MusicDB: A Query by Humming System

Edmond Lau, Annie Ding, Calvin On

6.830: Database Systems
Final Project Report
Massachusetts Institute of Technology
{edmond, annie_d, calvinon}@mit.edu

Abstract

We engineer an end-to-end music search system called MusicDB that supports query by humming. We represent musical tunes and hums as time series and use a time warping distance metric for similarity comparisons. A multi-dimensional index structure is used to prune the search space of songs and efficiently return the top hits back to an intuitive UI. Our user experiments on a database of fifty midis are promising; we find that MusicDB returns the desired song within the top 10 hits with 52% accuracy and as the top hit with 24% accuracy. Moreover, we believe that substantial room for improvement in search quality can be achieved with more accurate pitch extraction software and a more solid midi parsing library than the ones we used. Because few query-by-humming solutions have been fully documented in the research literature, we document all the core components of the MusicDB architecture to enable readers to build and extend our system if desired.

1. Introduction

Many people often remember a short tidbit of a song but fail to recall the song's name. If you can remember lyrics that correspond to the song you are trying to recall, finding the song is as easy as performing a text query on a web search engine. Alternatively, other music search services, such as the one iTunes provides, support querying capabilities on the basis of metadata tags (title, artist, genre, etc.) associated with music files. Even more sophisticated systems such as Pandora [16] allow musicians and listeners to collectively annotate a music database to build customized online radio stations with similar songs.

A query by humming system allows a user to find a song even if he merely knows the tune from part of the melody. The user simply hums the tune into a computer microphone, and the system searches through a database of songs for melodies containing the tune and returns a ranked list of search results. The user can then find the desired song by listening to the results.

Building such a system, however, presents some significantly greater challenges than creating a conventional text-based search engine. Unlike lyrical content, there exists no intuitively obvious way to represent and store melodic content in a database. The chosen representation must be indexable for efficient searching. Furthermore, several issues unique to query by humming systems pose significant challenges to creating an efficient and accurate music search system:

1. *Users may not make perfect queries.* Even if a user has a perfect memory of a particular tune, he may start at the wrong key, or he may hum a few notes off-pitch throughout the course of the tune. Sometimes he may even drop some notes entirely or add notes that did not exist in the original melody. Additionally, no user is expected to be able to perfectly hum at the same tempo as the songs stored in the database. Finally, since none of these errors are mutually exclusive, a humming query may contain any combination of these errors.
2. *Accurately capturing pitches and notes from user hums is difficult,* even if the user manages to submit a perfect query. Currently existing software for converting raw audio data into discrete pitch information is mediocre at best and oftentimes will introduce a great deal of noise when extracting the pitches from a user's hum.
3. Similarly, *accurately capturing melodic information from a pre-recorded music file is difficult.* Properly extracting the melody from a given song is a field of study on its own but is absolutely critical for an accurate query by humming database. Creating a perfect query

would be of little use if the database contains inaccurate representations of the target songs.

We designed and implemented MusicDB, a query by humming system that fuses together techniques from the melody extraction machinery developed by Uitdenbogerd and Zobel [22] and the query by humming system implemented by Zhu and Shasha [24]. These techniques enable us to build a searchable database of polyphonic midi files that can return the desired song within the top 10 hits with 52% accuracy and as the top hit with 24% accuracy. Moreover, the quality and performance scales reasonably with increasing database size.

Our main contributions in architecting MusicDB and in writing this paper include:

- Building an end-to-end music search system over a database of real songs that accepts user humming queries and returns quality results with reasonable response times.
- Documenting, in detail, the core components of our system so that readers can recreate and extend our system in the future. While some previous research has been conducted in this field, few people have written about the implementation of an end-to-end system, and reconstructing the work of those systems that are actually documented in the published literature is non-trivial.
- Engineering an intuitive user interface that accepts user humming queries and returns a ranked list of results.
- Independently verifying that Uitdenbogerd and Zobel's melody extraction heuristics and Zhu and Shasha's indexing schemes actually work in practice.

The rest of this paper is structured as follows. We discuss related work to query by humming systems in Section 2 and introduce some core concepts involved in building a time series-based query by humming system in Section 3. We detail our MusicDB system's architecture and implementation in Section 4 and describe the graphical user interface to MusicDB in Section 5. In Section 6, we present the results of our tests on MusicDB's performance and scalability. We finally conclude in Section 7.

2. Related Work

Melody extraction is an essential part of query by humming systems using polyphonic music since the user typically hums the melody. Identifying the melody involves some music perception principles including: 1) the highest musical notes are generally perceived as the melody, and 2) notes close in pitch are usually grouped together mentally. Uitdenbogerd and Zobel created and tested four different melody extraction algorithms on a small set of midi files [24]. In their tests, they found the two best performing algorithms were:

1. Combining all musical information into one stream of events and keeping only the highest note at each time interval, resulting in monophonic music. This is also known as the skyline algorithm since it preserves the highest contour.
2. Processing each midi channel separately, running the skyline algorithm on each channel, and then choosing the channel with the highest average pitch as the melody.

Chai used the skyline algorithm on polyphonic midis with labeled melody tracks to build a query by humming system [4]. Thus, the track labels saved her the work of identifying the correct melody track of the midi.

The first generation of query by humming systems used a contour representation of melodies, usually in the form of string sequences representing the pitch intervals. This representation was widely accepted because people are well-known to be able to precisely hum correct pitch intervals, but not always at the right key [5, 13, 16]. The first full query by humming system implemented this approach by using the letters "U", "D", and "S" to represent pitch values going up, down, or remaining the same [5]. A later system that took the same approach extended this by accompanying these UDS strings with interval sizes [13]. Another system called CubyHum simply stored strings of interval sizes, using positive and negative values to denote pitch direction [17]. In all of these systems, hummed queries are converted into UDS or UDS-like strings, and the search is then performed by sequence matching the query string with all the melody strings in the database. To accommodate for user humming errors, some of these systems employed "fuzzy" string matching techniques that tolerated some degree of inconsistency between strings [5].

Several problems make the above approach unattractive. First, string matching is typically expensive even with the best algorithms. Second, just using contour information to query is not scalable because pitch direction does not provide enough information to distinguish songs from each

other in a sufficiently large database [13, 24]. Lastly, even with fuzzy string matching, contour-based systems remain heavily dependent on accurate note transcription tools that are not yet attainable with current technology.

Because of these issues with the contour approach, more recent systems have been developed to explore new ways of representing melodic information. Chai found that a representation combining time signature, pitch contour, and rhythmic information was more effective than systems that relied solely on pitch information [4, 11]. Most recently, Zhu and Shasha devised a method for representing melodies as time series [24]. In a time series representation, songs and user hums are sampled for discrete pitch values at regular time intervals. Using this system, they were able to leverage the existing body of research on time series indexing developed by the database community; their system also demonstrated superior performance to traditional contour-based systems. It is because of the scalability benefits of a time series representation that we also choose to represent songs and hums as time series in our MusicDB implementation.

Lastly, some systems have also experimented with using whistling instead of humming for query by humming based systems. For example, TANSEN only accepted whistling inputs from users and obtained comparable results to most humming systems [19]. Many other systems, such as Microsoft Research's query by humming system, are geared towards humming input but also allow whistling input as an alternative [13]. Whether one method is actually better than the other remains unclear because comparative studies on the issue have yet to be performed. This is possibly because many other variables besides humming versus whistling can affect query quality.

3. Core Concepts in Efficient Similarity Search for Time Series

The architecture of MusicDB's similarity search components draws heavily from previous work; in particular, it borrows the following three core techniques:

1. Local dynamic time warping as a distance metric for comparing time series.
2. Time-series indexing via dimensionality reduction to efficiently perform queries.
3. Piecewise aggregate approximation as a mechanism for dimensionality reduction.

Zhu and Shasha combined these techniques to build a query by humming system for a database of fifty short Beatles songs stored in monophonic midi files with cleanly defined melodies. MusicDB represents our engineering endeavor to extend their system with the ability to query over a database of pop songs that average three to four minutes in duration and that have polyphonic tracks where the melody is typically decorated with accompaniments.

In this section, we provide a high level explanation of each of these three techniques that function as a basis for MusicDB's similarity search components and refer the reader to related work for additional detail.

3.1 Local Dynamic Time Warping

If we treat each value in a time series as a separate dimension, then we can view a time series of length *n* as a point in *n*-dimensional space. Given a set of points *S* in this space, a point ***x*** is most similar to the point in *S* that minimizes some distance function. Simple functions such as Euclidean distance, however, are highly sensitive to one of its inputs being out-of-phase, missing a data point, or having additional noise points; all of these problems become critical in a query by humming scenario where hummers may miss notes, hum additional notes, or be off tempo and hum different portions of a song at a faster or slower pace than normal.

For MusicDB, we therefore use a distance metric called local dynamic time warping (LDTW) developed originally by the speech processing community [15, 18, 20] as dynamic time warping (DTW) and tweaked slightly by [24] in their query by humming system. Mathematically, the LDTW distance between two time series ***x*** and ***y*** is defined recursively by Zhu and Shasha as:

$$D^2_{LDTW(k)}(x,y) = D^2_{constraint(k)}(First(x), First(y)) + \min\left\{\begin{array}{c} D^2_{LDTW(k)}(x, Rest(y)) \\ D^2_{LDTW(k)}(Rest(x), y) \\ D^2_{LDTW(k)}(Rest(x), Rest(y)) \end{array}\right\}$$

$$D^2_{constraint(k)}(x_i, y_j) = \left\{\begin{array}{ll} D^2(x_i, y_j) & \text{if } |i-j| \le k \\ \infty & \text{if } |i-j| > k \end{array}\right\}$$

$$D^2(a,b) = (a-b)^2$$

Figure 1: In Euclidean distance, the ith point of one time series is directly compared to the ith point of the other and is highly sensitive to alignment errors. LDTW accounts for misalignment by stretching and shrinking certain portions to provide the best match.

Intuitively, LDTW employs dynamic programming to elastically shrink and stretch different portions of the time series to create the best fit as illustrated in Figure 1.[1] The "local" in LDTW refers to the parameter *k*, which constrains the amount of stretching that can occur and accounts for the fact that even the worst hummers have a limit to how far they can be off-tempo; it also reduces the computational complexity of LDTW from $O(n^2)$ to $O(kn)$. The interested reader is referred to [24] for further information regarding the LDTW algorithm.

3.2 Efficient Indexing and Matching of Time Series

Armed with LDTW and a time series query $\boldsymbol{q}$ of length *m*, we could theoretically iterate over all possible time series $\boldsymbol{x}$ of length *m* for all the songs in the database, compute LDTW($\boldsymbol{q}$, $\boldsymbol{x}$), and report the $\boldsymbol{x}$ with the minimum distance as the most similar match. Such an approach would, however, take $O(n \cdot (d - m))$ comparisons where *n* is the number of songs and *d* is the average duration in number of time points for a song; furthermore, the dynamic programming involved in an LDTW comparison is expensive.

Efficient similarity search methods on time series, as described by work done in [1, 7, 8, 10], typically involve applying some dimensionality reduction technique on the data and then indexing the transformed data in a multidimensional index structure such as an R-Tree. The key optimization in this architecture is that at query time, an inexpensive lower bounding distance function D′ where

$$D'(\mathit{Transform}(\boldsymbol{q}), \mathit{Transform}(\boldsymbol{x})) \leq \mathrm{LDTW}(\boldsymbol{q}, \boldsymbol{x}),$$

is used to prune out unlikely candidates in the transformed space before applying the computationally more expensive metric in the original space.

[1] Figure taken from E. Keogh and C.A. Ratanamahatana, *Exact indexing of dynamic time warping*, Knowledge and Information Systems, May 2004.

3.3 Piecewise Aggregate Approximation

A number of dimensionality reduction techniques have been used for indexing high dimensional data, including Discrete Fourier Transform (DFT), Discrete Wavelet Transform (DWT), and Singular Value Decomposition (SVD) [6]. We choose to use piecewise aggregate approximation (PAA), also known as piecewise constant approximation (PCA) [8], as our dimensionality reduction technique because both [24] and [9, 10] have shown that it offers a tighter lower bound to the true distance than the other techniques mentioned and therefore provides a higher pruning factor.

Mathematically, the PAA transform to reduce a length *n* time-series $\boldsymbol{x}$ into a compressed segment $\boldsymbol{X}$ of length *N* where $N < n$ is defined as:

$$X_i = \frac{N}{n} \sum_{j=\frac{n}{N}(i-1)+1}^{\frac{n}{N}i} x_j, i = 1,2,...,N .$$

Intuitively, the PAA transform simply divides the time series into *n/N* equal-length windows and represents the time series with the average value within each window.

4. Architecture of MusicDB

The MusicDB architecture consists of four processing pipelines connected to a graphical user interface. We defer discussion of the UI to Section 5, and focus on the four components in this section. These pipelines, illustrated in the architectural diagram of Figure 2, include:

1. A melody extraction pipeline that heuristically selects the likely melody channel from a polyphonic midi file and constructs a time series representation of the melody.
2. A loading pipeline that accepts the time series representation of the midi, records the midi in the catalog, and inserts a searchable representation of the midi into an R-tree.
3. A query input pipeline that accepts the user humming input and transforms it into our own storage representation.
4. A searching pipeline that uses the output from the query input pipeline to probe the index and return a ranked list of results.

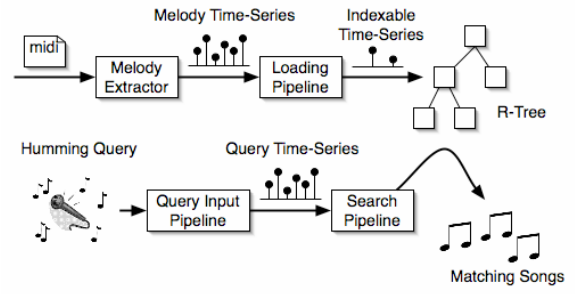

Figure 2: Architectural diagram of interactions between MusicDB pipelines.

Together the four pipelines expose the following query API to the UI front end:

```
// initializes the musicDB catalog and index
// with the midis in the specified directory by
// passing the midis through the melody extraction
// and loading pipelines
void initMusicDB(String midiDir)
// convert a recorded user hum (.wav) into a
// midi format – query input pipeline part 1
void wavToMidi(String wavFile, String midiFile)
// convert a user hum from a midi format to our
// own storage representation – query input
// pipeline part 2
QuerySegment processQueryFromMidi(
   String midifile, int maxQueryLength)
// submit a query to the database and get a ranked
// list of results -- searching pipeline
List<MusicSegment> query(QuerySegment query)
```

We construct a searchable database of midis by calling `initMusicDB` on a folder of midis to send each midi through the melody extraction and loading pipelines. Additional midis can be added to the database by feeding it through the same pipelines. This database is typically constructed offline, saved, and loaded prior to query time with a folder of midis. Executing a query against the database consists of two phases: transforming a user humming input into our basic segment data type with calls to `wavToMidi` and `processQueryFromMidi` and looking up the segment in MusicDB's index with a call to `query`.

We first describe the basic data structures and operator interface used by MusicDB, and then proceed to explain the machinery and algorithms used within each pipeline.

4.1 Segments and Filters

As in Zhu and Shasha's work, we represent each melody in a musical score using a time series representation. The basic data type in MusicDB is a time series of pitch values called a `Segment`, which we treat as a unit of data similar to a tuple in a traditional database but without a schema. A segment exposes the following basic interface:

```
int size()
double getValueAt(int index)
```

Segments may also be annotated with additional information not provided by the basic interface; for example, a `MusicSegment` stores additional information that links it back to the original midi file, such as the `MidiId`, start time, and duration.

Segments are processed by `Filters`, which correspond to the pull-based operators in traditional databases and export the familiar iterator interface:

```
void open()
Segment getNext()
void close()
```

Each filter processes input segments and outputs zero or more transformed segments for each input segment. All three pipelines use segments as the underlying data representation. The loading and search pipelines manipulate segments using filters, while the query input pipeline produces a segment from a user hum query.

4.2 Extracting a Time-Series Representation of Melodies from Polyphonic Midi Files

We downloaded a collection of 400 polyphonic midis from the web. The purpose of the melody extractor is to determine a time series representation of a song's melody to be used for similarity matching against humming samples. To do so, the melody extractor uses the open-source JMusic library [3] to parse midi files into Java data structures, extracts the pitches of the song's melody using a combination of channel selection heuristics and the skyline algorithm, and outputs a segment capturing the time series information of the melody.

Midi Songs

We chose to create our database of songs using songs in the midi file format. This was a natural choice because the midi representation already discretizes the notes, making it easier to extract the pitch and timing information necessary for our song matching. Alternate music file formats such as wav, mp3, aiff, etc. would require complicated waveform and signal processing that could lead to many inaccuracies. Each of our songs is also mapped to a set of metadata attributes such as song name and song artist for eventual display in the GUI result list.

Figure 3: Melody extraction pipeline.

Melody Extraction

Converting the music from our midi collection into time series consists of several steps, illustrated in Figure 3. Since the user queries we want to match are hums of parts of melodies, our queried data needs to be in melody form as well. This raises the major challenge of extracting the melody from our midi files. Although the JMusic library makes extracting music information easier, it unfortunately also gives incorrect midi information at times, resulting in some inaccuracies in our transcriptions. Our approach to melody extraction is similar to that used by Uitdenbogerd and Zobel, but we fuse together parts of their two best performing algorithms to heuristically choose the melody channel and to extract a time series representation.

Channel Selection

Unlike Zhu and Shasha's database, the midi files we use are polyphonic with multiple overlapping music track channels, since this kind is most common for the popular music. However, the polyphonic midis introduce the challenge of finding a systematic way to choose the channel of the midi containing the melody. In our tests of 50 midi files, we find that the melody is usually contained in just one channel of the midi. However, in a few cases the melody is actually split across multiple channels. For simplicity, we work on the assumption that the melody will always be found in just one channel.

This channel is difficult to isolate because, unlike Chai's system [4], none of our midi files take advantage of the labeling functionality in the midi format. Thus, we use heuristics to analyze the attributes of each channel and to probabilistically determine the melody channel. First, we filter the channels so that only channels with melody-carrying instruments remain. For example, we remove all the drum channels since drums never carry the melody. We then analyze the music notes and estimate the average pitch of each channel. Since the melody is typically higher in pitch than the rest of the notes [22], we isolate the channels with the top 3 average pitch values. Finally, since some accompaniments can be higher pitched than the melody and either have very few notes (e.g.: a few high bells) or very many notes (chord accompaniments), we choose the channel with the most single notes per time segment as the melody.

Skyline Algorithm

The melody channel obtained at the end of channel selection can still have many overlapping sections called phrases, but we want to have only one note per unit of time in our final representation in order to match incoming queries. To do this we use the skyline algorithm [22], a simple approach to melody extraction. This algorithm prunes music down to just one pitch per time unit essentially by keeping the top pitch whenever simultaneous notes occur.

```
Function
Skyline(Vector<Note> notes)
1.  melody = new TimeSeries;
2.    for each Note N in notes
3.       //if there is any overlap between
4.       //time of note N and one of the
5.       //notes already contained in the
6.       //melody series, keep only the note
7.       //with the highest pitch
8.       if hasOverlap(N, melody) and
9.          N.pitch < melody[N.time].pitch
10.        continue
11.      melody[N.time] = N
12. return melody;
```

In order to run the skyline algorithm, we first transform each of the phrases into time series form. Since many of the time encodings of midi notes are not exact in rhythm, our transformation to time series involves forcing the notes into quarter note slots so that all notes can align properly for comparison. Each time sample represented a quarter note because that was the smallest interval our system could support while still maintaining fast search performance. Rests in this form are indicated by the lack notes at times. This representation makes it very easy to do the time overlap and relative pitch comparisons necessary for the Skyline algorithm.

```
Function
OrderIntoSlot(Phrase phrase)
1.  pitches <- Vector<TimeSample>
2.  time <- phrase.getStartTime()
3.  lastAssignedTime <- -1;
4.  for each Note N in phrase
5.     noteStartTime <-
6.      roundToNearestQuarter(time);
7.     assignedTime <-0;
8.     while (noteStartTime + assignedTime <
9.     lastAssignedTime)
10.       //the smallest note time used in
11.       //our system was a quarter note
12.       assignedTime+=smallestNoteTime;
13.    while(noteStartTime + assignedTime
14.     < time+N.getDuration()){
15.       //if the note is not a rest
16.       //add the note at the calculated
17.       //assigned time
18.       if(pitch>0)
```

```
19.         pitches.add(TimeSample(N.pitch,
20.           noteStartTime+assignedTime);
21.       else
22.         assignedTime +=smallestNoteTime;
23.       //update the time counts
24.       lastAssigned = noteStartTIme +
25.        assignedTime;
26.       time +=N.getDuration()
27.  return pitches;
```

The final result of the skyline algorithm and melody extraction is a time series representation of the music file, with a pitch and time for each note.

4.3 Indexing a Time-Series

The goal of the loading pipeline, illustrated in Figure 4, is to index the extracted melody time series for efficient querying. Prior to further processing, the extracted melody is inserted into the `MidiCatalog`, which assigns to it a unique `MidiId`. The MidiId allows us to match processed segments with the original midis.

The output of the melody extraction pipeline is sliced into overlapping windows of a fixed length via the segmentor filter. Each of these music segment windows carry the original `MidiId` and timing information pointing to its location in the original midi; each segment will eventually become a subsequence of the original that can be matched against user queries. We chose to set our segment size to thirty data points, expecting the average user query to be seven to eight seconds in duration. We experimentally determined a reasonable offset from the start of one window to the start of the next to be five time-series values, corresponding to roughly 1.25 seconds of music. After the time information is used for segmentation, the rests are discarded. This decision was supported by the many systems such as [24] which found that rest information was not necessary for making query matches. To account for humming queries being off-key, each window segment is normalized by a Mean Removal filter that subtracts the window's average pitch value from each point in the window. A humming query will be normalized in a similar fashion to remove the effects of the key.

Because multidimensional index structures typically degrade in performance past 8-10 dimensions, rather than inserting the normalized window segments into an R-Tree, we pass the segments through a PAA Transform that applies the PAA dimensionality technique in order to reduce the size of the data. Each of these compressed segments is then inserted into the R-Tree, along with information relating the segment back to the original midi file. For our R-Tree index, we use the open-source implementation of a Java Spatial Index Library [21], developed in UC Riverside. We explain how the index supports efficient queries in Section 4.5.

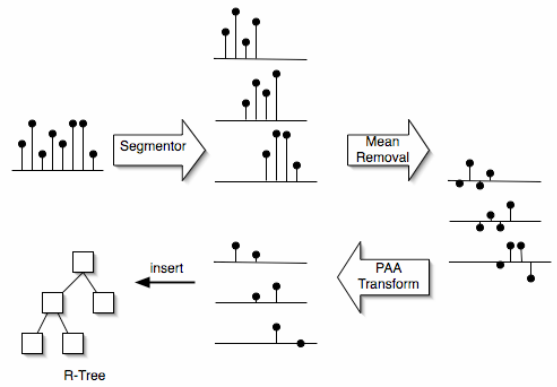

Figure 4: MusicDB loading pipeline. The arrows represent `Filter`s and the time series represent `Segment`s. A midi file entering becomes stored in a segment after melody extraction. The segmentor chops the melody segment into overlapping windows, each of which undergoes a mean removal and PAA transformation before being indexed in an R-Tree.

Processing a User's Input Hum Query

In order to obtain the user's hum, we used standard Java sound file streaming libraries to record the hum into monophonic wave format. We then wanted to convert the hum into a midi format so that we could process it like we did all of our other midis via a wave to midi file converter. Although there are many implementations of wave to midi converters, we used one of the few that run on Mac OS X since that was the platform we chose to support. As a result, we decided to use WaoN, an open-source notes transcriber that samples the continuous sound in a wave file to make the discrete representation of the midi file [23].

WaoN allowed us to convert the files and remove some of the noise while also increasing the volume of the query. However a substantial amount of noise still remained, and as a result we passed the resulting midi through a band-pass filter to remove all the outlying high pitch and low pitch noise.

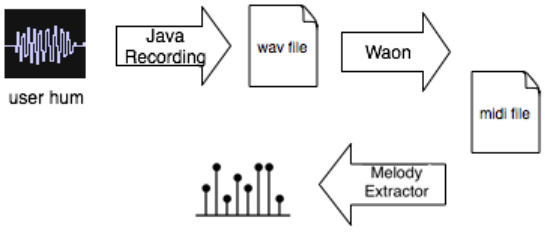

Figure 5: Humming extraction pipeline.

Once the noise was all removed, we were left with a short melody midi segment, which we would put through the same processing pipeline we used for the midis in our database. Figure 5 illustrates this entire process.

4.4 Efficient Querying Using the Database

Given a humming query, MusicDB returns the top K songs that match the query. The intention of the multi-dimensional index structure is to save computation over a sequential scan of all the window segments constructed by the Segmentor filter in Section 4.2. We use two techniques to calculate lower bounds on the true LDTW distance between the query segment and a music segment or a cluster of music segments in the transformed space; these techniques enable us to prune areas of the search space whose segments will definitely not make the top K songs without actually computing the LDTW distance for each segment.

The first technique, introduced by [10] but improved by [24], computes, in the transformed space, a lower bound of the distance between the query segment and an indexed segment using a mathematical construct called the query envelope. While a complete proof and explanation of the method is beyond the scope of this paper, we include the mathematical formula for this calculation for completeness.

First, we compute the K-envelope <***l,u***> of a query ***q***:

$$l_i = \min_{-k \le j \le k}(q_{i+j}), i = 1,...,n$$
$$u_i = \max_{-k \le j \le k}(q_{i+j}), i = 1,...,n$$

Then, we calculate the PAA transform of each of the segments ***l*** and ***u*** as if each was its own time series, using the PAA equation from Section 3.3 to obtain two length *N* sequences ***L*** and ***U***. The lower bounding distance LB_PAA is then computed as:

$$LB\_PAA(Q,D) = \sqrt{\frac{n}{N}\sum_{i=1}^{n}\begin{cases}(D_i - U_i)^2 & \text{if } D_i > U_i \\ (D_i - L_i)^2 & \text{if } D_i < L_i \\ 0 & \text{otherwise}\end{cases}}$$

The second technique, presented in [10], computes a lower bound between the query segment and the minimum bounding region ***R*** of an index node, with the possibility of pruning all segments in the search space beneath the index node. Letting $\boldsymbol{R^H}$ denote the upper bound of ***R*** and $\boldsymbol{R^L}$ denote the lower bound of ***R***, the mathematics for this technique looks like:

$$MINDIST(Q,R) = \sqrt{\frac{n}{N}\sum_{i=1}^{n}\begin{cases}(R_i^L - U_i)^2 & \text{if } R_i^L > U_i \\ (R_i^H - L_i)^2 & \text{if } R_i^H < L_i \\ 0 & \text{otherwise}\end{cases}}$$

Combining the two techniques, we arrive at the `K-NearestNeighborQuery` algorithm, which is as follows:

```
     Function
     K-NearestNeighborQuery(Query Q, Integer K):

     queue: MinPriorityQueue
     examined: SortedList<time-series, distance>
               // sorted by distance
     best_so_far: Map<MidiId => distance>
     results <- Set<MidiId>

1.   queue.push(root_node, 0)
2.   while queue is not empty
3.       top <- queue.pop()
4.     while examined is not empty
5.       closest <- first(examined)
6.       // any time-series closer than top is
7.       // closer than anything unexamined
8.       // and is therefore a nearest neighbor
9.       if LDTW(Q, closest) <= top.dist
10.        Add closest.MidiId to results
11.        Remove closest from examined
12.        if |results| = K
13.          return results
14.      else
15.        // everything behind closest is
16.        // farther because examined is sorted
17.        break
18.    if top is PAA point
19.      if results contains top.MidiId
20.        continue
21.      Retrieve from catalog the full time-
22.         series T for top.MidiId
23.      T' <- slice of T from index at top.start
24.            running for Q.length values
25.      S <- NORMALIZE(T')
26.      if LDTW(S,Q) < best_so_far.get(S.MidiId)
27.        best_so_far.put(S.MidiId, LDTW(S,Q))
28.        Remove from examined any time-series
29.          with MidiId = S.MidiId
30.        examined.insert(S, LDTW(Q, S))
31.    else if top is a leaf node
32.      for each data point D in top
33.        queue.push(D, LB_PAA(Q, D))
34.    else
35.      for each child C in top
36.        queue.push(C, MINDIST(Q, C)
37.
38.  while |results| < K and examined not empty
39.    S = examined.removeFirst()
40.    add S.MidiId to results
```

LB_PAA refers to the first technique, and MIN_DIST refers to the second. The algorithm is adapted from the KNN-Search algorithm described in [10] but modified in three major ways: 1) rather than returning the top K matches, we return the top K matches with the constraint that each of the K matches must come from distinct songs, 2) we extend the algorithm in lines 21-24 based on the ideas in [9] to handle queries longer than the time-series originally indexed, and 3) we normalize the retrieved segments before comparing in order to account for off-key queries. The implication of the second change is that a longer query should theoretically produce more accurate results. Experimentally, we have found that our algorithm is often able to only calculate LDTW distances for under 60% of the total segments before determining the top K matches.

Note that this algorithm differs from the search algorithm used in the query by humming system of [24] because their system uses an ε-range query to find all segments with LDTW distance no greater than ε away from the query and because they do not use the second pruning technique. We in fact implemented both algorithms and found our nearest neighbor algorithm to have better performance.

5. UI discussion

For the purposes of this project we wanted to make a simple but friendly user interface so that users could make queries easily. The UI comprises of two parts: one for obtaining user input for queries, and one for displaying query results.

Users can choose to input their queries either by humming or by playing the tune on an on-screen piano. In our implementation we decided to focus on the humming input usability since it was a more natural form of input, more efficient to use, and not limited only to users with prior training with a piano. Our humming interface, shown in Figure 6, allows users record hums using a microphone and play hums back to check the quality. Alternatively, the piano interface, as seen in Figure 7, is a simple piano where people can click on keys to play notes that will be transcribed into a query. These UI components included some extensions of the source code provided by the Java Sound Demo to render both the keyboard and waveform [14]. Once a query is ready, users can hit the "Search for Song" button to perform the query.

Upon completion of the query, the top 20 results will show in a ranked list, with the closest match ranked first. This listing also displays metadata such as the song name and artist who wrote or performed the song. From there, the user can select a song in the listing for playback and even play the segment of the song that matches the query by pressing the "Play matching segment" button. In essence, the results list can be viewed as a generated playlist of all songs found to be similar to the query.

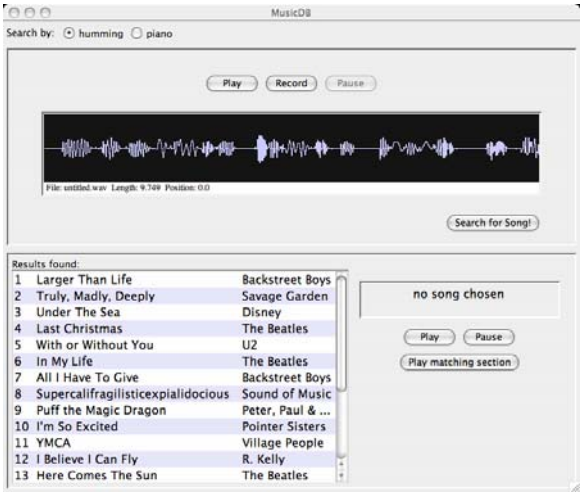

Figure 6: Query by humming UI.

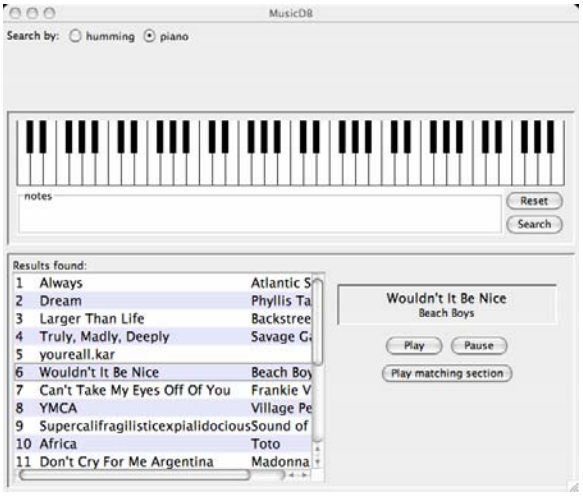

Figure 7: Query by piano input UI.

6. Results

In this section, we discuss and analyze some experimental results on MusicDB. These experiments focused on two major aspects of our query by humming system: quality and scalability. Quality is defined as how accurately our system returns results for a given query. Scalability is measured in terms of both quality and performance of the system as we increase the number of songs in the database.

6.1 Quality of Query by Humming

We created a database of 50 popular songs out of midi files collected from a popular Karaoke website. The midi files were loaded into the database using MusicDB's loading pipeline described Section 4.2. We then collected recordings of hums from different 10 users for a total of 50 humming samples. These recordings were pushed through the query input pipeline and the search pipeline to generate the results shown in Figure 8. From the figure, we can see that MusicDB returns the correct result within the top 10 for 52% of the time, and returns the correct result as the top hit 24% of the time. We compared our results against Zhu and Shasha's results from experiments of 40 humming samples on a database of 50 songs on their own query by humming system, shown in Figure 9.

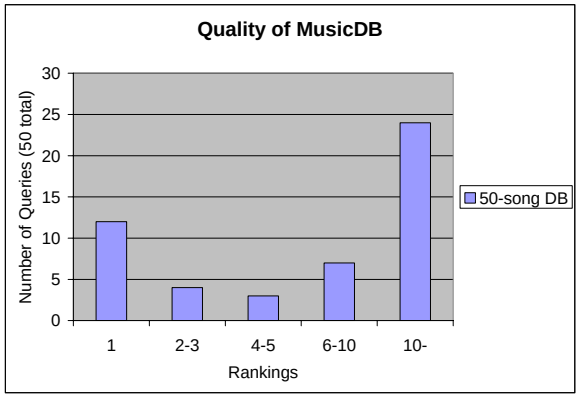

Figure 8: Aggregate rankings of 50 user hums passed through the MusicDB pipeline.

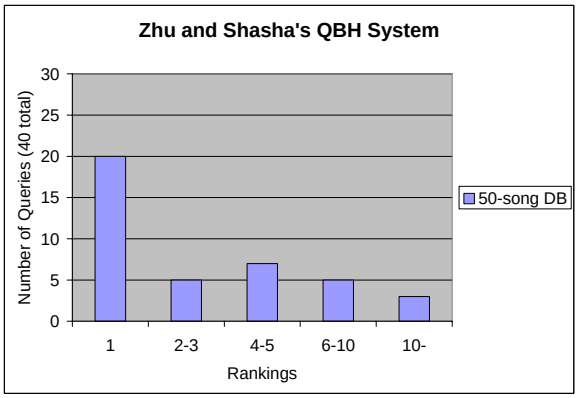

Figure 9: Zhu and Shasha's reported results for passing 40 user hums into their query by humming implementation.

The graphs suggest that MusicDB does not match Zhu and Shasha's system in terms of search quality. However, several differences in the methodologies used for our experiments account for the discrepancy.

First, Zhu and Shasha manually entered single-channel, monophonic songs into their database. As such, they avoided the melody extraction problem altogether and guaranteed that the correct melodies for their songs would be stored in the database. Our system used widely available multi-channel polyphonic midi files. Even though our melody extraction techniques successfully retrieved large portions of the melody for most midis, some of the midis were either improperly labeled, had their melodies divided into multiple channels, or had high accompaniment pitches that would occlude the melody notes in the skyline heuristic. Because our database did not have perfectly accurate representations of certain songs, MusicDB suffered from a loss in quality in rankings that Zhu and Shasha sidestepped completely, accounting for a significant amount of the discrepancies between our results.

Second, we also experienced some trouble with properly converting and extracting user midis from recorded wave files. Zhu and Shasha used a piece of commercial software, AKoff Music Composer 2.0 [2], to record and transcribe notes from a user's query. In contrast, MusicDB utilizes a free open-source tool, WaoN v0.1, to transcribe notes. It is highly likely that the commercial software affords much higher fidelity note transcription.

Third, Zhu and Shasha actually divided their 40 humming samples into a group of "better hums" and "poorer hums" and used different parameters for

each batch to produce more optimal results. We, however, felt that such a categorization of hummers was somewhat arbitrary and used a uniform set of parameters for all hummers, perhaps at the cost of reduced search quality.

Lastly, we did not entirely control for noise in the environments when collecting user hums. Consequently, the recordings varied in quality, and in general, recordings of low quality ended up returning bad results.

If we were to normalize for these differences, we believe that our search quality would increase substantially and that our results would actually be quite comparable to those reported by Zhu and Shasha.

6.2 Scalability of MusicDB Quality

To measure how well quality scales with database size, we ran the same experiments from the previous section on a database of 100 songs (double the original 50). We compare these results against our original experiment in Figure 10.

As illustrated by the chart, a general decrease in quality can be observed with the larger database. The decrease is rather slight, however, and indicate that MusicDB scales quite well in terms of search quality.

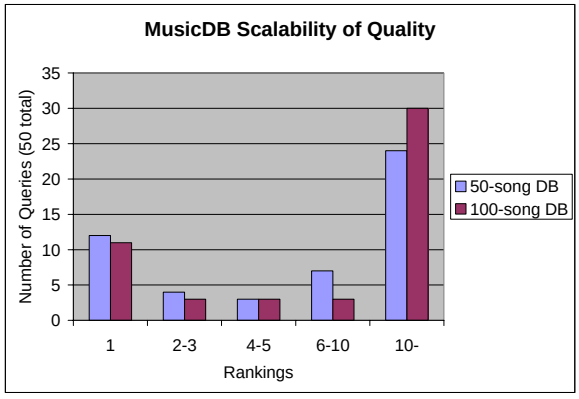

Figure 10: Aggregate rankings of 50 user hums passed on databases of 50 and 100 hums.

6.3 Scalability of Indexing Infrastructure

We conducted a few experiments to measure the scalability of MusicDB's performance as we increase the number of songs in the database. We collected performance numbers over databases of 50, 100, 200, 300, and 400 songs. Figure 11 shows the size of the catalog and the index structure (in terms of KB) as a function of the number of songs in a database. Each additional song increases the catalog and index size by roughly 35 KB, which is slightly less than the space used by a typical midi file. Figure 12 shows the total number of segments in the index, and Figure 13 shows the average fraction of all segments in the database examined by a single query. Figure 14 shows the average query time on 2.4 GHz Pentium 4 PC with 512 MB of RAM, 256 MB of which were reserved for the Java Eclipse environment.

Note that while the space overhead, the total number of segments, and the average query time increases roughly linearly with the number of songs in the database, the fraction of segments for which the true LDTW distance must be calculated actually decreases from 66% on a database of 50 songs to 59% on a database of 400 songs. We speculate the reason for this decrease to be that the parameters (page size, branching factor, etc.) for our R-tree is actually geared toward larger data sets. While this fraction represents a reasonable amount of savings, we conclude that the time-series indexing architecture we adopted from the cutting-edge research literature does not scale well enough to search millions of songs unless the constant factor can be drastically reduced.

Another observation is that we can decrease the space overhead of the index without affecting accuracy by increasing the compression ratio used during dimensionality reduction. For these experiments, we used a compression factor of 5. The lower-bounding distance metric in the transformed space, however, becomes less tight in the transformed space with increasing compression ratios, which in turn would lead to an increased number of segments examined and an increased querying time. Thus, a tradeoff exists between the space overhead of the index and query performance.

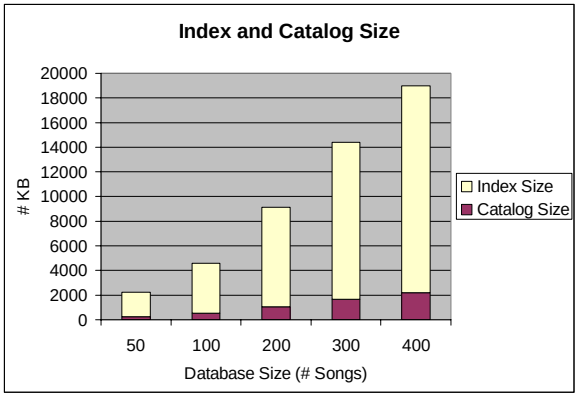

Figure 11: Index and catalog size as a function of the number of songs in the database.

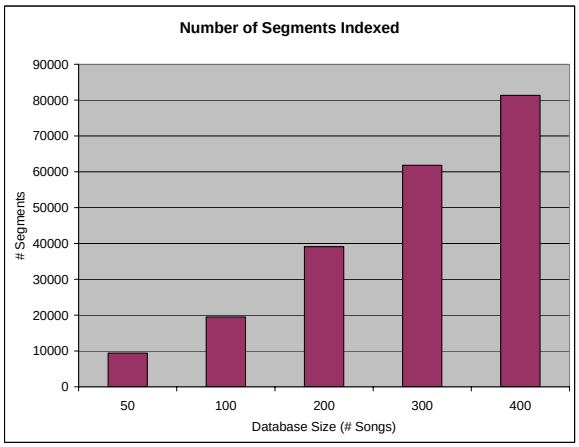

Figure 12: Total segments indexed as a function of the number of songs in the database.

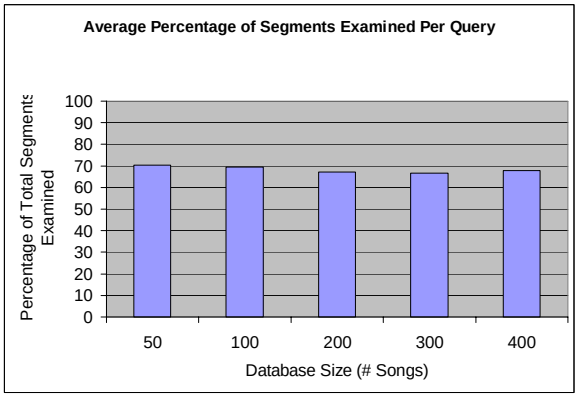

Figure 13: Average percentage of total segments examined per query as a function of the number of songs in the database.

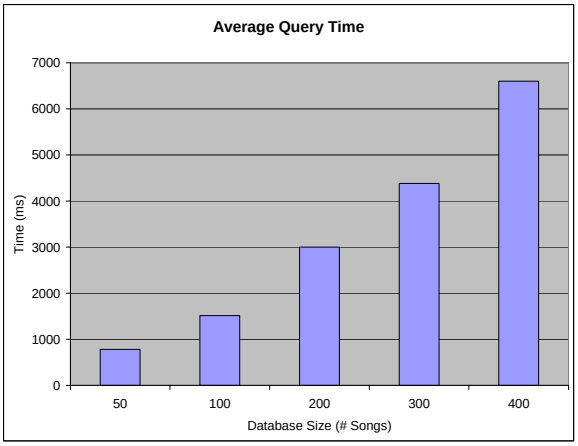

Figure 14: Average query time as a function of the number of songs in the database.

7. Conclusion

We have developed an end-to-end music search system called MusicDB with a UI that accepts humming samples as user queries and returns a ranked list of the top matching songs. Our system achieves reasonable performance, returning the desired song within the top 10 hits 52% of the time and as the top hit 24% of the time. In a second version of this system, we expect the search quality of MusicDB to improve drastically if we use a more solid midi parser than the buggy JMusic library we used and if we used a higher fidelity wave to midi converter.

Our main contribution in this paper is a detailed documentation of the steps involved in building a working query-by-humming system. We describe the channel selection and skyline algorithms involved in the extraction of melodies from polyphonic midis. We detail the representation of tunes and hums as time series, the time warping distance metric used in the research literature to perform similarity comparisons between time series, and an efficient indexing method to prune the search space and return a ranked list of results. We believe that our MusicDB demonstrates that query-by-humming systems are a promising new way to support music search.

8. References

[1] R. Agrawal, C. Faloutsos, and A. Swami, Efficient similarity search in sequence database. In *Proceedings of the 4th Conference on Foundations of Data Organization and Algorithm*, 1993.

[2] AKoff Sound Labs, Akoff music composer version 2.0. *http://www.akoff.com/music-composer.html*, 2000.

[3] A. Brown and A. Sorenson, JMusic: Music Composition in Java. *http://jmusic.ci.qut.edu.au*

[4] W. Chai. Melody Retrieval On The Web, Master thesis, 2001, *http://web.media.mit.edu/~chaiwei/papers.html*

[5] A. Ghias, J. Logan, D. Chamberlin, and B. Smith, Query By Humming – Musical Information Retrieval in an Audio Database. In *ACM Multimedia 95 – Electronic Proceedings*, 1993.

[6] K. V. R. Kanth, D. Agrawal, and A. Singh, Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *SIGMOD*, 1998.

[7] E. Keogh et. al. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*, 2001.

[8] E. Keogh et. al., Dimensionality reduction for fast similarity search in large time series databases. In *Journal of Knowledge and Information Systems*, 2000.

[9] E. Keogh and M. J. Pazzani, A Simple Dimensionality Reduction Technique for Fast Similarity Search in Large Time Series Databases. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications*, 2000.

[10] E. Keogh and C.A. Ratanamahatana, Exact indexing of dynamic time warping. In *Knowledge and Information Systems*, May 2004.

[11] Y. Kim, W. Chai, R. Garcia, B. Vercoe, Analysis of a contour-based representation for melody. In *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 1999.

[12] N. Kosugi, Y. Nishihara, T. Sakata, M. Yamamuro, and K. Kushima, A Practical Query-By-Humming System for a Large Music Database. In *8th ACM International Conference on Multimedia*, 2000

[13] L. Lue, H. You, and H. Zhiang, A New Approach To Query By Humming In Music Retrieval. In *IEEE International Conference on Multimedia and Expo,* 2001.

[14] Java Sound Demo, Sun Microsystems, Inc., *http://java.sun.com/products/java-media/sound/samples/JavaSoundDemo/*

[15] C. Myers, L. Rabiner, and A. Rosenberg, Performance tradeoffs in dynamic time warping algorithms for isolated word recognition. In *IEEE Transactions on Acoustics, Speech, and Signal Processing,* December 1980.

[16] Pandora, Music Genome Project, *http://www.pandora.com*

[17] S. Pauws, CubyHum: A Fully Operational Query by Humming System. In *ISMIR 2002, 3rd International Conference on Music Information Retrieval*, 2002.

[18] L. Rabiner, A. Rosenberg, and S. Levinson, Considerations in dynamic time warping algorithms for discrete word recognition. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1978.

[19] M. A. Raju, B. Sundaram, and P. Rao, TANSEN: A Query-By-Humming based Music Retrieval System, In *Proc. National Conference on Communications (NCC)*, 2003.

[20] H. Sakoe and S. Chiba Dynamic programming algorithm optimization for spoken word recognition. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1978.

[21] Spatial Index Library, Java v0.44.2b, *http://www.cs.ucr.edu/~marioh/spatialindex/*

[22] A. Uitdenbogerd and J. Zobel, Manipulation of Music For Melody Matching. In *ACM Multimedia, Electronic Proceedings*, 1998.

[23] WaoN – a wave-to-notes transcriber, Kengo Ichiki. *http://www.kichiki.com/WAON/waon.html*

[24] Y. Zhu and D. Shasha, Query by Humming: A Time-Series Database Approach. In *SIGMOD* 2003.