Department of Computer Science
Aalborg University, Denmark
June 12th, 2008

MASTER'S THESIS

A Generic Backend for Fast Audio Fingerprint Matching

Author:
Anders Skovsgaard

Supervisor:
Christian S. Jensen

Department of Computer Science
Aalborg University

DAT 6

TITLE:
A Generic Backend for Fast Audio Fingerprint Matching

PROJECT PERIOD:
Spring semester 2008

PROJECT GROUP:
d622b / Computer Science

AUTHOR:

Anders Skovsgaard

SUPERVISOR:
Christian S. Jensen

NUMBER OF COPIES: 3

TOTAL PAGE COUNT: 42

Synopsis

This thesis documents the development of a generic backend capable of finding the best match for audio fingerprints. It can be used with various fingerprint generators and audio sources subject to noise. The best matching fingerprint is determined by a scoring system, where the scores depend on the amounts of correctly positioned n-grams in the fingerprints. The developed algorithm uses n-grams and hash tables for fast lookup. Additionally, a similarity measure is developed to quickly create candidate sets. The candidate set contains a subset of the database fingerprints, that is estimated as a possible match. The search algorithm guarantees no false dismissals and parameters can be adjusted to alter the reliability of the results. Experimental performance studies shows that the solution is orders of magnitude faster than related work.

Preface

This master's thesis is a result of a four-month DAT6 project at the Department of Computer Science, Aalborg University. It takes its outset in an article composed at DAT5 during autumn of 2007. The present research builds on that article, and some of its findings are used as motivation for this thesis. The relation to the past work is covered in Section 2.3.3.

Reading Notes

References will be represented in the form [20]. The list of references can be found on page 40. This thesis starts by describe relevant background material in order to understand the next part, which covers the problem. A solution for this problem will then be described, and experimental performance studies follows. Finally, the thesis contains conclusions and proposals for future work.

Source Code and Tools

The development of this project involved several software tools. The algorithms are developed in C# 2.0 and run on the .NET 3.0 platform. For the implementation, Microsoft Visual C# 2005 Express Edition was used as the integrated development environment (IDE). The database server used the relational database management system, Microsoft SQL Server 2005 running on a Microsoft Windows 2003 Enterprise Server.

Source code is available from the enclosed CD-ROM.

Contents

Contents **2**

1 Introduction **4**

2 Preliminaries **6**
2.1 Audio Fingerprints 6
2.1.1 Generating String Fingerprint 7
2.1.2 Time Shifting 9
2.2 String Comparing with N-grams 10
2.3 Related Work 10
2.3.1 Existing Audio Identification Systems 10
2.3.2 Search Algorithms 11
2.3.3 Previous Work 13
2.4 Problem Statement 14

3 Scalable Fingerprint Matching **15**
3.1 Architecture 15
3.2 Fingerprint Matching using N-Grams and Hash Tables 16
3.2.1 Indexing the Database Fingerprints 16
3.2.2 Querying the Database 17
3.3 Limits of the Search Algorithm 19
3.4 Filtering Trivial Fingerprints 20
3.5 Push rather than Brute Force 21
3.5.1 Early Termination 22
3.5.2 Non-Unique N-gram Search 23
3.5.3 Supporting Streaming Fingerprints 24
3.5.4 Pseudo Code for the Proposed Optimization 25
3.6 Other Optimizations 27
3.6.1 Table Division 27
3.7 Performance Factors 28
3.7.1 Running Time 28

3.7.2 Space Consumption 29
3.7.3 Memory Usage 30

4 Experimental Performance Study **31**
4.1 Test Setup 31
4.2 Choice of Parameter Settings 32
4.2.1 N-gram Length 32
4.2.2 The *MinMatch* Parameter 32
4.2.3 The *WinningFactor* Parameter 33
4.3 Query Performance 35
4.3.1 Comparison with Previous Work 35
4.3.2 Worst Case Database Scenario 36
4.3.3 Summary 37

5 Conclusion **38**

6 Future Work **39**

REFERENCES **40**

Chapter 1

Introduction

Music has existed as long as anyone can remember. It is everywhere, from the natural sounds such as birdsong to music on the radio. Music exists on numerous mediums and has been subject to comprehensive development through the last centuries. The development has gone from analogue mediums such as gramophone records and cassettes to digital compact discs and files on computers and mobile phones. And the development continues this day with new medium and formats for better sound reproduction emerging.

Today, it is possible to access catalogues with millions of music files through the Internet and within minutes be able to listen to a song on a desktop computer. This development has also moved to mobile phones where music files can be found and downloaded or retrieved from a desktop computer and listened to while on the move. With all this music everywhere, people may find themselves listening to a song they like, but do not know the name of. This is a problem for both the producers of music and the consumers. The goal of the producers is to sell music and when the consumers can not find the music they are interested in, the producers loose costumers.

The problem can be addressed by audio recognition software that has the ability to record a piece of a song and return meta data such as the artist, title, and where to buy the song. Several methods exists for audio recognition, but most of today's audio recognition software is proprietary. The methods all share some basic steps in order to recognize a song. First, a piece of the song is recorded. This recording is then compressed and transferred to a backend server over the Internet. The backend server subsequently performs a lookup and returns the meta data of the song.

Since the recognition software is proprietary, it is not known how the lookups in the database are performed or which features the software extracts from the audio signal to perform the lookups. Much effort has been put into the area of finding robust features from audio that are not as vulnerable to noise and distortion. These

features could, e.g., be beats per minute or dominant frequencies present in the signal. Together, these features could form a fingerprint of the signal. But since an audio signal from a microphone can contain additional speech, background noise, and other sources of noise, it is impossible to create the exact same representation of a signal as the original. Thus, some margin of error is needed when searching the database for a given fingerprint.

The database lookup requires, in addition to a robust fingerprint algorithm, a fingerprint matching algorithm. This matching algorithm should be able to find a match in spite of varying numbers of errors in the fingerprints and perform fast lookups in databases containing millions of fingerprints. It should be able to search all fingerprints in the database in order to be positive that no better fingerprint exists.

This thesis introduces a complete framework for fast fingerprint matching capable of searching millions of fingerprints. It works with any chosen features from an audio signal given that the features yield a sufficient separation between fingerprints of distinct songs and also contain some similarities when they derived from the same song. The proposed framework is scalable, searches while recording, and returns a match with no false dismissals when enough data from the signal is present. The representation of a fingerprint is a string of characters where the size depends on the length of the signal. Each character represents a small piece of the signal and can be assigned a different meaning depending on the extracted features.

String searching is a well explored research area, and this thesis introduces new ideas for string matching built on top of existing search techniques.

The remainder of this thesis is organized as follows. Initially, the preliminaries are descriebed in Chapter 2 to give some background knowledge before introducing the related work and the problem statement. Then the overall architecture is described in Chapter 3 followed by a description of previous work used by this thesis. In Section 3.3 the limits of the previous work is outlined, followed by the proposed solution. The algorithm is then described in details and possible optimizations are given before the experimental performance studies, in Chapter 4. This thesis is closed by a conclusion and future work proposals.

Chapter 2

Preliminaries

In order to understand the problems with fingerprint generators and fingerprint matching, some background knowledge is described in this chapter. Several problems that should to be solved in order to gain fast lookup speeds on large collections of fingerprints will be outlined.

Section 2.1 describes the difficulties in creating a robust fingerprint. In Section 2.2, a fuzzy string search technique, which is used in this thesis, is outlined. Related work and previous work are covered in Section 2.3. Some parts of the previous work are used in this thesis, and problems that arose during that work serve as motivation for this thesis.

2.1 Audio Fingerprints

An audio signal is represented as a wave-form that shows the amplitude over time. In Figures 2.1 and 2.2, two different audio signals are illustrated. Having the exact same audio signal gives the exact same wave-form, but when the signal changes, so does the wave-form. A signal with background noise therefore changes the wave-form. This makes it difficult to find a reliable match between two audio signals when these contain noise. To obtain a robust representation of an audio signal, only important features from the signal should be preserved. This can reduce the differences between two noisy signals. Often the query fingerprints contain different kinds of noise, and the database fingerprints are generated free from noise. The extracted features should be used when generating fingerprints of audio signals. Some fingerprint generators [10] make use of the most dominant frequencies in the signal by using the Fourier Transformation [8], which outputs the amplitude of each frequency present in the signal. Along with other features, such as the Beats-Per-Minute, a high quality fingerprint generator can be developed. A high quality fingerprint generator is capable of creating different fingerprints for two

different audio signals and creating similar fingerprints for identical audio signals even though they might be subject to "random" noise. By only extracting features that are of relevance in a given context, different wave-forms ideally give the same representation.

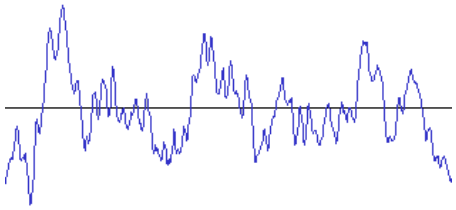

Figure 2.1: Audio signal in wave-form.

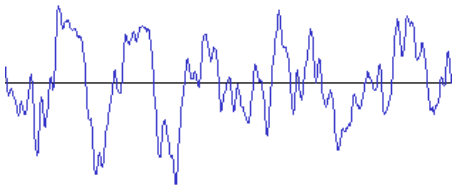

Figure 2.2: Audio signal in wave-form.

But the sources of noise are heterogeneous, such as additional speech, a distorted signal because of different speakers or microphones, or "random" background noises. This makes it the difficult to develop a completely robust fingerprinting technique. Therefore, the search method must be able to contend with fingerprints that contains different amounts of errors.

2.1.1 Generating String Fingerprint

String searching is a well-explored research area and is well-suited for representing audio fingerprints. Numerous index types and search algorithms exist for strings and fuzzy string search. Therefore, this thesis utilizes strings as representation for fingerprints and builds on top of existing string search techniques.

In order to build a string fingerprint which is based on features in a signal, first a codebook of mean feature vectors should be prepared. These mean vectors a created from a representative set of samples from the audio signal database. Features are extracted continuously for each block of the samples, and a vector for each block of audio is created. This is illustrated in Figure 2.3. By using the K-means clustering algorithm [6], k numbers of mean vectors are found. The k mean vectors are each assigned a character. The fingerprint generator used in this thesis, uses a codebook of 64 vectors, each of 16 dimensions. Therefore, the BASE64 charset can be used to represent the vectors. Each vector correlates to the Bark scale [4], that is a scale to measure the subjective human perception of sounds.

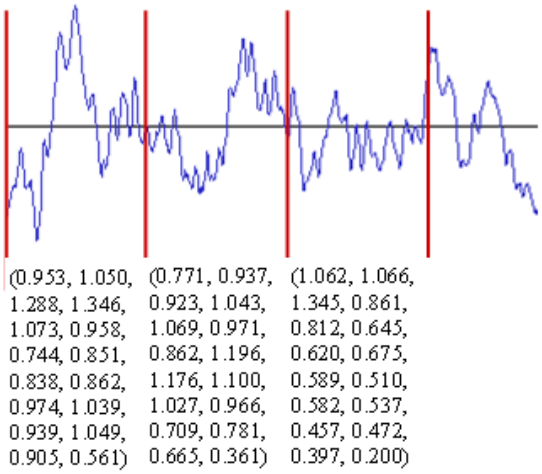

Figure 2.3: Vectors created from features extracted from blocks of the audio signal.

Having build the codebook, audio signals can be converted to fingerprints. This is done by extracting the feature vectors for each block of the signal, and by using a nearest neighbour search on the vector, the nearest codebook vector is found. Finally, the corresponding codebook vector character is assigning. Figure 2.4 illustrates an audio signal that is divided into blocks of audio. Each block has been compared to the codebook and the nearest vector is assigned to the block. When an audio block has been assigned a vector the correlated symbol can be assigned to that block as illustrated in Figure 2.4. The result of these operations is a string fingerprint for each audio signal in the database. As audio signals may vary in size, the fingerprints are of different lengths.

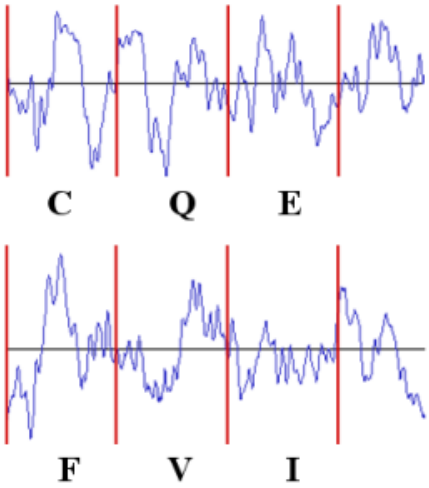

Figure 2.4: Characters assigned to each block of the audio signal.

2.1.2 Time Shifting

The recording of an audio signal may begin at random locations in the signal. It could happen that the recording begins in the middle of an audio block that exists in the reference database signal. This time shifting can influence a query in a way where an query audio signal, identical to the reference audio signal, obtains a different fingerprint. This is illustrated in Figure 2.5 where two identical audio signals obtains different fingerprints because they start at different locations. By starting at different locations one audio block may be dominated by the signal in a neighbor block in such way that the fingerprint assigns the neighbor symbol. Therefore, the fingerprint matching algorithm has to be able to take into account that characters can be positioned "incorrectly" due to time shifting.

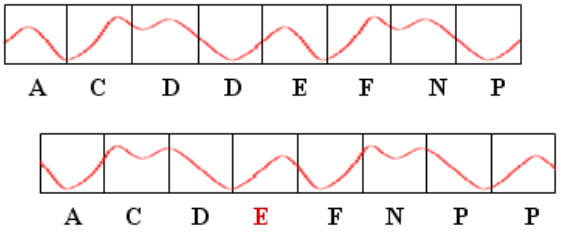

Figure 2.5: Time shifting that influences the fingerprint.

2.2 String Comparing with N-grams

It will be mentioned in Section 2.3 that string comparing can be done in different manners, but this thesis uses n-grams because other related methods for approximate substring matching are not performing faster [20]. An n-gram is a continuous division of a string in sub-strings of length n. Having a string s1 = "example" and a n-gram length of 2 gives the following n-grams: "ex", "xa", "am", "mp", "pl", and "le". Having another string s2="exomple" that looks much like s1 gives the following n-grams: "ex", "xo", "om", "mp", "pl", and "le". It is clear to see that s1 and s2 contains several similar n-grams. This is the intuition behind n-gram string matching. The greater similarity between two strings, the more n-grams they share. As the length of the n-gram increases, the number of n-grams in a string decreases. Formally, the number of n-grams of length n in a string of length s is $s-(n-1)$. Consequently, the number of common n-grams between two strings is reduced as n increases.

2.3 Related Work

The following section covers related work in terms of existing audio identification systems and approximate string search, as well as previous work by the author.

2.3.1 Existing Audio Identification Systems

There already exists several audio identification systems for free usage. Products for the desktop computer and mobile phones are available for free download and are widely used. The next sections will describe some of the most popular products for audio identification.

TrackID

Since the cell phone, W850i by Sony/Ericsson, was introduced in 2006 [17], all high-end mobile phones from Sony/Ericsson have been equipped with a native music recognition application called "TrackID". TrackID works by recording music with the built-in microphone, and after a fixed amount of time, it sends the recorded signal to a backend server. The backend server then returns the matching song. The application is proprietary, which means none of the mechanism behind are known. It is not known which features are extracted, nor are their representations known. Also, it is not known how the backend server performs the database lookup. TrackID uses Gracenote's extensive music database in order to recognize the music requested by the users. Gracenote collects its music from all major companies in the music industry and indexes it for fast retrieval. Gracenote sells these services to

various companies [5]. Also, desktop music programs such as iTunes use Gracenote technology to tag music files with the correct meta data.

Tunatic

Another audio identification system called "Tunatic" runs only on desktop computers [18]. It identifies all genres except classical music. The recognition is based on audio samples of various lengths. When enough data has been recorded, Tunatic returns meta data of the found song. Again, this requires an extensive database of music, but rather than buying this from music companies like Gracenote does, Tunatic gets it from its users. Users can create fingerprints for every song on their computer and send these fingerprints to Tunatic's backend server along with meta data of the song. With enough users doing this an extensive music database can be built without costs other than server and network expenses.

As with the TrackID application, Tunatic is proprietary, and no information is available on how the fingerprints are created or how the database lookups are performed.

MusicBrainz's "FutureProofFingerprintFunction"

MusicBrainz has made a description of a fingerprint functioning they called "FutureProofFingerprintFunction" (FPFF) [11]. This was indented to replace their existing fingerprint function "TRM", but because of no volunteer developers, they are now cooperating with MusicIP and use their closed-source PUID technology [12]. The FPFF is an open source description of a fingerprint for music recognition. It describes how to go from a digital signal to a fingerprint that represents the extracted features. The fingerprint is a string of symbols each representing continuous portions of the audio signal. An implementation of this, done by Heljoranta [7] is used in this thesis as the fingerprint generator. However, there are some limits of this implementation: it is not as robust against noise since it is not developed to support microphones. But people with knowledge in digital signal processing are free to replace this fingerprint generator, as the proposed solution is generic with regard to the fingerprint generator.

2.3.2 Search Algorithms

String search is a popular subject within database technology. Exact string search has for years been a well-researched area that has give rise to different index structures that speed up performance, e.g., Hash tables [3] and B-trees [14]. Also fuzzy string search has gained wide usage, since companies often gets data from multiple data sources with duplicate entries. This could, e.g., be costumer information

gathered from different sources that need to be paired. In order to do this, fuzzy string search has been developed in order to pair, e.g., "John Smith" with "Juhn Smith" [9].

Also in genome research string search is extremely popular. Genomes are often represented as long strings, and searching for genomes is done by fuzzy string search. If a scientist has some DNA with unknown origin, a fuzzy string search can be applied in order to find the best matching genome in the database. Since the DNA string does not necessarily match completely, fuzzy string search techniques for fast lookup have been developed. In the following, different search algorithms are described. Each has their own purpose to fulfill, which is the reason that none of them can be directly applied to audio fingerprint matching.

Edit Distance

The Edit Distance is the number of opertions required to transform one string into another. Several algorithms that define and measure this metric exist. One of the more famous algorithms is the Hamming Distance [15], developed by Hamming in 1950, that works on strings of same length. The number of substitutions necessary to transform one string into another, is defined as the distance between the two strings. However, this is inadequate for this solution since the query fingerprint is of different length than the database fingerprints.

Another popular and more sophisticated metric is the Levensthein Distance algorithm that was developed by Vladimir Levensthein in 1965 [19]. This distance algorithm is often used by spell checkers in order to measure how similar two words are. The distance between two strings is the number of operations needed to transform one string into the other. The operations available are insertion, deletion, and substitution. To implement a simple spell checker using only the Levenshtein Distance, every word needs to be measured against a dictionary of words. As the words get longer and the number of operation to match the strings get larger, this approach might not be the optimal. Also, when the dictionary gets larger, the number of necessary transformations increases. Comparing with the strings generated from audio signals, it is clear to see that Levensthein Distance is inappropriate, because each string is extremely long and the dictionary could consist of millions of strings.

Gravano et. al. has developed a solution that based on the Levenshtein Distance finds candidate sets for appoximate string matcing [9]. This solution, however, is not suitable for the purpose this thesis covers, since the query fingerprints and the database fingerprints are of great difference in terms of lengths.

BLAST

Basic Local Alignment Search Tool (BLAST) is a popular algorithm in genome researching for comparing genomes [16]. It looks at how many symbols a query and a database string have positioned at the same place. The more correctly positioned symbols there are in the database string, the greater score the database string gets. BLAST also supports insertions and deletions to support gaps. BLAST does not, however, support the assignment of scores to symbols that are positioned almost correctly. This is needed in order to recognize audio, since time shifting can occur. It utilizes suffix trees to accomplish fast lookup, but the performance is not as good as the performance of SSAHA, which is described below.

SSAHA

Sequence Search and Alignment by Hashing Algorithm (SSAHA) is an algorithm with the same objective as BLAST: to find similar DNA sequences [20]. It claims that it outperforms BLAST by using n-grams and hash tables. However, SSAHA has the same limitations as BLAST in terms of symbol position diviation. In order for a database string to obtain a score the n-gram has to be positioned at the correct position. Therefore, the time shifting is not supported by SSAHA. The past work by the author extended SSAHA to support the time shifting as described below.

2.3.3 Previous Work

In a project at the seventh semester at Aalborg University, the author and two others created the first steps towards an audio identification system [2]. This section describes parts of our previous research that are relevant to this thesis.

The goal was to create an audio identification system capable of recognizing music recorded from a microphone on a cell phone. The "FutureProofFingerprint-Function" [7] was used throughout the project as the fingerprint generator, and an algorithm for fingerprint matching based on SSAHA was developed. The developed algorithm had some shortcomings that is the motivation for this thesis.

Fingerprint Matching

The developed algorithm for fingerprint matching was based on the SSAHA algorithm, but, designed specifically to fit the fingerprint searching. As described in Section 2.1.2, time shifting might influence the positions of characters. Therefore, the developed fingerprint matching algorithm allowed symbols to be positioned at neighbor positions. Since the strategy used by the SSAHA algorithm had proven to be fast for approximate substring matching, SSAHA was modified to support

time shifting. How the indexing and search are performed is described in detail in Section 3.2, where the limitations of this technique are also described.

2.4 Problem Statement

Having described how an audio signal is represented and the intuition behind n-gram-based string matching, this section outlines the problem addressed in the thesis.

To be able to recognize a given audio signal, a database of digital audio files has to be available. This database is used in order to compare query audio signals against database audio signals and find the best matches. Obviously, the database has to be of great size in order to recognize all possible queried audio signals. In the case of music recognition, the database requires the music from all companies in the music industry. With the database in place, each file should be converted to a fingerprint and indexed for fast lookup.

When receiving the query audio signal, it should be converted to a fingerprint in the same manner as the database audio signals and then looked up in the database. Since the audio signal can be subject to noise, two otherwise identical audio signals are most likely to have different fingerprints. This means that approximate substring search is needed; an exact string search is likely to return no matches. Also, the query audio signal can start at arbitrary positions in the complete signal, which rules out a complete approximate string search in the database.

Instead a fast, approximate substring search should be applied in order to find a match in the database. If a query audio signal q is identical to d in the database, but is missing the beginning of the signal, q is a sub signal of d. Having converted the audio signals to fingerprints results in the problem of finding the best approximate substring. This problem is partially solved by previous work by the author. However, the existing solution only works on small datasets. This thesis will focus on creating a fast, generic backend for searching datasets of great size. This is under the assumption that suitable features have been extracted and used to generate the fingerprints. The backend should find the best approximate substring in the database.

The following will often refer to database results as "best match" and "next best match". This is with regard to the fingerprint string and not the audio signal itself. When the best match is found, this is the fingerprint in the database that best matches the queried fingerprint. The returned meta data will therefore also be with regard to the best fingerprint match. If the fingerprint generator is malfunctioning, the expected best audio match may not be consistent with the best fingerprint match. This thesis focuses on the fingerprint string matching and not the audio fingerprint generator.

Chapter 3

Scalable Fingerprint Matching

Having covered the preliminaries and related work, the following introduces a new strategy, and details behind creating a backend for scalable fingerprint matching. First the architecture of the complete audio identification system is outlined in Section 3.1. Then Section 3.2 describes in details how the previous developed search algorithm operates in order to point out where the problems are in Section 3.3. Finally, the proposed solution is described and other optimizations are discussed.

3.1 Architecture

To create a complete solution for fingerprint matching a number of collaborating components with different purposes is necessary. First of all, the end user has to be able to use the application easily through different mediums. This could, e.g. be a mobile phone, a desktop computer or a phone line. Picture 3.1 shows a setup where the client application has been developed for a mobile phone. The client application contains the fingerprint generator, a client/server implementation over the Internet and a GUI in order to display the results from the backend server. When an audio signal is recorded, the fingerprint is generated and continuously streamed to the backend server. The streaming continues until enough data has been recorded in order to find a match from the fingerprint database. Finally, the backend server returns the relevant metadata to the client application. This could be the song title, artist name and links to where to buy the music. The client application could be on any device with audio recording equipment, such as desktop computers, laptops, portable audio players etc. A setup where the end user does not use a client application directly is also possible. This could be through a phone line that is connected to the client application. Thus, the user only has to dial a curtain number and then communicate with client application that could provide the meta data with speech through the phone line.

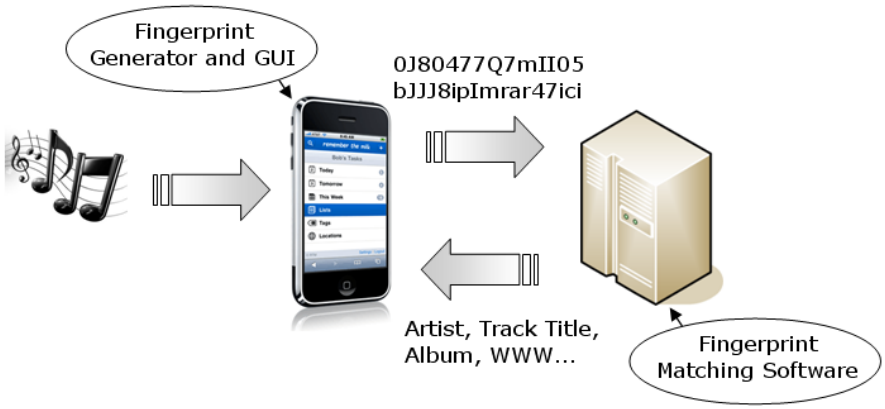

Figure 3.1: The architecture of a complete audio identification system.

The emphasis in this thesis will be on the backend. In order to create a complete audio identification system for end users, a client on any given medium should be developed and interplay with the backend.

3.2 Fingerprint Matching using N-Grams and Hash Tables

This section describes the fingerprint search algorithm developed by Skovsgaard et al.[2] to provide an overview of how the best match is determined. The matching strategy is also used in this thesis. First, the indexing is described, followed by how the actual fingerprint search is performed.

3.2.1 Indexing the Database Fingerprints

To be able to query the database fingerprints an index initially has to be created. The index is build from all fingerprints in the database. The structure is conceptually simple and uses a hash table for fast lookup. An example index from two fingerprints have been created in Figure 3.2. When indexing the fingerprints every n-gram for every position is extracted and placed in the hash table with the n-gram itself as key. The n-gram extraction has to be for every position and not a step length of the n-gram size, since this is more robust against errors in the fingerprints. The n-gram extraction is illustrated in the left table in Figure 3.2. The value of the hash table is a list of objects, each containing the song identifier and the position of the n-gram. If the n-gram does not exist in the hash table a new entry is created and the object is added as value. If, however, the n-gram already exists, the object is added to the already existing list. This is illustrated

by the list of n-gram occurrences in the right side of Figure 3.2. The list contains tuples on the form $S \times P$ where S is the song identifiers and P is the positions.

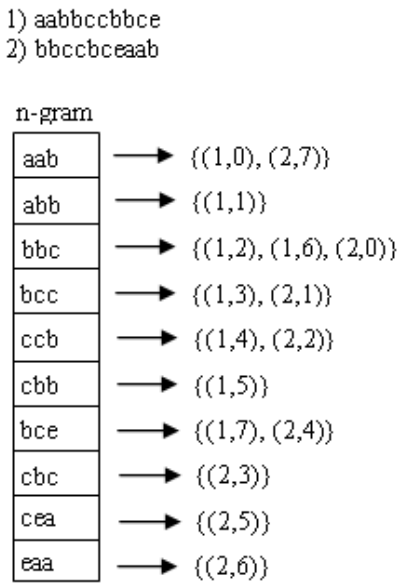

Figure 3.2: The index.

3.2.2 Querying the Database

Having build the n-gram index, the database is ready to be searched with a query fingerprint. An algorithm for this has been developed in past work by Skovsgaard et al. [2]. When querying the database with a fingerprint, several database fingerprints can have similarities with the query fingerprint. In order to distinguish these candidates a scoring system has been created. The database fingerprint that obtains the highest score is the best match to a given query fingerprint. One point is given to a fingerprint for every n-gram correctly positioned in the fingerprint. This principle is based on the research made by SSAHA [20], but extended by Skovsgaard et al. to support the time shifting problem described in 2.1.2, such that a character may deviate by one position.

The fingerprint search algorithm is illustrated in Figure 3.3 and starts by extracting each n-gram from the left side of the query fingerprint. Each n-gram is looked up in the index in Figure 3.2 to find all occurrences of the n-gram in the database fingerprints. This list contains objects with song identifier and the position of the n-gram in the database fingerprint. The position is used to create a "position group", *pgroup*. The *pgroup* is calculated by taking the position in the index object and subtracting the position of the queried n-gram. In the example

in Figure 3.3 first the "aab" n-gram is looked up, giving two occurences. The first is positioned in the same position as the queried n-gram, giving *pgroup* 0. The next occurrence is positioned at position 7 giving the *pgroup* 7. These *pgroup* are created to keep track of which n-grams that are linked together in terms of correct positions. If two n-grams is within the same *pgroup* they are mutually positioned correctly with regard to the query fingerprint.

When a new *pgroup* is created it is added as key to a hash table H_{position} with a new hash table H_{songnr} as value. The hash table H_{songnr} maintains all song identifiers that is already known to be within the given *pgroup*. The scoring system is maintained while searching, and when a new *pgroup* is created the song identifier is, if not existing, added to a new H_{songnr}. The value of H_{songnr} is a list of objects containing the score of the given *pgroup*. This list can contain more than one entry because of the support for character deviation. A new element is added to this list whenever a *pgroup* does not exists for a given song identifier. Then the score is set to 1. If the *pgroup* already exists, the score is incremented by 1. Also if *pgroup*-+1 exists this group is incremented by one. This is to support the time shifting problem by also giving one point to the n-grams that deviate by one position. It is also illustrated in Figure 3.3 by the n-gram "bce" that correctly should be in *pgroup* 1 but since *pgroup* 0 exists this is incremented by one. In Figure 3.3 the query fingerprint has found eight matching occurrences in the index of which five *pgroup* have been created. The fingerprint with the most points is fingerprint "1" with three points.

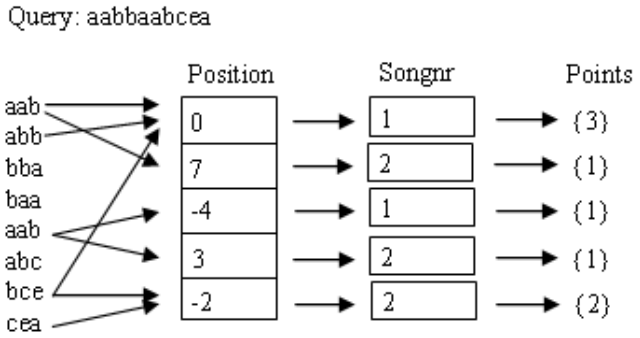

Figure 3.3: Fingerprint matching

When the highest score is a predefined constant higher than the next best the search can be terminated and the fingerprint with the highest score is returned as best match. If this does not occur within a predefined period of time, the search is terminated and no match is found.

3.3 Limits of the Search Algorithm

Although the above outlined fingerprint search algorithm is an efficient search method it has some shortcomings. It is fast on a small dataset such as in Figure 3.2, since the lists in the index only contains few elements. This gives a small number of iterations when searching. But as the dataset gets larger, so does the number of elements in the lists, and thereby the performance of a query will decrease since more objects have to be evaluated. The evaluation of an object is the process of lookups in the index and in the two runtime hash tables or create new keys if they do not exists, and maintain the scores. This becomes the bottleneck of the algorithm as the index increases, and limits the matching algorithm significantly in terms of scalability.

To improve on the search speed, the lists in the hash table have to be reduced. The past work iterated over every object in the list within a given n-gram. This was to ensure that no better match exists somewhere in the database. Since the lists of objects in the index determines for how long the search should continue, the size of these lists has the greatest impact on the performance. Whenever a fingerprint is added to the index numerous objects are created within several n-gram lists. Therefore, the search algorithm has to evaluate even more objects. Thus, the number of necessary evaluations increases linearly with the number of fingerprints. This is not scalable since the number of fingerprints could be from thousands to millions. Therefore, a reduction of the lists will improve the search speed significantly.

One method of solving this scalability problem could be by have a prioritized index that contains only a small subset of the dataset. This subset could for instance be the 100 most played songs in the radio or the most requested songs in system. If the requested song is given a convenient score when searching the subset, it could be roughly estimated that this must be the correct match. If it does not, however, get a high enough score the next 100 most played songs could be searches and so on. This would reduce the search time dramatically. However, some problems emerge with this solution. How high should the convenient score for a correct match be? And when should the search be aborted, if the requested song does not match any song from the database? Also, it is impossible to guarantee that the best match in the prioritized list is the best from the full dataset, since a song with higher score could exists somewhere else in the database. Thus, it is clear that this solution is not desirable if the requested songs are not in the prioritized lists.

3.4 Filtering Trivial Fingerprints

Since the greatest improvements of the search speed can be achieved by reducing the lists in the index, this section proposes another solution without the above mentioned problems. We propose a solution that filters out trivial fingerprints. Trivial fingerprints refer to fingerprints that have less possibility of obtaining a high score. Having the fingerprints A: "aabbccddee" and B: "qqwwwmmmde" and the query fingerprint Q: "aabbccdee" it is clear to see that B and Q have no other similarities than the "de" characters. But A and Q are almost identical. Thus, A should be chosen as the best match. If two fingerprint have minimal similarities they are trivial fingerprints for the given search, since other fingerprints with more similarities most likely can get higher scores. Therefore, it is not necessary to use computational time to iterate over these trivial fingerprints. Instead, time should be used on the most similar fingerprints.

To measure similarity, the number of common n-grams between two fingerprints have to be calculated. A fingerprint f has $g = s - (n - 1)$ n-grams, where s is the number of characters in f and n is the n-gram length. As the number of common n-grams in two fingerprints approaches the smallest of the two g, they are more likely to be similar since they share more n-grams. Using the above fingerprints A, B and Q as example and a n-gram length of 2, the smallest g is 9-(2-1) = 8 n-grams and can be found in fingerprint Q. A and Q has n-gram "aa", "ab", "bb", "bc", "cc", "cd", "de" and "ee" in common. That is exactly the smallest g and therefore their similarities are expected to be high. On the other hand is B and Q. They only share the n-gram "de" and the expected similarity is therefore very low. Estimating the best matching fingerprint only by looking at the common n-grams gives good conditions for fast lookup in databases. The database could be designed as depicted in the ER-diagram [1] in Figure 3.4 using the Crow's Foot notation. Thus, issuing a SQL query as in Listing 3.1 gives the desired output.

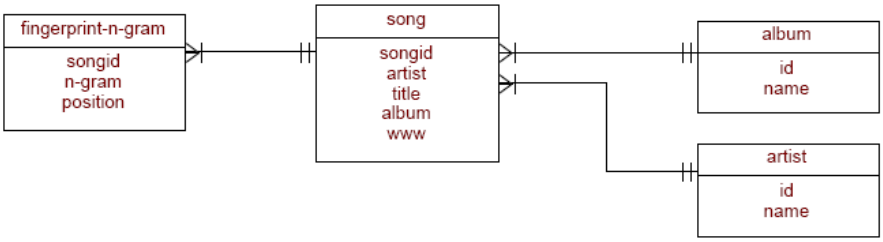

Figure 3.4: Entity-relationship diagram for the proposed solution.

```
1 SELECT   songid, count(n-gram) AS ng
2 FROM     fingerprint-n-gram
3 WHERE    n-gram in ('aa', 'ab', 'bb', 'bc')
```

```
4 GROUP BY songid
5 ORDER BY ng DESC
```

Listing 3.1: Query the retrieves the most common n-grams.

There are, however, some exceptions that make this solution unpractical. To prove this by example a new fingerprint C: "deedqqbccdqqabbaa" is introduced. Having the query fingerprint Q, the common n-gram between C and Q are "aa", "ab", "bb", "bc", "cc", "cd", "de" and "ee" where g is still 8 n-grams. This means that C and Q share exactly the same n-grams as A and Q, thus having the same high expected similarity. However, it is clear to see that C and Q have no similarities whatsoever, and is very unlikely to become a best match. Therefore, the similarity measure outlined can not be used solely as a solution for finding the best match. It can, however, be utilized in order to reduce to number of objects that have to be evaluated as the next section describes.

3.5 Push rather than Brute Force

With the above conclusion that similarity measure based on common n-grams can not be a sole solution, this section proposes a new method using the strategy from above. The objective is still to reduce the large lists in the index. Thus, some objects in the lists must be removed giving a remaining candidate set of objects. The objects that should be filtered out are those that belong to fingerprints that are estimated as trivial, and therefore not expected to make a good match. The resulting candidate set should consist of those fingerprints that have the highest probability of being the best match. Thus, filtering out those fingerprints which are extremely time-consuming to evaluate. The above mentioned similarity measure gives a good candidate set, since similar fingerprints will naturally share many common n-grams. However, even though they have a high similarity, they can have a low number of correctly positioned n-gram as the above example showed.

It can occur that the fingerprint with the most common n-grams is not the best match due to incorrectly positioned n-grams. Therefore, the returned fingerprints need some post processing to calculate the amount of correctly positioned n-gram. This is done by modifying the algorithm proposed by Skovsgaard et al. [2] in such way that the candidate objects are pushed to the algorithm, in order to calculate the score, rather than brute forcing every fingerprint in the index. As mentioned above the existing search algorithm terminates once a predefined distance between the two best matches is reached. This strategy can not be directly used in the modified algorithm since selected fingerprints are pushed to the algorithm. This means that the algorithm can not keep track of the current score in every fingerprint, as with the past solution. To be able know when to terminate and to

guarantee that no other match can exists in the database, the following strategy can be applied.

Let a query fingerprint have the highest possible number of n-grams, σ, in common with α database fingerprints. Each of the α fingerprints are then evaluated with the modified algorithm in order to determine how many of the n-grams that are positioned correctly. The best score, ϵ, is given to fingerprint ϕ. If ϵ is not equal to σ it is possible, that other fingerprint with lower common n-grams than σ contains more or the same amount of n-grams as fingerprint ϕ. Therefore, all fingerprints with ϵ common n-grams have to be pushed to the modified algoritm in other to find the best score.

If the best score in this search is given to fingerprint δ and it is below ϵ it is known that no other fingerprint with a higher score than ϕ exists in the database. This is due to the fact that a fingerprint cannot get a score that is higher than the number of common n-grams. And since all fingerprints with ϵ n-grams have been evaluated, no fingerprint can exists with more correctly positioned n-grams than fingerprint ϕ.

If fingerprint δ had a higher score than ϵ the best match in the database would be δ.

3.5.1 Early Termination

To improve the search time by not evaluting all candidates and to guarentee that the search stops once a reliable match has been found, the following describes methods for early termination.

WinningFactor

If two or more fingerprints have the same highest score it is not possible to determine which the best match is. The search can not be terminated with more than one matching fingerprint, since the end user expects only one match. Therefore, a longer query fingerprint should be used for searching. By having more n-grams in the query fingerprint, it is most likely that the distance between the two best matching fingerprints increases. Then the overall search can be terminated when the distance between the best and the next best match is over a predefined percentage called the *WinningFactor*. If the *WinningFactor* is set to 10% then the search will return the result once the next best match has a score that is more than 10% lower than the best match. As the *WinningFactor* increases, the higher the distance between the matches will be before returning. This is shown in the pseudo code in Listing 3.2 at line 29 and 31.

MaxTime

If the *WinningFactor* is not reached within a predefined number of seconds of audio data, called *MaxTime*, one of two actions can be taken. The first possibility is to abort the search and return no match. The second action could be returning two or more possible best matches, if they all have the same or almost same score. This could be depending on the choice of the end user. The user could prefer an ambiguous result rather than no result.

This early termination factor is shown in the pseudo code in Listing 3.2 at line 17. If the search has not been terminated and the *round* variable has incremented over the TQL threshold the search is terminated with no match found.

MinMatch

False positives results can occur in the case where the database does not contain the queried fingerprint, or the queried fingerprint is extremely subject to noise and other disturbing sources. If only a small number of the n-grams are positioned correctly, and there are many more n-grams in the query, it is likely that the found fingerprint is a false positive. This is due to the fact that fingerprints with a small number of correctly positioned n-grams are less similar. To help avoiding false positives and also preventing iterating through fingerprints of no interest, a similarity threshold, *MinMatch*, between the query fingerprint and best matching fingerprint are to be selected. This threshold should be based on the formula: (common n-grams / number of n-grams in the query) * 100. That means that database fingerprints not containing the predefined percentage of n-grams from the query fingerprint will be discarded. If a given search can not exceed *MinMatch* the iteration should be terminated, since it is not possible to find a reliable result. This is used in the pseudo code in Listing 3.2 at line 19 where the current iteration is aborted and more query data is searched.

As *MinMatch* approaches 100 then false positives is less likely to occur since it forces more n-gram to be correctly positioned. However, depending of the amount of noise in the signal it is possible that a correct match would be suppressed in case of a high *MinMatch* and no match would be returned. Choosing a too low *MinMatch* would increase the probability of false positives. Thus, *MinMatch* should be carefully selected with respect to the robustness of the fingerprint algorithm.

3.5.2 Non-Unique N-gram Search

If a query fingerprint contains, e.g., 16 n-grams, but only 3 unique n-grams, the highest number of unique common n-grams is 3. This may cause some problems

in the search when assuming 3 is the highest possible score. But the 16 n-grams could be correctly positioned giving a score of 16. Therefore, when building the index all equal n-grams should be suffixed with an unique identifier. This could be the number of occurrences of the given n-gram. Thus, giving records with n-gram values: "gram", "gram2", "gram3". When performing the search the queried n-grams should also be suffixed with the identifier for all equal n-grams. This simple solution solves the problem of the highest score assumption being only the unique common n-grams.

3.5.3 Supporting Streaming Fingerprints

The backend should support streaming fingerprints and be able to terminate the search whenever enough data has been received to guarentee a best match. Therefore, some adjustments of the above mentioned strategy must be made. As the fingerprint is streamed to the backend in blocks of audio, the block size should be chosen. The amount of data in each block could be from a few n-grams and up to the $MaxTime$ value. The amount could be chosen to be the average amount of data necessary to identify an audio signal. This will, however, give longer than necessary search time for some client users. Therefore, data blocks of one seconds could also be chosen. Since only the block gets processed at each query and not the complete signal the overhead of small blocks is minimal.

Having a block size of one seconds and dividing the signal into blocks of 62.5ms gives 16 n-grams per block. The first block of n-grams should be queried to the database. If the best match has ϵ n-gram correctly positioned the search should continue until all fingerprints with $\epsilon - (\epsilon * (WinningFactor * 0.01))$ common n-grams has been evaluated. Notice that the best match, ϵ, is constantly maintained as shown in Listing 3.2 at line 29. It is necessary to evaluated this number of fingerprint, since the *WinningFactor* value is the termination factor for when the best match is considered correct and the backend guarantees that no other fingerprint in the database can be a better match. If the next best match has less than $\epsilon - (\epsilon * (WinningFactor * 0.01))$ correctly positioned n-grams the search could be terminated and the best match returned. However, if this is not the case, more of the streamed fingerprint should be searched. Again 16 n-grams is searched and all fingerprints down to the *WinningFactor* threshold is evaluated. Now the score is added to the previous result in order to give a score for the total 2 seconds of streamed fingerprint. If the distance between the two best matches is over the *WinningFactor* threshold the search is terminated or else it continues in the same manner.

Before each candidate is pushed to the algorithm the $MinMatch$ parameter is used as a termination threshold. If the number of common n-grams in a fingerprint is below $MinMatch$, the remaining candidates will be considered too different and

will be discarded.

3.5.4 Pseudo Code for the Proposed Optimization

Having described the strategy and arguments for the solution above, this section gives an exact overview of the searching algorithm in form of pseudo codes for the database searching in Listing 3.2 and the score calculation in Listing 3.3.

```
1  Variables:
2  Q - query song.
3  L - temporary list of grams.
4  J - number of n-grams to process
5  H - sorted highscore list.
6  W - WinningFactor
7  TFP - MinMatch
8  TQL - MaxTime
9  BEGIN
10 DATABASE-SEARCH(L, round)
11    if L is empty
12       L ← first J n-grams in Q
13       round ← 0
14    do
15       tmpsong ← extract next fingerprint from database where the results is
              sorted by most common n-grams from L.
16       tmpcommon ← number of n-grams that tmpsong has in common with J.
17       if round*(J/16) > TQL then // the search has taken too long
18          terminate search and return no best match
19       if (tmpcommon / J) * 100) < TFP // it is not possible to satisfy TFP
              with this part of the signal - get more
20          break
21       scores ← EVALSCORE(tmpsong, round, J)
22       totalscore ← scores[0] // evaluation with modified matching algorithm
              - the total score
23       score ← scores[0]-scores[1] // evaluation with modified matching
              algorithm - the score of the iteration
24       if tmpsong exists in H then
25          tmpsong.score ← totalscore
26       else
27          add tmpsong to H with tmpsong.score ← totalscore
28    while
29       tmpcommon > H[0] - (H[0]*(W*0.01)) // search until W is guarenteed
30
31    if H[0]-(H[0]*(W*0.01)) > H[1]
32       terminate search and return tmpsong as best match
```

```
33
34    L ← next J n-grams in Q
35    round ← round + 1
36    DATABASE-SEARCH(L, round)
37 END
```

Listing 3.2: Pseudo code for the database search.

```
1  Variables:
2  Q - query song.
3  N - size of gram.
4  S - song table that has song id as key and pgroup with corresponding
       highscore as value.
5  Output:
6  Song with most points.
7  BEGIN
8  EVALSCORE(Q, round, J)
9     songlist ← S[Q] // get if it exists to correctly use previous pgroup and
          score.
10    querypos ← 0
11    while querypos < characters in Q-N-1
12       querysubstring ← characters from position querypos to querypos+N
13       for each gram g in database fingerprint with id Q, that is equal to
             querysubstring
14          posgroup ← g.position - querypos + (round*J)
15          gsonglist ← all entires from songlist positiongroup equal to
                posgroup
16          if gsonglist is not empty
17             foreach songobject s in gsonglist
18                if querypos != s.queryround // do not give more than one
                      point per query n-gram.
19                   give s one more point
20          else
21             tcounter ← -1
22             while tcounter <= 1 // For character diviation
23                add posgroup+tcounter as key to songlist and a new songobject
                       as value
24                tcounter ← tcounter + 1
25       querypos ← querypos + 1
26    return score of best songobject, score of best songobject before this
          iteration
27 END
```

Listing 3.3: Pseudo code for the match algorithm.

3.6 Other Optimizations

This section describes other possible optimizations that can be made in order to improve the search time. However, the use of these optimization are not generic as the above described algorithm. These optimizations are depending on the specific fingerprint generator used.

3.6.1 Table Division

As of now, all fingerprints are within the same table. This table could contain millions of fingerprints and the search time is naturally depending on the number of fingerprints. If the amount of fingerprints in the table could be minimized, the search time would be improved. In some audio signals it is possible to extract curtain features, such that a classification of the signal can be estimated with only a portion of the signal. This classification could be used to divide the large main table into smaller tables. Thus, the classification of a fingerprint determines, in which table it should be located. When the query signal is recorded the features to determine the classification are extracted while generating the fingerprint. This classification should then be used for fast lookup in the specific table containing the fingerprints within the given classification. The performance gain is naturally depending on the number of groups a classification can have and the distribution of the fingerprints. Also, the classification has to be robust on all queries in order to work. To adjust the algorithm in Listing 3.2, in order to comply with this optimization, is simple. An extra variable, C - classification of query, should be added. And line 15 should be corrected to " extract next fingerprint from database with classification C where the results is sorted by most common n-grams from L." If the robustness of the classification can not be completely guaranteed then all tables must be searched in order to comply with the no false dismissal guarantee.

Genre Based

Research made by P. Ahrendt et al. [13] shows a technique that estimates the genre with only a small portion of a song. There are five genres "Classical", "Heavy", "Jazz", "Popular" and "Techno". These genres can be determined at all locations in a song. However, since some songs can have pieces that contains different genres more sample data could be needed in order to fully determine a songs genre. Since it is possible to distinguish the five genres the main fingerprint table could be divided in five smaller table - one for each genre. When querying the database, the genre is extracted and the corresponding table searched. If the database songs are evenly distributed across the genres then each table will be reduced with a factor five. This will improve the search time even further.

3.7 Performance Factors

This section describes aspects that affect to performance of the proposed solution and also compares with previous work. Running time is covered in Section 3.7.1, and space consumption is covered in Section 3.7.2.

3.7.1 Running Time

The search time is depending on several factors that influence the search algorithm. First of all are the different parameters to the algorithm which are the n-gram size, the *WinningFactor*, the $MaxTime$ and the $MinMatch$. All these have great influence on the performance of the algorithm. But also the fingerprint generator and the dataset influence the search time. Having a fingerprint generator that is not capable of making a good distinction fingerprints in-between results in many fingerprints that will be candidates for the best match. The result is a slower search time since more fingerprints has to be evaluated. However having a robust fingerprint generator that can distinguish the fingerprints in-between improves performance.

Worst Case

Here we consider the worst-case scenario. But as this only has relevance when the parameters are set to values that will make no sense in a practical setup, it is not likely to happen. However, it is described to outline the limitations of the algorithm and intended parameter values. The worst case setup occurs when the *WinningFactor* is set to 100, the $MinMatch$ is set to zero, and the $MaxTime$ is set to infinity. Having the *WinningFactor* set to 100 will first of all cause the algorithm to evaluate all fingerprints until $MaxTime$ is reached. Since the $MaxTime$ is infinity, the search will not be aborted at any time. With the $MinMatch$ set to zero, all entries with only one matching n-gram will be evaluated. This is clearly an undesirable situation, since for every block of the query, all database fingerprints with only one matching n-gram have to be evaluated. Thus, the search time will be slower than what is achieved in previous work, because the same amount of fingerprints will be evaluated in addition to the overhead of the database similarity lookup. An important factor is also the n-gram size. Having a small n-gram size will increase the possibility of common n-grams, since the number of unique n-gram decreases. Thus, the worst case n-gram size is 1.

Another worst case scenario is related to the dataset. Assume every fingerprint in the database contains the exact same n-grams, just at different positions. Also assume, that the query fingerprint contains the same n-grams. This would cause every fingerprint to have the same high similarity. Depending on the positioning

of the n-grams, a full table search could me necessary. However, this setup is very unlikely to occur, since all the audio signals would have to be extremely similar.

Typical Case

Having described the worst-case setup, we cover the intended setup where the parameters are set to values that fit the dataset. The *WinningFactor* has great influence on the reliability of the results. Setting it too low may yield incorrect results, and the impact of setting it too high was outlined above. The value could be set to 10, thus limiting the number substantially from the full dataset. The selection of an appropriate value for $MaxTime$ depends again on the use. For music recognition, a good setting could be about 10-15 seconds of audio. This means that the search is sure to terminate.

Since the choice of a good value for $MaxTime$ depends on the robustness of the fingerprint generator, the value to use can vary. But the fingerprint generator can be expected to be robust such that a least 80% of the n-grams are positioned correctly. This also filters out all fingerprints with less common n-grams than $MinMatch$.

As mentioned above, the size of the n-grams is also an important factor. With large n-grams it is less likely that fingerprints have n-grams in common, since many unique n-grams exist. Previous work [2] has shown that a n-gram size of three is appropriate for the tested fingerprint generator. But with a more robust fingerprint generator, a higher n-gram size can be chosen, since errors in the fingerprints are less likely to occur.

3.7.2 Space Consumption

Since each fingerprint is divided into n-grams of fixed length and has associated some meta data, the space consumption of the solution depends on the number of fingerprints, the lengths of these, and the n-gram length. The number of n-grams in a fingerprint is determined by the length of the n-gram, n, and the length of the fingerprint, s: $s-(n-1)$. Thus, the number of symbols a fingerprint occupies is: $n \times (s-(n-1))$. This gives a worst case space consumption when the n-gram length is $n = s/2$. The best case space consumption occurs when $n = 1$ or $n = s$ resulting in s symbols. However, when $n = 1$, the number of unique n-grams is low, causing the search to be slower since more fingerprints can have n-grams in common.

Having a collection of i fingerprints of length s with best-case n-gram length thus gives a n-gram space consumption of $i \times s$. The worst case space consumption would be $i \times (s/2 + 0.5)^2$.

3.7.3 Memory Usage

As shown in Listing 3.3 the calculation of a fingerprint's score requires some memory while post-processing the result from the database. However, the amount of memory needed is determined by the number of candidates and the number of corretly positioned n-grams. As the size of the candidate set increases, so does the amount of necessary memory. If many of the candidate fingerprints are poor candidates, the amount of *pgroups* will increase and cause a higher memory consumption. However, compared to the previous work, the memory consumption is orders of magnitude less, since the candidate set is a subset of the dataset. The previous work calculated the score of every fingerprint in the dataset with the n-grams from the query. The proposed solution reduces this number dramatically, thus leading to lower memory usage during post processing.

The database server that finds candidate fingerprints will use more memory with the proposed solution, as it has to perform the *GROUPBY* operation. The previous work used a lookup at each n-gram, causing many more database requests, but each of lower memory consumtion than the proposed solution. However, the proposed solution only makes one database request in order to find the candidate set.

Chapter 4

Experimental Performance Study

4.1 Test Setup

This section will describe the test setup for the following experimental performance studies. All test results were created using a database server with an Intel Celeron 2.66GHz CPU with 1GB of physical RAM. The harddrive was a Seagate Barracuda 160GB 7200RPM. The operating system installed was Microsoft Windows 2003 Enterprise Server, and the database server was Microsoft SQL Server 2005.

The computer querying the database and doing post-processing was an Intel Pentium 4 3.20GHz CPU with 1GB of physical RAM. The harddrive used was a Western Digital Raptor 36GB 10.000RPM with Microsoft Windows XP Professional SP3 as operating system.

All post-processing algorithms that queried the database server were implemented in C#.

To test the performance of the algorithms, a large database with fingerprints was necessary. A dataset based on 1,000 real songs was created using the FPFF fingerprint generator. But in order to test the algorithms on a larger dataset, hundreds of thousands of complete music files are needed. The fact that this number of music files is hard to find and the tremendous task of transforming them into fingerprints, led to the choice of another solution. Instead, the 1,000 music files were analyzed in order to see what was characteristic about the fingerprints. These characteristics were then used to create a large dataset containing randomly generated fingerprints. The fingerprints were created from the BASE64 charset with a length that could vary from three to five minutes of audio. It was discovered that the same symbols often occur up to three times next to each other; this observation was also used to build the random data. The result is fingerprints where some have a high similarity and some have less similarity. This seems to be a good simulation of a fingerprint generator that is capable of distinguishing

among different audio signals.

The largest dataset generated contains 250,000 fingerprints with 547,912 unique n-grams, which are located at 852,818,455 different positions within the fingerprints. This gives an average fingerprint length of 3,411 n-grams, which corresponds to about $3\frac{1}{2}$ minutes of audio.

4.2 Choice of Parameter Settings

The proposed algorithm has some different parameters that can be adjusted to fit the architectural context. The following sections will outline the performance results and discuss what the different value entails.

4.2.1 N-gram Length

The n-gram length has influence on the space consumptions as described in Section 3.7.2 and the performance of the algorithm. As the n-gram length becomes larger more unique n-grams will emerge, thus leading to fewer common n-grams when querying. This has great influence of the performance since the database lookups will become faster and the candidate set smaller. The side effect is, however, that it requires an extremely robust fingerprint generator that is unlikely to generate errors in the query fingerprint compared to the database fingerprint. If the n-gram length is of great length, one error in a block of the fingerprint would lead to no matching n-grams for a large part of the fingerprint. As the n-gram length is reduced the performence decreases, since less unique n-grams can emerge. Therefore, the n-gram length should be carefully selected. In the past work, a performance study of the optimal n-gram length, with a fingerprint generator that is not very robust was found to be three [2]. Since this fingerprint generator is also used in these performance studies, three will be the value for the n-gram length. It is important to note that the n-gram length should be chosen to fit the used fingerprint generator as the performance very likely could be improved.

4.2.2 The *MinMatch* Parameter

This test studies what happens when the search is performed with different values for the *MinMatch* parameter. Five fingerprints have been randomly selected from the database and n-grams corresponding to 8 seconds of audio have been extracted. The input was streamed to the algorithm with one second blocks resulting in 16 n-grams per second. The number of unique database fingerprints found as candidates was summed for the complete query and an average of the five fingerprints was

taken. Each fingerprint was searched with a varying value of the *MinMatch* parameter as show in figure 4.1.

When the *MinMatch* value is low, a large number of fingerprints are excepted to be good candidates, since many n-gram contains smiliar n-grams over the small threshold. As the *MinMatch* increases the candidate set decreases dramatically. When the value reaches 60 it is less than 100 and with the value 80 there is only one candidate left; that is the best match. Therefore, the *WinningFactor* will never be used when the *MinMatch* is above 80 with fingerprints with no noise in this dataset. This would of cause be different if the dataset contained more similar fingerprints. If the fingerprints was subject to great amounts of noise the *MinMatch* factor must be carefully selected since a too high value could result in no matches. A value of 80 can be used with the fingerprint generator used in this thesis, however, a higher value could be selected with more robust fingerprint generators.

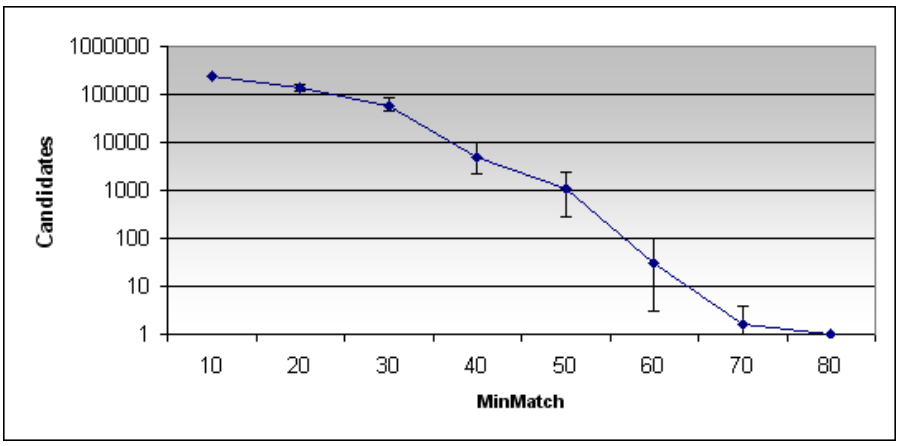

Figure 4.1: Worst case scenario for previous work

4.2.3 The *WinningFactor* Parameter

The *WinningFactor* parameter determines when a fingerprint is expected to be the best match. The higher value the more difference between the best and the next best match there is, thus making the result more reliable. But as the *WinningFactor* increases more fingerprints have to be examined since they may be within the *WinningFactor* threshold. With a small *WinningFactor* value less fingerprints have to be evaluated since only the fingerprints with high number of similar n-grams are candidates. However, a very low *WinningFactor* value can cause wrong results, since at a given time a wrong fingerprint can be estimated as the best match with a small distance to the next best match.

The level of reliabitity has its costs as figure 4.2 shows. The graph was created by taking five random fingerprints and extracting n-grams corresponding to 8 seconds of audio. The n-grams was streamed to the algorithm with one second intervals. The noise was made by replacing symbols corresponding to the percentage spread equaly in the fingerprint. The *MinMatch* was set to 0 since a higher value could become the dominant threshold and giving useless results.

With a fingerprint with no noise the number of necessary evaluated fingerprints are within a few fingerprints while the *WinningFactor* is below 40 percentage point. This is because only a few database fingerprints have the necessary number of fingerprints in common with the query fingerprint. Notice that when there is no noise in the signal and *WinningFactor* uses the same limit as *MinMatch*, the candidate set or number of evaluated fingerprints are the same. As the *WinningFactor* is set to 50, about 1.000 fingerprint have to be evaluated because more can be within the threshold for the next best match.

A fingerprint with noise will naturally not get as good a score as one without noise. Therefore, more fingerprint in the dataset have to be searched, since the threshold for the next best match depends on the best found score. With 10% of noise in the query fingerprint the number of evaluated fingerprints naturally increases as shown in figure 4.2. For both 10% and 20% of noise the number of evaluated fingerprints are within 1.000 when the *WinningFactor* is below 30. The past work evaluated at all times n-grams from all 250.000 fingerprints since n-grams from the test queries is present in every fingerprint in the dataset. Thus, the candidate set is clearly orders of magnitude smaller than the dataset.

When noisy fingerprints are searches, the *MinMatch* parameter will often become most dominant, because the *WinningFactor* threshold will evaluate fingerprints with less common n-grams than the *MinMatch* parameter allows.

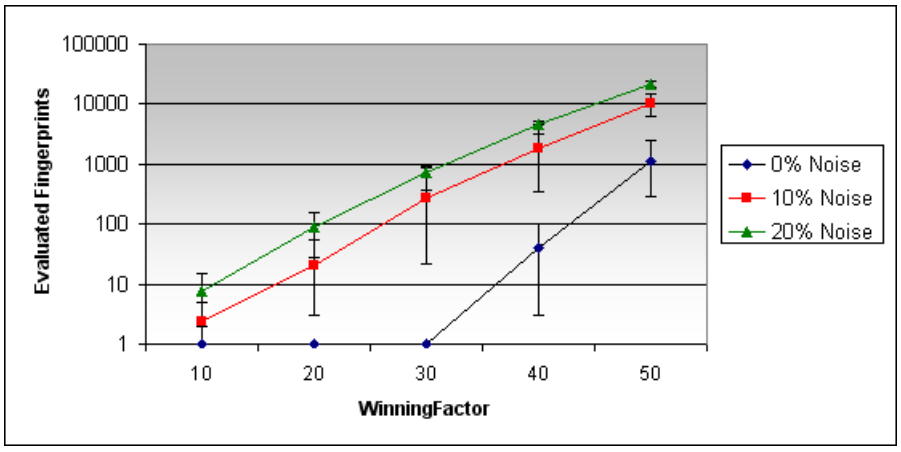

Figure 4.2: The *WinningFactor* with different number of errors.

As mentioned above a very low *WinningFactor* may result in a false positive result. However, at no time in the test was a "false" best match over the *WinningFactor* threshold. This is also logical because if a fingerprint has many n-grams in common with a "false" match it must also have many fingerprints in common with the "true" match. However, if the fingerprint is extremely noisy it could be possible that the *WinningFactor* will be reached for a "false" match. But then the fingerprint generator is not robust enough and the *WinningFactor* should be set to a higher value for this context. A *WinningFactor* of 10 will therefore be suitable for this setup.

4.3 Query Performance

This section contains tests that shows the performance of the overall system when querying. The tests are performed on different datasets in order to show the positives and negatives aspects of the proposed solution.

4.3.1 Comparison with Previous Work

This test was made in order to show the difference between the previous work and the proposed solution on an average case. It was performed on a dataset of varying size to show how the algorithm performs on small and large collections of fingerprints. The query fingerprints contained n-grams corresponding to 8 seconds of audio and the n-gram length was set to 3. The datasets was created with the above described random fingerprint generator. Five random fingerprint was selected from the database that was present in all ten datasets. This was to simulate a query song that exists in the database. To force both algorithms to query the complete query, the *WinningFactor* was set to the maximum value 100. In the proposed algorithm the *MinMatch* was set to 80% thus allowing 20% of errors in the candidate set.

Figure 4.3 shows the result of the performance test. Both algorithms has found the same best match with the same score within ten seconds when the dataset contains 25.000 fingerprints. But as the dataset gets larger it is clear that the previous work does not scale. The proposed algorithm, however, is below five seconds at all times in the test and up to 20 times faster than the previous work. Thus, it is clear that the proposed solution has the intended results.

The performance of both algorithms is effected by the number of similar n-grams in the fingerprints. If numerous fingerprints in the database contains many of the same n-grams and the query also contains all these n-gram the performance of both algorithms will become slower. The past work has then problem of large lists in the index that it has to iterate through. The proposed algorithm has to do more work

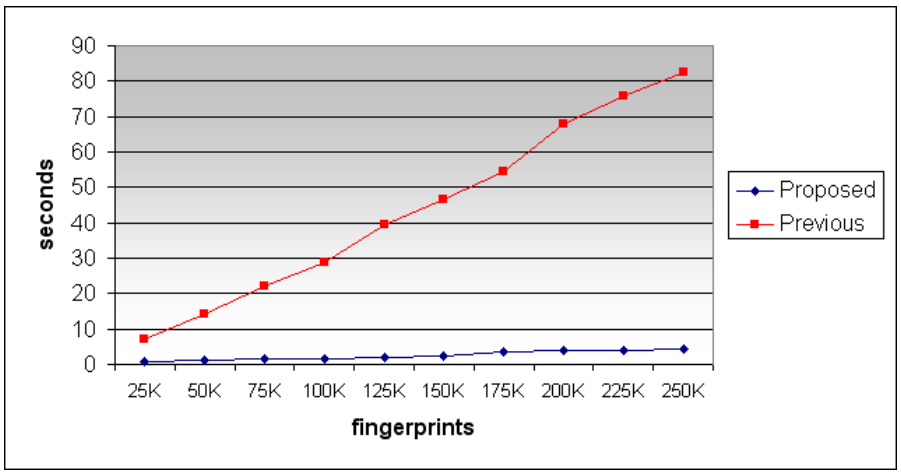

Figure 4.3: Previous work compared with the proposed solution.

at the database level since the *GROUPBY* operation has to compare the result of the queried n-grams. This in combination with the fact that there naturally will be more candidates also slows the performance of the proposed algorithm. This is described in details in the next section.

4.3.2 Worst Case Database Scenario

To show how much faster the database server performs under difficult conditions compared to the past work, a worst case scenario has been created. As mentioned above the greatest impact on the performance is when the lists within a queried n-gram in the index is of great size. As more lists in the index gets larger the performance decreases. This naturally also influences the proposed algorithm when querying the database for similar matches but the effect is far from the previous work as figure 4.4 shows.

A new dataset has been created where every fingerprint contains the same n-grams except one fingerprint that contains one unique n-gram more than the others. All n-grams are of size 3, and are positioned at different locations giving unique fingerprints. When the query fingerprint contains the same n-grams and the one unique, corresponding to 8 seconds of audio, it causes the past work to iterated over significantly large index lists. This has great influence on the performance as figure 4.4 shows. But also the proposed solution gets a worst case scenario on the database server as the figure also shows. This is because more objects has to be grouped in order to count n-gram contained in the fingerprints.

As described in Section 3.7.1 the worst case would occur if the query contained 100% of the n-grams in the database, since all fingerprints would have a similarity

factor of 100.

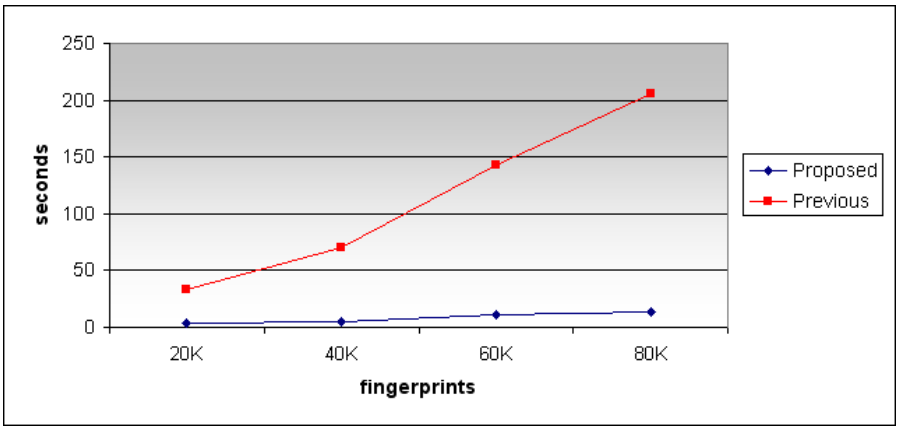

Figure 4.4: Worst case scenario for previous work

4.3.3 Summary

Having performed the tests of the proposed solution it is clear that it is orders of magnitude faster than previous work, and still guarentee that no order fingerprint in the database is a better match. The different parameters was tested, and explained to give a intuition of what values will be suitable for a given context. The values used in these tests was designed to work well with the fingerprint generator developed by Heljoranta [7]. Other fingerprint generators will produce a different dataset, and therefore other values should be applied. This, however, will not change the fact that the proposed solution is orders of magnitude faster than the previous work, since the logic of the candidate set will stand. To further improve the performance, it would be possible to distribute the fingerprint dataset across several database servers. Thus, the requirements to the database server could be minimal as the dataset increases. Each database server could then return a candidate set for the portion of the dataset it maintains.

Chapter 5

Conclusion

The goal of this thesis was to create a scalable backend for audio fingerprint matching that improves the search speed of existing techniques. The requirements were that the search technique should guarentee that no better match can exist in the database. The proposed solution complies with this requirement as the algorithm evaluates the fingerprints necessary in order to find the best fingerprint match. To improve the search speed, a new search strategy was developed that is capable of quickly finding an estimated candidate set. Having such a candidate set that is a subset of the original dataset results in significant fewer fingerprint evalutions. The experimental studies show that the proposed algorithm searches orders of magnitude faster than the previous technique that it improves upon.

The developed solution is efficient with fingerprint generators of good quality. Also, the search algorithm can be used with generators that produces different number of errors in the fingerprints. For example a setup on a desktop computer that recognizes digital audio files will contain no noise, and thereby no errors in the fingerprint. In contrast, mobile phones that record audio in noisy environments are likely to contains errors. The parameters should therefore be set in accordance with the system context. However, it is difficult to set the parameters to suitable values as different client users may be in different environments. As the environments changes, such a bus, a bar, or at home, the level and the diversity of the ambient noise will vary and therefore optimal parameters can be difficult to predict.

The experimental performance studies shows that the solution is scalable with respect to the database size. Futher, the solution can be a distributed across several backend servers to improve the performance.

Chapter 6

Future Work

Potential users of this backend might be discouraged by the difficulties in determining which parameters to use for a specific setup. The experimental performance studies show and explain what happens when the different parameters are set to specific values. But the parameters always depend on the fingerprint generators and the overall system usage. A high-quality fingerprint generator and a homogeneous and stable audio recording system can have parameter values that are suitable for low numbers of errors. However, a system that produces a high number of errors should adjust the parameters to accommodate this.

If the system contains different microphones that can produce different amounts of noise or the usage environment is heterogeneous, the parameters are difficult to set. To make the system easy to install and to be more compatible with different environments, a future work project could be to extend the solution to automatically determine and set the parameters for the specific context. For example, if the input signal is noisy and the parameters are set to be suitable for low number of errors, most likely, no matches will be found. However, if an input signal is without errors, and the parameters are set to support high amounts of errors, the search speed would become unnecessary slow. If the system could determine good values for the parameters before searching, the system could both be more easy to install, and the performance and quality could be improved even futher. The parameters could, e.g., be set according to which mobile phone or microphone that was used for recording the query song.

Testing a complete solution that is to be deployed in heterogeneous environments is difficult. The variations of the noise and the music quality are endless. Therefore, a future project could be to develop an application that can simulate changing environments. It could, e.g., simulate being in a bus while recording a song from a speaker of different quality or being amongst noisy people. Thus, the fingerprint generator and the parameter settings of the algorithm could be tested for correctness before releasing a solution.

REFERENCES

[1] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[2] A. Skovsgaard, L. T. Sørensen and M. Larsen. Audio Identification using N-grams. *Project at Aalborg University, Denmark* (Jan. 2008).

[3] D. Knuth. Sorting and searching. *The Art of Computer Programming, volume 3* (1973), 506–542.

[4] E. Zwickler. Subdivision of the Audible Frequency Range into Critical Bands (Frequenzgruppen. *The Journal of the Acoustical Society of America 33* (Feb. 1961), 248–+.

[5] Gracenote. Gracenote: Powered by gracenote, `http://www.gracenote.com/powered_by_gracenote/`.

[6] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathemtical Statistics and Probability* (1967), 281–297.

[7] J. Heljoranta. Future proof fingerprint function, `http://www.fsfe.org/en/fellows/juha/fpfpf`.

[8] J. W. Cooley, and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19* (1965), 297–301.

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan and D. Srivastava. Approximate string joins in a database (almost) for free. *The VLDB Journal* (2001), 491–500.

[10] M. Betser, P. Collen and J. Rault. Audio identification using sinusoidal modeling and application to jingle detection. *Austrian Computer Society (OCG)* (2007).

[11] MusicBrainz. Future proof acoustic fingerprinting techonology, http://musicbrainz.org/doc/FutureProofFingerPrint.

[12] MusicBrainz. Musicip, http://musicbrainz.org/doc/MusicIP.

[13] P. Ahrendt, A. Meng, J. Larsen and S. Lehmann. The clever toolbox - the art of automated genre classification. *ISP Group, Informatics and Mathematical Modelling, Tehcnical University of Denmark* (2005).

[14] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf. 1* (1972), 290–306.

[15] R.W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical 26* (1950), 147–160.

[16] S. F. Altschul, W. Gish, W. Miller, E.W. Myers and D. J. Lipman. Basic local alignment search tool. *J Mol Biol 215*, 3 (October 1990), 403–410.

[17] Sony/Ericsson. Sony/ericsson, http://www.sonyericsson.com/cws/products/mobilephones/overview/w850i.

[18] Tunatic. Tunatic, http://www.wildbits.com/tunatic/.

[19] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady 10* (Feb. 1966), 707–+.

[20] Z. Ning, A. J. Cox and J. C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Research 11* (2001), 1725–1729.

Resumé

This thesis documents the development and results of a generic backend capable of finding the best match for audio fingerprints.

The goal of this thesis is to create a backend that is scalable without loosing the reliability of the results. In past work by the author an algorithm was developed that could determine the best matching fingerprint. However, the solution was not scalable. It used a scoring system that was based on research made by a bioinformatics research team.

Since an audio signal can be subject to noise, the queried fingerprints does not match exactly with the database fingerprints. Thus, an approximate substring search is necessary. This thesis uses n-grams and hash tables for fast lookup. The best matching fingerprint is determined by a scoring system, where the scores depend on the amounts of correctly positioned n-grams in the fingerprints. The previous work was not scalable, since it evaluated all possible fingerprints.

This thesis provides a new strategy that involves the creation of a candidate set of fingerprints. The candidate set is a subset of the database fingerprints, where the candidate fingerprints are estimated as possible matches. This estimation is based on the number of n-grams, that the query fingerprint and the database fingerprints have in common. By not examining the n-gram positions a fast candidate set can be created. This candidate set is then searched by an algorithm that evaluates the scores of each candidate. If the fingerprints in the candidate set does not comply with the requirements for a best match, more fingerprints from the database will be evaluated.

The search algorithm guarantees no false dismissals and parameters can be adjusted to alter the reliability of the results. Experimental performance studies show which parameter values that are suitable for the suitable fingerprint generator. However, these values depends on the fingerprint generator used and the system context. The candidate set is orders of magnitude smaller than the dataset. Often the candidate set only contains one candidate which is the best match. The result is a solution that is orders of magnitude faster than related work.