Efficient Retrieval from Vast Music Collections

Master's Thesis

Claus Åge Jensen
Ester Moses Mungure
Kenneth Rand Sørensen

June 13th, 2006

Department of Computer Science, Aalborg University, Denmark

The Faculty of Engineering and Science
Aalborg University

Department of Computer Science

Efficient Retrieval from Vast Music Collections

PROJECT PERIOD:
DAT6, F10S, CIS4,
February 1st, 2006 -
June 13th, 2006

PROJECT GROUP:
E1-209

STUDENTS:
Claus Åge Jensen
Ester Moses Mungure
Kenneth Rand Sørensen

SUPERVISOR:
Torben Bach Pedersen

COPIES: 7

REPORT PAGES: 61

APPENDIX PAGES: 6

TOTAL PAGES: 67

ABSTRACT:

When considering the development of musical digitization, new challenges emerge within the field of Music Information Retrieval, where our focus of research is querying on vast music collections. For that purpose we introduce and evaluate the *Music On Demand framework* where songs are queried as a continuous stream. When querying songs a listener is able to influence the songs ahead in the stream dynamically by performing the following actions: *play similar songs*, *play random songs*, *skip songs*, *restrict collection* and *specify collection*. In order to do so, a generic music data model and associated query functionalities are defined.
Applying bitmap indices to index metadata as well as musical similarity derived from the musical content, we enable support for efficient retrieval within vast music collections by the use of bit-wise operations. The retrieval process concerns a combination of both the metadata and the similarity of songs. In this context we examine the use of the *Word-Aligned Hybrid* compression scheme and the *Attribute Value Decomposition* technique for representing content based similarity.
Experimental test results show that our framework implementation ensures efficient access to music within vast music collections, at the cost of only a small additional space consumption when compared to the stored music files.

Contents

Efficient Retrieval from Vast Music Collections

Claus Åge Jensen <caj@cs.aau.dk>
Ester Moses Mungure<moses@cs.aau.dk>
Kenneth Rand Sørensen <krs@cs.aau.dk>

June 13th, 2006

Abstract

When considering the development of musical digitization, new challenges emerge within the field of Music Information Retrieval, where our focus of research is querying on vast music collections. For that purpose we introduce and evaluate the *Music On Demand framework* where songs are queried as a continuous stream. When querying songs a listener is able to influence the songs ahead in the stream dynamically by performing the following actions: *play similar songs*, *play random songs*, *skip songs*, *restrict collection* and *specify collection*. In order to do so, a generic music data model and associated query functionalities are defined.

Applying bitmap indices to index metadata as well as musical similarity derived from the musical content, we enable support for efficient retrieval within vast music collections by the use of bit-wise operations. The retrieval process concerns a combination of both the metadata and the similarity of songs. In this context we examine the use of the *Word-Aligned Hybrid* compression scheme and the *Attribute Value Decomposition* technique for representing content based similarity.

Experimental test results show that our framework implementation ensures efficient access to music within vast music collections, at the cost of only a small additional space consumption when compared to the stored music files.

1 Introduction

In recent years the field of music distribution has changed from being medium based to becoming digitized, in particularly with the advent of lossy compression techniques such as the MP3 format. Moreover, the accessibility of digital music is constantly improving as high speed Internet connections are becoming more and more common. As a consequence of this development, digital music stores are emerging, causing the consumer

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.

behaviour to change through a high degree of accessibility and collaborative filtering techniques. Based on these techniques, personal recommendations are derived based on the preferences of other individuals.

As a consequence of easy access to music and decreasing storage costs, the current tendency is that personal music collections grow in size. Thus, we are facing the challenge of supporting query functionalities on *vast music collections* where only limited or no prior knowledge about the content of the music collection is available.

To ensure efficient handling of query functionalities with respect to vast music collections, this paper introduces the *Music On Demand framework* henceforth referred to as the MOD framework. The main characteristic of the framework is to combine metadata of music with musical similarity derived from the musical content of the respective songs. In doing so, two basic properties of the framework is ensured. First, using metadata it is possible to browse the entire music collection in order to select subsets thereof, e.g., all songs by U2. Second, with respect to the musical similarity between songs, the framework ensures an efficient foundation for indexing content based similarity between songs.

The most significant contribution of the MOD framework, is the introduction of bitmaps used for indexing purposes in order to ensure efficient management of both metadata and content based similarity. Representing the entire music collection as well as subsets thereof as bitmaps, we are able to use bit-wise operations to ensure efficient generation of multi attribute subsets representing, e.g., all songs of Madonna released in the year 2005. These subsets may in turn be applied as restrictions to the entire music collection. Similarly, using bitmaps to represent groupings of similar songs with respect to a given base song, we are able to identify and retrieve similar/dissimilar songs efficiently using bit-wise operations. We believe to be the first to use bitmap indexing techniques to facilitate retrieval and restriction queries in vast music collections. An important aspect of our contribution in this connection is to use bitmaps to combine metadata with content based music similarity. As a consequence of this combination, we are able to ensure low response times on retrieval queries, where similarity requests are applied on a restricted music collection.

Relating to the experimental test results of the MOD framework, 10.6GB of space is required to index 100,000 songs. When compared to the space consumption required to store the actual digitized music collection, the indices comprise only a minor overhead of 2.7% of the stored audio files when considering an average file size of 4MB. Moreover, when retrieving songs from a restricted music collection, the MOD framework is able to reply with a similar song in 23ms and a random song in 14ms in average.

Based on the concept of impulsive user interactions, the MOD framework enables listeners to perform the following five basic actions; *skip songs* that are disliked, *play similar songs* with respect to a given base song, *play random songs*, perform *metadata restrictions* supporting a hierarchical metadata structure and manage *user specified collections*. Compared to our previous research considering playlist generation [JMS05], this paper describes a different approach using infinite streams of music rather than the well-known concepts of static playlists. Hence, instead of specifying the content or

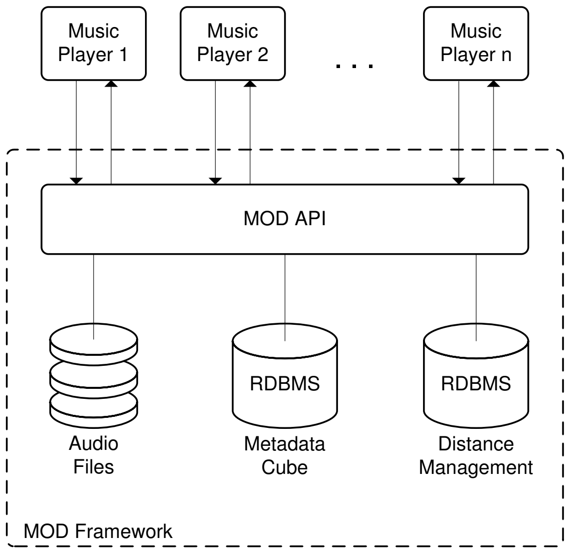


Figure 1: *Architecture of MOD framework.*

characteristics of the generated playlist on beforehand, the stream abstraction enables a listener to influence the songs ahead in the stream by performing one of the five basic actions.

In Figure 1 a simplified illustration of the framework architecture is presented. In this connection, the audio files, the metadata and the distances between songs are maintained by the framework where the metadata and the distances are stored separately in an RDBMS. Using the framework, a music device running a music player application is able to request songs one at a time. In addition, multiple music player applications are able to interact with the framework simultaneously. In the remainder of this paper we refer to any music device running the music player application simply as a *music player*.

The organization of the remainder of this paper is as follows. Initially, in Section 2 we describe related work. In Section 3 we present a motivating use case scenario as well as a formalization of the system interplay between system objectives and usage of the MOD framework. Section 4 defines a music data model along with formal descriptions of the associated query functionalities. The technical design of the MOD framework is described in Section 5, where emphasis is put on bitmap indexing techniques. In Section 6, query evaluation techniques are described. In Section 7 we present experimental results for an implementation of the MOD framework. Finally, in Section 8 we conclude on our research with respect to using bitmaps for indexing of vast music collections. Moreover, the section presents a number of suggestions for future work.

Additionally, Appendix A presents the database table definitions for the applied tables. Appendix B describes the functionality of the MOD API. The enclosed CD-ROM contains an electronic copy of this paper and the source code for the MOD framework.

2 Related Work

Within the field of MIR (Music Information Retrieval), much effort has been put into the task of enabling music lovers to explore individual music collections as is the case for, e.g., [NDR05, Lüb05]. Within this context, several research projects, e.g., [ME05, Pam05], have been conducted in order to pursue a suitable similarity measure for music, for which reason a feature representation of the musical content is required. One such feature representation is constituted by the MFCC (Mel-Frequency Cepstral Coefficients) features known from the field of speech recognition [JPH00]. An alternative approach to the MFCCs is found in the MPEG-7 audio features as specified in [SS02]. MPEG-7 uses well-defined components such as the beat, pitch, etc. of the music, whereas MFCCs consider the overall musical impression with respect to human perception.

When considering indexing of high dimensional musical feature representations, existing indexing techniques such as, e.g., the M-grid [DN05] and the M-tree [CPZ97] can be applied. However, as a consequence of the subjective nature of musical perception, the triangular inequality property of the metric space typically can not be obeyed for a similarity measure. Hence, as the M-tree and the M-grid, a.o., rely on the use of a metric space, these turn out to be insufficient. As a consequence, additional techniques are to be applied, to ensure a suitable foundation for musical similarity search.

In accordance with the different feature representations of musical content, the current research is going in the direction of automating the task of finding similar songs within music collections. In this context it is suggested that the combination of several similarity measures ensures the most valid results. In [AP02] the authors describe a similarity measure of what is denoted as the *global timbre* of music. Similarly, in [PFW05], the authors present a similarity measure where spectral similarity is combined with three additional similarity measures based on fluctuation patterns. An alternative approach used to measure similarity between songs is described in [LS01], where the authors compare signatures of songs using EMD (Earth Mover's Distance) [RTG00]. The signatures are constructed based on k-means clustering of MFCCs in accordance with their location in the MFCC vector space. The objective of clustering is thus to group feature vectors such that the location of feature vectors of the same group also are close in the MFCC vector space. The signature for a given song is then based on the mean, covariance and weight of each cluster.

The research presented in this paper is based upon the results of the conducted research within the field of content based music similarity in general. In particular, we choose to extend this concept further by combining the musical content similarity and metadata of music with respect to the retrieval process of songs. For this purpose we apply an efficient indexing technique by way of bitmap indices within the MOD framework.

To ensure efficient retrieval of read-mostly data, bitmap indices are popular data structures for use in commercial data warehouse applications [KRT$^+$98]. In addition, bitmap indices are used with respect to bulky scientific data in order to represent static information. One such approach is described in [SDHS00], considering High-Energy Physics.

As a different approach to automated similarity measures, the company Pandora Media, Inc. uses a human expert panel in order to capture the musical details of songs in association with the Music Genome Project$^{\text{TM}}$ [Pan06]. Though the task of finding similar songs within Pandora$^{\text{TM}}$ to some extent is exposed to subjective opinions of the expert panel, it is conceptually very similar to our work. Using dynamic user interaction such as “I like this song” and “I don’t like this song”, Pandora$^{\text{TM}}$ is able to generate a personal playlist that may be influenced dynamically. With respect to the MOD framework, we rely on automated similarity algorithms rather than a human expert panel, to ensure both objectiveness and scalability. Furthermore, we are able to perform explicit metadata restrictions on music collections.

An approach using musical content to find similar songs is described in [PPW05], where immediate user interaction in terms of skipping behaviour is used to restrict the music collection. Upon returning a similar song to the listener, it is determined whether the song to return is closer to either a previously accepted song or a skipped song. In either case the distances between the individual songs are to be consulted, in order to determine what song to return. Unlike this approach we do not rely on the actual distances when determining what song to return, as songs are clustered into groups of similar songs. As a consequence of this grouping, a more compact representation of the songs is possible. Moreover, as all songs within a group are considered alike with respect to similarity, any of the songs can be retrieved without compromising the quality of the retrieved song. Hence, the retrieval process becomes more efficient.

In most commercial media players such as Winamp$^{\text{TM}}$, the metadata of music presumes a flat structure. However, to enable an enriched description of the metadata of music, we choose explicitly to view metadata in the form of a *multidimensional cube* known from the literature of multidimensional databases [PJ05, Tho97]. The metadata of music is thus considered as a number of metadata dimensions, modelled in a hierarchical manner, which constitutes a multidimensional cube. Through this approach we are able to select songs in accordance with the individual levels of a given hierarchy of a metadata dimension.

Looking at the current organization of metadata attributes, e.g., the genre of a song, there exists no standard for what genres to make available, and even the different compression formats have their own types of genres. To clarify, ID3-tags used in the MP3 format contains a fixed number of different genres [ID305], whereas the Ogg Vorbis format has an unlimited number of genres as anything specified as text is allowed to represent genres [xip05]. However, the authors of the presented paper are convinced that a standard for the metadata of music will emerge as the use of digitized music increases. To support a richer description of the metadata of music, it is moreover expected that metadata is becoming hierarchical to facilitate navigation within vast music collections. Also, to better classify music into genres, ongoing research, e.g., within the Intelligent Sound research project [Int06a], is being conducted in order to perform genre classification using the musical content.

In an earlier research paper, we investigated the possibilities to explore music collections based on the musical content of the associated audio files [JMS05]. In order to present the chosen songs, a playlist was then generated as a sequence of songs. In the context of this paper, however, it is believed that precomputed playlists do not suit the impulsive behaviour of humans interacting with a music player. Therefore, the concept of playlist generation is reduced to the task of returning a single song based on dynamic user interaction. Still, over time, the list of played songs resembles a playlist.

3 Motivation and System Interplay

In this section we study a motivating use case scenario concerning music lovers who wish minimal interaction with their music player of choice, as a consequence of physical restraints. A simple example of a physical restraint may involve interacting with a portable music palyer while wearing gloves. For the sake of convenience we restrict this study to focus solely on the exercise routines of a single person named Jane. Additionally, the use case scenario describes a vision of usage for a music player handling a vast music collection, where traditional navigational methods are both cumbersome and time consuming [JMS05]. The study serves as the basis for presenting the usability of the MOD framework in terms of the interplay between a listener and a music player using the framework.

3.1 A Vision of Usage

This quiet winter morning Jane is going for a run in the hilly terrain around the area where she lives, for which reason she chooses to wear lined gloves to keep her fingers from freezing. To accompany her during her regular jogging sessions, Jane has purchased a portable music player, as she has learned that the music accompaniment to exercise provides an important beneficial effect to the exercise experience [Mas86].

In the last couple of months since Jane bought the music player, she has marked a number songs in her entire music collection that she believes provide her the best motivation for her workouts. However, for today's jogging session Jane finds herself in a dilemma, as she would like to listen to some new releases recently added to the collection as well as her favourite workout songs. Hence, prior to engaging in the impending session, Jane restricts her music collection to respond only with songs released in the year 2006 and songs marked as her favourite workout songs. For the remainder of the case study, we refer to this restricted music collection simply as the *music collection*.

Entering the dull sunlight of dawn, Jane turns on the music player and engages in her jogging exercise with great enthusiasm. To ensure a variation in the songs being listened to, she initially requests the music player to play songs chosen randomly among the songs in the music collection. Soon a new and unknown song is being played and Jane enjoys how the beat of the song motivates her to perform even better. To retain this motivation, Jane decides to listen to songs similar to this particular new song. To do so, she applies a single push on the *Similar Song* button of the music player, causing the playing mode to change. Thus, for as long as the music collection allows, and presuming that Jane does not change the mode of the music player, a steady flow of high beat and energized music is ensured, just as Jane prefers at this particular time. Upon pushing the button, Jane again is pleasantly surprised by the ease in interacting with the music player, even though the sun has still not raised over the horizon and she wears lined gloves.

After a while, the terrain changes and Jane decides once again to listen to songs chosen randomly from the music collection. While running uphill the music player starts to play a new song that does not at all appeal to the taste of Jane. Afraid that more songs of the same style is going to be played during this stage of exhaustion, Jane decides to eliminate all resembling songs from being retrieved. She does so by pushing the *Skip* button on the music player. Fortunately, the following song is one of Janes favourite workout songs performed by the group AC/DC. She knows that the music collection contains numerous songs performed by this group, and she believes firmly that the sound of AC/DC will help her climb the hill. Hence, she pushes the *Similar Artist* button and restricts the music collection even further in accordance with the artist name of the currently playing song. For as long as Jane does not interact with the music player, she will be presented with a continuous stream of randomly chosen AC/DC songs.

Reaching the top of the hill, Jane believes it is about time to head back to her home. She feels exhausted from climbing the hill and decides to slow down on the way back in order to recuperate. For this purpose she enters the menu of the music player, and changes the initial restriction (songs released in the year 2006 and favourite workout songs) to include only her favourite relaxation songs.

Back home in the warmth of her living room, Jane reflects on the just completed exercise session. Even though the music collection used for this exercise session is vast, she was required to interact with the music player as few as four times during the whole run, and still she was presented the songs which corresponded with her current state of mind. Additionally, because Jane is being motivated at the right times with the right music, the benefit of the exercise is enhanced and her performance increased [BT90, TLL$^+$04].

3.2 Formalization of System Interplay

In Table 1, three system objectives are presented which correspond to the characteristics of the described usage. The purpose of these objectives is to formalize the interplay between a listener and the music player using the MOD framework in accordance with the use case scenario.

The first objective, *Replies with a song*, states what possible uses ensure that the listener is supplied with a steady stream of music in accordance with the playing mode of the music player.

The second objective, *Affects music collection*, determines what uses cause a restriction to the music collection. In case of the skipping behaviour, the restriction is implicitly induced in accordance with the musical content of a skipped song, indicating that a song similar to a skipped song can not be retrieved from the music collection. Metadata on the other hand is used solely for explicit restrictions. Worth emphasizing in this connection is restriction on similar metadata, where the metadata of the current playing song may be used to restrain the music collection.

The last objective, *Uses current song*, states which of the uses apply the current playing song as basis for song retrieval or a collection restraint. A special case for this objective however, is the marking of favourite songs, as this usage apply to neither of the previously mentioned objectives. The task of marking a song as a favourite song implies that a separate collection of songs is maintained to support the amendment of favorite songs.

Table 1: *Table describing the interplay between usage and system objectives.*

	System Objectives		
Usage	Replies with a song	Affects music collection	Uses current song
Find random songs	yes		
Find similar songs	yes		yes
Skip song		yes	yes
Restrict on metadata		yes	
Restrict to similar metadata		yes	yes
Mark favourite songs			yes

To summarize on the interplay between usage and system objectives, the nature of interaction indicates a dynamical environment, where the listeners may react impulsively in accordance with their current state of mind. Hence, the framework is to be susceptible to frequent interaction while still allowing for long-term planning of what music to make available for the listener. As it is impossible to foresee patterns in the interaction process with the music player, a *single song approach* is pursued, where a playlist is generated dynamically while requesting one song at a time. Hence, rather than stating on beforehand what music to hear using static playlists, listeners may intervene in the playlist construction and thus explicitly put influence on the content of the dynamically constructed playlist.

4 Data Model and Query Functionality

The purpose of this section is to describe a music data model and the associated query functionalities for music retrieval. Initially, we define a data model followed by a description of the formal semantics of the operators to be used on the music data model. As the listener may choose to interact with the music player at any point in time, the operations dealing with music retrieval only returns a single song at a time.

In the following, a subscript notation, $Operator_{parameters}(arguments)$, is used to identify parameters for the operators. Furtermore, $\mathbb{N}$ denotes the domain of positive integers.

4.1 Music Data Model

This section describes a music data model constituted by a number of elements and their corresponding domains. The purpose of these elements is to support the functionalities of the MOD framework by applying each of them on an instance of the data model in order to fulfil the usages specified in Section 3.2. Definitions 1 to 3 of the presented music data model are based on our previous research described in [JMS05].

To extend the usage of metadata selection we introduce a metadata dimension in order to apply an abstraction to a hierarchical representation of the metadata of music. This metadata dimension is described in Definition 1 where the hierarchical ordering of the metadata of music is described as two partially ordered sets (posets). The first poset represents the hierarchical ordering of dimension levels and the second poset represents the hierarchical ordering of the dimension values. The idea used to model dimensions as two posets is inspired by [PRP02]. The use of hierarchies reflects a generic approach for the support of both flat and hierarchical metadata structures. As for the latter case, involving a hierarchical structure, elements such as year of release may be grouped into a superordinate dimension level, forming, e.g., a decade.

Definition 1 (Metadata dimension) *A metadata dimension d_j is defined as a 2-tuple $d_j = (L_j, V_j)$, where L_j is a poset of* dimension levels *and V_j is a poset of* dimension values. *Let $\mathbb{D}$ be the domain containing all such metadata dimensions.*

A poset of dimension levels L_j is defined as a 2-tuple $(LN_j, \sqsubseteq_j)$, where $LN_j = \{ln_{j1}, \ldots, ln_{jn}\} \in 2^{\mathbb{LN}}$ is a set of unique level names, and $\mathbb{LN}$ is the domain of all possible level names. $\sqsubseteq_j$ is a partial order on the level names in LN_j with $\top_j \in LN_j$ and $\bot_j \in LN_j$ being the unique top and bottom elements of $\sqsubseteq_j$, respectively. A level name $ln_{jk} \in LN_j$ is a name identifying a set of dimension values, where $1 \leq k \leq n$. The poset of dimension levels is referred to as a metadata dimension schema.

Let $\mathbb{DV}$ constitute the domain of all possible dimension values. Then the function $LevelValues : \mathbb{LN} \rightarrow 2^{\mathbb{DV}}$, takes a level name as input and returns the set of dimension values associated with the given level name.

A poset of dimension values V_j is defined as a 2-tuple $(DV_j, \sqsubseteq_{d_j})$, where $DV_j = \bigcup_k LevelValues(ln_{jk})$ is the set of all dimension values from all the level names in metadata dimension $d_j \in \mathbb{D}$ for $1 \leq k \leq n$. $\sqsubseteq_{d_j}$ is a partial order on the set of dimension values DV_j. The union used is a disjoint union that ensures the uniqueness among dimension values from all levels, i.e., the sets of dimension values for different levels are pairwise disjoint. The dimension value $\top_{d_j} \in DV_j$ is the unique top element of the partial order $\sqsubseteq_{d_j}$. With the notation $dv \in d_j$ we shall denote a dimension value belonging to a metadata dimension $d_j \in \mathbb{D}$, where $dv \in DV_j$. The poset of dimension values is referred to as a metadata dimension instance.

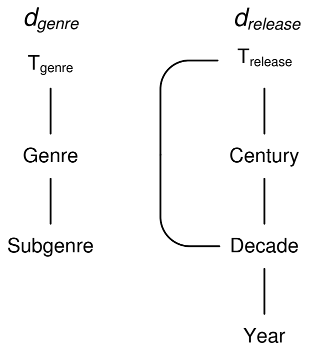
Given two level names $ln_{jk}, ln_{jp} \in \mathbb{LN}$, where both ln_{jk} and ln_{jp} are in a metadata dimension $d_j \in \mathbb{D}$, the partial order $\sqsubseteq_{d_j}$ between dimension values satisfies that $dv_1 \sqsubseteq_{d_j} dv_2$, iff $dv_1 \in LevelValues(ln_{jk}),\ \ dv_2 \in LevelValues(ln_{jp})$ and $ln_{jk} \sqsubseteq_j ln_{jp}$.

As given by Definition 1, a metadata dimension consists of both dimension levels and dimension values, where a dimension level has a number of associated dimension values. Using posets to model hierarchies we achieve that both regular and irregular dimension hierarchies are supported. Irregular hierarchies occur when the mappings in the dimension values do not obey the properties stating that a given hierarchy should be *onto*, *covering* and *strict* [PJ05]. Informally, a hierarchy is onto if the hierarchy is balanced, covering when no paths skip a level and strict if a child in the hierarchy has just one parent. In Section 5.4 we are going to elaborate on how irregular hierarchies are handled. To illustrate the intuition behind the hierarchical structure of the metadata dimensions, consider Example 1.

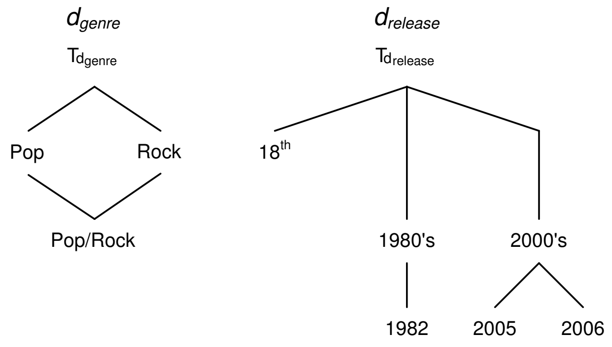
Example 1 *Let a metadata dimension $d_{genre} = (L_{genre}, V_{genre})$ represent the genre of songs and the corresponding schema. The poset of dimension levels L_{genre}, defined as $(LN_{genre}, \sqsubseteq_{genre})$, thus consists of the level names $LN_{genre} = \{\top_{genre}, Genre, Subgenre\}$ and a partial order $\sqsubseteq_{genre}$ on these. The partial order ensures an ordering of the level names in the d_{genre} dimension and is given by the reflective and transitive closure of the order $Subgenre \sqsubseteq_{genre} Genre \sqsubseteq_{genre} \top_{genre}$. The schema for the metadata dimension d_{genre} is presented in Figure 2(a).*

The poset of dimension values V_{genre}, is defined by the 2-tuple $(DV_{genre}, \sqsubseteq_{d_{genre}})$ and consists of all the dimension values $DV_{genre} = \{\top_{d_{genre}}, Rock, Pop, Pop/Rock\}$, and a partial order $\sqsubseteq_{d_{genre}}$ on these. This partial order is based on the partial order of level names $\sqsubseteq_{genre}$, and is given by the reflective and transitive closure, shown as the metadata dimension instance d_{genre} in Figure 2(b). This example illustrates a non-strict hierarchy, as the dimension value "Pop/Rock" has two parents, namely the dimension values "Pop" and "Rock"

In addition, the schema and an instance for the metadata dimension $d_{release}$ is shown in Figure 2. The metadata dimension $d_{release}$ is constructed in a similar fashion as the



(a) *Metadata dimension schemas.*



(b) *Metadata dimension instances.*

Figure 2: *Schema and instance for the metadata dimensions d_{genre} and $d_{release}$.*

metadata dimension d_{genre}. The metadata dimension $d_{release}$ implies a non-covering hierarchy, as the songs are not required to have a century associated. Omitting a given level, the number of steps required to traverse the dimension is reduced, which in turn improves the usability. Hence, from a user point of view, information may be added only where it suits a given purpose. Aside from the release hierarchy being non-covering, it is also non-onto as the songs from the 18^{th} century have no decade associated.

■

The metadata of music is composed of descriptive attributes such as artist, title, etc. For the remainder of this paper, we assume the existence of the metadata attribute and their associated metadata dimension schemas as shown in Figure 3. The metadata attributes are presented as dimension values in Definition 2, where a metadata item and the corresponding schema are defined.

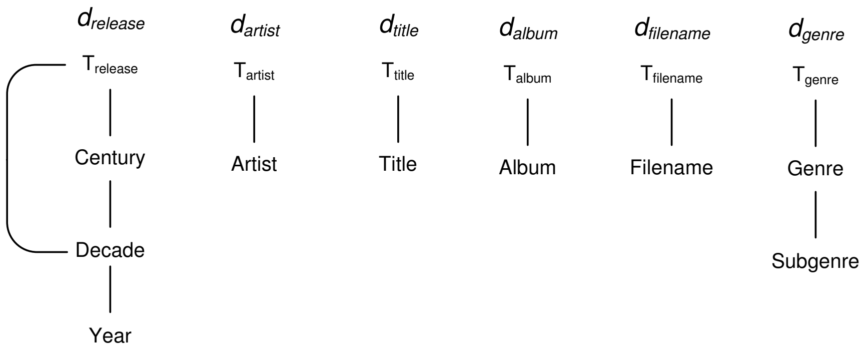


Figure 3: *Schemas representing the metadata dimensions of a song, as used within the context of this paper.*

Definition 2 (Metadata item and schema) *A metadata item m is defined as an n-tuple $m = (dv_1, dv_2, \ldots, dv_n)$, where $n \in \mathbb{N}$ and dimension value $dv_j \in \mathbb{DV}$, for $1 \leq j \leq n$. A metadata schema ms for the metadata item m, is an n-tuple $ms = (d_1, \ldots, d_n)$, where the metadata dimension $d_j \in \mathbb{D}$ and the dimension value $dv_j \in d_j$ for $1 \leq j \leq n$. Let $\mathbb{M}$ be the domain of all such metadata items, and let $\mathbb{MS}$ be the domain of all such metadata schemas. In connection with the metadata item and schema two functions exists. The function $Schema : \mathbb{M} \rightarrow \mathbb{MS}$, takes a metadata item m as input and returns the corresponding metadata schema ms. The function $MetaValue : \mathbb{M} \times \mathbb{D} \rightharpoonup \mathbb{DV}$ takes a metadata item m and a metadata dimension d_j as input and returns the dimension value dv_j from metadata item m, which corresponds to the metadata dimension d_j given that $dv_j \in d_j$.*

Example 2 *To illustrate the structure of the metadata item, assume the existence of the three metadata dimensions d_{artist}, d_{title} and d_{genre}. In this case the song of the "Rock" genre with the title "The Fly" performed by the band "U2" is represented as the metadata item $m =$ ("U2", "The Fly", "Rock"). The associated metadata schema ms is given by $ms = Schema(m) = (d_{artist}, d_{title}, d_{genre})$.*

■

In Definition 3 a song and its associated domain is described. For this purpose, the existence of a feature representation of the musical content of an audio file is assumed. In addition, the function $dist$ may be used to calculate the content based similarity between two songs with respect to their associated feature representations.

Definition 3 (Song) *A song s is defined as a 2-tuple $s = (m, f)$, where $m \in \mathbb{M}$ is a metadata item and $f \in \mathbb{F}$ is a feature representation of the musical content of the song, where $\mathbb{F}$ is the feature domain for all feature representations. Let $\mathbb{S}$ be the domain containing all such songs. A function $dist : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}$, that obeys the symmetric and identity properties of a metric space, takes two songs as input and returns the distance between the feature representations associated with the songs as a real number.*

As users have audio files and not songs as presented in Definition 3, we assume the existence of a function $SongInstanceGenerator$ that constructs a song instance from a given audio file. During this process a feature representation of the audio file is generated, and the available metadata are extracted to be included in the song instance. Additionally, to provide the full picture we introduce the function $FetchMusicFile$ that returns the music filename associated with a given song instance. To retrieve the music filename, the function uses the filename from the $d_{filename}$ metadata dimension presented in Figure 3.

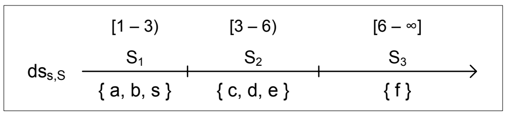
When considering the content based similarity between songs, Definition 4 presents a distance store that introduces an abstraction to the already defined distance function applicable to the feature representations of songs as described in Definition 3. The distance store provides the ability to group songs into partitions based on their respective distances to a given base song. As no unique correct answer exists as to whether two distinct songs are considered similar, the grouping does not compromise the quality of the distance function. Moreover, songs may be grouped into partitions while omitting the base song, which allows for the generation of a composite distance store.

A distance store is said to be a complete partitioning of the distance domain, implying that no partitions are omitted and that no partitions should ever lap over one another. Thus, each partition constitutes a unique and non-overlapping distance interval.

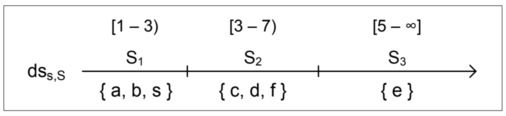
Definition 4 (Distance store) *Given an optional base song $s \in \mathbb{S}$ and a set of songs $S \in 2^{\mathbb{S}}$, a distance store $ds_{[s],S}$ is defined as sequence of sets of songs, $ds_{[s],S} = S_1 \cdots S_n$, where it applies that $S_i \in 2^{\mathbb{S}}$ for $1 \leq i \leq n$ constitutes a partition and $n \in \mathbb{N}$ denotes the number of partitions. In addition, it applies that $S_i \subseteq S$, $\bigcup_i S_i = S$ and $S_j \cup S_k = \emptyset$ for $1 \leq j, k \leq n$, where $j \neq k$. In case the optional base song s is present, it holds that $\forall t \in S_i(\forall u \in S_{i+1}(dist(s,t) < dist(s,u)))$. Let $\mathbb{DS}^n$ be the domain containing all such distance stores, where n corresponds to the number of partitions.*

Within the context of Definition 4, the rearrangement of songs from the set of songs S into partitions, reflects how similar songs are to the base song s. To support this understanding, one can think of the partitioning as starting from partitions holding highly similar songs and gradually degrading towards partitions holding less similar songs. As songs becomes more and more dissimilar, it is likely that the average listener con not judge the difference of the similarity degree. Hence, 10 to 20 partitions is typically sufficient to ensure a diversified distribution of the songs. To ensure a fair basis for comparison with respect to content based similarity all songs within a given music collection should belong to the same distance store domain $\mathbb{DS}^n$, indicating that all associated distance stores are to contain the same number of partitions. In Example 3 the structure of the distance store is illustrated.

Example 3 *Given a base song s and a music collection represented by the set of songs S that contains song s and six additional songs denoted by a through f, we say that $S = \{s \;\; a \;\; b \;\; c \;\; d \;\; e \;\; f\}$. The songs within S have individual distances to the base song s given as 0, 1, 2, 3, 4, 5 and 6, respectively. In Figure 4(a) a valid distance store is illustrated. However, the distance store shown in Figure 4(b) constitutes an invalid distance store, as the song f is located in partition S_2 and song e is located in partition S_3. As a consequence, the intervals of the two partitions overlap for which reason the properties of the complete partitioning are disobeyed.*



(a) *Valid distance store.*



(b) *Invalid distance store.*

Figure 4: *The structure of a distance store.*

■

As it can be seen from Example 3, the base song may be present within the given set of songs. The distance to the base song is in this case zero as we compare the song with itself. Hence, it will be located in the partition representing the most similar songs within the distance store. In the case an application using the MOD framework maintains a history of the played songs, the history will ensure that, when playing similar songs to song s, the song s is chosen for playback only when it is discarded from the history.

4.2 Retrieval Operators

For the purpose of retrieving songs from a music collection, this section presents operators for the retrieval of songs chosen randomly and songs chosen in accordance with the musical similarity derived from the musical content of the songs. Within the retrieval process of either similar or random songs, the aspect of skipped songs is considered.

Initially, Definition 5 presents a helper operator used to perform a union on two distance stores. This operator is essential with respect to handling of skipped songs, as the operator may be used to construct a single composite distance store for all skipped songs. Having all skipped songs represented by a single distance store, we obtain the minimum distances from any skipped song to all the remaining songs within the music collection. This in turn eases the task of finding both similar and random songs as we shall see in Definitions 6 and 7. Hence, based on the composite distance stores it is determinable whether a given song is similar or dissimilar to any of the skipped songs.

Definition 5 (Distance store union) *Given two distance stores $ds_{r,P} = P_1 \cdots P_n \in \mathbb{DS}^n$ and $ds_{t,Q} = Q_1 \cdots Q_n \in \mathbb{DS}^n$ the operator $\cup : \mathbb{DS}^n \times \mathbb{DS}^n \rightarrow \mathbb{DS}^n$ constructs the union of the two distance stores denoted $ds_S = S_1 \cdots S_n \in \mathbb{DS}^n$, where the optional base song is omitted. All the songs contained within the two sets of songs P and Q are represented exactly once in the set of songs S associated with the constructed distance store. Hence, a song u that is an element in both $P_i \in ds_{r,P}$ and $Q_j \in ds_{t,Q}$ is now contained only once in the constructed distance stored such that $u \in S_k$, where $S_k \in ds_S$ and $k = min(i,j)$ for $1 \leq i,j \leq n$. The semantics of the $\cup$ is defined as follows:*

$$ds_{r,P} \cup ds_{t,Q} = ds_S, \text{ where } S = P \cup Q, \; ds_S = S_1 \cdots S_n, \; S_i = (P_i \cup Q_i) \setminus \bigcup_{j=1}^{i-1} S_j \tag{1}$$

In Example 4 we elaborate on the construction of a composite distance store using the distance store union operator.

Example 4 *Using the distance store union operator introduced in Definition 5, Figure 5 illustrates a union performed on the two distance stores $ds_{a,P}$ and $ds_{e,Q}$. Six songs are contained in both distance stores and denoted by a through g indicating that $P = Q$. Within the composite distance store ds_S, it is ensured that any song is represented only once while favouring early occurrences of each of the songs, i.e., a song is represented by the partition holding the most similar songs of the two partitions. Hence looking at the partition S_2 of the distance store ds_S the song d is omitted as song d is already considered by the previous partition S_1. Similarly, in partition S_3 no songs are included as all songs occur in previous partitions.*

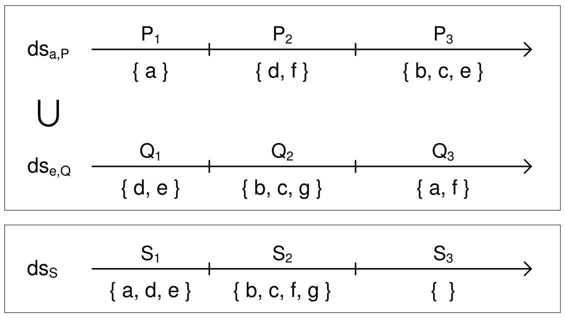


Figure 5: *The union process of two distance stores.*

■

In Definition 6 we describe an operator for retrieval of random songs. In order for the operator to handle the aspect of skipped songs, we assume the existence of a function $Random_n(S)$ that, given a set of songs S, randomly chooses and returns a set of songs containing $n \in \mathbb{N}$ songs from S. For this function it applies that $n \leq |S|$. The purpose of the function $Random_n(S)$ is to ensure the quality of the song returned by the operator described in Definition 6. If $n = 1$ only a single candidate song would be chosen and returned to the listener. However, it may be the case that this song is very close to an already skipped song, for which reason the chosen song is an unacceptable candidate. Instead, choosing several songs as possible candidates and returning only the song least similar to all the skipped songs, the quality of the returned song increases. Moreover, we assume the existence of the function $Random(S)$, that given a set of songs S returns a single song chosen randomly among the songs in the set of songs S.

Definition 6 (Random song) *Given a set of songs $S \in 2^{\mathbb{S}}$, a set of skipped songs $S^{skip} \in 2^{\mathbb{S}}$, a set of played songs $S^{hist} \in 2^{\mathbb{S}}$ and a number of songs $q \in \mathbb{N}$ indicating the retrieval quality, the operator $RandomSong : 2^{\mathbb{S}} \times 2^{\mathbb{S}} \times 2^{\mathbb{S}} \times \mathbb{N} \rightharpoonup \mathbb{S}$ retrieves a song chosen randomly among the songs within the set of songs S. The possible songs that are candidates for retrieval should not be contained within the set of skipped songs S^{skip} or the set of played songs S^{hist}. Moreover, to avoid retrieving a song similar to any skipped songs, while ensuring randomness, a subset of songs is chosen randomly as candidate songs. The distances from all skipped songs to each of the candidate songs are consulted, and the candidate song considered least similar to any of the skipped songs is returned. The semantics of the $RandomSong$ is defined as follows:*

$$\begin{aligned} &RandomSong_{S^{hist},S^{skip},q}(S) = s \in \mathbb{S}, \;\; \text{where} \;\; s = Random(S'_j), \;\; ds_{S'} = S'_1 \cdots S'_n \\ &= \bigcup_{i=1}^{m} ds_{s_i^{skip},S'}, \;\; ds_{S'} \in \mathbb{DS}^n, \;\; 1 \leq j \leq n, \;\; n \in \mathbb{N}, \;\; S^{skip} = \{s_1^{skip}, \ldots, s_m^{skip}\}, \\ &m \in \mathbb{N}, \;\; S' = Random_q(S \setminus S^{hist} \setminus S^{skip}), \;\; \nexists S'_k (j < k \leq n \land |S'_k| > 0), \;\; k \in \mathbb{N} \end{aligned} \tag{2}$$

With respect to the quality of the retrieved song, Definition 6 introduces the parameter q stating a number of candidate songs chosen randomly among the songs contained in the music collection. Within this context, the value of q should be large enough to ensure an acceptable quality of the returned song and still small enough to avoid any unnecessary overhead. However, as the candidate songs are chosen randomly within a vast music collection, chances are that even a small q is able to ensure retrieval of an acceptable song. Assuming that $q = 10$ it is likely that one or more songs are dissimilar to the skipped songs. In Example 5 we elaborate on the retrieval of random songs.

Example 5 *To illustrate the selection process of songs chosen randomly, we initially assume that a music collection represented by the set of songs S contains seven songs denoted a through g. For this collection it holds that the songs a and e are skipped and that the song g is a played song and thus contained in the history. Moreover, assuming*

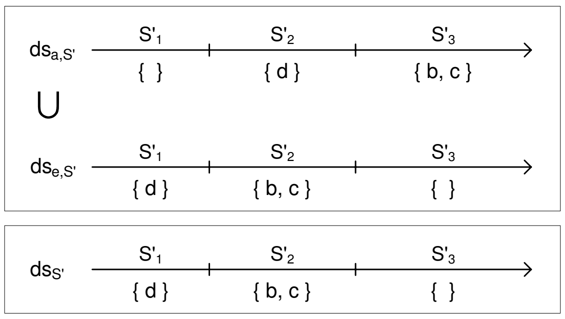


Figure 6: *The selection process of a random song.*

that $q = 3$ we may obtain three candidate songs chosen randomly among the possible songs in the music collection, i.e., songs b, c, d and f. The chosen candidate songs b, c and d are included in the set of candidate songs S'. Based on this information, Figure 6 illustrates the construction of the composite distance store $ds_{S'}$ for the skipped songs a and e. In doing so a union is performed on the distance stores of the skipped songs, $ds_{a,S'}$ and $ds_{e,S'}$, where the distance stores are restricted to contain only songs from the chosen set of candidate songs, S'. The song to return is the candidate song most dissimilar to both skipped songs, which in this case is either the song b or c.

■

In Definition 7 the operator for retrieving songs similar to a given seed song is presented. In addition, an example presenting the selection of a similar song is described in Example 6.

Definition 7 (Similar song) *Given a set of songs $S \in 2^{\mathbb{S}}$, a set of skipped songs $S^{skip} \in 2^{\mathbb{S}}$, a set of played songs $S^{hist} \in 2^{\mathbb{S}}$ and a seed song $s_0 \in \mathbb{S}$, the operator $SimilarSong : 2^{\mathbb{S}} \times 2^{\mathbb{S}} \times 2^{\mathbb{S}} \times \mathbb{S} \rightharpoonup \mathbb{S}$ retrieves a song from the set of songs S most similar to the seed song s_0. The songs that are candidates to retrieval should not be contained within the set of skipped songs S^{skip} or the set of played songs S^{hist}. Moreover, to avoid retrieving a song similar to any skipped song the composite distance store for all skipped songs is consulted with respect to the distance store of the seed song containing all candidate songs. The song considered least similar to any of the skipped songs and most similar to the seed song is returned. The semantics of the $SimilarSong$ is defined as follows:*

$$\begin{aligned}
&SimilarSong_{S^{hist},S^{skip},s_0}(S) = s \in \mathbb{S}, \text{where } \; s = Random\big(S'''_i\big), \;\; ds_{s_0,S'''} = S'''_1 \cdots S'''_n, \\
&S'''_i = S''_i \setminus \bigcup_{j=1}^{i} S'_j, \;\; ds_{s_0,S''} = S''_1 \cdots S''_n, \;\; ds_{s_0,S''} \in \mathbb{DS}^n, \;\; ds_{S'} = S'_1 \cdots S'_n \\
&= \bigcup_{j=1}^{m} ds_{s_j^{skip},S'}, \;\; ds_{S'} \in \mathbb{DS}^n, \;\; 1 \leq i \leq n, \;\; n \in \mathbb{N}, \;\; S' = S'' = S \setminus S^{hist} \setminus S^{skip}, \\
&S^{skip} = \{s_1^{skip}, \ldots, s_m^{skip}\}, \;\; m \in \mathbb{N}, \;\; \nexists S'''_k (1 \leq k < i \wedge |S'''_k| > 0), \;\; k \in \mathbb{N}
\end{aligned} \tag{3}$$

Example 6 *In Figure 7 we elaborate on how similar songs are retrieved in accordance with the constraints induced by the skipped songs. It is assumed that the music collection represented by set of songs S contains seven songs denoted a through g. For this collection it holds that the song g is in the history. The distance store $ds_{s_0,S''}$ holds the associated partitions for a given seed song s_0. The composite distance store $ds_{S'}$ is composed of the skipped songs a and e, respectively. For each of the partitions $S'_i \in ds_{S'}$ the associated songs are accumulated to become included in the partition S'_{i+1} denoted by the parentheses in the figure. Then, for each of the partitions $S''_i \in ds_{s_0,S''}$ the content is restricted with the content of partition $S'_i \in ds_{S'}$. Thus, from the restricted distance store $ds_{s_0,S'''}$ the song to return is any of the songs most similar to the seed song s_0. In this case either of the songs c or f may be returned.*

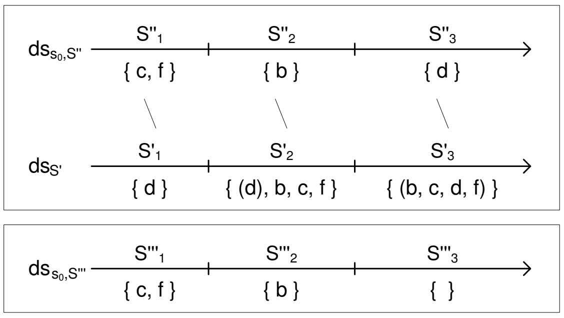


Figure 7: *The selection process of a similar song.*

■

4.3 Restriction Operators

In addition to the retrieval operators, we define a number of restriction operators, which consider the descriptive metadata attributes of the music.

In Definition 8 a selection operator is described, where a music collection is restricted in accordance with a single metadata attribute.

Definition 8 (Select) *Given a set of songs $S \in 2^{\mathbb{S}}$, a dimension value $dv \in \mathbb{DV}$ and a metadata dimension $d_j \in \mathbb{D}$ the operator $Select : 2^{\mathbb{S}} \times \mathbb{DV} \times \mathbb{D} \rightharpoonup 2^{\mathbb{S}}$ constructs a set of songs S' containing exactly the songs that are within and below the dimension value dv when concerning the partial order of dimension values within the associated metadata dimension d_j. The semantics of the $Select$ is defined as follows:*

$$\begin{aligned}&Select_{dv,d_j}(S) = S' \in 2^{\mathbb{S}}, \text{ where } dv \in d_j,\\ &s \in S' \Leftrightarrow s = (m, f) \in S \wedge MetaValue_{d_j}(m) \sqsubseteq_{d_j} dv\end{aligned} \tag{4}$$

In order to restrict on metadata, the set operators $\cup$ and $\cap$ may be applied to perform logical operations on selected subsets of the entire music collection. The subsets are produced using the $Select$ operator. In Example 7 the combined usage of the three operators is explained.

Example 7 *Assuming that the set of songs S represents the entire music collection, a restriction to S denoted as S' may be used to reduce the number of retrievable songs. Restricting the music collection to contain only songs performed by the artist U2 that are released in the 1990's and with songs of the classical genre, the restriction on S is equal to $S' = (Select_{1990's,d_{release}}(S) \cap Select_{U2,d_{artist}}(S)) \cup Select_{Classical,d_{genre}}(S)$*

■

In addition to the described selection operator, Definition 9 describes how a music collection may be restricted in accordance with the metadata for a given song.

Definition 9 (Similar Meta) *Given a set of songs $S \in 2^{\mathbb{S}}$, a seed song $s_0 = (m, f) \in \mathbb{S}$ and a set of metadata dimensions $\{d_1, \ldots, d_n\}$, which is a subset of the metadata dimensions represented by $Schema(m)$, the operator $SimilarMeta : 2^{\mathbb{S}} \times \mathbb{S} \times 2^{\mathbb{D}} \rightharpoonup 2^{\mathbb{S}}$ constructs a set of songs containing exactly the songs from S having metadata associated with the dimension values of the given metadata dimensions. The semantics of the $SimilarMeta$ is defined as follows:*

$$SimilarMeta_{s_0,\{d_1,\ldots,d_n\}}(S) = S' \in 2^{\mathbb{S}}, \text{ where } S' = \bigcap_{i=1}^{n} Select_{MetaValue_{d_i}(m),d_i}(S) \quad (5)$$

To elaborate on the usage of the $SimilarMeta$ operator, Example 8 uses a single song to apply a restriction to the full music collection.

Example 8 *Assume that the song being played by a music player is described by the metadata item $m = ($"U2", "The Fly", "Rock"$)$ as presented in Example 2. Based on this song, the listener of the music may choose to restrict the music collection by one or more of the available metadata attributes. Thus, a possible restriction is to restrict by artist, which ensures that only songs played by U2 are retrievable within the entire music collection.*

■

5 Technical Design

The purpose of this section is to introduce the key design considerations with respect to the MOD framework. The basis for these considerations is the data model presented in Section 4.1. Within this context, the area of responsibility may be divided into two branches constituting a music player and the MOD framework, respectively, as shown in Figure 1 on page 3. Though the music player acts on top of the framework, its area of responsibility is limited to the handling of query parameters with respect to collection maintenance, in order to decide as to whether a given song is valid to return. With regard to the MOD framework, the area of responsibility is more diversified, as both aspects concerning musical content similarity and metadata restrictions are to be taken into consideration.

Prior to elaborating on both branches of responsibility, we relate to the theory of bitmap indexing, as this technique constitutes an efficient indexing approach for processing complex ad hoc queries in read-mostly environments [CI98]. In particularly, this approach seems interesting when considering the human interaction with vast music collections, where such complex ad hoc queries are present in terms of combining retrieval and restriction operators.

Reverting to the objectives of the data model and the query functionalities described in Section 4, sets of songs are essential elements within the presented definitions. We assume that a certain known order of the songs within a music collection exists. Hence, a subset of songs from the music collection can be represented by a bitmap, i.e., a sequence of bits, following the same order as the order of the songs within the music collection, where 1 bits are found only for the songs contained in the subset. Thus, aside from representing the overall music collection of available songs, bitmaps may moreover be used to represent subsets of the music collection such as songs having a similar metadata attribute, the skipped songs or the songs contained in the history. Moreover, having a known order of the songs within the music collection a single song is uniquely identified by its position within the music collection, indicating that the first song is located at position one.

In the following we describe bitmap indices and the applied compression schema. Moreover, with respect to the music player, we briefly cover the handling of query parameters. Finally, we present the design considerations with respect the distance management and handling of metadata.

5.1 Bitmap Indexing

Bitmap indices supply an efficient indexing structure in order to accelerate decision support [Joh99]. The basics behind bitmap indexing is to use a sequence of bits to indicate whether an attribute in a record is equal to a specific value. Using the *equality encoding* scheme for bitmap indices, each distinct attribute value is encoded as one bitmap having as many bits as the number of records in the relation [CI98, AKS02]. In this connection it is notable that the bitmaps for a specific attribute contain mutual exclusive bits, indicating that for any position across all the bitmaps in the index only a single 1 bit is allowed. As a consequence, bitmaps for high cardinality attributes tend to be sparse indicating a low *bit density* calculated as $\delta = \frac{i}{n}$ where i denotes the number of 1 bits and n denotes the total number of bits in the bitmap.

The position of each bit in the bitmaps denotes a separate record from the indexed relation. For a bitmap corresponding to a given attribute value the 1 bits are found only where the associated records contain the attribute value represented by the bitmap. To illustrate this, a simple example of a bitmap index usage involves a relation holding a gender attribute with the associated value domain {*male, female*}. In accordance with the equality encoding scheme, this attribute implies that two bitmaps are required, one for each gender. If, e.g., the first bit in the male bitmap is set to 1, it implies that the attribute value of the first record of the indexed relation is “male”.

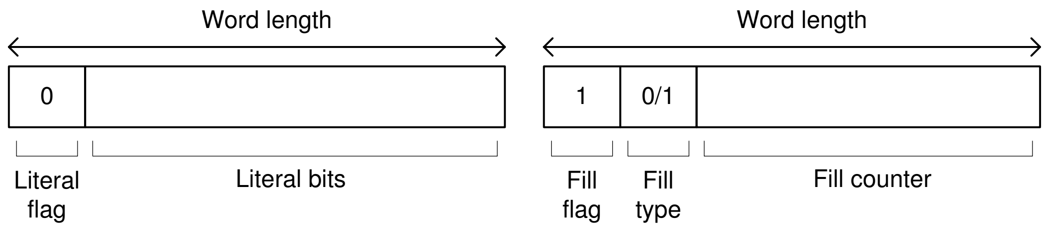
To observe the advantage of the bitmap indices, we must initially relate to the computer architecture, where a *word* usually consists of 32 or 64 bits, depending on the architecture [AKS02]. Using this knowledge, we know that a single bit-wise instruction computes, e.g., 32 bits at once. However, the main advantage of introducing bitmap indexing is notable mainly when performing selections on multiple attributes across several relations, where bit-wise bitmap operations can replace expensive joins performed between the involved relations [OG95, OQ97]. Supposedly that a listener wishes to select all music performed by the artists Madonna and U2 released in the year 2005. For this particular example, a bit-wise OR is performed on the appropriate bitmaps of the artist relation in order to generate the combined bitmap representing the songs performed by both Madonna and U2. In addition, performing a bit-wise AND on the combined bitmap and the bitmap representing all songs released in the year 2005 associated with the release relation, the wished selection is achieved. Additionally, using bitmaps to represent the history of played songs and the collection of skipped songs, bit-wise operations may be used to ensure that neither songs recently played nor songs contained is the collection of skipped songs are returned to the listener.

Moreover, appending an additional record to a relation, only the bitmap for the associated attribute value is to be updated with the appropriate 1 bit, indicating that consecutive 0 bits are omitted from the end of the remaining bitmaps. By omitted these bits the exhaustive task of appending additional 0 bits to all bitmaps, each time a new record is added to the relation, is avoided. Still, performing bitwise operations on bitmaps of unequal length is supported as the shortest bitmap is virtually padded with 0 bits.

Bitmap Compression

As bitmaps consists solely of 0 and 1’s it may be possible to compress the bitmaps significantly depending on the number of consecutive 0 or 1 bits. Thus, having bitmaps with highly clustered 0 and 1’s, a better compression is achievable than having a uniform scattering of the 1 bits. In a worst case scenario the compressed version of a bitmap occupies even more space than the uncompressed version. A consequence of bitmap compression is moreover an increased overhead when performing bit-wise operations, for which reason a tradeoff between the space consumption and the performance should be considered.

With regard to vast music collections, query performance can not be neglected even when dealing with compressed bitmaps. Hence, using the *WAH (Word-Aligned Hybrid)* [WOSN, WOS06] compression approach the bitmaps are compressed using words as a unit of grouping. A WAH compressed bitmap consists of a sequence of WAH words, which each can be of the *literal word* or *fill word* type. In Figure 8, we present the structure of a literal word and a fill word. The MSB (Most Significant Bit) in the WAH



(a) *WAH literal word.* (b) *WAH fill word.*

Figure 8: *Compression scheme for WAH literal words and fill words.*

words is an identifying flag indicating whether the current word is a literal word or a fill word. Having a literal word, as illustrated in Figure 8(a), the remaining literal bits are used to store the bits from the uncompressed bitmap. Hence, the number of literal bits corresponds to an alignment group. The fill word, illustrated in Figure 8(b), uses the second MSB to indicate whether 0 bits or 1 bits are to be counted. The remaining bits are used to store an integer specifying the number of consecutive alignment groups that are of the specified fill type. Hence, word alignment is ensured as the number of consecutive bits represented by the fill word is a multiple of the number of literal bits in a literal word. In Example 9 a WAH compression of a sample bitmap is performed.

Example 9 *Assuming a word length of 32 bits, the bitmap containing the 128 bit sequence 1, 20×0, 3×1, 79×0, 25×1, is split into groups of 31 consecutive bits, which is the size of the alignment groups. The groups can be seen in Figure 9 represented both as a number of bits and in hexadecimal notation. Moreover, the WAH compressed bitmap is presented in the figure, where it consist of a literal word followed by a fill word and two literal words.*

31-bit groups	1, 20×0, 3×1, 7×0	31×0	31×0	10×0, 21×1	4×1
Groups in hex	40000380	00000000	00000000	001FFFFF	0000000F
	literal word	fill word		literal word	literal word
WAH (hex)	40000380	80000002		001FFFFF	0000000F

Figure 9: *A bitmap split into alignment groups and the corresponding WAH compressed bitmap [WOS06].*

■

As a consequence of aligning bitmaps in words, bit-wise instructions such as bit-wise AND, OR and XOR may be performed directly on literal words within the compressed bitmaps, avoiding additional expensive decompression and compression techniques. Example 10 illustrate such an alignment while performing a bit-wise AND operation.

Example 10 *Figure 10 illustrates how the alignment of groups enables a bit-wise AND to be performed on the two WAH compressed bitmaps A and B, both consisting of 128-bits. In the figure all words are represented in hexadecimal notation.*

	Grp. 1	Grp. 2	Grp. 3	Grp. 4	Grp. 5
	literal word	fill word		literal word	literal word
A	40000380	80000002		001FFFFF	0000000F
	fill word		literal word	literal word	literal word
B	C0000002		7C0001E0	3FE00000	00000003
	literal word	fill word			literal word
C	40000380	80000003			00000003

Figure 10: *Example showing how a bit-wise AND is performed on the WAH compressed bitmap A and B. The WAH compressed bitmap C is the result [WOS06].*

As can be seen from Figure 10, the alignment ensures direct use of bit-wise instructions for handling two aligned literal words (Grp. 4 and 5). For handling aligned fill words the type of the resulting fill word is found, while concerning merging of fill words having fill counters that overlap when they are aligned (Grp. 2). Similarly, handling an alignment of a fill word and a literal word the result of the bit-wise operation is either a literal word (Grp. 1) or fill word (Grp. 3).

■

5.2 Handling of Collection Query Parameters

In connection with the music player using the MOD framework, a number of collection parameters are to be managed. These parameters are subsets of the songs contained in the music collection, and represent the set of skipped songs, the history and as well the current metadata restricted collection. The parameters are passed along to the relevant queries. Common for all parameters is that they apply to only a single music player. In the following listing we elaborate on the characteristics of the respective collections.

- *Current collection*: The current collection is represented by a bitmap holding 1 bits only for the songs that fulfil the metadata restrictions of the listener. Thus, the content of the current collection may change on demand in order to accommodate the impulsive behavior of the listener interacting with the music player.
- *Set of skipped songs*: Holds all songs that have explicitly been skipped by the listener. As for all other subsets of songs from the music collection, the set of skipped songs is represented by a bitmap containing 1 bits for all skipped songs. Hence, when performing the actual skipping of a song, the bit representing the song is set to 1.
- *History*: The history is constituting the subset of recently played songs and is represented by a bitmap, expressing which songs are in the history. The size of the

history may be specified by the user. As bitmaps are unable to represent an ordering among songs, an explicit ordering is to be applied within the music player, for the purpose of specifying the order of the songs as they enter the history. To avoid duplicate songs only the most recent occurrence of two identical songs remain in the history.

5.3 Distance Management

In Section 4.1 a formal definition of the distance store is presented. To elaborate on the technical aspects of the distance management, this section describes how a number of distance stores constitute the handling of the distances between the songs managed by the MOD framework.

To ensure distance management, knowledge about all the distances between any two songs managed by the framework is required. The distances from a single base song to all the remaining songs are represented by a distance store. Hence, in order to include distances between all songs, n distance stores are needed, where n denotes the number of songs.

As defined in Section 4.1 a distance store consists of a number of partitions, each corresponding to an associated distance interval. Hence, each partition is represented by a single bitmap indicating the songs belonging to the associated distance interval. The collection of bitmaps required for a single distance store, constitutes a bitmap index for the distances of the songs with respect to the base song of the distance store. In Figure 11 an abstraction of a bitmap index of a distance store is illustrated for a collection of 10 songs having song s as the base song for the distance store.

All distance stores are subject to persistent storage within an RDBMS, handled by the MOD framework. The distance management relation contains partition records as presented in Figure 12. The first part of the record is the position of the base song of the distance store, which is presented as a 24-bit integer. Using an 8-bit integer for the storage, the second part represents a numbered index indicating the position of the partition within the distance store. The index ranges from 0 to $m - 1$, where m is the number

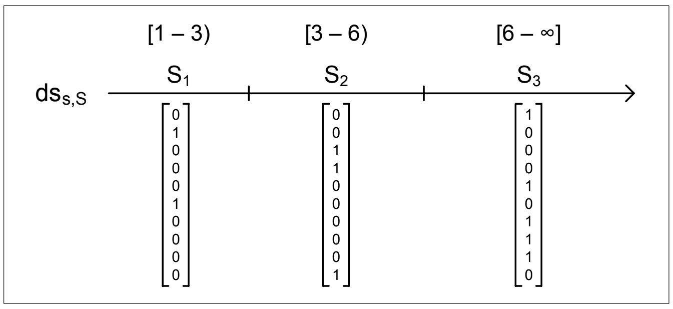


Figure 11: *A distance store representing songs within the partitions as a bitmap index.*

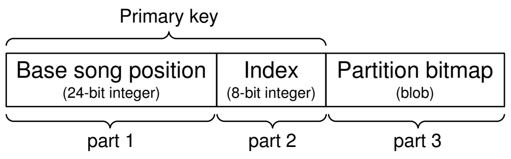


Figure 12: *Partition record of the distance management relation used for persistent storage of the distances between the songs managed by the MOD framework.*

of partitions in the distance store. The first and second part combined is specified as a composite primary key to ensure efficient lookup of a given partition bitmap from the relation. The third part contains the partition bitmap, which identifies all songs contained in the partition defined by the first and second part. The partition bitmap is stored as a blob. The primary index is specified as a clustered index to ensure that the blob entries are maintained within the index, in order to enable efficient access to all partitions associated with a given distance store.

Theoretical Worst-Case Space Analysis

In the following we conduct a worst-case space analysis with respect to the space consumption of the distance management.

The space occupied by the first two parts of the partition record increases linearly as the number of songs increases. The space occupied by the partition bitmap, i.e., the third part in the partition record presented, constitutes the most crucial part of the total space required for the distance management. In this worst-case analysis it is assumed that the 1 bits within a partition bitmap are located at certain places to archive the worst possible compression. In addition, all 1 bits are distributed equally among the given number of partitions. To illustrate, having just a single 1 bit represented in each word to compress, it implies that no space reduction is achievable when applying WAH compression, as all compressed words are literal words.

The space occupied in bytes by the distance management, where no compression is achievable in worst-case, is given by Equation 6, where i states the number of partitions and n states the number of songs. The word length is denoted by the abbreviation wl.

$$S(n,i) = \overbrace{\frac{32}{8} \cdot i \cdot n}^{\text{part 1+2}} + \overbrace{\left\lceil \frac{n}{wl-1} \right\rceil \cdot \frac{wl}{8} \cdot i \cdot n}^{\text{part 3}} = \left(4 + \left\lceil \frac{n}{wl-1} \right\rceil \cdot \frac{wl}{8}\right) \cdot i \cdot n \tag{6}$$

With respect to Equation 6, the worst-case space consumption applies only when density $\delta = \frac{n}{i} \geq \frac{n}{(wl-1)\cdot 2}$ which can be simplified as $i \leq (wl-1)\cdot 2$, indicating the threshold where no space reduction is achieved in worst-case, when applying WAH compression.

Having more than $(wl-1)\cdot 2$ partitions, a space reduction is possible in the worst-case when applying the WAH compression scheme as presented in Equation 7. In a worst-case scenario, where all 1 bits are uniformly scattered among and within the partition bitmaps of each of the distance stores, the space occupied requires a zero fill word and a literal word for each 1 bit in a partition bitmap. To count all the n distance stores for the n songs n^2 bits are needed.

$$S(n,i)_{compressed} = \overbrace{\frac{32}{8}\cdot i\cdot n}^{\text{part 1+2}} + \overbrace{2\cdot n^2\cdot\frac{wl}{8}}^{\text{part 3}} = \left(4\cdot i + n\cdot\frac{wl}{4}\right)\cdot n \tag{7}$$

As it can be seen from Equation 7, the space consumption of the third part is no longer bounded by the number of partitions i, but only by the number of songs n. The total worst-case space consumption is given by Equation 8.

$$S(n,i)_{total} = \begin{cases} S(n,i) & i \le (wl-1)\cdot 2 \\ S(n,i)_{compressed} & otherwise \end{cases} \tag{8}$$

Assuming a word length of 32 bits the calculations of the worst-case space consumption for the distance management is presented in Example 11.

Example 11 *Having a music collection with 1,000 songs, the worst-case space consumption induced by the distance management can be calculated. Assuming two cases where the number of maintained partitions is set to 12 and 100 respectively, the space consumption is calculated as follows:*

$$S(1000,12)_{total} = \left(4 + \left\lceil\frac{1000}{32-1}\right\rceil\cdot\frac{32}{8}\right)\cdot 12\cdot 1000 = 1.56MB$$

$$S(1000,100)_{total} = \left(4\cdot 100 + 1000\cdot\frac{32}{4}\right)\cdot 1000 = 8.01MB$$

Having 100 partitions we see that WAH compression is possible compared with 12 partitions. However, more space is required as more bitmaps are needed for the increased number of partitions. Performing two similar calculations for 100,000 songs, while maintaining the number of partitions at 12 and 100, respectively, we achieve the following:

$$S(100000,12)_{total} = \left(4 + \left\lceil\frac{100000}{32-1}\right\rceil\cdot\frac{32}{8}\right)\cdot 12\cdot 100000 = 14.43GB$$

$$S(100000,100)_{total} = \left(4\cdot 100 + 100000\cdot\frac{32}{4}\right)\cdot 100000 = 74.54GB$$

■

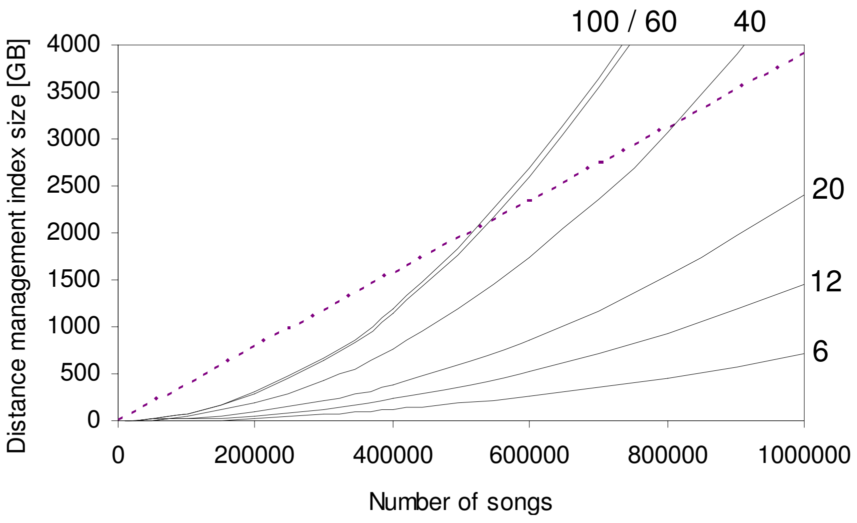


Figure 13: *The worst-case space consumption by the distance management given the number of partitions; 6, 12, 20, 40, 60 and 100. The dashed line represents the space needed to store the music files.*

The worst-case space consumption for the distance management presented in Example 11 indicates a squared increase of the space consumption, as the number of songs increase. This can however not be avoided as the number of distances between the songs is a square of the number of songs. In addition, the example shows a significant difference in the space consumption, as to whether compression of the partition bitmaps is possible or not. Having few partitions, i.e., $i \leq (wl - 1) \cdot 2$, implies that the number of partitions directly influences the total worst-case space consumption. On the other side, having many partitions, the total worst-case space consumption would be nearly independent of the number of partitions within the distance stores. This can be seen in Figure 13, where the change from 60 to 100 partitions is small. Drawing the line for 200 partitions it will be very close to the line representing 100 partitions. In addition, Figure 13 presents the space consumption for a given number of songs when using 6, 12, 20, 40, 60 and 100 partitions, respectively. Moreover, the space consumption of the actual music files is represented by the dashed line, assuming an average song size at 4.0MB.

Independent of the number of partitions, Figure 13 shows that having 200,000 songs indexed occupies less than 50% of the space required for the music files. However, having only 12 distance intervals, less than a 10% space overhead is introduced. Moreover, indexing 1,000,000 songs having 12 partitions, the overhead introduced by the distance management is below 50% for a theoretical worst-case.

As can be seen, the number of partitions play an important factor for the space consumption with respect to the distance management. In general it can be said, that a large number of partitions ensure a better reflection of the provided similarity measure. However, the more partitions introduced, the more time should be expected when executing the queries. This tradeoff is concerned in the Section 7.

Attribute Value Decomposition

When considering the usage of the distance stores it is likely that the number of partitions specified is below the given threshold for possible worst-case compression. Hence, considering the worst-case scenario, WAH compression does not achieve any space reduction.

In case the usage of bitmap indices obey the two following constraints, space reduction techniques different than compression techniques can be applied. The technique considered is *AVD (Attribute Value Decomposition)* as presented in [CI98].

- The attribute cardinality of the bitmap index should at all times be below a known constant.
- All items indexed by the bitmap index should presume exactly one value.

Considering the above constraints, the distance management obey the first constraint as the cardinality of the values to index equals the constant number of partitions chosen for a specific usage of the MOD framework. The second constraint is obeyed as all songs contained by the index belong to exactly one of the partitions, i.e., the distance from any song to the base song of the distance store is always able to be mapped to a single interval in the complete partitioning. Thus, obeying the two constraints, AVD can be applied to the distance management. Explaining the properties of AVD in brief, this technique encodes the numbers indicating the associated bitmaps in the bitmap index, i.e., the attribute values. The encoding is performed by the use of a base specification $< base_n, \ldots, base_1 >$, where n is the number of bases. A base should be understood as a base of a number system. An example of a base specification is base $< 3, 3 >$. This specifies a two digit number, where each digit has the base three. Such a specification allows to express nine values in the range form 0 to 8. Example 12 presents an example of a distance store using base $< 3, 3 >$ AVD encoding.

When applying the AVD bases, it should be ensured that the sum of the bases is as small as possible while the product of all bases provide at least the required number of partitions. Hence, to represent 12 partitions the base $< 3, 4 >$ constitutes a better approach than base $< 2, 6 >$ as the latter approach uses eight bitmaps rather than seven.

Example 12 *Assume that 12 songs are indexed in accordance with the distance store represented in Figure 14 having nine partitions. Performing AVD, the position of the 1 bits can be represented by a single index vector containing a projection of the indexed attribute values. These numbers can be encoded using a given base specification. In the case of the base specification $< 3, 3 >$ the encoding would be as presented in Figure 15 having two bitmap indices of each group of three bitmaps, i.e., the given bases. Each number in the index vector is now represented by the sum of two numbers given by each digit, where the second digit is a multiple of the product of the previous bases like in regular number representation. As an example the index, 3 is decomposed to $1 \cdot 3 + 0$, which corresponds to a 1 bit in b_2^1 and b_1^0, respectively.*

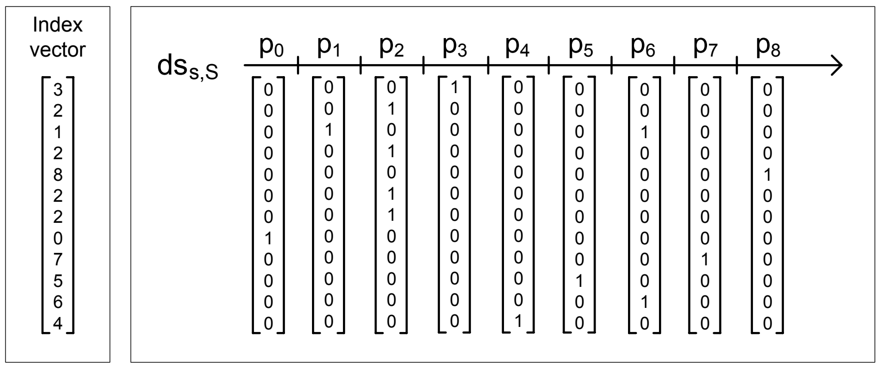


Figure 14: *Distance store and the associated index vector.*

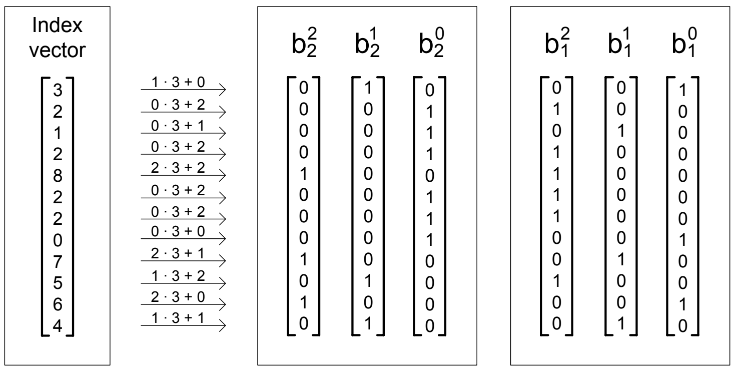


Figure 15: *Encoding process from the index vector to a base $< 3, 3 >$ encoded index.*

The bitmap for the most similar songs identified by the distance store can be accessed knowing the associated index of the partition in the distance store, in this case index zero. The zero index bitmap, p_0, is found by performing decomposition into the base specification $0 \cdot 3 + 0$, i.e., the corresponding bitmap is accessed by performing a bit-wise AND operation on the bitmaps b_2^0 and b_1^0.

■

As can be seen from Example 12, a distance store containing nine partitions can be represented by the use of six bitmaps when applying AVD. In this case an additional bit-wise AND operation is required for every access to a given partition. Moreover, applying AVD the space consumption is reduced by 33.3% at the cost of a single bit-wise operation for each access to a partition bitmap.

WAH compression could be applied on the AVD encoded bitmaps. However, consulting the worst-case space analysis a compression overhead is introduced, as this case can be assumed to be equivalent to the case of having fewer partitions in the non-AVD bitmaps. Thereby, the worst-case space consumption is as presented in Figure 13, except that a distance store with, e.g., nine partitions corresponds to a distance store with six partitions. To conclude from this, the way to ensure more efficient use of space for the partition bitmaps is to apply AVD.

The record needed to be stored in the distance management relation would presume the same structure as the partition record presented in Figure 12 on page 24. The index part is now just an index representing the location in the set of base encoded bitmaps. Hence, the space consumption can be calculated as described by Equation 9, where n states the number of songs.

$$\begin{aligned} S(n, < base_m, \ldots, base_1 >)_{total} &= \overbrace{\sum_{i=1}^{m} base_i \cdot n \cdot \frac{32}{8}}^{\text{part 1+2}} + \overbrace{n \cdot \sum_{i=1}^{m} base_i \cdot n \cdot {}^1\!/_8}^{\text{part 3}} \\ &= (32 + n) \cdot \sum_{i=1}^{m} base_i \cdot n \cdot {}^1\!/_8 \end{aligned} \tag{9}$$

Assuming a music collection with 100,000 songs, Example 13 presents the space consumption when applying AVD on the associated distance stores while using 12 partitions.

Example 13 *Having distance stores with 12 partitions, the AVD representation may be applied using base* $< 3, 4 >$*, in order to calculate the space consumption for 100,000 songs.*

$$S(100000, < 3, 4 >)_{total} = \big(32 + 100000\big) \cdot 100000 \cdot (4 + 3) \cdot {}^1\!/_8 = 8.15GB$$

■

Referring to the equivalent calculation of Example 11 on page 25 with respect to 100,000 songs represented by 12 partitions, we here see a space reduction of 43.5%.

In addition, applying WAH compression might in the best case reduce the space requirement even further depending on the clustering within the bitmaps. As mentioned, worst-case implies an additional overhead. Still this overhead may be desired as other elements within the MOD framework benefits from the WAH compression.

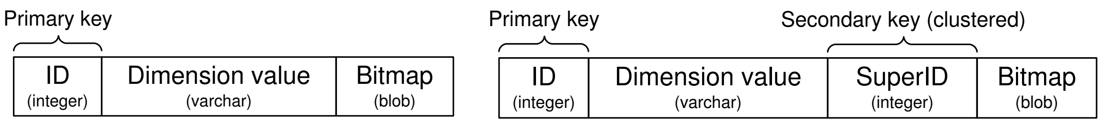
5.4 Metadata Management

As stated in Section 4.1, a set of descriptive metadata attributes are associated with any given song. In order to provide efficient access of these metadata attributes within a vast music collection, we apply a multidimensional cube to store metadata attributes and their associated bitmaps. In this section, we describe the design of such a multidimensional cube and the handling of both regular and irregular metadata hierarchies. In addition, we introduce the handling of user collections which are a part of the MOD framework.

Cube Representation

To represent the multidimensional cube in a relational database, we adopt the *snowflake schema* known from the theory of multidimensional databases [Tho97]. The snowflake schema is composed of a central fact table and a set of associated dimensions. The snowflake schema satisfies the structure of the metadata hierarchies by allowing a metadata dimension to be represented as a number of dimension tables. Each dimension level in the metadata hierarchy corresponds to a dimension table. The snowflake schema normalizes dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table as apply to the *star schema* approach [KR02]. While this saves space, it is known to increases the number of dimension tables thus resulting in more complex queries and reduced query performance [KR02]. However, as the purpose of the multidimensional cube in the MOD framework is to find the bitmaps, no expensive join queries are to be performed, as selections based on multiple attributes are performed by applying bit-wise operations on the corresponding bitmaps.

As stated, a metadata dimension in a relational database is represented as a number of dimension tables, where each dimension table corresponds to a level in a metadata hierarchy. According to the snowflake schema representing the metadata within the MOD framework, there exists two types of relations used as dimension tables. Records of both types of relations can be seen in Figure 16. The *level record* in Figure 16(a) is used for the highest level within each dimension. For efficient access, the relation is defined as clustered having the id attribute as the primary key. The *sub level record* in Figure 16(b) is clustered in accordance with the super id attribute, that is associated with a given superordinate level. This ensures an efficient foundation for hierarchical metadata navigation, as, e.g., the subgenres of a given genre are stored consecutively within the relation. However, as metadata may be accessed using id's, we maintain an index on the id attribute of the relation. The bitmap contained within each of the records, represents the songs which are associated with the dimension value of the records. Example 14 illustrates how a two level hierarchy is mapped to dimension tables.

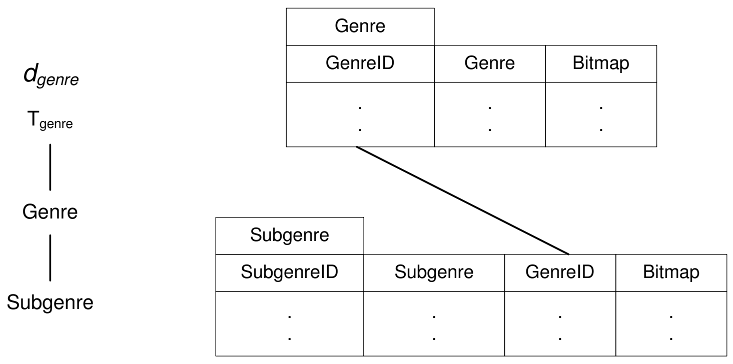


(a) *Level record for the highest level in a dimension.*

(b) *Sub level record for all the sub levels in a dimension.*

Figure 16: *The two record types used within the metadata dimension tables.*

Example 14 *Consider the metadata dimension schema for the genre dimension, presented in Figure 17(a). For such a schema, two dimension tables are required, namely a genre dimension table storing level records and a subgenre dimension table storing sub level records. The two dimension tables are presented in Figure 17(b). The genre id in the subgenre dimension table is a foreign key of the genre id within the genre dimension table.*



(a) *Schema.* (b) *Two dimension tables associated with the d_{genre} metadata dimension.*

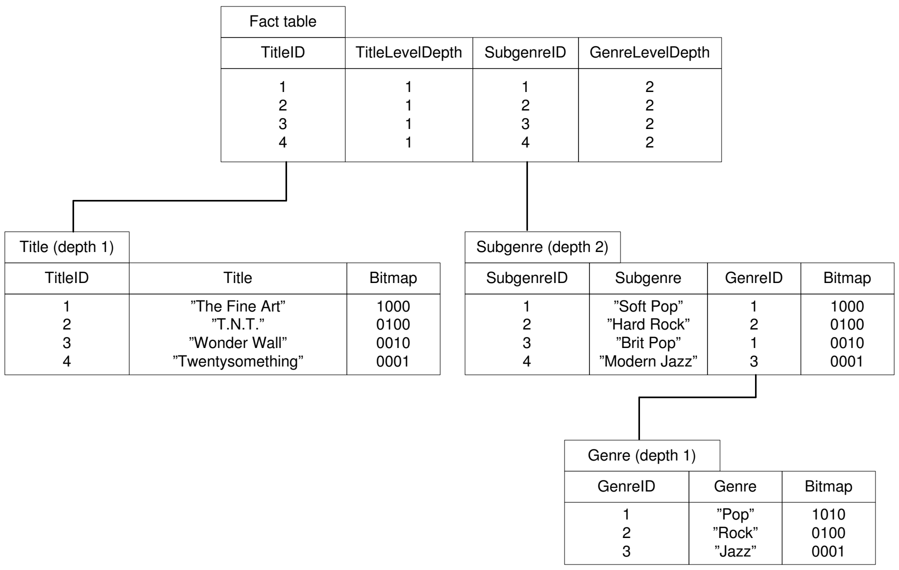
Figure 17: *The dimension tables defined from the schema of metadata dimension d_{genre}.*

■

Aside from the dimension tables, a snowflake schema consists of a fact table. The fact table contains a *fact record* for each of the songs managed by the MOD framework. For each dimension in the cube the fact record contains an id and a level depth for the most specific dimension value. The level depth corresponds to a given dimension table within the current dimension, where the highest level in a dimension has level depth one. The id is a foreign key to the id within the dimension table identified by the associated dimension and level depth. The mapping from a level depth and a dimension to a dimension table name is to be found in a hierarchy relation, which is stored persistent within the database. The reason for storing the level depth of the most specific dimension value is that the most specific dimension value may not necessarily be at the bottom level of the hierarchy in the case of a irregular hierarchy. In addition, the fact table states the order of the songs managed by the MOD framework.

In Example 15 we consider the structure of the snowflake schema representing the fact table and dimension tables discussed above.

Example 15 *In Figure 18 the snowflake schema for the metadata dimensions d_{title} and d_{genre} and the associated fact table are presented. From the fact table it appears that four songs are currently stored in the music collection, having the song titles; "The Fine Art", "T.N.T", "Wonder Wall" and "Twentysomething", respectively. Moreover,*



Fact table

TitleID	TitleLevelDepth	SubgenreID	GenreLevelDepth
1	1	1	2
2	1	2	2
3	1	3	2
4	1	4	2

Title (depth 1)

TitleID	Title	Bitmap
1	”The Fine Art”	1000
2	”T.N.T.”	0100
3	”Wonder Wall”	0010
4	”Twentysomething”	0001

Subgenre (depth 2)

SubgenreID	Subgenre	GenreID	Bitmap
1	”Soft Pop”	1	1000
2	”Hard Rock”	2	0100
3	”Brit Pop”	1	0010
4	”Modern Jazz”	3	0001

Genre (depth 1)

GenreID	Genre	Bitmap
1	”Pop”	1010
2	”Rock”	0100
3	”Jazz”	0001

Figure 18: *Illustrates a snowflake schema having the d_{title} and d_{genre} metadata dimensions.*

the songs belong to the subgenres; “Soft Pop”, “Hard Rock”, “Brit Pop” and “Modern Jazz”, respectively. Hence, this small music collection is represented by a bitmap with four bits. The first bit in the each bitmap corresponds to the first song in the managed music collection, the second bit to the second song, etc. Along with the foreign keys in the fact table, the level depths appears. From these it can be seen, that the most specific dimension value of all the songs corresponds to the bottom level of the hierarchies.

In addition, it can be seen from Figure 18 that aggregation of the bitmaps from a sub level to a superordinate level is applied within the dimension hierarchy by use of bit-wise operations on the associated bitmaps from the sub level.

■

Handling of Irregular Hierarchies

In Section 2 it is argued that metadata for digitized music, to some degree, is expected to become standardized. However, as not everyone can be expected to conform to such a standard, irregularities in the metadata hierarchies should be anticipated.

To support irregular hierarchies, i.e., *non-onto*, *non-covering* and *non-strict* hierarchies, different design techniques have been applied. To describe these, we initially look at how irregular hierarchies could occur with respect to the metadata of music and thereafter at the techniques applied to handle the irregularities.

Adding songs to the music collection, where the metadata of the songs is incomplete, causes a non-onto hierarchy for an instance of the metadata dimension to occur. To illustrate, consider the example shown in Figure 19 having two songs, where a song is added under the “Rock” genre solely, as the subgenre is omitted.

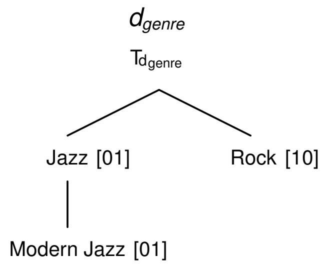


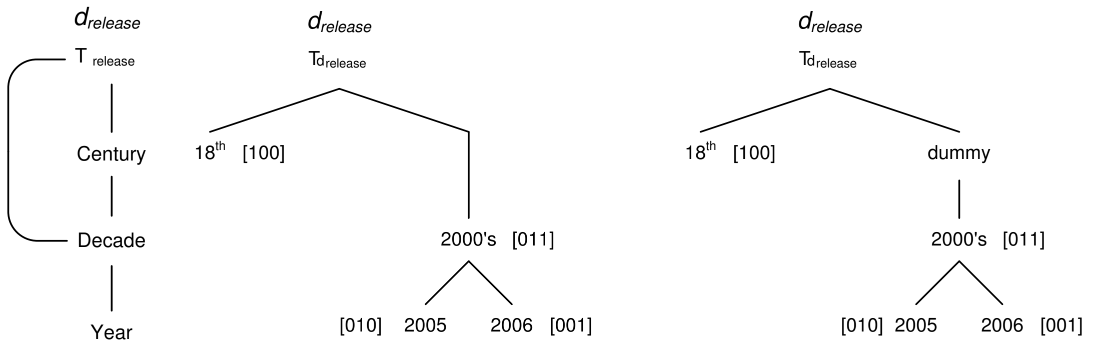
Figure 19: *A non-onto genre instance.*

Due to the fact that the bitmaps are stored in all the levels in the hierarchy, and that the cube is used to find bitmaps, the expected results will be retrieved from the hierarchy even though it is non-onto.

Should the metadata of music skip a level within a given dimension, the hierarchy becomes non-covering. To illustrate this type of hierarchy, consider the metadata schema presented in Figure 20(a), where the century level can be skipped if desired. In Figure 20(b) this is the actual case, where the dimension value “2000’s” is mapped directly to the dimension value “$\top_{d_{release}}$”, and thereby skipping the century level.

To enable support for non-covering hierarchies the dimension value “dummy” is to be inserted into the skipped hierarchy level [PJD99]. To illustrate, consider Figure 20(c) where a new “dummy” dimension value is inserted into the century level such that the dimension value “2000’s” is now mapped to “dummy” instead of the “$\top_{d_{release}}$” dimension value. As a result of inserting the “dummy” dimension value, a covering hierarchy structure is obtained. The hierarchy is now mapped to a regular hierarchy and can thus be represented by dimension tables.

To exclude “dummy” dimension values from being retrieved, the values are implicitly marked such that, whenever these values appear, the values in the sub level will be



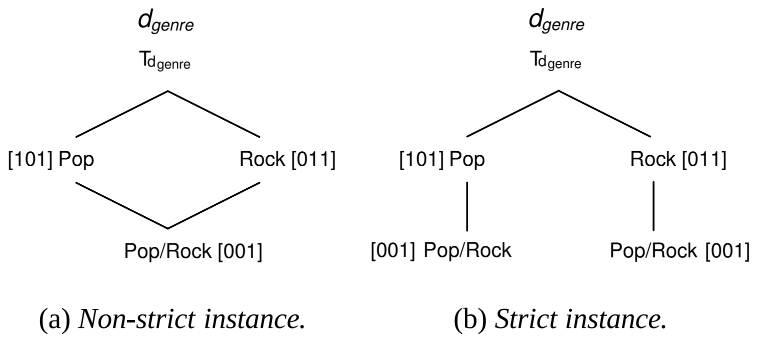
(a) *Schema.* (b) *Non-covering instance.* (c) *Covering instance.*

Figure 20: *Transformation of a non-covering hierarchy into a covering hierarchy.*

retrieved instead. Considering the example shown in Figure 20(b), the dimension values "2000's" and "18^{th}" are retrieved if the query states that all dimension values below the top level are to be retrieved. Hence, as we apply this hierarchical way of retrieval, we do not store bitmaps associated with the "dummy" dimension values.

As a consequence of handling non-covering hierarchies, the "dummy" dimension values inserted, impose an additional space overhead. However, the space overhead is small as bitmaps for the dummies are not present. Adding additional "dummy" dimension values, it is ensured that the non-covering hierarchy is transformed into a covering hierarchy.

A song with conflicting attribute properties could be added to the music collection, causing the hierarchy associated with the attributes to become non-strict. To illustrate this structure, consider Figure 21(a), where a song is associated with the subgenre "Pop/Rock" and thus to the two genres "Rock" and "Pop". To normalize the non-strictness of the hierarchy, duplicate dimension values are introduced as illustrated in Figure 21(b). As a consequence, redundant versions of the dimension value "Pop/Rock" are mapped the to both the "Pop" and "Rock" dimension values, respectively.



(a) *Non-strict instance.* (b) *Strict instance.*

Figure 21: *Transformation of a non-strict hierarchy into a strict hierarchy*

The consequence for transforming a non-strict hierarchy into a strict hierarchy is that duplicate dimension values are introduced, which in turn introduces an additional space overhead as the bitmaps are redundant. However, this is a necessity in order to support a non-strict hierarchy in a relational snowflake schema.

Management of User Collections

In addition to the efficient indexing of the metadata of music, the MOD framework supports user managed collections, i.e., collections that may be constructed and altered on demand by the user. The idea behind the user managed collection is that the user manually should be able to group a number of songs into a given collection, e.g., a collection of favorite songs.

Each of the user collections can be modeled as a separate bitmap. To store the associated bitmaps, a metadata dimension would be the obvious choice. Thereby, the metadata selections may include the user specified dimensions, e.g., the users collections of favorite songs can be intermixed with all recently released songs as a metadata restriction. However, modelling the favorite collections as dimension values within a user specified dimension should differ from the regular metadata defined dimensions.

The regular metadata dimensions, such as the genre, differs in the way that it is associated with the fact table, which is not the case for the user specified dimensions. The reason is that, having a position of a song the genre should be determinable. On the other hand, knowing, e.g., in what favorite collection a song is contained is unnecessary as this is not the goal of having user managed collections. Moreover, the fact that songs could be in either none or several user managed collections complicates the structure of the fact table. Thus, the purpose for using dimension tables to store user managed collections, is to enable a generalized selection support.

6 Query Evaluation

In this section we describe how to perform query evaluation of the query functionalities described in Section 4 while relating to the technical design presented in Section 5. The purpose of the query evaluation is to clarify the algorithms, that form the basis for an implementation. The algorithms are based on the concept of achieving a best worst-case implementation while also assuring a good average-case implementation.

In the following we initially introduce a cost model followed by a description of the evaluation techniques for both retrieval and restriction queries.

6.1 Cost Model

To support the query evaluation design, we introduce a basic cost model in order to count the number of operations performed within the following three categories of operations appointed to have the highest influence on the performance of the MOD framework.

The first category includes the bitmap functions $\mathrm{BitCount}$, $\mathrm{GetBitmapForSetBit}$ and Random. The function $\mathrm{BitCount}$ returns the number of 1 bits in a given bitmap. The task of $\mathrm{GetBitmapForSetBit}$ is to generate a bitmap where only the i^{th} is a 1 bit, whereas Random returns the position for a randomly chosen 1 bit among all 1 bits in the given bitmap. In a worst-case scenario each function require that all bits are counted once, for which reason they are treated equally in terms of evaluation costs.

The second category of operations is the bit-wise operations which are required in order to combine the bitmaps used for indexing. In general, bit-wise operations are considered to have low computational costs, as discussed in Section 5.1. However, as a consequence of bitmap compression, an increased performance overhead is introduced for which reason the bit-wise operations are considered in the cost model. Within this context, all used bit-wise operations are assumed to have equal computational costs for which reason they also are treated equally in terms of evaluation costs. However, to look into the ratio between the occurrences of the different bit-wise operators, they are presented as individual entries in the cost model. As a consequence of bitmap compression, the computational costs required to perform bit-wise operations exceed the evaluation costs related to the bitmap functions of the first category.

The last category of operations is related to the task of performing a database lookup, as these operations require I/O interaction. The operations are divided into two groupings; a metadata cube lookup and a distance management lookup. For each grouping it is assumed, that the index associated with each relation in the grouping is cached, indicating that one lookup is equivalent to one disk I/O. Moreover, in relation to a distance management lookup a range query is considered as a single disk I/O, as records of the distance management relation are clustered in accordance with individual base songs as described in Section 5.3. Also, the number of records to scan in a range query are few, as only a single distance store is concerned at a time.

6.2 Retrieval Queries

In the following we describe the retrieval functions $\mathrm{RandomSong}$ and $\mathrm{SimilarSong}$ used for the retrieval of a randomly chosen song and a song having similar musical content to a given seed song, respectively. In this connection we initially introduce the two helper functions $\mathrm{GenerateCompositeSkipDS}$ and $\mathrm{FetchRandomSongs}$. The task of $\mathrm{GenerateCompositeSkipDS}$ is to cache the composite distance stores representing the distance stores of all skipped songs for each of the individual music players interacting with the MOD framework. The composite distance store representing the distance stores of all skipped songs is denoted as the *composite skip distance store*. Using a unique user id representing a specific music player, the cached composite skip distance store is accessible for retrieval and manipulation. The purpose of $\mathrm{FetchRandomSongs}$ is to enable the possibility to retrieve a specified number of randomly chosen songs from a given music collection represented by a bitmap. Each of the four functions are presented as individual algorithms in the following.

To ease the description of the functionality of the algorithms used for query evaluation, we initially clarify the properties of various types of music collections. The music collection initially passed to the respective functions is denoted as the *search collection* and constitutes either the entire music collection or a subset of the entire collection. The search collection is a subset of the entire collection if a metadata restriction has occurred. Once the search collection has been restricted by the skipped songs and the songs contained in the history of played songs, the collection of the remaining songs is denoted as the *valid collection*. Performing a further restriction by all songs similar to the skipped songs we end up with a collection of songs denoted as the *candidate collection*.

All restrictions, i.e., $p \setminus q$, are performed using the syntax p AND $(p$ XOR $q)$ where p is the collection to restrict and q is the collection to restrict by. The alternative syntax, p AND NOT q, is unusable as the size of the entire music collection can not be derived from the individual bitmaps where consecutive 0 bits are omitted from the end of the bitmaps as described in Section 5.1. Additionally, for all pseudo code descriptions it applies that variable names with the prefix notation *b_* represent bitmaps.

Generate Composite Skip Distance Store

In Algorithm 1 we elaborate on the evaluation of $\mathrm{GenerateCompositeSkipDS}$. The function is applied in Algorithm 3 and 4 in order to generate and cache composite skip distance stores. The composite skip distance stores are cached in main memory. In case a composite skip distance store is cached and no changes has occurred on the set of skipped songs represented by the cached composite skip distance store, the cached version is returned rather than performing an attempt to generate a new composite skip distance store. Similarly, if songs are only added to the set of skipped songs, the cached composite skip distance store may simply be updated with information form the distance stores associated with the added songs.

As input parameters, the function takes a bitmap representing all skipped songs and a user id indicating the music player currently interacting with the MOD framework.

```
GenerateCompositeSkipDS($b\_skip, userId$)
1   $compSkipDS \leftarrow$ empty distance store
2   $b\_storedSkip \leftarrow$ FetchCachedSkipBitmap($userId$)
3   $cachedCompSkipDS \leftarrow$ FetchCachedSkipDS($userId$)
4   $b\_modifiedSongs \leftarrow b\_skip$ XOR $b\_storedSkip$
5   if BitCount($b\_modifiedSongs$) $= 0$
6      then ▷ No changes was made to the skipped songs since last
7           $compSkipDS \leftarrow cachedCompSkipDS$
8      else ▷ Check whether songs are only appended to the cached distance store
9           if BitCount($b\_modifiedSongs$ AND $b\_storedSkip$) $> 0$
10             then ▷ The cached distance store can not be extended and should be discarded
11                  $b\_modifiedSongs \leftarrow b\_skip$
12                  $cachedCompSkipDS \leftarrow$ empty distance store
13          $distanceStoreColl \leftarrow$ empty collection
14          for each position $pos$ of 1 bits in $b\_modifiedSongs$
15              do ▷ Fetch the distance store for the given song
16                  $distanceStoreColl$.Add(DistanceStoreLookup($pos$))
17          $b\_restriction \leftarrow$ empty bitmap
18          $size \leftarrow |distanceStoreColl|$
19          for each partition $b\_p$ in each $distanceStore$ in $distanceStoreColl$ and
                  partition $b\_q$ in $cachedCompSkipDS$ and partition $b\_r$ in $compSkipDS$
                  starting with the partitions representing the most similar songs.
20              do ▷ Compute a partition of the composite distance store
21                  $b\_compPartition \leftarrow b\_q$ OR $b\_p^1$ OR $b\_p^2$ OR $\cdots$ OR $b\_p^{size}$
22                  $b\_r \leftarrow b\_compPartition$ AND ($b\_compPartition$ XOR $b\_restriction$)
23                  $b\_restriction \leftarrow b\_restriction$ OR $b\_compPartition$
24          StoreCachedSkipBitmap($b\_skip, userId$)
25          StoreCachedSkipDS($compSkipDS, userId$)
26  return $compSkipDS$
```

Algorithm 1: *Pseudocode presenting query evaluation for* $\mathrm{GenerateCompositeSkipDS}$

Thus, individual composite skip distance stores are generated and maintained for each music player associated with the framework.

Stepping through the functionality of Algorithm 1, an attempt to fetch a cached composite skip distance store is initially performed using the user id as key (line 2 and 3). To see whether the content of the provided bitmap representing all skipped songs has been modified compared with the content of the bitmap representing all cached skipped songs, a bitmap representing all changes as 1 bits is generated (line 4). Based on the content of the provided bitmaps representing all skipped songs, the remainder of the pseudo code can be split into two distinct cases.

In the first case (line 5 to 7) no modifications have been performed to the provided bitmap representing all currently skipped songs, for which reason the cached composite skip distance store can be returned.

The second case (line 8 to 25) covers two individual scenarios. In the first scenario a new composite skip distance store is to be generated, as the cached instance of the composite skip distance store contains songs that are no longer to be skipped. For the second scenario additional songs are skipped and the cached composite skip distance store is to be extended to include the distance stores associated with the additional skipped songs.

Based on the content of the bitmap representing all modified songs, the algorithm either generates a new composite skip distance store or appends the distance stores associated with the additional songs to a cached instance of the composite skip distance store. Initially, it is clarified whether the current instance of the cached composite skip distance store is invalid and is to be discarded. Thereby, the bitmap representing all modified songs is set to represent all skipped songs (line 8 to 12). Traversing the bitmap representing all modified songs, the distance stores associated with all 1 bits positions are fetched (line 14 to 16). In order to generate the content of the composite skip distance store to return, the content of a number of distance stores are to be consulted. The involved distance stores are; all distance stores representing all modified songs, the cached composite skip distance store and the empty composite skip distance store to return. All distances stores are traversed starting with the partitions representing the most similar songs (line 19 to 23). An example of the generation process is presented in Example 16. Once the composite skip distance store is generated, the previously stored version is replaced (line 24 and 25).

Example 16 *In association with Algorithm 1 line 19 to 23, we assume that two additional songs a and e have been skipped as illustrated in Figure 22. Hence, the content of the distance stores $ds_{a,p'}$ and $ds_{e,p''}$ associated with the skipped songs a and e is to be appended to the content of the cached composite skip distance store ds_q. Initially a union is performed over the content of the partitions p'_1, p''_1 and q_1 and the result is stored in partition r_1 of the distance store ds_r. Then a union over the of content of the partitions p'_2, p''_2 and q_2 is performed and the result is stored in partition r_2 while restricting by the content of all previous partitions of ds_r, i.e., r_1. The same procedure apply to the last partitions, where the content of the partitions p'_3, p''_3 and q_3 is consulted. Once all distance stores have been traversed, the updated composite skip distance store ds_r can*

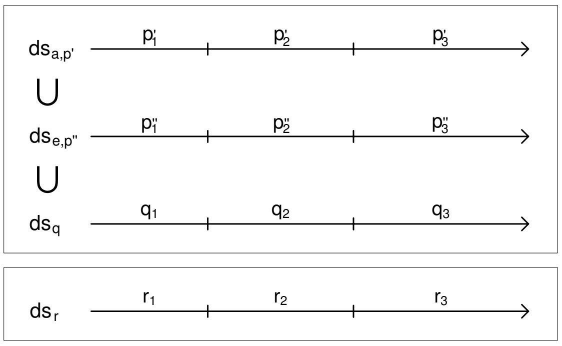


Figure 22: *Extending the composite skip distance store ds_q with the two additional skipped songs a and e.*

be returned. As all songs contained in the music collection are included in the individual distance stores, it applies that entire music collection equals $r = p' = p'' = q$.

■

To measure the costs of query evaluation with respect to $\mathrm{GenerateCompositeSkipDS}$, we collect information concerning the occurrences of bitmap operations and database lookups. The information is gathered in Table 2.

Table 2: *Worst-case costs for* $\mathrm{GenerateCompositeSkipDS}$. *The variables j and i denotes the number of skipped songs and the number of partitions, respectively.*

Cost Description	Evaluation Costs
Bit-wise Operation	
AND	$1 + i$
OR	$i(j + 1)$
XOR	$1 + i$
Bitmap Function	
$\mathrm{BitCount}$	2
Random	0
$\mathrm{GetBitmapForSetBit}$	0
Database Lookup	
Metadata Cube	0
Distance Management	j

Focusing on Table 2 we observe that the function $\mathrm{GenerateCompositeSkipDS}$ is bounded by $i \cdot j$ with respect to the bit-wise OR operation. As i denotes the number of partitions in the distance stores used for similarity groupings it is likely that i remains relatively small as described in Section 4. In a worst-case scenario the value of j resembles the number of all skipped songs, whereas in an average-case scenario j only corresponds to the number of additional skipped songs, which for this case can assumed to be small. Thus, in an average-case $\mathrm{GenerateCompositeSkipDS}$ is bounded by the number of partitions, i.

Fetch Random Songs

The functionality of FetchRandomSongs is presented in Algorithm 2. The task of this function is to generate a bitmap containing a specified number of 1 bits chosen randomly among the 1 bits of the provided search collection. As input parameters the function takes an integer indicating the number of songs to fetch and a bitmap representing the provided search collection.

```
FETCHRANDOMSONGS($songsToFind, b\_coll$)
 1  $b\_randomColl \leftarrow$ empty bitmap
 2  $collSize \leftarrow$ BITCOUNT($b\_coll$)
 3  $orgCollSize \leftarrow collSize$
 4  if $songsToFind > collSize$
 5     then ▷ Select the requested amount of songs randomly
 6          $count = 0$
 7          $missingSongs \leftarrow songsToFind$
 8          while $count < songsToFind$
 9               do ▷ for all the missing songs
10                  for $i \leftarrow 0$ to $missingSongs$
11                      do ▷ Add randomly chosen songs to the random collection
12                         $setBitNumber \leftarrow$ GENERATERANDOMNUMBER($1, collSize$)
13                         $b\_randomColl \leftarrow b\_randomColl$ OR
                                 GETBITMAPFORSETBIT($setBitNumber, b\_coll$)
14                  $count \leftarrow$ BITCOUNT($b\_randomColl$)
15                  if $count < songsToFind$
16                     then ▷ Restrict the search collection for the next iteration
17                         $b\_coll \leftarrow b\_coll$ AND ($b\_coll$ XOR $b\_randomColl$)
18                         $collSize \leftarrow orgCollSize - count$
19                         $missingSongs \leftarrow songsToFind - count$
20     else ▷ The provided collection should be returned
21          $b\_randomColl \leftarrow b\_coll$
22  return $b\_randomColl$
```

Algorithm 2: *Pseudo code presenting query evaluation for* FetchRandomSongs.

To elaborate on the functionality of Algorithm 2, it is initially ensured that the number of songs to include in the bitmap representing the randomly chosen songs can not exceed the number songs represented by the search collection (line 4). If too few songs are present in the search collection, the search collection is returned (line 21). Otherwise, as long as the required number of songs has not yet been found, a number of operations is continuously being invoked (line 8 to 19). First, for as long as songs are still missing, a random number is generated indicating the position of an arbitrarily chosen 1 bit from the search collection. Using the position of this 1 bit, a bitmap representing this particular position is generated and added to the bitmap representing all randomly chosen songs (line 10 to 13). Chances are that the same randomly generated number (line 12) occurs several times, for which reason not all required songs necessarily are found in one iteration. Should it happen that all songs are not found, the search collection is restricted by the already chosen songs, and the number of missing songs is decremented (line 15 to 19). In Table 3 we present the costs of query evaluation with respect to the function $\mathrm{FetchRandomSongs}$.

Table 3: *Worst-case costs for* $\mathrm{FetchRandomSongs}$*. The variable k denotes the number of songs to find randomly from the given collection.*

Cost Description	Evaluation Costs
Bit-wise Operation	
AND	k
OR	$\frac{k(1+k)}{2}$
XOR	k
Bitmap Function	
$\mathrm{BitCount}$	$1+k$
Random	0
$\mathrm{GetBitmapForSetBit}$	$\frac{k(1+k)}{2}$
Database Lookup	
Metadata Cube	0
Distance Management	0

From Table 3 we see that $\mathrm{FetchRandomSongs}$ is bounded by k^2 with respect to the bit-wise OR operation. In a worst-case scenario the value of k corresponds to the number of songs to fetch randomly. However, considering the random generation of positions the average-case is expected to improve, i.e., the bound of the function is closer to k than k^2.

Random Song

Referring to Definition 6 on page 15, the task of $\mathrm{RandomSong}$, is to find a subset of randomly chosen candidate songs from which the song least similar to any of the skipped songs is to be returned. The purpose of the selected candidate songs is to constitute a quality measure for the song to return.

The function $\mathrm{RandomSong}$ described in Algorithm 3, takes as input parameters three bitmaps representing the current search collection, the history and the set of skipped songs. In addition, an integer q is passed in order to specify the number of candidate songs among which to choose the song to return. Finally, the function takes a parameter representing a user id indicating the music player currently interacting with the MOD framework. The id is used to identify a cached composite skip distance store.

Stepping through the functionality of Algorithm 3, we initially generate a collection containing a specified number of randomly chosen songs (line 4). To ensure that songs

```
RandomSong($b\_coll, b\_hist, b\_skip, q, userId$)
 1  $filePath$ ← empty string
 2  $songPosition$ ← Null
 3  $b\_validColl$ ← $b\_coll$ AND ($b\_coll$ XOR ($b\_skip$ OR $b\_hist$))
 4  $b\_randomColl$ ← FetchRandomSongs($q, b\_restrictedColl$)
 5  $compositeSkipDS$ ← GenerateCompositeSkipDS($b\_skip, userId$)
 6  for each partition $b\_p$ in $compositeSkipDS$ starting with the partition representing the
            least similar songs.
 7      do ▷ Check if candidate songs are available
 8          $b\_candidateColl$ ← $b\_randomColl$ AND $b\_p$
 9          if BitCount($b\_candidateColl$) $> 0$
10             then ▷ Choose a position for a random song
11                  $songPosition$ ← Random($b\_candidateColl$)
12                  break
13  if $songPosition <> $ Null
14     then ▷ Fetch the file path for the song found
15          $songRecord$ ← FacttableLookup($songPosition$)
16          $filePath$ ← CubeLookupFetchAttr($songRecord.filenameID$, "$Filename$")
17  return $filePath$
```

Algorithm 3: *Pseudo code presenting query evaluation for* RandomSong.

similar to any of the skipped songs are not returned to the listener, a composite skip distance store is fetched (line 5). Traversing the composite skip distance store starting with the partition representing the least similar songs, each of the associated partitions may be consulted (line 6 to 12), which causes the collection of candidate songs to become generated (line 8). From the collection of candidate songs the position of a song chosen randomly from the partition representing the least similar songs is returned (line 11). In case a song is found, the retrieval of the file path initially involves a lookup in the fact table in order to fetch id's for all metadata dimensions of the song (line 15). Using these details a lookup in the filename dimension of the metadata cube retrieves the file path for the audio file of the found song (line 16).

To measure the costs of query evaluation with respect to RandomSong, we collect information concerning the occurrences of bitmap operations and database lookups. The information is gathered in Table 4.

As shown in Table 4, the operations occurring locally are dependent only of the number of partitions i. As reasoned in connection with GenerateCompositeSkipDS, the number of partitions in a distance store remains relative small, i.e., i remains relative small. Globally the function is bounded by $i \cdot j + k^2$. However, as the k songs are retrieved randomly, k is expected to be small as argued in Section 4. Hence, assuming that only a few songs are skipped at a time, i.e., j is small, we observe that the average-case costs are bounded by i.

Table 4: *Worst-case costs for* $\mathrm{RandomSong}$. *The variables j, k and i denotes the number of skipped songs, the specified quality and the number of partitions, respectively.*

Cost Description	Evaluation Costs: Local	Evaluation Costs: Local + Helper Functions
Bit-wise Operation		
AND	$1+i$	$2+2i+k$
OR	1	$1+i(j+1)+\frac{k(1+k)}{2}$
XOR	1	$2+i+k$
Bitmap Function		
$\mathrm{BitCount}$	i	$3+i+k$
Random	1	1
$\mathrm{GetBitmapForSetBit}$	0	$\frac{k(1+k)}{2}$
Database Lookup		
Metadata Cube	2	2
Distance Management	0	j
Helper Function		
$\mathrm{GenerateCompositeSkipDS}$	1	
$\mathrm{FecthRandomSongs}$	1	

Similar Song

Referring to Definition 7 on page 16, the task of $\mathrm{SimilarSong}$, is to find and return a single song considered most similar to a given seed song. In this context it is ensured, that no songs close to any skipped songs is returned.

As input parameters, the function $\mathrm{SimilarSong}$ described in Algorithm 4 takes three bitmaps representing the search collection, the history and the set of skipped songs. In addition, the position of the seed song is passed to the function, stating the position of the song within a bitmap corresponding to all songs in the music collection. Finally, the function takes a parameter representing an user id indicating the music player currently interacting with the MOD framework. The id is used to identify a cached composite skip distance store.

To a great extent, the functionality of Algorithm 4 reflects that of Algorithm 3, for which reason only the parts of $\mathrm{SimilarSong}$ which differ from those of $\mathrm{RandomSong}$ are described in detail.

After generation of the valid collection and the composite skip distance store, the distance store for the seed song is retrieved using the position of the seed song to perform a lookup in the distance management relation (line 5). To find the collection of candidate songs, the seed song distance store is traversed starting with the partition containing the songs most similar to the seed song. This is done while consulting the content of the corresponding partitions associated with the composite skip distance store (line 6 to 12).

```
SimilarSong($b\_coll, b\_hist, b\_skip, seedsongPosition, userId$)
 1  $filePath \leftarrow$ empty string
 2  $songPosition \leftarrow$ Null
 3  $b\_validColl \leftarrow b\_coll$ AND ($b\_coll$ XOR ($b\_skip$ OR $b\_hist$))
 4  $compositeSkipDS \leftarrow$ GenerateCompositeSkipDS($b\_skip, userId$)
 5  $seedsongDS \leftarrow$ DistanceStoreLookup($seedsongPosition$)
 6  for each partition $b\_p$ in $compositeSkipDS$ and $b\_q$ in $seedsongDS$ starting
            with the partition representing the most similar songs
 7      do ▷ Check if candidate songs are available
 8          $b\_candidateColl \leftarrow validColl$ AND ($b\_q$ AND ($b\_q$ XOR $b\_p$))
 9          if BitCount($b\_candidateColl$) $> 0$
10            then ▷ Choose a position for a random song
11                  $songPosition \leftarrow$ Random($b\_candidateColl$)
12                  break
13  if $songPosition <> $ Null
14     then ▷ Fetch the file path for the song found
15          $songRecord \leftarrow$ FacttableLookup($songPosition$)
16          $filePath \leftarrow$ CubeLookupFetchAttr($songRecord.filenameID$, "Filename")
17  return $filePath$
```

Algorithm 4: *Pseudo code presenting query evaluation for* SimilarSong.

Thus, restricting the partitions of the seed song distance store by the corresponding partitions of the composite skip distance store while considering only the songs contained in the valid collection, the collection of candidate songs is obtained (line 8). In the remainder of the algorithm, the position of a selected candidate song is used to retrieve the file path of the associated audio file.

In Table 5 we present the costs of query evaluation with respect to SimilarSong, where the operations occurring locally are dependent only of the number of partitions i. As reasoned in connection with GenerateCompositeSkipDS, the number of partitions in a distance store remains relative small, i.e., i remains relative small. Globally the function is bounded by $i \cdot j$ with respect to the number bit-wise OR operations occurring in GenerateCompositeSkipDS. Indeed, the costs of the worst-case scenario and the average-case scenario is bounded entirely by the helper function GenerateCompositeSkipDS. Therefor, the function is bounded by i.

When concluding on the costs for query evaluation of the four functions, we may observe that the number of songs contained in a music collection is not represented. However, as the bit-wise operations are applied on the bitmaps, these operations are implicitly influenced by the number of songs in the collection. In turn this entails that, as the size of a music collection increases the only additional cost to be applied is the execution of bit-wise operations.

Table 5: *Worst-case costs for* SimilarSong. *The variables j and i denotes the number of skipped songs and the number of partitions, respectively.*

Cost Description	**Evaluation Costs** Local	Local + Helper Funtions
Bit-wise Operation		
AND	$1 + 2i$	$2 + 3i$
OR	1	$1 + i(j + 1)$
XOR	$1 + i$	$2 + 2i$
Bitmap Function		
BitCount	i	$i + 2$
Random	1	1
GetBitmapForSetBit	0	0
Database Lookup		
Metadata Cube	2	2
Distance Management	1	$1 + j$
Helper Function		
GenerateCompositeSkipDS	1	
FecthRandomSongs	0	

6.3 Restriction Queries

The restriction operators defined in Section 4.1 have been omitted from being described as pseudo code, as they are considered both simple and are not computational demanding. When considering the individual functions related to the restriction operators, the function Select requires access to the database only once in order to locate the bitmap corresponding to the presented metadata attributes. The reason for counting only one disc I/O is that the table from which to retrieve the bitmap is known and the keys on which to perform a lookup are indexed. The index is moreover assumed to be present in the cache of the RDBMS.

With respect to the function SimilarMeta, the database may be accessed as many times as the number of distinct metadata attributes for a given song. In a worst-case scenario, all metadata attributes are passed as input parameters to the function, which in turn causes multiple database access the corresponding dimension tables to occur. Furthermore, to restrict the current search collection an equal number of bit-wise AND operations are to be performed.

Common for both functions is however, that their complexity does not depend on the number of songs to retrieve, but only the number of chosen dimensions. Therefore, changes to the metadata restriction causes only few database lookups and bit-wise operations to be performed.

7 Experiments and Test Results

In this section the MOD framework is evaluated using various configurations. The evaluation concerns the space consumption introduced by the framework as well as the query performance when random and similar songs are retrieved. Initially, we elaborate on the test setup. To conduct the individual tests an implementation of the MOD framework is required. Such an implementation has been constructed using Java 5.0 and MS SQL Server 2000. In Appendix A, the database table definitions for the applied tables are presented.

7.1 Test Setup

To support evaluation for both the retrieval process and the space consumption, a number of databases are constructed. Regarding the bitmaps within the databases, the bitmaps can be configured as being uncompressed or WAH compressed. Additionally, when concerning the distance management, the bitmaps for the distance stores can be represented using either AVD or not, which gives a total of four different bitmap representations. In addition, the number of partitions in the distance stores may vary, which implies even more possible configurations.

To conduct the evaluation, databases with a different number of songs are to be created. We construct these databases with basis in a personal music collection containing 1,000 songs. As we only have this limited number of songs available, we are to generate synthetic data in order to conduct tests on music collections of more than 1,000 songs. A possible solution to extend the existing dataset was to add the same 1,000 songs several times. However, considering the metadata of music, an erroneous distribution of, e.g., of the artist metadata attribute would occur, as it is intuitively wrong simply to scale the artists for a collection of 1,000 songs to, e.g., a collection of 100,000 songs.

To reflect a real life music collection, it is initially ensured that some artists are more productive than others when generating the synthetic data. In this connection it is moreover ensured that a given artist only has a limited career of maximal 30 years, while ensuring that artists with a long career are more productive than artists with a short career. It is assumed that all song titles and album names are unique and that an album contains between 10 and 15 songs. In this context all songs on an album are considered of the same genre. Upon adding synthetic data to the music collection, entire albums are added in continuation of one another, to reflect a frequent real life situation.

Additionally, considering the distances between the individual songs, the use of identical songs would create a very special and unrealistic case for the distances managed. Hence, when creating distance stores for synthetic data we apply random distances between the songs. The random distances are chosen such that the number of songs within each of the partitions of the distance stores gradually increases, starting from the partition representing the most similar songs. As we assume a highly diversified music collection, only few songs are located in the partitions representing the most similar songs.

To reduce insertion time when generating the databases, that index a large number of songs, we insert the relations for the distance stores directly into the database relation rather than through the MOD API. The reason why this way of insertion is faster, is because we omit the symmetric property of the distances. This implies that the distances between songs are not symmetric when requesting similar songs, which does not introduce a problem as we are only concerned with the response time and not the quality of the retrieved songs. The total insertion time is considered to be beyond the scope of this paper, as the required distance calculations constitute the absolute majority of the time used for insertion.

When inserting the real 1,000 songs, we apply the Intelligent Sound Processing toolbox R1 [Int06b] to calculate the distances between all songs in the music collection. The algorithm used is a training algorithm based on a statistical model applied on MFCCs. The algorithm is implemented in MatLab. Using the MOD API, the 1,000 songs are inserted in less than 45 seconds, when excluding the time of the exhaustive distance calculation. The configuration used for insertion is based on 100 partitions where both WAH compression and AVD are applied.

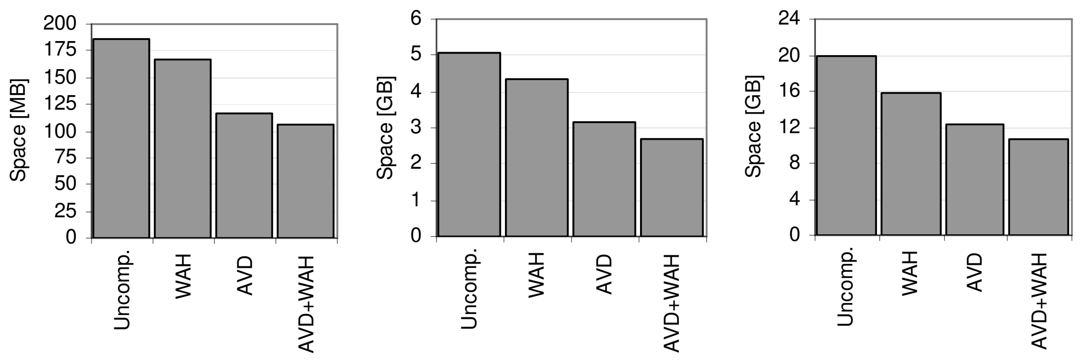
To conduct performance tests with respect to requesting random and similar songs, a query evaluator application is constructed. The query evaluator models a music player application using the MOD framework, for which reason it implements history management, metadata restriction and handling of skipped songs. To ensure real conditions when performing the queries, an initial setup is performed. Thus, to ensure that songs are maintained in the history, 100 random songs are queried and “played” on initialization. In addition, the collection is restricted to 75% of the entire collection in a random manner and the number of skipped songs is as default set to 50 songs. Considering the properties of the skipping behaviour of the MOD framework, 50 songs constitutes a rather large collection of skipped songs as all songs resembling any of the skipped songs are restricted from being retrieved. Then, after initialization, the actual query performance tests are conducted. The query execution time logged by the query evaluator is the average of 50 requests, where the history parameter is altered for each request. Between each run of the query evaluator the cache of the SQL Server is emptied in order to ensure a fair comparison.

The tests are performed on a Pentium M @ 1.7 GHz supplied with 1GB of main memory, running both the query evaluator and the SQL Server. Thereby, we conduct all our tests on a 32-bit architecture. The harddisk used rotates at 7200rpm and has a 16MB cache.

7.2 Space Consumption

As explained above, four different bitmap representations exists within the MOD framework; Uncompressed, WAH, AVD and AVD+WAH. This section thus considers the actual space consumption by each of the four bitmap representations.

Using 12 partitions, the space occupied is presented in Figure 23 for databases indexing 10,000, 50,000 and 100,000 songs, respectively. As can be seen from the figures, the uncompressed bitmaps occupy most space, whereas WAH compression reduces the



(a) *Space consumption for 10,000 songs.* (b) *Space consumption for 50,000 songs.* (c) *Space consumption for 100,000 songs.*

Figure 23: *Space consumption of the framework configured to each of the four different bitmap representations, while having the number of partitions fixed to 12.*

overall space consumption. The distance store contributes with a large number of the bitmaps used for the indexing by the MOD framework. Thus, representing the distance stores using AVD the total space consumption in reduced. As can be seen from the third pillar in the graphs in Figure 23, this is found to be the case when compared with WAH compression. Finally, the fourth representation, AVD combined with WAH compression, introduces an additional reduction of the occupied space compared to applying only AVD. It can be seen that these results apply independently of number of songs. The variations within the results are presented in Table 6, where the uncompressed representation is compared with each of the three remaining representations in order to deduce the actual reduction gained.

Table 6: *The variations of space reduction when comparing the uncompressed type with each of the remaining types.*

No. of songs	Uncomp. –> WAH	Uncomp. –> AVD	Uncomp. –> AVD+WAH
1,000	7.0%	24.6%	26.0%
10,000	11.0%	37.2%	42.9%
50,000	14.6%	37.8%	46.7%
100,000	20.2%	38.4%	46.6%

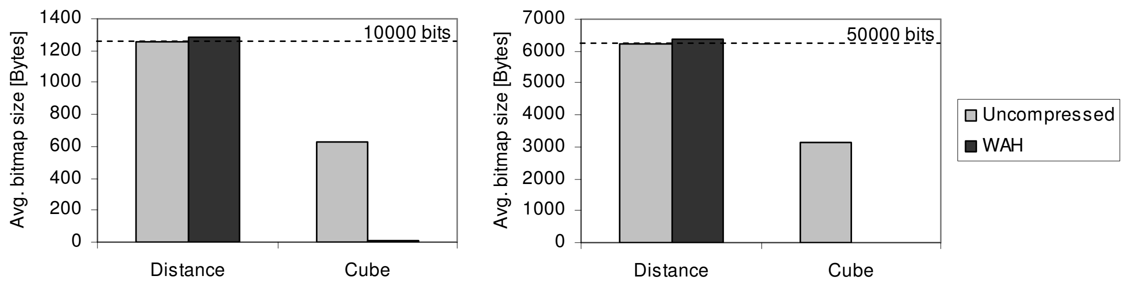
Based on Table 6 we conclude that, the most space optimized indexing is achieved, when applying both AVD and WAH compression. In doing so a reduction of 46.6% for indexing 100,000 songs is achieved when compared to the space consumption of the uncompressed music collection. Additionally, we can compare the space consumption with the space needed to store the actual digitized music. Assuming average sized songs, 1,000 songs occupy 4GB, which can be scaled linearly to 40GB for 10,000 songs and so on. The results of the index size for AVD+WAH compared to the number of songs and the space occupied by those are presented in Table 7. As expected, it can be seen from Table 7, that the more songs that are indexed, the more space is required in average for each song.

Table 7: *Comparison of space consumption of songs and index size.*

No. of songs	Space consumption of songs	Index size	Index size / song
1,000	4GB	3.4MB	3.5kB
10,000	40GB	106.7MB	10.9kB
50,000	200GB	2.7GB	56.6kB
100,000	400GB	10.6GB	111.0kB

Moreover, it is interesting to consult the average bitmap sizes for the applied indexing. The average compression that is possible when considering all bitmaps used for indexing is reflected by the reduced space consumption as shown in Table 6. However, it is expected that a difference exists when considering the bitmaps of the metadata cube and the distance management in isolation. Intuitively, the bitmaps within the metadata cube contains many consecutive 0 bits and are by this subject to high compression while the bitmaps of the distance management are more diversified with respect to occurrences of 0 and 1 bits.

In Figure 24 the average bitmap sizes are presented for bitmaps within the metadata cube and bitmaps within the distance management, respectively. The distance management is configured to use AVD, for which reason the 12 partitions are represented



(a) *Average bitmap sizes for 10,000 songs.* (b) *Average bitmap sizes for 50,000 songs.*

Figure 24: *Average bitmap size for the distance management and the metadata cube while having 12 partitions and applying AVD.*

using only seven bitmaps. In the figures the dashed horizontal lines indicates the threshold size needed to store 10,000 and 50,000 bits, respectively. Concerning the average bitmap sizes for the distance management (70,000 and 350,000 bitmaps, respectively) the WAH compression yields a space overhead, i.e., no compression is possible. The average bitmap sizes for the metadata cube (21,216 and 105,802 bitmaps, respectively) is, as expected, reduced significantly when applying WAH compression. In this case nine bytes are used for the average bitmap for both 10,000 and 50,000 songs. This in turn causes the pillars representing the cube to become invisible on Figure 24. In the case of uncompressed bitmaps the average bitmap size within the cube is much below the threshold size of 10,000 and 50,000 bits respectively. The reason for this is that 0 bits in the end of the bitmaps is omitted, as explained in Section 5.1.

Finally, it is interesting to see how the number of partitions used within the distance management influence the space consumption for the four different bitmap representations. Figure 25 presents test results for 10,000 songs having a varying number of partitions. As it can be seen, the space consumption having uncompressed bitmaps increase linearly as the number of partitions increases. The other three bitmap representations seem to reach an upper bound. The growth of the WAH compressed type was expected to become minimal around 62 partitions when applying worst-case WAH compression as discussed in Section 5.3. This seems to be confirmed by the test results in Figure 25. Reverting to the theoretical worst-case calculations presented in Equation 8 on page 25, the results using 100 partitions would be 766MB for indexing 10,000 songs. This can be compared to the test results for WAH compression given 100 partitions which implies a space consumption of 480MB. A difference was expected as we compare against worst-case calculations. We can conclude from this that WAH compression performs better in practice than compared to the worst-case. Moreover, the resulting line nears an asymptote around the expected 62 partitions.

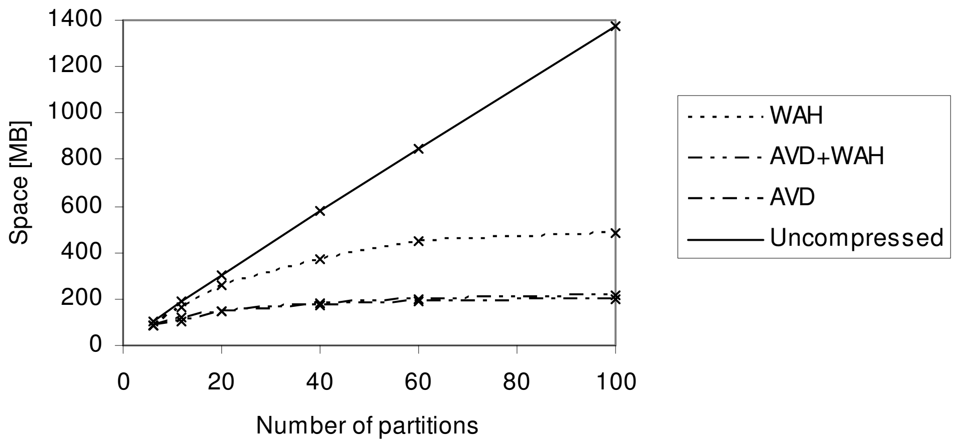


Figure 25: *Space consumption for indexing 10,000 songs on all bitmap representations.*

The remaining two bitmap representations are very close as seen on Figure 25. This indicates that WAH compression on an AVD represented distance store does not gain a notable reduction. In fact, for 6 and 12 partitions a minor overhead is introduced. For 20 partitions and above a minor reduction is achieved. Still the space consumption seems to be bounded as the number of partitions increased. The bound, however, is no longer influenced by the WAH compression but instead by the chosen bases of the AVD representation. Table 8 presents the bases chosen for the different number of partitions. These reflects the same development with respect to databases applying AVD as illustrated in Figure 25.

Table 8: *The chosen AVD bases and the number of bitmaps required for each distinct number of partitions.*

Number of partitions	Chosen AVD bases	Bitmaps required
6	<3, 2>	5
12	<4, 3>	7
20	<3, 3, 3>	9
40	<4, 4, 3>	11
60	<4, 4, 4>	12
100	<4, 3, 3, 3>	13

To conclude on the space consumption, we have obtained results showing that we are able to create an index for 10,000 songs occupying 106MB of space when applying both WAH and AVD. The space consumption should be compared to the size of the digitized music, which in the case of 10,000 songs is estimated to 40GB. Considering the space consumption per song, it increases squared as the number of songs increases. The main reason for this is that the squared number of distances between all songs should be maintained within the distance management. However, the space consumption for indexing 100,000 songs is found to be 111kB per song, which is just 2.7% of the assumed average song size of 4MB.

Concerning the different bitmap representations, the most space efficient indexing is found to be when applying both WAH and AVD. Compared to the uncompressed representation we gain more than a 40% space reduction with respect to vast music collections. In general, the space reduction obtained by applying AVD is significant and AVD is thereby an obvious choice within the distance management. Moreover, only applying AVD we avoid the WAH compression overhead. However, the bitmaps found within the metadata cube are then not reduced in size as it would have been the case when applying WAH compression. Hence, the total space reduction is more optimal applying both WAH and AVD.

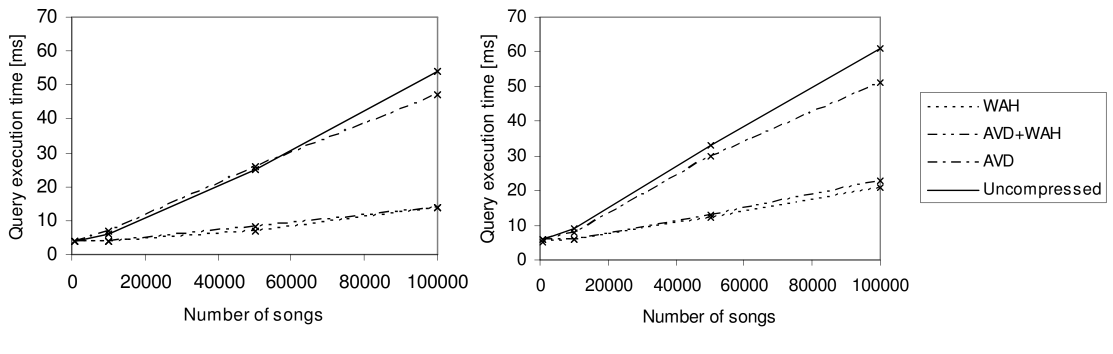
To optimize the space consumption one might suggest to separate the usage of WAH compression such that we apply WAH compression on the metadata cube and solely AVD on the distance management. However, such a solution should be able to manage and combine the diversified bitmap representations. Implementing and testing such an approach is subject for future work.

Furthermore, we have examined the impact of the number of partitions within the distance stores of the distance management. As explained, the more partitions chosen, the better the distance stores are able to reflect the similarity measure applied. However, a rather limited number of partitions might in fact not influence the quality experienced by the subjective listener. The space consumption is found to be bounded by the chosen bases used for the AVD representation when varying the number of partitions. Hence, it is interesting to note, that an increase to the number of partitions only has insignificant influence on the number of bitmaps needed to represent the bases, as described in Table 8.

7.3 Query Performance

In addition to the space experiments, we consider query performance when evaluating queries to obtain random and similar songs, respectively. As for the space experiments we conduct the query performance experiments on the four different bitmap representations as well as for databases indexing a different number of songs.

In Figure 26, the average query execution time is presented for both random and similar songs. From the figure we can deduce that, for both types of queries, the query execution time increase as the number of songs increases. All average query execution times on a collection of 100,000 songs are found to be at most 60ms. In case of solely applying WAH compression we have obtained an average query execution time at 14ms and 21ms for querying random and similar songs, respectively. Comparing the two types of queries the results obtained reflect each other as the number of songs indexed increases, except that all average query execution times for a random song are a little



(a) *Random songs.* (b) *Similar songs.*

Figure 26: *Average query execution time used to handle a request for random and similar songs, respectively. A fixed number of 12 partitions for the distance stores is used.*

faster than for the corresponding similar song query. The reason for this difference is due to, that a random song is retrieved within a small subset of the entire collection. In average, the bitmap representing this small subset has many omitted 0 bits in the end. Therefore, bit-wise operations perform faster.

Moreover, it can be seen that the two WAH compressed representations yield faster query evaluation compared to the uncompressed representations. This was however not expected as WAH applies an additional computation overhead when performing bit-wise operations on the bitmaps as explained in Section 5.1. The reason for this is explainable by the reduced size of the bitmaps when searching for a candidate song in partitions of a distance store. Requesting a random song, the bitmap representing the random set of songs chosen is small when applying WAH compression. Similarly, when requesting a similar song, the song to return is often found by consulting only the partitions representing the most similar songs. As stated earlier, these partitions only hold few songs, for which reason the corresponding bitmaps are small when applying WAH compression. Performing a bit-wise operation where one of the bitmaps in the argument is compact, i.e., it contains few literal words, the performance is increased compared to the case of having two large bitmaps in the argument.

All the previously presented query performance experiments was conducted having 12 distance partitions. Varying the number of partitions might influence the results. Experiments to consult this issue have been conducted specifying the number of partitions at 6, 12, 20, 40, 60 and 100. However, the results indicated an insignificant increase in the query execution time as the number of partitions increases. The tests were conducted on databases indexing 10,000 songs where the maximum average query execution time measured was 14ms for requesting a similar song given an uncompressed bitmap representation and 100 partitions. More deviating results might have been observed when indexing more songs in the consulted databases. However, these experiments have not been conducted due to lack of available space in order to store the required number of databases indexing 100,000 songs.

As the previous tests have been conducted while measuring an average query execution time, it is interesting to consider a worst-case query execution time. The worst-case execution time is to be found when an additional number of songs have to be skipped, i.e., the skipped songs are to be appended to the cached composite skip distance store. In the case of a music player interacting with the MOD framework, the usual way to interact would be either continuing with an unchanged collection of skipped songs or having a single additional skipped song. On the contrary one might imagine a music player able to skip several songs simultaneously, which is reflected by the worst-case query execution times presented in Figure 27. The results are the maximum query execution time when performing three independent requests, where we force generation of the composite skip distance store for the skipped songs.

As can be seen from Figure 27 the results are linear, which reflects the expected linearity of appending skipped songs to the composite skip distance store. Independent on the chosen bitmap representation, less than 250ms is required to generate the composite skip distance store when none or a single additional song is skipped. When skipping 100

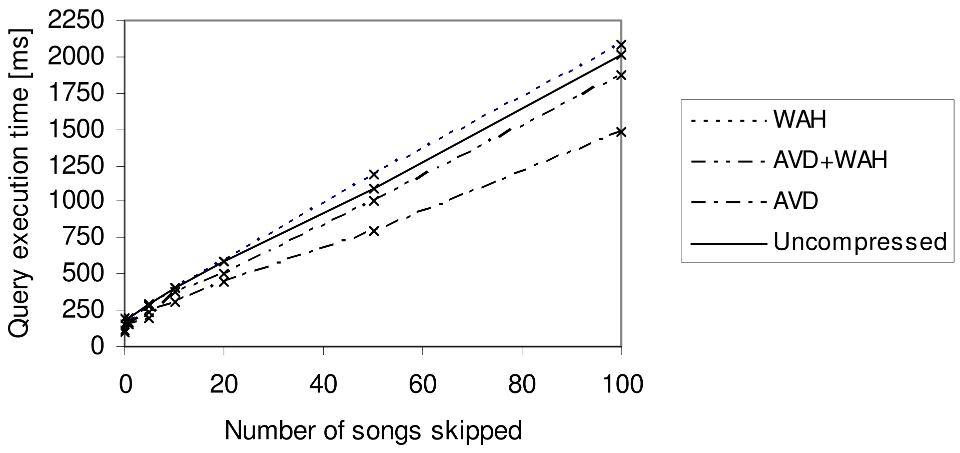
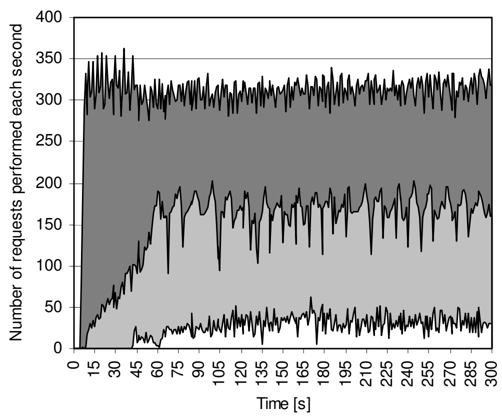


Figure 27: *Query execution time in the worst-case is considered to be equal to the case of generating the composite skip distance store. The tests are conducted for each of the four bitmap representation, while using 12 partitions for indexing 100,000 songs.*

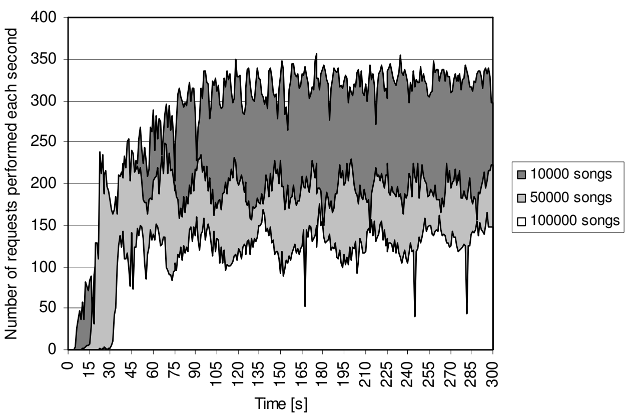
songs for each bitmap representation, we initially see that the WAH representation takes as long as 2.1s to construct the composite skip distance store. For the same amount of skipped songs, the AVD representation performs faster compared to the three other representations. As we consider generation of the composite skip distance store, distance stores for all the skipped songs should be retrieved from the database. Using an AVD representation of the distance stores, fewer records should be fetched, which explains the improved query performance. However, applying both AVD and WAH compression no reduction is achieved. The reason for this is that, when applying WAH compression we introduce an overhead in execution time when accessing the partitions of the distance store as they are represented by AVD.

The previous tests consider the query performance for single requests. However, the MOD framework is not limited to a single-user environment, as it may be used in a server setup. In the following we conduct a throughput test to examine how many requests the MOD framework is able to handle over time, when a different number of songs are indexed. To conduct the tests we create multiple *request threads*, which simulates music players, including history management, restriction and handling of skipped songs. As for all of the other tests performed, we restrict the collection to 75% of the available songs and specify 50 randomly skipped songs. The request threads performs both random and similar requests, switching between performing 20 random requests and 20 requests of similar songs for a single seed song. The tests are conducted by instantiating 50 threads, where one half starts by requesting similar songs, while the rest starts by requesting random songs.

In Figure 28 the results obtained by execution of the throughput test are presented. The results have been obtained running both the SQL Server and all request threads on



(a) *Neither AVD nor WAH are applied.*



(b) *Both AVD and WAH are applied.*

Figure 28: *The number of requests accomplished per second, where 50 threads continuously perform both random and similar requests.*

a single machine, as specified earlier. The graphs indicates that, for all the test setups, no requests are served in the beginning of the conducted tests. In addition, some time elapses until the number of requests served stabilize. The reason for this behavior is that the composite skip distance stores are generated during the first requests.

Figure 28(a) presents the results when applying neither AVD nor WAH. In this case we are able to serve around 300, 150 and 25 requests per second for indexing 10,000, 50,000 and 100,000 songs, respectively. As expected from the previous results obtained, the performance decreases when the number of indexed songs increases. When applying both AVD and WAH we have obtained the results presented in Figure 28(b). With respect to 10,000 songs we see that AVD and WAH does not increase the number of request the MOD framework can handle. However, having more songs we observe an increase. For 50,000 and 100,000 songs we are able to handle around 175 and 100 requests per second, respectively.

Assuming an average request frequency for each listener, the number of requests per second can be turned into a the number of users that can be served simultaneously. Moreover, assuming an average duration of three minutes per song, the average request frequency of a listener can be assumed to be once every three minutes.

Assuming an average duration of three minutes per song, the average request frequency of a listener can be set to once every three minutes. Hence, converted into seconds the frequency is $5.56 \cdot 10^{-3}$ request per second. Thereby, serving 100 requests per second can be deduced to correspond to approximately 18,000 simultaneous listeners. Indeed, considering the test setup with respect to hardware limitations, the obtained results seems very promising.

To conclude on the performance tests we have found that the databases applying WAH compression are the fastest to perform both random and similar queries. Using solely WAH compression we have obtained an average query execution time of 14ms and 21ms for requesting a random and similar songs, respectively on a database indexing 100,000 songs. Additionally, applying AVD for the distance stores a minor overhead is introduced, now having average query execution times of 14ms and 23ms, respectively.

Moreover, concerning the amount of partitions specified we have found that query execution time seams only to be influenced insignificantly as the number of partitions increases. These tests have only been conducted using 10,000 songs, for which reason other results may be found from databases indexing 100,000 songs.

Concerning the performed throughput test we have found that the MOD framework configured to use both AVD and WAH, is able to support 18,000 simultaneous users running on a regular laptop, when 100,000 songs are indexed.

Finally, with respect to both the space consumption and performance experiments we are able to conclude that applying both AVD and WAH compression provides the best tradeoff between the space consumption and the average query execution time. Assuming 12 partitions we are to index 100,000 songs using 10.6GB of space and perform queries of a random and similar song in 14ms and 23ms, respectively. We believe this is sufficient to support most applications using the MOD framework.

8 Conclusion and Future Work

To ensure efficient navigation within vast music collections, we have presented and evaluated the *Music On Demand framework* capable of supporting retrieval of songs in a continuous stream. The listener can retrieve either randomly chosen songs or songs similar to a given seed song. Furthermore, the listener is able to influence the stream dynamically, by skipping disliked songs or by restricting the collection by combining metadata attributes such that only desired songs are subject for retrieval. In addition, the listener can specify new collections representing a subset of the entire collection, which may be used for restriction purposes.

We have constructed a generic music data model and query operators, where a combination of both the metadata of music and the musical content of songs is included. In this context, songs may be retrieved with respect to their metadata and/or their content based similarity.

In order to ensure efficient retrieval of songs from a vast music collection, we have applied bitmap indices to index the metadata of music. The bitmaps for this purpose are maintained in a multidimensional cube mapped to a snowflake schema in an RDBMS.

Due to the fact that musical similarity is a subjective matter, there exist no unique correct answer as to whether two distinct songs are considered similar. Thus, we have applied bitmap indices to represent groupings of similar songs with respect to a base song. Based on that we are able to identify and retrieve similar/dissimilar songs efficiently using bit-wise operations on the bitmaps. We believe to be the first to use bitmap indexing techniques to facilitate retrieval and restriction queries in vast music collections, which combines the metadata and the content based similarity of music.

To reduce space consumption we have applied the Word-Aligned Hybrid compression scheme in order to compress bitmaps. In addition, we have examined the use of the Attribute Value Decomposition technique applied on the bitmap indices within the distance management. Experiments have shown that Attribute Value Decomposition should be applied in the MOD framework as the technique reduces the space consumption significantly at the cost of only a small decrease of the query performance. When considering both space consumption and query performance, the best test results were obtained having the framework configured to use both the Word-Aligned Hybrid compression scheme and Attribute Value Decomposition. Indexing 100,000 songs, 10.6GB of space is occupied, while querying of randomly chosen songs and similar songs are performed at an average of 14ms and 23ms for each song, respectively. In addition, the MOD framework may be used in a server setup, where a single instance serves multiple music players. A throughput test on 100,000 songs, indicates that the MOD framework running on a standard laptop is able to serve 18,000 simultaneous users. Indeed, considering the test setup with respect to hardware limitations, the obtained results seems very promising.

As future work, we address the use of the Word-Aligned Hybrid bitmap representation in order to optimize to the performance of the MOD framework. Based on the test results, a significant reduction of the space occupied by the metadata of music was observed when Word-Aligned Hybrid compression was applied. However, concerning the distance management, Word-Aligned Hybrid compression implies an overhead. For this reason we suggest a framework which allows to apply Word-Aligned Hybrid compression on the bitmaps within the metadata cube, while omitting it from the bitmaps within the distance management. Such an extension should enable support for combining both types of bitmaps within the bit-wise bitmap operations.

With respect to the costs found from the algorithms used to perform query evaluation, we observe that the operations are bounded by the bit-wise operations. In case that numerous bitmaps are to be combined using regular bit-wise operations, *lazy* implementations of the Word-Aligned Hybrid compressed bitmap operations could increase the overall performance of the algorithms. Hence, rather than generating several intermediate results, the full result is generated and returned only when required. Thus, instead of instantly performing several bit-wise operation in order to obtain the intermediate results, the bitmaps are to be stored in a special structure delaying the combination until the result is required. Hence, special multi bitmap bit-wise operations may consult all the bitmaps in parallel when deducing the result and thereby increase the performance.

By applying the similarity measure from the Intelligent Sound Processing toolbox R1 to calculate the distances between any two songs, we were able to supply satisfactory results for both finding similar songs and avoiding retrieval of songs similar to skipped songs. Moreover, as we apply the MOD framework we are able to answer fast. Finally to conclude, we believe that the MOD framework provides efficient retrieval of music within vast music collections.

Acknowledgment

We would like to thank the authors of the Intelligent Sound Processing toolbox R1 for providing the similarity measure algorithm we have used within the project. In addition, we would like to thank the Intelligent Sound project for giving us the oppotunity to present our work in progress on the 2nd Intelligent Sound workshop, Sæby 2006.

References

[AKS02] A.Silberschatz, H.F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., 2002. ISBN 0-07-120489-x.

[AP02] J. Aucouturier and F. Pachet. Music Similarity Measures: What's the Use? In *Proceedings of the 3rd International Conference on Music Information Retrieval*, pp. 157–163, 2002.

[BT90] S. H. Boutcher and M. Trenske. The Effects of Sensory Deprivation and Music on Perceived Exertion and Affect During Exercise. *Journal of Sport and Exercise Psychology*, 12(2):167–176, 1990.

[CI98] C. Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 355–366, 1998.

[CPZ97] 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 Very Large Data Bases*, pp. 426–435, 1997.

[DN05] C. Digout and M. A. Nascimento. High-Dimensional Similarity Searches Using A Metric Pseudo-Grid. In *Proceedings of the 21st International Conference on Data Engineering Workshops*, pp. 1174–1183, 2005.

[ID305] ID3v2. ID3v2 - The audience is informed, 2005.

http://www.id3.org (Seen June 4th, 2006).

[Int06a] Intelligent Sound Project. Intelligent Sound – A research project on search in sound files, 2006.

http://www.intelligentsound.org (Seen May 26th, 2006).

[Int06b] Intelligent Sound Project. Intelligent Sound – Home of the MATLAB toolbox, 2006.

http://isound.kom.auc.dk (Seen June 4th, 2006).

[JMS05] C. Jensen, E. Mungure, and K. Sørensen. A Foundation for Playlist Generation based on Musical Content., 2005. DAT 5 project at Aalborg University.

https://www.cs.aau.dk/library/files/rapbibfiles1/1137101143.pdf
(Seen May 26th, 2006).

[Joh99] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 278–289, 1999.

[JPH00] J. R. Deller Jr., J. G. Proakis, and J. H. Hansen. *Discrete-Time Processing of Speech Signals*. IEEE Press, 2nd edition, 2000. ISBN 0-7803-5386-2.

[KR02] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2002. ISBN 0-471-20024-7.

[KRT$^+$98] R. Kimball, L. Reeves, W. Thornthwaite, M. Ross, and W. Thornwaite. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses*. John Wiley & Sons, Inc., 1998. ISBN 0-471-25547-5.

[LS01] B. Logan and A. Salomon. A Music Similarity Function based on Signal Analysis. In *Proceedings of IEEE International Conference on Multimedia and Expo*, pp. 745–748, 2001.

[Lüb05] D. Lübbers. SoniXplorer: Combining Visualization and Auralization for Content-Based Exploration of Music Collections. In *Proceedings of the 6th International Conference on Music Information Retrieval*, pp. 590–593, 2005.

[Mas86] P. M. Maslar. The Structural Components of Music Perception: A Functional Anatomical Study. *The Arts in Psychotherapy*, 13(3):215–219, 1986.

[ME05] M. Mandel and D. Ellis. Song-Level Features and SVMS for Music Classification. *Presented at the 2nd Annual Music Information Retrieval Evaluation eXchange*, 2005.

http://www.music-ir.org/evaluation/mirex-results/articles/audio_genre/
mandel.pdf (Seen June 11th, 2006).

[NDR05] R. Neumayer, M. Dittenbach, and A. Rauber. PlaySOM and PocketSOMPlayer, Alternative Interfaces to Large Music Collections. In *Proceedings of the 6th International Conference on Music Information Retrieval*, pp. 618–623, 2005.

[OG95] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[OQ97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 38–49, 1997.

[Pam05] E. Pampalk. Speeding up Music Similarity. *Presented at the 2nd Annual Music Information Retrieval Evaluation eXchange*, 2005.

http://www.music-ir.org/evaluation/mirex-results/articles/audio_genre/pampalk.pdf (Seen June 11th, 2006).

[Pan06] Pandora Media, Inc. Music Genome Project^{TM}, 2006.

http://www.pandora.com/mgp.shtml (Seen May 26th, 2006).

[PFW05] E. Pampalk, A. Flexer, and G. Widmer. Improvements of Audio-Based Music Similarity and Genre Classification. In *Proceedings of the 6th International Conference on Music Information Retrieval*, pp. 628–633, 2005.

[PJ05] T. B. Pedersen and C. S. Jensen. Multidimensional Databases. In *The Industrial Information Technology Handbook*, pp. 1–13, Chapter 12. CRC Press, 2005. ISBN 0-8493-1985-4.

[PJD99] Torben Bach Pedersen, Christian S. Jensen, and Curtis E. Dyreson. Extending Practical Pre-Aggegation in On-Line Analytical Processing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 663–674, 1999.

[PPW05] E. Pampalk, T. Pohle, and G. Widmer. Dynamic Playlist Generation Based on Skipping Behavior. In *Proceedings of the 6th International Conference on Music Information Retrieval*, pp. 634–637, 2005.

[PRP02] D. Pedersen, K. Riis, and T. B. Pedersen. A Powerful and SQL-compatible Data Model and Query Language for OLAP. In *Proceedings of the 13th Australasian Conference on Database Technologies*, pp. 121–130, 2002.

[RTG00] Y. Rubner, C. Tomasi, and L. J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40 (2):99–121, 2000.

[SDHS00] K. Stockinger, D. Düllmann, W. Hoschek, and E. Schikuta. Improving the Performance of High-Energy Physics Analysis through Bitmap Indices. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pp. 835–845, 2000.

[SS02] P. Salembier and T. Sikora. *Introduction to MPEG-7: Multimedia Content Description Interface*. John Wiley & Sons, Inc., 2002. ISBN 0-471-48678-7.

[Tho97] E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems.* John Wiley & Sons, Inc., 1997. ISBN 0-471-14931-4.

[TLL$^+$04] G. Tenenbaum, R. Lidor, N. Lavyan, K. Morrow, S. Tonnel, A. Gershgoren, J.Meis, and M. Johnson. The Effect of Music Type on Running Perseverance and Coping with Effort Sensations. *Psychology of Sport and Exercise*, 5(2): 89–109, 2004.

[WOS06] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing Bitmap Indices With Efficient Compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[WOSN] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on Design and Implementation of Compressed Bit Vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory

http://crd.lbl.gov/˜kewu/ps/PUB-3161.pdf (Seen June 5th, 2006).

[xip05] xiph.org. Ogg Vorbis Documentation, 2005.

http://www.xiph.org/vorbis/doc/Vorbis_I_spec.html (Seen June 5th, 2006).

A Database Table Definitions

This appendix contains database table definitions for the database tables managed by the MOD framework. All the definitions are presented by SQL Server DDL (Data Definition Language) commands, which are used to create the tables. In the following, these are presented separately for the metadata cube and the distance management.

Metadata Cube

The DDL commands are presented separately for each metadata dimension specified.

Album Dimension

```
CREATE TABLE Album (
   ID     int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Album  varchar(50)       COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap image                                            NULL
)
```

Artist Dimension

```
CREATE TABLE Artist (
   ID     int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Artist varchar(50)       COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap image                                            NULL
)
```

Filename Dimension

```
CREATE TABLE Filename (
   ID       int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Filename varchar(512)      COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap   image                                            NULL
)
```

Genre Dimension

```
CREATE TABLE Genre (
   ID     int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Genre  varchar(50)       COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap image                                            NULL
)

CREATE TABLE Subgenre (
   ID       int INDENTITY(1,1) PRIMARY KEY NONCLUSTERED       NOT NULL,
   Subgenre varchar(50)        COLLATE Danish_Norwegian_CI_AS NOT NULL,
   Genre    int                                               NOT NULL,
   Bitmap   image                                             NULL,
   CONSTRAINT FK_Subgenre_Genre FOREIGN KEY (Genre) REFERENCES Genre(ID)
)
CREATE CLUSTERED INDEX IX_Subgenre ON Subgenre(Genre)
```

Release Dimension

```
CREATE TABLE Century (
   ID      int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Century varchar(50)       COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap  image                                            NULL
)

CREATE TABLE Decade (
   ID      int INDENTITY(1,1) PRIMARY KEY NONCLUSTERED       NOT NULL,
   Decade  varchar(50)        COLLATE Danish_Norwegian_CI_AS NOT NULL,
   Century int                                               NOT NULL,
   Bitmap  image                                             NULL,
   CONSTRAINT FK_Decade_Century FOREIGN KEY (Century)
              REFERENCES Century(ID)
)
CREATE CLUSTERED INDEX IX_Decade ON Decade(Century)

CREATE TABLE Year (
   ID     int INDENTITY(1,1) PRIMARY KEY NONCLUSTERED       NOT NULL,
   Year   varchar(50)        COLLATE Danish_Norwegian_CI_AS NOT NULL,
   Decade int                                               NOT NULL,
   Bitmap image                                             NULL,
   CONSTRAINT FK_Year_Decade FOREIGN KEY (Decade) REFERENCES Decade(ID)
)
CREATE CLUSTERED INDEX IX_Year ON Year(Decade)
```

Title Dimension

```
CREATE TABLE Title (
   ID     int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Title  varchar(512)      COLLATE Danish_Norwegian_CS_AS NOT NULL,
   Bitmap image                                            NULL
)
```

Cube Management

```
CREATE TABLE Dimension (
   DimID int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Name  varchar(50)       COLLATE Danish_Norwegian_CI_AS NOT NULL
)

CREATE TABLE Hierarchy (
   DimID      int                                        NOT NULL,
   Name       varchar(50) COLLATE Danish_Norwegian_CI_AS NOT NULL,
   SuperName  varchar(50) COLLATE Danish_Norwegian_CI_AS NOT NULL,
   LevelDepth tinyint                                    NOT NULL,
   CONSTRAINT FK_Hierarchy_Dimension FOREIGN KEY (DimID)
              REFERENCES Dimension(DimID)
)
CREATE CLUSTERED INDEX IX_Hierarchy ON Hierarchy(DimID)
CREATE TABLE UserDimension (
   DimID  int IDENTITY(1,1) PRIMARY KEY CLUSTERED          NOT NULL,
   Name   varchar(50)       COLLATE Danish_Norwegian_CI_AS NOT NULL,
   UserId int                                              NOT NULL
)

CREATE TABLE UserHierarchy (
   DimID      int                                        NOT NULL,
   Name       varchar(50) COLLATE Danish_Norwegian_CI_AS NOT NULL,
   SuperName  varchar(50) COLLATE Danish_Norwegian_CI_AS NOT NULL,
   LevelDepth tinyint                                    NOT NULL,
   CONSTRAINT FK_UserHierarchy_UserDimension FOREIGN KEY (DimID)
              REFERENCES UserDimension(DimID)
)
CREATE CLUSTERED INDEX IX_UserHierarchy ON UserHierarchy(DimID)
```

Distance Management

```
CREATE TABLE Distance (
   RefSongPosition int   PRIMARY KEY CLUSTERED NOT NULL,
   Bitmap          image                       NULL
)
```

B Music On Demand API

This section outlines the functionalities provided by the MOD API. The functionalities provide the possibility to retrieve songs, manage user dimensions and browse through the entire collection using metadata. In addition, it is also possible to insert songs into the framework and close the database connection. The MOD API is presented in the following as Java methods together with the JavaDoc documentation.

```
public interface MODAPI {

   /******************************* Song Retrieval ***********************************/

   /**
    * Finds and returns a random song. The song returned is neither in the
    * list of skipped songs nor similar to any of the skipped songs.
    *
    * @param collection as a bitmap presenting songs in the restricted music collection.
    * @param history    as a bitmap presenting songs recently played which are
    *                   not subjected for retrieval.
    * @param skiplist   as a bitmap presenting songs that are currently skipped.
    * @param quality    stating the number of candidate songs to be found.
    * @param userid     identifies the music player application.
    * @return a bitmap containing one 1 bit for the randomly chosen song from the
    *         collection.
    * @throws EmpytBitmapException if there was no available song to return.
    */
   public int[] getRandomSong(int[] collection, int[] history, int[] skiplist,
             int quality, int userid) throws EmptyBitmapException;



   /**
    * Finds and returns a song similar to the given seed song. The song
    * returned is neither in the list of skipped songs nor similar to any of the
    * skipped songs.
    *
    * @param collection as a bitmap presenting songs in the restricted music collection.
    * @param history    as a bitmap presenting songs recently played which are
    *                   not subjected for retrieval.
    * @param skiplist   as a bitmap presenting songs that are currently skipped.
    * @param seedSong   as a position of a song for which a similar song should be
    *                   retrieved.
    * @param userid     identifies the music player application.
    * @return a bitmap containing one 1 bit for the chosen song that is similar to the
    *         seed song.
    * @throws EmpyBitmapException if there was no available song similar to the
    *         seed song.
    */
   public int[] getSimilarSong(int[] collection, int[] history, int[] skiplist,
             int seedSong, int userid) throws EmptyBitmapException;
```

```
/************************* User Dimensions Management ******************************/

/**
 * Creates a user dimension and its hierarchies.
 *
 * @param dimName            the name of the dimension table to be created.
 * @param dimensionHierarchy the levels to be created as dimension tables.
 * @param userID             the id of the user who owns the dimension.
 * @throws Exception if an invalid dimension hierarchy specified.
 */
public void createDimension(String dimName, String[][] dimensionHierarchy,
            int userID) throws Exception;


/**
 * Inserts a song into a user collection. If the collection does not exist, a new
 * is created.
 *
 * @param songInfoToInsert   indentifies the user collection of where the song
 *                           should be inserted.
 * @param bitmapSongToInsert song to be inserted presented as a bitmap having one
 *                           1 bit.
 * @param userID             identifies the music player application.
 */
public void insertSongInUserDimension(Song songInfoToInsert,
            int[] bitmapSongToInsert, int userID);


/**
 * Deletes users level in a database.
 *
 * @param levelName name of the level to be deleted.
 */
public void deleteLevel(String levelName);


/**
 * Deletes a user collection.
 *
 * @param levelName  the level name where the collection is to be deleted from.
 * @param collection a user collection to be deleted.
 */
public void deleteCollection(String levelName, String collection);


/**
 * Deletes a song in a users defined collection.
 *
 * @param levelName      to identify a user collection.
 * @param collectionName name of a collection of songs, from where the song is
 *                       to be deleted.
 * @param bitmapTodelete bitmap containing one 1 bit representing the song to
 *                       be deleted.
 */
public void deleteSongFromCollection(String levelName, String collectionName,
            int[] bitmapTodelete);
```

```
    /**************************** Browsing management ***********************************/


    /**
     * Fetches all the available dimensions for the given user in a database.
     *
     * @param userID identifies the user.
     * @return SongDimensionInfo which represents all dimensions and
     *                           all user dimensions related to the userID
     */
    public SongDimensionInfo[] getAvailableDimensions(int userID);


    /**
     * Fetches the sub dimension values for a given level.
     *
     * @param songDimInfo identifies the level to fetch sub values from.
     * @return a SongDimenensionInfo containing the sub dimension values.
     * @throws Exception if more than a single level was specified.
     */
    public SongDimensionInfo getSubItems(SongDimensionInfo songDimInfo) throws Exception;


    /**
     * Returns a full collection presented as a bitmap where all bits are set to 1.
     *
     * @return full set bitmap.
     */
    public int[] getFullCollection();


    /**
     * Fetches song information.
     *
     * @param bitmap     representing the song to fetch information.
     * @param dimensions a number of dimensions from where song information are to be
     *                   fetched.
     * @return song containing the information requested.
     */
    public Song getInfo(int[] bitmap, String[] dimensions);


    /********************* Song insertion and closing of connection *********************/


    /**
     * Inserts a number of songs and all their corresponding metadata into a database.
     *
     * @param songToInsert a number of songs to be inserted.
     */
    public void insertSong(Song[] songToInsert);


    /**
     * Closes the connection to the database safely.
     */
    public void close();

}
```