The Faculty of Engineering and Science
Aalborg University

Department of Computer Science

A MUSIQUE FRAMEWORK
– Efficient and Extensible Framework for Audio Music Similarity Querying

PROJECT PERIOD:
F10S
February 1st, 2006 –
June 13th, 2006

PROJECT GROUP:
d627a

STUDENTS:
Kim Schulz
Kovarththanan Rajaratnam
Per Bech Jensen

SUPERVISOR:
Christian S. Jensen

ABSTRACT:

In this paper we present a framework for implementing similarity query enabled multimedia systems on top of a Relational Database. Focus is on flexibility and different implementations of the components of the framework are proposed. Special focus is put on the index structure M-Grid which can be used in the framework.Formal description, Design, implementation and optimizations of the M-Grid is described and evaluated. Evaluation of the framework proves the flexibility and robustness, and that it is a suitable platform for both test systems and real-world implementations – which is also further supported by different case study implementations.
Keywords: Algebra, Music, Similarity, Flexibility, Multi-dimensional, M-Grid, JDO, Framework.

COPIES: 7

REPORT PAGES: 50

APPENDIX PAGES: 7

TOTAL PAGES: 57

A MUSIQUE FRAMEWORK

Efficient and Extensible Framework for Audio Music Similarity Querying.

Master Thesis by

Kovarththanan Rajaratnam, Kim Schulz and Per Bech Jensen

<{krj,kim,pbj}@cs.aau.dk>

Department of Computer Science, Aalborg University, Denmark

June 13, 2006

Contents

Abstract

In this paper we present a framework for implementing similarity query enabled multimedia systems on top of a Relational Database. Focus is on flexibility and different implementations of the components of the framework are proposed. Special focus is put on the index structure M-Grid which can be used in the framework.Formal description, Design, implementation and optimizations of the M-Grid is described and evaluated. Evaluation of the framework proves the flexibility and robustness, and that it is a suitable platform for both test systems and real-world implementations – which is also further supported by different case study implementations.

Keywords: Algebra, Music, Similarity, Flexibility, Multi-dimensional, M-Grid, JDO, Framework.

Acknowledgments

This project took place within the Intelligent Sound research project, funded by the Danish Research Council for Technology and Production Sciences as project no. 26-04-0092.

Chapter 1

Introduction

1.1 Motivation

In recent years, audio music listeners around the world have moved from being CD buyers to a more buy-what-you-like approach, where audio music is bought track-wise rather than as a compilation. At the same time our audio music collections (commercially and private) have grown increasingly every day, and with no ”albums” to organize by, the task of organizing the audio music can often become tiresome.

The listening patterns have also changed from listening to single albums to now where the listeners combine songs in a playlist according to different factors instead – e.g., mood, artist, genre, and tempo.

Both of these changes yield a new approach for organizing the audio music and generating playlists. One approach is to look more on the actual content of the audio music. Her it is possible to find specific characteristics about the music – its features – and use these to both organize by and query the audio music collection with. An example could be audio music organized by the beat of the song. The user could then query the audio music collection for songs with the same beat (songs having similar feature data for the beat), and from this, get a playlist that is similar to this particular beat feature.

With this project we have addressed some of the problems surrounding the changes needed in order to set up a system that support indexing of audio music with respect to its features. The system should also support queries for similarity according to one or more features of the audio music – Hence make it possible to generate playlists.

We handle this problem by implementing a modular framework, that can ease the task of implementing and test a system supporting these tasks.

The framework is based on the Intelligent Sound Algebra (ISA) [1] which is a similarity query enabled algebra for audio music databases.

We call the project “A MUSIQUE Framework”, which is short for “Audio MUsic SImilarity QUEry Framework”.

1.2 Audio Music Queries

The recent changes to the way people organize their audio music and the way they look at the audio music to generate playlists has led to other ways of querying an audio music collection.

A playlist is a convenient method for ordering audio music a user wants to listen to. The songs in the playlist are mostly played in a sequential order and therefore the ordering has great impact on the listening experience — e.g., a heavy metal song between two or more pieces of classical music would disturb the relaxation of the listener. Often a music listener selects the audio music according to his mood.

This presents some interesting queries the user could perform in order to get a playlist he likes. We will focus on three specific query types: Range queries, Nearest Neighbor (kNN) queries, and Transition queries.

If the user is in the mood for some classical rock and loves the song “Hotel California” by The Eagles, he could consider performing a range query like:

Give me all songs that are X similar to Y

where X is the precision of similarity (in the range [most similar ; least similar]) and Y is “Hotel California”. If he sets the X value to a too high precision he might not get more than one or two songs in his playlist. To circumvent this, he could then either loosen his precision value or he could consider performing a kNN query like:

Give me the k songs that are most similar to Y

This would return *k* songs that are "the most similar" to the song *Y*.

Maybe the user already has a playlist full of songs. The songs are however in a random order, which could again disturb the listening experience. Therefore the user may wish to order the songs in a manner such that the transition between the songs is as smooth as possible. A transition query for this could be like:

Give me a transition between songs in the playlist D

Other transition queries could be derived from this, if the user wants to have further control over the final playlist. This could for instance be:

- having a specific start and/or end song in the playlist.
- having a smooth transition in only a part of the playlist.
- having a specific number of songs in the final playlist.

While listening to the songs in a playlist the user may bump into songs that he does not like. Therefore he might want to update the playlist to exclude songs being similar to the specific song that he did not like. The query for that could be made from a combination of a Difference operation and a Range query and could be described with:

Remove all songs from the current playlist that are X similar to Y

All of these queries could be interesting for the user to perform, when working with his audio music collection, but today this is not a possibility in any system to our knowledge.

1.3 Prerequisites

In order to build a framework for similarity queries, there is a minimum number of areas that needs to be covered in order to make it acceptable. Some of the requirements are absolutely necessary, while others are just to optimize performance and get an optimal user experience with the system.

The following requirements are all necessary to get a similarity system to work:

Good Features Without good features extracted from the audio music, the system will never work. The feature data is the fingerprint for a song and without a "good" fingerprint, the system will not be able to find similar songs.

Good Distance Function When working with songs of variable length and with a dynamic content, it is of severe importance that a good distance function is present. The distance function is the final step when measuring how similar two songs are.

A choice has to be made when it comes to the distance function. Should we adapt the songs to a fixed length for direct distance measure or should the distance function adapt the song dynamically.

Support for High-Dimensional Data When it comes to feature data from audio music files, it is typically high-dimensional. There can be done a lot to reduce the number of dimensions, but in the end it is still either of a quite high number of dimensions or useless due to approximations.

This means that the system has to be able to handle high-dimensional data in order to work correctly with all types of feature data.

Besides the above requirements, there are the following good-to-have performance requirement:

Fast Index Structure There are often large amounts of feature data and traversing it for similar songs is often a tedious task. An index structure for the data can help speeding up similarity queries.

1.4 Contributions

The contributions of this project can be grouped into the following parts:

Core System A Java™ implementation of a middleware that includes what is needed to support the new way of indexing and querying by feature data. The middleware is a modular framework and includes a query server (including a query language), a query processor, an index structure, and a Data Mapper that maps an Object-Relational Mapping (O/RM) of the data objects to the Underlying Relational DataBase Management System (RDBMS). Focus is on designing and implementing a flexible system, and not on finding and testing different audio music features – hence the choice of features to use is left open to the user of the system.

Index Structure To explore the problems surrounding indexing high-dimensional feature data, a Java™ implementation of the M-Grid [2] index structure is conducted. The implementation is constructed to be part of the XXL [3] framework, and flexibility is of severe importance in order do evaluations of the system.

Case Studies Two case studies on how the core system can be used, has been explored by implementation and evaluation. The first case is an on-line Web-Player which is a proof-of-concept implementation of a client for the core system. The Web-Player showcases all the features that the core system offers to its clients. The second case is an on-line music shop that sells audio music by the track. The music shop is meant as a real-world example of how the core system could be used in already existing systems.

Evaluation A thorough evaluation of the implemented parts is conducted to determine positive sides and drawbacks of the flexibility in the system. Different implementations of the changeable parts are tested to see if a pattern can be found in the result.

1.5 Related Work

In this section we will look at three groups of related work. The first is about implementation of frameworks and similarity enabled systems. The second is about M-Grid and related index structures, and finally, the last group is about algebras and similarity query enabled algebras in particular.

Framework construction is a common way to implement a system with flexibility in focus. Since we use Java™ for our implementation, a project worth mentioning is the XXL Framework [3] which implements several different indexing structures (trees) and methods to handle disk I/O. The structure used in XXL for its indexing structures is copied in our implementation of the M-Grid – and the M-Grid is hereby possible to use as part of XXL.

In the area of implementing frameworks that supports similarity queries in databases there has however (to our knowledge) not been done any work besides ours. Projects like the Cuidado Project [4] has developed a framework to work with similarity queries in audio music, but their primary focus is on the descriptors for the feature data. Our framework is based on database theory and does not deal with feature extraction so, the Cuidado framework does not cover the same area of tasks as ours.

Uitdenbgerd et al. [5] presented in 1999 a framework that performs audio music similarity matching in three steps. This framework is used to compare a range of techniques for determining the similarity of two pieces of audio music. The framework does however rely on midi-based music files and the ability to fetch transcriptions from these. Since our framework relies only on the extracted feature data and not the audio music files and not midi music files, this approach would not be sufficient for our needs.

Seeing our middleware as a complete system that serves some client application with the ability to do similarity queries, it comes very close to the system developed by Welsh et al. [6]. Their database in the system is however implemented by the use of flat-files and to calculate the distances they use a brute-force approach. They have, as us, implemented a client for the system to test it with real-world users.

M-Grid is the index structure we have chosen to focus on in our project. Digout introduces the M-Grid for the first time in his thesis [7] and later in a technical paper [2]. The M-Grid distinguishes itself from other similar index structures by being able to handle metric data and not just vector data, which its close relative VA-File only can handle [8]. Both the VA-File and the M-Grid uses pivot points to form a pseudo-grid to split the space (vector and metric) into cells to represent the high-dimensional data in lower dimensions. The concept of using pivot points for proximity searches in metric spaces have earlier been covered by Bustos et al [9].

In Digouts work, some questions and problems (like "full coverage") are, however, left unanswered, and we have therefore elaborated on these in our work. Some of the problems lack a formal description so this is also given in this paper.

Algebras for audio music and audio music similarity queries in particular is not a research area that has been studied a lot. Bonatti et al. [10] constructed an extension to Relational Algebra (RA) to support similarity queries. They investigated different approaches for the operators. Some of their operator ideas are quite similar to the ones in the Intelligent Sound Algebra (ISA) [1], but the data structure they work on is normal relations and not like the nested Dataset structure we have presented. Earlier work by Kosch et al. [11] researched in implementing a similarity based algebra for multimedia databases. Their work did, in contrast, only focus on image files and not audio music. The features that can be extracted from images can be seen as the feature for a single sampling period of an audio music file. In our case we do however have several sequences of features of non-equal length for each song and the vast amount of data this produces introduces some new problems that Kosch did not foresee.

1.6 Paper Outline

The rest of this paper is structured as follows:

Chapter 2 introduces the architecture of our core system by describing the main parts: Query Processor, Query Server and Index structure.

How the system will work with multiple features in similarity queries is further described in details in Chapter 3.

In Chapter 4 the index structure — M-Grid — is described in details and the implementation design is reasoned for.

In Chapter 5 and 6 the project is evaluated – First by looking at some case studies, and then by doing a formal test and evaluation.

Finally in Chapter 7 we give a summary of the project and elaborate on future work. In the back there is Appendices describing parts of the system suite and the work-flow when using them, known issues and the configuration files used for the system.

Chapter 2

Architecture

2.1 Overview

In this chapter we will go through the main components of the architecture of our middleware. An overview of the components is shown in Figure 2.1. In the figure, components marked with a ***** are changeable (also applies to Figures 2.2 and 2.4). Some components consist of several sub-components which are considered important parts of the main component. Each component is described and argued for in this chapter, and the technologies used in them are listed.

In short, the setup is constructed as a middleware on top op an RDBMS that function as persistent storage for the object data. To handle the Object-Relational Mapping (O/RM) between the Query Processor and the RDBMS, the Data Mapper component is inserted. Section 2.2 has a further description of the functionality of the Data Mapper and describes how it is used to translate between the ISA data model and relations in the RDBMS.

The central part is the Query Processor component which handles evaluation of queries formed as trees of nested ISA operators. The Query Processor is further described in Section 2.3.

Queries from clients are received in the Query Server component, which is described in Section 2.4. The queries are expressed in a query language – XQL – which we have developed for this particular purpose.

The index structure is a component of its own. In Section 2.5 and later in Chapter 4 we describe the requirements there is to the index structure, and how we implemented them.

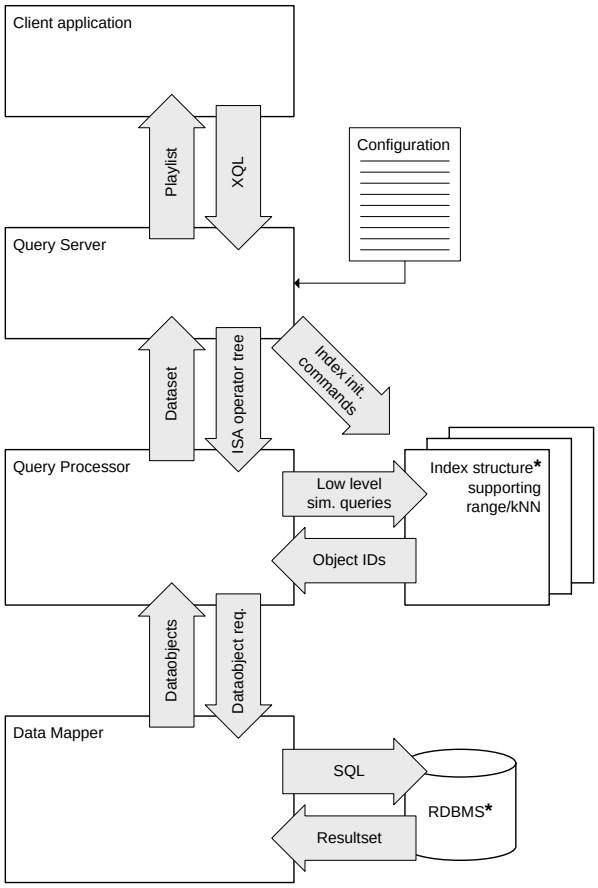

Figure 2.1: *Overview of the main components in the architecture. Components marked with a* ***** *are changeable, and alternative implementations of the component can be inserted instead.*

2.2 Data Mapper

In previous work [1] we introduced a new data model with a main data structure, the Dataset (visualized in the top of Figure 2.3), able to hold a set of Dataobjects (songs) each having some metadata and an amount of multi-dimensional feature data (collected in the Feature Sequence Set (FSS) part). The data model is used in the Query Server and Query Processor modules.

To facilitate persistence of the model, so it is not necessary to store all data in main memory, we introduce the component Data Mapper. This component provides an abstraction over the data model, so that the persistent storage handling is hidden from other components. For the O/RM between the data model and the persistent storage, Java Data Objects (JDO) [12] is used, which is a specification for the Java™ programming language for handling persistence of objects. It has been chosen to use the RDBMS as storage manager, so the O/RM is mapping the in-memory Dataobjects into tables in the RDBMS (middle and bottom of Figure 2.3 showing class- and E-R diagram respectively).

The usage of JDO, the actual JDO implementation, and the RDBMS will be described in the following. The Data Mapper and the components it consists of is shown in Figure 2.2.

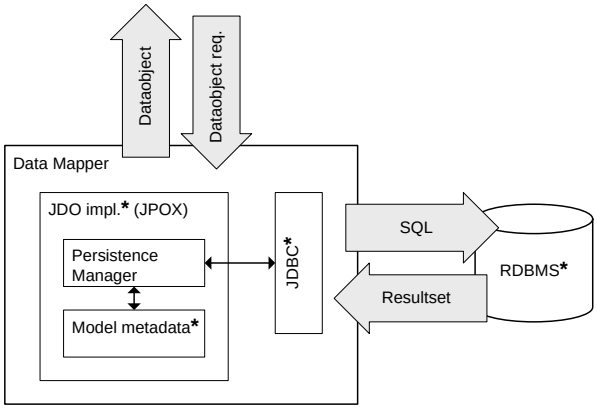

Figure 2.2: *The Data Mapper component handling O/RM of the data model between memory and RDBMS.*

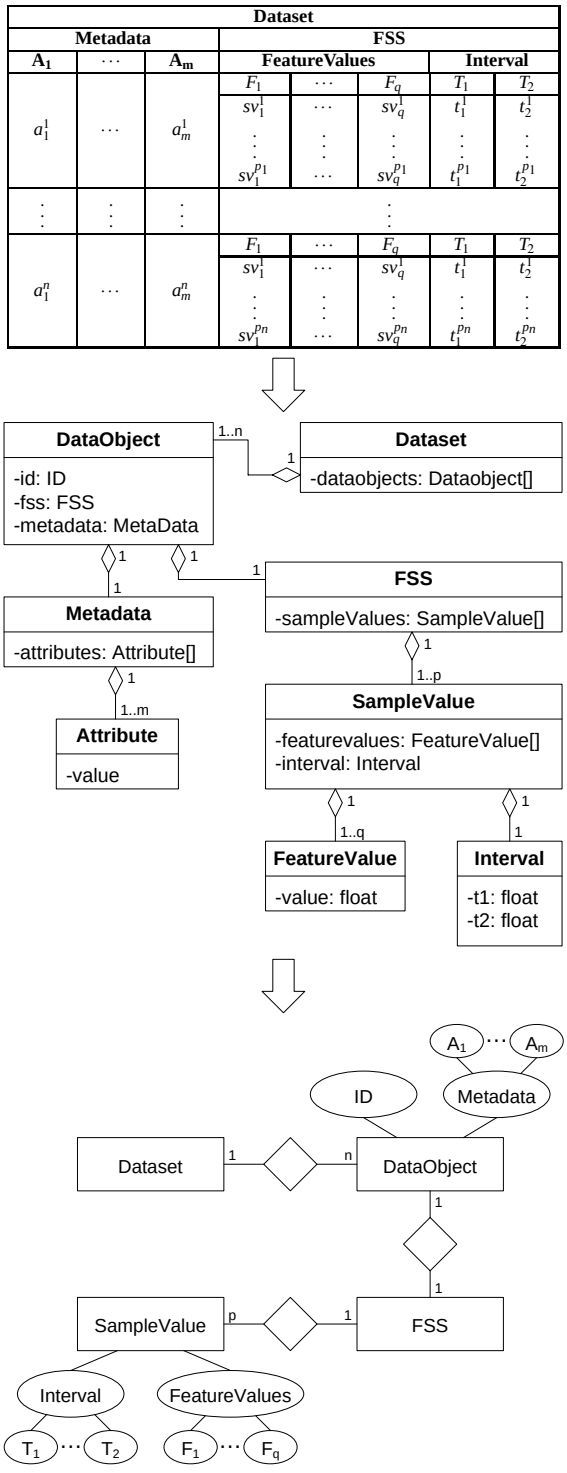

Figure 2.3: *A simplified visualization of the three steps in the data mapping:*
1) *The Dataset in its original form.*
2) *A class representation of the Dataset.*
3) *An E-R diagram of the RDBMS version.*

2.2.1 Object/Relational-Mapping using JDO

A JDO implementation use JDO metadata as a schema descriptor in order handle the O/RM. The metadata contains a description of which elements of the model to make persistent, the connection between the elements, and which tables in an RDBMS to map the elements to.

The Java classes representing the data model are then modified according to the metadata in order to the make the class instances "persistence-capable." By persistence-capable is meant that the data in the object can be read and stored elsewhere (e.g., in an RDBMS) and later be restored to object state again. The data model metadata can be defined in many different ways in order to specify mappings to different relational schema.However, this does not affect the Dataobjects in the system itself.

2.2.2 The JPOX JDO Implementation

The actual JDO implementation used in this system is Java Persistent Objects (JPOX) that supports several RDBMSs (see next section). JPOX handles the translation from several different query languages (JDOQL, JPOXQL, SQL) into RDBMS specific SQL.

It also tags persistent-capable objects with information about their persistence-state. That is, whether they have been filled with persistent data from the database (called a detached object) or if they contain no data (called a hollow object). The application needs to detach an object if it wants to use the data in it. If it changes any of the data then it needs to attach it again in order to make the data persistent (stored in the RDBMS). The actual attaching/detaching of the objects is handled by the JPOX component, Persistence Manager (see Figure 2.2).

2.2.3 RDBMS and JDBC

The changeable RDBMS component is a normal RDBMS supported by the JPOX JDO implementation. JPOX supports the following RDBMSs:

- PostgreSQL
- MySQL
- (Oracle)
- MS SQL Server
- SAPDB/MaxDB
- Cloudscape/Apache
- HSQL DB Engine
- Sybase
- DB2
- McKoi
- Pointbase
- Firebird
- Informix

JPOX uses a Java Database Connectivity (JDBC) [13] driver in order to connect to the RDBMS and execute the SQL queries generated from the O/RM. The JDBC driver is changeable but is specific to the RDBMS used. We have successfully tested our system with JPOX connecting to both PostgreSQL and MySQL. Oracle is in parentheses because it seems that JPOX has a problem handling the BLOB type for that RDBMS.

2.3 Query Processor

A set of operators has been defined in our previous work [1] – the ISA operators – working on Datasets, as mentioned in Section 2.2. A subset of these operators are implemented and may be used to describe queries on Datasets. These operators include the simple relational operators: Select (σ) and Project (π), but also the similarity operators supporting multiple features: Range, kNN, and Transition.

Here, we briefly describe the architectural principles regarding the query evaluation and the nested operator structure. The Query Processor, and especially the multi-feature support, is described in details in Chapter 3. An overview of the Query Processor component is shown in Figure 2.4.

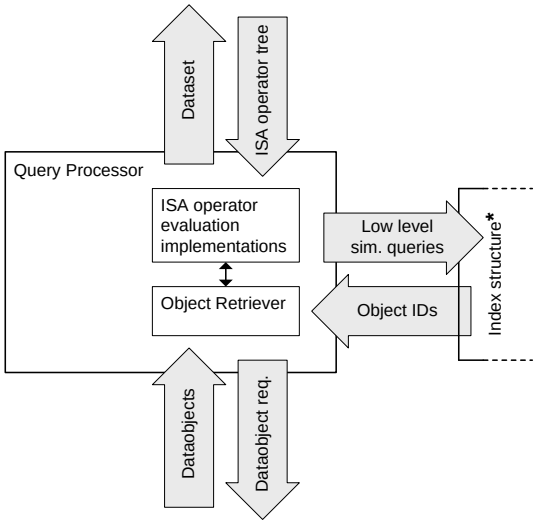

Figure 2.4: *The Query Processor handling evaluation of queries expressed as ISA operator trees, using the Index structures, and the Data Mapper component.*

2.3.1 Evaluation Principles

In general the Query Processor makes use of the Data Mapper (see Section 2.2) during query evaluation – all the way down to the RDBMS. But in case of similarity queries, the indices are used to do a faster evaluation of which songs should be in the result. To handle multiple features, several indices are used – one for each feature. Evaluation using several indices, indexing different feature data, is not a trivial task, which is why this issue is handled in a chapter of its own (see Chapter 3).

Whenever Dataobjects from the input Dataset are needed they are retrieved from the Data Mapper. The actual handling of the retrieval is done by the Object Retriever (see Figure 2.4).

All query processing is done with a naive evaluation approach, meaning that the received query is evaluated as-is, without any query optimizations performed.

2.3.2 Operator Design

The ISA operators are implemented using inheritance. This means that every operator inherits from a basic operator having the input Dataset argument as an attribute in terms of an operator which can be evaluated to a Dataset (see Figure 2.5). Each of the operators defines its specific parameters as attributes in the corresponding class – e.g., the Range operator has a query object and a range as attributes.

The basic operator defines an abstract compute-method used for evaluating a Dataset. The operators all implement this method by first calling the compute method of the nested operators. This returns the Dataset that defines the input to the current operator on which further computations are performed.

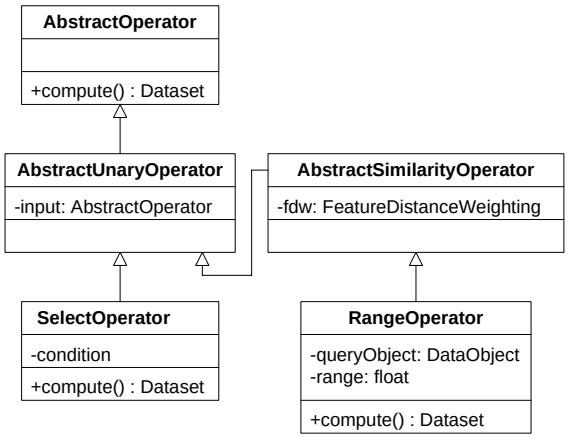

Figure 2.5: *Class diagram showing the operator design using inheritance.*

2.4 Query Server

To connect all the parts in the middleware there is a need for a service. The Query Server works as the main process in the middleware and handles the following: Loading of the index structures, listening for incoming queries from the clients, transformation of the queries into an operator-tree to be evaluated in the Query Processor, and transformation/delivery of the result back to the client.

In the following these parts are described in more detail, referring to the abstract overview of the Query Server shown in Figure 2.6.

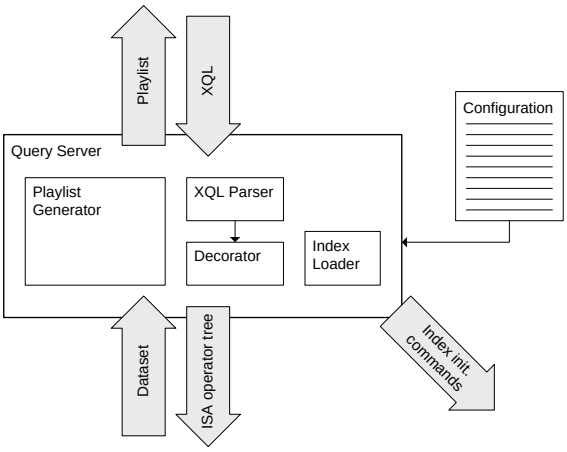

Figure 2.6: *The Query Server component handling interaction with clients, including parsing of incoming queries and transformation of the result into a playlist. Furthermore taking care of loading the index structures to be used.*

2.4.1 Loading the Index Structure

The index structures to be used for query processing are to be loaded properly before any processing can occur. The indices to use, and with which parameters, are specified in the configuration file depicted in Figure 2.6. An example of the configuration of the complete middleware is shown in Appendix B.

2.4.2 Query Language

To be able to test the system from different client applications, we have developed a generic and simple query language named XQL. The language is based on XML and simply wraps around the implemented operators from ISA. Using a container language like XML is generally not an optimal solution, since it has a lot of syntax not really relevant for the actual query. However, it is very easy to construct, extend and parse – hence very usable for a test framework like this.

```
<range>
  <song>
    <id>1432</id>
  </song>
  <rangeval>20</rangeval>
  <order>true</order>
  <refine>false</refine>
  <fdws>
   <fdw>
      <feature>AudioSpectrumEnvelope</feature>
      <distance>Manhattan</distance>
      <weight>1.0</weight>
   </fdw>
  </fdws>
</range>
```

Figure 2.7: *Example of a Range Query in XQL.*

The language syntax supports Select, Project, Range, kNN, Transition, Insert and Delete which is enough to test the queries described in Section 1.2.

An example of a Range query expressed in XQL can be seen in Figure 2.7

2.4.3 Query Parsing

The Query Server uses the library XStream [14] to parse the XML-based input and translate it into a command-tree, which is translated into the corresponding operator-tree. Well-known parser and compiler techniques are used or this process. The XQL Parser component is constructed with the XStream library [14], and handles the first steps transforming the raw query (expressed in XQL) into an Abstract Syntax Tree (AST) representing the command-tree. After that, the Decorator component transforms the command-tree into the corresponding operator-tree. During the transformation the different operators are decorated with settings specific for the individual operators (e.g., which index to use).

2.4.4 Serving Clients

The Query Server creates a TCP/IP socket and listens for connections on it. A new thread is spawned every time a connection from a client is made – hence multiple simultaneous client connections are supported. When a client connects, the input is directed to the Query Parser, and the process continues like mentioned in the previous section. The operator-tree from the Decorator is given to the Query Processor for evaluation. After the Query Processor has evaluated the query into a Dataset, it is transformed into a playlist (containing only the meta-data part of the Dataobjects) via the Playlist Generator and returned to the client.

2.5 Index Structure

In order to optimize the entire system for similarity queries, an index structure is a commonly used solution. The index structure is used solely for the similarity queries, since the Underlying RDBMS already keeps indices for the persistent Dataset (see Section 2.2) stored there – hence these are used for the relational queries (e.g., Select).

To keep the middleware as flexible as possible, a clean interface is kept against the index structure. This result in the possibility for changing the index structure with others as long as they obey the interface. The following criteria are required for the index to be usable in the middleware:

- Built-in Range query support.
- Built-in kNN query support.
- Preferably built-in Transition support.
- Support for high-dimensional data.

In our case we choose to use the M-Grid [2, 7] index structure for our implementation. The M-Grid supports all of the above criteria with the exception of the transition support. This we choose to add to the M-Grid while implementing it. The M-Grid index structure is described in full details in Chapter 4.

2.6 Problems and Solutions

During the design of this system, multiple aspects regarding the implementation of a middleware had to be considered and tested. For this an overall Unit testing system was setup to test both functionality and interfaces and to identify problems in the structure.

In most cases, everything worked as we planned it during the design of the system, but in one case an alternative solution was needed.

JPOX was the JDO implementation used for handling the connection between the in-memory data structures and the persistent copies in the underlying RDBMS. During test we did, however, discover some serious memory consumption problems, when performing simple Select queries such as the following:

```
SELECT * FROM DATASET
  WHERE ID=1
  OR ID=2
  OR ID=5
  OR ID=1000
  OR ...
```

When the number of successive restrictions on the ID attribute reached 20 or more, JPOX consumed hundreds of megabytes, when it in fact only should take up around 1MB. If this was the case only for a short period of time it would be acceptable. But because the memory was never freed, caused the system to run very slow after processing even a small number of such queries due to memory-to-disk swapping. The memory consumption issue made JPOX a bottleneck to the system – hence preventing us from testing the performance of other parts of the system. Communication with the JPOX developer team did not reveal any problems with our use of JPOX, but the developers did not consider this as an important bug to fix.

In order to fix this problem, we implemented a work-around in the Data Mapper component. The solution is to introduce new sub-component called the JDBC Connector, which is working directly on the RDBMS. The component is used instead of JPOX and can be enabled by a switch in the configuration file. Figure 2.8 shows how the Data Mapper component ended up looking.

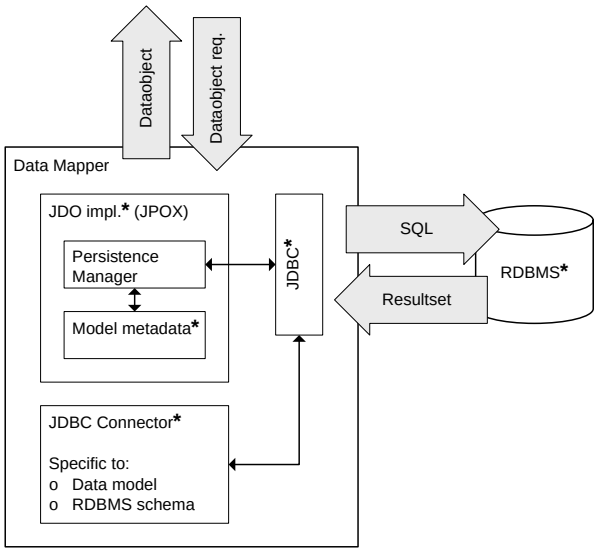

Figure 2.8: *Data Mapper as it ended up looking after memory problems with JPOX.*

The new sub-component, JDBC Connector, in the Data Mapper basically emulates the functionality of JPOX in Select queries, except that it does this without using as much memory. It translates the incoming Dataobject request into SQL and communicates directly with the RDBMS via the JDBC driver.

Chapter 3

Similarity Queries on Multiple Indices

3.1 Overview

In this chapter, we describe how the two similarity queries, Range and kNN, are processed in the presence of multiple features. In order to explain this, we first present some definitions of the similarity queries in the presence of a single feature. This is done in Section 3.2. In Section 3.3, we introduce the use of multiple features and in detail describe how input from the client needs to be mapped in order to account for the use of multiple features. The initial similarity query definitions are in that relation reformulated to add multi feature support for Range and kNN queries.

3.2 Similarity Queries

To explain how the similarity queries are processed in our framework, we here give a generic introduction to the similarity queries. To do so we assume that $\mathbb{S}$ is the domain of all objects, I_i is in the domain $\mathbb{I}$ for all indices such that ∂^{I_i} is a distance function in the domain $\mathbb{D}$ used for indexing objects in index I_i. The signature for the distance function is: $\mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_+$. With this, a Range query can be defined as follows, where the result contains all objects $o \in S \subseteq \mathbb{S}$ within distance r from the query object o'.

Range

$$Range_{o',r,I_i}(S) = \left\{ o \in S \mid \partial^{I_i}(o', o) \leq r \right\} \tag{3.1}$$

In the case of a kNN query, the query definition is a bit different since it has to take into account that the result should be ordered according to the distance. The generic definition of a kNN query is as follows:

kNN

$$\begin{aligned} kNN_{o',k,I_i}(S) &= \langle o_1, \ldots, o_k \rangle \\ &\quad where\ F^{order}_{o',\partial,\langle o_1,\ldots,o_k\rangle}(S)\ holds \end{aligned} \tag{3.2}$$

Here $F^{order}_{o',\partial,\langle o_1,\ldots,o_k\rangle}(S)$ is a function that is able to determine if a list $\langle o_1, \ldots, o_k \rangle$ is sorted according to the ascending distance to o'. Furthermore no other object o in S can have a smaller distance to o' than any element in the list $\langle o_1, \ldots, o_k \rangle$. Formally:

$$\begin{aligned} &F^{order}_{o',\partial,\langle o_1,\ldots,o_k\rangle}(S) = \\ &\bigwedge_{i=2}^{i=k} \Big(o_i, o_{i-1} \in S \wedge \big(\partial(o', o_{i-1}) \leq \partial(o', o_i) \big) \Big) \wedge \\ &\nexists o \in S \backslash \{o_1, \ldots, o_k\} \big(\partial(o', o) < \partial(o', o_k) \big) \end{aligned} \tag{3.3}$$

A straightforward implementation of these operators can be created by performing a brute-force scan of all objects $o \in S$. The task is to avoid this brute-force approach, and address the handling of the data in an optimal manner.

3.3 Multiple Features

In order to support multiple features, we use an index I_i for each feature. Support for p features results in a list of indices $I_1, \ldots, I_p$. The only requirement for these indices is that they index the same set of objects, S. Hence, we omit the S as argument when we in the following present definitions and functions working on multiple indices.

3.3.1 Integration of Features

The problem of using multiple features is that the space of the feature domains can be different, i.e., the clustering of the objects is different and hence, the distance between two objects vary from feature to feature. This

means that it is not possible to compare the distances from different feature domains. We need a common domain for the distance values. This has been chosen to be in the interval $[0,1]$, which we denote as logical distances. The real distances of each feature domain are mapped into logical distances, using the maximum distance between any two objects in the feature domain:

$$F_{I_i}^{max} = \max_{o_1,o_2 \in S} \left(\partial^{I_i}(o_1,o_2)\right) \tag{3.4}$$

Equation 3.4 is used for mapping between the real and logical distances in the following way:

$$F_{I_i}^{R\to L}(d) = \frac{d}{F_{I_i}^{max}} \tag{3.5}$$

The inverse mapping is straight-forward:

$$F_{I_i}^{L\to R}(d) = d \cdot F_{I_i}^{max} \tag{3.6}$$

Having mapped the real distances into their logical counterpart we need to combine these. In that connection we introduce a number of weights $\omega_1,\ldots,\omega_p$, which put more or less importance to the logical distances calculated from the feature domain indexed by $I_1,\ldots,I_p$. We require the weights being within the interval $[0,1]$ and sum to 1. The combined logical distance ∂^L is the sum of each of these weighted logical distances. Formally:

$$\partial^L_{\langle (I_1,\omega_1),\ldots,(I_p,\omega_p)\rangle}(o,o') = \sum_{i=1}^{p} \left(F_{I_i}^{R\to L}\left(\partial^{I_i}(o,o')\right)\cdot \omega_i\right) \tag{3.7}$$

3.3.2 Range

In order to support a range query with a query range r on multiple indices $I_1,\ldots,I_p$ with the associated weights $\omega_1,\ldots,\omega_p$, we need to process all indices. An object o is included in the result if the combined logical distance as given by Equation 3.7 is smaller than r. The signature for this operator is $Range : 2^{\mathbb{S}} \times \mathbb{S} \times \mathbb{R}_+ \times (\mathbb{I} \times \mathbb{R}_+)^p \mapsto 2^{\mathbb{S}}$. Formally:

$$Range_{o',r,\langle (I_1,\omega_1),\ldots,(I_p,\omega_p)\rangle}(S) = \Big\{ o \in S \mid \partial^L_{\langle (I_1,\omega_1),\ldots,(I_p,\omega_p)\rangle}(o,o') < r \Big\} \tag{3.8}$$

3.3.3 kNN

A kNN query on multiple indices $I_1,\ldots,I_p$ with the associated weights $\omega_1,\ldots,\omega_p$ should return a list of size k. Therefore, we require that $\|S\| \geq k$. The result of a kNN query is guaranteed to return the k nearest objects, according to the distance given by the logical distance function in Equation 3.7. Formally, the kNN operator is defined as shown below. It has the signature $kNN : 2^{\mathbb{S}} \times \mathbb{S} \times \mathbb{N}_+ \times (\mathbb{I} \times \mathbb{R}_+)^p \mapsto 2^{\mathbb{S}}$:

$$\begin{aligned} &kNN_{o',k,\langle (I_1,\omega_1),\ldots,(I_p,\omega_p)\rangle}(S) = \langle o_1,\ldots,o_k\rangle \\ &\quad \textit{where } F^{order}_{o',\partial^\omega,\langle o_1,\ldots,o_k\rangle}(S') \textit{ holds} \\ &\quad \textit{with } \partial^\omega(o,o') = \partial^L_{\langle (I_1,\omega_1),\ldots,(I_p,\omega_p)\rangle}(o,o') \end{aligned} \tag{3.9}$$

3.3.4 Transition

No support for Transition on multiple indices has been proposed in this paper. The reason is that we believe that using multiple indices for this type of query is not intuitive for the user. Generating a "smooth" transition in a list of seed songs requires that all indices agree on all songs in the transition such that a "smooth" transition is formed. However, two dissimilar songs (as judged by the user) may be similar according to some feature, since each feature only captures a part of the characteristics of a song.

Instead we will elaborate on Transition on a single index in Section 4.6.6 and 4.6.7, where we give a formal description and show why it is complex even for just one index.

3.4 Properties of Multi-Feature Range

The output from the multi-feature range operation include all objects from the input having a combined logical distance, within the query range r as defined in Equation 3.8. The combined logical distance is defined in Equation 3.7 as the sum of the logical distances from each of the indices involved. In the rest of this section properties derived from these facts are described.

3.4.1 Pruning Range

There exists a pruning range r_i^{prune} for every index I_i with an attached weight ω_i in a range query (having query range r). The pruning range specifies when the logical distance, in that feature domain alone, reaches the query range r. The pruning range r_i^{prune} for an index I_i is defined using the query range r and the weight of that particular index ω_i. Formally:

$$r_i^{prune} = \frac{r}{\omega_i} \tag{3.10}$$

When the logical distance to an object exceeds the pruning range, then the object can not be in the result of the range query, since the combined logical distance exceeds the query range.

3.4.2 Inclusion and Exclusion Properties

In general four cases exist, when considering both the query range r and the pruning ranges $r_1^{prune}, \ldots, r_p^{prune}$ for the indices $I_1, \ldots, I_p$. Consider the following four cases of logical distances in different feature domains, from a query object o' to a candidate object o:

1. *All inside the query range r.*
   The candidate is **inside** the query range.
2. *One or more outside the pruning range* r_i^{prune}.
   The candidate is **outside** the query range.
3. *All between the query range r and the pruning range* r_i^{prune}.
   The candidate is **outside** the query range.
4. *One inside the query range and the rest between the query range r and the pruning range* r_i^{prune}.
   The candidate **may be inside** the query range.

An implementation may benefit from the cases listed above, especially case 1–3 that strictly determines the candidate to be in- or outside the query range.

An example of a multi-feature range query is shown in Figure 3.1. In the example the range query is performed with query object o' on a set S with four objects $o_1, \ldots, o_4$ using three indices $I_1, \ldots, I_3$ weighted $\frac{1}{2}$, $\frac{1}{4}$, and $\frac{1}{4}$ respectively.

Table 3.1 shows the logical distances to the objects in each of the feature domains. The rightmost column holds the combined logical distance ∂^L of each object using Equation 3.7. Using a query range $r = 0.25$ would therefore return a set including the objects o_1 and o_4.

Object	∂^{I_1}	∂^{I_2}	∂^{I_3}	∂^L
o_1	0.20	0.15	0.05	0.15
o_2	0.60	0.10	0.50	0.45
o_3	0.40	0.50	0.90	0.55
o_4	0.05	0.50	0.40	0.25

Table 3.1: *Logical and combined distances from the query object o' to the candidate objects $o_1, \ldots, o_4$.*

3.4.3 Usage in the Actual Implementation

In the actual implementation of the multi-feature range operator, which makes use of the range operation in the index structure, it is case 2 that is the most suitable. This is because the index implementation supports an input candidate set containing objects, which are the only objects allowed in the result. During evaluation of the range operation in the index structure, only the distances to objects within the candidate set are calculated.

So, when evaluating the multi-feature range operator output, the indices are consulted one at a time, using the result of the previous calculation as a candidate set to the next one. Thereby we avoid investigating objects already excluded by the pruning range, during evaluation on previous indices.

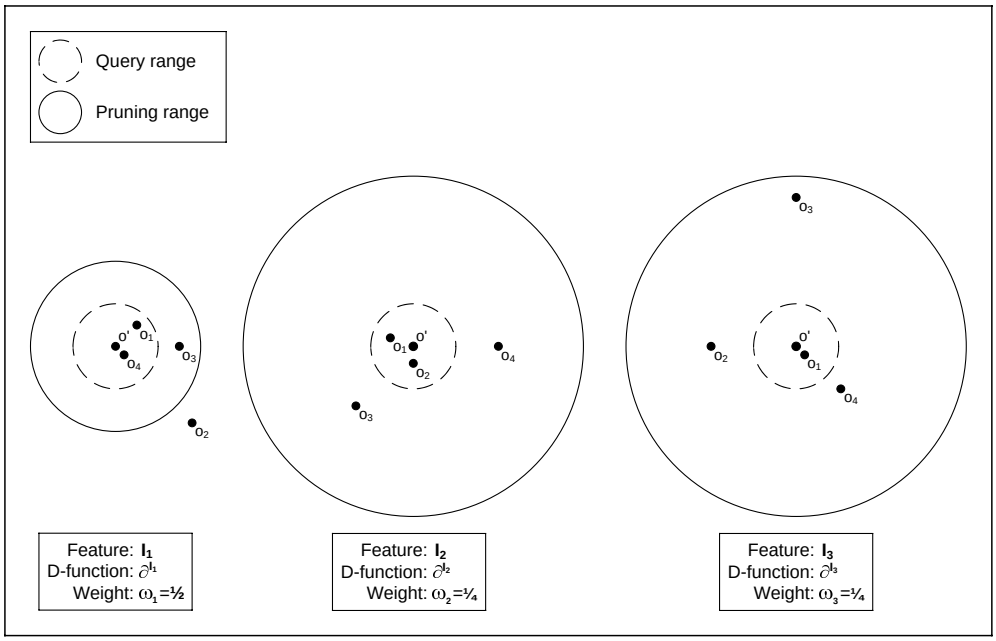

Figure 3.1: *Visual representation of a multi-feature range query using three indices on a set containing four objects. Shown is the query and pruning ranges.*

Chapter 4

M-Grid

4.1 Overview

In this chapter we describe the M-Grid [2] index structure in details. This includes the theory behind the different parts of the structure, the implementation designed as a framework, issues with some of the architectural components, and finally optimizations added to the structure initialization and query evaluation not mentioned in the original proposal.

After providing the motivation for the M-Grid index structure in Section 4.2, three sections follows that present the theory behind the M-Grid and the query types it supports.

In Section 4.3 the metric property is presented, and an important pruning theorem is introduced that provides the basis for the correctness of the M-Grid. The M-Grid itself and all its different parts are formally defined in Section 4.5. The queries supported by the M-Grid are formally defined in Section 4.6 using the theory from the previous sections.

Moving on to the implementation, Section 4.7 presents the design principles used and argues for why a flexible framework is a good idea for this implementation. Section 4.8 introduces the architecture of the implementation and describes the initialization process using the components of the architecture. Being familiar with the architecture, issues specific to each of the components, are described in the following sections: Pivot points (Section 4.9), clustering (Section 4.10) and block handling (Section 4.11).

Having described both initialization and querying of the M-Grid, adding and deleting objects is described in Section 4.12.

Section 4.13 describes some further optimizations not proposed in the original M-Grid article [2].

4.2 Motivation

The motivation for the M-Grid is the presence of high-dimensional feature vectors such as those described by the MPEG7 standard [15]. If all feature vectors are extracted as specified in the standard, the size of the vectors can become quite large – in the area of 180,000-430,000 real numbers for a standard 3 minute piece of music[1]. It has been shown that index structures such as the R-Tree and its derivatives suffers from the "dimensionality curse" [16]. Therefore another approach such as mapping to a lower dimensional space is desirable in order to avoid the curse.

Furthermore, studies have shown different results regarding the matter, what is "similar," since this notion is very subjective, and highly dependent on each individual [17]. Therefore we assume that the feature vectors extracted from the songs are representative for the music, i.e., similar songs have feature vectors that lies close to each other in the vector space compared to non-similar songs. Using this assumption a metric distance function can be used to calculate the distance between two feature vectors giving a notion of their similarity. Furthermore the metric distance function also guarantees total recall and precision [18].

4.3 Metric Property

In order to answer similarity queries, existing index structures such as the M-Grid [2], the M-Tree [16] and the Slim-Tree [19], all rely on indexing the similarity between the objects to be queried. The similarity between the objects is specified by a total metric distance function, ∂, with the following properties, where $o, o', p \in \mathbb{S}$.

[1] Using 10-24 dimensions and a sample length of 10ms.

1. Symmetry:
   $\partial(o,p) = \partial(p,o)$
2. Non-negativity:
   $o \neq p \implies \partial(o,p) > 0 \bigwedge \partial(o,o) = 0$
3. Triangular inequality:
   $\partial(o,o') \leq \partial(o,p) + \partial(p,o')$

The triangular inequality property is the most interesting one, since it enables us to avoid distance calculations under some given circumstances, which we will explore in more detail. If we reformulate the triangular inequality assuming that a reference object (pivot point) p exists and that the distance from all objects to p is known, we reach the following theorem.

Theorem 4.1 (Pruning Criterion (e.g., [20])) *Given a range query with query object o' and range r, no object o satisfying the following inequality using a reference point p belongs to the result:*

$$\left| \partial(o',p) - \partial(o,p) \right| > r$$

This theorem is illustrated in Figure 4.1 that shows a reference point p, a query point o', three objects o_1, o_2, o_3, the query range r, and the pruning radii $\partial(o',p) - r$ and $\partial(o',p) + r$.

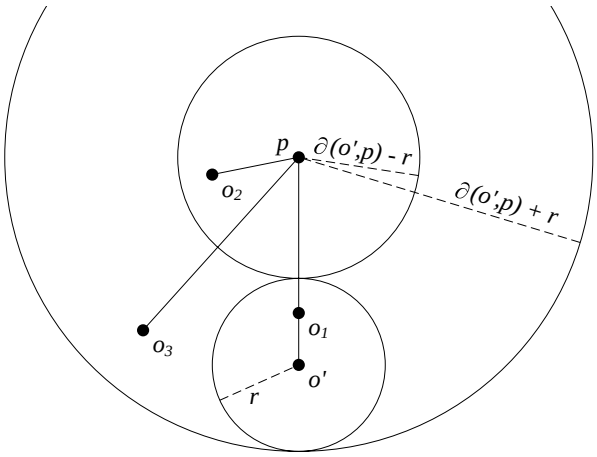

Figure 4.1: *Pruning of objects in a range query evaluation using pruning radii $\partial(o',p) - r$ and $\partial(o',p) + r$.*

Theorem 4.1 is satisfied for an object o if and only if o lies outside the two pruning radii. Clearly, object o_1 does not satisfy the inequality and cannot be pruned – and it should not, since it is within the query range. Object o_2 is clearly outside both the query and the pruning radii, and it does satisfy the equation, and can therefore be pruned. The last object, o_3, is between the pruning radii, so it cannot be pruned.

So, given that the distances from every object o to a pivot point is known, Theorem 4.1 can be used in the filtering process during query evaluation. It maintains the recall property, but to gain precision, refinement has to be performed on the result of the filtering process.

The absence of the triangular inequality in effect would mean that no objects could be pruned without actually applying the distance calculation. Even though, normally, the I/O cost is of an magnitude larger than the computational cost, existing studies have shown that computational cost can exceed the I/O cost. This happens when the feature vectors are of high dimension, i.e., when greater than ten [16]. As the dimensionality of the feature vectors increases, the ratio between the distances of the nearest of farthest neighbor decreases. This is the case for feature vectors such as MPEG7 [15], MFCC [21], and MAR [22].

4.4 Informal Presentation

By using Figure 4.2, we will give an informal introduction to the definitions and terms used later in this paper. The figure shows "a part of" a space that we will refer to as the original vector space $\mathbb{S}$. The objects $o_3, \ldots, o_{12}$ and p_1, p_2 are located in this space. The objects p_1 and p_2 are special reference objects that we will refer to as pivot points.

Every object $o \in \mathbb{S}$ can be mapped into a new space denoted pivot space $\mathbb{P}$, where the values of the new object o' is formed by distances to each of the pivots.

The pivot points are special because circles are created originating from these, and all distances from each object to each pivot are known (e.g., $\partial(o_9, p_1)$). Two adjacent circles from a pivot forms what is referred to as a ring. The intersection of rings from each pivot form regions.

The regions can also be expressed in terms of the ring number for each pivot they are intersections of. This representation is denoting cells in pivot space. An example is the cell denoted $\langle 2,3 \rangle$ in pivot space, representing regions in vector space, formed by the intersections of the second and third ring of pivot p_1 and p_2 respectively.

Each cell is attributed with a point, referred to as the cell center, which is used when a distance that involves the cell needs to be computed – e.g., the point $c_{\langle 2,3 \rangle}$ is the center of cell $\langle 2,3 \rangle$.

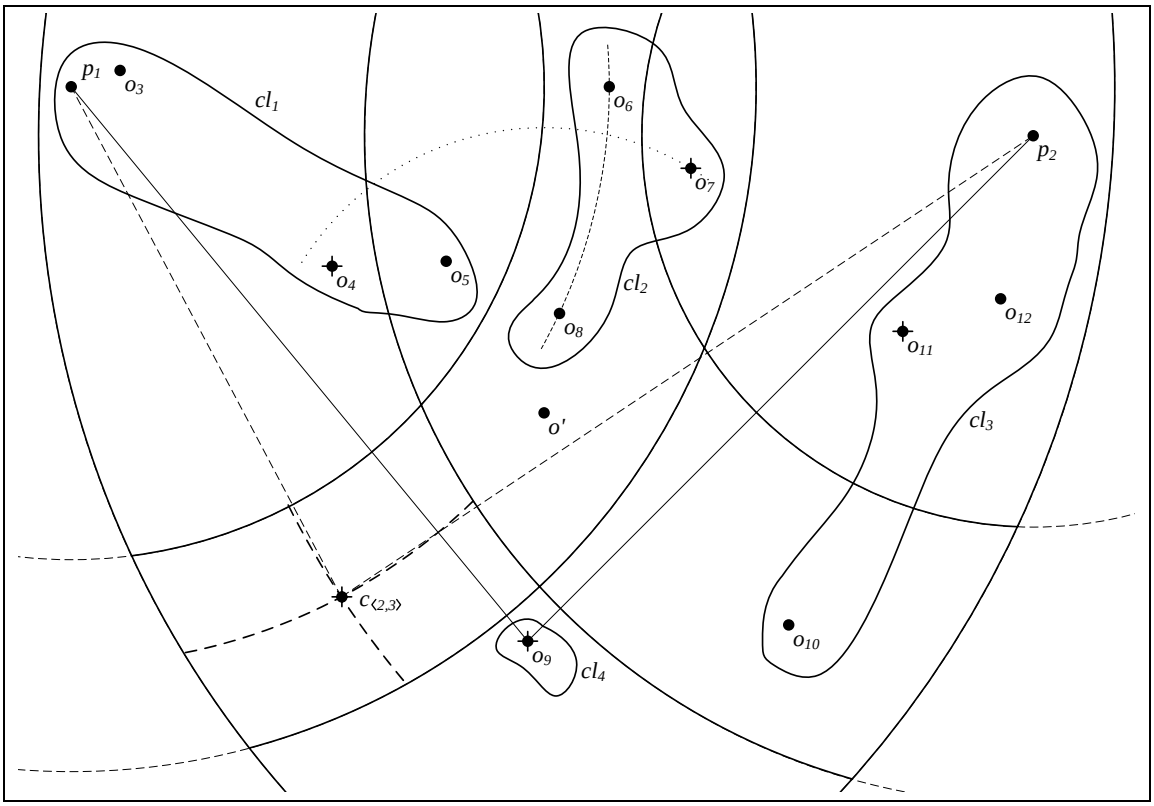

Figure 4.2: *Introducing the notations used in the M-Grid.*

All shown objects are partitioned into clusters, where close-by objects are placed in the same parts, which will be denoted as a cluster. The shown clusters $cl_1, \ldots, cl_4$ are formed such that all objects from the same cell are placed in the same cluster, and these clusters can span multiple cells (e.g., cluster cl_1). Since multiple objects exists in a cluster, a representative object of the cluster can be marked. This object is denoted the centroid (e.g., cluster cl_1 has o_4 as its centroid).

Assuming all distances from all objects $o_3, \ldots, o_{12}$ to the pivots p_1 and p_2 are known, these distances can be used to approximate the distance between any two objects. This new distance in the pivot space is denoted ∂^m. Using the difference of pivot distances as a distance measure in the pivot space, it is seen in the figure that the distance in pivot space between o_6 and o_8 using p_1 is 0, but the distance in pivot space between o_5 and o_8 in close to the distance in vector space. The distance in pivot space depends highly on which pivot is used as reference point. Therefore the largest value computed is used as the value of ∂^m for the distance between a *cell* and a *cluster*. The underlying theory of the M-Grid heavily relies on this simple fact.

4.5 Definitions

An M-Grid G is formally defined as a seven-tuple $(S, P, \partial, Rings, Cells, Clusters, CCArray)$ where:

- $S = \{o_1, \ldots, o_n\} \subseteq \mathbb{S}$ is a set of objects.
- $P = \langle p_1, \ldots, p_k \rangle \subseteq S$ is a finite list of reference points (pivots) used for mapping objects in S to the pivot space $\mathbb{P}$.
- $\partial : \mathbb{S}^2 \mapsto \mathbb{R}_+$ is a distance function having the metric property as defined in Section 4.3.
- *Rings* defines a list of $m+1$ circles for each of the k pivots all having center in the pivots specifying m pivot rings.
- *Cells* is a finite set of well defined regions.
- *Clusters* is a finite list of sets partitioning the objects $o \in S$ into logical clusters.
- *CCArray* is a relation over cells, clusters, and a Boolean that indicates whether the cluster has one or more objects in the cell.

The domain of all M-Grids is denoted $\mathbb{G}$. In the following the last four parts of the M-Grid tuple are described into more detail and formally defined.

4.5.1 Metric Mapping

The objects in the original high-dimensional vector space S are mapped into a new k-dimensional pivot space $\mathbb{P}$ using the pivots P and the distance function ∂. Notice that the number of pivots defines the number of dimensions in the new space.

The function $F_{metric} : \mathbb{S} \mapsto \mathbb{P}$ maps the objects $o \in S$ into the k-dimensional space formed by the pivots, where the i^{th} dimension value is the distance to the i^{th} pivot. Formally:

$$F_{metric}(o) = \langle \partial(o, p_1), \ldots, \partial(o, p_k) \rangle \tag{4.1}$$

4.5.2 Pivot Rings

The mapping of each object $o \in S$ into the pivot space $\mathbb{P}$, using F_{metric}, allows us to partition the objects. This involves choosing m rings for each pivot $p \in P$ such that each ring partitions the pivot space into regions having an amount of objects in the interval $\left[\lceil \frac{n}{m} \rceil - (\delta + 1)\ ,\ \lceil \frac{n}{m} \rceil + \delta\right]$ where $n = \|S\|$ and δ a margin. By using this partitioning, the objects are divided roughly into equally large parts, where we assume that objects have unique distances to the pivots. Formally:

$$\begin{aligned} &Rings = \langle \langle r_{10}, r_{11}, \ldots, r_{1m} \rangle, \ldots, \langle r_{k0}, r_{k1}, \ldots, r_{km} \rangle \rangle \\ &\quad where\ \forall i \in \{1, \ldots, k\} \Big(\\ &\qquad r_{i0} = 0\ \wedge\ r_{im} = \max_{o \in S}(\partial(o, p_i))\ \wedge \\ &\qquad \forall j \in \{1, \ldots, m-1\} \big(\\ &\qquad\quad \left\lceil \frac{n}{m} \right\rceil - (\delta + 1) \leq \\ &\qquad\quad \left\| \{ o \in S \mid r_{ij-1} \leq \partial(o, p_i) \leq r_{ij} \} \right\| \leq \left\lceil \frac{n}{m} \right\rceil + \delta \big) \Big) \end{aligned} \tag{4.2}$$

In the equation we make a special case for the lowest radius r_{i0}, which is set to zero. This is done in order to adhere to our definition of a ring earlier such that a ring i is formed by the radii r_i and r_{i-1}. Furthermore the largest ring r_{im} is also set to the largest distance between any two objects in $o, o' \in S$, in order to assure that all objects are covered.

4.5.3 Cells

In the vector space an intersection of k rings, one from each pivot $p \in P$, forms a region. Each region maps to a corresponding cell in pivot space $\mathbb{P}$. The cell being mapped to is represented by the ring number for each pivot.

By using the rings just expressed, it is now possible to define a cell, which is simply the intersection of k rings, one from each pivot $p \in P$. Each cell has a lower and an upper bound circle for each pivot ring, i.e., each cell is delimited by $2 \cdot k$ circles. A cell is represented by the ring number for each pivot.

The domain of cells using this representation is denoted $\mathbb{C}$. The cells are formally defined below, where m is the number of rings per pivot:

$$Cells = \left\{ \langle c_1, \ldots, c_k \rangle \mid c_1, \ldots, c_k \in \{1, \ldots, m\} \right\} \tag{4.3}$$

Figure 4.3 shows an interesting property of the cells generated by Equation 4.3. The shaded regions in the original vector space can be uniquely identified. But mapping the regions into cells in the pivot space removes this uniqueness, since both regions are mapped into the same cell, identified by the pivot rings. Therefore the cells in pivot space forms what is called a pseudo-grid. No regions are mapping to the cells $\langle 1,1 \rangle$, $\langle 1,2 \rangle$ and $\langle 2,1 \rangle$, since no intersections exist for these combinations of the pivot rings. Furthermore the areas spanned by the dashed lines are not considered as regions since a valid region requires that there is an intersection of pivot rings stemming from all k pivots.

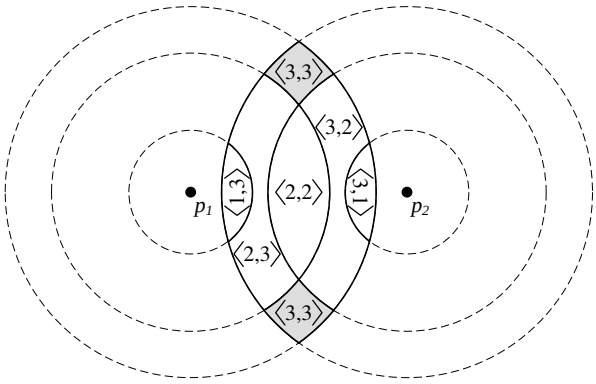

Figure 4.3: *Example where two regions in vector space are mapped into the same cell in pivot space – $\langle 3,3 \rangle$.*

From the above description and example we make the following observation:

Observation 4.1 (The $\mathbb{S} \mapsto \mathbb{P}$ mapping is irreversible) *More than one region in vector space $\mathbb{S}$ can map to the same cell in pivot space $\mathbb{P}$.*

Observation 4.2 (The $\mathbb{S} \mapsto \mathbb{P}$ mapping is not onto) *There may be one or more cells in the set Cells $\subseteq \mathbb{P}$ that no region in vector space $\mathbb{S}$ maps to.*

Having defined *Cells*, a function mapping from objects $o \in S$ to the cell containing the object in the pivot space is needed. In other words a function with the signature: $F_{cell} : \mathbb{S} \mapsto \mathbb{C}$, expressed formally as:

$$\begin{aligned} &F_{cell}(o) = \langle c_1, \ldots, c_k \rangle \textit{ where the } c_i \textit{ is given as follows} \\ &\quad r_{ic_{i-1}} < d_{p_i} \leq r_{ic_i}, \textit{ where} \\ &\quad\quad F_{metric}(o) = \langle d_{p_1}, \ldots, d_{p_k} \rangle \end{aligned} \tag{4.4}$$

In order to define the distance of an object o in the pivot space $\mathbb{P}$ to a cell, we define a point in the cell to be used as a reference point. The point chosen is the center of the cell, found with a function having the signature: $F_{center} : \mathbb{C} \mapsto \mathbb{P}$. Such a function is defined below:

$$\begin{aligned} F_{center}\big(\langle c_1, \ldots, c_k \rangle\big) = \; &\langle d_{p_1}, \ldots, d_{p_k} \rangle, \textit{ where} \\ &d_{p_i} = \frac{r_{ic_i} + r_{ic_{i-1}}}{2} \end{aligned} \tag{4.5}$$

4.5.4 Clusters

In order to connect the cells with close-by objects, we introduce a new list of sets, denoted *Clusters*, that partitions the objects. The domain of all possible clusters is denoted $\mathbb{K} = \{c \mid c \subseteq \mathbb{S}\}$.

For the M-Grid to behave correctly, a number of post-conditions to Equation 4.6, defined below, have to be satisfied. As long as these conditions are satisfied then how the clusters are formed does not effect the correctness of the M-Grid. In order to increase the performance, close-by objects should be placed in the same cluster. The formal definition of the clusters and the post-conditions follows:

$$\textit{Clusters} = \langle cl_1, \ldots, cl_h \rangle \textit{ where } \bigcup_{i=1}^{h} (cl_i) = S \tag{4.6}$$

Non-overlapping clusters
Equation 4.7 states no two clusters include the same object. This requirements is to ensure the correctness.

$$\forall c_1, c_2 \in \textit{Clusters}\big(c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset\big) \tag{4.7}$$

All objects clustered
Equation 4.8 ensures that objects $o \in S$ are all indexed and therefore can be retrieved at a later stage, when performing a query.

$$\forall o \in S \; \big(\exists c \in \textit{Clusters}\,(o \in c) \big) \tag{4.8}$$

To determine the distance to a cluster, a representative object for the cluster is needed – the "centroid." For that, a function with the signature $F_{centroid} : \mathbb{K} \mapsto \mathbb{S}$ is needed. This is expressed in the following equation, which picks an object o from a cluster with the minimal total distance to the other objects of that cluster. By doing so, an object which lies in the "center" of the cluster is chosen.

$$\begin{aligned} &F_{centroid}(\textit{cluster}) = o \in \textit{cluster, where} \\ &\nexists o' \in \textit{cluster}\Big(\sum_{o^* \in \textit{Clusters}} \partial(o', o^*) < \sum_{o^* \in \textit{Clusters}} \partial(o, o^*) \Big) \end{aligned} \tag{4.9}$$

4.5.5 Closeness in Pivot Space

We need to relate the *Cells* and *Clusters* defined in the previous sections, making it possible to define which *cluster* is closest to a given *cell*. A problem arises because *Cells* lies in the pivot space while *Clusters* lies in the vector space. Since a mapping of $\mathbb{S} \mapsto \mathbb{P}$ is irreversible (see Observation 4.1), the only option is to map *cluster* into the pivot space.

In order to find the distance between objects in pivot space, we cannot use ∂ and therefore need a new distance function $\partial^m : \mathbb{P}^2 \mapsto \mathbb{R}_+$. To motivate this we use Figure 4.2, where we only designate p_1 as pivot and only consider objects o_6 and o_8. By using only one pivot, the pivot space is simply a one-dimensional space where values are distances to p_1. It can be seen that the objects o_6 and o_8 lie at the same distance from p_1, i.e., $\partial(o_6, p_1) = \partial(o_8, p_1)$. If we instead choose pivot p_2, it can be seen that $\partial(o_6, p_1) \neq \partial(o_8, p_1)$. So choosing pivot p_2 is more favorable in the given case.

The distance in the pivot space puts a lower bound on the original distance in the vector space. In order to get the tightest bound, the maximum difference of the pivot distances, cf. Theorem 4.1, is chosen as ∂^m as defined in the following:

$$\begin{aligned} &\partial^m\big(\langle m_{11}, \ldots, m_{1k} \rangle, \langle m_{21}, \ldots, m_{2k} \rangle\big) = \\ &\quad \max_{i=1}^{k} (m_{1i} - m_{2i}) \end{aligned} \tag{4.10}$$

s Using this new distance function, it is possible to define a function that, given a cell, can map to the closest cluster. The signature for this function is: $F_{closest} : \mathbb{C} \mapsto \mathbb{K}$.

The following definition ensures that cells intersecting a cluster are pointing to the that cluster, otherwise the ∂^m function is used to determine the closest cluster:

$$
\begin{aligned}
&F_{closest}(cell) = cluster \in Clusters \text{ where} \\
&\quad F_{repr}(cell, cluster) \vee \\
&\quad \neg F_{repr}(cell, cluster) \wedge \nexists cluster' \in Clusters \Big(\\
&\qquad \partial^m \left(F_{metric} \left(F_{centroid}(cluster') \right), F_{center}(cell) \right) < \\
&\qquad \partial^m \left(F_{metric} \left(F_{centroid}(cluster) \right), F_{center}(cell) \right) \Big)
\end{aligned}
\tag{4.11}
$$

4.5.6 CCArray

Being able to estimate the distance between a *cell* and a *cluster* allows us to improve the execution of a range query. In order to know which clusters to investigate when evaluating a range query, a connection between *Cells* and *Clusters* is needed. That connection is the CCArray, which associates each $cell \in Cells$ with the $cluster \in Clusters$ that is the closest one, together with a Boolean indicating whether the cluster intersects with the cell (i.e., whether the cell is represented). This allows us to decide whether to fetch the cluster when performing a range query. If a range query is performed, all cells that are not represented need not to be visited and can therefore be pruned. For that, a predicate with the signature $F_{repr} : \mathbb{C} \times \mathbb{K} \mapsto \mathbb{B}$ is needed, where $\mathbb{B} = \{ff, tt\}$ is the domain of the Boolean values:

$$
F_{repr}(cell, cluster) = \exists o \in cluster \left(F_{cell}(o) = cell \right) \tag{4.12}
$$

Figure 4.2 shows that no clusters are represented in cell $\langle 3,3 \rangle$ but in $\langle 2,3 \rangle$ there is. The *CCArray* can now be formally defined as follows:

$$
\begin{aligned}
CCArray = \Big\{ \langle cell, cluster, repr \rangle \Big| \\
cell \in Cells \wedge \\
cluster = F_{closest}(cell) \wedge \\
repr = F_{repr}(cell, cluster) \Big\}
\end{aligned}
\tag{4.13}
$$

If the conditions set in Equation 4.7-4.8 are satisfied then the *CCArray* defined in Equation 4.13 is guaranteed to point to all h clusters. This can be formalized in the following post-condition. However, there are some more implementation-specific issues that need more elaboration, which is addressed in Section 4.10.2.

These issues are only relevant for some of the clustering algorithm used and therefore not relevant in the formal theory of the M-Grid.

Full coverage
This post-condition is formally defined as:

$$
\begin{aligned}
&\forall cluster' \in Clusters \\
&\quad \exists \langle cell, cluster, repr \rangle \in CCArray \big(\\
&\qquad cluster = cluster' \wedge repr = tt \big)
\end{aligned}
\tag{4.14}
$$

4.6 Similarity Queries

In this section we describe how to perform the similarity queries – Range, kNN and Transition – (defined in the ISA paper [1]) using the M-Grid. We have changed the semantics of the original Range operator slightly. It has been redefined to return a set of objects within the given query range.

By using the formal definition of the M-Grid, we can precisely show which cells, clusters, and distance computations are involved in the queries. This makes the cost of evaluating a query very clear. Furthermore this approach also makes the actual implementation of the queries straightforward.

After each definition, we give an implementation-specific description of the definition in a number of steps. It has been attempted to map these steps to the formal definition to the maximum extent.

4.6.1 Metric Range

A common property for all the similarity queries is a need for performing a range query in pivot space $\mathbb{P}$. This is defined in the metric range function below, which has the signature: $Range^m_{o',r}(G) : \mathbb{G} \times \mathbb{S} \times \mathbb{R}_+ \mapsto 2^{\mathbb{C}}$.

$$
\begin{aligned}
&Range^m_{o',r}(G) = \Big\{ \langle c_1, \ldots, c_k \rangle \in Cells \Big| \\
&\quad Rings = \langle \langle r_{11}, \ldots, r_{1m} \rangle, \ldots, \langle r_{k1}, \ldots, r_{km} \rangle \rangle \wedge \\
&\quad F_{metric}(o') = \langle d_{p_1}, \ldots, d_{p_k} \rangle \wedge \\
&\quad \bigwedge_{i=1}^{k} \Big(\exists s,t \in \mathbb{N}_+ \\
&\qquad \exists r_{is}, r_{it} \in \langle r_{i1}, \ldots, r_{im} \rangle \nexists r_{is^*}, r_{it^*} \in \langle r_{i1}, \ldots, r_{im} \rangle \big(\\
&\qquad\quad r_{is} \leq d_{p_i} - r \wedge r_{is^*} > r_{is} \wedge r_{is^*} \leq d_{p_i} - r \wedge \\
&\qquad\quad r_{it} \geq d_{p_i} + r \wedge r_{it^*} < r_{it} \wedge r_{it^*} \geq d_{p_i} + r \wedge \\
&\qquad\quad c_i \in [s,t] \big) \Big) \Big\}
\end{aligned}
\tag{4.15}
$$

1. Map the query object into the pivot space (line 3).
2. Construct for each pivot an interval of ring numbers for the smallest and largest ring that surrounds the range from the metric query point (lines 4–7).
3. Return all cells having ring numbers within the constructed ring number intervals (line 8).

Figure 4.4 shows an example with two pivots p_1 and p_2 and a query object o' depicted in vector space. The query range from query object o' intersects the shaded regions. The pivot ring intervals described in the second step above are found to be $[2,3]$ and $[1,3]$, and using combinatorics the cells $\langle 2,1\rangle$, $\langle 2,2\rangle$, $\langle 2,3\rangle$, $\langle 3,1\rangle$, $\langle 3,2\rangle$ and $\langle 3,3\rangle$, are returned.

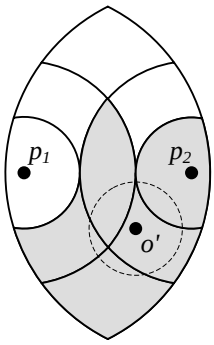

Figure 4.4: *Simple example showing a range query in pseudo-grid with two pivots depicted in vector space. The shaded regions are intersected by the query range, forming the pivot ring ranges* $[2,3]$ *and* $[1,3]$ *used in the metric range operation.*

4.6.2 Range

The definition of the range operation on an M-Grid G returns a set of objects $o \in S$ given a query object o' and a radius r. More formally, this function has the signature $Range_{o',r} : \mathbb{G} \times \mathbb{S} \times \mathbb{R}_+ \mapsto 2^{\mathbb{S}}$. The formal definition follows:

$$
\begin{aligned}
Range_{o',r}(G) = \Big\{ o \mid\ & \\
& Cells' = Range^m_{o',r}(G) \wedge \\
& Clusters' = \big\{ cluster \mid \exists \langle cell, cluster, tt \rangle \\
& \qquad \in CCArray(cell \in Cells') \big\} \wedge \\
& S' = \big\{ o \mid o \in cluster \wedge cluster \in Clusters' \wedge \\
& \qquad \partial^m\big(F_{metric}(o), F_{metric}(o')\big) \leq r \big\} \wedge \\
& o \in S' \wedge \partial(o,o') \leq r \Big\}
\end{aligned}
\tag{4.16}
$$

The steps in the definitions can be described by the following:

1. Find all $Cells'$ within the range r from the query object o' (line 2).
2. Find all $Clusters'$ pointed to by the represented $Cells'$ found (line 3–4).
3. Prune all objects that satisfy Theorem 4.1 (line 5–6).
4. Perform distance computation in the vector space $\mathbb{S}$ and prune according to the given query range r (line 7).

4.6.3 kNN

The definition of the kNN operation on an M-Grid G returns a list of the k nearest objects $o \in S$ given a query object o' and a number k. The signature of the function is: $kNN_{o',k} : \mathbb{G} \times \mathbb{S} \times \mathbb{N}_+ \mapsto 2^{\mathbb{S}}$. The formal definition given below deviates slightly from the implementation. The reason is that the following definition shows what the final result is and which cell, cluster, and object distances need to be computed in order to reach the result. The formal definition of the KNN operation follows:

$$
\begin{aligned}
& KNN_{o',k}(G) = \langle o_1, \ldots, o_k \rangle \ \textit{where} \\
& \quad r = \partial(o', o_k) \wedge \\
& \quad Cells' = Range^m_{o',r}(G) \wedge \\
& \quad Clusters' = \big\{ cluster \mid \exists \langle cell, cluster, repr \rangle \\
& \qquad \in CCArray\big(cell = F_{cell}(o') \vee \\
& \qquad\qquad cell \in Cells' \wedge repr = tt\big) \big\} \wedge \\
& \quad S' = \big\{ o \mid o \in cluster \wedge cluster \in Clusters' \wedge \\
& \qquad\qquad \partial^m(F_{metric}(o'), F_{metric}(o)) \leq r \big\} \wedge \\
& \quad F^{order}_{o',\partial,\langle o_1,\ldots,o_k\rangle}(S')
\end{aligned}
\tag{4.17}
$$

The implementation of the kNN operation is more complex than the Range operation because it consists of two parts: the fetching of the nearest *cluster* and a Range query in order to avoid false dismissals.

The distance of the k^{th} object in the nearest *cluster* (wrt. the query object o') is used as an initial radius for the following range query. The reason why this second step is needed after the k^{th} element is found in the nearest *cluster* is that there may be other clusters containing objects within the distance of the k^{th} element. Always performing the range query, ensures the correctness of the algorithm, i.e., no false dismissals occur.

The following steps concerns processing of the nearest *cluster*:

1. Find the *cell* that contains the query object o' via $F_{cell}(o')$.
2. Find the closest *cluster* by looking up in the CCArray using *cell*.
3. If k elements are not found in the nearest *cluster*, range queries with increasing radii are performed until at least k elements are found.

The second part consists of a Range query, where a list of candidate clusters are traversed. After investigation of a cluster in the list, the radius can possibly be tightened, implying that some clusters can be pruned from the list of candidate clusters:

1. Maintain a *result* list of the k best objects.
2. Let r be the largest distance in *result*.
3. Maintain a list of *Clusters* within radius r.
4. Visit a single *cluster* in *Clusters* and remove it from the list (see Section 4.13.3 for further details on the impact of the ordering of the clusters in the list).
5. Update *result* and r.
6. Repeat step 4-5 until the list of *Clusters* is empty.

When the algorithm terminates, the *result* list contains the k nearest objects from o'.

4.6.4 InBetween

Regardless of the kind of Transition, we need a notion of an object o being near the line between two objects o_1 and o_2. This is computed wrt. a given distance function ∂. The signature for this operation is $InBetween_{S,o_1,o_2}(G) : \mathbb{G} \times 2^{\mathbb{S}} \times \mathbb{S}^2 \mapsto \mathbb{S}$. Formally:

$$InBetween_{S,o_1,o_2}(G) = o \in S \text{ where } \nexists o' \in S \left(\partial(o',o_1)+\partial(o',o_2) < \partial(o,o_1)+\partial(o,o_2)\right) \quad (4.18)$$

An example is shown in Figure 4.5 where the *InBetween* operation is used to find the object $o \in \{o_1,\ldots,o_4\}$ being "most in between" objects o' and o''. According to the definition, this means the object having the shortest total distance to both o' and o'', where the minimum obviously is the distance between those two objects (the dashed line).

Assuming $\partial(o',o'') = 1$ the total distances from each of the objects $o_1,\ldots,o_4$ are 1.022, 1.118, 1.005 and 1 respectively. Looking at the figure it can be seen that the object o_4 is the best since it is placed on the path between o' and o''. Comparing o_1 and o_3 that have the same orthogonal distance to the dashed line, o_3 is being favored since it is closer to the middle of the dashed line — thereby having the shortest total distance. But even though o_2 and o_3 have equal horizontal distance to each of the objects o' and o'', still o_3 is favored since it is closer to the dashed line.

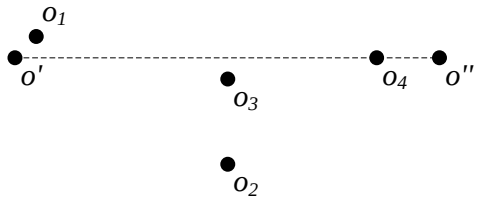

Figure 4.5: *Example showing the principle of the InBetween operation choosing which object $o \in \{o_1,\ldots,o_4\}$ is most in between o' and o''. The best candidate is o_4, followed by o_3, o_1 and last o_2.*

4.6.5 Annulus

As a prerequisite to defining the Transition operator described in Section 4.6.6 and 4.6.7, we introduce a new function that returns a set of cells within and intersecting a given upper and lower bound using the Metric Range function, $Range^m$. This function has signature: $Annulus^m_{o',r_{max},r_{min}}(G) : \mathbb{G} \times \mathbb{S} \times \mathbb{R}^2_+ \mapsto 2^{\mathbb{C}}$. Formally:

$$Annulus^m_{o',r_{max},r_{min}}(G) = \Big\{ cell \mid cell \in \Big(Range^m_{o',r_{max}}(G) \setminus Range^m_{o',r_{min}}(G)\Big) \bigcup \Big(Range^m_{o',r_{max}}(G) \bigcap Range^m_{o',r_{min}}(G)\Big) \Big\} \quad (4.19)$$

1. In line 2 we find all cells intersecting the upper bound, and all cells within the bound set by r_{max} and r_{min}, but not intersecting with range r_{min}.
2. Since we also need the cells intersecting r_{min}, we address this issue in line 3.

4.6.6 TransR

In the following we describe the "Evolving Transition" operation that was presented in the ISA [1]. The Transition operation produces a list of songs, that has a smooth transition between each successive pair of songs in the list. The smoothness is determined by criteria set by the user.

The operation has been modified to only accept two seed songs denoted o_1 and o_j that represents the start and an end song respectively. The operation computes a list of songs $\langle o_1, \ldots, o_j \rangle$, and it guarantees that the total distance formed by the path in the list from the start to end song is minimized. Furthermore, the distance between every two successive songs is between the two given ranges r_{min} and r_{max}.

The signature for of the transition operation is $TransR_{o_1,o_j,r_{min},r_{max}}(G) : \mathbb{G} \times \mathbb{S}^2 \times \mathbb{R}_+^2 \mapsto 2^{\mathbb{S}}$. Formally:

$$
\begin{aligned}
&TransR_{o_1,o_j,r_{min},r_{max}}(G) = \langle o_1, \ldots, o_j \rangle \ \textit{where} \\
&\bigwedge_{i=1}^{j-2} \Big(Cells' = Annulus^m_{o_i,r_{min},r_{max}}(G) \cap \\
&\qquad Annulus^m_{o_{i+2},r_{min},r_{max}}(G) \bigwedge \\
&\quad Clusters' = \big\{ cluster \mid \\
&\quad \exists \langle cell, cluster, tt \rangle \in CCArray(cell \in Cells') \big\} \wedge \\
&\quad S' = \big\{ o \mid o \in cluster \wedge cluster \in Clusters' \wedge \\
&\qquad \bigwedge_{j \in \{0,2\}} r_{min} \leq \partial(o_{i+j}, o) \leq r_{max} \big\} \wedge \\
&\quad o_{i+1} = InBetween_{S',o_i,o_{i+2}}(G) \Big)
\end{aligned}
\tag{4.20}
$$

The equation only defines the output — not the actual implementation of the operation.

The implemented algorithm that reflects the definition can be separated into two cases, where the base case can be used to explain the equation. The base case occurs when there is a song within the ranges r_{min} and r_{max} from both o_1 and o_j. The recursive step occurs when no song exists that satisfy the ranges. Let us briefly describe the base case:

1. Find all *Cells* within the ranges r_{min} and r_{max} for both songs o_1 and o_j (lines 2–3).
2. For all the found *Cells*, fetch the corresponding *Clusters* (lines 4–5).
3. Prune all objects outside the ranges given by r_{min} and r_{max} for both o_1 and o_j (lines 6–7).
4. Find the song that lies most in between o_1 and o_j by minimizing the total distance (line 8).

This base case ensures the termination of the algorithm. If no in common cells are found, we need to perform the recursive step, which finds a song o_m most in between o_1 and o_j and recursively apply the transition algorithm on both o_1, o_m and o_m, o_j:

1. Find all *Cells* "in the middle" by using the metric range query with $r = \frac{dist(o_1,o_j)}{2}$ on both o_1 and o_j.
2. For all common *Cells*, fetch the corresponding *Clusters*.
3. Find the song o_m which lies most in between o_1 and o_j.
4. Perform the transition recursively with $TransR_{o_1,o_m,r_{min},r_{max}}$ and $TransR_{o_m,o_j,r_{min},r_{max}}$ and concatenate the results.

This recursive step can be extensive wrt. both time and I/O and even though the algorithm does terminate, it does not guarantee that any results are produced if the given restriction on the ranges cannot be satisfied.

4.6.7 TransK

Here we describe the "Sized Transition" operation that determines a smooth transition with a fixed number of songs. The number k specifies how many songs there should be between the given start and end song o_1 and o_j. The ability to give a number k as parameter instead of a range $[r_{min}, r_{max}]$, gives a more intuitive way of querying for a transition, since no knowledge about the underlying features is required (such as the distribution of the objects in the vector space). The choice of features will influence the distances between the songs and choosing "correct" ranges may not be straightforward.

To avoid this particular problem, we provide a new operation *TransK* taking two seed songs o_1 and o_j and a number k. The operation that returns a list of songs, has the signature: $TransK_{o_1,o_j,k} : \mathbb{G} \times \mathbb{S}^2 \times \mathbb{N}_+ \mapsto 2^{\mathbb{S}}$. Formally:

$$
\begin{aligned}
&TransK_{o_1,o_j,k}(G) = \langle o_1, \ldots, o_j \rangle \ \textit{where} \\
&\bigwedge_{i=1}^{k} \Big(r = \frac{\partial(o_i, o_{i+2})}{2} \wedge \\
&\quad Cells' = Range^m_{o_i,r}(G) \cap Range^m_{o_{i+2},r}(G) \wedge \\
&\quad Clusters' = \big\{ cluster \mid \exists \langle cell, cluster, repr \rangle \\
&\qquad\qquad \in CCArray(cell \in Cells') \big\} \wedge \\
&\quad S' = \big\{ o \mid o \in cluster \wedge cluster \in Clusters' \big\} \wedge \\
&\quad o_{i+1} = InBetween_{S',o_i,o_{i+2}}(G) \Big)
\end{aligned}
\tag{4.21}
$$

As previously, the implementation and the above formal definition differs. The key idea is the same as in Equation 4.20, which is to use a divide-and-conqueror approach to find a smooth transition. The base case occurs when $k = 0$, and the recursive case when $k > 0$, which is described in the following.

In order to generate a good result, we need to distinguish between when k is even and odd because when k is even the result of $\frac{k+1}{2}$ is not a natural number.

- Odd
  1. Find all *Cells* "in the middle" by using the metric range query with $r = \frac{\partial(o_1,o_j)}{2}$ on both o_1 and o_j.
  2. For all common *Cells*, fetch the corresponding *Clusters*.
  3. Find the song o_m in the "middle" between o_1 and o_j.
  4. Recursively apply *TransK* to o_1, o_m and o_m, o_j, using $k = \frac{k-1}{2}$ in both transitions.
  5. Concatenate the results.
- Even
  1. Compute the offset factors $f_1 = \frac{k}{2\cdot(k+1)}$ and $f_2 = 1 - f_1$.
  2. Find all *Cells* "in the middle" by using the metric range query with $r_1 = \partial(o_1,o_j)\cdot f_1$ and $r_2 = \partial(o_1,o_j)\cdot f_2$ on o_1 and o_j.
  3. For all common *Cells*, fetch the corresponding *Clusters*.
  4. Find the song o_m in the "middle" between o_1 and o_j.
  5. Recursively apply *TransK* to o_1, o_m and o_m, o_j, using $k = \frac{k}{2} - 1$ and $k = \frac{k}{2}$ respectively.

4.7 Design Principles

The original implementation of the M-Grid [2] was solely done as a proof of concept and was not intended to be reusable. This makes it difficult to explore and optimize the M-Grid by testing other ideas such as different clustering algorithms, as suggested in the original article [2]. The authors main purpose was merely to test and verify the theory, and no consideration has been given to re-usability or extendability.

In our work we extend the M-Grid with support for the similarity algebra we described in our previous work [1]. Furthermore we also incorporated the M -Grid into the existing framework XXL [3], which is an OO implementation of tools that allow fast prototyping in Java. We do so because we believe that ideology behind XXL, such as reproducibility and extendability is necessary. This is mainly because audio similarity indexing is a changing research area with a need for quick implementations of new theories.

Some problems arose as described in Section 4.10, which has led to us implement requirements explicitly in the source code for the M-Grid to prohibit the problems. We have done so by using the concept of design by contract [23]. As an example we have have implemented the full-coverage requirement as a post-condition that all clustering classes automatically posses by inheritance. Since this enforcement (and the others) are quite time consuming, this check is run-time configurable in order to avoid this overhead in a production system.

4.8 Construction of the M-Grid

In this section, we describe how the theory presented in Section 4.5 is implemented by presenting an overview of the entire M-Grid framework. Many of the equations have been mapped to specific components, and some new components have been introduced in order to improve flexibility and performance.

A picture of the framework can be seen in Figure 4.6. In this figure, the components marked with a black triangle denoted components that can be replaced or extended. As can be seen, many components can be changed, which provides much flexibility when different algorithms need to be tested. The dashed boxes denotes helper components that are only present in order to improve performance. In the following, we describe the different components in the order they appear in the construction of the M-Grid.

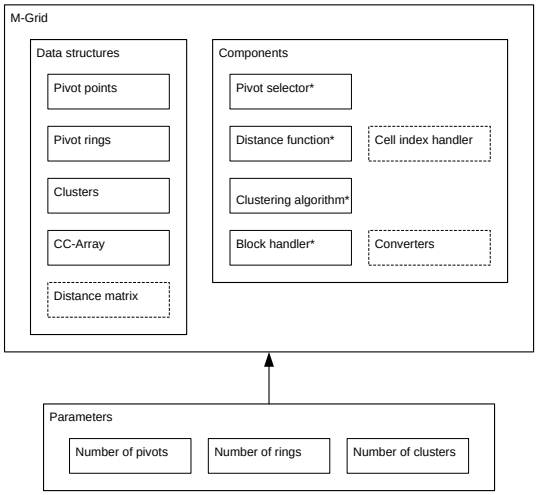

Figure 4.6: *Overview of the M -Grid framework. The components, with an * attributed, denotes changeable components, whereas the dashed ones denote components used only for optimization. The dashed arrow indicates parameters used for the M-Grid construction.*

Parameters The parameter box encapsulate the configuration used for the M-Grid, that is used as input to the M-Grid as denote by the arrow in the figure. It contains such parameters as the number of pivot points, rings, and clusters. These parameters are used throughout the entire construction of the M-Grid.

Distance Function Before the construction of the M-Grid can commence, a distance function needs to be defined. This component encapsulates such a distance function and can be implemented in different ways as long as the metric properties specified in Section 4.3 are satisfied.

Pivot Selector This component selects the most suitable pivots from the entire set of objects. It can perform an exhaustive search through all objects or select a random subset and evaluate these. This option is described in further detail in Section 4.9.2. After the selection of pivots, rings can be formed that partition the vector space. Furthermore, the distance calculations performed in this step, are stored in the Distance Matrix, so that redundant computations are avoided if other components also need to perform some of the same computations.

Clustering Algorithm After the selections of the pivots, the objects need to be clustered. Different criterias can be used in order to determine the clusters, such as the maximal amount of objects in a cluster, the total distance from the centroids to the objects in the cluster, etc. However, the requirements specified by the post-conditions in Section 4.5.4 need to be satisfied. The Distance Matrix is also used in this step.

CCArray The connection between *Cells* and the closest *cluster* $\in$ *Clusters* is maintained by the CCArray. This is the underlying data structure of the M-Grid, and this cannot be changed. This data structure makes use of the Cell Index Handler, which enables fast look-up of *Cells* in the CCArray. The hashing algorithm that provides this with $\Theta(1)$ complexity is described in further detail in Section 4.13.2.

Block Handler When the construction of the M-Grid is done, the generated *Cells*, *Clusters* and entries in the CCArray need to stored on disk. This is handled by the Block Handler component, which can be extended/changed to store the contents in different files if it is more efficient to do so. In order to save the contents, it uses different Converter components to write the contents to the disk.

4.9 Pivots

In this section, we describe how to select some pivots (also referred to as vantage points in the literature) that are used as reference points in the construction of the M-Grid. The choice of pivots influences the run-time performance of the M-Grid, since the number of distance calculations can be reduced in a similarity query. Therefore the selection of pivots needs consideration and a theoretical foundation for determining "suitable" pivots needs to established.

4.9.1 Pruning using Pivots

The first step in evaluating a similarity query, given a query object o', is to map the query object into the pivot space $\mathbb{P}$, where each dimension value v_i is defined by the distance to the pivot p_i. This step involves k distance computations. In order to make the M-Grid efficient we usually require that $k \ll n$ where $n = \|S\|$, so choosing this small subset is an important task. This is done to apply filtering and thereby reduce the number of expensive I/O's and distance computations. To do so, we need to redefine the Pruning Criterion (Theorem 4.1) in the presence of multiple pivots as in the following theorem:

Theorem 4.2 (Pruning Criterion with Pivots) *Given a range query on a set of objects S with query object o' and range r, all objects $o \in S$ satisfying the following can be safely pruned from the result:*

$$\partial^m \left(F_{metric}(o'), F_{metric}(o) \right) > r$$

This theorem uses Equation 4.10, which defines the distance between objects in the pivot space. Since the distance is computed wrt. each pivot where the largest value is used as the distance, we see that as long as *one* of the pivots satisfy the criterion, the object o can be safely pruned.

Figure 4.7 illustrates this situation with a range query $Range_{o',r}(G)$ on an M-Grid with two pivots and their corresponding pruning radii. Like with a single reference point, the objects o_1 and o_2 lie outside both the pruning radii and can therefore be pruned. Also o_3 and o_4 satisfies Theorem 4.2, since they are outside the pruning radii of p_2 and p_1. Only o_5 and o_6 are within the pruning radii and can therefore not be pruned, but have to go through a further refinement step.

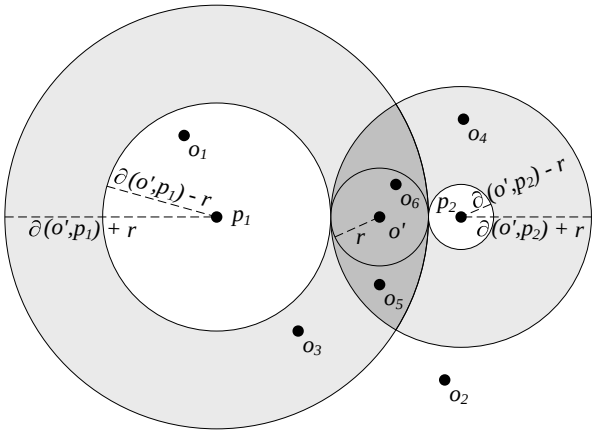

Figure 4.7: *Range query pruning using two pivots with each two pruning radii. Objects $o_1, \ldots, o_4$ are all pruned leaving only a single false-positive result object o_5, and still including the true-positive o_6.*

Using Theorem 4.2 in combination with two or more pivots increases the pruning power.

4.9.2 Pivot Selection

The pivot selection process can be summarized in the following steps:

1. Choose a subset of candidates from the original objects $o \in S$:

$$Candidates = \{o \in S \mid isCandidate(o)\} \quad (4.22)$$

where $isCandidate : \mathbb{S} \mapsto \mathbb{B}$ is a predicate determining whether an object should be considered a candidate.

2. From this subset, evaluate each candidate in turn:

$$\begin{aligned} &P = \langle p_1, \ldots, p_k \rangle \text{ where} \\ &p_1, \ldots, p_k \in Candidates \wedge \\ &\nexists p \in Candidates(\mu_S(\{p\}) > \mu_S(\{p_1\})) \wedge \\ &\bigwedge_{i=2}^{k} (\mu_S(\{p_i, \ldots, p_1\}) \geq \mu_S(\{p_{i-1}, \ldots, p_1\})) \wedge \\ &\nexists \{p_1^*, \ldots, p_k^*\} \subseteq Candidates \setminus P \Big(\\ &\quad \mu_S(\{p_1^*, \ldots, p_k^*\}) > \mu_S(P) \Big) \end{aligned} \quad (4.23)$$

where $\mu_S(P)$ specifies the efficiency of a set of pivot points (see Section 4.9.3).

Since pivot selection can be done in different ways, we provide an option in our framework for implementing new algorithms as long as they fulfill the described interface as described in the enumeration, i.e., they provide an *isCandidate* predicate.

4.9.3 Efficiency Criterion

In order to determine if some pivot p_1 is better than another pivot p_2, a decision criterion for this is defined. Usually, a pivot p_1 is better than p_2 if p_1 is an outlier [24]. A good outlier is a pivot that lies far away from the other objects in the set S, but being an outlier in itself does not guarantee a good pivot, hence the need for a efficiency criterion arises.

To increase the probability of satisfying Theorem 4.2, and thereby gain an effective filtering, the mean of the distance distribution of S, also denoted as $\mu_S(P)$ is introduced. Here P is a set of pivots and is used to discriminate the objects of set S. The pivot selection from a set of candidates maximizes the value of $\mu_S(P)$.

The pivot selection used is the Incremental Selection Algorithm (ISA), since it has been shown to be effective [24]. The algorithm is defined below in terms of the efficiency criterion:

$$\begin{aligned} \mu_S^{IS}(P) = \sum_{i=1}^{n} \max_{p \in P} \mid \partial(p, o_i) - \partial(p, o_{i-n}) \mid \\ \text{where } o_i, o_{i-n} \in S \end{aligned} \quad (4.24)$$

4.10 Clustering Algorithms

In the original M-Grid article [2], it is stated that "any clustering technique applicable to vector space" can be applied. There are however some assumptions that are not made very clear in that article. In the following we present some of the problems affecting the clustering algorithms and we define some criterias that these must fulfill in order to be used. We do so by first defining the a correctness criteria as the following:

4.10.1 Correctness

When using an index structure an important property is the correctness of the query results. By correctness is meant that the index structure maintains the recall property and if possible a high degree of precision.

The recall property informally measures the percentage of the number of relevant results retrieved vs. the total number of relevant results (degree of false dismissals avoided). An approach to achieve this is by ensuring "full coverage" within the CCArray (formally defined in Section 4.5.6).

The precision, however, measures the number of relevant results retrieved vs. total number of results retrieved (degree of false positives avoided).

Formally, the properties can be expressed as follows, where A denotes the number of relevant results retrieved, B the number of relevant results not retrieved and C the number of irrelevant results retrieved:

$$Recall = \frac{A}{A+B} \tag{4.25}$$

$$Precision = \frac{A}{A+C} \tag{4.26}$$

4.10.2 Full Coverage

By full coverage is meant that the M-Grid has a reference to every object inserted. Objects not being referenced are not being investigated when performing a similarity query, and therefore will never appear in any query results. Since every object participates only in a single cluster, full coverage is reduced to having a reference to every cluster. For this to happen every cluster essentially has to be the one closest to some cell within the pseudo-grid.

A situation can arise where a cluster is placed such that it is in fact not being the closest cluster to any cell – hence it is never referenced and full coverage is not satisfied. Figure 4.8 shows a simplified picture of such a situation, where the dark cluster is not the closest cluster to any cell and is therefore not pointed to.

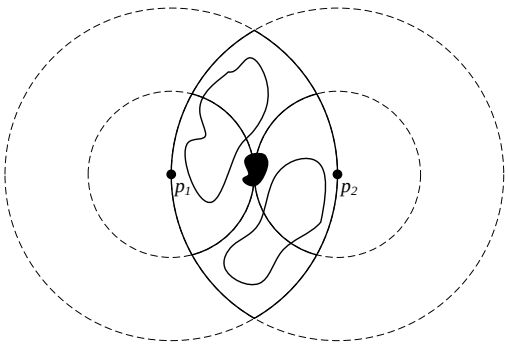

Figure 4.8: *Example of a clustering where one cluster will never be pointed to (the dark one in the middle). The two surrounding clusters is placed directly in a cell – hence they are closest to that cell. No cell is closest to the dark cluster.*

Motivated by the figure we need to formally define the criteria for valid clustering algorithms as follows:

Observation 4.3 (Valid Clustering) *A valid clustering is one specifying a set of clusters $C = \{c_1, \ldots, c_n\}$ such that the following holds:*

$$\forall c', c'' \in Clusters\ \forall o' \in c', o'' \in c''\big(c' \neq c'' \Rightarrow F_{cell}(o') \neq F_{cell}(o'')\big)$$

The intuitive interpretation of Theorem 4.3, is that we require all clustering algorithms to keep all objects within the same cell in the same cluster. The original authors mentions this in an informal manner, where we explicitly require this of the clustering algorithm used, since it may otherwise produce false dismissals.

4.10.3 Algorithms

One of the places where our M-Grid implementation shows its strength is in the flexibility regarding clustering algorithms. In the following we describe some of the clustering algorithms that have been implemented to test the M-Grid.

K-Means Algorithm

The authors of the original M-Grid article uses the K-Means algorithm [25] to find clusters due to the simplicity of the implementation. Our experiments, however, have shown that K-Means in its original version is not a suitable algorithm for M-Grid. This is because it does not satisfy the property presented in Observation 4.3. Our current implementation has been carried out in order to satisfy this property. The pseudo algorithm for K-Means is as follows:

1. Randomly select h objects as cluster centroids – these points represents initial group centroids.
2. Assign each object o to the cluster satisfying either in the following order:
   (a) Cluster already pointed to by cell where object o is placed in.
   (b) Cluster with the closest centroid.
3. When all objects have been assigned, recalculate the centroids of the h centroids.
4. Repeat steps 2–3 until the centroids no longer move or the number of iteration is reached.

Space-Dividing Algorithm
As the name indicates this hierarchical clustering algorithm divides the vector space. The space is divided by grouping the rings for each of the pivots. Combinations of ring groups from each pivot forms a cluster.

An M-Grid having k pivots, each with m rings, that are grouped into g groups per pivot, gives m^k cells divided into g^k clusters and thereby having $(m/g)^k$ cell per cluster. This includes the cells not mapped to by any region in the vector space $\mathbb{S}$ (see Observation 4.2). Since the cells are constructed from the pivot rings, and clustering is based solely on the rings, the property of having at most one cluster overlapping a cell is satisfied. The cluster of a subspace not containing any objects is purged from the result.

An example could be an M-Grid with two pivots with two rings each and a division of the rings by two – i.e., $k = 2$, $m = 2$ and $g = 2$. This gives a pseudo-grid with $m^k = 4$ cells, with the Space-dividing clustering algorithm divides into $g^k = 4$ clusters having $(m/g)^k = 1$ cells per cluster. This is shown in Figure 4.9.

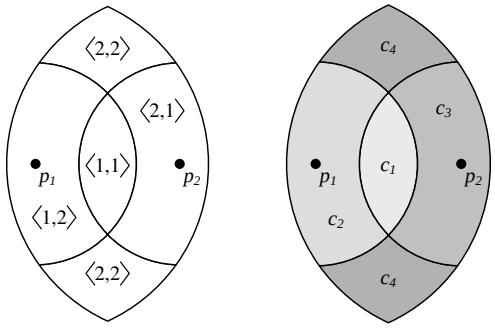

Figure 4.9: **a)** *Pseudo-grid with two pivots each with two rings.*
b) *Applying the Space-dividing clustering algorithm gives four clusters where no two clusters overlap the same cell.*
Only overlapping pivot rings are shown.

Average Linkage Algorithm
The Average Linkage clustering algorithm is of type Hierarchical Agglomerative that is a bottom-up approach combining small clusters to larger ones. Initially the algorithm creates a cluster for every cell in the pseudo-grid, where each cluster contains the object(s) in the cell covered. Hereafter the two closest clusters – measured by the distance between their centroids – are merged into a new cluster containing all objects from the original clusters and with a recalculated common centroid.

This process is continued until the specified number of clusters has been reached. The algorithm maintains the necessary full coverage property since it starts by using non-overlapping clusters (one per cell) which are then combined, so every cell is overlapped by exactly one logical cluster. The actual cluster is formed by the objects covered by the logical one.

Average Linkage Algorithm with Quality Threshold
This algorithm adds a quality threshold to the clustering merging step of the Average Linkage Algorithm. This means that two clusters are only merged if the sum of their objects is below a specified threshold τ_{ALA}. If S is the set of objects, *AvgSize* the average size of the objects in S, *MetaSize* the size needed for storing metadata and *BlockSize* the block-size of the file system, the threshold is set to:

$$\tau_{ALA} = \frac{BlockSize - MetaSize}{2 \cdot AvgSize} \tag{4.27}$$

This way we ensure that no cluster exceeds a block-size. It makes sense to use as much of a block as possible, since the difference in cost between reading only a part of a block and reading the entire block is negligible. Furthermore, increasing the number of clusters, allow us to more effectively prune clusters when performing kNN queries. This optimization is handled in Section 4.13.3.

4.11 Block Handling

In this section we describe how the block handling is performed in order to reduce I/O. This handling is implemented in the class BlockHandler which handles all issues related to I/O with the underlying file-system. The BlockHandler addresses how the clusters are stored and retrieved, making it possible to potentially avoid retrieving all objects in the cluster.

In the following description we use the term block and cluster interchangeably, since a block is the on-disk representation of a cluster with additional metadata (needed for deserialization) associated.

4.11.1 The Physical Layout Within Clusters

In order to potentially stop a sequential scan of the objects in a cluster during a similarity query, the objects needs to be stored in a particular order. The approach used is sorting the objects by increasing distance to a pivot p, which does not add any additional space overhead [20].

Choosing the most suitable pivot has already been defined in Equation 4.24, the only difference being that the set of objects S used to evaluate the efficiency being restricted to $o \in cluster$. Intuitively, the definition ensures that a pivot is chosen such that the distance between the objects is maximized according to the chosen pivot. This means that given an arbitrary range query, we can better

discriminate the objects and thereby prune candidates. In addition to that, the retrieval of objects from the cluster can be stopped immediately after the first object has been pruned by the pivot sorted according to, using Theorem 4.2. However, this sequential scan of the objects within the cluster can only be stopped if the scan happens in ascending order wrt. the distance to the query object o'.

4.11.2 Block Placement

The layout within the clusters determines the pruning power when the clusters are investigated. This will give an increase in performance. However, more efficiency can be achieved by giving more consideration to how the clusters are stored on the disk. Placing nearby clusters sequentially on disk, makes it possible to read all the clusters at once. This means clusters that are likely to be investigated in the same query can be read with only one random seek to the start of the first cluster on disk.

In Figure 4.2, it can be seen that storing the clusters cl_1 and cl_2 together will likely lead to more sequential reads. The reason is that Range queries intersecting cl_1 will likely also intersect cl_2. Nonetheless, this likelihood depends on the placement of the query object o' in the vector space. Therefore, we need to use a query independent criterion for determining which clusters to store physically close on disk.

An approach that can be used, is space filling curves [26]. This allows us to order close-by objects in a n dimensional space linearly, such that close-by objects are placed together in the sequence. By changing the cluster placement algorithm it is possible to implement this behavior, which can easily be done.

4.11.3 Accessing the Clusters

When performing a Transition query, we need to access a possibly large amount of clusters. In order to minimize the random movement of the HDD arm when iterating the clusters, the implemented BlockHandler reorders the given set of clusters in order to produce a sequential schedule. It has been shown that reading a set of disk pages sequentially can be up to 12 times faster than reading the same set of disk pages randomly [27].

This traversal is different from the one originally presented for Range queries [2], where the clusters are traversed according to the distance to the query object. We do not use this approach since all clusters necessarily have to be processed to guarantee correctness. This is due to the assumption that all clusters intersecting our query region may contain candidate objects. Therefore, by reordering according to physical layout, we reduce random I/O.

4.11.4 Cluster Cache

Since the similarity queries Trans and kNN may both incrementally expand the query range in order to satisfy the query, it may be desirable to maintain the last fetched blocks in memory. Therefore an LRU buffer is used to cache the blocks read from the disk. Since XXL already provides such functionally, the BlockHandler has been implemented to pipe the results through the LRU buffer, and the benefits of an LRU strategy is achieved without much work.

4.12 Modification Operators

The M-Grid is not designed for large modifications after the initial construction of the index structure. In order to achieve a good run-time performance, the M-Grid has been designed for bulk loading. This does not, however, imply that it is all static, since it does support insertion and deletion under some performance degradation.

The reason for the degradation is that insertion and deletion affect the choice of the pivots, cluster centroids and the constructed pseudo-grid. If many new objects are inserted/deleted, the initial conditions for how the pivots, clusters centroids and the pseudo-grid were chosen may change. We will elaborate on in this the following sections.

4.12.1 Insertion

In order to add a new object o, the correct cluster *cluster* has to be found. This is done by finding the cell that contains the given object $cell = F_{cell}(o)$ and from here find the closest $cluster = F_{closest}(cell)$. By using the cell, the cluster can be found efficiently (in constant time) via the CCArray. Now the task is simply to insert the object o in the cluster found, and setting the corresponding Boolean entry $repr = tt$ in the CCArray because the cell now is non empty. This procedure introduces a few problems that are not addressed originally [2]. These are as follows:

1. The physical block containing the cluster may not have enough space allocated.
2. The cluster needs to be readjusted since the centroid may have changed.

Not Enough Space (Issue 1) In order to address the first issue, we provide the ability to allocate additional space when the initial cluster is written to disk. The additional space can either be a constant or a percentage of the size of the cluster. This option is only sufficient as long as the amount of new objects to be inserted at run-time is known a priori. This is practically impossible when considering expanding music communities similar to Pandora [28].

Therefore, another approach has to be presented, which addresses the problem of when a block becomes full. The implemented BlockHandler transparently handles this issue, by moving the (over-sized) block to a new location on the disk when the block is written.

Readjusting the Cluster (Issue 2) The insertion of a new object o introduces a new problem since the "balance" of the cluster may have changed. Therefore a new centroid has to be chosen within the cluster, which will affect the CCArray, where all the cells have to be updated. This update not only effect the cells pointing to the changed cluster, but possibly also other clusters since the new centroid may have moved the cluster closer to some other cells. Therefore, the entire CCArray needs to be scanned and updated appropriately according to the procedure described in Section 4.5.6. It is not sufficiently to only examine the adjacent cells to the cluster where the new object o has been placed. The reason is that we can not identify the cells that has the possibility to point to the updated cluster. Therefore, all cells have to be processed. We have not focused on how to efficiently identify which cells need only be checked when adding a new object o.

4.12.2 Deletion

In order to delete an existing object o, the correct *cell* and *cluster* has to be to found, similarly to the approach presented in the previous section. If $o \in S$ then it is per construction of the M-Grid required to be in the *cluster* pointed to by the *cell* in which it lies. Therefore, only that single cluster has to be investigated. Only a subset of the *cluster* needs to be read to verify whether the *cluster* contains the object o because we can use the fact that the cluster is sorted by the distance to some pivot p (see Section 4.11.1). We can use the $dist = \partial^m\left(F_{metric}(o), F_{metric}(p)\right)$ as a stopping condition when iterating through objects in the *cluster*. In other words, if the current investigated object o' has a $dist' > dist$, we can stop. If the object o is found, we can simply delete it and check whether the cluster is empty. Now the same problem and solution as described in the previous section applies (Issue 2).

If the number of objects in the cluster reaches zero after the deletion, the CCArray needs adjustments so that cells pointing to the *cluster* are reset so that $repr = ff$. This ensures that even though entries in the CCArray contain cells that intersects the changed *cluster*, they are not visited when performing Range/Transition queries (since $repr = ff$).

4.13 Optimizations

This section considers the different optimizations done to the original M-Grid to improve its efficiency. The section is divided into four parts where the first describes the Distance Matrix. Its purpose is to cache the distance computations and thereby avoid redundant computations. The second part describes the Cell Index Handling that allows us to perform a look-up in the CCArray in constant time. The third part concerns how the kNN operator can be optimized. Finally, the last part describes how the clusters are converted to raw bytes that can be stored disk.

4.13.1 Distance Matrix

The process of constructing the M-Grid requires that several inter-object distances are computed. If a set of objects of size n needs to be indexed, we will in worst case need to perform n^2 calculations – due to the calculations needed during pivot selection and clustering. Since the modules presented in Section 4.8 work independently, they may repeat the step of calculating some of the distances.

In order to avoid this, we provide the Distance Matrix, which encapsulates the calculation of all the distances. This single module handles all the distances using dynamic programming and we hereby avoid redundant calculations. The Distance Matrix is built incrementally as results become available during the whole initialization process of the M-Grid.

The modules using the Distance Matrix are PivotSelector, and ClusterHandler. Furthermore Distance Matrix supports sorting the distances in descending order, simplifying the process of constructing the pseudo-grid.

4.13.2 Cell Index Handling

In order to answer all similarity queries, the first step is always to map the given query object q into the $cell = F_{cell}(q)$ that contains the object. After the *cell* has been determined it is necessary to find the closest *cluster*. In order to determine the closest *cluster* in constant time, a perfect minimal hashing function [29] is provided. This function is able to map a *cell* into a unique index in the range $[0, m^k)$ where m denotes the number of rings and k the number of pivots. The used function is as follows:

$$F^{idx}_{\langle c_1,\ldots,c_k\rangle}(G) = \sum_{i=1}^{k}\left((c_i - 1)\cdot m^{i-1}\right) \tag{4.28}$$

4.13.3 Cluster Iteration

The implementation of the kNN operator can be done in an efficient manner, provided that information regarding the inter-cluster distances are available. If this is

the case, it is possible to investigate each cluster' in the candidate list of clusters within the range r as specified in Section 4.6.3 in ascending order specified by the distance $\partial\left(o', F_{centroid}(cluster)\right)$, where o' is the query object. The optimization achieved by this is that it is likely that not all clusters have to processed, since each processed cluster may tighten the range radius r and thereby prune some cluster. Since the centroids are the representatives of each cluster (see Equation 4.9), the described scenario is likely to occur.

This is illustrated in Figure 4.13.3 where $k = 4$ and using the query object o'. The initial lookup of the query object o' yields a cell pointing to the nearest cluster cl_1. The k^{th} nearest object in this cluster is o_3 with the distance r_3 from the query object. This distance r_3 is used as an initial query range, which returns the clusters $\langle cl_3, cl_2 \rangle$, since they are pointed to by the cells intersected by the query range r_3. The order, of how the list of clusters is examined, has an impact on the runtime performance of the query execution.

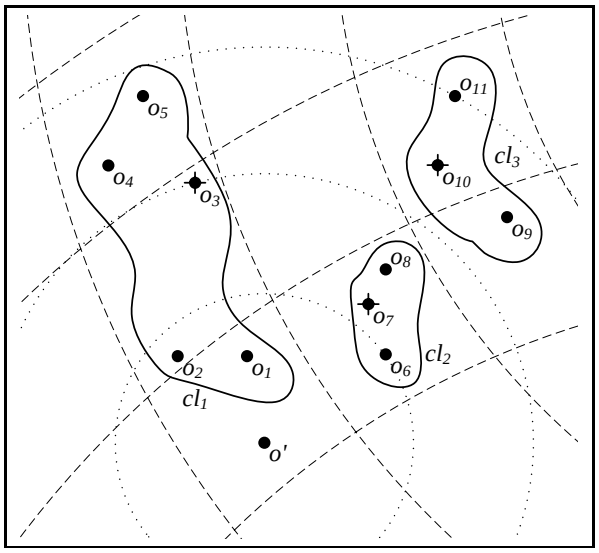

Figure 4.10: *Performing a kNN query with o' and $k = 4$. This yields the result $\langle o', o_1, o_2, o_6 \rangle$.*

Traversing the list $\langle cl_3, cl_2 \rangle$ in this order will still yield the same radius r_3 after investigating cluster c_3, since o_3 is still the k^{th} nearest object. Finally the cluster c_2 is examined and the algorithm terminates, because there are no more clusters to investigate. This gives the result $\langle o', o_1, o_2, o_6 \rangle$.

If we however had traversed the list of clusters in the order $\langle c_2, c_3 \rangle$, this would have resulted in a tightened radius of r_6 due to object o_6, which would allow us to prune the cluster c_3 since it outside the tightened radius r_6. This indicates that the performance is affected by the order of how the clusters are examined. However, the correctness of the result is not affected, since the result in both cases is $\langle o', o_1, o_2, o_6 \rangle$.

4.13.4 Converters

The XXL framework provides converters for many typical data structures that is used in order to serialize the data in the M-Grid in a well-defined manner to the disk. Since new classes have been introduced to form the M-Grid, converters for these have been implemented.

We follow the guidelines of XXL and do not rely on Java's own serialization mechanism. Serializations are fixed at compile time and only allow a single serialize method for each class. Instead, by extending the Converter interface from XXL, it is possible to construct converters that are able to implement different conversion strategies. The strategies can be changed at runtime. This is relevant because in XXL the majority of the converters are simple, since attributes of a class are merely (de)serialized in a fixed order.

We, on the other hand, give another option that integrates the aspect of providing space vs. computation trade-offs. If some attributes of a class can be calculated through some computation (i.e., decompression, etc.), it can be desirable to be able to trade computation for storage. The motivation for this stems from the fact that we work with large feature vectors (many sample values and dimensions).

Chapter 5

Case Studies

5.1 Overview

In this chapter we will introduce two case studies. The first case study (Section 5.2) is a proof-of-concept web player that combine all the features that the core system offers to a client application. The second case study (Section 5.3) is an example of, how a real-world Internet based music shop can implement some of the features from the core system, into their already running system.

5.2 Case Study: Web Player

The Internet is becoming a larger part of our life every day and nearly every new electric device can be connected to it – including devices for playing music.

In this case study we look at an implementation of a web-based music player that supports and exploits all the features in the middleware. We will go through the setup, design and usage of the system.

5.2.1 The System Setup

To make the player available to as many potential users as possible, it was chosen to implement it as a normal web-page. This way users can access it from every Internet-connected computer (even some devices like mobile phones and PDAs) via an Internet browser. In our implementation of the WebPlayer the focus has however only been on supporting playback from browsers on normal PCs. The player should be seen as a proof-of-concept implementation to show all the features of the middleware and at the same time be a test-bed for the system.

The setup can be divided into five parts - client, web server (with player), middleware server, music file storage, and database server. Figure 5.1 shows the setup of all of its modules. In most cases it would be more feasible to have most of the modules of the server side on the same physical server. However, since all communication between the different server modules is via network (XQL, JDBC, HTTP), there is no problem splitting it up as shown in the figure.

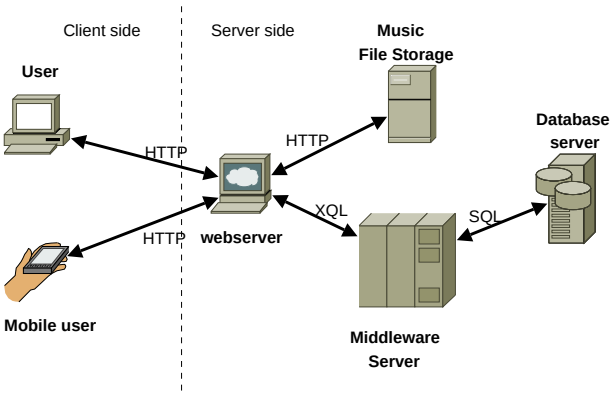

Figure 5.1: *Overview of how the setup of the WebPlayer service could be. The entire server-side could be on the same machine or spread out on several different machines.*

The communication between the client and the web server is normal HTTP requests done by the browser. From the web server and to the middleware a TCP socket is opened and our XQL is used for the communication. When the user selects a music piece to listen to, the actual music file is fetched from the file storage server via HTTP requests. The middleware server uses a DBMS server as back-end for metadata and and feature data. The communication to this server is done through JDBC as described in Section 2.2.3.

The WebPlayer itself is implemented using popular web technologies such as:

- PHP on the server-side.
- XHTML 1.1 and CSS2 for the design.
- Asynchronous Javascript And XML (AJAX) for the client-side to server-side communication.
- Flash for the music playback.

The Graphical User Interface (GUI) of the WebPlayer makes heavily use of AJAX for communication with the web server. This makes the WebPlayer appear more like a normal application and less like a web page. It also implies that the web page holding the player does not have to reload when new commands (e.g., a range query) are sent and it can therefore continue to play the music while work is done in the background. When answers to requests are received, they will appear on the page asynchronously of when they were sent. There is however implemented a request buffer that controls that one request from a client will not be overruled by another request from the client.

5.2.2 Features

The WebPlayer supports most of the main features of the middleware system and present it to the users in a user friendly manner. Among the main features are:

- Display of random playlist.
- Playback of music with display of metadata.
- Possibility to select which features to use in queries.
- Range and kNN queries.
- Evolving and sized Transition requests.
- Look-up of artists and titles.
- Fully customizable layout.

To make the WebPlayer more suited for testing of the middleware, almost every part has been made configurable through a configuration window. Among the configurations are:

- Size of k in kNN.
- Range (percentage).
- Option for sorting and refining Range results.
- Transition evolution min/max and size.
- The feature types to use and how to weight each of them.

Figure 5.2: *First view of the WebPlayer.*

5.2.3 Usage

Since the WebPlayer exposes some features to the user, which he would normally not expect (like feature/distance function selection), we now go through the basic usage of the WebPlayer.

To use the WebPlayer, the user needs to point a web browser (tested with MS Internet Explorer and Mozilla Firefox) to the URL where the WebPlayer is installed. Figure 5.2 shows a screen-shot of the WebPlayer as it looks when the user first sees it.

On the left is the main menu. This gives access to:

- The Player.
- The configuration window (described later).
- Help window.
- Statistics window (described later).
- About page.

Right in the middle of the page, is placed two of the key components of the WebPlayer. The upper part is the actual WebPlayer area that shows the metadata for the song currently playing. It also contains the flash component that handles the playback. Underneath the player area is links for transition generation and random playlist fetching. Below this, the user will find the playlist which is the primary working area when using the player. Initially the playlist is empty, but a click on the "load songs with random playlist" will fill the playlist with random songs.

In the playlist the user will find information about artist and song title. Besides this, every song has links to do a range query, a kNN query or delete the song from the list.

If the user selects to perform one of the queries, then a new song list area shows up underneath the playlist. As soon as the server has found the matching result for the query, this song list will get populated with the songs in the result. The song list will show artist name, song title and how much the song deviate from the query song. The deviation is shown as a percentage where 0% is perfect match (the query song itself), and 100% is the song in the database being most different from the query song. Figure 5.3 shows the WebPlayer after the playlist is populated, a range query has been performed and the result has been returned.

Figure 5.3: *The main part of the WebPlayer with a populated playlist and result of a range query.*

Configuration of the WebPlayer

All configuration of the WebPlayer can be handled by the user via the configuration window (see Figure 5.4). When the user saves the configuration changes, it is saved in the current browser session and also in a cookie on the user's computer. This way the settings will also be the same the next time the user opens the browser and loads the player. However not all configurations can be saved over time.

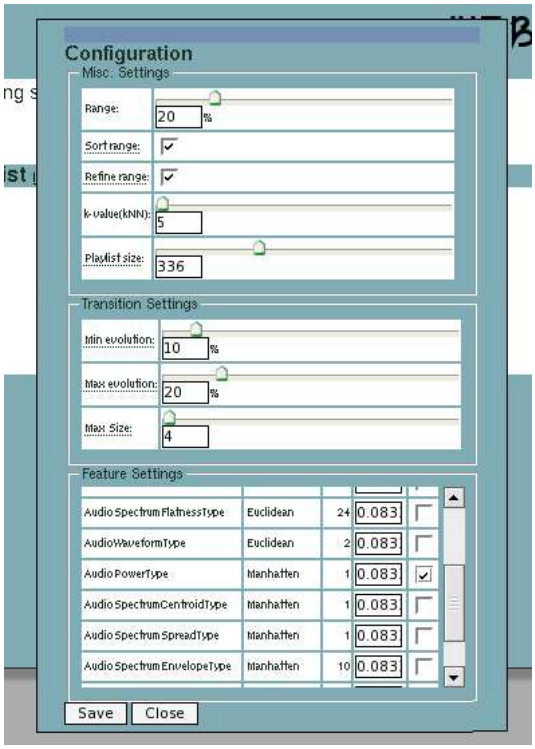

Figure 5.4: *Every WebPlayer setting can be configured via the configuration window.*

At the bottom of the configuration window, the user can select which feature/distance function pairs to use (implies which index to use on the server-side). The list in which the user chooses this is dynamic and can be changed on the server side over time – hence the features available one day do not have to be the same the next day. For this reason the user has to set this setting every time the WebPlayer is reloaded in the browser.

Each of the feature/distance function pairs can be weighted so that the result from one index has more impact on the final result than the result from another. The weight is given by a number between 0 and 1 and will get normalized on the server-side. All the settings in the configuration window reflects the actual contents of the server – e.g., the max size of the playlist is the actual number of songs in the database.

Some of the settings are percentages going between 0% and 100%. These percentages actually gives you a range between 0% deviation and the longest distance between any two songs in the system.

Construction of a Transition

A transition is another way of constructing a playlist. In a transition the goal is to get a smooth transition between the songs in the playlist.

In the WebPlayer you can construct two types of transitions – Evolving Transition and Sized Transition (see Sections 4.6.6 and 4.6.7). The user can select a start song and an end song. This is done by typing in the artist name (the input field proposes names as you type) and when an artist is selected, the user can select a song among the songs this artist has made. Finally, the user can select the type of transition to use and then wait for the transition to be constructed. Figure 5.5 shows an ongoing construction of a transition where the first song is selected and the WebPlayer is proposing artist names for the second song.

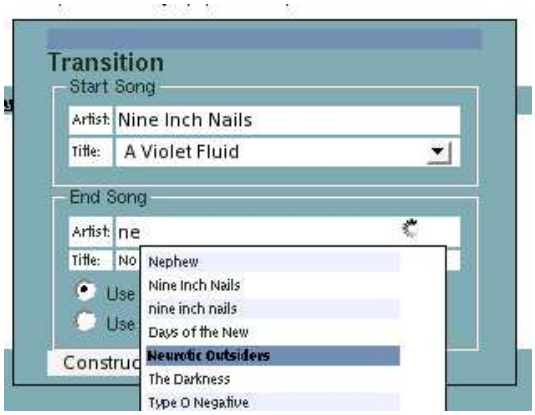

Figure 5.5: *The construction of a Transition is ongoing in the transition window.*

It is not always possible to construct a transition that satisfies the configuration set by the user. In such cases the user will not get a transition but rather a message giving some tips on how to get the transition to give a result. The user then have to change the configuration and try again.

Statistics and Settings

A special window has been added such that the user of the WebPlayer can get further information about each feature/distance function pair. The information is primarily statistics, but also the main settings used when initializing the index for that feature/distance function pair. Figure 5.6 shows how such a statistic for a single feature could look like.

Figure 5.6: *A closeup of the statistics and setting for a single function/distance function pair.*

If one of the links below each of the statistics blocks saying "Show full cluster stats" is pressed, then another window will show up with further debug information about that specific index. This debug information contains data about how and where the objects are placed in the clusters. This gives an idea of how well the clustering has been performed. The data in the statistics window is updated every time the window is opened, but only values that change over time will be updated. The settings will only change if they are changed on the server-side and the server-side system is restarted.

5.2.4 Evaluation

When using the WebPlayer, the user has a good opportunity to evaluate the features of the middleware. The features of the player works as supposed with the original design of the middleware, which shows that the design is well implemented. As a user wanting to test the results of using different audio music features, the player gives great opportunity to investigate this. There is however some issues to consider.

The first one is that if the user selects several audio music features to use in the queries, then the system is starting to become slow (due to extra calculations). Because of this we recommend that the user does not use more that 2-3 audio music features at the same time with a maximum of 50 dimensions in total.

Another case is the rendering of the result of a query. Even though the middleware server responds quite fast, then the rendering of a long playlist in a browser can be a slow process (2+ seconds). This can make the system seem less responsive when the middleware actually has returned its result. To give the user an idea of the actual query time spend in the middleware, this time is shown under the playlist.

A final case is that a plug-in or flash element is needed to do the actual playback of the music. This however tends to give compatibility issues when moving between browser brands.

5.3 Case Study: On-line Music Shop

In this second case study we will look at a more real-world example. MusicMatcher is an on-line Webshop we have created for the purpose of showing how our core system can be implemented in an already running Webshop.

The main idea behind the MusicMatcher Webshop is to sell audio music by the track and not by album as it normally is. Similar shops have arrived on the Internet throughout the last couple of years (e.g., iTunes and Amazon), but what makes MusicMatcher different from them, is that it actually looks at the audio music contents (features) itself, when it recommends similar songs to the shop user. In most of the on-line shops providing such similarity recommendations today, this is done using statistical information about what the other users buying the same piece of audio music has bought.

Figure 5.7 shows a part of the Webshop as it could look in a browser. This is just the top of the Webshop, that shows the main navigation panel, some information to the customers, the quick-search and the top of the genre selector. To the right the top-part of the top-10 list is shown.

5.3.1 The System Setup

The setup for the music shop is the same as shown in Figure 5.1, but would in a real-word setup also have a system for payment transactions. The entire implementation surrounding the actual buying of the music is however beyond the scope of this project and therefor not implemented in this case study. Only the main functionality which has importance in relation to the middleware is implemented.

5.3.2 Features

The MusicMatcher web shop combines the strength of both the metadata and the feature data of the music. The metadata is used to group the music into categories (genre, artist, album, year) which normally makes it easy to find things when you know what you are looking for – e.g., finding the song *"U2 - One"* is just going into the Rock genre, finding the artist *"U2"* and then select the title "One".

Figure 5.7: *A top view of the MusicMatcher Webshop as seen when the user first enters the page.*

Figure 5.8 shows a close-up of the part of the Webshop where the user can select a genre to get an artist list. In the bottom of the figure is shown the result of the user selecting the genre ”rock”.

Figure 5.8: *A close-up of the area of the shop where the user can find music grouped by genre, and below that, by the artists in a specific genre.*

Alternatively the visitor can use the quick-search (see Figure 5.9) to find an artist or song by the name or the title. Whenever the shop visitor has found a song he likes, then the use of feature data is engaged. Now the visitor is given the possibility to find songs similar to the one the visitor knows. This way the visitor gets exposed to music it is likely he will find interesting and most likely buys more.

Figure 5.9: *A close-up of the quick-search in action. Notice the auto-completion that proposes possible song titles.*

Figure 5.10: *A close-up of the result of a similarity query. The list gives fast access to do further similarity querying or buying the track.*

It is the shop-owner that selects which music feature(s) the system should use to find the similar songs and which similarity query to use to get the result. Figure 5.10 shows a result list of a similar query.

As the figure shows, the Webshop gives the user an easy overview of the found songs (ordered by similarity deviation). The first song is the song the visitor used in the query. From here on the visitor can easily listen to the songs, and buy it if he likes it. If he did not get enough music yet, then he can do even further similarity queries for any of the songs in the list.

The visitor can actually construct a playlist with a smooth transition between songs by simply doing a similarity query for a song he likes. Hereafter the visitor takes the best match (second song) in the result list and adds this to the playlist and then do a similarity query for this song. The visitor keeps doing this until he has enough songs in his playlist. In this case the playlist would of cause be the contents of the shopping basket.

5.3.3 Evaluation

The Webshop case story is meant as a proof-of-use implementation and is not meant as a complete system. It does however show off the potential of using the middleware in a real-world system like a music shop – and thereby show that our system is not only usable for tests. With a setup like this where similarity queries (in this case kNN) are show to the user when he requests it, the speed of the system seems fairly fast. An indication that shows that the system is working is however needed in order to get the user to wait without browsing on in other directions.

Chapter 6

Evaluation

6.1 Overview

In this chapter we will go through a structured evaluation of the middleware implementation. In Section 6.2 we will provide an overview of the test setup. In Section 6.3 we evaluate how our addition of the Distance Matrix to the M-Grid optimizes the system, and in Section 6.4 we evaluate different pivot selection algorithms. The next thing we test is the different distance functions. This is done in Section 6.5. Section 6.6 we describe the test of some the clustering algorithms implemented, and in Section 6.7 the effect of the cluster cache is evaluated.

6.2 The Setup

With our flexible middleware at hand (especially the M-Grid), an interesting question is: *How does different implementations of the modules impact the performance?*. An example could be how much difference it makes to use one clustering algorithm instead of another one. We differentiate between two impact points – "Initialization" where, e.g., the index structure is constructed and "Run-time" where the system accepts and processes queries from clients.

6.2.1 Default setup

The way the tests are performed, is by keeping all but one parameter fixed and evaluating the effect of this dynamic parameter. All tests are performed on a setup consisting of 942 songs (mixed MP3 files), where number of songs is denoted n. One feature ("AudioSpectrumEnvelopeType" defined in the MPEG-7 standard) having 10 dimensions is used. If nothing else is mentioned, the following setup is default:

- Manhattan as distance function.
- Average Linkage with Quality Threshold as clustering algorithm.
- Full Pivot Selector as the pivot selection algorithm.
- Block Handler with cluster caching enabled.

6.2.2 Test Configuration

The computer used for the tests had the following configuration:

- CPU: Intel Xeon 2.8GHz
- RAM: (limited by Java's -Xmx parameter)
  - RAM: 2GB for bulk loading the indices
  - RAM: 1GB for servicing queries
- OS: Red Hat Enterprise Linux WS release 4
- Java: Sun Java version 1.4.2_08-b03

6.2.3 Test Query Properties

When performing queries we set out to retrieve around 1% of the total database size. In this case this amounted to $k = 10$. All the times shown in the tables are in seconds and are for 100 random queries run in succession. In order to make the tests repeatable and fair, we ensure that the random functionality in all tests are seeded with the same value.

6.2.4 Abbreviations Used

During the remaining parts of this chapter the following abbreviations may be used:

- Init time or just init is used for initialization time.
- RT is used for run-time.
- Clustering algorithms:
  - KM: K-Means
  - SP: Space-Dividing
  - AL_{QT}: Average Linkage with Quality Threshold

6.3 Distance Matrix

The Distance Matrix has been introduced in order to reduce the number of redundant computations. The M-Grid presented in Figure 4.6 shows that the different components are separated and have no knowledge of the existence of the other components. This separation leads to a more clean design [30].

Table 6.1 shows that, by using the Distance Matrix, we avoided a large number of distance calculations. The clustering algorithms have, as the table shows, different needs in terms of the number of distance computations needed. The Distance Matrix is in all cases an extremely good optimization.

The column *Actual* contains the number of unique calculations between pairs of objects actually performed. When a distance between two objects has been calculated, it can be reused afterwards. This is expressed in the *Avoided* column that describes how many percent of the needed calculations that were reused, without having to calculate the distance.

Clustering algo.	Needed	Actual	Avoided
KM	351976	307882	12,7%
SD	300426	151240	49,7%
AL_{QT}	89282	45097	49,5%

Table 6.1: *Evaluation results which shows how many unique computations were performed and needed. Also shown is the percentage of the computations that were avoided.*

Results will later show that the AL_{QT} is the most run-time efficient, so here we will briefly describe, how the algorithm makes use of the Distance Matrix. The algorithm uses a bottom-up approach, where smaller clusters are merged into larger ones until the number of clusters has been reached. Furthermore the quality threshold determines the maximum allowable number of objects in a merged cluster.

During each iteration of the algorithm the two closest clusters are determined. Having h clusters to investigate this gives h^2 distance calculations to perform. When the Distance Matrix is used, the only unknown distances in each step are the distances between all clusters and the cluster merged in the previous iteration. This reduces the number of distance calculations to h.

When merging two clusters cl_1 and cl_2 having m_1 and m_2 objects respectively, a centroid for the new cluster is to be calculated. Without the Distance Matrix this implies performing all calculations between all objects, which gives $(m_1+m_2)^2$ calculations. Using the Distance Matrix this is reduced to $m_1 \cdot m_2$, since the inter-object distances in cl_1 and cl_2 are already known.

6.4 Pivot Selectors

The performance of the M-Grid is based on how the pivot points are placed and hereby how the pseudo-grid is created. Due to this, the method for selecting the pivot points is very important – hence it is made changeable. In our evaluation we have tested the two ways of finding pivots mentioned in Section 4.9.2 (Random and Full Candidate set).

Table 6.2 shows the results (init and RT) of our evaluation of the two pivot selector implementations Random and Full. In the case of the Random Pivot Selector the tests were conducted with a candidate set of size 10%. This means that the Random Pivot Selector performs only 10% of the calculations performed by the Full Pivot Selector. This can also be seen from the init time that is about 90% less for the Random Pivot Selector.

Candidate set	Init	RT
Full	14059	391
Random	1289	504

Table 6.2: *Evaluation results when changing the pivot selector algorithm.*

What can also be seen is a large difference in the run-times for the two pivot selectors, where the Full Pivot Selector performs better. Therefore another test is conducted, that uses other sizes of the candidate set, to reveal, when the Random Pivot Selector begins to either be as slow as the Full Pivot Selector or when it starts to have impact on the run-time tests. Table 6.3 shows the resulting timings for the Random Pivot Selector with 10%-90% candidate sets (with intervals of 10%):

Size	Init	RT
10%	1289	504
20%	2281	449
30%	3622	492
40%	4392	423
50%	4830	415
60%	5388	402
70%	6273	393
80%	7132	392
90%	8492	392

Table 6.3: *Evaluation result for the Random Pivot Selector with different candidate set sizes.*

These numbers are depicted in Figure 6.1. The figure shows that there is a linear dependency between the init and run-time. The run-time increases as the init time decreases.

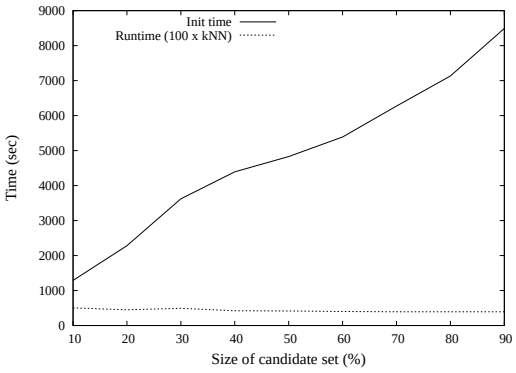

Figure 6.1: *Evaluation results showing both init and run-time performance when increasing the candidate set.*

6.5 Distance Functions

One of the main problems to consider when testing distance functions is to choose a function that is able to implement the notion of similarity. Other things to consider are, e.g., the precision of the distances (do we work with 0 or a hundred decimals), the number of calculations needed for one distance result, etc.

In our evaluation we have primarily focused on the metric distance functions Euclidean and Manhattan that both maintain the metric property defined in Section 4.3. This is because M-Grid is designed for metric functions, and investigating non-metric distance functions in M-Grid would be a complete project of its own. In Table 6.4, the timings for the two algorithms are shown:

Function	Init	RT
Euclidean	11059	410
Manhattan	9070	386

Table 6.4: *Evaluation results of distance functions.*

As the timings in the table clearly show, the Manhattan distance function is the fastest one both in run-time and initialization. This is primarily due to the fact that it only uses simple operations like minus and plus, whereas Euclidean also uses multiplication and square root. Our findings confirm what has been presented in earlier experiments [31].

6.6 Clustering Algorithms

The clustering algorithm dictates how the objects are ordered on the disk and which objects to recalculate during a query, hence these ought to have some impact on the run-time performance.

We have implemented and investigated four different clustering algorithms and seen how they affect the system, and how well they actually cluster the objects. Table 6.5 shows the result of our evaluation of the four different clustering algorithms.

We have strived to place the same amount of objects in each cluster as we query for (1% of n) plus an additional 25%. This has been chosen in order to reduce the number of clusters to visit.

Furthermore our studies have shown that around 5% of the total number of objects n are placed far away from the others in groups ranging from 1–3 objects. This leads to the formation of clusters, where roughly 25% of the total number of clusters are small. The remaining 95% of the objects are partitioned into the remaining 75% of the clusters. Using the number of objects in each cluster that we strive for, we define the total number of clusters:

$$noC = \left\lceil \frac{n \cdot 95\%}{n \cdot 1\% \cdot 125\%} \cdot \frac{100\%}{75\%} \right\rceil = 102 \qquad (6.1)$$

This leads to a configuration of the algorithms being as follows:

- Common configuration
  - Number of clusters: $noC = 102$
- AL_{QT}
  - Max. number of objects in each cluster: $noO = \lceil n \cdot 1\% \cdot 125\% \rceil = 12$
- K-Means
  - Number of iterations: $noI = 8$

Algorithm	Init	RT	Pruned	Computations	Sum
KM	14084	488	7690	50237	57927
SD	9853	634	12848	55044	67892
AL_{QT}	11233	389	6904	48475	55379

Table 6.5: *Evaluation results when changing the clustering algorithm.*

The results of our evaluation are shown in Table 6.5. If we look at the init time (of the clustering process only), it is clear that the SpaceDividing algorithm is the fastest one. If we weigh this result against the others, then it is easy to see that the SpaceDividing algorithm is also the least optimal one when considering the run-time performance.

The columns *Pruned* and *Computations* indicate the number of objects that were pruned due to Theorem 4.1, and the actual distance computations performed on objects not pruned respectively. These numbers explain the reason behind the performance of AL_{QT}. We see that the sum of *Pruned* and *Computations* (the *Sum* column) has the smallest value for AL_{QT}. This value equals the number of objects retrieved from disk, which indicates that the clustering performed by AL_{QT} is better.

This can mainly be attributed to the threshold parameter it uses to set the maximum number of objects per cluster. This ensures that no large clusters are created.

The maximum number of objects to store in a cluster is a trade-off between I/O and recalculation. Avoiding large clusters is good, since we have to do recalculation of every object in the cluster if we do a query that just reaches one of the objects. The other way around we wish to avoid too much I/O. Our studies have shown that smaller but well-spread clusters is optimal.

6.7 Block Handler

The only optimization in the Block Handler is the Cluster Cache. We test the M-Grid to see how good an optimization it actually is to have the Cluster Cache in the Block Handler turned on. Table 6.6 shows the timings with the Cluster Cache turned on and off.

Cluster Cache	RT	Disk I/O
ON	387203	440
OFF	398334	839

Table 6.6: *Evaluation results with enabled/disabled Cluster Cache.*

The results indicate that the Cluster Cache provides a speed boost to the queries by reducing the number of accesses to the disk in half. However, this reduction only introduces a small run-time performance increase. We attribute this is to efficient disk caching by the operating system.

Chapter 7

Summary and Future Work

7.1 Summary

We have, throughout this paper, introduced, designed, and discussed the implementation of a framework for enabling similarity queries on audio music in database systems. Focus has be put on the flexibility of the system, and different implementations of the parts are considered.

In Chapter 2 we described the modular architecture of the system and gave a detailed description of each module. Problems with the JPOX component was discovered and an alternative solution was proposed.

Chapter 3 dived deeper into the aspects of performing similarity queries, on high-dimensional data, across several audio music features at the same time. A solution was proposed that we believe gives reasonable results in kNN and Range queries. Applying the solution to Transition queries was however disregarded since it was identified as not being clear for the user how to use it.

In Chapter 4 we focused on the M-Grid index structure and gave a formal description of its functionality. We discussed the implementation and how we optimized the index structure from how it was originally designed. Furthermore we added support for Transitions queries on the M-Grid.

To show that our system implementation works as described, we introduced two case studies in Chapter 5. One case study was a showcase of all the features of the system and was implemented as an on-line music player. The second study was an on-line music shop that shows that it is possible to use the system in a real-world setup. We believe that the case studies gives a good overall idea of how the system will perform and how it could be used.

An evaluation of the system, and M-Grid in particular, is given in Chapter 6. The evaluation went systematically through the systems changeable parts by changing only one of them at a time and keeping the rest in a default setup. The results show that using the Average Linkage with Quality Threshold clustering algorithm, Full Pivot Selector and Manhattan distance function was the optimal setup with the implemented parts. Even more optimal setup could maybe be found with alternative implementations of the parts, but this is left for the users of the system to explore in the future.

We believe that our overall goal of implementing and testing a framework for constructing similarity query enabled database systems has been fulfilled. The system works as described and gives, in our opinion, a good foundation for further studies in relation to audio music databases and similarity queries.

7.2 Future Work

In the following we propose some areas of interest for future work.

Insertion and Deletion

Much of our work has been focused on the construction of the M-Grid and processing of the similarity queries. Insertion and deletion was only mentioned briefly in Section 4.12, and future work should address the relatively static nature of the M-Grid. We have shown that the M-Grid can handle many insertions/deletions as long as they are distributed in the vector space. Bulk insertion of many similar objects poses a problem, since they most possibly would be placed within the same cluster. Future research should investigate whether it is possible to:

- change pivot points at run-time effectively.
- detect when the performance starts deteriorating after bulk insertions/deletions.

System Adaptable
Another area where more research can be done, is to make the framework adapt according to the Underlying disk setup – e.g., a linear scan of a cluster may act differently if the disk(s):

- are put in a RAID setup.
- are uses by a different file-system.
- have a different disk-layout (e.g., blocks, sectors).
- use a different addressing method (e.g., CHS, LBA).

Multi-Feature Queries
In our study we have discovered that a fast index alone is not enough to get a reasonable speed when doing queries across multiple features – and in our case across multiple indices. The vast amount of I/O and calculations needed to compute precise results are too many to process within the time-limit of what a normal user would like to wait.

There are several different approaches to deal with this problem. One could be to loosen how precise the result is (according to the formal definition). Since the users opinion about the result is very subjective, the perfect result for one user could be the wrong result for another – hence it could be argued that the precision does not matter that much in the end, within a reasonable margin. Another approach is to restrict the queries the user could perform. If the user could e.g., only do simple queries like: *Give me a song similar to song X*

In that case most of the results could be pre-computed and only information about nearest neighbor of each song should be stored. Maybe it should not even necessarily be the nearest neighbor, but just a near neighbor, leading to even less calculations performed. Future work can be to research how imprecise a result can be, without the user perceive it as wrong.

User Tests
An interesting research area could be to look at how users would work with, e.g., the WebPlayer. Work-flow could be logged and investigated to see how the average user uses the new similarity queries available. This could lead to discovery of whether the user would actually prefer one query type over another, if one feature is more used than others and what that makes a user choose music as he does. If the user does not like the result of a similarity query, possibilities are that other users will not like the result either. If the users can tell the system that a result is less good, then the system can use this to make the next result for the same query better.

Collaborative Filtering
A final area where more research could be focused is on collaborative filtering in connection with content-based similarity queries using feature data. One possibility could be to use the collaborative filtering to fine-tune the weights that are used with the features. If the user does not like a result it is likely that other users performing the same query does not like it either.However, if the user could tell the system that he did not like the result, then the system could use this to weight the features in future queries. Future work can be to research how to incorporate the collaborative filtering concept in the already existing system.

7.3 Acknowledgment

We are grateful to Mario Nascimento and Christian Digout for providing the source-code for the M-Grid, which has inspired parts of our work. Furthermore we thank the Intelligent Sound research project [32] for providing storage for our MP3 files and other evaluation related data.

Bibliography

[1] K. Rajaratnam, K. Schulz, and P. B. Jensen. Similarity enabled algebra for music databases. 2005. M.Sc. Thesis, Aalborg University, Department of Computer Science.

[2] C. Digout and M. A. Nascimento. High-dimensional similarity searches using a metric pseudo-grid. In *Proceedings of the 21st ICDEW Conference*, pages 1174–1189, 2005.

[3] J.v.d. Bercken, B. Blohsfeld, J.-P. Dittrich, T. Schäfer, M. schneider, and B. Seeger. XXL - a library approach to supporting efficient implementation of advanced database queries. In *proceedings of the 27th VLDB Conference*, pages 39–48, 2001.

[4] H. Vinet, P. Herrera, and F. Pachet. The cuidado project – new applications based on audio and music content. In *Proceedings of ICMC Conference*, 2002.

[5] A. Uitdenbgerd and J. Zobel. Melodic matching techniques for large music databases. In *Proceedings of the 7th ACM international conference on Multimedia*, pages 57–66, 1999.

[6] M. Welsh, N. Borisov, J. Hill, R. v. Behren, and A. Woo. Querying large collections of music for similarity. 1999. Technical Report UCB/CSD00 -1096, U.C. Berkeley Computer Science Division.

[7] C. Digout. Metric techniques for high-dimensional indexing. 2004. Technical Report TR 04-19 Department of Computing Science, University of Alberta Edmonton, Alberta, Canada.

[8] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on VLDB*, pages 194–205, 1998.

[9] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. In *Proceedings. XXI Internatinal Conference of the Chilean*, pages 33–40, 2001.

[10] S. Adali, P. Bonatti, M. L. Sapino, and V. S. Subrahmanian. A multi-similarity algebra. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 402–413, 1998.

[11] S. Atnafu, L. Brunie, and H. Kosch. Similarity-based operators and query optimization for multimedia database systems. In *International Database Engineering and Application Symposium*, pages 346–355, 2001.

[12] Sun Microsystems. Java data objects (jdo). `http://java.sun.com/products/jdo/`.

[13] Sun Microsystems. Java database connectivity. `http://java.sun.com/products/jdbc/`.

[14] The XStream project. `http://xstream.codehaus.org/`.

[15] J. M. Martinez, editor. *MPEG-7 Overview (Version 10)*. ISO/IEC JTC1/SC29/WG11, 2002.

[16] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on VLDB*, pages 426–435, 1997.

[17] Last.fm. Collaborative music filtering. `http://www.last.fm/`.

[18] A. Silberschatz, H. F. Korth, and S. Sudarshan, editors. *Database System Concepts* 4^{th} *edition*. McGraw-Hill, 2002.

[19] A. Traina, C. Traina Jr., B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Proceedings of International Conference on EDBT*, 2000.

[20] J. French J. E. Barros, W. Martin, P. M. Kelly, and T. M. Cannon. Using the triangle inequality to reduce the number of comparisons required for similarity-based retrieval, 1996.

[21] P. Ahrendt, A. Meng, and J. Larsen. Decision Time Horizon for Music Genre Classification Using Short Time Feature. In *Proceedings of EUSIPCO*, pages 1293–1296, 2004.

[22] A. Meng and J. Shawe-Taylor. An investigation of feature models for music genre classification using the support vector classifier. In *International Conference on Music Information Retrieval*, 2005.

[23] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[24] B. Bustos, G. Navarro, and E. Chavez. Pivot selection techniques for proximity searching in metric spaces. *PRL: Pattern Recognition Letters*, 24(x):33–40, 2003.

[25] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, 1967.

[26] H. Chen and Y. Chang. Neighbour-finding based on space-filling curves. *Information Systems*, 30(3):205–226, 2005.

[27] T. Kahveci and A. K. Singh. Optimizing similarity search for arbitrary length time series queries. *IEEE Transactions on Knowledge and Data Engineering*, 16(4):418–433, 2004.

[28] Music Genome Project. Pandora music player. `http://www.pandaora.com/`.

[29] M. V. Ramakrishna and P.-Å. Larson. File organization using composite perfect hashing. *ACM Trans. Database Systems*, 14(2):231–263, 1989.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, editors. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1998.

[31] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *Proceedings of the 8th International Conference on Database Theory*, pages 194–205, 2001.

[32] The Intelligent Sound research project. `http://www.intelligentsound.org/`.

[33] H.Crysandt. The MPEG-7 Encoder Project. `http://mpeg7audioenc.sourceforge.net/`.

[34] The Apache XML Project. XMLBeans. `http://xmlbeans.apache.org/`.

All URLs were tested valid as of 13–06–2006.

Appendix A

Suite Overview

A.1 Overview

In this appendix, we describe some of the tools implemented in connection with this project. All the mentioned tools play an active role in the test and evaluation of the middleware and are therefore considered part of the final system. The functionality of each tool is described briefly, and special technologies used are mentioned. Along with the coverage of the tools throughout this appendix, we also show figures that places each tool in the work-flow.

A.2 The MPEG-7 Audio Encoder

To get a well-defined set of features to test our middleware with, we have chosen to use features defined in the MPEG-7 standard [15]. In order to generate these features we needed a tool that extracts the features from audio music files and encode them into the MPEG-7 format, which is based on XML. Such a tool already existed. The "MPEG Audio Encoder" [33] is a simple Java-based tool that supports extraction of a subset of the features defined in MPEG-7 from audio music files in the WAV format.

The MPEG-7 Audio Encoder used in our system is the original version that has been modified in the followed areas:

- Added support for the MP3 file format.
- Added support for GZip compressed output – *.mp7.gz files.
- Minor optimizations for speed.

The MPEG-7 Audio Encoder takes an XML-based configuration file (defined by the original version of the tool) and an audio music file as input, and returns a GZip compressed, XML-based MPEG-7 file containing all the extracted features.

The MPEG-7 Audio Encoder is the first step in the Dataset Encoding in the work flow (See Figure A.1).

A.3 The MPEG-7 Audio Parser

The MPEG-7 Audio Parser is our own tool. It reads the mp7 files generated by the MPEG-7 Audio Encoder. The tool is based on XML Beans [34] and is designed as a library to be used in other tools. XML Beans is used to create a mapping between instances of an XML schema and the corresponding Java object tree. The XML schema used in the tool, is the part of the MPEG-7 specification that handles audio description using features.

The tool parses a GZip compressed mp7 file into an FSS object holding data for multiple features of a single song.

The MPEG-7 Audio Parser is the second part of the work-flow in the Dataset Encoding (See Figure A.1).

A.4 The Dataset Encoder

In order to encode large amounts of music data (MP3 files) into something we can use in our system, we have created the Dataset Encoder. An overview of how the Dataset Encoder is used is shown in Figure A.1.

The Dataset Encoder takes a directory (with possible sub-directories) as input and traverses the top-level directory for MP3 files, which are encoded into a Dataset stored on disk. After that, the Dataset Encoder calls itself recursively on the sub-directories. The output when done is a Dataset file for the input directory and each of the nested sub-directories having MP3 files. This output structure has been chosen because audio music files often are categorized in directories (e.g., one directory for each album).

The Dataset Encoder uses the MPEG-7 Audio Encoder and MPEG-7 Audio Parser to get object trees representing the feature data of the input songs.

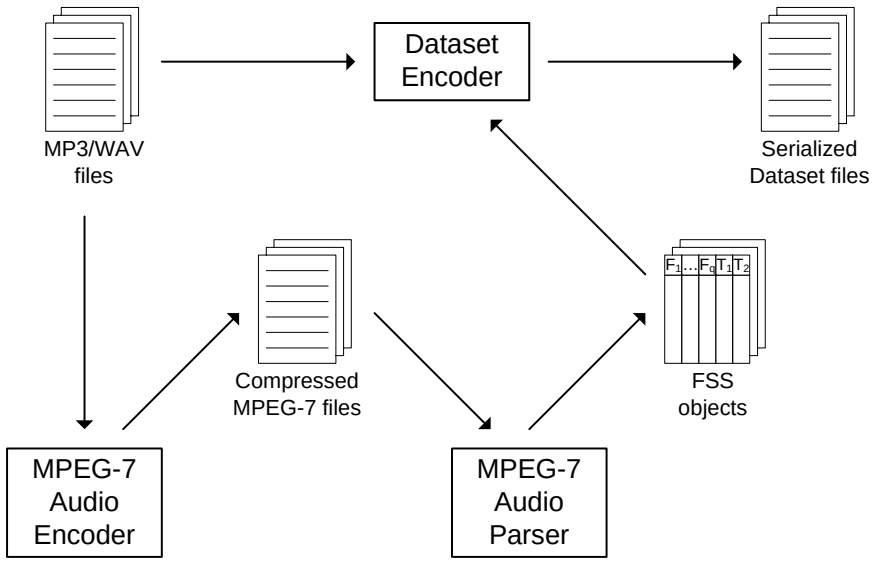

Figure A.1: *The Dataset Encoder uses the MPEG-7 Audio Encoder and the MPEG-7 Audio Parser to encode a set of audio music files (MP3/WAV) into a number of Datasets stored on disk.*

The temporary compressed mp7 files are preserved, since the encoding of those is a time-consuming task. If, for some reason, the encoding is interrupted, all mp7 files already encoded are still there. These mp7 files are reused and parsed directly.

The song metadata is retrieved by either reading the ID3 tag information from the MP3 file itself, or by parsing it from a file having the same name as the song file, but with a .txt file extension.

When both metadata and feature data for a song has been parsed, a Dataobject is created. Having done this with every song in a directory, the Dataobjects are collected in a Dataset, which is stored on the disk using the XXL converters mentioned in Section 4.13.4.

A.5 The Dataset Merger

The Dataset Merger is a simple tool working on the output from the Dataset Encoder. In the same manner, it recursively traverses the input directory and possible subdirectories for input Dataset files. All the files found are deserialized using the XXL converters, that were used in the Dataset Encoder.

The schema for both the metadata and FSS part of the Datasets are then compared. All Datasets having schemes equal to the first one investigated are merged into a new Dataset holding all objects. The last step is serialization of the new Dataset, which is stored on the disk. The converters are again used for this.

A.6 The Database Creator

This tool is a key component in the overall system, since it is responsible for bulk loading data into the underlying RDBMS and creation of the index structures.

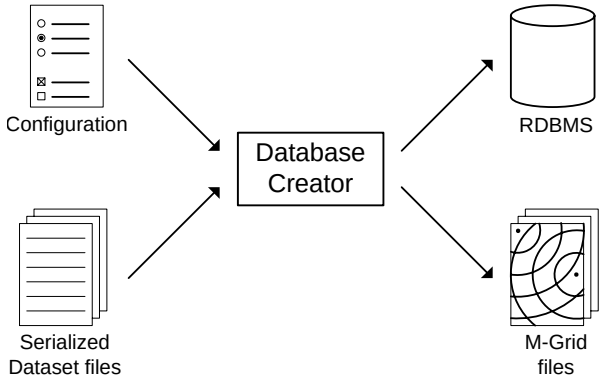

Figure A.2: *The Database Creator bulk loads the information in a number of Dataset files into an RDBMS, and constructs index structures (M-Grids) using a configuration file.*

All steps performed by the Database Creator are highly configurable using an XML-based configuration file, which is described in Appendix B. In addition the configuration also a set of Dataset files to be bulk loaded is input to the tool. An overview of the input to and output from the Database Creator is shown in Figure A.2. A more detailed view of the workings of the Database Creator is shown in Figure A.3.

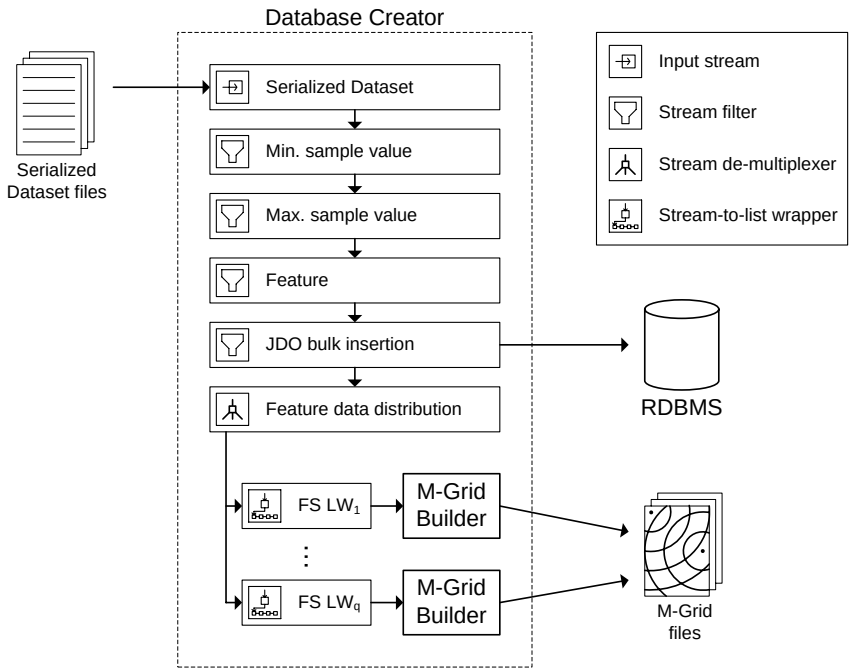

Figure A.3: *Inside the Database Creator, the Dataobjects from the input Datasets are filtered using streams. During this, the data is bulk loaded into the RDBMS. After filtering, the Dataobjects are distributed to the M-Grid Builders that handle initialization of the M-Grids. The state information is stored in the M-Grid files shown.*

The set of Dataset files are turned into a continuous stream of Dataobjects to be treated one at the time. The Dataobjects in the stream are filtered using different stream filters. A stream filter is a class satisfying two things: Its constructor takes at least an input stream as input, and it provides a nextObject() method that returns the next object in the stream after applying the filter. The following stream filters have been implemented:

- **Feature filter** removes unused features from the FSS part of the Dataobjects.
- **Min. sample value filter** passes only Dataobjects having a minimum amount of sample values in the FSS part.
- **Max. sample value filter** cuts off sample values after the specified maximum number of sample values.
- **Sample value averager** averages every given k number of sample values in order to reduce the amount of feature data.
- **ID Assigner** assigns a unique ID to every Dataobject passing the filter – all Dataobjects are preserved.
- **JDO Inserter** inserts the Dataobjects in the RDBMS using bulk insertion. In this process the Dataobjects are also assigned unique IDs specified by the JDO implementation (JPOX).

Implementing new filters is straightforward, and the possibilities are almost unlimited.

The next step is to create the index structures using what is left of the Dataobjects not removed by the filters. All index structures (M-Grids) defined in the configuration must have feature data from all Dataobjects, but necessarily for the same feature. Therefore, we have created a demultiplexer class that distributes the Dataobjects in the input stream to a number of input streams – one for each index structure to be created.

The class handling the initialization process of the M-Grid, the M-Grid Builder, requires a list of feature data (feature sequences). The distributed input streams are converted into lists using stream-to-list wrappers called feature sequence list wrappers. Each such wrapper stores the Dataobjects in the stream on disk and provides both list iterations and retrieval by index number afterwards. For efficiency, an attached LRU buffer holds the Dataobjects most used in memory.

The output from the Database Creator is an updated RDBMS containing the Dataobjects from the Dataset files, and a number of files needed for the M-Grid index structures to start up.

A.7 The Query Server (Middleware)

The Query server is what makes the parts in the middleware a complete system. It is the main process of the system, and it handles all communication with the client applications. The Query Server is described in more detail in Chapter 2. An overview of the input to and output from the Query Server is shown in Figure A.4.

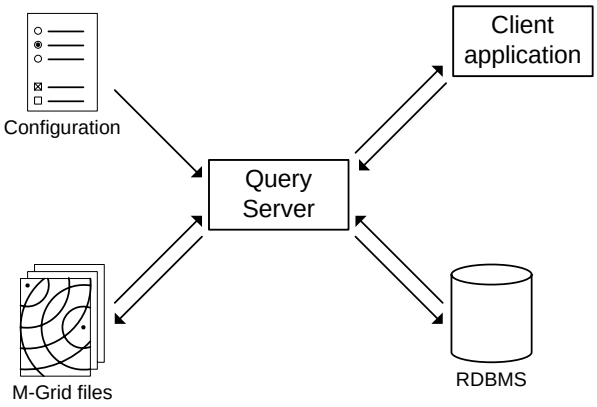

Figure A.4: *The Query Server uses all the information generated by the other tools. The configuration is used for connecting to the RDBMS and for loading the M-Grid index structures.*

The configuration used in the Database Creator is reused here, since the Query Server needs most of the same information used in the Database Creator. The Query Server connects to the underlying RDBMS using the information provided in the configuration. Furthermore, it loads all index structures using the names of the M-Grid files.

A.8 The WebPlayer

The WebPlayer is meant as an alternative to a normal GUI player that uses our system as back-end. In addition being a music player, is it also meant as a showcase for the features that the system exposes to client systems. A description of the features of the WebPlayer and the setup in which it is uses can be found in Section 5.2. The WebPlayer can be seen as the Client Application in Figure A.4.

Appendix B

The Configuration File

The entire middleware, no matter if it is constructing the indices or is running as a Query Server, is configured by a central configuration file. The configuration file is based on XML and formatted in an easy-to-understand format. Every single setting can be configured in the configuration file, and is easy to adapt to new settings if, e.g., alternative index implementations needs alternative settings.

The use of the configuration file can be split into two parts. The settings used by the Query Server, and the settings used by the DB Creator.

The following is an example of the part of the configuration file used by the Query Server:

```
<dbconfig>
  <!-- Index structures -->
  <indexconfig>
    <!-- 1st MGrid -->
    <index>
      <type>MGrid</type>
      <settingsfile>data/mgrid1.settings</settingsfile>
      <indexfile>data/mgrid1</indexfile>
      <useclustercache>true</useclustercache>
      <usedistancematrix>true</usedistancematrix>
      <pivots>4</pivots>
      <rings>10</rings>
      <feature>AudioSpectrumEnvelopeType</feature>
      <distfunc>Manhattan</distfunc>
      <clustering>
        <type>ALAQT</type>
        <clusters>102</clusters>
        <iterations>4</iterations>
      </clustering>
      <pivotselection>
        <type>FullPivotSelector</type>
        <pivotsamples>4</pivotsamples>
        <percentsamples>10</percentsamples>
      </pivotselection>
    </index>
    <!-- More MGrids -->
    <index>...</index>
  </indexconfig>
  <!-- Query Server port -->
  <port>45555</port>
  <!-- Data Mapper connection -->
  <dsctype>JDBC</dsctype>
</dbconfig>
```

The Database Creator uses the same configuration for the indices (indexconfig tag), and besides that it uses the following settings:

```
<dbconfig>
  <!-- Input Datasets -->
  <datasetconfig>
    <dataset>
      <dsfile>rock.mp7.ds</dsfile>
    </dataset>
    <dataset>
      <dsfile>soft.mp7.ds</dsfile>
    </dataset>
    <dataset>
      <dsfile>electronic.mp7.ds</dsfile>
    </dataset>
  </datasetconfig>
  <!-- Filter tree -->
  <filtertree class="jdoinserter">
    <!-- JDO Inserter filter -->
    <jdoconfig>jdo.properties</jdoconfig>
    <maxbatchsize>10000000</maxbatchsize>
    <!-- Feature filter -->
    <input class="featurefilter">
      <features>
        <feature>
          <name>AudioSpectrumEnvelopeType</name>
        </feature>
      </features>
      <!-- Max. sample values filter -->
      <input class="maxsamplevaluefilter">
        <keepcount>600</keepcount>
        <!-- Min. sample values filter -->
        <input class="minsamplevaluefilter">
          <mincount>600</mincount>
        </input>
      </input>
    </input>
  </filtertree>
</dbconfig>
```

Appendix C

Known Issues

C.1 Timeout in Webplayer/Webshop

Both of the case study implementations uses PHP as the server-side programming language. In PHP there is a default timeout for execution of the scripts and this is normally 30 seconds. The timeout can be set in the configuration of PHP, and can in some setups be changed during run-time – this is however not the common setup.

This basically means that if a query takes longer than the specified max execution time, then the script will stop executing.

This is handled in in the case studies by simply giving a "no result found" reply if the timeout is reached – hence the user things that the query did not have a result.

C.2 Configuration Parsing

If the user inputs an incorrect tag/setting in the configuration file, then main middleware wont start. It does however handle this by trowing a descriptive error message telling where the error in the configuration file the error might be. This makes the issue seem like it is not an issue but rather a feature of the system.