Giving Music more Brains

A study in music-metadata management

Arjan Scherpenisse

Universiteit van Amsterdam

Giving Music more Brains

A study in music-metadata management

Arjan Scherpenisse

Master's Thesis

Artificial Intelligence,
Multimodal Intelligent Systems

Supervised by
Prof. Dr. M.L. Kersten
Dr. P. Boncz

November 2004

FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA

The research described in this thesis was performed during the author’s internship contract at Centrum voor Wiskunde en Informatica (Centre for Mathematics and Computer Science), within the Information Systems research cluster.

This thesis is ready to be marked.

Date: Author’s signature:

This thesis is ready to be read by the second readers.

Date: Supervisor’s signature:

Contents

Chapter 1

Introduction

The digital era has a profound influence on every aspect of our daily lives. In the music field, more and more music is produced in or transfered to digital formats, which range from conventional compact discs to MP3 files. Even so, music playing devices tend to become smaller (iPods, hard-disk walkmans), while their storage and networking capacities increase. In this setting, we see the need arise for the digital music player to autonomously provide music recommendations reflecting the personal music taste of its user, obtained from other music players in the network .

The paper of Boncz [9] describes such a recommendation system as an example application for *AmbientDB*, a new distributed database platform. In this setting, each AmbientDB-equipped music player exploits the knowledge about music preferences contained in the playlist logs of the thousands of users, united in a global peer-to-peer (P2P) network of AmbientDB instances.

1.1 Music recommendation using a P2P database system

In the application of music recommendation using a P2P database system, we can identify three main problems: the music identification problem, the music collection problem, and the music recommendation problem.

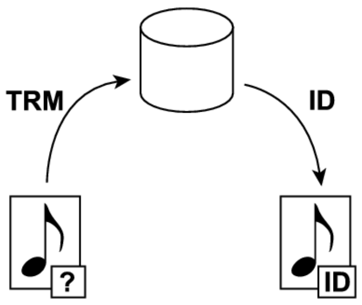


Figure 1.1: *The identification problem*

The music identification problem can be formulated as the problem of how to transform the inherently fuzzy metadata of a digital music track into a globally unique, consistent track identifier. The MusicBrainz project provides a solution in the form of an open source, centralized metadata database, maintained by human moderators.

The music collection problem deals with the collection of relevant playlog meta-data: what data do we collect (eg. the schema), and where do we store it (centralized vs. decentralized). The AudioScrobbler project [6] provides a fully centralized storage for all play log data for all participants in the project.

The third problem is about the actual music recommendation algorithm. What queries can be expected from a recommender application using a typical *collaborative filtering* algorithm, and what are the consequences of those queries with respect to the AmbientDB database system [9]. What typical schemas and search accelerator structures (indexes) might be used to maintain the scalability of the recommender application.

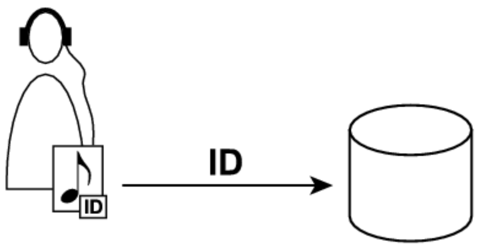


Figure 1.2: *The collection problem*

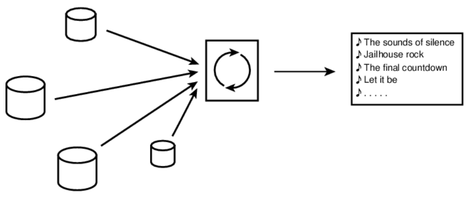


Figure 1.3: *The recommendation problem*

The research goal of this project is the investigation of several of the sub-problems regarding the *music identification problem*.

The original idea behind this thesis was to address all three problems and combine them into a music recommendation plug-in for a music player application, the recommendation plug-on built on top of the AmbientDB API, but as time progressed it became clear the music identification problem alone already contained many sub-topics with unanswered questions, so the research focus was shifted primarily onto that area.

1.2 Research topics

We have identified two problems related to the music identification problem, and which the MusicBrainz project, a centralized music identification effort, is currently experiencing. After investigating them we formalize a solution and provide experimental results. Besides that, a large part of this thesis consists of an investigation of the music metadata, which was required to gain insight into the problems. The exact research topics are formulated below.

- **Domain issues** – To understand the problem better we need to characterize the music data: the structure, growth, size and cleanness of it.
- **Aid moderators / clean up data** – We investigate the automatic cleaning of dirty data “the database way” by looking at different variants of the *similarity join*.
- **Database server scalability** – What will happen with the MusicBrainz project if the amount of data and user base grows by several orders of magnitude? We will investigate which server architecture will be most able to handle the increasing load.

1.3 Thesis overview

First, in chapter 2, we give an overview of scientific research fields related to the work in this thesis. Next, chapter 3 provides background information on subjects used in this thesis like music metadata and the MusicBrainz project. The next three chapters address the research topics as formulated above: In chapter 4, we give a survey of the field of music metadata and propose a model of the expected growth of the field over time. Chapter 5 proposes an aid to the MusicBrainz community by integrating similarity joins and similarity selects. Chapter 6 recognizes the need for a scalable multi-server database architecture and experiments with several of such architectures. This thesis ends with a discussion of the chapter conclusions in the light of the problem of P2P music recommendation using a database, and points out several directions for future research.

Chapter 2

Related research

As this thesis touches upon many subjects, much related literature has been published. In this chapter we will give an overview of the three relevant fields: Peer-to-peer databases, database cleaning algorithms and collaborative filtering.

2.1 Peer-to-peer databases

This section consist of an overview of database literature which relate to distributed and P2P databases. The survey gives an overview of the field and its issues.

The paper which provided the starting point for this research is the paper by Boncz and Treijtel, [9]. The paper proposes AmbientDB, a relational database management system, built on top of a Chord [54] P2P network. The relational algebra of AmbientDB is divided into three levels: abstract algebra, which is the standard relational algebra; concrete algebra, which is the translation of the abstract algebra into an algebraic form to handle local, distributed and partitioned tables; and a data-flow algebra which deals with the flow of the data between parent and child nodes. The final part of the article addresses the exploitation of Chord's built-in distributed hash tables as an indexing mechanism for query optimization.

Alonso and Korth [2] list several issues in "nomadic computing". Most importantly they see the need for adequate power supply: query processors need to include the power of a mobile system in their cost model. Furthermore, systems can be disconnected and reconnected at any time. Solutions have to be found as how to deal with transaction consistency and synchronization with the non-mobile system. Finally they mention the user interface: portable devices sport little screen real estate and pen-based input, which require novel user interface elements.

The article by Sartiani et al. [13] describes XPeer, a P2P-distributed XML database system. The system is designed to have zero administration: no human intervention is needed to administer the DBMS. The system is also very flexible: each node in the network can have its own schema. Nodes are grouped together into clusters, each cluster having a single super-peer, which aggregates all schema descriptions (tree guides) from its children. The super-peers together form a tree-based network. The query algebra XPeer supports is a large subset of XQuery FLWR. When processing a query, it is transformed into a location-specific algebra by traversing the super-peer tree, which aids in discovering interesting locations (nodes) for the query by inspecting the local tree-guide. After obtaining this location-specific expression, the issuing peer itself is responsible for executing the query.

Demers et al. describe in [15] three different algorithms and variants for the maintenance of a *fully replicated* database. With "Direct mail", each update is mailed to every other site directly from the originating site. This can be unreliable, because some sites could be down. In the "Anti-entropy" variant every node periodically chooses a random other node and synchronizes its whole database. This guarantees database convergence, but poses a heavy load on network traffic as the two databases need to be compared. The final variant, "Rumor mongering", facilitates updates spread like "rumors": sites are initially ignorant of an update; then they become infected, and spread the update to other sites until some end condition is satisfied. In both anti-entropy and rumor mongering, record deletions must be handled in a special way, since the item might be "resurrected" if the delete is not propagated to every site. This is solved by keeping "death certificates" at some (but not all) sites.

Gribble et al. comment in [24] on issues related with establishing a data management layer in a P2P setting. The strength of the distributed nature of P2P is that it is scalable: its performance grows as more nodes participate. Current P2P systems (Napster, Gnutella, etc) only do very simple data management or still rely on a central server containing the catalog. Four roles are described which a node in a P2P DBMS can perform: data origins, which provide original content, storage providers, which serve materialized views of content, query evaluators, which evaluate (parts of) a query, and query initiators, which initiate queries. Furthermore, the "data placement problem" is identified: how can we evaluate queries while keeping the (evaluation-, data transfer-) costs associated with it as low as possible. The authors introduce Piazza, their prototype implementation, which is still under development.

The article by Madden et al. [36] presents a method for collecting aggregate values from a sensor-network of independent devices called "motes". The devices are arranged in a tree-like network topology, which is discovered ad-hoc, and updated regularly to account for motes going offline. An overview of commonly used aggregates is given, and they are classified in several categories (eg. (non)monotonicity, exemplary/summary, etc). A pipelined strategy is described in which motes collect aggregate values (partial state records for computing the aggregate) in a streaming fashion. In every unit of time ("epoch"), every mote combines partial state records from its children and forwards it to its parent. The rest of the article deals with handling motes going offline, and reducing communication overhead. Methods are described to deal with these problems in an efficient way. An article by the same authors with W. Hong [35] describes the design of an autonomous query processor using this sensor network. Different sensor-network specific extensions to the SQL query language and its power-management related issues are mentioned.

Papadimos and Maier propose in [47] an XML-based query plan architecture. Query plans are sent to a server, which partially instantiates it with data it knows about, and sends the mutated query plan to other servers which it thinks can resolve some other parts of the query plan. This (parallel) architecture is compared to a "pipelined" version. The pipelined version performs better in terms of data transfer and server load, but worse in terms of evaluation time, as the plan is evaluated sequentially. A drawback of the system is that each server must know a way how to resolve encountered partial query plans, or (at least) provide a mapping to other servers that know how to.

2.2 Database cleaning algorithms

One of the research questions of this thesis is, whether it is possible to provide an automatic means to clean up the existing music databases. In this light, this section surveys published database-cleaning algorithms.

Gravano et al. describe in [34] a method for calculating an approximate join between two tables in which rows are joined on two attributes of the tables which are within some minimum edit distance k. The solution the authors of the article propose is to decompose the attributes on which the join is to be calculated into $Q-grams$: substrings of length q, and to perform a join on this table by using three types of filtering. *Length filtering* is based on the fact that if $abs(|s_1|-|s_2|) > k$, the edit distance can never be lesser than or equal to k. *count filtering* takes into account the number of similar Q-grams, and *position filtering* takes into account the relative position between corresponding Q-grams.

In [23], Gravano implements a sample-based cosine-similarity text join in a DBMS. Two relations are joined by splitting the attributes into tokens: Q-grams, each weighted by the TF/IDF measure. A naive text join algorithm (their baseline implementation) would join any of the two attributes which have at least a single token in common, and calculate the cosine similarity for each of these combinations. The authors propose a sample-based approach based on the fact that for a cosine similarity to be high (which is desirable), at least two weights of the same token must be high too. The token weight space is sampled with a probability for each token proportional to its weight. This sampled relation is used in the calculation of the cosine similarities. The sampling relies on a parameter S, which specifies that for each token at least S samples must be taken approximately.

Ananthakrishna et al. describe in [3] a system for eliminating fuzzy duplicate entries from a relational database by exploiting the hierarchical structure in the database. The duplicate elimination task consists of grouping all rows from the join on the table hierarchy which can be marked duplicate. Once such groups are found, a canonical, representative tuple for each group can be taken as the replacement. The actual duplicate elimination is done top-down: from the most general relation ("country") to the most specific ("customer"). Tuples from the relations are compared on the textual similarity (token containment metric), and on the co-similarity of the tuples in their child relation (foreign key containment metric). The decision whether two tuples are duplicates of each other is made by a weighted vote of both of these metrics.

In the article by Cohen [14], a logic called WHIRL is proposed which reasons explicitly about textual similarity of local names in a database, to deal with the lack of global identifying names. Similarity is measured here using a vector-space model. Using data extracted from the world wide web, it is shown that WHIRL is much faster than naive variants. However with regard to this thesis, this paper uses a specialized system to deal with the properties of the logic language, so it is not directly applicable in an regular database system.

2.2.1 Rank aggregation

Fagin, in [16], describes an efficient similarity search algorithm which integrates similarities from multiple dimensions in the database in a nearest-neighbor fashion. The approximate rankings obtained from each dimension are combined by a highly efficient aggregation algorithm. The algorithm aggregates the rankings from each dimension by taking the median values, and returns the top-*n*. A nice property of the algorithm is that it is database-friendly: virtually no random accesses are needed and only a small part of the database has to be accessed to obtain the top *n*.

2.3 Collaborative filtering

As this thesis started as an effort to build a decentralized, database-based recommendation engine, we have put much work in researching the state of the collaborative filtering (CF) literature: its relation to implicit rating of items as well as the decentralization of the algorithms. The survey below gives an overview of important established collaborative filtering algorithms.

The earliest real CF paper is the GroupLens paper by Resnick et al. [50]. Written in 1994, the article describes the GroupLens architecture: an open system for rating newsgroup articles. Separate servers are set up for processing ratings and predicting ratings for articles to users. The collaborative filtering technique used is a correlation based method, no evaluation (or implementation of any other technique) is done. The article focuses on the impact of the rating method on newsgroup usage.

The empirical analysis by Breese et al [31], which gives an overview of correlation-based (GroupLens) and vector-similarity based methods, Bayesian networks, Bayesian clustering, as well as a baseline-method, ’popularity’. Evaluated on three different datasets, the correlation based methods and Bayesian networks turned out to be the best performing algorithms in most of the cases.

The paper by Upendra [56], describes “Ringo”: a music recommendation system which works by email. Users initially rate a number of artists (on an absolute 1-7 scale), and then can get recommendations mailed to them. The system utilizes several collaborative filtering methods, being Feynman’s Mean Squared Difference, the Pearson-R Algorithm, the Constrained Pearson-R Algorithm, and an Artist-to-Artist based algorithm. The evaluation focuses on qualitative results, obtained from the 2000+ user base. One of the future research items that are listed is “Distributed Ringo” - just what we want to implement.

Goldberg [22] proposes “eigentaste”: a new collaborative filtering algorithm which requires that every user, before receiving recommendations, first rate every item in a predefined gauge set. This yields a dense set of ratings, thus dealing with the sparseness problem that is recognized in many CF problems. On this dense rating matrix, the user-correlation is computed and then PCA is applied onto this matrix to perform dimensionality reduction. The reduced PCA space is then clustered. On recommendation, a user preference vector is first projected into the PCA space, and the corresponding cluster is found. Then, items from this cluster are returned as the recommendations. The eigentaste algorithm performs as well as a regular 80-NN CF algorithm (with similarity measure based on Pearson correlation), but is significantly faster in generating recommendations (constant instead of linear time).

The paper by Canny, [12], describes a method for collaborative filtering in which every clients preference is hidden from the other clients. This is established by calculating a ’user preference aggregate’, using encrypted data from all clients. The aggregate calculated is an instance of the SVD collaborative filtering method (eigentaste), but modified so it works with encrypted client vectors. The aggregate is public, and can be used to generate recommendations for users, but it contains no specific data about particular clients.

Hofmann, in [27], describes a probabilistic modeling method for collaborative filtering which uses latent variables z for modeling relations between users and items. The variables z model different aspects of the users preference. A co-occurrence model of variables is constructed and fitted through standard Expectation Maximization. Evaluation, done on three proposed variants, shows that the co-occurrence model with the preference extension is superior to both other methods. Also, qualitative experiments yield satisfied users.

In a later article, [26], Hofmann extends the co-occurrence model to incorporate numerical ratings (instead of just binary ones). The distribution of these ratings is assumed to be of a Gaussian shape. User normalization is applied due to the fact that users might exhibit different rating patterns (even when rating on the same scale). The model (which has two parameter sets: the hidden states Z and the Gaussian parameters) is then again calculated with the EM algorithm. The Gaussian (normalized) version outperforms the baseline memory-based (Pearson-R) method in a 'rating prediction' scenario.

2.3.1 Implicit rating

Collaborative filtering systems require ratings of an explicit kind: numbers on a fixed scale or in a specific range. However, there are many applications in which explicit rating of items is inappropriate; also, in some systems the benefits of rating items are not directly clear. Using implicit ratings removes the effort of explicitly rating an item: the rating is established through observing a users actions on the item. In our music recommendation application we include such implicit rating: music preference is expressed by the (frequent) playing of songs.

Nichols, in [44], refers to the GroupLens article, and describes an extension that has been made to it to include implicit ratings. The article concludes with the notion that the effectiveness of implicit rating systems remains unclear at the moment, because little experimentation is done.

Oard et al. [45] refine the notion of implicit feedback by dividing it into three categories: examination, retention and reference. Observing 'examination' actions can provide first cues of the users interest, observing 'retention' action confirm this interest (items get saved or printed); the 'reference' category observes actions which establish links between objects. For using implicit feedback, two strategies are identified: the rating estimation strategy, in which a rating is inferred from the observed actions, and used to predict new ratings, and the predicted observations strategy, in which past observations are used to predict user behavior in response to new information.

2.4 Conclusion

This chapter has provided us with insight in the current state of the literature regarding our research topics. We can conclude that the P2P-database literature is still in its infancy; there is not much known about good methods to do databases on a peer to peer network. However, the older articles about distributed databases have helped us in formalizing solutions in the Server Architectures chapter.

Much is written about data cleaning, however, there are few articles which propose system-independent algorithms which can be implemented in any DBMS. The methods proposed by Gravano were very helpful implementing cleaning algorithms in MonetDB and PostgreSQL, in chapter 5.

Finally, the articles about collaborative filtering were many, and they provided interesting reads, especially given the author's background in Artificial Intelligence. As the research focus of this project shifted away from music recommendation to music data management, however, it became clear that the collaborative filtering field could no longer contribute to the new research topics, and thus will the P2P distribution of recommendation algorithms remain an area for future research.

Chapter 3

Background information

This chapter provides background information for gaining insight into the music identification - collection - recommendation problems. The chapter starts off with an overview of musical metadata properties in section 3.1, and gives an overview of the inner workings of the MusicBrainz system in section 3.2. Then it turns to investigate the properties of the AudioScrobbler project in section 3.3. Finally, background on the two database systems we have used is provided in section 3.4.

3.1 Music metadata survey

Digital music can contain a wide variety of metadata. Below is a survey of the most common metadata which can be stored with and is used to describe the content of digitized music files.

- **Audio properties** – Properties which can always be extracted from the raw audio data are the length of the song (in time units), and values based on analysis of the audio file. Common techniques are spectrum analysis, wavelet-based techniques, number of zero-crossings and calculating Haar-transforms.

- **Meta-tags** – Music files stored on digital devices often have associated textual metadata, through a data block embedded in the music file. The most common tag format supported by most players, is ID3v1 [29] (with a fixed set of tags: song title, artist, album, year, comment, genre). Its successor, ID3v2 [30], features a variable number of fields and a more elaborate choice in field types, but is less widely used. Even less used file formats which provide metadata tags are the Ogg Vorbis file format [46] (whose tags are also embedded in the FLAC file format [17]) and the APE file format [4].

- **Fingerprints** – A novel approach to music matching is to calculate a unique “signature” from the acoustical properties of the audio file which will remain unaffected by transformation of audio format, compression, and even some forms of distortion.

  Several different implementations of these techniques exist. Philips [7] has developed and published about an algorithm which produces fuzzy hashes. Hashes of perceived identical songs differ by a significantly lower number of bits from hashes of non-identical songs.

  Microsoft Research have developed [11] a robust technique called Distortion Discriminant Analysis, which extracts fuzzy “traces” from the signal, which are then matched in a very efficient way against a database of precomputed signatures.

  TunePrint [55] claims to have a fingerprinting technology using “psychoacoustic and statistical technologies” as well, (the company was started by an MIT student), but the company seems dead.

  Finally, the technique used by the MusicBrainz project is the service from a company called Relatable [49]. This proprietary algorithm uses acoustical properties derived from the signal to calculate the hash. The actual fingerprint-calculation routine is proprietary.

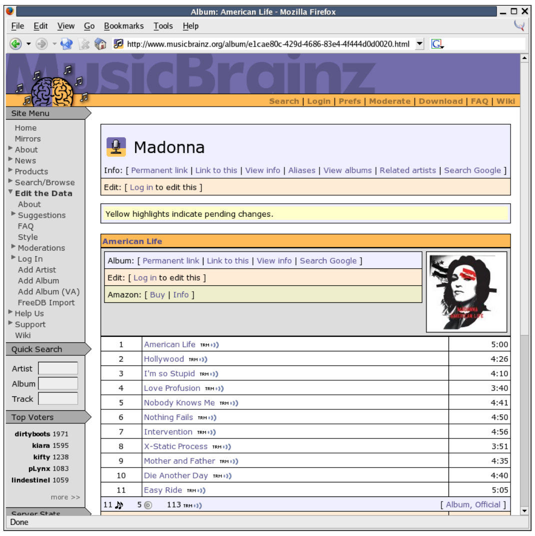
The problem with all types of metadata is that they are fuzzy, in some degree. There is no single property which uniquely describes an audio track. From a computer point of view, this is quite logical, but from a human, perceptual point of view, it is not. Fact is, there is a correlation between the (perceptual) similarity of tracks and the similarity of metadata of tracks. For instance, if a track user A has is 180 seconds long, it is likely that the same track, owned by user B (which might be a completely different file format, bit rate etc), is about 180 seconds too. This “perceptual equivalence” is an important quality which is typically incorporated by metadata similarity measures.

3.2 About MusicBrainz

MusicBrainz is a community-driven music meta-database that attempts to create a comprehensive music information site. MusicBrainz collects information about digital music using a tagger program and makes it available to the public so that users can semi-automatically can annotate their digitized music collection with complete and correct metadata.

MusicBrainz can be accessed through two different interfaces: the website and the MusicBrainz tagger. The web site, shown in figure 3.1, serves as a portal for on-line browsing and editing of the music data. The dedicated MusicBrainz Tagger program is a program to identify (to “tag”) music files on a users’ local machine.

There is a rather large (57k+) user community around MusicBrainz which ensures that the data (which enters MusicBrainz’ central database automatically through the tagger program) is kept “clean” of spelling errors and other mistakes.



Madonna

Info: [Permanent link | Link to this | View info | Aliases | View albums | Related artists | Search Google]

Edit: [Log in to edit this]

Yellow highlights indicate pending changes.

American Life

Album: [Permanent link | Link to this | View info | Search Google]

Edit: [Log in to edit this]

Amazon: [Buy | Info]

1	American Life TRM	5:00
2	Hollywood TRM	4:26
3	I'm so Stupid TRM	4:10
4	Love Profusion TRM	3:40
5	Nobody Knows Me TRM	4:41
6	Nothing Fails TRM	4:50
7	Intervention TRM	4:56
8	X-Static Process TRM	3:51
9	Mother and Father TRM	4:35
10	Die Another Day TRM	4:40
11	Easy Ride TRM	5:05

11 ♫ 5 ◎ 113 TRM [Album, Official]

dirtyboots	1971
kiara	1595
kifty	1238
pLynx	1083
lindestinel	1059

Figure 3.1: *The MusicBrainz website*

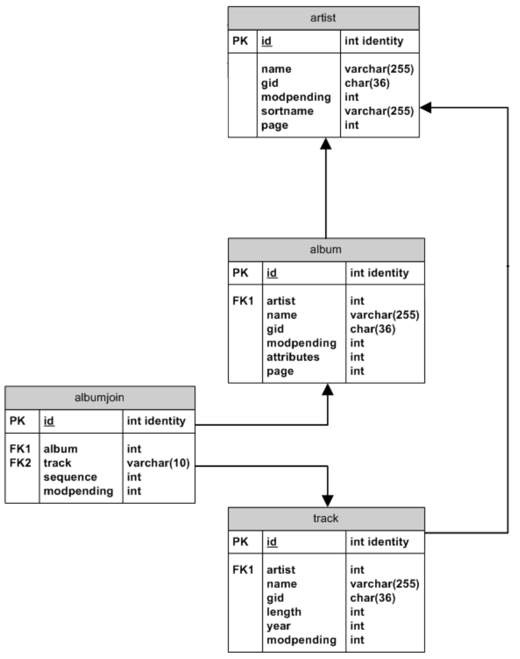


Figure 3.2: *The MusicBrainz schema (partial)*

3.2.1 The database

The core of the MusicBrainz project is the music metadata database. The database, which is a PostgreSQL database (see section 3.4), holds information about recorded music. The goal is to create a complete database, holding all music metadata information about every instance of recorded music.

The database holds information about artists, tracks, albums, in the obvious schema, depicted in figure 3.2. The `artist` relation holds all information about artists. Artists can have multiple albums, so the `album` relation has a foreign-key relationship to the artists relation. Because it is the case that the same track can occur on different albums (eg. in compilation albums), there is a many-to-many relationship between `album` and `track`, through the `albumjoin` relation[1].

[1]Despite that the schema allows it, in the current MusicBrainz incarnation every instance of a track occurs on *one* album only, so identical tracks on different albums are viewed as different tracks.

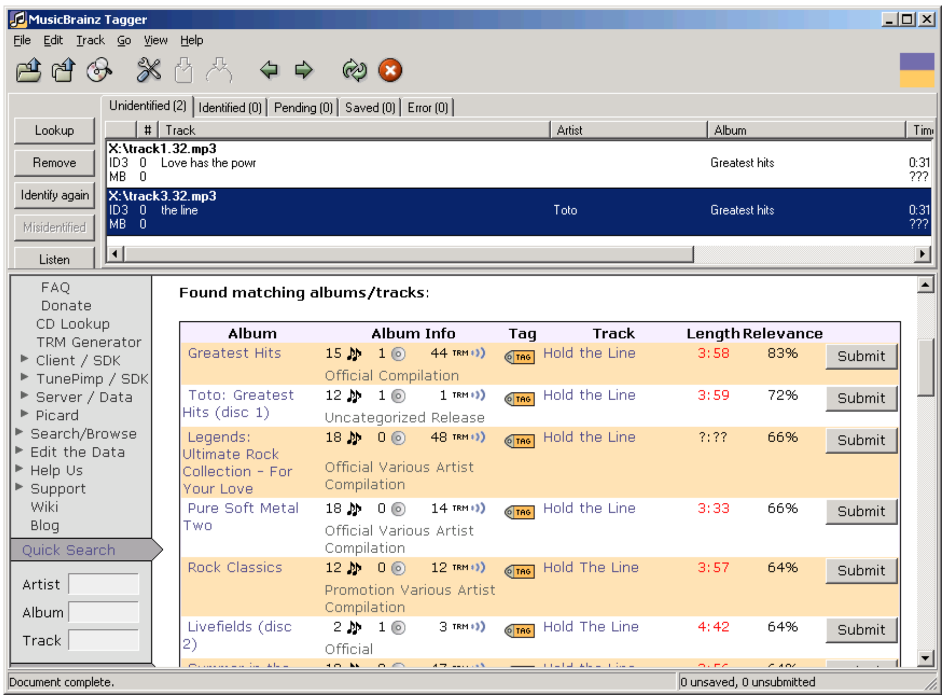
Besides these relations, the database contains additional relations with information about individual CD's, management of moderations and moderators, and statistics about the data. However, in this thesis only the primary tables as displayed in figure 3.2 are of our concern.

3.2.2 The tagger

The tagger program is a helper program with a twofold purpose. On the one hand it aids users by cleaning their digital music collection of spelling errors through automatic identification and labeling of the music files. On the other hand it helps the MusicBrainz community because new music metadata previously unidentified by the tagger can be submitted to the MusicBrainz database, causing the database to grow.

The tagger identifies music by looking at a generated *audio fingerprint*, which is generated from the acoustical data of the audio file. The fingerprinting service is kindly offered by Relatable [49], a company specialized in content identification technology. However, the actual fingerprint-generation code is proprietary: to protect the algorithm, the service is offered through a web service hosted at a dedicated machine.

The tagger program extracts perceptual properties from the audio file and submits this information (566 bytes) to the Relatable web service. The response is a 36 characters long fingerprint, known as the *TRM hash*. This hash serves as a fuzzy key for the song in the database which is represented by the audio clip. Note that the hash is *fuzzy*, which means that there is a non-zero (but quite low) probability that two tracks yielding the same hash are non-identical, and reverse.



	#	Track	Artist	Album	Tim
X:\track1.32.mp3					
ID3	0	Love has the powr		Greatest hits	0:31
MB	0				???
X:\track3.32.mp3					
ID3	0	the line	Toto	Greatest hits	0:31
MB	0				???

Found matching albums/tracks:

Album	Album Info	Tag	Track	Length	Relevance	
Greatest Hits	15 1 44 TRM Official Compilation	TAG	Hold the Line	3:58	83%	Submit
Toto: Greatest Hits (disc 1)	12 1 1 TRM Uncategorized Release	TAG	Hold the Line	3:59	72%	Submit
Legends: Ultimate Rock Collection - For Your Love	18 0 48 TRM Official Various Artist Compilation	TAG	Hold the Line	?:??	66%	Submit
Pure Soft Metal Two	18 0 14 TRM Official Various Artist Compilation	TAG	Hold the Line	3:33	66%	Submit
Rock Classics	12 0 12 TRM Promotion Various Artist Compilation	TAG	Hold The Line	3:57	64%	Submit
Livefields (disc 2)	2 1 3 TRM Official	TAG	Hold The Line	4:42	64%	Submit

Document complete. 0 unsaved, 0 unsubmitted

Figure 3.3: *The MusicBrainz tagger, showing a list of song suggestions on the encounter of a "miss".*

The MusicBrainz database has a `trm` relation, storing the unique TRMs, and a `trmjoin` relation which joins `trm` to `song`. These relations imply that there can be multiple songs which have the same TRM, and multiple TRMs pointing to a song, which follows from the fuzzy nature of the TRM as stated in the previous paragraph.

We now describe in detail the procedure at which the MusicBrainz tagger identifies tracks. Additionally, the full pseudo-code is given in algorithm 3.1.

For every track in a users collection, the TRM hash is calculated using Relatable's service. For this TRM, the MusicBrainz track id is looked up, by performing a double join on the corresponding tables. The SQL code is shown in query 1.

Query 1: *The MusicBrainz "track identification" query*

```
SELECT track.gid
FROM TRM, TRMJoin, track
WHERE TRM.TRM = (trm) AND
      TRMJoin.TRM = TRM.id AND
      TRMJoin.track = track.id
```

If this query results in a "hit" (eg. there is at least one result), there is a 84.4% chance that the TRM resolves to 1 track [41]. Even when the result is a single row, a similarity check on the track's metadata is performed to double-check the match. This is necessary since it might be the case that the TRM is "weak", resolving to more than one track, but the other tracks for the TRM are not yet in the database. If this double-check is not successful, the user is given the option to manually submit information about the track and the album the track came from.

The chance that a "hit" occurs, given all tag requests, is called the "hit ratio", H. This *hit ratio* is defined as $H = \frac{hits}{misses+hits}$.

```
SELECT trackid FROM trmjoin WHERE trm = trm
if query result == HIT then
  Log a HIT
  SIMILARITY MATCH between metadata and query results
  if similarity condition satisfied then
    Return trackid to user
  else
    Ask user to enter / confirm data (moderation)
  end if
else {query result == MISS}
  Exact metadata lookup using MusicBrainz
  if query result then
    Log MISS → HIT
  else
    Log MISS → MISS
  end if
  User intervention is required to link the file to the correct track.
end if
```

Algorithm 3.1: *The MusicBrainz track identification process*

If the query results in a "miss", the TRM does not yet exist in the database. In this case, either the song is unknown, or the TRM is a new TRM for a known song. MusicBrainz performs *exact* selections[2] on the track and artist names, to provide the user with suggestions to aid in manual identification of the song. In figure 3.3, the tagger is displayed showing such a list of song suggestions, obtained using direct lookup of the artist name.

For every track that resulted in such a "miss", the user gets the option to manually identify the tracks by using the MusicBrainz website to browse to the correct album the track originated from. Once the right track is found, the user can click a small button next to the track's name on the site to link the newly generated TRM id to the track id which is already in the database. This new (TRM, track) pair is stored in an internal list and can be sent to the database server by clicking the "submit" button. This button basically triggers on the server for every (TRM, track) pair a query as in query 2.

[2]This exact lookup is an issue which might be improved: section 5.6 introduces the *similarity select* on the database, to provide better, "fuzzy" suggestions.

Query 2: *The MusicBrainz "submit" query*

```
INSERT INTO trmjoin (trm, track)
SELECT *
FROM ((trm id), (track id)) AS data
WHERE NOT EXISTS
      (SELECT trm FROM trmjoin WHERE trm = (trm)
       AND track = (track));
```

This query basically inserts the (trm, track) pair, while making sure that the pair is not already in the database.

3.2.3 The human factor

Every action which will cause mutations in the database (being the editing of the data through the web interface and the submission of new data through the tagger) goes through a moderation work-flow before the mutation is actually applied. Such an action creates a moderation proposal (represented by the `moderation` relation), on which users can vote. Once a moderation has had enough "yes" votes, the vote gets accepted and the proposed change forwarded into the database.

To ensure correctness and consistency, the data in MusicBrainz must confirm to a strict set of guidelines, which are published on their website [42]. It is often the case that erroneous data is entered into the database, or that the data does not comply with the guidelines. Through the web interface, all errors in the data can be corrected. For instance, if a new artist "The Baetles" was entered, a user who encounters this can create a moderation proposal to merge this artist into the already existing artist "The Beatles". Users who have edited much data and voted on many moderations, thus earning a good reputation, can be elected to become "auto-moderator". An auto-moderator can edit the MusicBrainz data directly, unhindered by the moderation process.

It is clear that the MusicBrainz project requires a lot of man-power to ensure the lasting cleanness of the data. This is only viable if the moderator base grows with the same rate as the music collection (and the user base).

3.3 AudioScrobbler

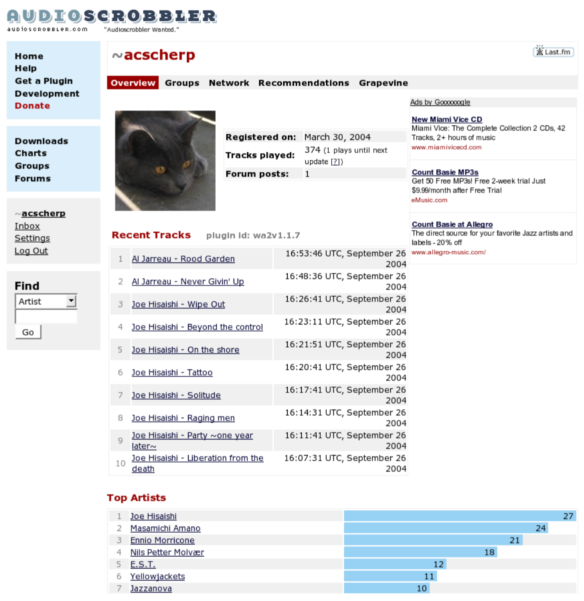
In the Data Survey chapter we use data sets obtained from the AudioScrobbler project while looking at the usage of music data and the collection sizes of users. Although the project and its issues are not of our primary concern as it is not part of our research topics, in this section we briefly introduce the AudioScrobbler system and highlight the problems the project has been facing.

AudioScrobbler is a community web site, building up a detailed profile of its users' musical tastes. The system, facilitating a plug-in installed in a users digital music player, sends the metadata of every song being played to the centralized AudioScrobbler server. Using this information, the server builds a 'Musical Profile' of the user, which can then be used to (manually) find users with alike musical tastes. A long-outstanding item on AudioScrobbler's wish list are personalized music recommendations, however this has not yet been implemented.

The musical profiles of every user can be viewed on the web site. An example of a page of a users profile is in figure 3.4. The profile is highly linked: clicking on a track or artist name lists statistics on how much that track/artist is played, and by who.

During the 7 months (March - October '04) that I have actively followed the news around the project, it became clear that the project is dealing with severe scalability problems. The system is centralized and under a constant load (about 10,000 users use the system on a daily basis). The database server has been upgraded a few times with help from donations from users; a "subscription"-based system has been made to gain revenue from users. The registration of new users was even closed for about a month to reduce the load on the server, in anticipation of a server upgrade. Due to this server upgrade and a change in database schema, the server has been running smoothly again in the last two months.

Data cleanness seems to be another principal issue for the project. In its first version, the AudioScrobbler plug-in did not use any music identification whatsoever before submitting tracks: all metadata was sent to the server, being clean or not. Moreover, in the absence of metadata-tags, the metadata was guessed from the file names of the music files. It quickly became clear that this was not the right way and in its second iteration,



~acscherp

Registered on:	March 30, 2004
Tracks played:	374 (1 plays until next update [?])
Forum posts:	1

Ads by Goooooogle

New Miami Vice CD
Miami Vice: The Complete Collection 2 CDs, 42 Tracks, 2+ hours of music
www.miamivicecd.com

Count Basie MP3s
Get 50 Free MP3s! Free 2-week trial Just $9.99/month after Free Trial
eMusic.com

Count Basie at Allegro
The direct source for your favorite Jazz artists and labels - 20% off
www.allegro-music.com/

Recent Tracks plugin id: wa2v1.1.7

1	Al Jarreau - Rood Garden	16:53:46 UTC, September 26 2004
2	Al Jarreau - Never Givin' Up	16:48:36 UTC, September 26 2004
3	Joe Hisaishi - Wipe Out	16:26:41 UTC, September 26 2004
4	Joe Hisaishi - Beyond the control	16:23:11 UTC, September 26 2004
5	Joe Hisaishi - On the shore	16:21:51 UTC, September 26 2004
6	Joe Hisaishi - Tattoo	16:20:41 UTC, September 26 2004
7	Joe Hisaishi - Solitude	16:17:41 UTC, September 26 2004
8	Joe Hisaishi - Raging men	16:14:31 UTC, September 26 2004
9	Joe Hisaishi - Party ~one year later~	16:11:41 UTC, September 26 2004
10	Joe Hisaishi - Liberation from the death	16:07:31 UTC, September 26 2004

Top Artists

1	Joe Hisaishi	27
2	Masamichi Amano	24
3	Ennio Morricone	21
4	Nils Petter Molvær	18
5	E.S.T.	12
6	Yellowjackets	11
7	Jazzanova	10

Figure 3.4: *The AudioScrobbler website*

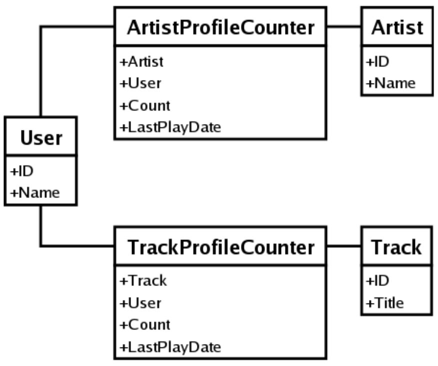


Figure 3.5: *The most recent AudioScrobbler schema (partial)*

the AudioScrobbler schema uses the MusicBrainz service to attempt identification of the music that is playing. If the MusicBrainz lookup is not successful, however, the (dirty) metadata still get submitted. Besides this, to reduce the database size, the choice was made for the new schema to store the submitted data in an aggregated way, losing detailed information about entry submission times and played tracks. Section 4.1 goes into detail on both schemes.

The latest development in the AudioScrobbler area is that a beta test of a moderation process much like MusicBrainz is going on, in which moderators can correct submissions with missing data, manually linking them to the correct MusicBrainz database entries.

3.4 Database systems

This thesis will continuously mention two different database platforms which we have used in investigating and evaluating the various research topics: MonetDB and PostgreSQL. This section introduces both systems, focusing some more on MonetDB as this is a relatively unknown system.

PostgreSQL [48] is considered the most stable and mature open-source database platform. It originated as a research prototype created by UC Berkeley, and is since 1996 actively developed by an open-source developers community. The version we have used was version 7.2.4, using a default configuration.

MonetDB [38] is an open-source database developed by the INS1 cluster of the Center of Mathematics and Computer Science (CWI). MonetDB is a main memory database with high performance in query-intensive applications, and is still under active, research-driven development.

The low-level language for interaction with the MonetDB database is the MonetDB Intermediate Language, MIL. This language is a procedural language, extended with operations and data types specialized for database algebra. Other front-ends to MonetDB, like the SQL front-end, convert queries at run-time to MIL code which the database then executes. Although we have used the MIL language in chapter 5 for MonetDB-specific query optimization, we mostly interacted with MonetDB through its SQL interface, so that queries would be portable between MonetDB and PostgreSQL and performance figures could be compared.

In this thesis I have always used the most recent development version, watching the system and especially the SQL front-end becoming more mature in the course of the months. An extensive discussion of MonetDB, its background and inner workings can be found in [8].

Chapter 4

Data survey

To gain better understanding of the music identification problem, we did an investigation of the music data on the Internet. Knowledge about the music data and its domain is crucial to correctly make choices on many of the research questions. This chapter characterizes the size, structure and growth of the music metadata domain.

Section 4.1 summarizes the different data sources which we have used in this chapter. Section 4.2 lists the different entities which are important to us and which we will investigate, and then surveys these entities and interprets the results by investigating their growth, structure, size and cleanness. Finally, section 4.3 proposes an analytical model for the music data properties based on conclusions we have drawn in the previous sections, and discusses the consequences this model has for the future.

4.1 Data sources

Below a listing of the different data sources that were used in order to create the statistics in the rest of this chapter.

- **the MusicBrainz database**

  A full dump of the MusicBrainz database, in the schema as depicted in figure 3.2. The main tables have the following sizes:

Table	*Rows*	*Size*
album	184172	17MB
albumjoin	2284368	58MB
artist	112515	9,4MB
artistalias	12913	0,7MB
track	2284522	175MB
trm	2207487	106MB
trmjoin	2996347	65MB

- **MusicBrainz - Tagger subsystem access logs**

  We also used the access logs of the web server the MusicBrainz tagging back-end runs on. These logs provided insight in the amount of ’tag’ actions that were made by clients. The result of the tag action (hit or miss), however, remained unclear as the response data of the web server requests were not logged.

  The access logs consist of the relation *(profile, date)*. Every tuple in the relation represents a tag action of a user on a specific date/time. The original log file contained the timestamps and IP addresses of every occurrence of a tag request. To convert the IP address to a user profile identifier, we have assumed that every user had a unique and static IP address.

- **MusicBrainz - Miss/Hit logs**

  To gain insight in the distribution of hits/misses the taggers generated, a special debugging flag in the MusicBrainz code was turned on for a week on request, logging the number of TRM lookups that resulted in a ’hit’ (the TRM was already in the database) or in a ’miss’ (the TRM did not exist yet). The log was collected during three days. Each day, about 200.000 log entries were collected.

This Miss/Hit log is a relation which *only* holds the result of the lookup. A HIT was recorded if the requested TRM existed in the database. If not, MISS → MISS was logged if the consecutive *exact* metadata lookup did not yield any result, and MISS → HIT otherwise. See algorithm 3.1 for the details on how this information was obtained.

- **AudioScrobbler - new schema**

  The AudioScrobbler project has released several log files to the public. The project currently collects play-log information from users into an aggregate form; this data is available as a $(profile, artist, playcount)$ relation. This *new data log* was recorded during 3 months end 2003 / begin 2004.

  In the new data log, there are 5476 distinct users. Together, they played songs of 16870 distinct artists. In total, 12265282 played songs were logged. We do not know the collection size (in number of tracks) of the users, since AudioScrobbler only stores play-counts per artist, not per track. If we assume that every AudioScrobbler user has played every artist in his collection at least once, the average AudioScrobbler user has 232 artists in his collection.

Distinct users	5476
Distinct artists	16870
Distinct tracks	N/A
Total log entries	12265282
Mean collection size	232 artists

- **AudioScrobbler - old schema**

  Before November 2003, AudioScrobbler kept data in a more elaborate form: namely play-log entries per track and per user, in the following relation: $(profile, artist, track, timestamp)$. This *old data log* was recorded during two months in 2003.

Distinct users	12431
Distinct artists	119094
Distinct tracks	826793
Total log entries	4516232
Mean collection size	62 artists 202 tracks

- **RIAA record sales figures 2004** [51]

  The record sales figures of the Recording Industry Association of America of 2004.

- **The 'How much information' 2003 report** [28]

  This report by UC Berkeley is an attempt to estimate how much new information is created each year. Newly created information is distributed in four storage media: print, film, magnetic, and optical, and seen or heard in four information flows: telephone, radio and TV, and the Internet. Especially the section on Peer-to-peer data traffic was of interest to us.

4.2 Data characteristics

This section gives a survey of data characteristics of the music data domain. We will investigate the following entities:

- **Database / domain sizes and structure** - We need to know how large the data sets are we will be dealing with, both in its current states (as in current music databases), as well as the (estimated) size of the music domain. Moreover, we need to investigate the structure of these entities and how is the data in these relations distributed.
- **User collection** - What can we say about the digitized music collection of our users: what is its typical size, and how do collections relate to each other, regarding overlap.
- **Data usage** - How is the music data used: how do users of Internet music communities use its databases.
- **Data growth** - How fast does the music domain, music databases and their user bases grow.

4.2.1 Database / domain sizes and structure

We need to know what the order of magnitude is of the data we are dealing with. Therefore we look at the sizes of existing music databases and take a guess at the size of the *Music Domain*.

Table 4.1 shows some statistics about three popular online music databases.

Database	Artists	Albums	Tracks
MusicBrainz	95,458	153,832	1,906,204
AllMusic.com	*203,115*	636,565	5,340,244
FreeDB	*432,958*	1,356,893	*16,857,088*

Table 4.1: *Music Database system statistics*

Note: the cursive numbers indicate numbers which were unavailable to us. They have been estimated by extrapolating the data with the calculated artist/album and album/track ratios from the MusicBrainz data set.

We expect that in its startup phase, music databases grow fast, as there is a lot of catching up to do. In time, as the database size converges to the domain size, the collection starts to grow slower, and in the limit, the collection size is a linear increasing function of the time (if we assume every year about the same amount of new original music is released), identical to the growth of the domain.

The size of the music domain is hard to estimate. We have made an "educated guess" by extrapolating from the RIAA sales data, which gives information about the number of yearly released albums, and by assuming that this growth number has been constant over the last 50 years. The size of the music domain is in table 4.2.

Database	Artists	Albums	Tracks
Domain	1,350,000	4,500,000	54,000,000

Table 4.2: *Music domain statistics*

Figure 4.1 shows that by far, most artists in the MusicBrainz data base have only a single album: the *median* of the number of albums per artist is 1. However, the *mean* number of albums per artist (the number of albums divided by the number of artists which have at least a single album) is around three.

If we regard the *weighted average* of the number of albums, where the weighting factor is proportional to the "popularity" of the artist (see below), the average number of albums is 28. This high number is probably due to the fact that popular artists are represented better in the database than lesser-known artists. (note that an 'album' is everything from an official album to a bootleg and an inclusion on a compilation album).

The weighing coefficients for this weighted average are calculated from the new AudioScrobbler data set: the weighting factor of an artist x is its *popularity*: the number of *(profile, artist)* tuples where artist equals x divided by the total number of *(profile, artist)* tuples:

$$w_x = \frac{|\sigma_{artist=x}(profile, artist)|}{|(profile, artist)|} \tag{4.1}$$

Figures 4.2, 4.1 and 4.3 show respectively the track distribution per artist (how many artists have how many tracks), the album distribution per artist (how many artists have released how many albums), and the track distribution per album (how many albums have how many tracks on them).

The "peak" at 12 tracks in figure 4.2 relates to fact that most albums have 12 tracks and because most artists have one album.

Figure 4.1 shows that there are many artists with few albums.

Note on the limitations of the MusicBrainz schema: every album has exactly one artist, or a special "various artists" marker. Every track occurs on exactly one album. If there are more albums with the same track on it, that track is still counted as multiple tracks in the database. Albums are not only official albums, but also bootlegs, unofficial compilations and CD singles. For popular artists it is therefore not uncommon to have more than 70 "albums".

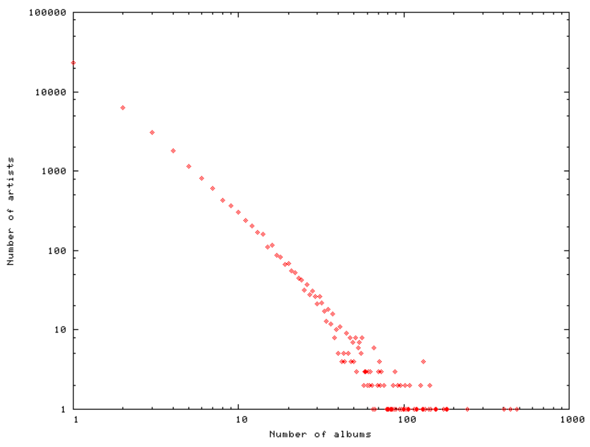


Figure 4.1: *album distribution per artist*

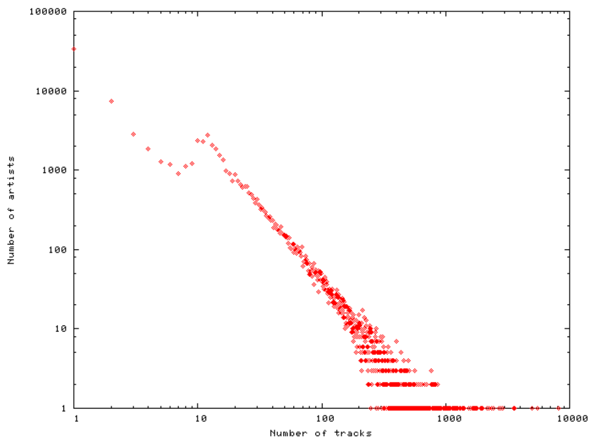


Figure 4.2: *track distribution per artist*

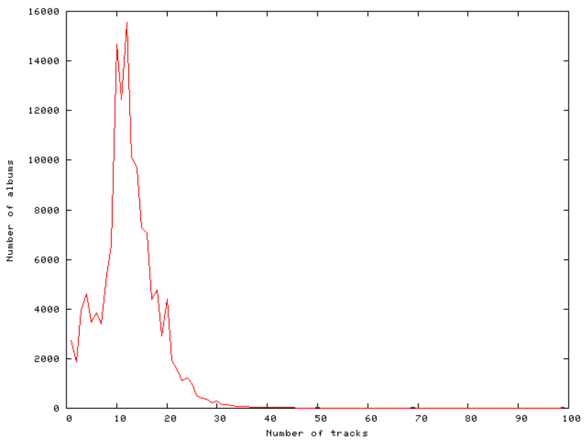


Figure 4.3: *track distribution per album*

4.2.2 User collection sizes

Another thing to model is the collection size of a user. We would like to know what the size of the digital music collection of an average user is. In the “How much information” report, an investigation was done on a 40k node subnet of the Kazaa P2P network, and 1,2M MP3 files were counted. This means that an average user connected to Kazaa has a music collection of 30 tracks. This average also includes users which have no interest in playing digital music, however. If we want to capture the collection size of the “digital music-minded” user, we better can look at the AudioScrobbler data set instead.

The collection sizes for users in the graph of figure 4.4, is obtained by looking at the play-log data of AudioScrobbler (the old schema), under the assumption that a user has played every item in his music collection at least once while the AudioScrobbler plug-in was active. Obviously, we can not say anything about data that has not passed through the system.

The graph plots the size of the collection on the X axis versus the number of users that have a music collection of this size on the Y axis. The picture depicts more or less a power law: the shape of the graph approximates linearity in log(x),log(y). This means that most of the users have a very small collection.

This data set also shows that the average user has played 202 tracks, so we can conclude that the average collection has at least 202 items. You might think an average users’ collection is still larger than that. The dataset used for this experimentation might just not contain all the data we are looking for. Users which have only tried AudioScrobbler once are also taken into account: hence the logarithmic nature of the collection sizes graph. To make a better estimate, we would have to filter out all non-regular users and only look at the users who use AudioScrobbler on a very regular (say, three-daily) basis. This remains a point for future research.

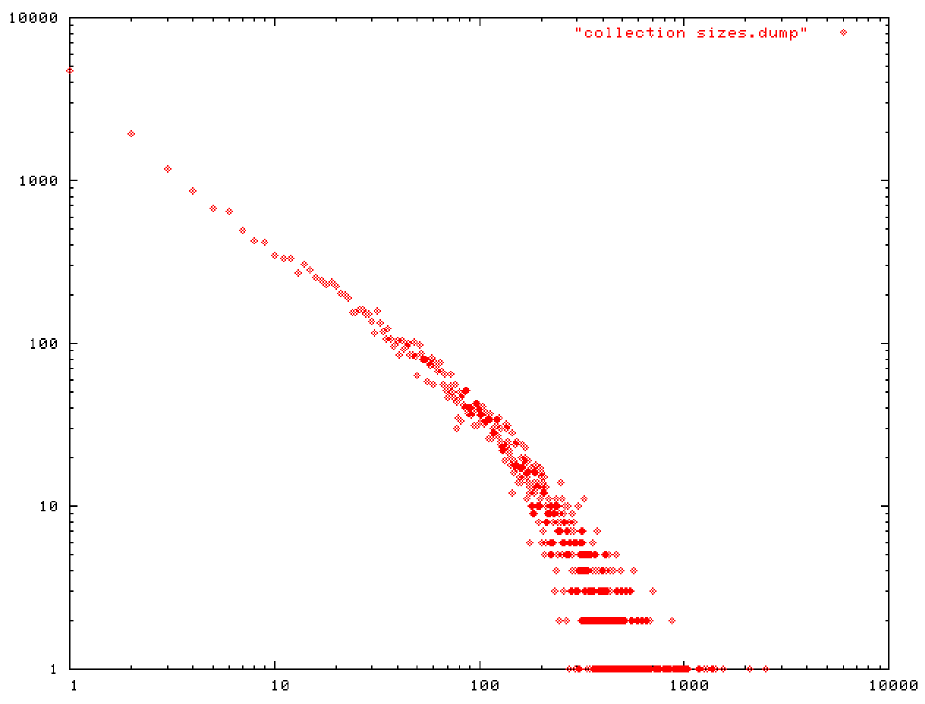


Figure 4.4: *User collection sizes*

4.2.3 User collection overlap

Users often have overlapping musical tastes. We have performed an experiment in which we plotted a percentage of the user base against the percentage of the distinct tracks (with respect to all tracks in the database) this percentage of users had in their collection. This information was derived from the new AudioScrobbler dataset, which used MusicBrainz identifiers to identify the tracks.

If every person had different tracks, this would be a straight line from $(0, 0)$ to $(1, 1)$. However, as users often have the same tracks in their collection (so-called popular music), this curve is expected to bend. The collection overlap graph is shown in figure 4.5. The curve indicates that there is not much overlap in the musical collection of users. If there was, the curve would be more bended towards the point $(0, 1)$. People tend to have the same popular music, but have also many less-popular items. We will use this result later on in the Server Architectures chapter when we look at distributed systems.

Figure 4.6 shows the popularity distribution of artists, with some annotated artist names for clarity. The popularity distribution is a ranked list which was established by sorting the list of all artists on the number of distinct users who has played that artist. This curve is not a straight log/log line, but is bent outwards. This indicates that there are many artists which are “reasonably” popular.

4.2.4 Data usage

To investigate how music data is typically used, we want to obtain knowledge about the nature and frequencies of the typical queries. Music databases are setup with a purpose in mind. For example, the primary purpose of the FreeDB database is to aid the identification of music CD’s. AllMusicGuide.com’s primary goal is to give information about music to assist in buying it.

MusicBrainz data usage

The primary purpose of the MusicBrainz database is to provide a means to uniquely identify music files and to enrich them with clean metadata from the database. Therefore, the “tagging” action is expected to contain the most frequent queries. As seen in the introduction in section 3.2.2, the most dominant query is the “MusicBrainz identification query”, query 1. Here we will focus on the frequency of this query per day and per

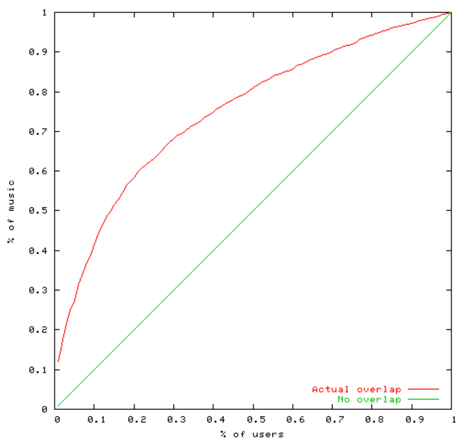


Figure 4.5: *Music collection overlap*

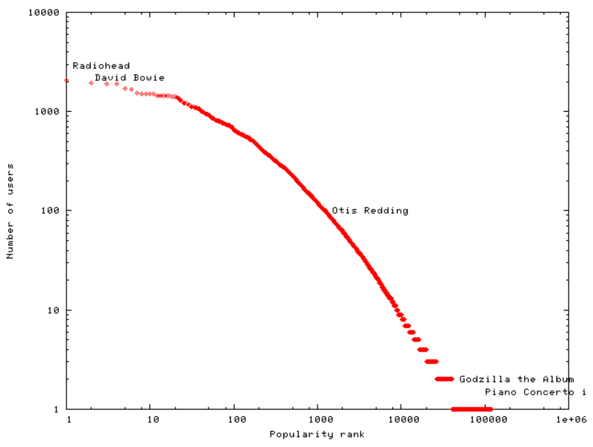


Figure 4.6: *Artist popularity is almost log/log*

user. The data set which can indicate this is the log file of the TRM identification requests.

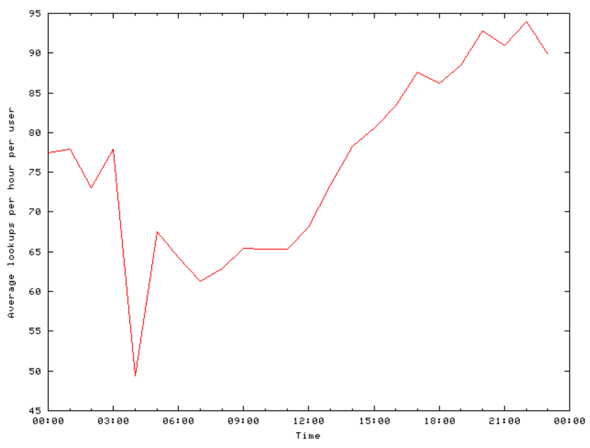


Figure 4.7: *Average MusicBrainz lookups per user per hour*

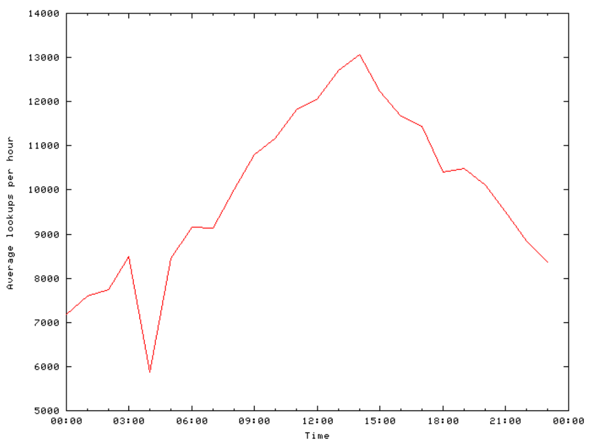


Figure 4.8: *Average total hourly MusicBrainz lookups*

Figure 4.7 shows the average *lookup rate* per user: how many lookups does the average user do in an hour. Figure 4.8 shows the average distribution of all lookups per day on an hourly basis. The average total number of lookups per day is the integral of this graph, which is 238k. An interesting finding is also that per day, 5% of the registered MusicBrainz users use the tagger (to tag at least one song). This was measured by dividing the average number of unique users by the average total number of registered users for the month April 2004.

Besides the "tagging" query, the other dominant request is the "submit" query: query 2 in the introduction. However, we do not have information on how often this query occurs. We can find an upper bound on the number of submits by assuming that every "miss" in the database is manually corrected and submitted: the number of submits per day then would be $(1 - H(t)) \times 238k = 64k$. In the server architectures chapter we will use this upperbound to estimate the performance of the master/slave model.

AudioScrobbler data usage

The goal of AudioScrobbler is to build a profile for the musical taste of every user, and using some sort of collaborative filtering technique to generate recommendations: a play-list with songs the user might like.

The most dominant load for the database is the update load, since the play-logs of every AudioScrobbler user are stored in the central database, and play-log entries get submitted every time a song is played by a user.

Apart from this load, there are scheduled processes which run nightly to rebuild the charts of most popular artists and tracks. The global artist-similarity matrix is rebuilt periodically as well. From this matrix, recommendations are generated for every user on a very infrequent basis. Furthermore, there are simple lookups done through the website (last X tracks played, and most popular tracks/artists for users).

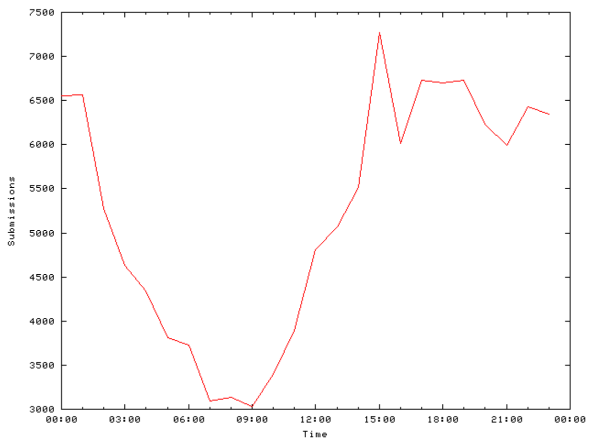


Figure 4.9: *Average hourly AudioScrobbler submissions*

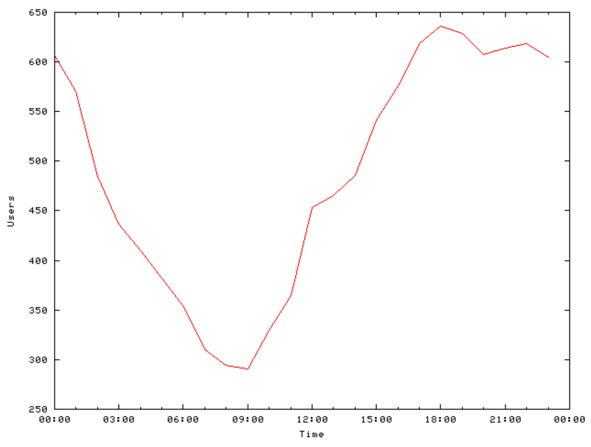


Figure 4.10: *Average hourly AudioScrobbler submissions per user*

To make the update load more insightful, the average number of active users of the system are plotted in figure 4.10. This graph shows the same trend as figure 4.9, which displays the *total* load of the system per hour (in the GMT-6 timezone). It shows that during the afternoon/evening hours the load is high, while in the morning it is low. This is due to the fact that most AudioScrobbler users are located in the United States.

Both graphs show the same trend because the average number of songs a users plays in an hour (and thus submits) is constant: about 11 songs per hour.

4.2.5 Data growth

Domain growth

According to the “How much information? 2003” report, every year there are about 90,000 new audio CD titles (albums) released in the market. Given that the average album has 12.4 tracks (derived from the MusicBrainz database), this amounts to 1.1 million new tracks per year.

Numbers on how many distinct artists are among these new releases and how many “new” artists pop up every year (given that an artist is only an artist if he has released at least one album), are unavailable, unfortunately.

User base growth

In figure 4.11, the number of unique users for the AudioScrobbler project is plotted against time for the period January - March 2004. The graph was constructed by counting the number of distinct users that have made at least one AudioScrobbler submission per day.[1]

Figure 4.12 shows the number of users of the MusicBrainz tagging program plotted against time for April 2004. The number of MusicBrainz users is obtained by counting the number unique IP addresses that made at least one MusicBrainz lookup (TRM identification request) per day.

The graphs show that both projects have a growing user base.

[1]AudioScrobbler has currently disabled the sign up for new users because they fear the system will not bear a much higher load than it is having now. The AudioScrobbler graph shows linear growth but in a irregular pattern; this is due to the downtime due to system overload the system has been experiencing.

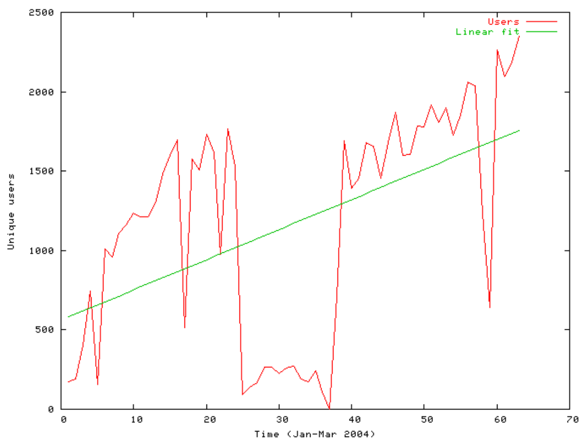


Figure 4.11: *AudioScrobbler Users per day (Jan'04 - mar'04)*

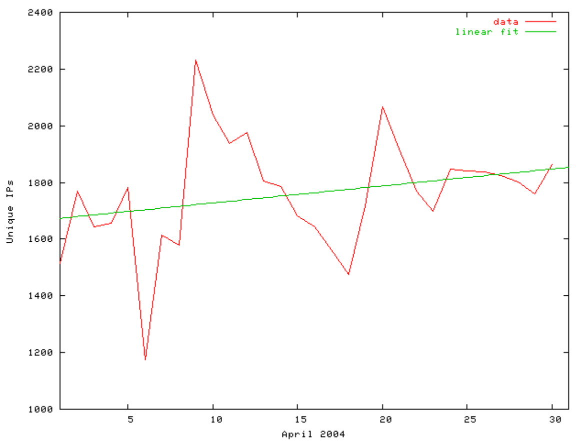


Figure 4.12: *MusicBrainz Users per day (April 2004)*

4.3 An analytical model for music data

As explained in the introduction, the recommender system can be divided in three levels: the identification of the audio files (tagging), the collection of the play-logs (logging), and the generation of play-lists (recommendation). For each of these steps, we propose a model which estimates what the query volume for each of the steps will be, given the number of users using the system.

Before the models can be derived, we first need to make explicit what processes to model, and what parameters are involved.

The first parameter set we need to model is the physical music domain. All domain parameters are denoted with a D-superscript. For simplicity, we only model the number of albums (D^D, discs), the number of tracks on those albums (T^D), and the number of artists (A^D). To be able to model the growth of the system (which is our primary goal), all these parameters are modeled against time.

Like the domain parameters, the database parameters model the number of albums, tracks and artists (D, T and A, respectively) which are available in the music metadata database. The entities with a U-subscript denote the number of albums, tracks or artists *per user* of the system. We also need to take into account the number of users (U) that will use the system at a given time. See section 4.3.2 for the model.

The last set of parameters are the system usage parameters, in section 4.3.3.

The tagging of a track consists of the identification of the track by assigning it a global unique ID. A user has to tag each track in his collection only once: after tagging, the ID is written in the comment field of the textual metadata of the file for quick reference.

The number of tag actions over all users of the system that are executed per day is called the tag frequency: F_{TAG}.

The number of (TRM, track id) pairs that get submitted per day, at a specific number of users, is called the "submit frequency", F_{SUBMIT}.

The *hit ratio*, $H(t)$, is the ratio at which misses and hits occur in the tagger subsystem. It is an indication of how many of the total identification requests result in a "hit" (see section 3.2.2).

While we focus mostly on the music identification problem, in making the models we also look at logging and the recommendation aspect of the system. The frequency at which users play music songs is modeled as F_{PLAY}; the frequency at which users want to get recommendations from the system through an auto-generated play list as $F_{GENERATE}$.

In table 4.3, the three groups of parameters are summarized.Note that the time, t, is always measured in days, and the point $t = 0$ is fixed at January 1st, 2003.

	Domain parameters
$T^D(t)$	The track domain: the set of all existing music tracks at time t
$A^D(t)$	The artist domain: the set of all existing artists at time t
$D^D(t)$	The disc domain: the set of all existing discs (albums) at time t
	Database parameters
$T_u(t)$	The set of tracks of a user u, at time t.
$A_u(t)$	The set of artists of a user u, at time t.
$D_u(t)$	The set of albums (discs) of a user u, at time t.
R_u	The point in time on which user u registers with the system
$U(t)$	The set of all users at time t.
$T(t)$	The set of all tracks in the system at time t. This is the multi-union of $T_u(t)$ for every user u at time t.
$A(t)$	Similar to $T(t)$, for artists
$D(t)$	Similar to $T(t)$, for discs
	System usage parameters
$F_{TAG}(U)$	The number of tag actions per day at a specific number of users.
$F_{LOOKUP}(U)$	The number of submit actions per day at a specific number of users.
$H(t)$	The hit ratio over time; eg. the number of hits over the total number of tags at a point in time.
$F_{PLAY}(U)$	The number of requests for logging the “play song” event at a given user base
$F_{GENERATE}(U)$	The number of requests for generating a play-list at a given user base

Table 4.3: *Terminology*

4.3.1 Domain parameters

We now model the growth of the music domain in time. All figures mentioned are derived from the MusicBrainz database. Furthermore, we assume that the music domain has a linear growth.

From the RIAA data, we know that every year, about 90000 new albums entered the market worldwide in 2003, which is about 246 per day. If we extrapolate this, and assume that albums have been entering the market for the last 50 years, this gives us the following $A^D(t)$ function:

$$D^D(t) = 246t + 4500000 \tag{4.2}$$

An average music album consists of 12 tracks. This average is a constant over time. Therefore, the number of tracks is simply $12D(t)$:

$$T^D(t) = 12A^D(t) = 2952t + 54000000 \tag{4.3}$$

On average, an artist has published 3.25 albums , therefore the number of albums is simply $\frac{1}{3.25}D^D(t)$:

$$A^D(t) = \frac{1}{3.25}D^D(t) = 75t + 1350000 \tag{4.4}$$

4.3.2 Database parameters

A, T and D will converge to A^D, T^D and D^D, respectively, when the database size grows equal to the domain size. Before that time, however, these functions are different, having steeper angles, since the users of the system catch up with the domain.

From the MusicBrainz logs, we can model $U(t)$, $A(t)$, $T(t)$ and $D(t)$ as functions of time. From the graph, it shows that the three lines approximate exponential functions (they are linear on a logarithmic scale).

$$\begin{aligned} log(T(t)) &= 1.905 * 10^{-3} t + 13.487 \\ T(t) &= 720.211 * 10^3 e^{1.905*10^{-3} t} \end{aligned} \tag{4.5}$$

We do the same for $A(t)$:

$$\begin{aligned} log(A(t)) &= 1.608 * 10^{-3} t + 10.627 \\ A(t) &= 41.249 * 10^3 e^{1.608*10^{-3} t} \end{aligned} \tag{4.6}$$

And for $D(t)$:

$$\begin{aligned} log(D(t) &= 1.896 * 10^{-3} t + 10.972 \\ D(t) &= 58.238 * 10^3 e^{1.896*10^{-3} t} \end{aligned} \tag{4.7}$$

With the current growth, the system reaches the "stable state" at $t = 2329$, with respect to tracks, at $t = 2329$, which is somewhere in 2009 (because $t = 0$ is stated at January 1st, 2003, and t is in days):

$$\begin{aligned} T^D(t) &= T(t) \\ 2952t + 54000000 &= 720211e^{1.905*10^{-3} t} \\ 720211e^{1.905*10^{-3} t} - 2952t - 54000000 &= 0 \\ t &= 2329.16 \end{aligned}$$

However, it can be expected that this state will be reached at a later point in time, as the number of old releases entered in the system gradually decreases with time, because there are many "rare" releases.

The number of users is also log-linear:

$$\begin{aligned} log(U(t)) &= 3.066 * 10^{-3} t + 9.025 \\ U(t) &= 8.307 * 10^3 e^{3.066*10^{-3} t} \end{aligned} \tag{4.8}$$

4.3.3 System usage parameters

A rough estimate of $F_{TAG}(t)$ can be obtained from the MusicBrainz logs. We have the number of users in a certain time frame, and we have the number of submissions in a certain time frame.

We know that every item in a users collection is only tagged once[2]. If we assume that by far the largest part of a user's collection is tagged *directly after registration* with MusicBrainz, we can assume that there is a linear relation between the number of users and the number of submissions. We can estimate F_{TAG} simply by taking the average of the submissions per day divided by the number of users of the system on that day. This is a rough estimate, an upperbound on the number of tag actions per day given the number of users.

$$F_{TAG}(U) = 0.16|U| \tag{4.9}$$

If any user would tag only one song per day, then 16% of the total users use the system each day. Of course, the actual number of users per day is much smaller as a user typically tags more than one song. In fact, section 4.2.4 estimates the active MusicBrainz users per day as 5% of the total user base. Note that the tagging frequency also depends on the number of new music entering the collections of (already registered) users. We leave this factor out, since we have not enough data to model the growth of a user's collection.

[2]This is due to the design of the tagger: if the program is given the same file twice, the second time it will recognize that the file was already identified previously

$F_{SUBMIT}(U)$ can be estimated if we know the hit ratio: at most, a submit is done for every miss, if every user identifies each unidentified track and submits the result. In that case, $F_{SUBMIT}(U)$ becomes:

$$\begin{aligned} F_{SUBMIT}(U) &= (1 - H(t))F_{TAG}(U) \\ &= (1 - 0.73)0.16|U| \\ &= 0.0432|U| \end{aligned} \tag{4.10}$$

$F_{PLAY}(U)$ can be estimated in the following way: the average user listens to music every day for one hour. Since the average song length is a little over 5 minutes, about 11 songs can be played in an hour. The frequency for the F_{PLAY} event then simply becomes:

$$F_{PLAY}(U) = 11|U| \tag{4.11}$$

We expect a user to click the “generate” button once a day. Therefore $F_{GENERATE}(U)$ becomes:

$$F_{GENERATE}(U) = |U| \tag{4.12}$$

The last parameter is $H(t)$: the hit ratio. We assume this ratio to be constant valued. The value is deduced from the MusicBrainz hit/miss log files as 0.73.

$$H(t) = 0.73 \tag{4.13}$$

As the database grows, we expect this ratio to drop. When the project is in its startup phase, the database grows fast, because users all put different music in the database. Since there is so much music available, there is a lot of “catching up” to do. The miss ratio will decrease proportional to the size of the database.

Once the project has caught up, the growth curve will flatten. In this phase, the database growth will be proportional to the number of new music releases. In this phase, the miss ratio will not decrease further because new music is released on a constant basis. The ratio will always be higher than zero because there will always be queries about music that is not already in the database.

Due to the “flash crowd effect”, this ratio might be higher than one would expect, because “hot” information will be asked for more than information longer in the database, due to popularity of new releases. But, as soon as these new releases are added in the database, they do not cause any more misses, thus lowering the ratio.

4.4 Conclusion

The survey of the principal entities in a music database provided us with an “educated guess” regarding the size of the music domain. The usage and growth of music databases could be estimated by using various data sets. Regarding the user collection, different data sets yielded different results when trying to estimate the user collection size. More work is needed on this subject to provide a better estimate. Regarding collection overlap, we concluded that, even though there is an elite of “very popular artists”, most users have only a few overlapping items in their music collection.

We concluded that in the last section we have successfully established an analytical model of the music domain, modeling the growth of the domain both in terms of time and in the terms of the size of the user base size of a music identification system. The chapter about server architectures will use this model to shed light on scalability issues.

Chapter 5

Music data cleaning

5.1 Introduction

Song metadata has an inherent fuzzy nature, due to spelling errors, incomplete data or compression. While for a human it is immediately clear that two songs are alike, for a computer it is not. This chapter explores different database-based methods for cleaning and fuzzy lookup of textual music metadata through the use of the *similarity join* and the *similarity select*.

Section 5.2 first describes various similarity metrics and preprocessing techniques. Next, before we actually attempt any cleaning, section 5.3 investigates the cleanness of two of the data sets, comparing them against manually sampled ground truth. Section 5.4 explains how the MusicBrainz project manages to keep its database clean. Next, section 5.5 surveys different similarity joins and explains how this type of join facilitates database cleaning. Section 5.6 describes and evaluates variants of the similarity select and describes how it can be integrated into the MusicBrainz project. Finally, section 5.7 sums it all up and draws conclusions from the findings in this chapter.

5.2 String similarity

As this chapter mostly deals with the cleanness of *textual* metadata, the need for estimating the similarity between strings arises. String similarity is a wide area of research, covering a range of fields, from molecular biology to information retrieval. This section reports on three metrics: the edit distance, the normalized edit distance and cosine similarity.

5.2.1 Similarity metrics

The *Levenshtein distance* [33], named after the inventor, is a measure of similarity between two strings *A* and *B*, which represents the minimum number of *edit operations* needed to get from *A* to *B*. An edit operation is a deletion, insertion or a replacement of a character. For example, the Levenshtein distance for the strings `Beatles` and `Bangles` is 3 (starting from Beatles, we need to remove the first e, replace the t with a n, and insert the g after the n, which are three operations). The Levenshtein distance is often referred to as the edit distance, or the *regular* edit distance.

An extended version of the edit distance, called the *Levenshtein with transposition*, defines the swapping of two adjacent characters as a single operation, instead of two operations (insert, delete). This tends to capture this typical spelling error better, because swapping two cahracters (like we just did) is a common mistake.

The edit distance measure suffers from the "short string problem": the fact that the distance between almost-similar *short* strings is relatively higher (with respect to string length) than between almost-similar long strings. The edit distances between "abc" and "xyz", and between "Hello there, everybody" and "Hello there every-buddy" are both 3, while one intuitively thinks the first two strings differ more than the last two.

Normalized edit distance

This short string problem calls for an unified similarity measure, independent of string length and preferably with a fixed range. The *normalized edit distance* is such a length-normalized version of the edit distance.

There are many variants known in the field for calculating a normalized edit distance [37, 5, 43]. The one we will use in this paper is the method as utilized in the GNU `diff` program, and is based on the algorithm by Myers [43]:

$$\frac{characters\ in\ common}{average\ length\ of\ strings}$$

The normalized edit distance yields a number between 0 and 1, where 1 indicates that the strings are similar, and 0 that they are total dissimilar. The documentation points out that this method is "admittedly biased towards finding that the strings are similar, however it does produce meaningful results". The reason for choosing this particular algorithm was the availability of the source code [20].

Linear edit distance

While the negative effect of the Levenshtein edit distance measure is the "short string" problem, the algorithm does have nice properties to integrate it with a Q-gram join algorithm, As we will see later, there is a direct relation between the edit distance of two strings and the number of shared q-grams.

As an attempt to deal with the "short string" problem, while at the same time keeping these properties intact (which standard normalized edit distance algorithms do not), we propose the *Linear Edit Distance*.

The Linear Edit Distance (LED) is a boolean predicate returning true if two strings match, and false otherwise. The formula for the predicate is the following linear equation:

$$LED(a,b) = D(a,b) \leq k + c \times min(|a|,|b|) \tag{5.1}$$

The formula implies that strings a and b are considered similar if they fall within the boundary. k is the minimum edit distance measure (at minimum string length 0!) and c is the steepness of the slope. Figure 5.1 visualizes the minimum string length boundary for $k = 0.25$ and $c = 0.25$.

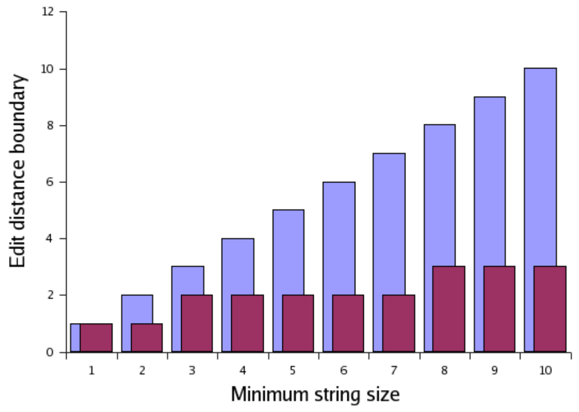


Figure 5.1: *A linear edit distance boundary*

Cosine similarity

The Cosine Similarity function[1], used in Information Retrieval to measure the similarity between two documents, can be used for string similarity as well. First, both strings need to be split up in tokens, which can be letters, words or Q-grams (substrings of the string of length q).

[1]Gravano [23] reports on successfully implementing this method in a database.

Then, the tokens are assigned weights, based on the frequency in the string (TF, term frequency), and their frequency in the whole collection (IDF, inverse document frequency). The similarity between the tokens of the string is calculated by taking the sum of the multiplications of the weights of co-occurring tokens.

String preprocessing

Before doing similarity calculations, the strings can be preprocessed, to account for, for instance, phonetic spelling errors.

The simplest implementation of a phonetic representation is the `soundex` function [53], which is very simple but not so powerful. It was originally intended for use in surname indexes, as it captures how names sound rather than how they are spelled. However, this method looks only at the first four "significant" consonants: "Washington" is coded identical as "Washington city" (W-252). Also, it assumes the first word character is spelled correctly. The soundex function is implemented in many database systems, probably due to its simplicity rather than its functionality.

Another technique is the so-called "letter-to-sound" algorithm[10], which transforms words in their possible pronunciation. Commonly used in text-to-speech systems [19, 18], it utilizes an extensive state-machine, with thousands of (automatically learned) rules. The disadvantage of the letter-to-sound algorithms is that the state machine used is dependent on the language which is used. This implies that the language of the string must be known before parsing can begin. Guessing language might be hard since typical metadata strings (eg. artist names) are short.

Other than a small experiment with using soundex (which we do not report on) we have not experimented with any of the string preprocessing techniques due to lack of time.

5.3 Data cleanness

For a general impression of the cleanness of music data, we look at the artist names in the raw play logs obtained from the AudioScrobbler old data set, and the artist names from the MusicBrainz database. We compare the cleanness of both data sets. Our hypothesis is that the MusicBrainz database will be much cleaner, as it is actively moderated by humans.

For every data set and for every similarity predicate, a sampling of 50 random artist names was done, and for each artist, the number of artist names which fall within the similarity predicate was counted. Such a group of artists is referred to as an *artist cluster*.

The following similarity predicates were used:

1. **plain edit distance**, the Levenshtein distance between two strings, with a transposition cost of 1. For the similarity boundary, $k \leq 3$ was used.
2. **normalized edit distance**, a string-length normalized string similarity measure. Similarity boundaries of 0.75 and 0.8 were used.
3. **linear edit distance**, the edit distance measure which has a linear relation with the minimum length of the two strings. Two sets of parameters were used: $k = 0.25$, $c = 0.25$, and $k = 0.25$, $c = 0.35$.
4. **ground truth** - ground truth was provided by manually browsing through the database for each sample artist, with a "common sense" clustering criteria.

In the results table, in table 5.1, the mean, standard deviation and median are reported of the resulting cluster sizes.

The unnormalized edit distance measure tends to be more prone to noise: the cluster sizes for small strings become very large. This is inherent in the fact that the measure is unnormalized: For instance, the string "ABC" has an edit distance of only 3 to the string "XYZ". Many artist names are short so the effect of this is clear (see the mean cluster size of the plain edit distance). However, the median cluster sizes of the unnormalized edit distance are still small, indicating that many small clusters are found, which is good.

The normalized clustering with boundary $k \geq 0.8$ performs well on the AudioScrobbler data set: it is close to the ground truth. However, the same setting tends overfit on the MusicBrainz data set: with this setting the clusters are larger than the ground truth.

Dataset	Method	Mean	Stdev	Median
AudioScrobbler	normalized ($k \geq 0.75$)	8,5	22,9	2
	normalized ($k \geq 0.8$)	5,0	13,2	2
	plain ($k \leq 3$)	60,9	220	3
	linear ($k = .25, c = .25$)	2,4	3,4	1
	linear ($k = .25, c = .35$)	5,1	13,1	2
	ground truth	5,2	11,5	1
MusicBrainz	normalized ($k \geq 0.75$)	4,2	10,7	2
	normalized ($k \geq 0.8$)	2,3	3,3	1,5
	plain ($k \leq 3$)	35,8	123	1
	linear ($k = .25, c = .25$)	1,6	1,2	1
	linear ($k = .25, c = .35$)	2,3	3,4	1,5
	ground truth	1,1	0,4	1

Table 5.1: *Artist name cleanness*

The linear edit distance with $k = .25, c = .25$ performs reasonably well: the values on the MusicBrainz data set are the closest to the ground truth, while on the AudioScrobbler the method tends to under-fit: the cluster sizes are too small. The second parameter set clearly does better on the AudioScrobbler set, even having the best performance of all methods on this set, but this variant over-fits on the MusicBrainz set.

We have seen that automated clustering depends much on the chosen boundary. Even estimating the boundary by taking a clean artist database and estimating the average distance between all artists would lead to under- and overfitting at the same time, since the boundary is an average. We can only conclude that automated clustering will always make mistakes. A too-small boundary in one case leads to a too-large boundary in the other, and vice versa. The boundary can be said to be dependent on the used data set, since it depends on how clean the data is.

Artist	MusicBrainz	AudioScrobbler
Chet Atkins	1	3
The Beatles	1	41
Jane Krakowski	1	1
Chapel Of Rest	1	1
Sick of It All	1	6
Fifth Column	2	1
Boom Bip & Doseone	1	6
Radiohead	1	65
Austin Leeds	1	2

Table 5.2: *Sample cluster sizes from ground truth for the two data sets*

Table 5.2 shows a sample from the manually clustered artist names for the two data sets. This sample, together with the ground truth means from table 5.1, clearly shows that the MusicBrainz database is more “cleaner”. This was to be expected as MusicBrainz is human-moderated while AudioScrobbler (in the old schema, which did not use MusicBrainz) was not.

This table also shows it can be assumed that the more popular an artist is, the more spelling variants will exist. Given that “metadata noise” is uniform, more-occurring items will be more prone to it. The number of spelling errors appears to be proportional to the frequency of the item.

5.3.1 Data cleanness conclusion

The MusicBrainz database is remarkably clean. On the manual sampling of this database, almost no “duplicate” artists were found. Never was the cluster size bigger than 2. How MusicBrainz does this job of maintaining cleanness of data is explained next, in section 5.4. The AudioScrobbler data set is not clean at all, due to the lack of verification of the input (all submitted metadata make it into the database, no matter how wrong).

Due to the findings in this section, we have chosen to use the AudioScrobbler data set for testing the cleaning methods presented later in this chapter.

Automated clustering quality depends on the chosen boundary. We conclude that this boundary is dataset-dependent. Our proposed "Linear Edit Distance" measure performs as good as the normalized edit distance.

5.4 How MusicBrainz keeps its data clean

The MusicBrainz project provides a solution to the music identification problem in the form of a centralized database maintained by human moderators.

Its tagger program tries to automatically identify and label songs from the collection of a user. As explained in the introduction (section 3.2.2), for every music file, a TRM hash is generated by using Relatable's proprietary audio fingerprinting web service. On the client side, specific properties of the audio track are calculated, like the song length, number of zero crossings, Haar coefficients and spectral coefficients from the Fourier Transform. These numbers are sent to the TRM server, the "black box", which returns an integer with the TRM id.

Query 3: *The join query to find track identifiers*

```
SELECT track.gid
FROM trm, trmjoin, track
WHERE trm.trm = <<trm>> AND
      trmjoin.trm = trm.id AND
      trmjoin.track = track.id
```

The MusicBrainz database is then queried for tracks corresponding to this TRM hash, by joining the `trm`, `trmjoin` and `track` relations (see query 3), in the hope to find a single track identifier. Depending on the number of results of this query, the following action is taken.

1. **One track found**: An exact match for a song is found; meaning that the song was already in the database and only had a single TRM id associated with it. The song is appropriately re-tagged with MusicBrainz identifiers (track id, artist id, album id). This happens in 84.4% of the cases [41], given that the TRM id already exists in the database.

2. **Multiple tracks found**: In MusicBrainz terminology, this case is dubbed a "TRM Collision"; the TRM id identifiers multiple tracks. MusicBrainz tries to find the correct track by calculating a similarity value between the current track's metadata and the metadata of the stored tracks in a server-side Perl script. This is explained below.

3. **No tracks found**: *This does not necessarily mean that the track does not exist already in the database!* It can be the case that the track exists, but that the newly generated TRM hash is different from the TRMs for the track stored in the database. If no tracks are found, tracks are looked up by using the metadata from the source track, only using a *direct* lookup. This direct lookup will probably not return the best results to the user, because the metadata used in the lookup is dirty. We dub this problem the "direct lookup problem".

   Next, the result list from the direct lookup is ranked using a server-side Perl script (see below), and the user is provided with this list of matching tracks of which he can pick the right one. If the right track is not in the list the user can browse the MusicBrainz website to look it up, or enter the track details manually. The responsibility of finding the right track in this case is given to the user entering the track. The moderation and voting system is used to guarantee the cleanness of the data.

Calculating the similarity between metadata from a single song and a list of song metadata is done in MusicBrainz in the Perl source code, after all results have been fetched by the database. By looking at the artist name, album name, track name, number of the track on the original album, and track duration, a total metadata similarity (a number between 0 and 1) is calculated as a weighted sum of the individual similarities between these properties.

Similarities between the string properties (artist, track, album) is calculated by using the `fstrcmp` function (the number of string edits, normalized on the string length between 0 and 1). Similarity of the track number is just 1 if they are the same and 0 if they are not. The similarity of track durations is $1 - \frac{|dur_A - dur_B|}{30000}$ for $|dur_A - dur_B| \leq 30000$, or 0 otherwise.

Above similarities are weighted: some similarities count more than others. If metadata properties are missing, the weights for the other properties are adjusted accordingly so they still sum to 1.

5.4.1 What's next

The moderated data management solution is a nice solution for guaranteed data cleanness. It is questionable however if this moderated solution will scale, because the moderator base must grow with the same rate as the music data otherwise they will not be able to keep up.

To see if an automated system can overcome this scalability problem, we will look at the *Similarity Join*, a database-based automated cleaning method, next, in section 5.5. We will mostly be investigating its scalability properties.

To overcome the "direct lookup problem", we will investigate the performance of the so-called *similarity select* on two database platforms in section 5.6.

5.5 The similarity join

A similarity join is a special kind of theta-join, joining on a similarity predicate: a function on two non-key attributes of the two relations which must satisfy some condition.

Such a similarity join would ideally provide an efficient way to cluster "similar" artists which reside in an *artist(id, name)* relation, by performing the similarity *self-join* on the relation. The similarity predicate might be that the edit distance between any artist must be less or equal to some constant k, or that the normalized similarity be greater than some probability p.

The join creates a relation *artistSim(a, b, s)*, representing the similarity matrix between any two items in *artist*. a and b denote the keys of the artist-relation self-join, and s the similarity of (the artist names of) these two. This similarity is given by a similarity function, in our case the edit distance. As said, since we are only interested in items with a high similarity, we only store those tuples for which $s \leq k$.

Similarity functions can be implemented in any modern DBMS by using a User-Defined Function (UDF).

The similarity predicate we use in this section is the Linear Edit Distance. Since the Q-gram joining algorithm is dependent on algorithmic properties of the (regular) edit distance measure, we cannot use the normalized edit distance measure. We prefer this measure over the regular edit distance because the latter is dependent on the string length.

5.5.1 The Naive way

The Naive way of implementing this in SQL is by calculating the Cartesian product of the artist table with itself and apply the UDF on the n^2 tuples afterwards, selecting only those withing $sim \leq k$. Given that the similarity function is a symmetric function and we are not interested in the similarity between identical items, we can reduce the Cartesian product size to $(n^2 - n)/2$ by adding a greater-than restriction on the join keys.

The implementation of this is given in query 4.

Query 4: *The naive similarity join*

```
INSERT INTO artistSim
SELECT a.id, b.id, editdistance(a.name, b.name)
FROM artist as a,
     artist as b
WHERE a.id > b.id AND
      editdistance(a.name, b.name) <=
      (k + c * MIN(LENGTH(a.name), LENGTH(b.name)))
```

This naive similarity join is very expensive. Using `set explain='plan';` in MonetDB/SQL we can analyze the commands MonetDB will execute. As expected, the optimizer can do nothing but a full nested-loop join using the '>' operator, multiplex a `editdistance` call on this result, and then filter out the tuples which do not comply to the boundary.

5.5.2 The Q-gram similarity join

In the paper of Gravano [34], a method is proposed to speed up the similarity join, based on the notion that if a string *a* will be within edit distance of *b*, some properties on the *Q-grams* of the strings must hold.

Q-grams are substrings of length *Q* of a string, pre- and postfixed with special characters to mark the beginning and end of the strings. For example, the collection of 3-grams of the string "Beatles" would be {`'##B', '#Be', 'Bea', 'eat', 'atl', 'tle', 'les', 'es$', 's$$'`}. The artist relation gets decomposed in a *artistName-Qgram(id,pos,qgram)* relation, and joins on this table are performed by using three types of filtering.

The three filtering techniques are based on the following observations: *length filtering* is based on the fact that if $|a.x| - |b.y| \leq k$, the edit distance can never be smaller than *k*. *count filtering* takes into account the number of similar Q-grams, and *position filtering* takes into account the relative position between corresponding Q-grams.

This pre-filtering will thus throw out all string combinations of which we know that the edit distance predicate will fail on. The edit distance now only has to be calculated on a much smaller subset of the Cartesian product.

Gravano's method originally used the unnormalized edit distance measure. The drawback of this approach is that this regular edit distance is dependent on the length of the strings: as we have seen, it "favors" short strings. We think that Gravano's method therefore cannot be used in real-world text-retrieval applications.

The pre-filters applied to the join make use of the specific algorithmic properties of the edit distance, so we cannot simply replace the final step of the query, the actual call to the edit distance function with a call to a normalized edit distance function. However, we can put our proposed *Linear Edit Distance* measure to use here since all this measure does is to make the similarity boundary string-length dependent. The final "Q-gram linear edit-distance join" is in query 5.

Query 5: *The Q-gram similarity join*

```
INSERT INTO artist_sim
SELECT   a.id, b.id, a.name, b.name,
         (k+c*MIN(LENGTH(a.name), LENGTH(b.name)) AS boundary

FROM     artist a, artist_name_qgram aq,
         artist b, artist_name_qgram bq

WHERE    -- inner joins
         a.id = aq.id AND
         b.id = bq.id AND
         a.id < b.id AND
         aq.qgram = bq.qgram AND
         -- position filtering
         ABS(aq.pos - bq.pos) <= boundary AND
         -- length filtering
         ABS(LENGTH(a.name)-LENGTH(b.name)) <= boundary

GROUP BY a.id, b.id, a.name, b.name

HAVING   -- count filtering
         COUNT(*) >= LENGTH(a.name)-1-(boundary-1)*3 AND
         COUNT(*) >= LENGTH(b.name)-1-(boundary-1)*3 AND
         -- final, expensive filter step
         editdistance(a.name, b.name) <= boundary;
```

5.5.3 The Q-gram similarity join in MIL

Even though the Q-gram join is a good improvement over the naive join, it can be made even more optimal. The Q-gram join generates for the Q-gram relation a large cross-product before applying position- and length filtering. The quadratic nature of this cross-product will very quickly cause memory problems.

To overcome these problems, we implement a custom joining algorithm on the lower database level using the MonetDB Intermediate Language (see section 3.4). Although we could have used Postgres, we prefered MonetDB because of MIL's ease of use and the availability of support for the database platform within the CWI.

The implemented algorithm performs the position and length filtering while performing the Q-gram join. The actual join resembles the *sort-merge-join* algorithm (see [52], sec. 12.6.5). Looping over the sorted Q-gram ids and sub-sorted Q-gram positions, the algorithm makes a mini-cross-product for each *k*-window within the same Q-gram. The advantage over this algorithm compared to the original SQL-basd query plan is that the full Cartesian expansion of the Q-gram relation with itself is prevented by doing the filtering *online* — while constructing the product.

The `qgramselfjoin` algorithm is listed below, in algorithm 5.1.

```
Require: columns are sorted on $qgram$, and sub-sorted on $pos$
  $qgram$ = the oids of the Q-grams
  $pos$ = the positions of each q-gram within the artist name
  $len$ = the length of each artist name
  $id$ = the artist id
  $n \leftarrow |qgram|$
  while $i < n-1$ do
    $j \leftarrow i+1$
    while $j < n \wedge qgram_i = qgram_j$ do
      if $stringid_i = stringid_j$ then {Only add Q-gram pairs from different strings}
        next $j$ iteration
      end if
      if $abs(len_i - len_j) > k$ then {Length filtering}
        next $j$ iteration
      end if
      if $abs(pos_i - pos_j) > k$ then {Position filtering - no need for next iteration as pos is sub-sorted!}
        break from $j$ loop
      end if
      add $(stringid_i, stringid_j)$ pair to qgram candidates
    end while
    $i \leftarrow i+1$
  end while
```

Algorithm 5.1: *The Q-gram self-join*

`qgramselfjoin` yields a relation with (id, id) pairs of corresponding q-grams of each artist. The post-processing that needs to be done is the count filtering: discarding artist id pairs that have less than *k* q-grams in common. This yields a list with unique artist id pairs. The final step consists of filtering this list with the actual linear edit distance predicate to yield the actual matching artists.

The join was implemented as the custom C function (`qgramselfjoin`) which worked on sorted, aligned columns of row values. Supporting MIL code was written to load the columns from the database, call the `qgramselfjoin` function and lastly performs count filtering and the expensive editdistance step on the resulting pairs. This supporting MIL code is listed in appendix A.

5.5.4 Other similarity joins

In [23], Gravano et al. implement a sample-based cosine-similarity text join inside a DBMS. The two relations are joined by splitting the join attributes into tokens, which then are TF/IDF weighted. The tokens are in this case Q-grams, but they could be anything (words, characters, phonemes). The authors propose a sample-based approach which is based on the notion that for a cosine similarity to be high, at least two weights of the same token must be high too. Therefore the token weight space is sampled with a probability for each token proportional to its weight. They use this sampled relation for the calculation of the cosine similarities. The sampling relies on a parameter S, which specifies that for each token at least S samples must be taken approximately.

As expected, the baseline case does not scale well. At 1000 artists, it takes more than 10 minutes to calculate the query, whereas for 100 artists it takes about 10 seconds.

Another way of data cleaning using joins is described by Ananthakrishna [3], proposing a system exploiting the inherent hierarchical structure of database relations. The duplicate elimination task consists of grouping all rows from the join on the table hierarchy which can be marked duplicate. Once such groups are found, a canonical, representative tuple for each group can be taken as the replacement. Tuples from the relations are compared on the textual similarity (token containment metric), and on the co-similarity of the tuples in their child relation (foreign key containment metric). The decision whether two tuples are duplicates of each other is made by a weighted vote of both of these metrics. For the actual comparing of tuples, TF/IDF weighting is used on tokens of the attributes of the tuple (for textual similarity), and on the whole tuples for child relations.

If we see the AudioScrobbler music database as a two-level hierarchy (artist, track), the foreign key containment metric can be expressed as in query 6. If we assume not much artist overlap can be derived from the track table, the second `normalized_editdistance` call (on the artist-artist relation) is not very expensive because most artists with small overlap are filtered out.

Preliminary tests pointed out that such a join was still very expensive, taking minutes for only a few hundred artist tuples. Exploring this track further and integrating this approach with the token containment metric remains an area for future research.

Query 6: *Matching artist based on similar tracks*

```
  SELECT a.id, b.id, COUNT(*)

FROM     artist a, track ta,
         artist b, track tb

WHERE    a.id < b.id AND
         a.id = ta.artist AND
         b.id = tb.artist AND
         -- assumption: track titles are spelled correctly (!)
         ta.title = tb.title
         -- ... expensive 'naive track join' version would be:
         -- normalized_editdistance(ta.title, tb.title) >= y

GROUP BY a.id, b.id

HAVING   -- prefilter: at least k common track names
         COUNT(*) >= k
         AND normalized_editdistance(a.name, b.name) >= y
;
```

We have not performed enough tests on both of these methods to really say something about their feasibility with respect to database cleaning. However, the article results confirm our gut feeling that the similarity self-join will simply not scale and thus will not be the best tool for the cleaning task.

5.5.5 Similarity join results

We have implemented and evaluated the naive similarity join, the Q-gram join in SQL and the optimized Q-gram join in MIL. All experiments were done on the MonetDB database system, either using the SQL or the MIL (Monet Intermediate Language) front-ends. As parameters to the linear edit distance, $k = 3.0$ and $c = 0.0$ were used. These parameters have no effect on the performance of the algorithms, however.

In figure 5.2, the performance of the three methods are compared. The Q-gram/SQL join is a clear improvement over the naive version. Although the graph of the Q-gram/MIL variant is of the same shape as the other two, indicating quadratic time, it performs much better. At 3000 tuples it is 50 times faster than the naive join, and at 5000 tuples about 30 times faster than Q-gram/SQL.

Although we can optimize the join algorithm as we have seen, using Q-grams and even using a specialized join strategy, the time to calculate the results remains quadratic in the number of input tuples. This is inherent from the fact that a self-join is used: whichever way we optimize or filter, the number of output tuples (and thus, execution time) will remain quadratic with respect to the number of input tuples.

We therefore can only conclude that it is simply not possible to build an efficient, scalable cleaning algorithm based on the naive or Q-gram similarity join.

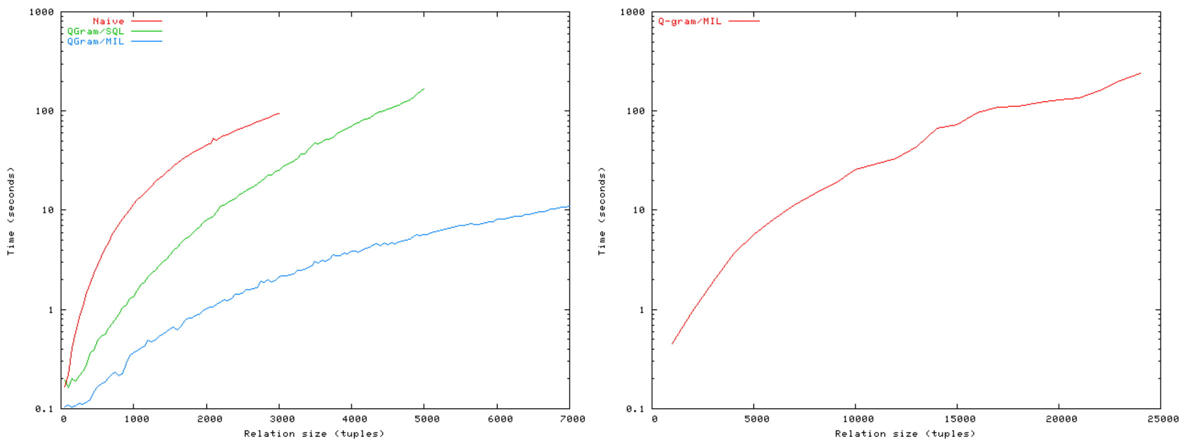


Figure 5.2: *Similarity join performances under increasing relation size*

5.5.6 Clustering the similarity matrix

Even if we assume the similarity join can be performed, we still have to deal with the next problem: what to do with the outcome. In this section we propose a database cleaning method, based on the outcome of the similarity join.

The calculated $artistSim$ relation is in essence a set of graphs, where every tuple in the $artistSim(a, b, s)$ relation describes an edge of one of the weighted, undirected graphs. Every node in the graph is a specific spelling variant of an artist. We assume that each of these graphs is a cluster of possible spellings for a single artist[2]

For each graph (also called a “cluster”) of similar artists, we want a winner (the strongest supported spelling variant) to be chosen. The actual cleaning of the database consists of propagating the identifier of the winner to all spelling variants in the cluster; i.e. every occurrence of the identifier of the artist with the wrong spelling in the database needs to be replaced with the identifier of the winner of the cluster. So the final outcome of the clustering step is an $artistTrans(id_{old}, id_{new})$ relation.

First, we “flatten” every cluster, creating a $cluster(cluster_{id}, artist_{id})$ relation. This step cannot be expressed in standard SQL because of its recursive nature. An iterative variant is given in pseudo-code, in algorithm 5.2.

The final step is the actual cleaning by looking at the calculated clusters and, for each cluster, choosing a winner, for instance the “most supported artist”, the one with the most votes. Another option would be to recalculate for each artist in the cluster the similarity and choose the artist with the lowest average edit distance to every other artist.

This can be done by extracting the “mean” artist like in the iterative example. Choose the artist with the lowest inbetween-edit distance as the ’prototype member’ of the cluster (the ’mean artist name), by dropping all other artists from the same cluster from the artist table.

[2]Note that this working assumption is not very sound: there might be a path in the artist_sim table between different artists, like this (in the case `match_level=1`): `the beatles - the baetles - the baenles - the banles - the bangles`, this is a single cluster, all edges having a edit distance of 1, but it describes different artists.

```
$cluster\_id \leftarrow 0$
while $|artist\_sim| > 0$ do
  nextartist ← SELECT a_id FROM artist_sim LIMIT 1;
  agenda.push(nextartist)
  while a ← agenda.peek() do
    agenda.pop()
    INSERT INTO cluster VALUES (cluster_id, a);
    bid ← SELECT b_id FROM artist_sim WHERE a_id = a UNION SELECT a_id FROM artist_sim WHERE
    b_id = a;
    agenda.push(bid)
    DELETE FROM artist_sim WHERE a_id = a OR b_id = a
  end while
  $cluster\_id \leftarrow cluster\_id + 1$
end while
```

Algorithm 5.2: *The clustering algorithm*

Query 7: *Choosing the winner in a cluster*

```
CREATE TABLE cluster_tmp (
        clusterid INTEGER NOT NULL,
        artistid INTEGER NOT NULL,
        avgdist INTEGER NOT NULL);

INSERT INTO cluster_tmp
SELECT ca.clusterid, ca.artistid, AVG(editdistance(a.name, b.name))
FROM cluster ca, artist a,
     cluster cb, artist b
WHERE ca.artistid = a.id AND
      cb.artistid = b.id AND
      ca.clusterid = cb.clusterid
GROUP BY ca.clusterid, ca.artistid;

CREATE VIEW bestartists AS
        SELECT artistid, clusterid FROM cluster_tmp WHERE avgdist IN (
                SELECT MIN(avgdist) FROM cluster_tmp GROUP BY clusterid);

-- delete from artists table
DELETE FROM artist WHERE id NOT IN (
        SELECT artistid FROM bestartists);

-- create/insert artist_trans table
CREATE TABLE artist_trans (
        old_id INTEGER NOT NULL,
        new_id INTEGER NOT NULL);

INSERT INTO artist_trans
        SELECT c.artistid, b.artistid
        FROM cluster c INNER JOIN bestartists b USING (c.clusterid = b.clusterid)
        WHERE c.artistid <> best.artistid;

-- update tracks
UPDATE track SET artist=new_id WHERE id = (SELECT old_id FROM artist_trans);

-- clean up
DROP TABLE artist_trans;DROP VIEW bestartists;DROP TABLE cluster_tmp;
```

This proposed method is very experimental. In a preliminary test we have concluded that the method does work, but we have not performed any evaluation on the fitness for the task: it was merely the easiest way to express clustering and winner-choosing in SQL.

5.6 The similarity select

Although we have obtained significant performance speedups in the similarity join, we have concluded that cleaning the database using joins remains an open problem. What we *can* do however, is applying the obtained knowledge of string similarities and Q-gram joining to another sub-area of the music matching problem: the metadata lookup in the MusicBrainz tagger program.

As seen in section 5.4, the MusicBrainz server assists the moderator by computing the similarities of artist/track pairs of a *subset* of the total artist/track set, by pre-selecting of artist/track pairs which have an attribute which direct-matches to a field from the song metadata, and presenting this ranked set to the user as a suggestion list, to let the user pick the right song, so that he doesn't have to browse to the song using the website.

This method may fail to select appropriate song suggestions because the fuzzy metadata is used to select suggestions based on direct-match lookups: the "direct lookup problem".

The improvement we propose is that, instead of doing direct lookups to provide suggestions to the moderator, a similarity select on the database level is done, by comparing each artist/track pair from the database with the fuzzy metadata of the song, presenting the top-N in a similarity-ranked list. This similarity select is expected to yield higher quality results, because the similarities between the track metadata and every track in the database is calculated, instead of relying only on a subset obtained through direct match of partial metadata.

Having obtained the track subset, MusicBrainz facilitates a fairly advanced similarity select in which similarities between various metadata properties (artist, album name, track name, duration, track number) are taken together in a weighted sum to compute the final similarity. In the absence of metadata properties the weight of the properties that does exist get altered.

Although it is possible to express this similarity measure in SQL, in this chapter we try to focus on the real issue: testing and optimizing the performance of a similarity select on a single property. For completeness' sake, the real query and accompanying weight matrix which MusicBrainz would use if they port their Perl-similarity select inside the database, is listed in Appendix B.

Performing database optimization on similarity measures on multiple columns at once (as MusicBrainz does, Perl-wise in the most naive way) is a whole area of research. The paper by Fagin [16] is a good exploration of several techniques for the aggregation of such multiple ranked columns. However, it appears that even on a single column the performance of a similarity select is already very low, so we recognize the need to deal with optimizing that first. The implementation of efficiently combining multiple ranked columns in a database will be regarded as future research.

In the rest of this section we will concentrate on the performance of the *single* similarity select: The similarity select which uses a single metadata property, thus selecting from a single column. Three versions are described and evaluated: the *naive*, the *Q-gram* and the *Q-gram/MIL* variants, in the same fashion as the section on similarity joins.

As a running example, we use the artist table (from AudioScrobbler, containing 118847 distinct artist names), doing the similarity select on the artist name. The convention for all three methods is that the track metadata for which suggestions have to be selected is already inserted in the `metadata` table. From this table, we only use the `artist` field here and do the similarity select against the `artist` table, on its `name` attribute.

5.6.1 The naive similarity select

The naive similarity select performs exactly what it says: the similarity select in its simplest form. The query plan is essentially a single scan over the artist relation, with a sort afterwards.

Query 8: *The single similarity select query*

```
SELECT artist.name, editdistance(m.artist, a.name) AS sim,
       (k+c*MIN(LENGTH(a.name), LENGTH(m.artist)) AS boundary
FROM artist a, metadata m
WHERE editdistance(a.name, m.artist) <= boundary
ORDER BY sim DESC;
```

5.6.2 The Q-gram similarity select

The Q-gram similarity select uses Q-gram tables on both the artist name from the database and the artist name from the metadata. Its implementation, in query 9 is very similar to the Q-gram similarity join. As with the latter, we have replaced Gravano's regular edit distance with our Linear Edit Distance similarity measure.

Query 9: *The Q-gram similarity select query*

```
SELECT artist.name, editdistance(a.name, m.artist) AS sim,
       (k+c*MIN(LENGTH(a.name), LENGTH(m.artist)) AS boundary

FROM artist a, artist_qgram aq,
     metadata m, metadata_qgram mq
WHERE
     a.id = aq.id AND
     m.id = mq.id AND
     aq.qgram = mq.qgram AND

     -- position filtering
     ABS(aq.pos - mq.pos) <= boundary AND
     -- length filtering
     ABS(LENGTH(a.name)-LENGTH(m.artist)) <= boundary

GROUP BY a.id, b.id, a.name, m.artist

HAVING -- count filtering
       COUNT(*) >= LENGTH(a.name)-1-(boundary-1)*3 AND
       COUNT(*) >= LENGTH(m.artist)-1-(boundary-1)*3 AND
       -- final, expensive filter step
       editdistance(a.name, m.artist) <= boundary;

ORDER BY sim DESC;
```

5.6.3 The Q-gram similarity select in MIL

As query 9 shares similarity with the Q-gram similarity join, apart from the similarity select not being a self-join, we felt like implementing the Q-gram similarity *select* in MIL as well.

The MIL variant of the Q-gram similarity select works in the same fashion as the `qgramselfjoin` algorith: looping only over the Q-grams of the query (metadata.artist), we build up a histogram relation in (candidate id, q-gram count) pairs, only counting those q-grams which comply with the position and count filtering conditions.

An extra optimization was established for even further pruning of the candidate histogram: the query Q-grams are looped over in reverse order of Q-gram frequency, and new candidates only were added to the histogram in the iterations of the loop that are within the Linear Edit Distance bounds of the query string, because adding "late" candidates will not yield results candidates anyway. For the sake of completeness, the full MIL source code (by P. Boncz) is in query 10.

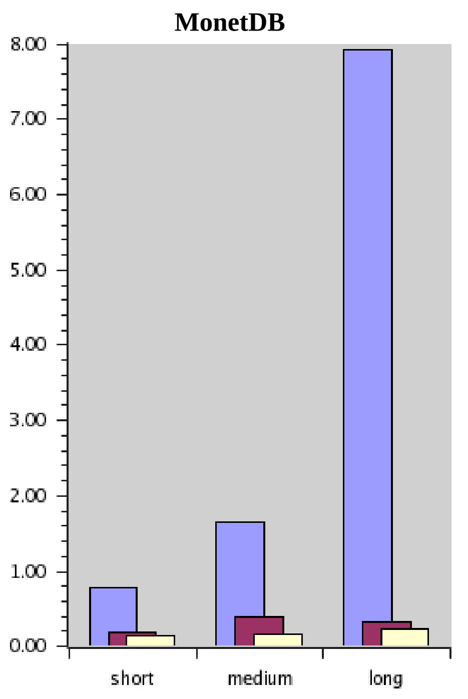
Query 10: *The Q-gram select in MIL*

```
proc qgramselect(bat[void,oid] qgrams_strings,
                 bat[void,sht] qgrams_pos,
                 bat[void,sht] qgrams_len,
                 bat[void,oid] qgram_first,
                 bat[void,str] qgram_val,
                 str s, flt c, int k) : bat[flt,oid]
{
  var query := str2qgrams(s); # generate q-grams for the query string
  var len := sht(query.count());
  var lim := 1 + 3*(k + int(c * flt(len)) - 1);
  var res := bat(oid,int,100000);
  var n := 0;

  # analyze qgrams (rare qgrams first to generate less candidates)
  qgram_val.join(query.reverse()).sort()@batloop() {
      var qgram := int($h);
      var pos := sht($t);
      var lo := qgram_first.find(oid(qgram));
      var hi := oid(int(qgram_first.find(oid(qgram+1)))-1);

      # filter_sel filters on candidates that have the approx. right position
      var filter_len := qgrams_len.reverse().select(lo,hi).reverse();
      var filter_lim := [flt]([min](filter_len,len)).[:*=](c);
      filter_lim := [sht](filter_lim).[:+=](sht(k));
      var filter_sel := qgrams_pos.reverse().select(lo,hi).reverse().
                        [-](pos).[<](filter_lim);

      # refine filter_sel with a test on approximately the right length
      filter_sel.[:and=](filter_len.[:-=](len).[<](filter_lim));

      # test whether we already have info on the candidates that are left
      # after filtering
      var filter_id := filter_sel.[ifthen](qgrams_strings.reverse()
                       .select(lo,hi).reverse());
      filter_sel := filter_id.outerjoin(res);

      # add new id counts to existing candidates
      # (65536=1 because count is shifted 16 bits)
      res.replace(filter_id.reverse()
         .join(filter_sel.select(0,int(nil)).[:+=](65536)));

      if ((n :+= 1) <= lim) {
          # only insert new candidates in first iterations
          # (otherwise they cannot make it anyway)
          filter_sel := filter_sel.uselect(int(nil)).mirror();
          filter_lim := [int](filter_sel.join(filter_lim)).[:+=](65536);
          res.insert(filter_id.reverse().join(filter_lim));
      }
  }

  # perform selection to eliminate candidates that did not cut it
  var res_strings := res.mark(0@0).reverse();
  var res_lim := res.reverse().mark(0@0).reverse().[and](65535);
  var res_hit := res.reverse().mark(0@0).reverse().[>>](16);
  res_strings := [>](res_hit, res_lim).[ifthen](res_strings);
  return res_strings;
}
```

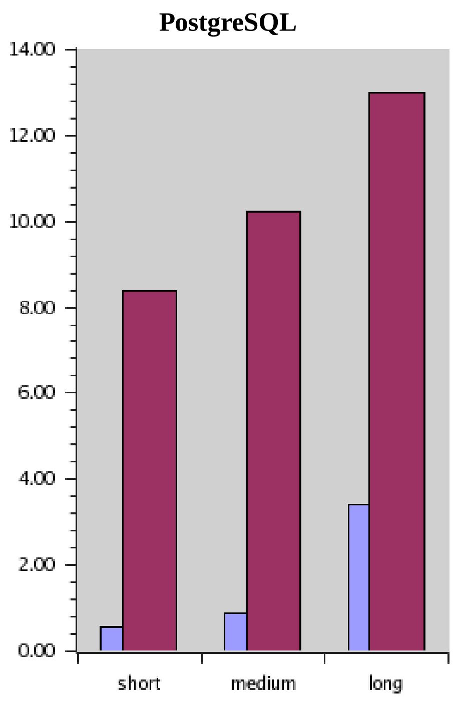
5.6.4 Experiment and evaluation

The quantitative test of the three similarity select methods were performed on a short string (length 3), a medium string (length 20) and a long string (length 100). All experiments were done on the MonetDB database system, either using the SQL or the MIL (Monet Intermediate Language) frontends. The figures in figure 5.3 are the average of 100 runs for each string/method combination.

The results show us that on MonetDB, both the Q-gram select algorithms perform very well compared to the naive variant. The optimizations of the Q-gram select in MIL do not seem to matter very much. The MIL method is faster, but not by a significant amount. This is in contrast with the Q-gram join in MIL, whose



type	short	medium	long
Naive	0.766732	1.634125	7.917407
Q-gram	0.177519	0.380594	0.320267
Q-gram/MIL	0.120078	0.15051	0.229289



type	short	medium	long
Naive	0.560202	0.861131	3.381054
Q-gram	8.368573	10.192538	12.954163

Figure 5.3: *Similarity select results*

performance was an order of magnitude better than the Q-gram join in SQL. The explanation for this is that in the similarity select, there is no large tuple product instantiated in the query, in contrast with the Q-gram join. Thus the memory problems that occurred at Q-gram join/SQL do not occur in the Q-gram select in SQL.

The performance figures on PostgreSQL show another thing, however. The naive variant is faster than the naive on MonetDB, but shows the same behaviour: execution times increase as the string length does. However, the performance of the Q-gram join on PostgreSQL is much worse than the naive variant, even though we created indexes on all joined columns. The generated query plan (investigated using `EXPLAIN ...`) shows no strange behaviour: just three hash joins are used, all on indexed keys, with an aggregation and some filtering on top. Seemingly, the bad performance of PostgreSQL is due to system internals unknown to us.

Concluding, we can say that the Q-gram similarity select opens a perspective into database-oriented textual similarity queries. A hand-optimized algorithm with pruning of unlikely candidates increases performance even more, but not by a significant amount, and therefore we would recommend the SQL-based Q-gram select because of its platform-independence. However, as we have seen in the PostgreSQL results, its performance seems to be very much dependent on the chosen database.

We have not compared the *quality* of this method to the quality of the results the MusicBrainz tagger program returns, as we only have investigated the single-column similarity select, whereas MusicBrainz uses multiple columns to calculate the similarity in Perl. Despite this we expect that the quality of the multiple-column similarity select will at least be as well as the result from the MusicBrainz tagger.

5.7 Conclusion

This chapter has given us insight into data cleanness and the usability of similarity measures, the data cleanness solution of MusicBrainz, and the application of similarity joins and selects to overcome problems with that approach.

With respect to investigating the cleanness of data sets, we concluded that the quality of automated clustering depends on the chosen boundary, and that this boundary is dataset-dependent. Our proposed “Linear Edit Distance” measure performs as well as the normalized edit distance, and has been successfully applied to

(Q-gram) similarity joins and selects as a replacement for the regular edit distance measure.

Similarity joins proved to difficult optimize. Although the join itself could be optimized using Q-grams and using a specialized join strategy, the (even filtered) join result remained quadratic, and we concluded that it is simply not possible to build an efficient, scalable cleaning algorithm based on the naive or Q-gram similarity join. The clustering of the resulting similarity matrix deserves more attention in future research. Although parts of our proposed method could not be expressed in SQL due to its recursive nature, the experimental method worked, but we performed no real evaluation.

Thanks to its good results on the MonetDB database, the Q-gram similarity select opens a perspective into database-oriented textual similarity queries. A hand-optimized algorithm did not gain significant extra performance boost, and therefore we recommend the SQL-based Q-gram select. The performance of the latter was, however, below our expectations on the PostgreSQL platform.

5.7.1 Suggestions to the MusicBrainz developers

The moderated data-management solution of MusicBrainz has proven to work: hand-sampling the MusicBrainz database proved it to be very clean.

The similarity joins that we have investigated can not gain the needed fine-grained detail nor contain the background knowledge (eg. the moderation guidelines) that human moderators have. Besides that, they run in quadratic time in the database size, and this will be a problem when bootstrapping. We conclude that the application of similarity joins for maintaining cleanness of the MusicBrainz database will not work.

The MusicBrainz moderators can be helped, however, by getting better feedback from the tagging system by using a similarity select instead of direct string lookups. We believe that the quality of the similarity-selected results will be better, and at least just as good. A second application inside MusicBrainz is the application of the similarity select in the automatic approval of new TRMs for songs that have a very high (say, 95%) similarity with one of the already existing songs in the database, reducing moderator overhead.

However, there remain two open research points in the similarity select area: the investigation of the bad performance of the Q-gram select on PostgreSQL, and the efficient integration of multiple columns into the Q-gram similarity select.

If the PostgreSQL similarity select cannot be improved, an option for MusicBrainz might be to keep a synchronized MonetDB database next to their master PostgreSQL server. This MonetDB database would then function as a “similarity search” server, handling only queries which require similarity selects. The question here is if the improved quality of results gained by implementing this would be bigger than the effort of implementing and maintaining such a solution.

Chapter 6

Server architectures

6.1 Introduction

The music identification system as implemented by MusicBrainz currently uses a single-server setup. This setup will very probably face scalability problems in the near future if the popularity of the system increases. This chapter outlines different scenarios which are implementations of the music identification system, with the purpose of investigating which scenario will be most applicable to replace the current MusicBrainz single-server scenario with.

In terms of network topology, the scenarios range from fully centralized to fully distributed setups. Every scenario is described in detail, and its key strengths and weaknesses are described. For some models, an experiment to test a key weakness is conducted.

We model the performance of the different scenarios given the increasing popularity of the music application. The main parameter is the number of users of the system, $|U|$. For each scenario we predict how the overall system load will scale in the number of users. The increasing popularity of the music application is modeled by regarding different sizes of the user base, discerning the following three stages.

- **Current state** - The number of users that is using the music identification service (MusicBrainz) currently: $|U| = 45k$.
- **FreeDB popularity** - The case that the system is as popular as the current FreeDB music database is now: $|U| = 25M$.
- **World dominance** - The case that every (periodically) net-enabled Hi-Fi audio device has embedded support for the system: $|U| = 3B$.

The outline of this chapter is as follows. First, the architecture of six different scenarios are described. Then, in section 6.8, we describe the implications of the system architectures and provide results of the experiments which we have done for some architectures. Finally, section 6.9 discusses the outcome of the experiments and formulates a conclusion on which architecture is preferred, at each of the three growth stages.

6.2 Model 1: Fully centralized

The fully centralized version is the setup as it is used now in the MusicBrainz project. Every client connects to the same server, and this single server processes every request. A schematic picture of this setup is given in figure 6.1 (the “computer case” being the server, the “monitors” being the clients of the system).

The pro of using this setup is that the data is always consistent because there is only one server. The obvious drawback is that this solution will not scale well in the number of users. Besides this, a single server is not very reliable. Every user will be affected if the server goes down or is unreachable.

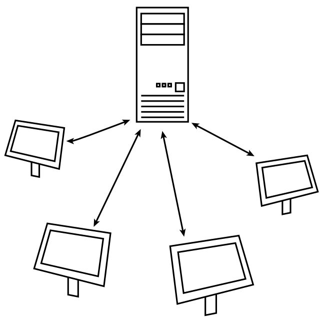


Figure 6.1: *Fully centralized*

6.3 Model 2: Master / slave

This setup, depicted in figure 6.2, uses a ’Master’ server which has the master copy of all data. Furthermore there are a number of “slave” servers which all have a full replica of the master database. These slave servers only handle queries which do not modify the data. Update queries are sent through to the master server, which modifies the master copy.

On updates, the master server maintains a “pending” table which holds all modifications to the database. This “pending” table gets pushed periodically to all slave servers, bringing them up-to-date with the latest changes and additions to the database. This setup is similar to the *DBMirror* replication extension for the PostgreSQL database system.

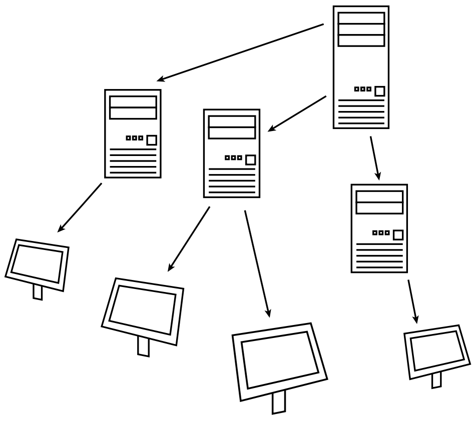


Figure 6.2: *Master/slave*

The strengths of this setup are that the read load can be distributed amongst any number of slave servers. Furthermore, the availability of the whole system is increased: If a slave node fails, other slave nodes can take over, because every slave node has a full replica of the database.

The weakness is that all writes to the database are still handled by a single server, leaving a bottleneck, because the write bandwidth of the whole system is dictated by the write bandwidth of the master server. Besides that, a minor weakness is that the data on the slaves will be inconsistent with the master data between updates if the data changes on the master. This setup is more reliable to node failure than the centralized variant: only if the master fails, updates cannot be done to the database.

6.4 Model 3: Equal servers, single “conflict resolution” server

In this scenario, we have a number of servers which all handle both reads and writes on the database, keeping a log of the changes to the database since last synchronization, similar to model 2. Because any data can be changed or added, conflicts between updates may occur. A typical conflict is that the same artist is entered on two different systems, and thus given two different identifiers, while the entities are essentially equal. The scenario is depicted in figure 6.3.

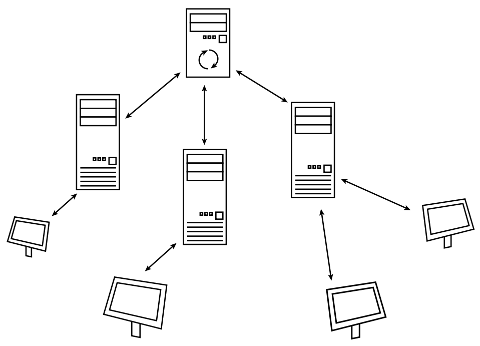


Figure 6.3: *Equal servers with conflict resolution server*

To resolve any conflicts that may occur, a dedicated “conflict resolution” server periodically requests all delta records from every server, and tries to resolve conflicts that might have occurred. The conflict resolution server also keeps a copy of the “clean” database because conflicts may occur between new and existing items too.

Conflicts are resolved by performing the similarity join on every added entity type in the database (artists, tracks, albums). The resulting similarity matrix is clustered (as explained in chapter 5, and winning identifiers are chosen. The results of this resolution are then applied to the local database as well as propagated back to each server.

As we will see in system 4, local database updates are marked “unofficial” until they have been processed by the conflict resolution system.

6.5 Model 4: Equal servers

In this setup, depicted in figure 6.4, every server is equal: there is no server with a special status. The servers all have a full copy of the database, and a client can use any server to read / write to. The servers periodically synchronize their databases, in a “direct-mail” fashion [15].

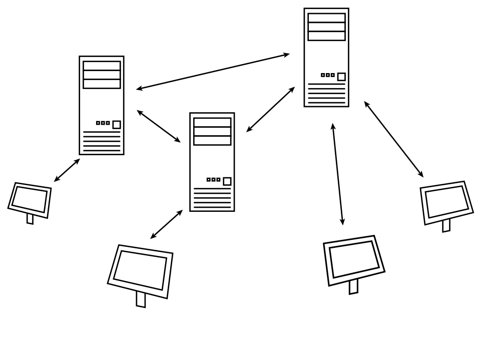


Figure 6.4: *Equal servers*

When new information enters the database of a server (for instance, a new artist), the server gives this entity a fresh identifier, but marking it “unofficial”. It might be the case that at another server, identical information was entered and given another identifier. During database synchronization, similarities between entities marked “unofficial” are detected by performing similarity joins. On a match, the identifier which has gained the most support (number of hosts supporting it), is propagated to the other server and is incremented.

After a full round of synchronization is done, so similarity joins between any unofficial identifiers from the nodes’ databases yield no more results, every identifier is marked official. Note that the unofficial identifiers must be globally unique among all participating servers. This can be accomplished by prefixing the identifiers with a “node identifier”. To keep clients consistent, a mapping from ’old unofficial identifier’ to ’new unofficial identifier’ has to be maintained on the server on which the identifier has changed. That way, clients can update their list of unofficial identifiers, changing identifier value (for unofficial identifiers) or identifier status (for identifiers which have become official).

6.6 Model 5: P2P - Chord ring

This model uses *Chord* [54], a distributed lookup protocol that addresses the problem of how to efficiently locate a node which stores a particular item. The topology of a Chord network is a ring-shape, with “shortcuts”, a few of which can be seen in figure 6.5. Essentially, the Chord ring is a distributed index, mapping keys onto nodes.

In this scenario, nodes with enough resources (e.g. bandwidth) for doing so participate in a Chord ring. Other nodes will first have to search the network for a chord node as their entry point into the ring. Because Chord only supports the lookup of exact keys, the Chord network, which is topologically a *n*-degree graph, consisting of a ring-structure with “shortcuts” to get *n*log(*n*) lookup, will be used to broadcast non-exact lookup queries (eg. similarity selects) and range queries.

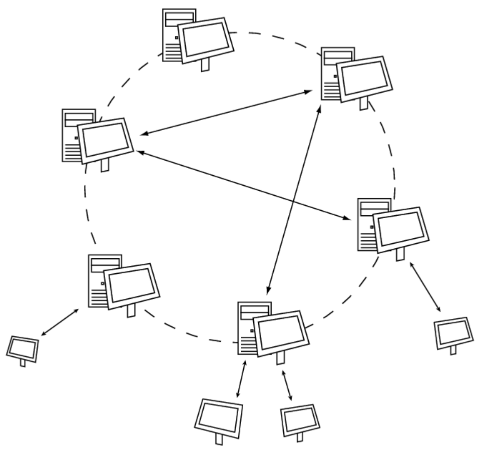


Figure 6.5: *Chord ring*

The TRM fingerprint is used as the primary index in the Chord ring. When a song is being tagged by the Tagging program, its calculated TRM is looked up in the Chord index and a (list of) node(s) on which the track(s) are/is stored is returned. If the TRM is not found in the index, this could mean that either a) the song is a new song in the database, or b) this TRM id is a new TRM id resolving to a track already found.

In case b), we might use the track metadata to lookup the track in the chord index, just like MusicBrainz does in the centralized case (see section 5.4). However, this cannot guarantee that the track will be found because of the metadata fuzziness. A “similarity-broadcast” has to be performed to find out about the existence of the track.

The “similarity broadcast” is done by flooding the network, starting with our neighbors, and doing a similarity select on the track metadata. If one of these selects succeeds, we know with high probability this fingerprint is a new TRM for an already existing track, and we can add the TRM id to the track. If the select does not succeed, a new track is automatically added to the database[1] based on the tracks metadata and TRM id, or, in a more MusicBrainz-like fashion, the user is presented the option to complete all track (and possibly album) metadata, and submit this information as a new moderation request.

The strength of this scenario is that the system (counting only the Chord nodes) is fully distributed: it will scale, and the availability will be high. The use of Chord indexes is fast ($O(nlogn)$).

However, Chord only supports exact key lookup: range queries are not possible, for one thing. For range queries, and more importantly, similarity joins and selects, a broadcast over the network has to be done, “Gnutella-style”, as we will see when we look at the evaluation of this model.

[1]In the automatic case, the basis on which we decide a similarity join to succeed when the TRM does not resolve must be defined very carefully: It can be the case that two *distinct* songs have very similar metadata (e.g. two different live recordings of the same song). This is why, at MusicBrainz, this whole track identification is left solely to the user.

6.7 Model 6: P2P - Gossip propagation

The network architecture of the "gossip-based" propagation is fully P2P and ad-hoc: no nodes have a special status and the network topology is "random", more-or-less based on node nearness. This topology shares similarities with the Gnutella [21] network. For the propagation of key-based updates, a system similar to the "rumor mongering" principle described by Demers et al. [15] is used: nodes are ignorant of an update until they become "infected" and spread the update to neighbors.

For conflict resolution of database inserts, every participant of the P2P network synchronizes regularly with a randomly selected node. This node must be a neighbor of some degree, to prevent network contention (nodes can be bottlenecks, as can be seen in figure 6.6).

When two nodes synchronize, a similarity join between the nodes is done: sending all new data from the target to the initiating node, which resolves conflicts by using a voting system. When identifiers win a voting round, they get rewarded an extra vote, gaining more weight. This will increase the chance that at a next voting round, the identifier will win again.

Because data storage is cheap with respect to communication overhead, clients can keep a full replica of the data. Of course, most of the time this data is out of sync with the other copies of the database in the network, but regular synchronization and the fact that only small amounts of data are changed at a time (with respect to the read load of the database) should be able to deal with this.

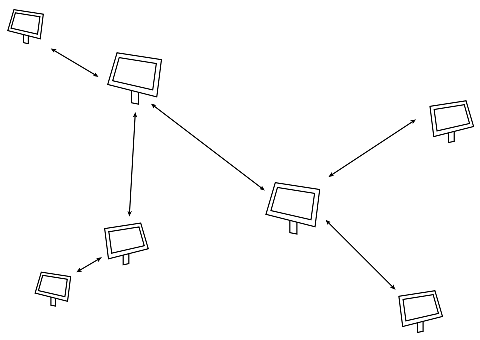


Figure 6.6: *P2P gossip-propagation*

We hope that using this system, somehow the "real" artist names bubble to the top and that the dominant identifiers in an artist cluster will eventually converge.

6.8 Model evaluation

This section discusses the performances of the different server architectures under the growing user base size. Where appropriate, we discuss the results of the conducted experiments. All performance benchmarks conducted in this section were performed on a Pentium IV 1,8 GHz laptop, with 512MB of RAM. The MonetDB version was 4.3.19, using factory-default configuration settings. For PostgreSQL, version 7.4.2 was used, also using its default configuration.

6.8.1 Central server

For modeling the scalability of the system, we looked at the upperbound of the performance of the database by just looking at the most frequent query. As the data survey of chapter 4 points out, the music identification (tagging) task is the most widely used task of MusicBrainz (the others being submitting new tags, browsing the website and committing moderations). This implies that we need to look at the tag frequency, F_{TAG}, and its involved queries.

The model of F_{TAG} shows a linear relation between the number of users and the number of selects. Because for every tag action there needs to be done at least one select (namely, the TRM lookup), even if the "tag" results in submitting a new TRM, the select queries dominate the update queries, and therefore we only look at the select queries in the single-server setup.

Two queries are used in the experiment: the "MusicBrainz track identification query", in query 1 in chapter 3, which is run for every MusicBrainz track identification request, and the query which looks up actual track information (a join on `track`, `album` and `artist`) based on the track identifier(s) that was returned by query 1. This lookup query was only run if query 1 resulted in a "hit", so in 73% of the cases (remember that $H(t) = 0.73$).

Table 6.1 shows the maximum tag frequency benchmarked on the two different database systems.

DBMS	F_{TAG}
PostgreSQL	31.035×10^6 (359 lookups / sec)
MonetDB/SQL	15.501×10^6 (179 lookups / sec)

Table 6.1: *Performance of the track identification query*

So, the ultimate upperbound of the number of users at which the system (using PostgreSQL) collapses is:

$$\begin{aligned} F_{TAG}(U) &= 0.16|U| \\ 52.790 \times 10^6 &= 0.16|U| \\ |U| &= \frac{1}{0.16} 31.035 \times 10^6 \\ &= 193,97 \times 10^6 \end{aligned} \tag{6.1}$$

The upperbound of the system performance is at 193M users. The actual number of users at which the system collapses will be lower, because there are other queries (eg. moderations, browsing the web site) running as well which we have not incorporated. This upperbound indicates that the system can *at most* survive the first two phases of user base size. Since the queries are CPU-bound (it uses an index which fits in memory), the performance will scale with the CPU speed. However, if the database size grows, the indexes will not fit into the memory and the query will become IO-bound. This will be problematic for the largest user base size. Also, given a large user base, a single server setup is simply too unreliable.

6.8.2 Master / slave

Because of the master / slave setup the "read bottleneck" is gone, since the read load is distributed over a number of different servers. However, the update load, which is still handled by a single server, remains to scale linear in the number of requests.

The music identification task being the most prominent task of the MusicBrainz system, the "submit tags" action is the dominating action which modifies the database. This action links newly generated TRMs to tracks in case the TRM lookup failed. It is assumed that the user has selected the right track through the web interface first. The query itself is the “MusicBrainz submit query”, displayed in query 2 in chapter 3.

In this case, new `trmjoin` and `trm` records are inserted, and the indexes updated. Note that a single “submit” thus consists of two queries: an insert in the `trm` table, inserting the new hash, and an insert in the `trmjoin` table, linking the hash to the track.

Table 6.2 shows the maximum submit frequency benchmarked on the two different database systems.

DBMS	F_{SUBMIT}
PostgreSQL	7.948×10^6 (92 submits / sec)
MonetDB/SQL	16.934×10^6 (196 submits / sec)

Table 6.2: *Performance of the submit query*

Using PostgreSQL, we can calculate the number of users at which the system collapses, given that each “miss” results in an insert into the database.

$$\begin{aligned} F_{SUBMIT} &= 0.0432|U| \\ 7.948 \times 10^6 &= 0.0432|U| \\ |U| &= \frac{1}{0.0432} 7.948 \times 10^6 \\ &= 184.01 \times 10^6 \end{aligned}$$

A user base of 184 million will surely take the master/slave setup to its knees. Note that this is slightly less than the read-load, which was estimated at 193M users. However, it might be the case that our assumption of that every “miss” results in a database modification is not sound. However, we have to stick with it as we have no data which indicates the real fraction of this number. The 184 million is a lowerbound, since the actual number of update queries will be lower (thus allowing for *more* users) than our estimate.

By upgrading the I/O bandwidth and the CPU, we think that the update bottleneck can be stretched enough so that this setup will comfortably survive the first two phases of system popularity.

6.8.3 Equal servers with conflict resolution

Model 3, equal servers with a conflict resolution server, will supposedly work better than models 1 and 2, because both “select” and “update” bottlenecks have been removed by distributing them both over multiple servers. However, a new bottleneck is introduced by the special resolution server. The conflict resolution server periodically gathers all delta records from the equal servers, combines this with its own local copy of the full database, and executes a similarity join to cluster the results.

As seen in chapter 5, figure 5.2 shows that the performance for every similarity join remains quadratic in the number of tuples. In this scenario, the cost of this join is even quadratic in both the number of delta records and the number of servers: the total number of deltas is nd and on this number a self join is performed: $(nd)^2 = n^2d^2$.

This finding implies that this solution will not scale. If an extra server is added to handle increased update / select load, the load on the resolution server increases because this new server is incorporated into the resolution process. Its load scales quadratic with the number of delta records this new server adds.

6.8.4 Equal servers

Like model 3, this model suffers from the non-scalability of the similarity join. Although these joins only occur between two nodes at a time, nodes need to communicate with all other nodes to synchronize their changes. Given that there are n servers, $\frac{1}{2}(n \times (n-1))$ synchronizations need to be done each update round, each these synchronizations means a similarity join.

Especially this server setup appeared appealing because of its “semi-P2P” nature: no single server formed a bottleneck to the system. Maybe it had been better had we considered an alternative synchronization strategy: for instance bootstrapping the system with a clean (MusicBrainz) database, and applying the similarity select for conflict resolution of new entries.

6.8.5 P2P - Chord ring

Since the scalability properties of the Chord infrastructure are very good (it guarantees $n\log(n)$), we focus on the broadcast queries like the "similarity broadcast": the existing Chord network is flooded, starting at the neighbors.

This flooding mechanism seems nice, however, it suffers from the so-called "Gnutella-effect": only very frequent information will be found in reasonable time, because there will be many flooding queries running on the network in parallel, causing contention at key nodes and nodes with low bandwidth.

Because we want to avoid this effect, we investigate the partial replication of the database at each node. Every client replicates a certain percentage of the entries in the database. Ideally, the entries are drawn from the inverse popularity distribution of the item (the most infrequent items are replicated most), because the popular items are already distributed in the clients local music collection. However, in practice, the entries are drawn according to the uniform distribution: Since the popularity distribution approximates a power law, its inverse (the "unpopularity" distribution) approximates an uniform distribution, in the tail.

Assuming a uniform distribution of item probabilities, we can derive the likelihood L that an item will be found while searching within n clients, given that each client a fraction p of the database. From probability theory, this likelihood is

$$L = 1 - (1 - p)^n \tag{6.2}$$

At a single client, the probability that this client *does not* have an item i, is $(1 - p)$. So, the probability that n clients do not have an item i, is $(1 - p)^n$, and L is the complement of that.

If we fix the likelihood (eg. at 99%, $L = 0.99$), and specify that we want to broadcast only to n nodes (eg. 500), we can calculate p, the amount of replication we have to do at a single client in the following way:

$$\begin{aligned} 1 - (1 - p)^n &= L \\ 1 - L &= (1 - p)^n \\ ln(1 - L) &= nln(1 - p) \\ p &= 1 - exp(\frac{1}{n} ln(1 - L)) \end{aligned}$$

So, if we want to have a 99% chance of finding the item in 500 nodes, we need a replication of .00916, almost 1%, of the collection, uniformly distributed at each node.

This finding implies that if we can do replication like this, broadcast-like queries only have to cover a relative small part of the P2P network to guarantee finding the items. We can conclude that if the network consists mostly of thick clients, with plenty storage space, partial data replication is "the" way to go if we want to efficiently use P2P-based solutions.

Chord schema

The data is distributed over the nodes, and Chord indexes are created on the most important, "key-like" fields (TRM, artist name, etc).

For artists for example, for every artist, all known "misspellings" of the artist name are stored, along with a vote saying how much this particular spelling was used. The canonical ("real") artist name is the most strongly supported name, the name with the highest number of votes, following the law of the large numbers. Other options for determining the real artist name would be the one with the lowest average string similarity with regard to all other spelling variants, or use an IDF measure like Ananthakrishna [3] does in his paper.

For artist lookups, each node maintains a cache of frequently used, clean, artist names. On a fuzzy artist name X lookup, first we look into our own cache and select the nearest good artist name. If none is found, the message is broadcast to our neighbors, doing a fuzzy lookup (similarity join). This goes on until finally the correct artist name pops up. This name can then be used to lookup the node on which the artist information (including tracks etc) is stored. To that node, a vote on the spelling X might be added.

For track lookups, we can use the chord index with the calculated TRM. If this yields a single result, it is a direct hit. If it yields multiple results, a similarity join between those hits can be done to calculate the most probable track, given that the current track has other metadata (id3 tags, relevant filename). If the TRM lookup yields no result, a similarity broadcast on the track metadata similar to the broadcast at an artist lookup can be done. This broadcast is necessary because a track can have multiple TRMs and our calculated TRM happens to be a new TRM for the track. If this broadcast yields no satisfactory result, the track might be added to the database as a new track with a single TRM.

6.8.6 P2P - Gossip propagation

The advantage of this model is that it is, again, fully distributed (scalable, highly available). Nodes have full or partial copies of the database, and are able to access all data within their subnet easily. Scalability should therefore not be a problem in this solution.

The problem with this model is the propagation of updates through the network. It may take a long time for updates committed at a single node to propagate to every node in the network, and it may even be the case that updates get “out-voted” due to the existence of consistently wrong information, the updates will not be propagated at all. Besides that, low-bandwidth nodes might become bottlenecks in the system if they become “bridges” between otherwise partitioned parts of the network.

In both peer-to-peer models it is a possibility is to consider an alternative to the similarity join to avoid performance problems on local nodes while synchronization is performed. If every node joining the network is bootstrapped by replicating a subset of a clean MusicBrainz database, lasting data cleanness might be possible by application of the similarity select instead of the join.

6.9 Conclusion

For each server architecture we draw an individual conclusion regarding its scalability properties. Next, we apply the conclusions to the three stages of increasing popularity of the music identification system.

A music identification system using a single-master setup has an upperbound of 193 million users on current hardware. However the real number of users will be lower as this estimate assumes the system only supports the tagging operation (not counting the queries involved in the “miss” case or other simultaneous queries). We conclude that this setup has unpredictable scalability and might be too unreliable (in terms of system uptime) to handle a large user base.

A master/slave architecture will not break down before 184 million users, given current hardware, since our update frequency is a pessimistic estimate. By upgrading the I/O bandwidth and the CPU, we think that the update bottleneck can be stretched enough so that this setup will comfortably survive the first two phases of system popularity (stated at $|U| = 45k$ resp. $|U| = 25M$).

Both an “equal server” setup in which every server synchronizes regularly with other servers, and an “equal servers with conflict resolution” strategy which uses a dedicated sync server, suffer from the finding that the proposed synchronization mechanism, the similarity join, scales badly to large databases. Both setups might have performed better had we considered an alternative conflict resolution strategy in an earlier stage.

The only conclusion we have drawn for a music identification service based on P2P is that both proposed models rely on partial replication of data. The scalability of P2P models should be no issue, as P2P networks are supposed to be scalable *by design*, so the remaining issue is the synchronization strategy. Again, considering an alternative for the similarity join might resolve local performance problems.

Having looked at the performance issues that arise at each proposed server architecture, we now apply the conclusions to the three stages of the increasing popularity of the system.

- **Current state** – At about 45,000 users, the current MusicBrainz setup can be used without problems. The master/slave scenario might be considered here to provide a more reliable system.
- **FreeDB popularity** – At 25 million users, master/slave is still the preferred way to go, as the update load will stay below this limit.
- **World dominance** – At world dominance, we know that the more “centralized” models will fail to work, and we have to consider other scenarios, maybe using the Chord-ring architecture or gossip / lossy synchronization on the P2P network of clients. However, both the discussed P2P models still require profound research before a definite statement can be made.

Chapter 7

Conclusion

7.1 Conclusion

In the data survey chapter we have established an analytical model of the music domain, modeling the growth of the domain both in terms of time and in the terms of the size of the user base size of a music identification system. It must be noted that we of course do not know if this model will remain to hold its validity, as the model is based on many assumptions. Only as time progresses we will see whether music databases and their communities continue to show the modeled behavior. However, the model provides an "educated guess" to base further conclusions on.

The data cleanness chapter taught us that the proposed "Linear Edit Distance" measure is as accurate as the normalized edit distance measure, but both measures fail in providing a fully automated solution to perfect data cleanness. Furthermore, the implemented similarity joins remained of a quadratic nature, despite the applied optimizations. Lastly, the Q-gram similarity select proved to be an efficient similarity search and might be applied in the MusicBrainz database to lighten the work of manually keeping the database clean, although either its performance on PostgreSQL has to be improved or MonetDB has to be used for this particular query.

The nature of the music identification problem learned us that, given a small or medium-sized user base, the best server architecture is a simple master/slave replication setup. Given a large user base, P2P might become an option but this direction requires more research.

Getting back to the context of the original problem statement, music recommendation using a P2P database system, this thesis focused on the music identification sub-problem. Music identification has many interesting aspects, in a fairly wide range of research fields. We have attempted to survey these fields and possibly view them in a P2P context. We concluded that while peer to peer techniques remain an interesting research topic, they are not really suited to the music identification task. Other problems (music collection, recommendation) might benefit more from these architectures.

Our overall conclusion is: Given music identification as a sub-problem of P2P music recommendation, the most reliable way is to have a clean, manually moderated music metadata-database in a centralized, preferably master/slave-based setup. Automated methods like the similarity select might relieve the moderation task, providing accurate suggestions to the moderators.

7.2 Future Research

This section on future research is organized as follows. First, we mention the subjects from the global music identification - collection - recommendation system which I have not touched upon because of the scope and size of the problems. Next, I shortly discuss the items which surfaced during the course of this research but for which I did not have the time to look deeply into them.

With relation to the global P2P music recommendation problem, there are two main areas that deserve attention. As this thesis dealt with music identification mainly, the problem areas still left are the music collection and the music recommendation problem. The music collection problem may be researched by looking at the AudioScrobbler project and its data usage, and finding models for the P2P distribution of the raw play log data among the clients. I know this will pose a scalability problem in a centralized setup, because raw play log data just keeps on growing at a fast rate (for this reason, currently the AudioScrobbler project does not log “raw” log entries but only aggregates). The second issue is the research of an efficient algorithm for decentralized music recommendation based on distributed play logs. As far as I know, there is not done much research in this area, almost all collaborative filtering papers suppose a centralized system, and therefore it might be an interesting direction.

Regarding the music identification problem, below is an overview of the main points for future research which I have encountered in the course of this thesis.

- The music data survey can deserve a better estimation for the music collection sizes of users. Our method estimated this number using every AudioScrobbler user, thus producing a skewed estimation since one-time users were counted as well. A better method would be to somehow filter out all non-regular users first before estimating the collection sizes.
- The chapter on music data cleanness also noted some points for future research. It was noted that the research of alternative similarity-joining methods (hierarchical, cosine) deserved more attention, as well as the integration of the Q-gram join with the hierarchical (combining Q-gram– and foreign key metric). The section on similarity matrix clustering could be expanded by providing an evaluation for the proposed method and evaluating alternative clustering mechanisms. The similarity select provided two points: the integration of efficient column-wise rank aggregation algorithm, and the optimization of the Q-gram select on the PostgreSQL platform, as its performance was vastly worse than the naive implementation.
- Finally, in the architectures chapter some server architectures were only outlined but lacked any implementation and evaluation. Especially the two proposed P2P variants are candidates for further investigation. Apart from this, we have seen that a similarity-join based synchronization strategy is far from optimal, so research can be done in this direction as well, researching alternative synchronization strategies, possibly using the similarity select.

7.3 Thanks

I would like to thank Peter Boncz for assisting me and guiding me through the various subjects, and Martin Kersten for the bi-monthly talks with Peter and me, keeping us on the right track or putting us back onto it. Also thanks to all the people from CWI in the INS cluster, and especially Sjoerd Mullender for reviewing the English language in this thesis, and Niels Nes for fixing all those nasty MonetDB/SQL bugs I encountered.

Furthermore I thank Johan Pouwelse from TU Delft for providing insight into the P2P community and power laws, and Dave Evans and Robert Kaye from the MusicBrainz community for providing the different server logs, answering my questions on the mailing list and of course for the great MusicBrainz project as a whole. Also I would like to thank Richard Jones from the AudioScrobbler project for providing me the AudioScrobbler data sets.

Finally, I would like to thank my girlfriend, Ezra, for the mental support and patience with me, my parents and brother and sister, just because they’re such nice people, and everybody who I have forgotten to mention.

Appendix A

The Q-gram self-join in MIL

This appendix lists the code which performs the Q-gram similarity self-join in the MonetDB Intermediate Language. The join itself is implemented as the custom C function (`qgramselfjoin`) working on sorted, aligned columns of row values.

The MIL code below loads the columns from the database, performs initial ordering on Q-gram identifiers and sub-ordering on Q-gram position, and then calls the `qgramselfjoin` function which outputs pairs of “candidate identifiers”. Next we count the number of similar pairs, and apply count filtering and the expensive editdistance step on the resulting candidates. A normal application would then return these candidates to the user, but this specific function only prints the number of result tuples, as this was our interest.

```
# ### load columns from database
var artist_artist := mvc_bind(myc, "sys", "n_artist_subset", "artist", 0);
var artist_number := [int](mvc_bind(myc, "sys", "n_artist_subset", "number", 0));
var artist_len := [length](artist_artist);
var qgram_jid := [int](mvc_bind(myc, "sys", "n_artist_qgram", "id", 0));
var qgram_pos := [int](mvc_bind(myc, "sys", "n_artist_qgram", "pos", 0));
var qgram_qgram := mvc_bind(myc, "sys", "n_artist_qgram", "qgram", 0);

# ### project them onto eachother
var proj_artist_oids := artist_number.join(qgram_jid.reverse());
var proj_ids := proj_artist_oids.mirror().join(artist_number).reverse()
                .mark(0@0).reverse();
var proj_len := proj_artist_oids.mirror().join(artist_len).reverse()
                .mark(0@0).reverse();
var qgram_id := qgram_qgram.join(qgram_qgram.reverse().kunique().mark(0@0))
                .reverse().mark(0@0).reverse();
var proj_pos := proj_artist_oids.join(qgram_pos).reverse().mark(0@0).reverse();
var proj_qgram := proj_artist_oids.join(qgram_id).reverse().mark(0@0).reverse();

# ### sort them
var st := proj_qgram.reverse().sort().reverse();
var sortkey := st.CTrefine( proj_pos ).mark(0@0).reverse();

var sorted_qgram := sortkey.join(proj_qgram);
var sorted_pos := sortkey.join(proj_pos);
var sorted_ids := sortkey.join(proj_ids);
var sorted_len := sortkey.join(proj_len);

print("Before selfjoin: " + sorted_qgram.count());

# ### call the qgramselfjoin function

var candidates := qgramselfjoin(sorted_qgram, sorted_ids, sorted_pos, sorted_len, c, k);

# ^-- yields [oid, oid] bat with candidates.
```

```
print("After selfjoin: " + candidates.count());

# ### now, make the [oid, oid] pairs into the following:
# grouped_a: [void, oid] grouped_b: [void, oid], grouped_cnt: [void, int]
# (the oids are the indices into the artist_* bats)

var left := candidates.mark(0@0).reverse();
var right := candidates.reverse().mark(0@0).reverse();

var s1 := left.reverse().sort().reverse();
var s2 := s1.CTrefine(right);

var grouped_a := {min}(s2.reverse()).join(left).reverse().mark(0@0)
                 .reverse().join(artist_number.reverse());
var grouped_b := {min}(s2.reverse()).join(right).reverse().mark(0@0)
                 .reverse().join(artist_number.reverse());
var grouped_cnt := {count}(s2.reverse()).reverse().mark(0@0).reverse();

print("Pairs: " + grouped_a.count());

# ### group filtering:
var group_filter := ([>=](grouped_cnt, grouped_a.join(artist_len).[-]( (k-1)*3+1 ))).[and]
                    ([>=](grouped_cnt, grouped_b.join(artist_len).[-]( (k-1)*3+1 )));

grouped_a := group_filter.[ifthen](grouped_a);
grouped_b := group_filter.[ifthen](grouped_b);
grouped_cnt := group_filter.[ifthen](grouped_cnt);

print("Group filtered: " + grouped_a.count());

# ### And final filtering, using Linear Edit Distance
var kflt;
if (isnil(c) or c = 0.0) {
        kflt := flt(k);
} else {
        kflt := [min](grouped_a.join(artist_len),
                      grouped_b.join(artist_len)).[*](c).[+](k);
}

var dist_filter := [flt]([editdistance2](grouped_a.join(artist_artist),
                                         grouped_b.join(artist_artist))).[<=](kflt);

grouped_a := dist_filter.[ifthen](grouped_a);
grouped_b := dist_filter.[ifthen](grouped_b);

print("Number of final artistSim pairs: " + grouped_a.count());

# done!
```

Appendix B

The MusicBrainz Similarity Select

This appendix describes an implementation of the full similarity select (which is used by MusicBrainz in an Perl implementation) direct on the database level, in SQL. We use the same solution as MusicBrainz to deal with missing metadata: a matrix which holds weights to the available metadata, adjusting weights for missing values.

We use a table `_metadata` with a single record holding the metadata which is to be investigated. On this table, a view (`metadata`) is created which selects all fields from `_metadata` and calculates an *index* for use in the weight table. The SQL code for creating the view is shown in query 11.

Query 11: *The* `_metadata` *table and* `metadata` *view*

```
CREATE TABLE _metadata (
        artist VARCHAR(255),
        album VARCHAR(255),
        track VARCHAR(255),
        tracknum INTEGER,
        length INTEGER
);

CREATE VIEW metadata AS
SELECT artist, album, track, tracknum, length,
((CASE WHEN artist IS NULL OR char_length(trim(artist))=0 THEN 0 ELSE 16 END) +
 (CASE WHEN album IS NULL OR char_length(trim(album))=0 THEN 0 ELSE 8 END) +
 (CASE WHEN track IS NULL OR char_length(trim(track))=0 THEN 0 ELSE 4 END) +
 (CASE WHEN tracknum IS NULL OR tracknum=0 THEN 0 ELSE 2 END) +
 (CASE WHEN length IS NULL OR length=0 THEN 0 ELSE 1 END)) AS weightindex
FROM _metadata;
```

In the final similarity select, the metadata view joins on the weight table using the weight index. These tuples make a Cartesian product with the inner join of the artist,album and track tables, so that a similarity select is performed on each entry in the metadata view (however, this view will typically consist of one entry only). The full similarity select is listed in query 12.

Query 12: *The Similarity select query*

```
SELECT w0 * similarity(metadata.artist, artist.name) +
       w1 * similarity(metadata.album, album.name) +
       w2 * similarity(metadata.track, track.name) +
       w3 * (metadata.tracknum = track.tracknum) +
       w4 * (CASE WHEN abs(metadata.duration - track.duration) > 30000
       THEN 0.0 ELSE 1.0 - abs(metadata.duration -
            track.duration)/30000 END) AS sim

FROM metadata INNER JOIN
     weight USING (weight.idx = metadata.weightindex),
     artist INNER JOIN
     album ON (artist.id = album.artist) INNER JOIN
     track ON (album.id = track.album),

ORDER BY metadata.artist,
         metadata.album,
         metadata.artist,
         sim DESC;
```

To make this query possible we had to extend the `track` table of MusicBrainz with extra `album` and `tracknum` parameters. The table `weight` which holds the weight values for dealing with missing values is listed in table B.1.

	artist	album	track	tracknum	duration
0	0	0	0	0	0
1	0	0	0	0	0.95
2	0	0	0	0.95	0
3	0	0	0	0.25	0.70
4	0	0	0.95	0	0
5	0	0	0.75	0	0.20
6	0	0	0.75	0.20	0
7	0	0	0.65	0.10	0.20
8	0	1.00	0	0	0
9	0	0.80	0	0	0.20
10	0	0.80	0	0.20	0
11	0	0.70	0	0.10	0.20
12	0	0.50	0.50	0	0
13	0	0.40	0.40	0	0.20
14	0	0.45	0.45	0.10	0
15	0	0.35	0.35	0.15	0.15
16	0.95	0	0	0	0
17	0.75	0	0	0	0.20
18	0.85	0	0	0.10	0
19	0.60	0	0	0.10	0.25
20	0.48	0	0.47	0	0
21	0.43	0	0.42	0	0.10
22	0.43	0	0.42	0.10	0
23	0.38	0	0.37	0.10	0.10
24	0.50	0.50	0	0	0
25	0.45	0.45	0	0	0.10
26	0.45	0.45	0	0.10	0
27	0.40	0.40	0	0.10	0.10
28	0.33	0.33	0.34	0	0
29	0.30	0.30	0.30	0	0.10
30	0.30	0.30	0.30	0.10	0
31	0.25	0.25	0.25	0.125	0.125

Table B.1: *The content of the* `weight` *table*

Bibliography

[1] Lada A. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. 2002.

[2] Rafael Alonso and Henry F. Korth. Database system issues in nomadic computing. pages 388–392, 1993.

[3] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Gant. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 28th VLDB Conference, Hong Kong, China*, 2002.

[4] The APE tag version 2.0. `http://www.personal.uni-jena.de/~pfk/mpp/sv8/apetag.html`.

[5] A. N. Arslan and O. Egecioglu. Efficient algorithms for normalized edit distance. *J. Discret. Algorithms*, 1(1):3–20, 2000.

[6] The AudioScrobbler Project. `http://www.audioscrobbler.com/`.

[7] Audio Fingerprinting @ Philips. `http://www.research.philips.com/InformationCenter/Global/FArticleSummary.asp?lNodeId=927`.

[8] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, May 2002.

[9] P.A. Boncz and C. Treijtel. AmbientDB: Relational Query Processing in a P2P Network. 2003.

[10] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, 1995.

[11] Chris J. C. Burges, John C. Platt, and Jonathan Goldstein. Identifying audio clips with rare. In *Proceedings of the eleventh ACM international conference on Multimedia*, pages 444–445. ACM Press, 2003.

[12] John Canny. Collaborative filtering with privacy. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002.

[13] Giorgio Ghelli Carlo Sartiani, Paolo Manghi and Giovanni Conforti. Xpeer: A self-organizing xml p2p database system. 2003.

[14] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. pages 201–212, 1998.

[15] Alan Demers, Dan Greene, Carl Houser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. *SIGOPS*, 22(1):8–32, 1988.

[16] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of SIGMOD 2003, June 9-12, San Diego, CA.*

[17] The FLAC format. `http://flac.sourceforge.net/format.html`.

[18] The FLite speech synthesizer. `http://www.speech.cs.cmu.edu/flite/index.html`.

[19] The FreeTTS project. `http://freetts.sourceforge.net/docs/index.php`.

[20] Normalized edit distance source code. `http://search.cpan.org/src/MLEHMANN/String-Similarity-1/fstrcmp.c`.

[21] The Gnutella website. `http://gnutella.wego.com/`.

[22] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[23] Luis Gravano, Panagiotis G. Ipeirotis, Nick Koudas, and Divesh Srivastava. Text joins for data cleansing and integration in an rdbms. In *Proceedings of the 19th International Converence on Data Engineering (ICDE03)*, 2003.

[24] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *Proceedings of the Fourth Workshop on the Web and Databases (WebDB'2001)*, Santa Barbara, California, USA, 2001.

[25] Jaap Haitsma, Ton Kalker, and Job Oostveen. Robust audio hashing for content identification, 2002.

[26] Thomas Hofmann. Collaborative filtering via gaussian probabilistic latent semantic analysis. In *Proceedings of SIGIR 2003, July 28-August 1, 2003, Toronto, Canada*, 2003.

[27] Thomas Hofmann and Jan Puzicha. Latent class models for collaborative filtering. In *Proceedings of IJCAI'99*, 1999.

[28] How much information? 2003. `http://www.sims.berkeley.edu/research/projects/how-much-info-2003/`.

[29] The ID3v1 specification. `http://www.id3.org/id3v1.html`.

[30] The ID3v2 specification. `http://www.id3.org/develop.html`.

[31] Carl Kadie John S. Breese, David Heckerman. Empirical analysis of predictive algorithms for collaborative filtering, 1998.

[32] George Karypis. Evaluation of item-based top-*n* recommendation algorithms, 2000.

[33] V.I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones (original in russian). *Russian Problemy Peredachi Informatsii 1*, pages 12–25, 1965.

[34] H.V. Jagadish et al. Luis Gravano, Panagiotis G. Ipeirotis. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th VLDB Conference, Roma, Italy, 2001*, 2001.

[35] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks, 2003.

[36] Samuel Madden, Michael J. Franklin, and Joseph Hellerstein. Tag: Tiny aggregate queries in ad-hoc sensor networks. In *Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002*, 2002.

[37] Andrés Marzal and Enrique Vidal. Computation of normalized edit distance and applications. *IEEE trans. on Pattern Analysis and Machine Intelligence*, 15(9):926–932, 1993.

[38] The MonetDB RDBMS. `http://monetdb.cwi.nl/`.

[39] MonetDB Benchmarks. `http://monetdb.cwi.nl/Research/Benchmarks/`.

[40] The MusicBrainz Project. `http://www.musicbrainz.org/`.

[41] Musicbrainz database statistics. `http://www.musicbrainz.org/stats.html`.

[42] MusicBrainz Style Guidelines. `http://www.musicbrainz.org/style.html`.

[43] Eugene W. Myers. An o(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[44] David M. Nichols. Implicit rating and filtering. In *Proceedings of the 5th DELOS Workshop on Filtering and Collaborative Filtering, Budapest, Hungary, 1997*, pages 31–36. ERCIM, 1997.

[45] Douglas W. Oard and Jinmook Kim. Implicit feedback for recommender systems, 1998.

[46] Ogg Vorbis comment field recommendations. `http://reactor-core.org/ogg-tag-recommendations.html`.

[47] Vassilis Papadimos and David Maier. Mutant query plans. 2001.

[48] The Postgresql RDBMS. `http://www.postgresql.org/`.

[49] Relatable. `http://www.relatable.com/`.

[50] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work*, pages 175–186, Chapel Hill, North Carolina, 1994. ACM.

[51] Recording Industry of America (RIAA) sales figures 2003. `http://www.riaa.com/news/newsletter/pdf/2003yearEnd.pdf`.

[52] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Third Edition*. McGraw-Hill, 1997.

[53] The Soundex index. `http://www.archives.gov/research_room/genealogy/census/soundex.html`.

[54] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. pages 149–160, 2001.

[55] Tuneprint. `http://www.tuneprint.com/`.

[56] S. Upendra. Social information filtering for music recommendation, 1994.