Simulating Shazam: Acoustic Fingerprinting for Music Identification

A Comprehensive Study on Developing a Song Recognition System

Degree Project Report in Electrical Engineering

Rasmus Gyllenhammar
Christoffer Lennernäs

INSTITUTION FOR ELECTRICAL ENGINEERING

CHALMERS TEKNISKA HÖGSKOLA
Gothenburg 2024
www.chalmers.se

Degree Project 2024

Simulating Shazam: Acoustic Fingerprinting for Music Identification

A Comprehensive Study on Developing a Song Recognition System

Rasmus Gyllenhammar
Christoffer Lennernäs

CHALMERS

Department of Electrical Engineering
Chalmers Tekniska Högskola
Gothenburg 2024

Simulating Shazam: Acoustic Fingerprinting for Music Identification
A Comprehensive Study on Developing a Song Recognition System
Rasmus Gyllenhammar & Christoffer Lennernäs

Supervisor: Javad Aliakbari, Institution for Electrical Engineering
Examinator: Giuseppe Durisi, Institution for Electrical Engineering

Degree project 2024
Department of Electrical Engineering
Chalmers Tekniska Högskola
SE-412 96 Gothenburg
Telefon +46 31 772 1000

Cover: A spectrogram resembling the Chalmers Tekniska Högskola logo, generated in Python.

Written in LaTeX
Gothenburg 2024

Simulating Shazam: Acoustic Fingerprinting for Music Identification
A Comprehensive Study on Developing a Song Recognition System
Rasmus Gyllenhammar & Christoffer Lennernäs
Department of Electrical Engineering
Chalmers Tekniska Högskola

Abstract

This report explains the implementation, design and testing of the core functionalities of Shazam. Shazam identifies songs by capturing short audio segments and matches them against a sizeable database. The program is coded in Python and with MySQL for the database. To perform tests, both audio files and a microphone are used to catch the samples of the songs. The results of the project are deemed successful as it can detect songs from a 10 second sample of a song. In conclusion, the project demonstrates a strong foundation to continue developing the project to simulate Shazam.

Contents

1

Introduction

1.1 Background

In 2002, Shazam was created [1] allowing users to identify music with their mobile phone, even in noisy environments. Back then, before smartphones, Shazam worked by dialing the number 2580 and letting it observe and obtain a sample of the song during approximately 15 seconds. Consequently, it searches for a match in the database and would then send a SMS back with the name of the artist and album.

Nowadays, with the use of smartphones, Shazam has been turned into an app and has over 1 billion users and a database with over 10 million songs [2].

1.2 Purpose

This project delves into the realm of audio processing and pattern recognition used in Shazam. It involves simulating the functionality of Shazam. Shazam employs acoustic fingerprinting techniques to recognize songs based on short audio clips.

The primary objective of this project is to develop a software simulation that emulates the core functionalities of Shazam. This implies implementing algorithms for audio fingerprinting, time-frequency analysis, and matching techniques. The simulation should be capable of identifying songs from partial audio samples (around 5 to 10 seconds).

It would be highly beneficial to implement a database system to store and retrieve audio fingerprints efficiently.

1.3 Objective

The objectives of the project are:

- Read an audio file of a song.
- Process the audio file, retaining only necessary information to minimize data usage.
- Implement fingerprinting to find a unique representation for each song.
  - Develop an efficient method for searching and storing fingerprints.
  - Generate fingerprints for the songs.
- Create a database and store the fingerprints.
- Develop a function to compare and score the similarity of fingerprints.

The following figure shows the system overview of the project.

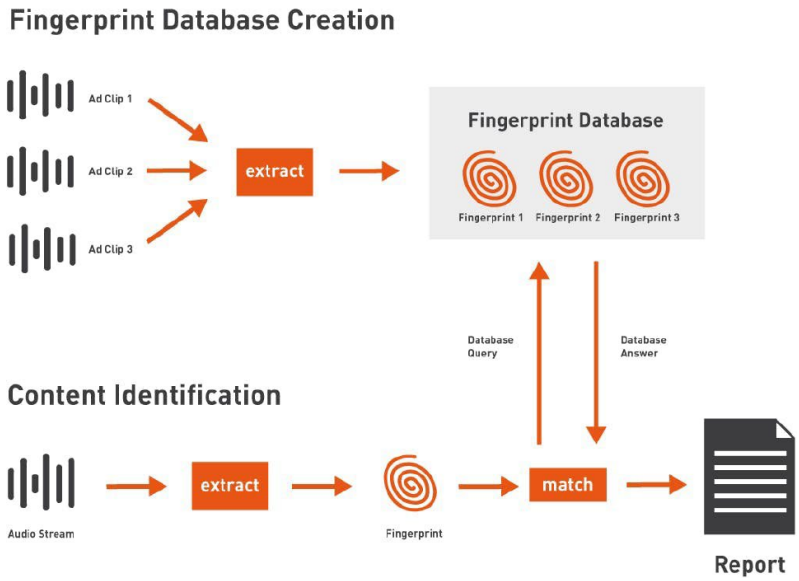

Figure 1.1: System overview.

1.4 Delimitations

This project have some delimitations due to different circumstances. The program that will be created will not have its main focus on to optimise the project's time performance. Due to time constraints, the emphasis will be on creating the core functionalities. Additionally, the project will have a smaller size database compared to the original Shazam since it is a significantly amount data to be stored on a laptop.

2

Theory/Background material

This chapter contains the essential information about audio processing, programming and background material which is vital to both understand and carry out the project.

2.1 Description of Audio processing for this project

Audio processing is a fundamental part in this project and this section will explain the details and important techniques in audio processing.

2.1.1 Fast Fourier Transform

Since a computer cannot store an infinite number of points, the signal needs to be sampled and represented with discrete values [3]. The rate at which this song is sampled is called the sample rate. This process involves converting the continuous signal into discrete bits for storage.

Fast Fourier Transform (FFT) is a mathematical technique used to transform a sequence of discrete samples of a signal from the time domain into the frequency domain [3].

In practice, audio signals are composed of many different sound waves with different frequencies. When applying FFT on the audio signal, it converts it to the frequency spectrum containing all the frequencies of the audio waves. Additionally, it helps distinguish the frequencies of the audio signal apart from noise [3]. Figure 2.1 shows an example of FFT on a sinus signal with some noise.

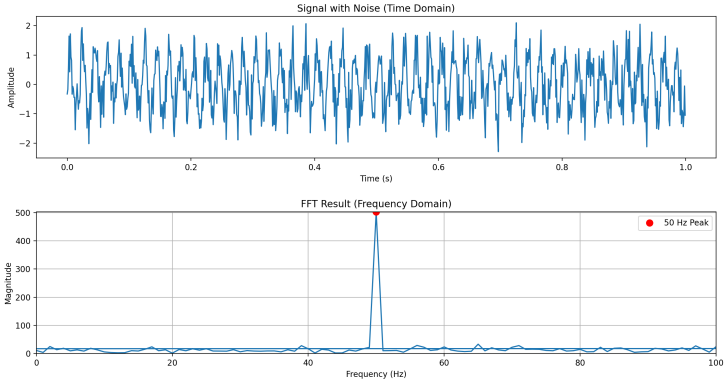

Figure 2.1: Shows a noisy audio signal in the time-domain (top) and frequency-domain (bottom).

The sinus signal has a frequency of 50 Hz and random noise is added in the first graph in Figure 2.1. Consequently, it is unclear what frequency component the sinus

signal has due to the noise in the time domain. By contrast, the second graph is the same signal but in the frequency domain and it displays the sinus signal's frequency component clearly.

2.1.2 Spectrogram

When there is a sound, it generates waves through the air which is perceived as sound [4]. As you listen to a song, these vibrations change in specific patterns over time, which your brain interprets as music. A spectrogram provides a visual representation of these sounds, reflecting the way our brains perceive them.

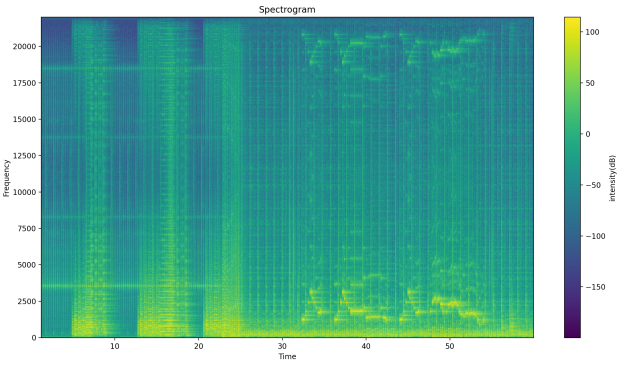

Figure 2.2: Spectrogram of song Baby Elephant

A spectrogram is a three-dimensional graph where the x-axis represents time, the y-axis represents frequency, and the z-axis represents magnitude. The z-axis is depicted using color intensity [5][6]. Figure 2.2 shows a spectrogram of the song "Baby Elephant". Yellow/Light green means it exists high magnitude at that point in the song. Meanwhile, the blue color represents low magnitude [6].

To create a spectrogram [6], the audio signal is divided into segments, and an FFT is applied to each segment, see Figure 2.3. The results of the FFT for all segments are then combined to create the spectrogram. The size of the segments can be adjusted as well as the overlap between the segments [7]. Larger segments simplify computations but sacrifice accuracy. On the other hand, reducing segment size enhances accuracy but increases computation.

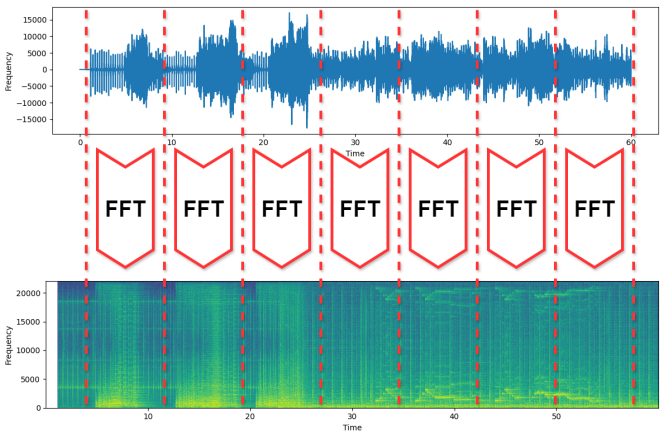

Figure 2.3: Converting audio signal from time domain to a spectrogram.

To divide the audio signal into segments, a windowing function is applied to isolate each segment by filtering out parts of the signal outside the segment. This process is known as windowing [3]. Because the windowing process is not perfect, some data

can be filtered out that are not supposed to. Therefore, the segments can be adjusted to overlap with each other to ensure that no samples are lost when filtered [7].

2.2 Maximum filter / Pooling layers

A maximum filter is frequently used in image processing. A maximum filter highlights the key features in an image. It operates by scanning through the image pixel by pixel and changes the currently scanned pixel's value with the maximum value within a defined neighbourhood boundary [6]. A neighbourhood boundary is the region around the currently scanned pixel. Additionally, the maximum filter have a setting controlling the size of the neighbourhood box. The pixel's value depends on if it is a light or darker color where lighter pixels have higher numbers [6]. Figure 2.4 illustrates an example of how a maximum filter works.

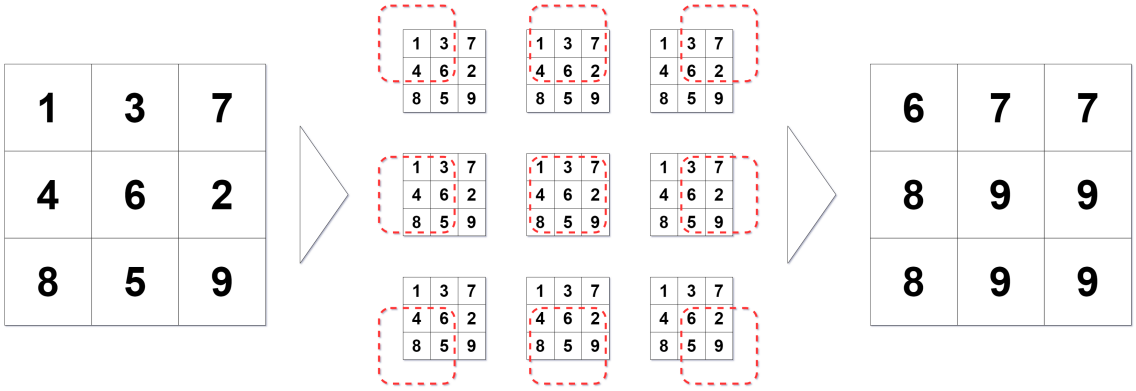

Figure 2.4: Functionality of maximum filter.

In Figure 2.4, each square is a pixel containing its initial value. Figure 2.4 also displays setting a neighbourhood boundary around every pixel. Within each neighbourhood boundary, the maximum filter searches for the maximum values and places those values in a new table.

2.3 Hash, Hashing and Hash Tables

Hash: A hash is a fixed-size string or number generated from input data of variable size, representing the original data uniquely [9].

Hashing: Hashing is the process of converting input data into a fixed-size hash using a hash function [9]. This technique serves multiple purposes. In the context of this report, hashing is used to enable fast data access in data structures like hash tables, where hash values function as indexes.

In python, there is a built-in hash function `hash()` [9][10], Figure 2.5 illustrates an example how it can be used.

```
>>> song_name = "Jag ringer på fredag"
>>> hash(song_name)
-6748037122888429936
```

Figure 2.5: The in-built Python hash function `hash()` is used to generate a hash.

As stated earlier, the hash value produced is unique as it will only be produced for the same input without fail [9][10].

Hash Table: A hash table is a data structure that allows for fast data retrieval by using hash values as indexes [11]. It consists of an array in which data is stored, and a hash function that maps input data to a specific index in this array. The primary advantage of hash tables is their efficiency in search, insertion, and deletion operations, typically achieving these operations in constant time, meaning the performance remains consistent regardless of the hash table size.

2.4 Description of essential libraries in python

The following list contains the essential packages for this project:

- `Numpy`: stands for numerical Python which is an computational library. It contains mathematical functions that help in computing numerical data [12].
- `SciPy`: This package is built on NumPy. SciPy provides signal processing, optimatizion and read functions. It contains functions such as FFT, spectrogram and wave file reading [13].
- `UnitTest`: is used to assess the code's functionality by breaking down the code into smaller segments. These tests are called testcases for each small code segments and every testcase will provide a result showing if the code is working or not [14].
- `TQDM`: displays a progress bar in loops. It also displays the amount of time it takes for the program to run [15].
- `JavaScript Object Notation`: JSON is a syntax for storing and exchanging data. The JSON data is formed with key/value pairs, where each key is a string describing the value [16].
- `Iteration-utilities`: It can iterate through an object and count how many attributes there are and how many of each attributes exists [17].
- `Matplotlib`: A library used to create visualization in Python. With the help of this package, it is possible to create static, animated and interactive plots[18].
- `PyAudio`: Offers convenient functionality for playing and recording audio across different platforms, including GNU/Linux, Microsoft Windows, and Apple macOS [19].
- `Mysql.connector`: Is essential in order to be able to connect to MySQL database. It is also used to interact with the MysQL server to work in the database through Python [20][21].

2.5 MySQL - Database

MySQL is an open source SQL database management system. It is used to add and access data stored in the database. Additionally, MySQL works by processing

requests (queries) from the client's devices or applications to the server, a configuration referred as client/server system. Moreover, MySQL uses Structured Query Language (SQL) to communicate with the database [21].

MySQL databases are relational, meaning the data are stored in tables with rows and columns instead of one big storage space [21]. This approach gives a flexible environment in the database. The user can create own regulations/directives inside the database through relationships between the data. Consequently, the database will follow these set of rules making the database consistent. Additionally, MySQL databases can handle a lot of data efficiently and it is also scalable.

3

Method and System Overview

The method chapter describes the system overview and methods used in the project.

3.1 Extracting an Audio Signal

Initially, a song was needed to be loaded into the program from an audio file, and its corresponding spectrogram was plotted. However, the graph contained excessive data and therefore only the peaks were retained. This transformed the graph into a distinctive pattern, similar to a star map or constellation map. Each "star" in this pattern represented the loudest sound at a specific moment in the song, enhancing its distinguish ability from background noise.

Furthermore, the program included a functionality for recording audio directly from the computer's microphone and saving the recorded audio file on the computer.

3.2 Fingerprinting

The data extracted from the constellation map underwent a process known as fingerprinting. This involved using a hash function to create unique hashes based on the information derived from the constellation map. By utilizing the local peaks identified in the constellation map, these hashes served as compact representations of the song's acoustic characteristics. The fingerprints were then stored in a database for future recognition purposes.

3.3 Database Management

Efficient management of the database is crucial for handling the large amount of fingerprints and associated metadata for various songs. Therefore, SQL database was used due to its scalability and effective data retrieval. This approach optimized storage and retrieval operations and improved the overall performance and effectiveness of the system.

3.4 Matching

The main component of the audio recognition system was its ability to match recorded songs with their corresponding fingerprints stored in the database. The audio recognition algorithm systematically compared the characteristics of the recorded song with all available fingerprints, identifying the most closely matched fingerprint and retrieved its associated metadata.

3.5 Testing and Program Assessment

To ensure the effectiveness and reliability of the program, testing and assessment was conducted. Experiments was designed to evaluate the accuracy, response time, and matching capabilities of the program. By utilizing the microphone from the computer to record songs, the performance metrics such as response time and the number of hashes used for comparison are measured. These tests provided valuable insights into the program's performance and helped to identify areas for improvement.

4

Implementation

In this chapter, the implementation is presented and discussed in detail.

4.1 Reading an Audio File

The first step was to be able to read an audio file into Python.

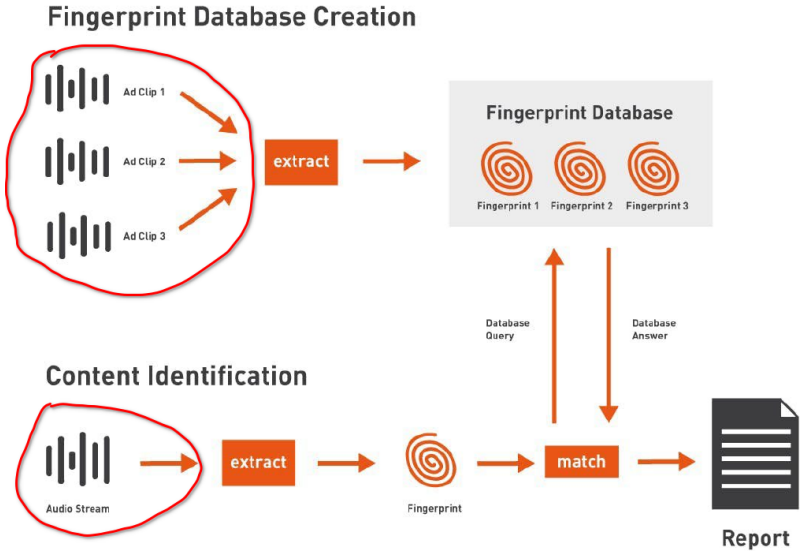

Figure 4.1: System overview.

The `read_wav` function is designed to read an audio file in WAV format with a standardized sample frequency of 48 000 kHz, ensuring consistency across different audio files. Below is an explanation of each step involved in the function.

File Conversion: The audio file specified by `filepath` is given as a parameter. The function begins by converting the input audio file to a single-channel (mono) audio signal with the standardized sample frequency of 48 000 kHz. This was done using the `AudioSegment.from_file` method from the `pydub` library.

Processing Audio Signal: The raw audio data from the converted audio segment was then extracted and converted into a NumPy array of 16-bit integers. This was achieved using the `np.frombuffer` function, which reads the raw audio data and converts it into a format suitable for numerical processing in Python. The resulting `audio_signal` array contains the audio samples of the converted audio file.

Segmenting the Song For testing purposes, two parameters, `stop` and `start`, were introduced to enable the segmentation of the song to simulate a recording.

Additionally, a function was created to initiate recording using the `PyAudio` library upon execution. This function is specifically designed to capture audio for the next 10 seconds. Subsequently, it generates a corresponding 10-second audio file.

4.2 Extracting an Audio Signal

Next step was to extract the audio signal. The point of this was to reduce the data used to make the algorithm more efficient.

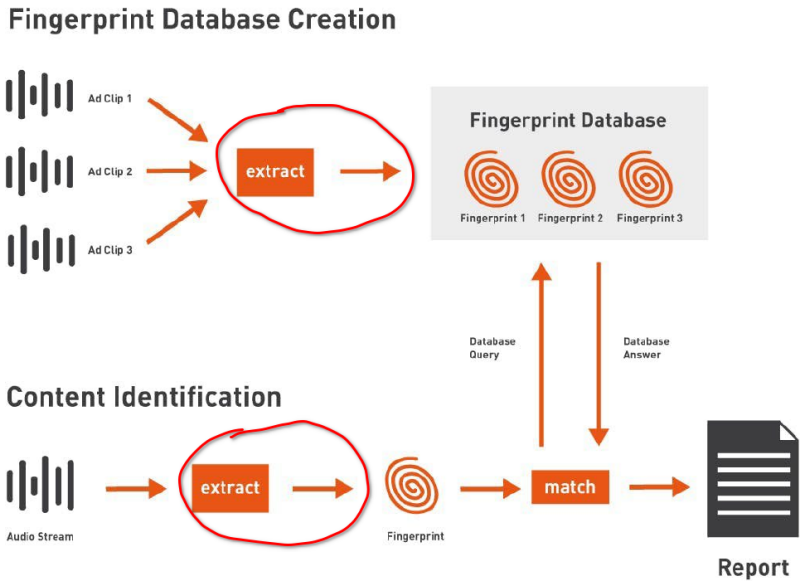

Figure 4.2: System overview.

4.2.1 Create Spectrogram Function

The `create_spectrogram` function generates a spectrogram from a given audio signal array. The spectrogram represents the signal's frequency spectrum over time, which is useful for analyzing the frequency content of the signal.

Function Definition: The function takes the following arguments

- `audio_signal`, which is the audio signal data from the `read_wav` function.
- `nperseg`, the number of samples per segment for the Short-Time Fourier Transform (STFT). This project used 1024.
- `noverlap`, The number of samples to overlap between segments. This project used 512.

Spectrogram Calculation: The function uses the `spectrogram` function from the `scipy.signal` module [22] to compute the spectrogram. The function returns the spectrogram matrix (`spec`).

4.2.2 Peak detection

The `peak_detection` function was designed to identify significant local maxima in a spectrogram.

Below is an explanation of each step involved in the function.

1. **Application of Maximum Filter:** The function begins by applying a maximum filter to the input spectrogram using the `maximum_filter` function from the `scipy.ndimage` module. This filter enhances the local maxima within the spectrogram, making it easier to identify significant peaks.

   Along with the input spectrogram, the size of each FFT window is set to 1024, determined by the filter parameter `PEAK_BOX_SIZE`.

   The filtered spectrogram is returned, which highlights the local maxima.

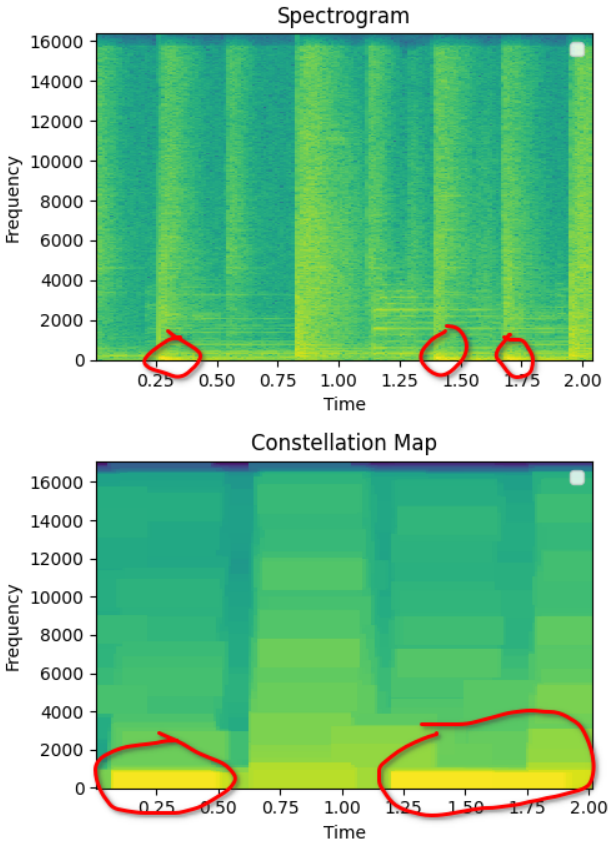

Figure 4.3: Demonstrates how local maximas are amplified.

2. **Boolean Mask and Local Maximas:** A boolean mask, `peak_mask`, was created by comparing the original spectrogram with the filtered spectrogram. Each element in `peak_mask` is set to `True` if the corresponding element in the original spectrogram has the same value as in the filtered spectrogram, and `False` otherwise.

   The coordinates of the `True` values in `peak_mask` extracted and stored to `x_peaks` `y_peaks` respectively, representing the indices of the local maxima in the spectrogram.

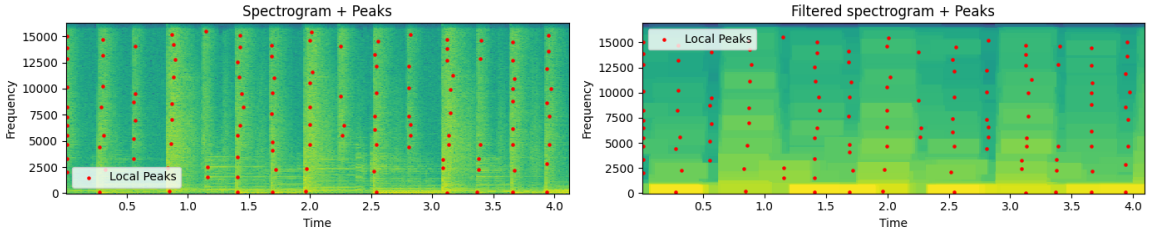

Figure 4.4: Spectrograms with their peak positions marked as red dots. Note that the peak positions are the same in both spectrograms.

3. **Threshold Filtering:** The peaks are filtered based on a predefined threshold. Only peaks with values greater than this threshold are considered significant. The `thresholded_peaks` variable indicates which peaks meet this criterion.

4. **Returning Filtered Peak Coordinates:** The function returns the coordinates of the significant peaks that meet the threshold criterion. These coordinates are the column and row indices of the significant peaks in the spectrogram.

4.2.3 Create Constellation Map Function

The `create_constellation_map` class method generates a constellation map from an audio file, utilizing all preceding functions to create a constellation map as depicted in Figure 4.5. This map visually represents the prominent peaks in the spectrogram.

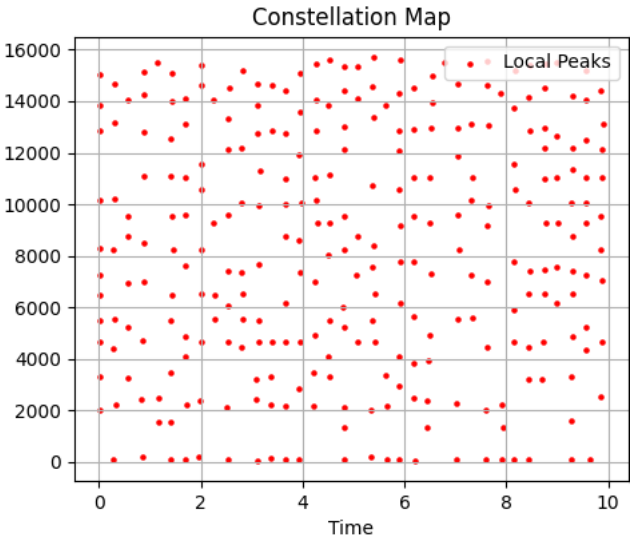

Figure 4.5: A Constellation Map. The red dots represents local peaks.

1. **Reading the Audio File:** The method starts by reading the audio file specified by `filepath`. It retrieves the audio signal data, potentially trimming it to the specified time range defined by `start` and `stop`.

2. **Calculating the Spectrogram:** Next, the method calculates the spectrogram of the audio signal. This process breaks down the audio signal into its frequency components over time, producing a two-dimensional matrix representation.

3. **Detecting Peaks:** The method then identifies the significant peaks in the spectrogram using a peak detection algorithm. These peaks represent prominent frequency components in the audio signal.

4. **Returning Results:** Finally, the method returns several components:

   - `spec`: The spectrogram matrix representing the frequency content of the audio signal over time.
   - `y_peaks`: The indices of the peaks along the frequency axis.
   - `x_peaks`: The indices of the peaks along the time axis.

4.3 Fingerprint

After generating a constellation map specific to a song, the next step was to utilize and preserve this data for efficient comparison with other datasets. This section

explores the concept of fingerprinting. A fingerprint serves as a distinct signature derived from the constellation map. This compact representation enables rapid identification and matching of audio segments.

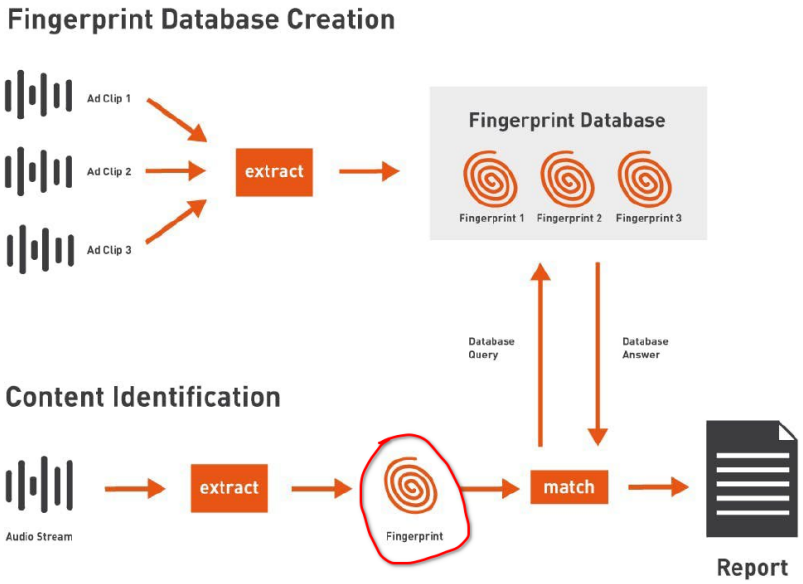

Figure 4.6: System overview.

In this section, the project's fingerprinting implementation is described in detail.

4.3.1 Hashing

The primary goal of hashing is to ensure the uniqueness of each point and reduce the data saved for fast retrieval. This is achieved by generating hashes from pairs of peaks.

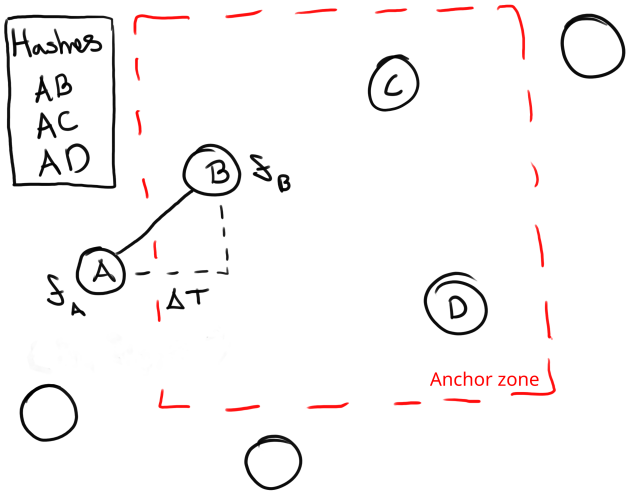

Figure 4.7: How the data is collected.

The method involves creating an anchor zone, where each data point within this zone is referenced back to a central anchor point. This approach collects data from the anchor point to every other point within the zone, as illustrated in Figure 4.7. For *every* point in the constellation map, this procedure is applied. This implies that, at some stage, every point serves as an anchor point.

For this purpose, the Python's built-in hash function was utilized, `hash()`. Each hash is generated based on the frequency of the anchor point, the frequency of the point in the zone, and the time difference between them, denoted as `hash(`$(f_a$`,` f_b`,` $\Delta T)$`)`. Refer to Figure 4.7 for a visual representation of the process.

Once a pair has been created, it is stored as a hash in the database with the following information:

Table 4.1: Pair point AB. Description of Hash, Anchor time, and Track ID

Hash	Anchor time	Track ID
`hash`$((f_A, f_B, \Delta T))$	Point A_n time	`hash`((song_name, artist))

Pairing points offers a distinct advantage, as paired points provide significantly greater uniqueness compared to a single point. For instance, considering a pair point AB in table 4.1, if each point contains 10 bits of anchor frequency data and the time difference between them can be represented by an additional 10 bits, the combined information amounts to 30 bits. This yields a total of $2^{30} = 1073741824$ possibilities, which is notably larger than the $2^{10} = 1024$ possibilities for a single point [6].

In line with the approach used for anchor points, we avoid storing the song name and artist directly. Instead, a unique hash was created from the song name and artist, which was is referred to as the Track ID as seen in Table 4.1. This hash serves as an identifier that can be associated with the specific song and artist.

For each anchor point, all hashes are stored within its zone, along with the timestamp for the anchor point and the Track ID. All of this information is compiled into a comprehensive list. This list constitutes the fingerprint.

4.3.2 Storing a Fingerprint

The `write_fingerprint` class method generates and stores a fingerprint for an audio track, utilizing several preceding functions to achieve this.

1. **Generating the Constellation Map:** Using the `create_constellation_map` function, the method generates the constellation map for a certain song.
2. **Hashing Points:** These points form the constellation map are then hashed using the `hash_points` method. This hashing process produces unique identifiers for each point, ensuring the fingerprint's uniqueness.
3. **Preparing Data for Storage:** The method filters out any empty hashes and prepares the data for writing, completing the fingerprint for the song. Table 4.2 shows how the fingerprint is stored as a list.

Table 4.2: List of stored points from Figure 4.7. Each point pairs hash, anchor point time and track ID are saved in the same table. Note: The pair column is not stored, it is just for demonstration purposes.

Pair	Hash	Anchor time	Track ID
AB	`hash`$((f_A, f_B, \Delta T_{AB}))$	Point A_i time	`hash`((song_name, artist))
AC	`hash`$((f_A, f_C, \Delta T_{AC}))$	Point A_i time	`hash`((song_name, artist))
AD	`hash`$((f_A, f_D, \Delta T_{AD}))$	Point A_i time	`hash`((song_name, artist))
⋮	⋮	⋮	⋮

These methods work together to generate and store fingerprints data for audio tracks, allowing for efficient identification and matching of audio tracks based on their unique acoustic characteristics.

4.4 Database

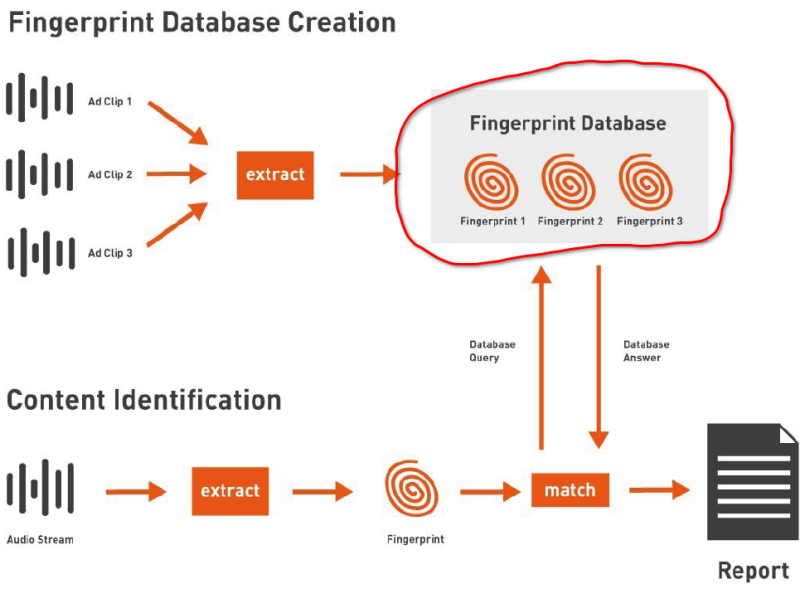

Figure 4.8: System overview.

In Python, a function was created with mysql.connector library to be able to connect to the database through Python.

A table called hash_table was created in the database with three columns named "id", "hash" and "point_A_time". Additionally, the hash column was indexed which made the search quicker. Furthermore, another table was created named metadata which contained the columns "id", "song_name", "artist". This was where the metadata was stored in the database.

A function was created in order to upload data to both tables. The function has parameters that are used to access the artist and the title of the song. Once the program received the metadata, it was uploaded to the table metadata.

Every song had a list contained with the hashes and the anchor times and the function looped through the list to upload to the database.

Likewise, two functions were created to read from the database, specifically retrieve data from the db and metadata tables. The function that was created for retrieving data from the db table had a query which selected the id and the point_A_time for a specific hash. Subsequently, the id retrieved from the db table was sent to the other function which was used to retrieve the metadata of that song which the id belonged to. Following the retrieval of the metadata, the program printed out the artist and the song name.

4.5 Matching

The final part was to implement functions that is able to compare fingerprints and score how good these fingerprint match.

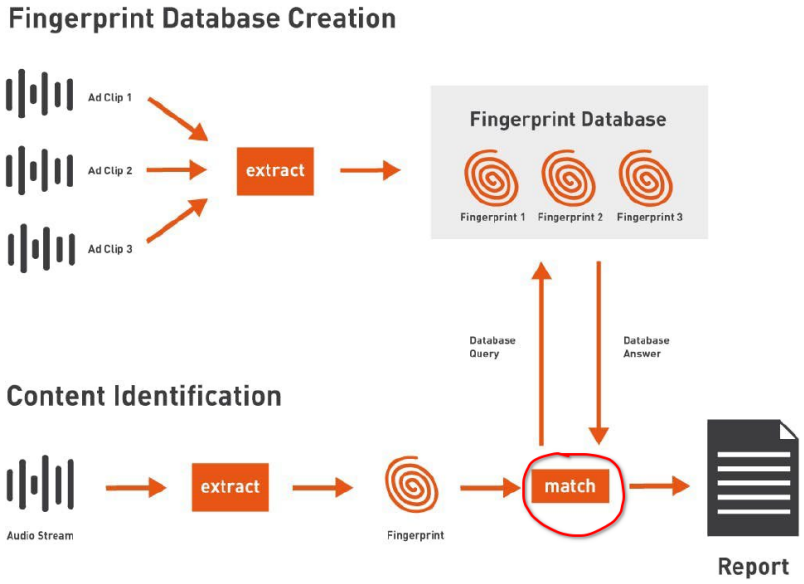

Figure 4.9: System overview.

4.5.1 The Compare Function

A fingerprint of the recorded song is generated by first creating a constellation map of the song and then generating hashes from this map, as described earlier. For demonstration purposes, the recorded fingerprint is compared with the matching fingerprint in the database, as illustrated in Figure 4.10.

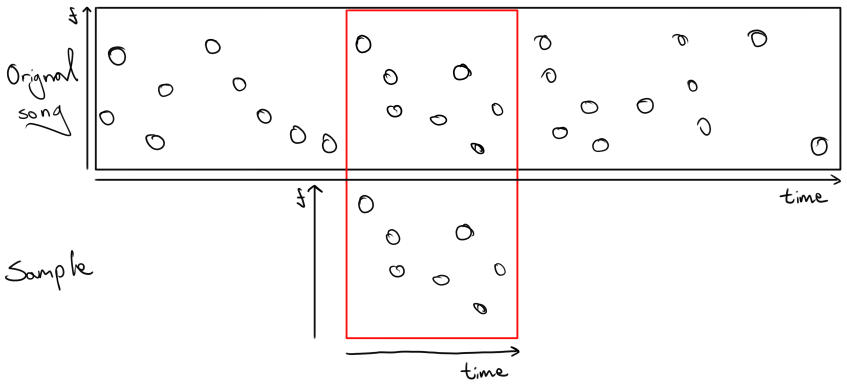

Figure 4.10: The original song (database) and the sample (recorded) each have their own constellation map and time axes. The red square indicates where in time the recorded song matches the database song.

In an ideal scenario, every hash matches exactly with its counterpart in the database. However, due to factors such as noise, this perfect alignment is rarely achieved in reality. Some points inevitably deviate.

The following steps are required to compare a recorded fingerprint with one in the database:

1. Each hash in the fingerprint is checked against the database to find matching hashes. Recall how the hash looks like $\texttt{hash}((f_A,\ f_B,\ \Delta T_{AB}))$. This ensures that the anchor point f_A is the same, the pairing occurs with the same point in the anchor zone f_B, and the time offset between them ΔT_{AB} are identical.
2. For every match found, the time-offset between the database hash's anchor point and the recorded song's anchor point is calculated and stored under the corresponding track ID.

To delve deeper into this process, the first step simply involves verifying whether the point pair possesses the correct frequency and offset by comparing the hash.

In the second step, the issue lies in not knowing which segment of the recorded song is being played. Therefore, relying solely on hash matching is insufficient, as longer songs might produce numerous hashes, thereby increasing the risk of a false positive match.

However, the relative offset between two points should be consistent between the database fingerprint and the recorded fingerprint if they are from the same song. Therefore, when numerous offsets are identical, it indicates that the recorded fingerprint corresponds to a specific segment of the song.

In Figure 4.11, these scenarios are depicted.

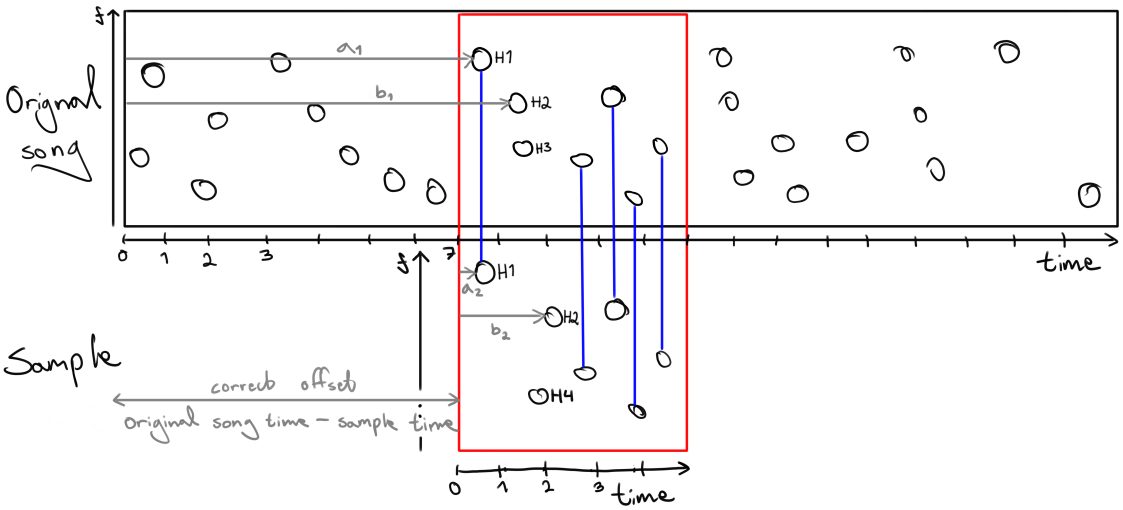

Figure 4.11: For the points labeled H1, H2, and so forth, each represents their respective hashes. Blue lines connect points that match, while those without blue lines do not match.

Let us go through all three scenarios:

1. When calculating the differential for H1, $a_1 - a_2$, the result shows an offset of zero in recorded time and seven seconds in database fingerprint time. Every other match with the correct match will yield the same value.

2. However, for H2, $b_1 - b_2$ does not equal seven seconds in database fingerprint time. Any other hash match with the incorrect time point will yield an arbitrary time-offset value.

3. For H3 and H4, the hashes are expected to be the same but the frequency does not match, so calculating the offset is unnecessary. It cannot be a correct match.

For the remaining points in Figure 4.11, the recorded fingerprint shares the same offset as the first offset because the points are aligned. This indicates a high likelihood of being the correct match, with an accuracy of the number of hashes in the recorded fingerprint divided by the total number of hashes, $\frac{5}{7} = 71\%$.

Below is a table where the offset is calculated for each point.

Table 4.3: Here we see that all the matches have the same offset.

Original Song (s)	Sample (s)	Offset (s)
7.6	0.6	7.0
8.3	2.3	6.0
8.4	1.4	-
9.7	2.7	7.0
10.2	3.2	7.0
10.9	3.9	7.0
11.3	4.3	7.0

5

Result

5.1 Database

The database consists of two tables called Metadata and Hash_table based on the data from the songs. Furthermore, the database contains 100 songs from the 1960's to the 2020s covering different genres. The various genres include Hard Rock, Rock, Pop, Hip-Hop, Reggae, Soul, House, Country and Electro. Moreover, the songs are varied from 3:30 minutes to 5 minutes in length.

5.2 Tests: Accuracy, Response time and Matching

We discuss next the results of the tests and assessments of the accuracy, response time and matching capabilities. The songs "Diamonds", "Dancing in the moonlight", "You shook me all night long", "Levels" and "Down under" were used. All these songs are from different genres. This is to ensure representations of different genres and then allowing testing the accuracy on the different genres.

The songs in the tests were cut into a smaller audio segment in order to simulate a 10 second recording of a song. It was done in the beginning, middle and at the end of each song. Subsequently, the audio segment was used to match with a fingerprint in the database.

The response time test was done when the database had 10, 50 and 100 songs. Five songs were used in the experiment. Additionally, each song was from a different genre in order to see if the genre also affected the matching. The genres were Electronic, Pop, House, Hard Rock and Rock. The response time was measured 100 times and the average was taken of those test for each song.

Subsequently, the response time with 1000, 5000 and 10 000 songs in the database was tested but only with one song. However, the database was simulated to contain these sizes of the database when measuring the response time. The reason for this was in order to save time.

Furthermore, another experiment was used to see if the matching got affected by the number of songs in the database. The size of the database had 10, 50 and 100 songs in each test. This assessment was also done with five songs.

5.2.1 Results of microphone with 10 songs

This the outcome of the microphone test with 10 songs in the database. Table 5.1 shows the details of the results.

Table 5.1: Results with microphone and 10 songs in the database

Song name	Start time[s]	Avg. response time	# of hashes	# of correct hashes	Correct offset	Correct match
Diamonds	4	0.30	569	17	Yes	Yes
	110	0.54	1027	3	Yes	Yes
	180	0.72	1405	5	Yes	Yes
Dancing in the moonlight	3	0.2	829	33	Yes	Yes
	60	0.64	2881	4	Yes	Yes
	120	0.5	2231	2	Yes	Yes
Levels	3	0.14	516	13	Yes	Yes
	74	0.43	1911	3	Yes	Yes
	156	0.41	1788	3	Yes	Yes
You shook me all night long	3	0.05	89	4	Yes	Yes
	88	0.42	1849	7	Yes	Yes
	120	0.42	1855	6	Yes	Yes
Down under	4	0.29	1177	17	Yes	Yes
	110	0.29	1236	3	Yes	Yes
	180	0.37	1469	7	Yes	Yes

5.2.2 Result of microphone tests with 50 songs

The table below displays the results for 50 songs in the database:

Table 5.2: Results with microphone and 50 songs in the database

Song name	Start time[s]	Avg. response time	# of hashes	# of correct hashes	Correct offset	Correct match
Diamonds	3	0.11	300	7	Yes	Yes
	110	0.2	761	2	Yes	Yes
	180	0.39	1648	12	No(53 s)	Yes
Dancing in the moonlight	3	0.43	838	24	Yes	Yes
	59	1.1	2369	5	Yes	Yes
	126	0.6	2516	2	Yes	Yes
Levels	3	0.31	531	11	Yes	Yes
	72	0.73	1567	10	Yes	Yes
	156	0.71	1739	5	Yes	Yes
You shook me all night long	3	0.14	153	2	Yes	Yes
	86	0.72	1520	7	Yes	Yes
	120	0.71	1529	6	Yes	Yes
Down under	4	0.63	1309	12	Yes	Yes
	109	0.6	1229	5	Yes	Yes
	184	0.64	1348	3	Yes	Yes

5.2.3 Result of microphone tests with 100 songs

The results of the tests with microphone and 100 songs in the database.

Table 5.3: Results with microphone and 100 songs in the database

Song name	Start time[s]	Avg. response time	# of hashes	# of correct hashes	Correct offset	Correct match
Diamonds	4	0.29	500	6	Yes	Yes
	110	0.4	767	5	Yes	Yes
	180	0.63	1293	8	No(66 s)	Yes
Dancing in the moonlight	3	0.4	779	31	Yes	Yes
	59	1.16	2473	8	Yes	Yes
	124	0.87	1837	2	Yes	Yes
Levels	3	0.26	643	15	Yes	Yes
	69	0.47	2049	4	Yes	Yes
	154	0.49	2115	4	Yes	Yes
You shook me all night long	2	0.07	168	2	Yes	Yes
	84	0.32	1346	6	Yes	Yes
	118	0.29	1564	4	Yes	Yes
Down under	4	0.3	1180	12	Yes	Yes
	109	0.28	1143	3	Yes	Yes
	178	0.29	1149	2	Yes	Yes

5.2.4 Time response with 1000, 10 000 and 100 000 size database

Table 5.5 table displays the results of the response time of the database when it has 1000, 5000 and 10 000 simulated songs in the database.

Table 5.4: Response time for different sizes of databases

song name	**Avg. response time**	**# of songs in database**
Diamonds	0.38	1000
Diamonds	0.78	5000
Diamonds	0.87	10 000

5.2.5 Correct matches with microphone

The table 5.5 displays the results of the matching accuracy with the microphone based on the different amount of songs in the database.

Table 5.5: Song accuracy for different sizes of databases

# of correct matches	# of incorrect matches	# of songs in database	% of correct songs
10	0	10	100
43	7	50	86
85	15	100	85

6

Discussion & Conclusion

This project was undertaken to design the core functionalities of Shazam and evaluate the performance of the algorithm. The experiments has found that the response time is fast and the matching accuracy is approximately 85% in general using the microphone. Thus, it indicates that the objectives are met, as the algorithm is capable of identifying songs and retrieving the correct metadata.

However, during testing, the accuracy significantly dropped when the recording is from a later segment of the song. Even in the ideal case this was the case. To speculate the reasons behind this, we think it may be because of windowing. Due to the fact that the FFT segments do not line up and therefore the windowing filters differently for the database and the recorded sample.

We also ran into memory error when trying to upload approximately 100000 songs into our database. The issue is that the computer have 16 GB or ram and possibly the computer only allows the program to allocate 8 GB. This means that if the table is larger than 8 GB of RAM it can not be loaded.

The solution to this is to split up the hash_table into multiple tables. In order to design this solution, we would need to create our own hash function that can determine which table the hash should be located in and then load that table to search for a specific hash. This would fix the memory error issue but it would be a little slower in terms of speed. Thus, further development would be creating an own hash function.

Furthermore, a general 80% matching accuracy suggests for further improvement. Future studies should focus on improving the matching accuracy when the recording is from a later segment in the song. The study should be repeated using a larger database to emulate the original Shazam.

Bibliography

[1] A. Wang, “An industrial strength audio search algorithm,” in *Proc. 4th Int. Symp. Music Inf. Retrieval (ISMIR)*, Oct. 2003.[Online]. Available: https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf, Accessed on: 2024-01-18.

[2] Shazam, “About us”. [Online]. Available: https://www.shazam.com/company(accessed on: 2024-01-20)

[3] L.Sevgi, “Fourier transforms and Fourier Series” in Electromagnetic Modeling and Simulation, 1st ed. John Wiley & Sons Inc, 2014, ch. 4, pp.71-94. [Online]. Available: https://ieeexplore.ieee.org/xpl/ebooks/bookPdfWithBanner.jsp?fileName=6798248.pdf&bkn=6798066&pdfType=chapter, Accessed on: 2024-01-25

[4] HowStuffWorks, "Understanding Sound Waves and How They Work ", 1970. [Online]. Available:https://science.howstuffworks.com/sound-info.htm?srch_tag=vzherf7j32o4cek7qr4kdawnjd3o2vxf(accessed on: 2024-06-09)

[5] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8650444

[6] C.Macleod, “abracadabra: How does Shazam work?”, Cameron Macleod. [Online]. Feb. 19, 2022. Available: https://www.cameronmacleod.com/blog/how-does-shazam-work(accessed on: 2024-01-18).

[7] SciPy documentation.*scipy.signal.spectrogram*. Accessed: 2024-01-25. [Online]. Available:https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html

[8] Wikipedia, “Spectrogram”, 2003. [Online]. Available: https://en.wikipedia.org/wiki/Spectrogram(accessed on: 2024-05-14).

[9] Javatpoint, “What is Hashing in C”. [Online]. Available: https://www.javatpoint.com/what-is-hashing-in-c (accessed on: 2024-05-14).

[10] Geeksforgeeks, “Python hash() method”. [Online]. Available: https://www.geeksforgeeks.org/python-hash-method/(accessed: 2024-05-14)

[11] Wikipedia, “Hash table”, 2001. [Online]. Available:https://en.wikipedia.org/wiki/Hash_table(accessed on: 2024-06-07)

[12] Geeksforgeeks, “Libraries in Python”. [Online]. Available: https://www.geeksforgeeks.org/libraries-in-python/(accessed on: 2024-05-14)

[13] SciPy documentation. *The main SciPy namespace*. Accessed: 2024-05-14. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/main_namespace.html

[14] Py documentation. *unittest — Unit testing framework*. Accessed: 2024-05-14. [Online]. Available: https://docs.python.org/3/library/unittest.html

[15] tqdm documentation. *tqdm*. Accessed: 2024-05-14. [Online]. Available: https://tqdm.github.io/

[16] W3schools, “What is Json?”.[Online]. Available: https://www.w3schools.com/whatis/whatis_json.asp(accessed on: 2024-05-14)

[17] Py documentation. *itertools — Functions creating iterators for efficient looping* . Accessed: 2024-05-14. [Online]. Available: https://docs.python.org/3/library/itertools.html

[18] Matplotlib documentation. *Matplotlib: Visualization with Python*. Accessed: 2024-05-14. [Online]. Available: https://matplotlib.org/

[19] PiPy. *PyAudio 0.2.14*. Accessed: 2024-06-08. [Online]. Available: https://pypi.org/project/PyAudio/

[20] W3schools, “Python MySQL”. [Online]. Available: https://www.w3schools.com/python/python_mysql_getstarted.asp(accessed on: 2024-05-14)

[21] MySQL documentation. *10.5 cursor.MySQLCursor Class*. Accessed: 2024-05-14. [Online]. Available: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor.html

[22] SciPy documentation. *scipy.signal.spectrogram*. Accessed: 2024-06-18. [Online]. Available: https://docs.scipy.org/doc/scipy-1.13.0/reference/generated/scipy.signal.spectrogram.html

INSTITUTIONEN FÖR något ämne
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS