Effective Bitmap Indexing for Non-metric Similarities

Claus A. Jensen, Ester M. Mungure, Torben Bach Pedersen,
Kenneth Sørensen, and François Deliège

Department of Computer Science, Aalborg University

Abstract. An increasing number of applications include recommender systems that have to perform search in a non-metric similarity space, thus creating an increasing demand for efficient yet flexible indexing techniques to facilitate similarity search. This demand is further fueled by the growing volume of data available to recommender systems.

This paper addresses the demand in the specific domain of music recommendation. The paper presents the *Music On Demand framework* where music retrieval is performed in a continuous, stream-based fashion. Similarity measures between songs, which are computed on high-dimensional feature spaces, often do not obey the triangular inequality, meaning that existing indexing techniques for high-dimensional data are infeasible.

The most prominent contribution of the paper is the proposal of an indexing approach that is effective for non-metric similarities. This is achieved by using a number of bitmap indexes combined with effective bitmap compression techniques. Experiments show that the approach scales well.

1 Introduction

Recommender systems are becoming increasingly present in many applications. The growing volume of data available to these recommender systems has created a increasing demand for efficient yet flexible indexing techniques able to facilitate similarity search. However, in many emerging domains, the similarity spaces are non-metric, meaning that existing approaches are not feasible. Therefore, a major challenge is to develop an indexing technique for non-metric similarity spaces. This paper tackles this challenge in the specific context of music recommendation, but the approach can be applied to a wide variety of systems.

The current tendency in music distribution is that personal music collections are being replaced by the notion of *streaming music* from commercial on-line music providers. Here, the challenge is to support query functionalities on *vast music collections* where only limited or no prior knowledge about the content of the music collection is available. However, similarity measures between songs are most often complex formulas computed on high-dimensional feature spaces. These similarity measures, such as the Earth Mover's Distance, often behave "strangely", as they are non-metric. In particular, the triangular inequality often does not hold. This means that existing indexing techniques for high-dimensional data are infeasible. The aim of this paper is to propose an indexing solution that supports the "worst case", namely *non-metric* similarity measures, well.

P. García Bringas et al. (Eds.): DEXA 2010, Part I, LNCS 6261, pp. 137–151, 2010.

This paper introduces the *Music On Demand framework*, referred to as the MOD framework, and presents two major contributions. The first contribution is an indexing approach that is effective for non-metric similarity measures. The second major contribution is effective query processing techniques to perform similarity search. Our approach essentially relies on using bitmap indexes combined with effective bitmap compression techniques.

This approach ensures 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 by Madonna released this year. 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 using bit-wise operations.

Extensive experiments show that the proposed approach scales with respect to the data set size and the number of concurrent requests. Query performance, throughput, and storage requirements are presented on a prototypical version of the MODframework. For example, the prototype system running on a standard laptop is able to support 36,000 simultaneous users on a database of 100,000 songs. Comparing our implementation with an equivalent B-tree solution, we improve the query execution time by an order of magnitude on a music collection containing 100,000 songs.

The paper is organized as follows. Section 2 describes related work. Section 3 presents an informal description of a data and query model for dynamic playlist generation. In Section 4, we elaborate on the application of bitmaps followed by an examination of the associated query processing techniques in Section 5. In Section 6, we discuss and evaluate the experiments conducted using bitmap indexing. Finally, in Section 7 we conclude and present directions for future work.

2 Related Work

Within the field of Music Information Retrieval, much effort has been put into the task of enabling music lovers to explore individual music collections [12, 14]. Within this context, several research projects [17, 13] have been conducted in order to pursue a suitable similarity measure for music, for which purpose a feature representation of the musical content is required. 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 [2, 18]. However, due to the subjectiveness of musical perception, the similarity measure described disobey the triangular inequality and are thus said to be *non-metric*. Several non-metric similarity measures exist [3, 11, 21].

When considering indexing of high dimensional musical feature representations, existing indexing techniques such as, e.g., the M-grid [7] and the M-tree [6] can be applied. However, as discussed above, the *triangular inequality* property of the metric space typically cannot be obeyed for a similarity measure. Hence, as the M-tree and the M-grid, and many other high-dimensional indexing techniques, rely on the use of a metric space, they turn out to be insufficient. In contrast, the approach presented in this paper does not require the triangular inequality to hold.

To ensure efficient retrieval of read-mostly data, bitmap indexes are popular data structures for use in commercial data warehouse applications [9]. In addition, bitmap indexes are used with respect to bulky scientific data in order to represent static information. One approach is related to High-Energy Physics [22]. However, to our knowledge, bitmap indexing has so far not been applied within music retrieval.

Many different approaches exist to support accurate music recommendation. A first approach uses musical content to find similar songs, where immediate user interaction in terms of skipping behavior is used to restrict the music collection [19]. 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. A second approach maps the task of finding similar songs to the Traveling Salesman problem (TSP) [21]. A single circular playlist consisting of all tracks from the entire music collection is generated, and the ability to intervene in the construction of playlist is taken away from the listener. In contrast, the single song approach presented in this paper, ensures that the construction of a playlist may be influenced dynamically. Unlike the present paper, none of these two approaches provide an indexing approach capable of handling arbitrary similarity measures.

Finally, in most commercial media players such as Winamp™, 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 [20, 23]. 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.

3 Data and Query Model

We briefly present the underlying music data model and the associated query functionalities of the MOD framework. The full details can be found in another paper [8].

Initially, we introduce a *metadata dimension* in order to apply an abstraction to a hierarchical representation of the music metadata. As shown in Figure 1, the hierarchical ordering of the metadata is described as two posets (partially ordered sets). The first poset represents the hierarchical ordering of dimension levels and the second poset represents the hierarchical ordering of the dimension values.

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* [20]. 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. The metadata of music is composed of descriptive attributes such as artist, title, etc. The metadata attributes are presented as dimension values where a metadata item and the corresponding schema are defined. To ensure that the model supports querying with respect to traditional navigational methods as well as musical similarity, a *song* is defined in terms of

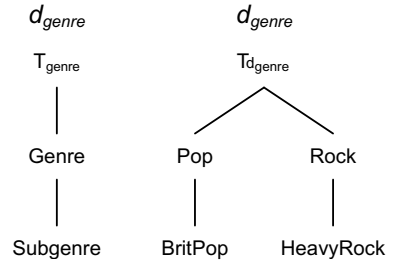
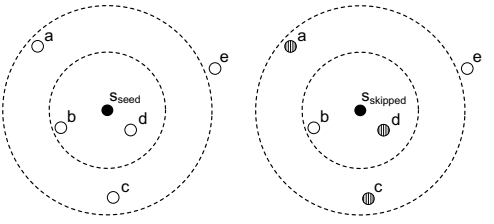


Fig. 1. Schema (left) and instance (right) for the metadata dimension d_{genre}



(a) Fetch similar song. (b) Fetch random song.

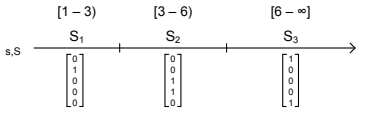
Fig. 2. Usage of the distance store

both tangible music metadata and musical content. In this context, metadata represents elements such as artist, genre, etc. and musical content corresponds to an arbitrary feature representation of the music. In addition, an arbitrary distance function $dist$ may be used to calculate the content-based similarity between two songs with respect to their associated feature representations. The distance function is allowed to be *non-metric* as long as the identity property is retained, i.e., $dist(x, x) = 0$.

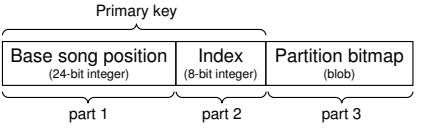
When considering the content-based similarity between songs, we introduce the *distance store* as an abstraction over an arbitrary distance function. A distance store is a complete partitioning of the distance domain, implying that no partitions are omitted and that no partitions should overlap. Thus, each partition constitutes a unique and non-overlapping distance interval.

The two retrieval operators *SimilarSong* and *RandomSong* constitute the main point of interacting with the framework. The purpose of *SimilarSong* is to retrieve a song similar to a given seed song, while in the same time avoiding that the retrieved songs resembles possible skipped songs. As the name indicates, the task of *RandomSong* is to retrieve a randomly chosen song from the music collection. In this connection the aspects of skipped songs also apply. Moreover, as the listener may choose to intervene in the construction of the playlist at any point in time, either of the operators only return a single song at a time. To fetch a similar song, using the *SimilarSong* operator, we are initially presented a specific seed song as shown in Figure 2(a). Assuming that each circle represents a partition of the distance store associated with song s_{seed}, we are to return a song from the innermost partition containing valid songs. In this context, a valid song is a song neither restricted nor skipped. As any song within the appropriate partitions are valid candidate songs, either song b or d may be returned as a song similar to s_{seed}. However, say that b is more similar to an already skipped song, d is the better candidate.

The distance store shown in Figure 2(b) constitutes a composite distance store representing all skipped songs as discussed above. When fetching a randomly chosen song, using the *RandomSong* operator, we initially pick a number of candidate songs among the valid songs in the music collection. In this case songs a, c and d are picked. As the candidate songs are chosen randomly within a vast music collection, chances are that even a small number of songs, e.g., 10, ensures retrieval of an acceptable song. Locating the positions of the candidate songs within the composite skip distance store we return the song most dissimilar to any of the skipped songs. In this case either song a or c is returned.

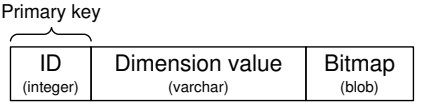


(a) Bitmap indexed distance store.

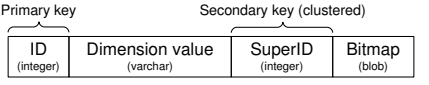


(b) Structure of the partition record.

Fig. 3. A distance store and a partition record



(a) Top level dimension record.



(b) Sub level dimension record.

Fig. 4. Structure of the metadata records

Finally, the MOD framework apply the ability to restrict the entire music collection in accordance with relevant music metadata such as genre, artist, etc.

4 Distance and Metadata Indexes

4.1 Distance Management

As indicated earlier, n distance stores are required for a music collection containing n songs, and each distance store has to hold all n songs. Thus, to cope with this n^2 space complexity a compact representation of the subsets is greatly needed. Moreover, as we deal with vast music collections that potentially may contain millions of songs, latency may occur when querying songs. By use of bitmap indexes [5], we have obtained not only a compact representation of the songs but moreover a very good query performance of the implemented music data model reducing the overall latency.

Assume now that a certain known order of the songs within a music collection exists, e.g., the order of insertion. A subset of songs from the music collection can then 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 (set) 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 of played songs. 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 assume a 32 bit computer architecture, whereby a single bitwise instruction computes 32 bits at once.

Using the *equality encoding* scheme for bitmap indexes, each distinct attribute value is encoded as one bitmap having as many bits as the number of records in the relation, i.e., the music collection [1, 5].

Selecting music in accordance with multiple attributes across several relations, bitwise bitmap operations may be used to replace expensive joins performed between the involved relations [15, 16]. Suppose 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 in the collection of skipped songs are returned to the listener.

Even though bitmap indexes constitute a compact representation of bulky data the nature of bitmaps provides the means for ensuring an even more compact representation by applying different techniques. In this paper, we discuss the use of two well-known bitmap compression scheme called the *WAH (Word-Aligned Hybrid)* [24], and an alternative representation technique known as *AVD (Attribute Value Decomposition)* [5].

In Section 3 the concept of the distance store was initially 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.

As described in Section 3, 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 3(a) an abstraction of a bitmap index of a distance store is illustrated for a collection of 5 songs having song s as the base song for the distance store. The music collection S is grouped into separate subsets S_1 through S_3 which constitute the individual partitions and their associated distance intervals. Considering the aspects of the WAH compression scheme, we need to consider whether a compression of the distance store is achievable. For that purpose we initially turn our attention to the structure of the partition record shown in Figure 3(b). Part 1 represents the positions of a given song and part 2 represents a numbered index indicating the position of the partition within the distance store. Together these constitutes a composite primary key. Part 3 contains the partition bitmap, which identifies all songs contained in the partition defined by part 1 and 2.

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 achieve 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. As the number of distance stores increases, the bit density in each store decreases and leads to a better compression ratio.

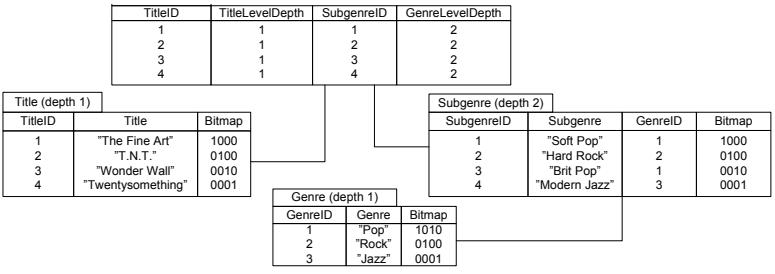


Fig. 5. Snowflake schema having the d_{title} and d_{genre} metadata dimensions

4.2 Metadata Management

To represent the multidimensional cube in a relational database, we adopt the *snowflake schema* known from multidimensional databases [23]. 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. While this saves space, it is known to increases the number of dimension tables thus resulting in more complex queries and reduced query performance [10]. 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 4. The *level record* in Figure 4(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 4(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 ids, 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.

In Figure 5 we consider the structure of the snowflake schema representing the fact table and dimension tables discussed above with respect to the metadata dimensions d_{title} and d_{genre}. From the fact table it appears that the involved 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 are shown. 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, aggregations of the bitmaps from a sub level

to a superordinate level are applied within the dimension hierarchy by use of bit-wise operations. As the bitmaps of the dimensions tables to a great extend contain only few 1 bits, these bitmaps may be highly compressed using the WAH compression scheme.

5 Query Processing

Having described the application of the bitmaps for both distance management and metadata management, these bitmaps may be combined in a single query in order to request songs from a restricted music collection. In this section we use pseudo algorithms to describe how to process such queries when requesting similar or random songs.

To describe the retrieval operators *RandomSong* and *SimilarSong* introduces in Section 3, we introduce the two helper functions *GenerateCompSkip* and *FetchRandomSongs*. The task of *GenerateCompSkip* 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 *FetchRandomSongs* is to enable the possibility to retrieve a specified number of randomly chosen songs from a given music collection represented by a bitmap.

The music collection initially passed to the respective two retrieval operators 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 4. The prefix notation $b\_$ is used to denote a bitmap.

The task of *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 operator *RandomSong* described in Algorithm 1, 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 operator 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.

The task of *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 operator *SimilarSong* presented in Algorithm 2 takes three bitmaps representing the search collection, the history and the set

Algorithm 1.
Pseudo code for *RandomSong*.

```
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_validColl)
 5  compositeSkipDS ←
    GENERATECOMPSKIP(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
          ← CUBELOOKUP(
              songRecord.filenameID,
              "Filename")
17  return filePath
```

Algorithm 2.
Pseudo code for *SimilarSong*.

```
SIMILARSONG
(b_coll, b_hist, b_skip, seedsongPosition, userId)
 1  filePath ← empty string
 2  songPosition ← Null
 3  b_validColl ← b_coll AND
      (b_coll XOR (b_skip OR b_hist))
 4  compositeSkipDS
      ← GENERATECOMPSKIP(b_skip, userId)
 5  seedsongDS
      ← DISTANCESTORELOOKUP
          (seedsongPosition)
 6  b_accSkip ← empty bitmap
 7  for each partition b_p in compositeSkipDS
    and b_q in seedsongDS starting with the partition
    representing the most similar songs
 8      do ▷ Check if candidate songs are available
 9      b_accSkip ← b_accSkip OR b_p
10      b_candidateColl ← validColl AND
          (b_q AND (b_q XOR b_accSkip))
11        if BITCOUNT(b_candidateColl) > 0
12          then ▷ Choose a position for a random song
13            songPosition
                ← RANDOM(b_candidateColl)
14            break
15  if songPosition <> Null
16    then ▷ Fetch the file path for the song found
17      songRecord
          ← FACTTABLELOOKUP(songPosition)
18      filePath
          ← CUBELOOKUP(songRecord.filenameID,
              "Filename")
19  return filePath
```

of skipped songs. In addition, the position of the seed song is passed to the operator, stating the position of the song within a bitmap corresponding to all songs in the music collection. Finally, the operator 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.

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 7 to 14). To ensure that only a song considered least similar to any skipped song is returned, the composite skip distance store is accumulated (line 9). Thus, restricting the partitions of the seed song distance store by the corresponding accumulated partitions of the composite skip distance store, the collection of candidate songs is obtained while only considering the songs contained in the valid collection. (line 10). 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.

6 Experiments

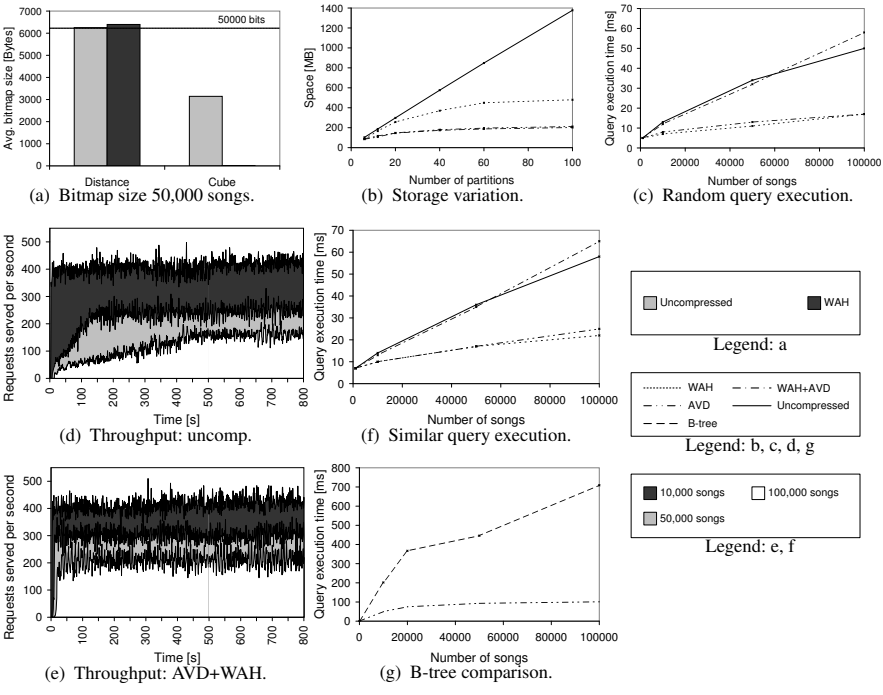
In this section the MOD framework is evaluated using various configurations. The evaluation concerns the space consumption introduced by the framework as well as the query performance when random and similar songs are retrieved. Moreover, we compare the MOD framework to an B-tree equivalent version. In the following, the tests are conducted using the MOD framework implemented with Java and MS SQL Server and are performed on a 32-bit Pentium 4 @ 3.0 GHz dual core with 3.25GB of main memory, running both the query evaluator and the SQL Server. The storage capacity is constituted by 3 x 400GB striped SATA disks. Each disk rotates at 7200rpm. The bitmaps within the databases 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.

For test purposes, synthetic music data is generated and added to the relevant music collections. In connection to this, aspects such as artist productivity, career duration and the number of songs on albums are considered to ensure a real-life reflection of the generated collections. Moreover, upon adding synthetic data to the music collections, entire albums are added in continuation of one another, which resembles the most common way of use. 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 only to decrease at the end. As we assume a highly diversified music collection, only few songs are located in the partitions representing the most similar songs.

6.1 Space Consumption

In general, a significant space reduction is obtained by applying AVD. AVD is thereby an obvious choice within the distance management. However applying only AVD, the bitmaps found within the metadata cube are then not reduced in size. Hence, the total space reduction is more optimal when applying both WAH and AVD. Moreover, it is relevant to consult the average bitmap sizes for the applied indexing. In this context, a difference is expected 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 thus subject to high compression whereas the bitmaps of the distance management are more diversified with respect to the occurrence of 0 and 1 bits, causing the compression techniques to become ineffective.

In Figure 6(a)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 using only seven bitmaps. In the figures the dashed horizontal lines indicate the threshold size needed to store 50,000 bits. The WAH compression yields a slight space overhead, i.e., no compression is possible. The average bitmap sizes for the *metadata cube* is reduced significantly when applying WAH compression. In this case, nine bytes are used for the average bitmap for 50,000 songs. This, in turn, causes the bars representing the cube



(a) Bitmap size 50,000 songs. (b) Storage variation. (c) Random query execution.

(d) Throughput: uncomp. (f) Similar query execution.

(e) Throughput: AVD+WAH. (g) B-tree comparison.

Fig. 6. results for storage, query execution and throughput

to become invisible in Figure 6(a).In the case of uncompressed bitmaps the average bitmap size within the cube is much below the threshold size 50,000 bits as the 0 bits in the end of the bitmaps are omitted.

Figure 6(b)shows how the number of partitions used within the distance management influences the space consumption for the four different bitmap representations. The space consumption of uncompressed bitmaps increases linearly to the number of partitions while the other three bitmap representations reach an upper bound. The growth of the WAH compressed type becomes minimal around 62 partitions. This is explained by the fact that on a 32-bit architecture, 31 consecutive bits have to be identical in order to constitute a compressible word. The WAH compression becomes effectives as soon as two consecutive compressible words are found, that is as soon as chains of 62 identical consecutive bits are present. The WAH compression occupies less space in practice than compared to the theoretical worst-case calculations. The remaining two bitmap representations are very close; this indicates that WAH compression on an AVD represented distance store does not gain a notable reduction. In fact, for 6 and 12 partitions an insignificant reduction is notable whereas for 20 partitions and above an important overhead is introduced.

6.2 Query Performance and Throughput

We conduct the query performance experiments on the four different bitmap representations. Unless stated otherwise, all tests assume that a given music collection is restricted

to 75% of the entire collection and that the number of partitions is fixed to 12. Moreover, 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. The test results are based on an average of 50 requests. Between each run the cache of the SQL Server is emptied in order to ensure a fair comparison on a cold cache.

In Figures 6(c) and (d), the average query execution time is presented for random and similar songs, respectively. All average query execution times on a collection of 100,000 songs are found to be at most 65ms. In case of solely applying WAH compression we have obtained an average query execution time at 17ms and 22ms 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 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. The reason for this is explainable by the reduced size of the bitmaps when searching for a candidate song in the border partitions of a distance store. The border partitions constitute the partitions representing the most similar and the least similar songs.

As can be seen from the figure the results are nearly linear, which reflects the expected linearity of appending skipped songs to the composite skip distance store. Independent on the chosen bitmap representation, less than 150ms is required to generate the composite skip distance store when none or a single song is skipped. When skipping 100 songs for each bitmap representation, we initially see that the WAH representation takes as long as 1.7s to construct the composite skip distance store. For the same amount of skipped songs, the two AVD representations perform faster compared to the two non-AVD 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 an additional reduction is achieved. The reason for this is that the length of the bitmaps representing the music metadata is reduced, whereby less data is the be retrieved from the database.

Next, 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. The request threads perform 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, and the other half random songs.

In Figure 6(e) and (f) the results obtained by execution of the throughput test are presented. The graphs indicates that, for all the test setups, no requests are served in the beginning of the conducted tests. The reason for this behavior is that the composite skip distance stores are generated during the first requests. In addition, some time elapses until the number of requests served stabilize. This is caused by the different query execution times related to the retrieval of random and similar songs. Figure 6(e) presents the results when applying neither AVD nor WAH. In this case we are able to serve around 400, 200 and 100 requests per second for indexing 10,000, 50,000 and 100,000 songs, respectively. When applying both AVD and WAH we have obtained the results presented in Figure 6(f). As expected from the previous results obtained, the performance decreases when the number of indexed songs increases. With respect to 10,000 song we see no notable increase in the number of requests served by the MOD framework. However, for 50,000 and 100,000 songs we are able to serve around 300 and 200 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. With 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 $5.56 \cdot 10^{-3}$ requests per second. Thereby, serving 200 requests per second on a database containing 100,000 songs, we are able to serve approximately 36,000 simultaneous listeners.

Finally, we now compare our framework to a "baseline" version using B-tree indexes for different size of the music collection. We chose a B-tree since other traditional indexes for high-dimensional data cannot be applied as the triangular inequality cannot be assumed to hold. The B-tree version has been indexed such as to achieve the optimal conditions for joining the tables described above.

We compare the performance of queries based on *metadata only*, i.e., without considering the similarity metric. To compare the two versions we execute 50 randomly generated restrictions in order the retrieve the filenames of the audio files associated with these restrictions. An example of such a restriction could resemble "all songs from the 70's that are of the genre Rock". The performed restrictions returns approximately 0.5% of the songs contained in the respective music collections. The outcome of this test is presented in Figure 6(g). For a given music collection containing 100,000 songs the B-tree version takes 709ms on average whereas the MOD framework using bitmap indexes used only 101ms. The bitmap indexes reduces the query time by a factor 7 while the space consumptions for the two approaches are *very similar*. If we should consider the similarities as well, the B-trees would have to index the distance stores for individual songs, each consisting of many songs, that must then be merged during query processing. B-trees are known not be an efficient way (neither time- nor space-wise) of doing this. We also know from the previous experiments that the time for handling both metadata and similarity in the bitmap version is not even twice of that for handling metadata alone. Since the B-tree version is much slower, even in the case where it has the best odds (metadata only), and the bitmap version can handle both metadata and similarity faster than the B-tree version can handle metadata alone, we can conclude that our bitmap approach is quite superior to using B-trees.

7 Conclusion and Future Work

Motivated by the proliferation of recommender systems performing similarity search in possibly non-metric similarity spaces, this paper proposes a innovative approach to flexible, yet effective, indexing for exactly such spaces. To illustrate this approach, the domain of music recommendation was chosen. Using a non-metric similarity measure, we are able to retrieve songs similar to a given seed song and avoid retrieval of songs similar to any disliked songs.

To facilitate musical similarity search, we introduced the *distance store* as an abstraction over an arbitrary similarity measure. The distance store constitutes a complete partitioning, where each partition represents a grouping of songs considered similar to a given base song. Applying bitmap indexes to represent each grouping, we are able to identify and retrieve songs similar/dissimilar to a given base song using bit-wise operations on the bitmaps associated with the individual groupings. Furthermore, in order to ensure efficient retrieval of songs based on metadata, we have constructed a metadata cube to which we applied bitmap indexing techniques. This multidimensional cube is mapped to a snowflake schema in an RDBMS, thus allowing a hierarchical representation of the music metadata.

We have thus demonstrated that bitmaps can be used to represent both metadata and non-metric distance measures. Using a single index method for the different music information, the MOD framework remains simple and highly flexible. Moreover, we have described how the framework applies bitmap compression using the Word-Aligned Hybrid compression scheme and the Attribute Value Decomposition technique. Experiments showed that the approach scaled well, both in terms of query performance, throughput, and storage requirements.

As future work, the MOD framework will be compared to other existing frameworks and indexes using various similarity measures, e.g., CompositeMap [4]. We will also address the use of bitmap 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 instead of consulting the bitmaps in a pairwise fashion, only to obtain a number of intermediate results, which again are to be consulted, the bitmaps could be stored in a special structure delaying the consultation until the result is required.

Acknowledgments. This work was supported by the Intelligent Sound project, founded by the Danish Research Council for Technology and Production Sciences under grant no. 26-04-0092.

References

[1] Silberschatz, A., Korth, H., Sudershan, S.: Database System Concepts, 4th edn. McGraw-Hill, New York (2005)
[2] Aucouturier, J., Pachet, F.: Music Similarity Measures: What's the Use? In: Proc. of ISMIR, pp. 157–163 (2002)
[3] Aucouturier, J.-J., Pachet, F.: Improving Timbre Similarity: How high's the sky? Journal of Negative Results in Speech and Audio Sciences 1(1) (2004)

[4] Zhang, Q.X.B., Shen, J., Wang, Y.: CompositeMap: a Novel Framework for Music Similarity Measure. In: Proc. of SIGIR, pp. 403–410 (1999)
[5] Chan, C.Y., Ioannidis, Y.E.: Bitmap Index Design and Evaluation. In: Proc. of SIGMOD, pp. 355–366 (1998)
[6] Ciaccia, P., Patella, M., Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proc. of VLDB, pp. 426–435 (1997)
[7] Digout, C., Nascimento, M.A.: High-Dimensional Similarity Searches Using A Metric Pseudo-Grid. In: Proc of ICDEW, pp. 1174–1183 (2005)
[8] Jensen, C.A., Mungure, E., Pedersen, T.B., Sørensen, K.: A Data and Query Model for Dynamic Playlist Generation. In: Proc. of MDDM (2007)
[9] Kimball, R., Reeves, L., Thornthwaite, W., Ross, M., Thornwaite, W.: The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses. Wiley, Chichester (1998)
[10] Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd edn. Wiley, Chichester (2002)
[11] Logan, B., Salomon, A.: A Music Similarity Function based on Signal Analysis. In: Proc. of ICME, pp. 745–748 (2001)
[12] Lübbers, D.: SoniXplorer: Combining Visualization and Auralization for Content-Based Exploration of Music Collections. In: Proc. of ISMIR, pp. 590–593 (2005)
[13] Mandel, M., Ellis, D.: Song-Level Features and Support Vector Machines for Music Classification. In: Proc. of ISMIR, pp. 594–599 (2005)
[14] Neumayer, R., Dittenbach, M., Rauber, A.: PlaySOM and PocketSOMPlayer, Alternative Interfaces to Large Music Collections. In: Proc. of ISMIR, pp. 618–623 (2005)
[15] O'Neil, P., Graefe, G.: Multi-table Joins Through Bitmapped Join Indices. ACM SIGMOD Record 24(3), 8–11 (1995)
[16] O'Neil, P., Quass, D.: Improved Query Performance with Variant Indexes. In: Proc. of SIGMOD, pp. 38–49 (1997)
[17] Pampalk, E.: Speeding up Music Similarity. In: Proc. of MIREX (2005)
[18] Pampalk, E., Flexer, A., Widmer, G.: Improvements of Audio-Based Music Similarity and Genre Classification. In: Proc. of ISMIR, pp. 628–633 (2005)
[19] Pampalk, E., Pohle, T., Widmer, G.: Dynamic Playlist Generation Based on Skipping Behavior. In: Proc. of ISMIR, pp. 634–637 (2005)
[20] Pedersen, T.B., Jensen, C.S.: Multidimensional Database Technology. IEEE Computer 34(12), 40–46 (2001)
[21] Pohle, T., Pampalk, E., Widmer, G.: Generating Similarity-based Playlists Using Traveling Salesman Algorithms. In: Proc. of DAFx, pp. 220–225 (2005)
[22] Stockinger, K., Düllmann, D., Hoschek, W., Schikuta, E.: Improving the Performance of High-Energy Physics Analysis through Bitmap Indices. In: Ibrahim, M., Küng, J., Revell, N. (eds.) DEXA 2000. LNCS, vol. 1873, pp. 835–845. Springer, Heidelberg (2000)
[23] Thomsen, E.: OLAP Solutions: Building Multidimensional Information Systems. Wiley, Chichester (1997)
[24] Wu, K., Otoo, E.J., Shoshani, A.: Optimizing Bitmap Indices With Efficient Compression. ACM TODS 31(1), 1–38 (2006)