19th International Conference on Database and Expert Systems Application

A Flexible Query Framework for Music Data and Playlist Manipulation *

Maria M. Ruxanda Christian S. Jensen
Department of Computer Science, Aalborg University, Denmark
{mmr,csj}@cs.aau.dk

Abstract

Motivated by the explosion of digital music on the Web and the increasing popularity of music recommender systems, this paper presents a relational query framework for flexible music retrieval and effective playlist manipulation. A generic song representation model is introduced, which captures heterogeneous categories of musical information and serves a foundation for query operators that offer a practical solution to playlist management. A formal definition of the proposed query operators is provided, together with real usage scenarios and a prototype implementation.

1 Introduction

Digital music is becoming increasingly widespread as personal collections of music grow to thousands of songs. Moreover, successful technologies like the Mp3 format and the iPod have changed the way music is distributed and consumed. In this context, systems that offer personalized music recommendations play a crucial role in entertainment services and on-line music sales. Several commercial music recommenders have already achieved popularity, including Last.fm (www.last.fm), Pandora (www.pandora.com), MusicIP (musicip.com), and MyStrands (www.mystrands.com). The research prototype of Sun, Search Inside the Music, is also notable. While these systems build on different notions of music similarity, they all deliver various forms of playlists to their users. Specifically, the concept of a playlist has emerged as a key unit of music exchange and delivery.

The popularity of music recommenders has created a demand for the development of efficient techniques to support music retrieval and playlist manipulation. Music recommender systems have to deal with two main aspects: the modeling of music similarity and the dynamic generation of playlists based on music similarity; and the representation, storage, and querying of music data to support similarity search and playlist operations.

*This research was supported by the Danish Research Council for Technology and Production Sciences project no. 26-04-0092 Intelligent Sound (www.intelligentsound.org).

Existing research focuses primarily on music similarity. While treating music similarity from different angles, various approaches for playlists generation have been proposed. For example, some approaches for playlist generation are pure audio-based [11], other employ a hybrid (combination of audio-content and social) music similarity [7]. The creation of playlists that meet given constraints has been addressed [1, 2], and approaches that incorporate user feedback have also been considered [9, 10].

Some work has been done on data and query models for music and playlist manipulation [3–5, 12, 14]. However, the existing work either disregards similarity queries and playlists [12, 14], or addresses specific scenarios of playlist manipulation [3–5]. The existing query models are limited when seen in the broad context of music recommenders that manage music and playlists in various ways.

This paper addresses the emerging needs of music recommender systems to retrieve similar music and manage playlists over large music collections stored in databases. We develop a flexible framework that encompasses, at the database level, the functionalities of music recommenders in a generic way, so that specific strategies of music recommendation can be easily plugged in. The contributions of our work are as follows: 1) we introduce a song representation model that captures the heterogeneous musical information — metadata, audio-content data, and social/collaborative data; the foundation of our model is a conceptual, generic song-tuple, which is an abstraction over any specific database storage design; 2) we treat playlists intuitively, as ordered lists of songs, and we define query operators as an extension of a relational algebra that supports the notion of order; the query operators deal with arbitrary music similarities and cover similarity searches and similarity-based generation, shuffling, and mixing of playlists; 3) we implement a prototype version of the proposed operators in the open source PostgreSQL database management system.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the song representation model. Section 4 introduces the formal definitions of our query operators, and Section 5 presents the prototype implementation. Finally, Section 6 concludes the paper.

1529-4188/08 $25.00 © 2008 IEEE
DOI 10.1109/DEXA.2008.23

2 Related Work

Rubenstein [12] introduce a music data model that extends the entity-relationship model and implements the notion of hierarchical ordering found in musical data. Wang et al. [14] propose a music data model and query language, exemplifying their use on musical instruments retrieval. However, both models lack an adequate framework to perform similarity searches and playlist operations.

Jensen et al. [5] propose a data and query model for dynamic playlist generation that supports arbitrary similarity measures. However, the retrieval operators are limited to the case of continuous playlists, where songs are retrieved one at a time taking into account the user's skipping behavior.

Work more similar to ours was recently proposed by Deliege and Pedersen [3, 4]. A music warehouse prototype capable of performing arbitrary similarity searches is described [3], but only nearest neighbors searches are covered. A query algebra manipulating playlists, seen as fuzzy lists, is introduced [4]. However, the query model is applicable when solely modeling user feedback on the music.

Most related work is perhaps the list-based relational algebra proposed by Slivinskas et al. [13], introducing the notion of ordered lists in the relational model. While playlists are conceptually ordered lists of songs, the operators in [13] are the same as the standard ones except that they contend with order; thus, they do not cover playlist manipulation.

Our work has several advantages over previous work. We capture clearly, and in a generic manner, all categories of musical information: metadata, audio-content, and social data. Compared for example to using dimension hierarchies [4, 5], our data model is more flexible as it abstracts over database design and can be accommodated in any warehouse schema, including those mentioned above. While it is natural to consider order when dealing with playlists, the ordered relational algebra [13] does not target playlist manipulation, and the fuzzy algebra [4] addresses specific scenarios. We propose query operators that extend the algebra in [13] and capture generic usage of playlists.

3 Generic Representation of Music

Outline. Playlists are made up of songs, which are being manipulated when playlists are managed. Thus, a format for the representation of a song in a music database is needed.

Currently, music recommendation approaches are based on three categories of musical information: 1) metadata attributes e.g., the title of song, the artist's name; 2) collaborative/social data obtained by, e.g., user profiling, user tagging, or mining web blogs; 3) feature vectors extracted from the audio signal. However, various music recommenders use different subsets of the three categories of information, or they use different instances of each, and various musical aspects generate different notions of music similarity.

We aim at capturing at the database level this heterogeneous information in a flexible and generic manner, so that particular implementations of music recommendation can be accommodated. Our approach is to abstract over the design details of a specific database schema. Thus, we propose to model the notion of a song item in a music database as a conceptual view, that encapsulates the various musical attributes and their associated notion of music similarity. Then, arbitrary database schemas can be plugged in, to physically underlie the relational view of song items.

A Song Representation Model. We propose a data representation model that captures at the song level the three heterogeneous categories of musical information.

Music metadata attributes are provided with the music, e.g., when it is ripped using media player tools. They are simple and can be easily stored in database relations.

The social data are obtained by capturing the use of the music by users. We identify two steps in representing this data in a database. The first step is that user profiling data (e.g., a user's ratings, playlists or tags) and entities composing any mined social data (e.g., descriptive words mined from web blogs) are grouped and captured in database relations. The second step is to represent aggregated social data at the song level. Such data, e.g., IDs of tags used for a song or IDs of playlists in which the song appears, can be captured in an array.

Example. We capture the top 3 most used tags for a song. The social features of the song "Frozen" by Madonna are represented as the array [71, 23, 450], where the elements are IDs denoting the tags "90s", "dance", and "electronic".

A variety of audio feature vectors may also be extracted from the audio signal of a song. Widely used features include the MPEG-7 and MFCC [6] coefficients. The audio data can be represented straightforwardly at the song level as an array of elements comprising of the extracted values.

According to recent research, hybrid music similarity based on both audio and social features, is performing well [7]. Thus, audio data, social data, and their combination are all important aspects of the music, each offering useful measures of music similarity.

It is convenient to use an abstract data type to encapsulate such arrays of features (audio, social, or hybrid) and their corresponding similarity measure [1]. This way the various musical aspects that characterize a song, can each be represented as an attribute of the song.

We model these arrays as the abstract data type, denoted MFD (Music-Feature-Data) — see Table 1. Note that audio and social feature vectors are usually multi-dimensional, and audio features are frequently extracted on samples of the song so that a large number of vectors is produced.

[1] While we focus on songs, artist similarity can be addressed by aggregating over the songs by an artist and by using artist-targeted social data.

The key operation on the MFD type is the distance computation $dist$. Concrete implementations of $dist$ can use, in addition to feature representation of songs, other music information stored in database relations. Also note that the similarity of songs can be evaluated in different ways, as long as specialized $dist$ implementations are provided.

The attributes that describe a song may be captured in a database view as shown in Table 2. A song is represented by a tuple with *n+m* columns, where n is the number of metadata attributes and m is the number of MFD attributes.

Table 1. MFD Definition.

Data fields	
len	Array with the number of feature vectors per song, for each kind of feature
dim	Array with the dimensionality of feature vectors, for each kind of feature
vals	Array with the feature values; in total $\sum_{i=1}^{N}(len[i] \cdot dim[i])$ values, where N= no. of kinds of features
Operations	
MFD(len,dim,vals)	Creates an instance of MFD
getValue() methods	Get the values of the data fields
$dist(mfd_1, mfd_2)$	Computes the similarity between two MFD instances; can have various implementations. e.g., cosine, L_p

4 Query Operators for Playlist Manipulation

Usage Vision. We consider common and key functionalities provided by music recommender systems that manipulate playlists. We interpret the notion of a playlist broadly, regardless of its output form (list of songs or a radio station).

We have investigated extensively various use-cases of the popular music recommenders Last.fm, Yahoo!Music, Pandora, MusicIP, and MyStrands. Our findings suggest that several operations constitute key functionalities:

1. Similarity neighbor search. We include nearest and farthest neighbors. These are useful to support the dynamic generation of personalized playlists. While nearest neighbor searches are used the most (e.g., include in a playlist songs close to those the user likes, and exclude songs close to those the user dislikes), far neighbors can be handy when shuffling playlists to achieve a jagged similarity transition.

2. Similarity range search. We include lower or upper bounded ranges. These are useful to support the generation of personalized playlists and the "hop"-like similarity-based shuffling of playlists (see next 4.).

3. Sizing of playlists. This refers to retaining the correct size of a playlist according to the user's input settings — playlists of a specified length, duration, or file size.

4. Similarity-based shuffling of playlists. We distinguish between two cases. The first is a monotonic-like similarity ordering of songs, for example smooth or jagged transition, where the similarity distance between adjacent songs in a playlist is either minimized or maximized. The second is a "hop"-like similarity ordering of songs, where the distance between adjacent songs is beyond/below a given threshold.

5. Smart mixing of playlists. A known example here is by alternating their songs (as done in Pandora). New ways are possible by applying a similarity-based shuffling to the mix.

Based on these, we define 6 query operators (see next), which work on the proposed song representation model and use a generic song similarity measure.

Proposed Query Operators. Conceptually, a playlist is an ordered sequence of songs. Therefore, it is convenient to treat relations as ordered lists. We thus use a relational algebra with this characteristic [13]. The operators of this algebra are the standard ones adapted to address order: projection (π), selection (σ), union ($\sqcup$), duplicate elimination ($rdup$), difference ($\setminus$), Cartesian product ($\times$), aggregation (ξ), sorting ($sort$), and the top k tuples of a relation (top_k).

We extend the algebra by defining six new query operators that allow easy manipulation of playlists. We use λ-calculus for the definitions. We consider that a relation contains song-tuples, where a song-tuple is as exemplified in Table 2. The basic notations are explained in Table 3.

The operation $topN_k : [\mathcal{R} \times \mathcal{T} \times \mathcal{O}_\Omega \times \mathbb{N}] \rightarrow \mathcal{R}$ returns k song-tuples of the input relation, which are the first k near or far neighbors of the input song-tuple t (the second argument), according to a music similarity measure. Since similarity measures are encoded within the MFD attributes of song-tuples, songs comparison can then be expressed using a sorting on MFD attributes. The third argument denotes a similarity ordering. We denote the set of all possible orders for MFD attributes from Ω by $\mathcal{O}_\Omega$. For example, $(dist(\texttt{Tags}, t.\texttt{Tags}), \texttt{DESC})$ is a similarity order.

$$topN_k \triangleq \lambda r, t, o, k.(r = \perp) \rightarrow \perp, \\ top_k(sort(r, o))$$

The operation $range : [\mathcal{R} \times \mathcal{T} \times \mathbb{R} \times \{<, \leq, >, \geq\}] \rightarrow \mathcal{R}$ returns all song-tuples of the argument relation that, in terms of similarity distance towards the argument song-tuple, are within the range (lower or upper) bounded by a threshold. The threshold value val is represented by the third argument. The fourth argument denotes a relational operator. The operation uses the basic function $compare$, which returns true or false, and implements a predicate.

$$range \triangleq \lambda r, t, val, op.(r = \perp) \rightarrow \perp, \\ compare(dist(head(r).MFD, t.MFD), val, op) \rightarrow \\ head(r) \ @ \ range(tail(r), t, val, op), \\ range(tail(r), t, val, op)$$

The operation $retainSize : [\mathcal{R} \times \Omega \times \mathbb{R}] \rightarrow \mathcal{R}$ takes as arguments a relation, a numerical attribute a of the argument relation, and a real number val. It scans the argument relation until the sum of the attribute a for all scanned tuples is exceeding the value val, and it returns the scanned tuples.

Table 2. A Relational Model for Song Representation.

ID	Title	Artist	Year	...	MFCC	Tags	Genre Classif. & Playlists	...
1	Let It Be	The Beatles	1970	...	MFD([15333],[6],$vals_1$)	MFD([1],[20],$vals_2$)	MFD([1,1],[15,100],$vals_3$)	...

Table 3. Basic Notations.

$S = (\Omega, \Delta, dom)$	a *song relation schema*; Ω is a finite set of attributes, Δ is a finite set of domains, $dom : \Omega \rightarrow \Delta$ associates a domain with each attribute. Specifically, $\Omega = \{\texttt{ID}, M_1, \ldots, M_n, F_1, \ldots, F_m\}$, ID is a key attribute, M_i are metadata attributes, and $dom(F_i) = MFD$
$t : \Omega \rightarrow \cup_{\delta \in \Delta} \delta$	*a song-tuple over schema* S, such that for every attribute A of Ω, $t(A) \in dom(A)$
r	*a relation*; it is a finite sequence of song-tuples over S such that $\forall\, t_1, t_2 \in r \quad (t_1 \neq t_2) \Rightarrow (t_1[\texttt{ID}] \neq t_2[\texttt{ID}])$
$\mathcal{T}, \mathcal{R}$	$\mathcal{T}$ is the set of all song-tuples; $\mathcal{R}$ is the set of all relations comprising song-tuples
$head(r)$, $tail(r)$, @ , $\perp$	have the same definition as in [13]; $head(r)$ returns the first tuple of r; $tail(r)$ returns r without its first tuple; @ prepends a tuple to a relation; $\perp$ denotes the empty relation

$$retainSize \triangleq \lambda r, a, val.(r = \perp) \rightarrow \perp,$$
$$head(\pi_a(r)).a > val \rightarrow \perp,$$
$$head(r) \ @ \ retainSize(tail(r), a, val - head(\pi_a(r)).a)$$

The operation $alternate : [\mathcal{R} \times \mathcal{R}] \rightarrow \mathcal{R}$ merges the two input argument relations by alternating their song-tuples.

$$alternate \triangleq \lambda r_1, r_2.(r_1 = \perp) \rightarrow r_2, (r_2 = \perp) \rightarrow r_1,$$
$$head(r_1) \ @ \ (head(r_2) \ @ \ alternate(tail(r_1), tail(r_2)))$$

The operation $neighborSimOrder : [\mathcal{R} \times \mathcal{O}_\Omega] \rightarrow \mathcal{R}$ re-orders and returns the song-tuples of the argument relation. The ordering is done so that a smooth or jagged similarity transition is enforced. It uses the $topN_k$ operation.

$$neighborSimOrder \triangleq \lambda r, o.(r = \perp) \rightarrow \perp,$$
$$(tail(r) = \perp) \rightarrow head(r),$$
$$head(r) \ @ \ neighborSimOrder(t \ @ \ (tail(r) \setminus t), o)$$
$$\text{where } t = topN_1(tail(r), head(r), o, 1)$$

Operation $hopSimOrder : [\mathcal{R} \times \mathbb{R} \times \{<, \leq, >, \geq\}] \rightarrow \mathcal{R}$ re-orders and returns the song-tuples of the argument relation. The ordering enforces, whenever possible, a "hop" similarity transition (adjacent song-tuples differ in terms of similarity by a threshold value). It uses the *range* operation.

$$hopSimOrder \triangleq \lambda r, val, op.(r = \perp) \rightarrow \perp,$$
$$(tail(r) = \perp) \rightarrow head(r),$$
$$(r_1 \neq \perp) \rightarrow head(r) \ @ \ hopSimOrder(top_1(r_1) \ @$$
$$(tail(r) \setminus top_1(r_1)), val, op)$$
$$\text{where } r_1 = range(tail(r), head(r), val, op),$$
$$head(r) \ @ \ hopSimOrder(tail(r), val, op)$$

5 A Prototype Implementation

We implemented a prototype version of the proposed query operations in the open source database management system PostgreSQL. The six operators were implemented as functions in PL/pgSQL — a procedural language available in the standard PostgreSQL distribution.

As our aim is to define generic query operators and abstract over a specific database schema, the prototype implementation assumes only that: 1) there is a relation/view storing song-tuples (as shown in Table 2); 2) there is a relation/view storing playlists; and 3) these playlist-tuples have at least three attributes denoting a playlist ID, a song ID, and a track order. We also assume for simplicity that the playlist ID is unique for playlists across all users. We judge the above assumptions to be realistic in the application setup.

The proposed query operations were implemented in two overloaded variants. The first assumes default names for the attributes of a playlist-tuple — see the operation signatures in Table 4. In the second variant, these attribute names are additionally given as arguments to the query operations.

For testing, we used a music database of $62,226$ Mp3 songs, having a simple schema with two relations Songs and Playlists, storing song-tuples and playlist-tuples, respectively. A song-tuple has the key attribute song_ID, the metadata attributes title, artist, album, albumYear, duration, genre, and one MFD attribute. The MFD attribute, denoted GCC, encompasses audio features (15-dim) representing genre classification coefficients [8]. To store it, we used an array of real values. As the similarity measure, we implemented the Euclidean distance as a function in PL/pgSQL.

Note that while we test in a simple setup, any storage schema and any number of MFD attributes can be used, and other similarity measures can be implemented in C, Perl, Python, or Java — which are supported by PostegreSQL.

Next, we exemplify how the six operators can support playlist operations in a real music recommendation setup (see Table 5). For instance, Example 1 can be used when playlists are generated based on a seed song, while Example 2 can be used for dynamic update of playlists (e.g., replacing a song the user dislikes with the song returned by the query). The other examples exhibit straightforward usages.

6 Conclusion

The paper proposes a relational query framework that supports a song representation model that flexibly captures heterogeneous musical information. The framework treats playlists intuitively, as ordered lists of songs, and defines query operators that accommodate generic music similarity

Table 4. Query Operations Signatures.

topN (**songsStatement** text, **songsTable** text, **refSong_ID** integer, **distance** text, **featureAttribute** text, **orderDir** text, **k** integer) : setof integer It returns the song_IDs of the k found neighbors.
range (**songsStatement** text, **songsTable** text, **refSong_ID** integer, **distance** text, **featureAttribute** text, **orderDir** text, **op** text, **threshold** real) : setof integer It returns the song_IDs of the songs found in the given range.
retainSize (**songsStatement** text, **songsTable** text, **cutAttribute** text, **threshold** real) : setof integer It returns the song_IDs that fit in the specified playlist size. If **cutAttribute**='length', it returns the first (int)**threshold** song_IDs.
alternate (**playlistsStatement** text, **arrayOfPlaylistsIDs** text) : setof integer It returns the song_IDs of the input playlists in alternating order. For efficiency, it does not use the song-tuples, but only the song_ID attribute stored in the playlist-tuples.
neighborSimOrder (**playlistsStatement** text, **arrayOfPlaylistsIDs** text, **songsTable** text, **distance** text, **featureAttribute** text, **orderDir** text) : setof integer It returns the song_IDs of the input playlist(s) in the new order. It can be used for both single playlist-shuffling or playlists-mixing.
hopSimOrder (**playlistsStatement** text, **arrayOfPlaylistsIDs** text, **songsTable** text, **distance** text, **featureAttribute** text, **orderDir** text, **op** text, **threshold** real) : setof integer It returns the song_IDs of the input playlist(s) in the new order. It can be used for both single playlist-shuffling or playlists-mixing.
where **songsStatement** is a table/view/select-statement comprising song_IDs; **songsTable** is a table/view storing song-tuples; **distance** is a function implementing a similarity measure; **featureAttribute** is the MFD attribute corresponding to the similarity measure; **orderDir** $\in$ {'ASC', 'DESC'}; **op** $\in$ {'<', '<=', '>', '>='}; **cutAttribute** is a metadata attribute of the song-tuples, e.g., 'duration'; **arrayOfPlaylistsIDs** is, e.g., '[12,3,5]'; **playlistsStatement** is a table/view/select-statement comprising playlist-tuples;

measures and encompass the functionalities offered by contemporary music recommenders. In addition, we provide a ready-to-use prototype implementation of the proposed query operators in PostgreSQL.

Table 5. Usage Examples.

Example 1. Find the first 10 nearest neighbors of the seed song ⟨song_ID=257⟩, that appeared on an album in year 2000.

```
SELECT * FROM topN('SELECT song_ID FROM
Songs WHERE albumYear=2000','Songs',257,
'Euclidean','GCC','ASC',10)
```

Example 2. Find the first far neighbor of the seed song ⟨song_ID=9⟩ that is dissimilar by at least 0.8 threshold, and that is played by artist Madonna.

```
SELECT * FROM range('SELECT song_ID FROM
Songs WHERE artist=\'Madonna\'','Songs',9,
'Euclidean','GCC','ASC','>=',0.8) LIMIT 1
```

Example 3. Reset the size of playlist ⟨playlist_ID=5⟩, so that the duration is 60 minutes.

```
SELECT * FROM retainSize('SELECT song_ID
FROM Playlists WHERE playlist_ID=5',
'Songs','duration',60)
```

Example 4. Create an alternating mix of the songs of playlists ⟨playlist_ID=3⟩, ⟨playlist_ID=2⟩, and ⟨playlist_ID=17⟩.

```
SELECT * FROM Songs as t1, (SELECT * FROM
alternate('Playlists','[3,2,17]')) as t2
WHERE t1.song_ID=t2.alternate
```

Example 5. Create a jagged transition mix of playlists ⟨playlist_ID=10⟩ and ⟨playlist_ID=11⟩.

```
SELECT * FROM neighborSimOrder('Playlists',
'[10,11]','Songs','Euclidean','GCC','DESC')
```

Example 6. Shuffle playlist ⟨playlist_ID=1⟩ by hop-similarity transition with a strict hop of at least 0.5.

```
SELECT * FROM hopSimOrder('Playlists','[1]',
'Songs','Euclidean','GCC','ASC','>',0.5)
```

References

[1] M. Alghoniemy and A. Tewfik. A Network Flow Model for Playlist Generation. In *Proc. ICME*, pages 84–95, 2001.

[2] J. Aucouturier and F. Pachet. Scaling up Music Playlist Generation. In *Proc. ICME*, pages 105–108, 2002.

[3] F. Deliege and T. B. Pedersen. Fuzzy Song Sets for Music Warehouses. In *Proc. ISMIR*, pages 21–26, 2007.

[4] F. Deliege and T. B. Pedersen. Using Fuzzy Lists for Playlist Management. In *Proc. MMM*, pages 198–209, 2007.

[5] C. A. Jensen, E. M. Mungure, T. B. Pedersen, and K. I. Sørensen. A Data and Query Model for Dynamic Playlist Generation. In *Proc. ICDE IEEE Workshop*, pages 65–74, 2007.

[6] J. Jensen, M. Christensen, M. Murthi, and S. Jensen. Evaluation of MFCC Estimation Techniques for Music Similarity. In *Proc. EUSIPCO*, 2006.

[7] P. Knees, T. Pohle, M. Schedl, and G. Widmer. Combining Audio-based Similarity with Web-based Data to Accelerate Automatic Music Playlist Generation. In *Proc. ACM Int. Workshop on MIR*, 2006.

[8] T. Lehn-Schiøler, J. Arenas-Garcia, K. Petersen, and L. Hansen. A Genre Classification Plug-in for Data Collection. In *Proc. ISMIR*, pages 320–321, 2006.

[9] E. Pampalk, T. Pohle, and G. Widmer. Dynamic Playlist Generation Based on Skipping Behavior. In *Proc. ISMIR*, pages 634–637, 2005.

[10] S. Pauws and B. Eggen. PATS: Realization and User Evaluation of Automatic Playlist Generator. In *Proc. ISMIR*, 2002.

[11] T. Pohle, E. Pampalk, and G. Widmer. Generating Similarity Based Playlists Using Traveling Salesman Algorithms. In *Proc. DAFx*, pages 220–225, 2005.

[12] W. Rubenstein. A Database Design for Musical Information. In *Proc. ACM SIGMOD*, pages 479–490, 1987.

[13] G. Slivinskas, C. Jensen, and R. Snodgrass. Bringing Order to Query Optimization. *SIGMOD Record*, 31:2:5–14, 2002.

[14] C. Wang, L. J., and S. S. A Music Data Model and its Application. In *Proc. MMM*, pages 79–85, 2004.