FUZZY SONG SETS FOR MUSIC WAREHOUSES

François Deliège and Torben Bach Pedersen
Aalborg University
Department of Computer Science

ABSTRACT

The emergence of music recommendation systems calls for the development of new data management technologies able to query vast music collections. In this paper, we define fuzzy song sets and an algebra to manipulate them. We present a *music warehouse* prototype able to perform efficient nearest neighbor searches in an arbitrary song similarity space. Using fuzzy song sets, the music warehouse offers a practical solution to the all musical data management scenarios provided: *song comparisons*, *user musical preferences* and *user feedback*. We investigate three practical approaches to tackle the storage issues of fuzzy song sets: *tables*, *arrays* and *bitmaps*. Finally, we confront theoretical estimates to concrete implementation results and prove that, from a storage perspective, arrays and bitmaps are both effective data structure solutions.

1 INTRODUCTION

Music recommendation systems have recently gained a tremendous popularity. Music lovers discover new ways of searching and sharing their favorite music. However, at such growing speed, the database element of any recommendation systems will soon become a bottleneck. Hence, appropriate musical data management tools are needed. Music Warehouses (MWs) are dedicated data warehouses optimized for the storage and analysis of music content. They are currently developed to respond to this is call.

The contributions of this paper are threefold. First, motivated by a case study [2], we propose three generic usage scenarios illustrating the current demands in musical data management. To answer these demands, we define fuzzy song sets and develop an algebra. Second, to demonstrate the usefulness of fuzzy song sets, a prototypical MW composed of two multidimensional cubes is presented. For each cube, concrete examples of queries inspired by the usage scenarios are provided. Fuzzy song sets prove to be an adequate data structure to manipulate musical information. Third, we discuss three solutions for storing fuzzy song sets and we construct theoretical estimates. A practical implementation shows that the structure overhead represents a major part of the storage consumption and that two solutions are viable for very large music collections.

A lot of attention was drawn to enable music lovers to explore individual music collections [6, 7]. Within this context, several research projects have been conducted in order to pursue a suitable similarity measure for music [8, 9]. A music data model, an algebra and a query language are introduced by Wang et al. [10]. However, the model lacks an adequate framework to perform similarity searches. Jensen et al. address this issue and offer a model that supports dimension hierarchies [5]. This paper tackles the storage issues when the scalability does not remain limited to a few hundred thousands songs.

Nearest neighbor searches are a popular topic in the database community for their usage in content based retrieval and similarity searches. Work on both high and low dimensional spaces can be found in the literature. However, existing indexing techniques do not apply to high dimensional musical features due to the subjective nature of musical perception, i.e., similarities do not form a metric. Work on indexes for non-metric space is presented in the literature [12]. Though the similarity function is non-metric, it remains confined in a pair of lower and upper bounds specifically constructed. MWs, however, should not be tightened to any similarity function.

The use of bitmaps in multidimensional databases is frequent. Different compression schemes exist to reduce the storage consumption of bitmaps. The Word Aligned Hybrid [11], WAH, and the Byte aligned Bitmap Compression [1], BBC, are two very common compression algorithms. BBC offers a very good compression ratio and performs bitwise logical operations efficiently. WAH performs bitwise operations much faster than BBC but consumes more storage space.

The remainder of this paper is organized as follows. Section 2 presents three search for information scenarios that could be treated by music recommendation systems. We proceed in Section 3 by defining fuzzy song sets and an algebra. In Section 4, two prototypical multidimensional cubes are presented and use of the algebra is illustrated through queries examples. Storage solutions are discussed in Section 5 and implementation results are shown in Section 6. Finally, we conclude in Section 7.

2 USAGE SCENARIO

Three examples of data obtained from music recommendations system are presented below.

The User Feedback The user's opinion about the previously suggested songs is a valuable piece of information. For each song played, the user can grade if the suggestion was wise based on the criteria provided, referred to as the query context. The query context can be the artist similarity, the genre similarity, the beat similarity, or any other similarity measure available to the system to perform a selection. The grading reflects if a proposed song was relevant in the given query context. For example, it is possible to retrieve the list of songs John liked when he asked for a list of rock songs or the ten songs Alice liked the most when she asked for songs similar to "U2: Where the streets have no name".

Typically, the data obtained should contain: *(i)* a reference to the profile of a registered user in the system, *(ii)* a reference to a query context provided by the user, and *(iii)* a list of songs and marks so that for each song proposed, the user can grade how much he liked a particular song being part of the proposition. Grades are given on a per song basis, they reflect if the user believes the song deserved its place among the suggested list of songs: strongly disagrees, neutral, likes, and loves. While the grade must not be a numerical value, we assume in the rest of the article that a mapping function to $[0, 1]$ has to be provided. When a user believes a song definitely deserves its place in the list, a high mark should be given.

The User Preferences Some songs should never be proposed to the user independently of the query context. On the contrary, some songs should be proposed more often as they are marked as the user's favorites. Therefore, a user should be able to grade any song on a *fan-scale* ranging from "I love it" to "I hate it" depending if he likes the song or not. For example, the music recommendation system database should be able to retrieve the list of songs Maria likes, and the songs she hates the most.

The User Musical Preferences contains two different pieces of information: (i) a reference to a user registered, and (ii) a list of songs associated with their respective grades on the fan-scale. If Rico hates a song, a low value should be used; if he loves it, a value close to 1 should be used instead. Musical profiles modify the frequency a given song appears in a *music recommendation list*.

The Songs Comparisons Finally, the music recommendation system should be able to compare any pair of songs. For each pair of songs, the system is able to provide a similarity value with respect to a given aspect of the song. The similarity values should indicate if two songs are "very different", "different", "somewhat similar", or "very similar" from the perspective of an given aspect of the song. For example, the song "We will rock you" by Queen is "very different" from the song "Twinkle, twinkle little star" with respect to their *genre similarity aspect*.

To compare songs, three pieces of information are necessary: *(i)* a reference to the first song of the pair being compared, *(ii)* a reference to the second song of the pair, *(iii)* a reference to the definition of a similarity function that maps to any pair of songs to a similarity value, and *(iv)* the similarity value reflecting how similar the two songs are. If two songs are very different, a value close to 0 should be used, if they are very similar, a value close to 1 should be used instead.

To keep the scenario as generic as possible, very few assumptions are made about the properties of the functions used to compute the similarity values. In particular, the similarity functions do not have to fulfill the mathematical properties of a metric, e.g., the non-negativity, the identity of indiscernibles, the triangular inequality, and the symmetry properties.

3 AN ALGEBRA FOR FUZZY SONG SETS

In this section, we introduce song sets as well as operators and functions to manipulate them.

Let X be the set of all songs. Then, a fuzzy song set, A, is a fuzzy set defined over X such that

$$A = \{\mu_A(x)/x : x \in X, \mu_A(x) \in [0, 1]\} \tag{1}$$

and is defined as a set of pairs $\mu_A(x)/x$, where x is a song, $\mu_A(x)$, referred to as the membership degree of x, is a real number belonging to $[0, 1]$, and / denotes the association of the two values as commonly expressed in the fuzzy logic literature [3]. $\mu_A(x) = 0$ when song x does not belong to A, and $\mu_A(x) = 1$ when x completely belongs to A.

3.1 Operators

The following operators are classically used in order to manipulate song sets. They form a closed algebra.

equality: Let A and B be two fuzzy song sets. A is equal to B iff for all song the membership degree of a song in A is equal to the membership degree of the same song in B.

$$A = B \Leftrightarrow \forall x \in X, \mu_A(x) = \mu_B(x) \tag{2}$$

subset: Let A and B be two fuzzy song sets. A is included in B iff for all song, the membership degree a song in A is lower than the membership degree of the same song in B.

$$A \subseteq B \Leftrightarrow \forall x \in X, \mu_A(x) \leq \mu_B(x) \tag{3}$$

Note that the empty fuzzy song set defined with the null membership function, i.e., $\forall x \in X, \mu(x) = 0$, is a subset of all fuzzy sets.

union: Let A and B be two fuzzy song sets over X. The union of A and B is a fuzzy song set with, for each song, a membership degree equal to the maximum membership degree associated to that song in A and B.

$$\begin{gathered} A \cup B = \{\mu_{(A \cup B)}(x)/x\} \\ \mu_{(A \cup B)}(x) = \max(\mu_A(x), \mu_B(x)) \end{gathered} \tag{4}$$

intersection: Let A and B be two fuzzy sets over X. The intersection of A and B is a fuzzy song set with, for each song, a membership degree equal to the minimum membership degree associated to that song in A and B.

$$\begin{gathered} A \cap B = \{\mu_{(A \cap B)}(x)/x\} \\ \mu_{(A \cap B)}(x) = \min(\mu_A(x), \mu_B(x)) \end{gathered} \tag{5}$$

negation: Let A be a fuzzy sets over X. The negation of A is a fuzzy song set with the membership degree of each song equal to its symmetric value on the interval $[0, 1]$.

$$\neg A = \{1 - \mu_A(x)/x\} \tag{6}$$

The following new operators are introduced specifically to manipulate song sets.

reduction: Let A be a fuzzy set over X. The reduction of A is a subset of A such that membership degrees smaller than α are set to 0.

$$\text{Reduce}_\alpha(A) = \{\mu_{A_{\overline{\alpha}}}(x)/x\}$$
$$\mu_{A_{\overline{\alpha}}}(x) = \begin{cases} \mu_A(x) & \text{if } \mu_A(x) \geq \alpha, \\ 0 & \text{if } \mu_A(x) < \alpha \end{cases} \tag{7}$$

The reduction operator changes the membership degree of songs below a given threshold to 0. It allows the construction of more complex operators that allow the reducing the membership degree granularity over ranges of membership degrees.

Top_k: Let A be a fuzzy set over X. The Top_k subset of A is a fuzzy song with the membership degree of all elements not having the k highest membership degree set to 0 and the membership degree of the k highest elements of A set to their respective membership degree in A.

$$\begin{gathered} \text{Top}_k(A) = \{\mu_{\widehat{A^k}}(x_i)/x_i| \\ \forall x_i, x_j \in X, 1 \leq i < j, \mu_A(x_i) \geq \mu_A(x_j)\} \\ \mu_{\widehat{A^k}}(x_i) = \begin{cases} \mu_A(x_i) & \text{if } i \leq k, \\ 0 & \text{otherwise} \end{cases} \end{gathered} \tag{8}$$

Note that the Top_k subset of A is not unique, e.g., when all elements have an identical membership degree. The Top_k operator returns a fuzzy song set with all membership degrees set to zero except for k elements with the highest membership degrees that remain unchanged. Top_k is a cornerstone for the development of complex operators based on relative ordering of the membership degrees.

average: Let $A_1, \ldots, A_i$ be i fuzzy song sets. The average of $A_1, \ldots, A_i$ is a fuzzy song set that assigns to each song a membership degree equal to the arithmetic mean of the membership degrees of that song in the given sets.

$$\begin{gathered} \text{Avg}(A_1, \ldots, A_i) = \{\mu_{avg(A_1,\ldots,A_i)}(x)/x\} \\ \mu_{Avg(A_1,\ldots,A_i)}(x) = \frac{\sum_{j=1}^{i} \mu_{A_j}(x)}{i} \end{gathered} \tag{9}$$

The average operator in fuzzy sets is the pendant of the common average operator and is very useful to aggregate data, a very common operation in data warehousing in order to gain some overview over large datasets.

3.2 Defuzification Functions

The following functions are defined on song sets. They extract information from the song sets to real values or crisp sets.

support: The support of A is the crisp subset of X that includes all the elements having a non-zero membership degree in A.

$$\text{Support}(A) = \{x \in X : \mu_A(x) > 0\} \tag{10}$$

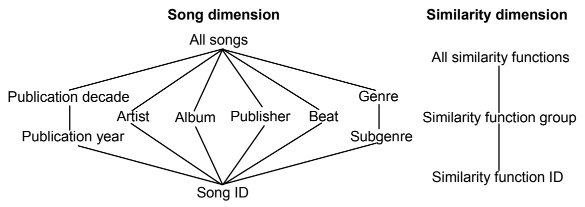


Figure 1. Closest Songs Cube Dimensions

cardinality: The cardinality of A is the sum of the membership degrees of all its elements.

$$\#A = \sum_{x \in X} \mu_A(x) \tag{11}$$

4 THE MUSIC WAREHOUSE CUBES

In this section, we present two data cubes to store the information presented in the scenarios. For each cube, the fuzzy song sets are used conformingly to Section 3.

4.1 The Closest Songs Cube

The closest songs cube provides a set of the closest songs from a *seed song* with respect to a similarity function. For each seed song and for each similarity function, the closest songs are stored using a fuzzy song set. The notion of similarity is represented by the fuzzy song set membership degree. The closest songs take a high membership degree while the farthest songs have a low membership degree.

The data cube has a song dimension and a similarity dimension. Both dimensions have a hierarchy as illustrated in Figure 1. The cube is composed of references to each dimension and a fuzzy song set. Typical queries are making use of the intersection, union, and reduction operators. The queries can be performed on the song seeds using pieces of information such as the artist or the creation year. Closest Songs Cube usage examples based on data in Table 1 are presented below.

In the following examples, we assume that fuzzy song sets and the algebra have been implemented in an abstract SQL datatype, called FSSET, using object-relational extensibility functionality like found in PostgreSQL [4].

Example 1 The *3 closest songs* to song: *"One" by U2* with respect the *beat or the rock* similarity:

```
SELECT SUPPORT(TOP 3(UNION(closest songs)))
FROM song AS a, similarity AS b, closest songs AS c
WHERE a.title = 'One' AND a.artist = 'U2'
AND (b.sim func = 'beat' OR b.sim func = 'rock')
AND b.sim id = c.sim id
AND a.song id = c.song id;
```

The use of hierarchies facilitates the use of aggregate functions. The following example illustrates the usage of the song dimension hierarchy to find songs similar to the ones sung by a given artist.

Example 2 The songs that are *very similar* with respect to their *beat* to the songs sung by *U2*:

```
SELECT SUPPORT(REDUCE 0.9(AVG(closest songs)))
FROM song AS a, similarity AS b, closest songs AS c
WHERE a.artist = 'U2' AND b.sim func = 'beat'
AND b.sim id = c.sim id AND a.song id = c.song id;
```

Song dimension

song id	title	artist	Jazz	Rock	Beat
1	We will rock you	Queen	Low	High	Medium
2	One	U2	Low	Medium	Medium
3	Hips Don't Lie	Shakira	Low	Low	High

Similarity function dimension

sim id	sim func	sim group
1	Rock	Acoustic
2	Jazz	Acoustic
3	Beat	Acoustic
4	Artist	Editorial

Closest songs fact

song id	sim id	closest songs
1	1	{ 1/1; 0.5/2; 0/3 }
2	1	{ 1/2; 0.3/1; 0.2/3 }
3	1	{ 1/3; 0.6/2; 0.4/1 }
1	2	{ 1/1; 0.2/2; 0.1/3 }
2	2	{ 1/2; 0.9/1; 0.2/3 }
3	2	{ 1/3; 0.8/2; 0.7/1 }

Table 1. Song Comparisons Cube Data

User dimension

user	country	age	favorite songs
John	USA	52	{ 0.8/1; 0.6/2; 0.3/3 }
Alice	Spain	41	{ 0.9/2; 0.5/1; 0.3/3 }
Maria	Greece	28	{ 0.6/1; 0.3/2; 0.1/3 }
Bob	Denmark	22	{ 0.1/1; 0.7/2; 0.7/3 }

Query dimension

query id	query
1	return some rock music
2	return some traditional music
3	return some latin american music

User Feedback fact

user	query id	feedback
John	1	{ 1/1; 0.5/2; 0/3 }
John	2	{ 1/2; 0.3/1; 0.2/3 }
Alice	1	{ 1/3; 0.6/2; 0.4/1 }
Alice	3	{ 1/1; 0.2/2; 0.1/3 }
Maria	1	{ 1/2; 0.9/1; 0.2/3 }
Bob	2	{ 1/3; 0.8/2; 0.7/1 }

Table 2. Feedback Cube

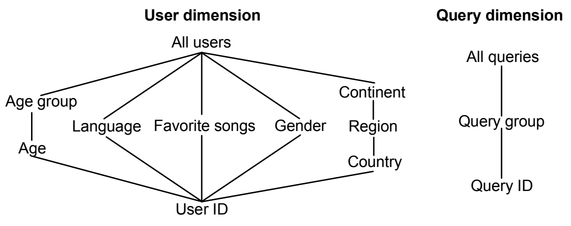


Figure 2. User Feedback Cube Dimensions

4.2 User Feedback Cube

The User Feedback Cube gathers relevance statistics about the songs proposed to users by the music recommendation system. As illustrated by Figure 2, the user feedback cube is composed of the user dimension and the query dimension. For each user and query, the user feedback is stored. The feedback given for a particular played song is stored as a membership degree representing how the proposed song is relevant in the context of the query. A very low membership degree is given when the users believes the song should not have been proposed. The Feedback and the Favorite Songs attribute are both defined using the FS-SET abstract datatype.

As with the Closest Songs Cube, it is possible to use aggregate functions along the dimension hierarchies.

Example 3 The 10 songs users living in Denmark liked the most when they asked for traditional music:

```
SELECT SUPPORT(TOP 10(AVG(c.feedback))
FROM user AS a, query AS b, user feedback AS c
WHERE a.country = 'Denmark' AND a.user = c.user
AND b.query = 'return some traditional music'
AND b.query id = c.query id;
```

Furthermore, it is possible to perform aggregation along fuzzy song sets defined in a dimension such as the favorite songs attribute in the user dimension.

Example 4 The 10 songs that users who dislike Shakira like the most when they ask for "Latin american music":

```
SELECT SUPPORT(TOP 10(AVG(c.feedback)))
FROM user AS a, query AS b, user feedback AS c
WHERE ARRAY(REDUCE 0.6(NEG(a.favorite songs))) &&
ARRAY(SELECT song id FROM song WHERE artist = 'Shakira')
AND b.query = 'return some latin american music'
AND b.query id = c.query id;
```

5 STORAGE

In this section, three different storage options for representing song sets in the MW are presented: tables, arrays, and bitmaps. To illustrate the discussion, a prototypical MW where songs are uniquely identified using 24 bits and membership degrees are stored using 8 bits. The proposed MW can reach a size of over 16 million songs and use up to 256 different membership degrees.

5.1 Table

The first solution is to represent the fuzzy song sets using a table with three columns: *(seed song, song, membership degree)*. Only the k closest songs are physically stored in the table. The selection is performed using the Top_k operator. Let s be the size of the seed song set, e the size of the song set, m the size of the set of all the values the membership degree can take, and f the number of fuzzy song sets associated to a given song seed. The size of the payload, i.e., the size of the data when not considering the overhead due to the DBMS, denoted p, can be calculated as follows.

$$p = s \cdot k \cdot (\log_2 s + f \cdot (\log_2 e + \log_2 m)) \text{ b} \qquad (12)$$

where $\log_2 s$, $\log_2 e$, $\log_2 m$ are the minimum number of bits required to store respectively a seed song, a song, and a membership degree. Using a default membership degree and a Top_{1000} operator reduces the theoretical size of the table to 109 GB when each of the 16 million song seeds is associated to one fuzzy song set attribute.

5.2 Array

A second approach is to use one dimensional arrays containing the songs and their associated membership degree for representing song sets. The data is stored in a table with two columns: *(seed song, array)*. As with tables, only the k closest songs should be physically stored. The size of the payload can be calculated as follows.

$$p = s \cdot (\log_2 s + f \cdot k \cdot (\log_2 e + \log_2 m))\ \text{b} \tag{13}$$

So, when storing the 1000 closest songs with respect to one attribute, the size of the payload is reduced to 63 GB. However, since the probability of having no songs for a particular membership degree is small, ordering the fuzzy song set by membership degrees allows membership degrees to be stored using one bit relatively to each other: a bit set means to move to the next lower membership degree, a bit unset means to keep the same membership degree. In the unlikely case of a gap in the sequence of membership degrees, a dummy element, referred to as the empty element, is used to jump to the next membership degree. For large gaps, successive empty elements are used. The compression ratio, r, obtained is as follows.

$$\text{r} = \frac{k \cdot (\log_2 m + \log_2 e)}{(k + x)(\log_2 e + 1)} \tag{14}$$

In order to be efficient, i.e., $r > 1$, the number of empty elements in the data set has to remain limited.

$$x < k \cdot \frac{\log_2 m - 1}{\log_2 e + 1} \tag{15}$$

Using a granularity of 8 bits for the membership degree, the compression is effective for $x < 280$. This is always verified as 256 different membership degrees exist. The compression ratio in the best and worst case scenarios are:

$$\begin{aligned} r^- &= k \cdot \frac{(\log_2 m + \log_2 e)}{(k + m - 1) \cdot (\log_2 e + 1)} \\ r^+ &= \frac{\log_2 m + \log_2 e}{\log_2 e + 1} \end{aligned} \tag{16}$$

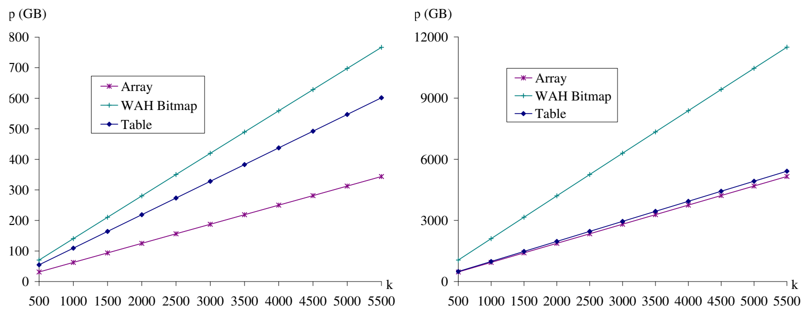
In our example, using 8 bits for storing the membership degree, the pessimistic ratio is 1.10 while the optimistic compression ratio reaches 1.28.

5.3 Bitmap

A third option is to use bitmaps to represent fuzzy song sets. Each bit indicates if a song belongs or not to the set of the k closest songs to a given song seed.

$$p = s \cdot (\log_2 s + f \cdot e)\ \text{b} \tag{17}$$

The bitmap size can be dramatically reduced using compression algorithms. The WAH compression offers good compression on sparse bitmaps while preserving query performance.



(a) 1 attribute per seed (b) 15 attributes per seed

Figure 3. Estimates Comparison

In the worst bit distribution, i.e., a random bitmap, the WAH algorithm reduces the size of the bitmap as follows.

$$p_{\text{WAH}}(n, d, w) \approx \frac{w \cdot n}{w - 1}\left(1 - (1 - d)^{2w-2} - d^{2w-2}\right)\ \text{b} \tag{18}$$

where n is the size of the bitmap in bits, d is the bit density, i.e., the fraction of bits set, and w is the word length, 32 on most computers. Using the Top$_k$ operator, the bit density is $d = \frac{k}{e}$. On a fuzzy song set of 2^{24} songs where only 1000 closest songs are physically stored, $n = 2^{24}$ b, and $d = \frac{1000}{2^{24}}$. The size of each bitmap reaches 63883 b. To represent the membership degree, a bitmap is constructed for the membership degree each of the song could possibly take.

$$p \approx s \cdot \left(\log_2 e + f \cdot p_{\text{WAH}}(e \cdot m, \frac{k}{e \cdot m}, w)\right)\ \text{b} \tag{19}$$

The size of the compressed bitmap for each song seed is only slightly increased to 63999 b. Therefore, in an MW of 2^{24} song seeds with one fuzzy song set attribute, the size of the database reaches 140 GB.

5.4 Estimates Comparison

Figures 3(a) and 3(b) show the expected size for storing 1 and 15 Fuzzy Song Set Attribute (FSSA), respectively, for each of the 2^{24} song seeds and for different values of k. The linearity of the WAH bitmap curves is explained by considering $\frac{k}{n} << 1$ and applying a binomial decomposition, the payload can then be approximated by $p_{\text{WAH}} \approx 2 \cdot w \cdot k$.

For one FSSA per song seed, arrays consume half the storage space of bitmaps. This difference vanishes when the number of fuzzy song set attributes per song seed increases. With respect to the storage requirements, WAH bitmaps seem to be a poor choice when the number of FSSA per song seed increases.

6 IMPLEMENTATION

This section describes the Closest Song Cube fact table implementation under PostgreSQL 8.2, well-known for its scalability. Therefore, some parts of the following are DBMS dependent. As explained in Section 5, songs can

		1 att.	15 att.
Table	Payload est. (MB)	345	3115
	Overhead est. (MB)	1777	1783
	Total est. (MB)	2226	4898
	Real size (MB)	2779	21500
	B-tree index size(MB)	1557	1557
Array	Payload est. (MB)	197	2961
	Overhead est. (MB)	4	4
	Total est. (MB)	201	2965
	Real size (MB)	729	21504
	Real size + LZ (MB)	256	2975
	B-tree index size(MB)	2	2
WAH Bitmap	Payload est. (MB)	395	5923
	Overhead est. (MB)	4	4
	Total est. (MB)	399	5927
	Real size (MB)	514	7696
	Real size + LZ (MB)	202	2923
	B-tree index size(MB)	2	2

Table 3. Storage comparison

be uniquely identified using 24 bits. The dataset used for the implementation consists of 51574 songs that were selected from two different sources: first, the Intelligent Sound [1] database that has extracted the genre information of over 150000 music tracks, second, the 5 million songs of the Music Brainz [2] database. Each of the 51574 songs is described using 15 attributes that represent the genre. The expected overhead in PostgreSQL can be estimated to 32 bytes per row and 20 bytes per page, where each page has a fixed size of 8 KB [4]. Since tuples are not allowed to span over multiple pages, PostgreSQL uses secondary storage tables, referred to as TOAST tables, to store large field values. TOAST tables can use a Lempel-Ziv, briefly LZ, compression technique to reduce their size.

The implementation results are shown in Table 3. For one FSSA per song seed, arrays and WAH bitmaps appear to be a better solution than tables when comparing their real size. The results also show that the theoretical overhead estimates were wrong. In Figure 3(a), the theoretical estimates showed that the size of arrays should be half the size of WAH bitmaps. This is far from being the case. To the contrary, WAH bitmaps are smaller than arrays.

For 15 FSSA per song seed, arrays and WAH bitmaps are again a better solution than tables. Again, the expectations shown in Figure 3(b) were different. First, the overhead is considerable compared to the data size. This conforms to the theoretical result that tables are a bad choice since an considerable overhead per row exists. However, when comparing arrays versus bitmaps, the implementation and theoretical results are again contradictory. WAH bitmaps only take a third of the storage space of uncompressed arrays, due to the overhead of the array data structure. LZ compressed WAH bitmaps in a TOAST table are reduced to 2923 MB, while LZ compressed arrays are reduced to 2975 MB, thus making the two solutions comparable. However, using LZ compression on WAH bitmaps requires further investigations. On one hand it will reduce the number of IOs. On the other hand, it will slow down the bitwise operations and therefore might reduce the advantages of the WAH compression scheme.

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

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

7 CONCLUSION

MWs are the practical answer to the need for efficient content management tools over vast music collections. The approach chosen in this paper is to treat song similarities in a relative space where songs are defined by comparison to each others. We have defined fuzzy song sets and presented an algebra to manipulate them. We have demonstrated the usefulness of fuzzy song sets and their operators to handle various information management scenarios in the context of a MW for which we have created two multidimensional cubes. Furthermore, the practical implementations solutions from a theoretical perspective were discussed. Finally, confronting theoretical estimates to practical implementation makes clear that the DMBS data overhead represents a major part of the storage requirements. Further study should be conducted on the performance aspects and optimized operators should be implemented with respect to the data structure choices.

8 REFERENCES

[1] G. Antoshenkov, Byte aligned data compression. *United States Patent*, 5363098, 1994.

[2] F. Deliège and T. B. Pedersen. Music warehouses: Challenges for the next generation of music search engines. In *Proc. of LSAS*, 2006.

[3] J. Galindo, M. Piattini, and A. Urrutia. *Fuzzy Databases: Modeling, Design and Implementation*. Idea Group Pub, 2005.

[4] The PostgreSQL Global Development Group. PostgreSQL 8.2.0 documentation. At *http://www.postgresql.org/docs/manuals/*, 2006.

[5] C. A. Jensen, E. M. Mungure, T. B. Pedersen, and K. Sørensen. A data and query model for dynamic playlist generation. In *Proc. of IEEE-MDDM*, 2007.

[6] D. Lübbers. SoniXplorer: Combining visualization and auralization for content-based exploration of music collections. In *Proc. of ISMIR*, 2005.

[7] M. Mandel and D. Ellis. Song-level features and support vector machines for music classification. In *Proc. of ISMIR*, 2005.

[8] R. Neumayer, M. Dittenbach, and A. Rauber. PlaySOM and PocketSOMPlayer, alternative interfaces to large music collections. In *Proc. of ISMIR*, 2005.

[9] E. Pampalk. Speeding up music similarity. In *Proc. of MIREX*, 2005.

[10] C. Wang, J. Li, and S. Shi. A music data model and its application. In *Proc. of MMM*, 2004.

[11] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1), 2006.

[12] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proc. of ICDE*, 1998.