. Shirali-Shahreza, “Fast and scalable system for automatic artist identification,” IEEE Transactions on Consumer Electronics, vol. 55, no. 3, pp. 1731-1737, Aug. 2009.

16. S. Lim, J. Lee, S. Jang, S. Lee, and M. Kim, “Music-genre classification system based on spectro-temporal features and feature selection,” IEEE Transactions on Consumer Electronics, vol. 58, no. 4, pp. 1262-1268, Nov. 2012.

17. J. Haitsma and T. Kalker, “A Highly Robust Audio Fingerprinting System,” in Proc. International Symposium on Music Information Retrieval, Paris, France, pp. 107-115, Oct. 2002.

18. S. Lee, D. Yook, and S. Chang, “An efficient audio fingerprint search algorithm for music retrieval,” IEEE Transactions on Consumer Electronics, vol. 59, no. 3, pp. 652656, Aug. 2013.

19. G. Cha. “An indexing and search strategy for fingerprint databases,” in Proc. 2012 International Conference Multimedia Computing and Systems, Tangiers, Morocco, pp. 7-12, May. 2012.

20. Y. Qian, H. Dou and Y. Feng, “A novel algorithm for audio information retrieval based on audio fingerprint,” in Proc. 2010 International Conference on Information Networking and Automation, Kunming, China, pp. 266 - 270, Oct. 2010.

21. A. Wang. “An industrial strength audio search algorithm,” in Proc. Storage and Retrieval Methods and Applications for Multimedia 2004, San Jose, CA, USA, pp. 582-588, Jan. 2004.

22. W. Dong. “High-Dimensional Similarity Search for Large Datasets,” PhD Dissertation, Princeton University, Nov. 2011.

23. S. Rubin, R. Bodík, and T. Chilimbi. “An Efficient Profile-analysis Framework for Data-layout Optimizations,” in Proc. the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, Portland, USA, pp. 140-153, Jan. 2002.