HIGH-LEVEL AUDIO FEATURES: DISTRIBUTED EXTRACTION AND SIMILARITY SEARCH

François Deliège, Bee Yong Chua, and Torben Bach Pedersen
Department of Computer Science
Aalborg University

ABSTRACT

Today, automatic extraction of *high-level* audio features suffers from two main scalability issues. First, the extraction algorithms are very demanding in terms of memory and computation resources. Second, copyright laws prevent the audio files to be shared among computers, limiting the use of existing distributed computation frameworks and reducing the transparency of the methods evaluation process. The *iSound Music Warehouse* (iSoundMW), presented in this paper, is a framework to collect and query high-level audio features. It performs the feature extraction in a two-step process that allows distributed computations while respecting copyright laws. Using public computers, the extraction can be performed on large scale music collections. However, to be truly valuable, data management tools to search among the extracted features are needed. The iSoundMW enables similarity search among the collected high-level features and demonstrates its flexibility and efficiency by using a weighted combination of high-level features and constraints while showing good search performance results.

1 INTRODUCTION

Due to the proliferation of music on the Internet, many web portals proposing music recommendations have appeared. As of today, the recommendations they offer remain very limited: manual tagging has proved to be time consuming and often results in incompleteness, inaccuracy and inconsistency; automatic tagging systems based on web scanning or relying on millions of users are troubled, e.g., by mindless tag copying practices, thus blowing bag tags. Automatic extraction of music information is a very active topic addressed by the Music Information Retrieval (MIR) research community. Each year, the Music Information Retrieval Evaluation eXchange (MIREX) gives to researchers an opportunity to evaluate and compare new music extraction methods [9]. However, the MIREX evaluation process has proved to be resource consuming and slow despites attempts to address these scalability issues [4, 11]. So far, concerns with copyright issues have refrained the community to distribute the extraction among public computers as most algorithms require the audio material to be available in order to perform the feature extraction [1] . The features are therefore extracted from a relatively small music collection that narrows their generality and usefulness. Additionally, the feature extraction, being run by private computers on a private music collection, limits the transparency of the evaluation process. These limitations call for the development of a system able to extract meaningful, high-level audio features over large music collections. Such a system faces data management challenges. Noteworthily, the impressive amount of information generated requires an adapted search infrastructure to become truly valuable.

Our intention is to create a system able to cater for different types of features. Present literature mainly focuses on features that have either absolute or relative values, thus motivating the handling of both kinds of features. In this paper, the exact selection of the features is actually not as important as it is to demonstrate how extraction can be handled on public computers and enabling researchers to compare results obtained by using different algorithms and features.

The contributions of this paper are two-fold. First, we propose a framework for collecting high-level audio features (that were recently proposed by [5, 6, 7, 13]) over a large music collection of 41,446 songs. This is done by outsourcing the data extraction to remote client in a two-step feature extraction process: (1) dividing the audio information into short term segments of equal length and distributing them to various clients; and (2) sending the segment-based features gathered during step one to various clients to compute high-level features for the whole piece of music. Second, we propose a flexible and efficient similarity search approach, which uses a weighted combination of high-level features, to enable high-level queries (such as finding songs with a similar happy mood, or finding songs with a similar fast tempo). Additionally, to support the practical benefits of these contributions, we propose a short scenario illustrating the feature extraction and search abilities of the iSoundMW.

For the general public, the iSoundMW music offers recommendation without suffering from a "cold start", i.e., new artists avoid the penalties of not being well known, and new listeners obtain good music recommendation before being profiled. For researchers, the iSoundMW (1) offers flexi-

[1] https://mail.lis.uiuc.edu/pipermail/evalfest/2008-May/000765.html

ble search abilities; (2) enables visual comparison of both segment-based and aggregated high-level features; (3) provides a framework for large scale computations of features; and (4) gives good search performances.

The remainder of the paper is organized as follows. Related work is presented in Section 2. Section 3 offers an overview of the system, explains the process of collecting the high-level audio features, describes how similarity search with weighted coefficients is performed, and how the search can be further optimized by using range searches. Section 4 illustrates the similarity search on a concrete example. Section 5 concludes and presents future system improvements and research directions.

2 RELATED WORK

Research on distributed computing has received a lot of attention in diverse research communities. The Berkeley Open Infrastructure for Network Computing framework (BOINC) is a well-known middleware system in which the general public volunteers processing and storage resources to computing projects [1, 2] such as SETI@home [3]. However, in its current state, BOINC does not address copyright issues, does not feature flexible similarity search, and does not enable multiple steps processes, i.e., acquired results serve as input for other tasks. Closer to the MIR community, the On-demand Metadata Extraction Network system (OMEN) [15] distributes the feature extraction among *trusted nodes* rather than public computers. Furthermore, OMEN does not store the computed results, and, like BOINC, does not allow similarity search to be performed on the extracted features.

Audio similarity search is often supported by creating indexes [10]. Existing indexing techniques can be applied to index high dimensional musical feature representations. However, as a consequence of the subjective nature of musical perception, the triangular inequality property of the metric space is typically not preserved for similarity measures [14, 16]. Work on indexes for non-metric space is presented in the literature [12, 17]. Although the similarity function is non-metric, it remains confined in a pair of lower and upper bounds specifically constructed. Therefore, using these indexes would impose restrictions on the similarity values that would limit the flexibility of the iSoundMW.

3 SYSTEM DESCRIPTION

In this section, we present an overview of the system followed by a more detailed description of a two-step process for extracting high-level features. Later, we describe how flexible similarity searches are performed and how they are further optimized using user-specified constraints.

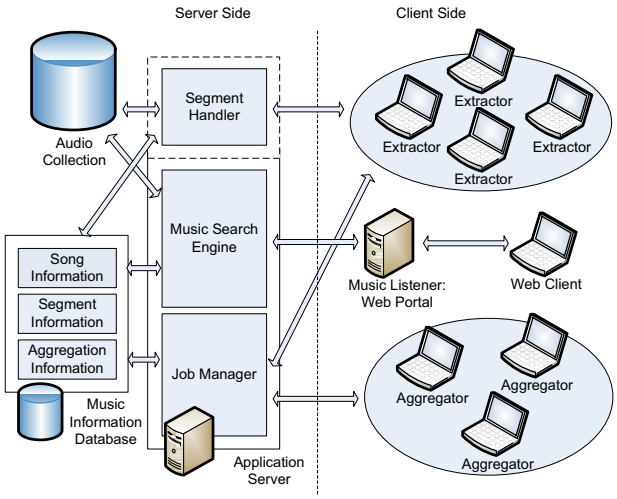

Figure 1. System architecture

3.1 iSoundMW Overview

The iSoundMW system has a client-server architecture, shown in Figure 1. The server side is composed of a central data repository and an application server that controls the extraction and similarity searches. The data repository stores all the audio collection in MP3 format and uniquely identifies each by a number. It also contains all the editorial information, e.g., the artist name, the album name, the band name, the song title, the year, the genre, and copyright license, and the physical information, e.g., the file size, the format, the bit rate, and the song length, that are stored in the music information database. Additionally, the Music Information Database holds all the extracted feature information.

The application server is composed of three distinct components. First, the *job manager* assigns extraction or aggregation tasks to the clients, collects the results, and prepares progress reports about each extraction process. Second, the *segment handler* splits the audio information into short term segments overlapping or non-overlapping of equal length, and makes them available to the clients they have been assigned. Future versions of the system will have multiple segment handlers, enabling multiple music collections to be analyzed without violating copyright for any of them. Third, the *music search engine* serves requests such as finding the most similar song to a given seed song with respect to a combination of features.

The client side is composed of three different types of clients. First, the *segment extractors* are receiving short term segments, and extracting their high-level features, e.g., the pitch, or the tempo. Second, the *segment aggregators* are performing computations based on the high-level features of the segments to produce other high-level features requiring features over different segments, e.g., the mood. Third, the *music listeners* perform similarity search queries on the extracted high-level features.

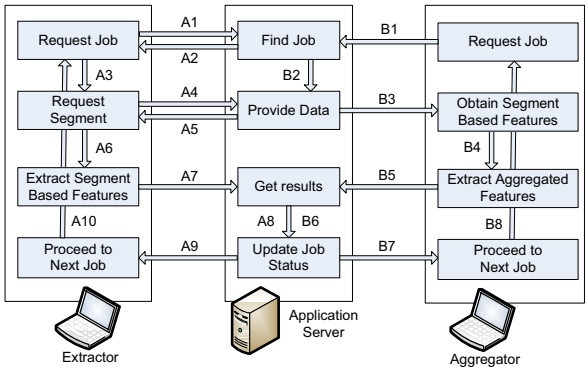

Figure 2. Two-step extraction process

3.2 Collecting the High-Level Features

The increasingly fast growth of music collections motivates the adoption of a distributed approach to perform feature extraction. Such approach, however, brings forward copyright issues, i.e., the copyrighted material in the music collection prevents the audio content to be freely distributed. We propose to address this issue by performing the feature extraction in a two-step process. In the proposed two-step extraction process, the full songs are not available to, or reconstructible by, the computers performing the extraction of the audio features.

Step 1, as illustrated by the A arrows in Figure 2, consists of dividing the audio information into short term segments of equal length and distributing them to the clients. Dividing the songs into segments offers the following advantages. First, it limits and normalizes the memory size and processor time needed on the client side to perform the feature extraction, as most extraction methods require resources proportional to the length of the audio source. Second, it avoids copyright issues as the audio segments are very short and distributing them falls under "fair use" since the segments are only temporally stored in memory on the client, and the full songs cannot be reconstructed by the clients.

Step 2, as illustrated by the B arrows in Figure 2, consists of sending the segment-based features gathered during step one to the clients. Using the features of step 1, the clients are computing high-level features for the whole piece of music. The computation of the aggregated features can be performed over the features obtained from multiple segments, as they are not subject to copyright issues. While having the segment based and the aggregated high-level features computed separately represents a potential overhead, it remains insignificant compared to the resources needed to perform each of the two feature extraction steps.

A *job* is the smallest atomic extraction or aggregation task that a remote client has to perform. Each job is uniquely identified for each extraction method and is composed of the following attributes: a song reference, a segment number, a starting point, an ending point, a lease time, a client reference, a counter of assignments, and a result holder. The assignment of jobs to clients is a critical part of the segment extraction process. Some randomness in the jobs assignment prevents clients to reconstruct the full song from the distributed segments as the segments assigned to a client will belong to different songs. However, since all the segments of a song have to be processed by step 1 before moving to step 2, assigning the segments randomly to clients delays the obtainment of results. Some locality constraints are therefore enforced in order to quickly acquire preliminary results and proceed to step 2.

In order to control which job should be assigned next, jobs are assigned in sequence. To avoid all the clients trying to obtain the same job, the assignment of a job is relaxed from being the minimal sequence number to being one of the lowest numbers in the sequence. Assigning jobs "nearly" in sequence still allows the application server to control which jobs should be prioritized for segment extraction and feature aggregation, e.g., when a similarity search is requested for a new song without any features extracted.

The current configuration of the system has shown its usability on a music collection of 41,446 songs composed of 2,283,595 non-overlapping 5 seconds segments. In terms of scalability, the job manager, supported by a PostgreSQL 8.3 database running on an Intel Core2 CPU 6700 @ 2.66GHz, 4GB of RAM under FreeBSD 7.0, was able to handle 1,766 job requests and submissions per second. Running on the same computer, the job manager and the segment handler were able to serve 87 clients per second; the bottleneck being the CPU consumption mostly due to the on-the-fly segmentation of the MP3 files. At this rate, an average bandwidth of 13,282 KB/s was consumed to transfer the segmented files of the data collection. Given that, by experience, the average processing of a segment by a modern desktop computer takes 5 seconds, the presented configuration would be able to handle over 400 clients. In a setup where the segment handler is run separately and not considering network limitations, the job manager could serve close to 9000 clients.

3.3 Similarity Search

Search among the collected features is a valuable tool to compare and evaluate extraction results. Similarity search raises two main challenges. First, similarities are of two types: similarities that can be computed rapidly on-the-fly and similarities that have to be pre-computed. Second, each similarity is tweaked dynamically with different user defined weight coefficients in order to adjust the final similarity value. In the following, we propose to find the 10 songs the most similar, with respect to a user defined weighted combination of features, to a given seed song.

Similarities that can be computed on the fly are stored in a single table, *abssim*: ("songID", "abs1", "abs2", ...), where

Abssim Table

SongID	Tempo	Pitch
1	1	0
2	2	3
3	3	2
...	...	...

Relsim Table

SeedID	SongID	Timbre
1	1	0
1	2	6
1	3	2
...	...	...
2	1	6
2	2	0
...	...	...

Figure 3. The absolute and relative similarity tables

the songID is the primary key identifying the song and abs1, abs2, ..., are the different attributes. The table is composed of 41,446 songs and 18 attributes, such as the tempo, the motion, the articulation, the pitch, and the harmonic. Similarities that cannot be computed on the fly have to be precomputed and stored for each pair of songs as some similarities are not symmetric and do not fulfill the triangular inequality. They are stored in a second table, *relsim*: ("seedID","songID","rel1"), where the "seedID" and "songID" refers to a pair of songs and "rel1" is the similarities value between the two songs. The relsim table has 1.7 billion rows and a timbre attribute. The abssim and relsim tables are illustrated in Figure 3.

The distance function, the similarity search is based on, is as follows:

$$\begin{aligned}\text{dist}(x,y) = a \times |\ \text{pitch}(x,y)| + b \times |\ \text{tempo}(x,y)| \\ + c \times |\ \text{timbre}(x,y)| + \ldots\end{aligned}$$

where x and y are two songs, tempo, pitch, and timbre are the computed differences between x and y for each feature, and a, b, and c are their respective coefficients. The pitch difference, like the tempo, can be computed on the fly: $\text{pitch}(x,y) = x.\text{pitch} - y.\text{pitch}$. The timbre difference is pre-computed and requires a lookup in the *relsim* table.

Assume the following query: find the 10 songs with the lowest distance from a given seed song. First, we compute the difference between the values of the seed song and the values of all the other songs in the table "abssim". This requires a random disk access using an index on the "songID" to read the values of the seed song, followed by a sequential scan to compute on the fly all the similarity values from the "abssim" table. Second, using an index on the "seedID", we select all the pre-computed similarities that correspond to the query songs. The song pairs with an identical "seedID" are contiguous on disk to minimize the number of disk accesses. Third, the 41,446 similarities from both resulting sets have to be joined. Fourth, the final distance function is computed and the 10 closest songs are returned.

Table 1 shows the average query processing time of the most costly operations involved in queries run on the MW described in Section 3. The performance is acceptable; most of the query processing time is cause by the join (20%) and the ordering (75%) operations. Hash joins have shown slightly better performance than, merge sort and nested loop joins. Merge joins should provide the performance results if the sort operation can be avoided by respecting the data organization on disk.

Hash join	123 ms
Merge join	141 ms
Nested loop	163 ms
Top K	405 ms
Total runtime	575 ms

Table 1. Time Cost of Similarity Search

3.4 Similarity Search within a Range

The query cost is mainly due to the join and the Top-K operations over the whole music set. We introduce similarity searches within ranges to reduce the size of the set on which the join and the Top-K are performed, thus decreasing the search time. We propose to search, within a user specified range for each feature, for the most similar songs to a given seed song.

The abssim table is used for each similarity search. Its small size allows sequential scans to be performed fast. Therefore, filtering out the values outside the query range effectively reduces the search time. Similarly, performing a filtering of the selected rows from relsim contributes to reducing the search time. However, several additional improvements can be made, they are illustrated in Figure 4.

First, a partial index on the seedID for the similarity values that are below a given threshold can be created. If the partial index has a good selectivity, speed is gained by trading the sequential scan for a few random disk access. This is of critical importance as the database grows larger. Using a threshold with a too high selectivity reduces the chances of the partial index being used.

Second, to further improve the search, one can create arrays containing pairs of songID and similarity value for each seedID. Arrays offer the following advantages. As the par-

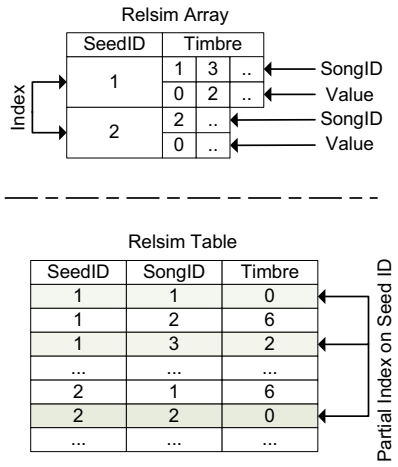

Figure 4. Range queries with arrays or a partial index

	Table	Array	Table + PI
Query time (ms)	130	15	54
Query time (%)	100	12	42
Size (MB)	70000	71550	70200
Size (%)	100	112	100

Table 2. Cost of Similarity Search within a Range

tial index, only the similar values are accessed, thus greatly reducing the cost of filtering. The similar values are clustered on disk for each seed song, similarity values are stored in the array in ascending order, and the array itself is compressed, therefore allowing a complete array to be retrieved in a single disk access. Accessing similarity values is done in an efficient way: one index lookup for locating the array, and one disk access to read the pairs of songID and value.

Table 2 presents the costs in search time and in disk space of each approach for the same set of constraints. As expected, specifying ranges for the similarity search significantly improves the response time compared to searching in the complete music collection; a simple filtering can improve the response time by a factor of 4. Using a partial index or arrays, further decreases the search time but come with a storage cost [8]. The search time improvements are dependent on the selected range as well as the selectivity of the threshold chosen for both the array and the partial index.

Similarity searches using the partial index are slower than arrays due to the random disk accesses that are required to read the values. However the partial index offers two major advantages. First, it does not require any backstage processing; the partial index is updated as new data is entered in the relsim table. Updating the arrays requires some additional processing that could be performed automatically with triggers. Second, the choice of using the partial index or not is delegated to the query planner, all queries are treated transparently regardless of the range chosen, i.e., no extra manipulation is needed to handle the array organization of the data when a threshold is reached.

4 THE ISOUNDMW

A web interface is connected to the iSoundMW. It offers a visual understanding of the functioning of the system and gives an opportunity to observe the information extracted from different songs and compare the result with the music being played simultaneously.

The setup is as follows. The music collection has a size of 41,446 songs in MP3 format, segments are non-overlapping and have a 5 seconds length. The Music Information Database is partially loaded with some segments information and some aggregated feature information, but some segments still have to be extracted and aggregated. Some clients are connected to the application server and are processing some segment extraction and feature aggregation jobs. If all the segments have not been extracted, the corresponding jobs will be pri-

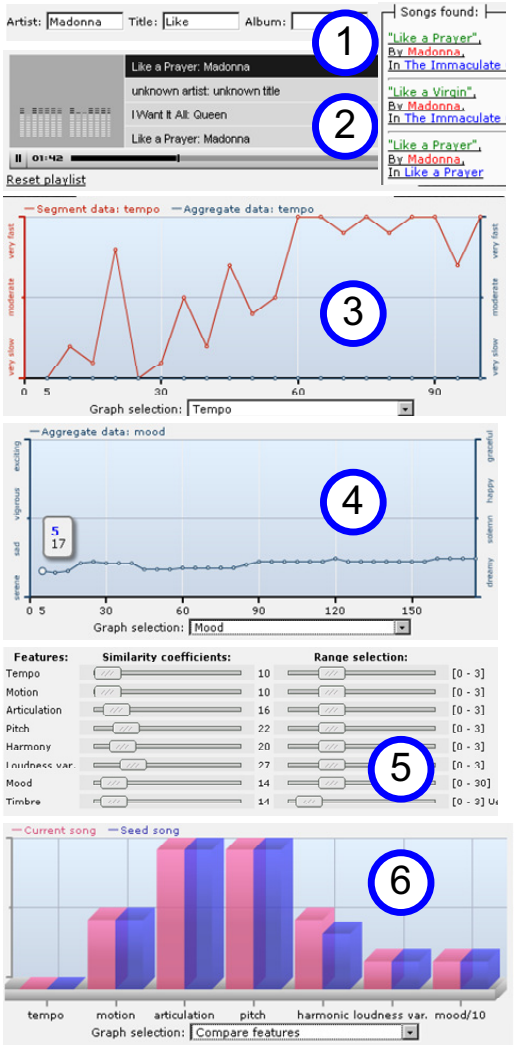

Figure 5. Web interface

oritized. to ensure that the extracted features are available. The user interface, as shown in Figure 5, consists of a music player and a graph showing the content of the extraction as the music is being played.

The main steps of a short scenario are presented below.
Step 1: A user searches for a famous song and provides its title, artist name, or album name, e.g., the user enters "Madonna" for the artist and "Like" for the title. The system retrieves a list of maximum 20 candidates present in the database based on the ID3 tags of the MP3 in the music collection, 3 songs in this scenario. The song "Like a Prayer" is listed twice, as it belongs to two different albums. The user selects one of the two "Like a Prayer" songs.
Step 2: The system searches for the 10 most similar songs to the song selected, places them in the playlist, and starts playing the songs. The most similar song to the song selected is generally the song itself, therefore the song appears first in the generated playlist. At this stage, the search is based on

default coefficients and one the complete database. The second version of the song "Like a Prayer" appears further in the generated playlist.

Step 3: As the song is being played, the tempo analysis based on the extracted segments and the aggregated features is displayed in the graph. For each 5 seconds (corresponding to a segment length), new points are placed on the graph. If the segments have not been extracted, a request is sent to the application server to prioritize the corresponding jobs. The graph updates as new extraction results are arriving from the extraction and aggregation clients.

Step 4: Any extracted features given on an absolute scale can be displayed on the graph, e.g., the user has selected to display the mood features.

Step 5: Moving to the playlist configuration panel, the user can select, for each of the extracted features, the weight to be used as a coefficient for the similarity search, e.g., we choose to put a high coefficient on the pitch and the mood. Once the tuning of the weights is done, when the user selects a new song, the system searches for the songs that, with the given coefficients, are the most similar to the song currently being played. Additionally, the user can select a range of values in which the search should be performed.

Step 6: When similar songs are being played, the user can select to compare the currently played song with the song originally selected to generate the playlist, e.g., the main difference between the two versions of "Like a Prayer" rely in the harmonic feature.

5 CONCLUSION AND FUTURE WORK

The automatic extraction of high-level features over a large music collection suffers from two main scalability issues: high computation needs to extract the features, and copyright restrictions limiting the distribution of the audio content. This paper introduces the iSoundMW, a framework for extracting high-level audio features that addresses these scalability issues by decomposing the extraction into a two-step process. The iSoundMW has successfully demonstrated its ability to efficiently extract high-level features on a music collection of 41,446 songs. Furthermore, the iSoundMW proves to be efficient and flexible for performing similarity searches using the extracted features. This is done by allowing users to constrain the search within a range and specify a weighted combination of high-level features. To further optimize the search, three different approaches are compared in terms of query time and storage. The threshold for building the partial index and arrays are decisive parameters to obtain good search performance.

Future work encompasses integrating different similarity functions in the features search, providing comparison between them, and enabling user feedback and its reuse for user specific recommendation.

6 ACKNOWLEDGMENTS

The authors would like to thank Jesper Højvang Jensen for participating in the distributed extraction and providing timbre similarities between songs. This work was supported by the Danish Research Council for Technology and Production, through the framework project "*Intelligent Sound*" [2] (STVF No. 26-04-0092).

7 REFERENCES

[1] The Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/.

[2] D. P. Anderson. BOINC: a system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[4] S. Bray and G. Tzanetakis. Distributed audio feature extraction for music. In *Proc. of ISMIR*, 2005.

[5] B. Y. Chua. *Automatic Extraction of Perceptual Features and Categorization of Music Emotional Expression from Polyphonic Music Audio Signals*. PhD thesis, Monash University, Victoria, Australia, 2007. http://www.cs.aau.dk/∼abychua.

[6] B. Y. Chua and G. Lu. Improved perceptual tempo detection of music. In *Proc. of MMM*, 2005.

[7] B. Y. Chua and G. Lu. Perceptual rhythm determination of music signal for emotion-based classification. In *Proc. of MMM*, 2006.

[8] F. Deliège and T. B. Pedersen. Using fuzzy song sets in Music Warehouses. In *Proc. of ISMIR*, 2007.

[9] J. S. Downie. The music information retrieval evaluation exchange (MIREX). *D-Lib Magazine*, 12(12):795–825, December 2006.

[10] J. S. Downie and M. Nelson. Evaluation of a simple and effective music information retrieval method. In *Proc. of ACM SIGIR*, 2000.

[11] A. F. Ehmann, J. S. Downie, and M. C. Jones. The music information retrieval evaluation exchange "Do-It-Yourself" Web Service. In *Proc. of ISMIR*, 2007.

[12] K.-S. Goh, B. Li, and E. Chang. DynDex: a dynamic and non-metric space indexer. In *Proc. of ACM Multimedia*, 2002.

[13] H. Jensen, D. P. W. Ellis, M. G. Christensen, and S. H. Jensen. Evaluation of distance measures between Gaussian mixture models of MFCCs. In *Proc. of ISMIR*, 2007.

[14] B. Logan and A. Salomon. A music similarity function based on signal analysis. In *Proc. of ICME*, 2001.

[15] C. McKay, D. McEnnis, and I. Fujinaga. Overview of OMEN. In *Proc. of ISMIR*, 2006.

[16] T. Pohle, E. Pampalk, and W. G. Generating similarity-based playlists using traveling salesman algorithms. In *Proc. of DAFx*, 2005.

[17] M. M. Ruxanda and C. S. Jensen. Efficient similarity retrieval in music databases. In *Proc. of COMAD*, 2006.

[2] http://www.intelligentsound.org