Music Databases: Indexing Techniques and Implementation*

Ta-Chun Chou, Arbee L.P. Chen and Chih-Chin Liu

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email: alpchen@cs.nthu.edu.tw

Abstract

In this paper, the music database with the search-by-content ability is studied. The *chords* are used to represent music. With the *chord*-representation model, the input fault tolerance ability is equipped. PAT-tree is proposed as the index structure. The "unstructured search" is an important characteristic of PAT-tree. We have implemented a music database system based on the *chord*-representation model and PAT-tree index structure.

1. Introduction

The requirement of managing various types of data in a database environment has drastically increased recently in accordance with the development of a variety of media. The music database systems try to manage music objects, such as MIDI files, WAV files, and other types of media associated with music. In managing music data, content-based retrieval is one of the major issues related to query processing. There are several requirements for content-based retrieval of music data. One of the requirements is "unstructured search", which means there is no restriction on the search points in music objects. Users are allowed to search from any position of music objects. Another requirement is the input fault tolerance. Sometimes the user's input does not coincide with the patterns in music objects. The music database system should be able to recognize similar patterns. Besides, efficiency and completeness are also important requirements in music databases. The last requirement is the saving of storage space. Due to the large amount of music data, the music database system should also consider the storage space for indexing music objects.

Little attentions have been drawn to the music database. Some researches focus on the storage and transcription of audio signals [4][8][9]. However, we need not only the storing of music objects, but also the querying of music objects. Other researches have been done with computer music, but only the audio signal processing is considered[10][11]. These researches recognize musical events from the audio signal, and establish the relationships between them. However, the representations and indexing techniques of music data are not explored. In this paper we consider how to provide the database with the ability of searching music by content.

To enable the music database with the search-by-content ability, we have to use meta data to represent the original music data. These meta data could be used to do the query processing. This job is similar to string searching, which finds all occurrences of a pattern in a text, where the pattern and the text are strings over some alphabets. There are some important algorithms for string searching[6]: the brute force algorithm, the Knuth-Morris-Pratt algorithm, different variants of the Boyer-Moore algorithm, the shift-or algorithm from Baeza-Yates and Gonnet, and the Karp-Rabin algorithm. In some cases, the texts are not exact, and we are looking for approximate patterns. The approximate string-matching problem is to find all substrings in a large text, which are close to the string under some measure of closeness. Many different approximate string-matching algorithms have been suggested[1][18].

All these algorithms focus on searching a string in one document. In the music database, we aim to search the whole database to find out all possible answers. To speed up the searching, we have to add additional data structures to index the database.

Jason[16] proposed a suffix tree for the sampling of user-specified patterns. A suffix tree is a Patricia-like tree constructed over all possible substrings of a text. A Patricia tree [15] is a digital tree where the individual bits of the keys are used to decide on the branching. A zero bit causes a branch to the left subtree, and a one bit causes a branch to the right subtree. Hence Patricia trees are binary digital trees. We use the suffix tree to index the music database.

This paper is organized as follows: we first discuss the music representation issues and our solutions in

* This work was partially supported by the Republic of China National Science Council under Contract No. NSC 85-2213-E-007-024.

Section 2. Next, we describe an indexing technique in Section 3. We briefly describe our implementation on this work in Section 4. Finally we give a conclusion in Section 5.

2. The Representation of Music Data

2.1 Music Framework

If the *staff* information of music objects is available, we can pose a music query against the staff information. The query condition may involve the static data of staffs such as tempos, beats, and keys. Besides, we can specify a short piece of music (*theme*) to retrieve the music objects containing similar piece of music. We call this kind of music querying as *query-by-theme*. When a music object is inserted into the music database, some pieces of the music object are specified as the themes of the music object. A theme consists of a *melody* and a *rhythm* which can be derived using pitch tracking techniques.

- *Melody*: the melody of a theme is the sequence of the pitches of all notes in the theme. For example, the melody of the theme of the Beethoven's Symphony No. 5 is "sol - sol - sol - mi - fa - fa - fa - re".
- *Rhythm*: the rhythm of a theme is the sequence of the durations of all notes in the theme. For example, the rhythm of the theme of the Beethoven's Symphony No. 5 is "1/2 - 1/2 - 1/2 - 2 - 1/2 - 1/2 - 1/2 - 4".
- *Chord*: Since we cannot expect that every user can pose a music query that exactly match one or more music objects in the music database, input fault tolerance should be provided. A chord consists of three (root, third, and fifth) or more notes which sound together in harmony. Representing a music object with chords instead of notes allows users to pose a music query with certain input faults.
- *Rhythm Signature*: A chord can be considered as a signature of a melody. It enables two melodies to be matched in an approximate way. Similarly, we can compose a rhythm signature for every rhythm such that approximate rhythm matching can be done.

Figure 2.1 shows a music framework. Class Music is an abstract class which models the special properties possessed by music objects. There are three attributes defined in the Music class. Attribute info, whose domain is the Music_Info class, stores the static music characteristics, key, beat, and tempo, of each music object. Attribute theme, whose domain is the Theme class, stores the themes in the music objects. In the music framework, a rhythm is an ordered list of durations of notes and a melody is an ordered list of pitches. The basic unit for recording the length of the duration of a note is a tick where 120 ticks equal to a quarter note. The value of a pitch is an integer ranging from 0 to 127 as defined in the MIDI specification. Attribute staff stores the staff of each music object.

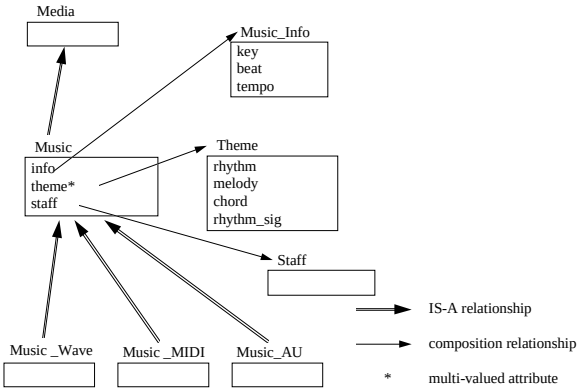

Figure 2.1 Music framework.

The Music class may have a number of subclasses which represent different formats of music objects. In the framework, we define three subclasses, i.e., Music_Wave, Music_AU and Music_MIDI to keep the music objects in Wave form, AU form and MIDI form, respectively. Note that to reflect the structural differences, some methods defined in the Music class may need to be redefined in these subclasses.

2.2 Music Representation Issues

It is not our goal to record music in detail, but to specify it in a rough and flexible way. This flexibility will benefit the processing of input faults. We cannot expect the users to always strike the right notes(suppose the input device is a keyboard). Moderate input faults should be allowed. There are four common types of input faults: duplication, elimination, disorder, and irrelevance. Duplication means a single note was input by the user more than once. For example, a *measure* with notes: do, do, do, mi, mi, sol was mistaken as the one with do, do, do, mi, mi, mi, sol, sol. mi and sol was duplicated. Elimination means a note that should have been input was missed. For example, a measure with notes: do, do, do, mi, mi, sol was mistaken as the one with do, mi, sol. do was missed twice and mi was missed once. Duplication and elimination often occur when a music *period* does not start from the beginning of a measure. Disorder means the order of a sequence of notes was disturbed. For example, a measure with notes: do, mi, sol, mi, do, mi, sol, mi was mistaken as the one with mi, do, mi, sol, mi, do, mi, sol. Disorder often occurs when the users are searching a music period of repeat. Irrelevant notes are the notes that have nothing to do with the music segment; they just appear for some reasons. For example, a measure with notes: do, do, do, mi, mi, sol was mistaken as the one with do, do, do, re, mi, mi, re, sol.

Therefore, to economically and efficiently represent the music is very important. *Figures* are minimal

elements that compose music. A figure could be as simple as containing several notes, for example, do-re-Me, sol-la-sol-Me, etc. From another point of view, a figure is a "shape" of a music segment. We use "shape" instead of rhythm, note, pitch, melody and so on, to represent music. Using the figure-representation method will save a lot of memory and disk space. In this paper, we use *chords* as figures to specify the music.

2.3 *Chord*

A *chord* is a combination of three or more notes which sound together in harmony. Figure 2.2 shows a few frequently-used chords.

C:	do,	mi,	sol
Dm:	re,	fa,	la
Em:	mi,	sol,	si
F:	fa,	la,	do
G:	sol,	si,	re
Am:	la,	do,	mi

Figure 2.2 Frequently-used chords.

Chords can be used to represent the musical properties of a song. Furthermore, chords are excellent representations that tolerate the user's input faults. For example, a measure with three notes: do, mi, and sol would be represented as a "C" chord while a measure with five notes: do, mi, mi, sol, and sol would also be represented as a "C" chord. These two measures have the same chord because they sound similar. As we have seen, the chord representation model can tolerate the missing or duplication of acoustically similar notes.

In addition, using chords in our music-representation saves a lot of storage resources, because we only record the corresponding chord of a measure instead of the notes, rhythm, or other information of the measure.

2.4 *Chord* Decision Algorithm

Different chord-sets may have different chord decision effects under a certain algorithm, and the effects dominate the search performance. We introduce a chord set(Figure 2.3), which is an extension of part of the traditional chord-set.

C1: do C2: do, mi C : do, mi, sol C7 : do, mi, sol, si
D1: re D2: re, fa Dm: re, fa, la Dm7: re, fa, la, do
E1: mi E2: mi, sol Em: mi, sol, si Em7: mi, sol, si, re
F1: fa F2: fa, la F : fa, la, do F7 : fa, la, do. mi
G1: sol G2: sol, si G : sol, si, re G7 : sol, si, re, fa
A1: la A2: la, do Am: la, do, mi Am7: la, do, mi, sol

Figure 2.3 A chord set.

With the chord-set, we decide the chords. The user-specified notes are divided into measures. For each measure we invoke the chord decision algorithm once. The chord decision algorithm is based on five principles:

1. Find out the chords that contain the most user-specified notes.
2. Preserve the minimal-length chords.
3. Preserve the chords whose roots have the maximal occurrence frequency.
4. Preserve the chords whose fifths have the maximal occurrence frequency.
5. Preserve the chords whose thirds have the maximal occurrence frequency.

Where "root" is the base note of a chord, "third" is the note which is three degrees larger than the root, and "fifth" is the note which is three degrees larger than the third.

For instance, the root of the "C" chord is do, third is mi, and fifth is sol.

In the first principle, we record all the chords that can be used to represent the measure of user-specified notes. In the second principle, we choose the "shortest" chords to represent a measure, because there is no need to use a longer chord to do it. The root is the base note of a chord. Therefore, in principle three, we use the occurrence frequency of the root to decide the chord. The fifth sounds most harmonic to the root. Therefore, in principle four, we use the occurrence frequency of the fifth to do our chord decision. The third sounds less harmonic to the root than the fifth, but still more harmonic than others. In principle five, we use the occurrence frequency of the third to decide the chord.

Example 1: Suppose a user inputs two measures of notes: |sol, mi, mi| fa, re, re|. For each measure, we apply the five principles to it.

Measure 1:

According to principle 1, the possible chords are: "C", "C7", "E2", "Em", "Em7", and "Am7". According to principle 2, the possible chord is "E2". We get the final answer: "E2".

Measure 2:

According to principle 1, the possible chords are: "D2", "Dm", "Dm7", and "G7". According to principle 2, the possible chord is "D2".

The user's input is then transformed into "E2, D2".

Example 2: Suppose a user inputs one measure of notes: |do, re, mi, do|. According to principle 1, the possible chord is "Dm7". The user's input is transformed into "Dm7".

The following chord decision algorithm gives a detailed description of the five principles:

```
Chord-decision algorithm(note_list)
 integer        i, candidate_number;
 Chord        *candidate_list;
begin
{step 1}
 i=0;
Candidate_list is initialized to include all the chords;
```

```
Sort note_list by the occurrence frequency of each
   note;
while (not end of note_list)
begin
   For the ith note in the note_list,
   find out the chords in the candidate_list , which
   contain this note;
   if (chords are found)
   begin
         Update candidate_list by the newly_found
         chords;
         candidate_number=the number of chords
   in the candidate_list;
    end;
    i=i+1;
end;
{step 2}
if (candidate_number>1)
then
begin
   Preserve the minimal-length chords in the
   candidate_list;
   candidate_number=the number of chords in the
   candidate_list;
end;
 {step 3}
 if (candidate_number>1)
 then
 begin
    Preserve the chords in the candidate_list,
    whose roots have the maximal occurrence
     frequency;
     candidate_number=the number of chords in the
     candidate_list;
 end;
  {step 4}
 if (candidate_number>1)
 then
 begin
       Preserve the chords in the candidate_list,
       whose fifths have the maximal occurrence
       frequency;
      candidate_number=the number of chords in the
       candidate_list;
 end;
  {step 5}
  if (candidate_number>1)
  then
  begin
      Preserve the chords in the candidate_list,
      whose thirds have the maximal occurrence
      frequency;
      candidate_number=the number of chords in the
      candidate_list;
   end;
     {step 6}
     if (candidate_number>1)
     then Choose the first entry in the candidate_list as
       the final result;
     {step 7}
     output the chord;
    end.
    where
     candidate_list : a linked list that contains chords
     candidate_number: the number of entries in the
candidate_list
     note-array: an array with twelve entries, each entry
contains a note and its  corresponding occurrence count
```

For example 1, suppose a user inputs two measures of notes: |sol, mi, mi|fa, re, re|. For each measure, we invoke chord_decision algorithm once.

Invokation 1:

Step 1: The candidate_list contains "C", "C7", "E2", "Em", "Em7", "Am7".

Step 2: The candidate_list contains "E2".

Step 3: Not used.

Step 4: Not used.

Step 5: Not used.

Step 6: Not used.

Step 7: Output "E2".

Invokation 2:

Step 1: The candidate_list contains "D2", "Dm", "Dm7", "G7".

Step 2: The candidate_list contains "D2".

Step 3: Not used.

Step 4: Not used.

Step 5: Not used.

Step 6: Not used.

Step 7: Output "D2".

Therefore, we transform the user's input into "E2, D2".

For example 2, suppose a user inputs one measure of notes: |do, re, mi, do|.

Invokation 1:

Step 1: The candidate_list contains "Dm7".

Step 2: Not used.

Step 3: Not used.

Step 4: Not used.

Step 5: Not used.

Step 6: Not used.

Step 7: Output "Dm7".

The user's input is transformed into "Dm7".

We do not discuss music theory in detail. There are many ways to do the chord decision, and we just provide a simple one to show our approach.

2.5 The Input Fault Tolerance Ability of the Chord-Representation Model

There is a trade-off between the input fault tolerance ability and the correctness of results.

Basically, we should tolerate reasonable input faults to facilitate the user's queries. In another aspect, we do not want to find out too many irrelevant songs, even if the target song is included.

Regarding the music data, there is a meaningful criterion: two music measures are considered the same, if they sound similar. The chord representation model is based on this criterion. Each chord represents an acoustic feeling, which is used to classify the music. If the user's input faults are not exorbitant, we can find a group of chords that include the target. The more measures of notes the user inputs, the smaller group we can get.

The chord-representation model is tolerant of the preceding four types of input faults. In the example of duplication, both the two measures, |do, do, mi, mi, sol| and |do, do, do, mi, mi, mi, sol, sol|, will be represented as the "C" chord. In the example of elimination, both the two measures, |do, do, do, mi, mi, sol| and |do, mi, sol|, will be represented as the "C" chord. In the example of disorder, both the two measures, |do, mi, sol, mi, do, mi, sol, mi| and |mi, sol, mi, do, mi, sol, mi, do|, will be represented as the "C" chord.

3. The Indexing Technique

3.1 The Index Structure

One of the most important features of music searching is "unstructured search", which means one can search from any point in a song, not only the beginning or the theme of the song. This is different from the traditional model of text searching.

The traditional model of text searching is based on keywords with relevance weights. However, for our music database, there is no basic structure of keywords which can be extracted from the songs, and queries are based on any point of the songs, not only the keywords. Therefore, we need a data structure that allows very efficient searching.

3.2 Using PAT-tree as Index Structure

In our approach, the problem of music query processing is transformed into that of inexact substring matching using PAT trees. A PAT-tree([5]) is a Patricia-like tree constructed over all possible substrings of a text. This structure was originally described by Gonnet[7] in the paper "Unstructured Data Bases". An external node (denoted as a box) in the PAT-tree of a string represents one of its substrings. The internal nodes (denoted as circles) are the individual characters of the substrings, which are used to decide on the branching. Figure 3.1 shows an example of a PAT-tree for the text “ababc”. The PAT tree has five external nodes which represent the substrings “ababc”, “babc”, “abc”, “bc”, and “c”. To find whether a given substring is contained in the text, each character of the substring is used to traverse the PAT tree. If the traversing stops at an internal node n without any character mismatch, the leaves of the subtree rooted at node n will be our answers after removing duplications. Otherwise the text does not contain the given substring.

text : ababc

substrings : ababc, babc, abc, bc, c

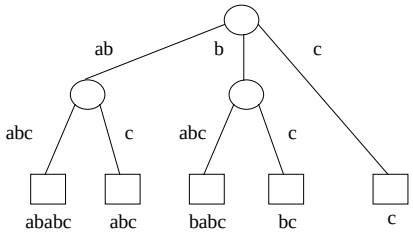

Figure 3.1 A PAT-tree example.

The PAT-tree is applicable to our music-indexing technique. Figure 3.2 shows an example of a PAT-tree with four music *periods*, i.e, S1, S2, S3, and S4. Each *period* is a *chord*-representation of the original *measures* of a song. The music period is the basic unit to retrieve. The procedure for constructing the PAT-tree of a music period will be expained in the next subsection.

S1: Am F2 Dm Am
S2: C C F C
S3: G Em C D
S4: E1 G Am Bm

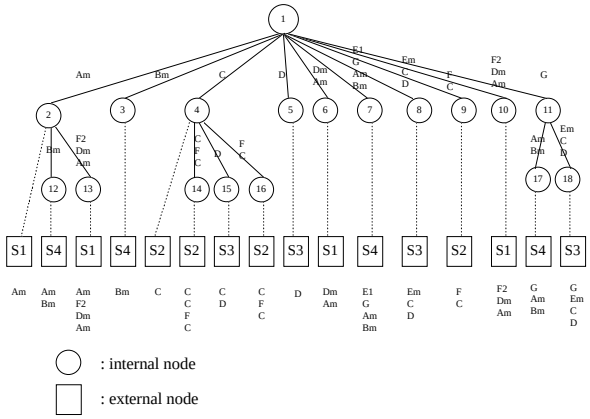

Figure 3.2 A PAT-tree with four periods.

Note that a chord is viewed as a single character in a substring. The external nodes in this PAT-tree are substrings of the music segments. The internal nodes

are the individual chords of the substrings. Besides, for each leaf, there is a name-set attached to it. The name-set is used to record the names of the segments which contain the substring.

3.3 The Operational Commands for the PAT-tree

As for the processing of the user's commands, we consider insert, search, and delete.

3.3.1 Insert

It is a three-step process to insert a music object into the PAT-tree. First, each music object is converted into the chord-representation by the chord-decision algorithm. Second, we decompose the chord-representation into substrings. Finally we add them to the PAT-tree.

For example: the user inserts a music object (S5) with four measures of notes: |do, do, sol, sol|la, la, sol|fa, fa, mi, mi|re, re, do|.

Step 1: The music object is converted into "C, Am7, F7, Dm7".

Step 2: The chord-representation is decomposed into substrings: C Am7 F7 Dm7, Am7 F7Dm7, F7 Dm7, and Dm7.

Step 3: The substrings are added to the PAT-tree. For each substring of chords to be inserted, we traverse the PAT-tree and stop at the node such that the path from the root node to the node will match the prefix of the chord string. We construct an internal node with the suffix of the chord string and link it to the stop node and a new external node attached with the chord string. Figure 3.3 shows the final result.

3.3.2 Search

In the music database, the search command usually serves as a "range searching". It means to find all the songs that at least have a segment which sounds like what the user inputs. For example, suppose a user wants to listen to a song ("Little Star"). At this moment, these notes: |do, do, sol, sol|la, la, sol| can be input to search all the "Little Star"-like songs. Then, the list of notes is transformed into the list of chords, which is used to traverse the PAT-tree. Under general conditions, we usually stop at an internal node, and the leaves of the subtree rooted at the current node are our answers. Finally we display the qualified songs after removing duplications. For example, the user inputs a measure, |do, do, sol, sol|, to find "Little Star"-like songs. We first transform the measure into a "C" chord, then use it to traverse the PAT-tree in figure 3.3. We stop at node five, and the leaves of the subtree rooted at node five are our answers after removing duplications. The search results are S2, S3, and S5.

3.3.3 Delete

To delete a song in a PAT-tree, the first step is converting the song into a chord-representation. Next, we decompose the chord-representation into substrings, and in step three we use them to search the PAT-tree. Every search for a substring will stop at a leaf. For each qualified leaf we have to check if its name-set contains the song to be deleted. If it does, we remove the name in the name-set. A leaf node is removed when its name-set is empty.

For example, the user wants to delete S5 in figure 3.3.

Step 1: We transform S5 into "C, Am7, F7, Dm7".

Step 2: The chord representation is decomposed into substrings: C Am7 F7 Dm7, Am7 F7Dm7, F7 Dm7, and Dm7.

Step 3: For each substring, we search the PAT-tree and delete the corresponding leaf.

Finally we get the PAT-tree in Figure 3.2.

3.4 Implementation Issues of PAT-tree

We can implement the PAT-tree as a "Patricia tree"[13], but that means we must deal with a lot of linked lists and the garbage-collection problem. There is another simpler implementation of PAT-tree, which was independently discovered by Manber and Myers[12] , who called the structure suffix array.

Nevertheless, both the two data structures may cause the PAT-tree unbalanced after a series of insertions and deletions. Therefore, we propose B+tree for the implementation of PAT-tree. Figure 3.4 shows a B+tree implementation of a PAT-tree. In this example, the bucket size is four.

S1: Am F2 Dm Am
S2: C C F C
S3: G Em C D
S4: E1 G Am Bm

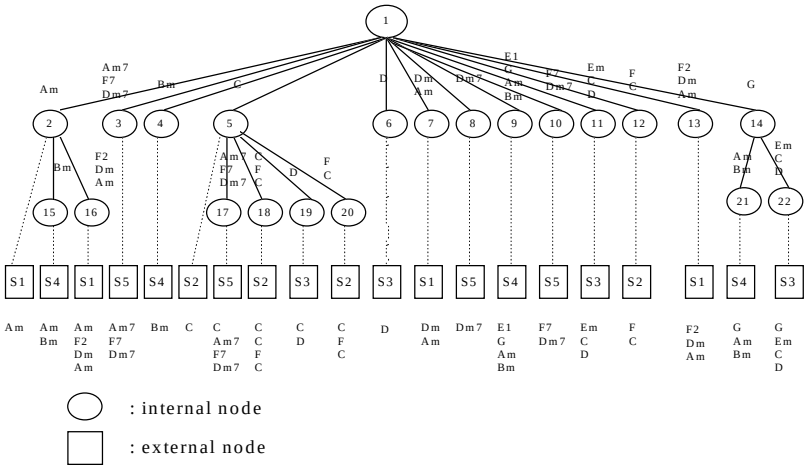

Figure 3.3 The PAT-tree after inserting period S5.

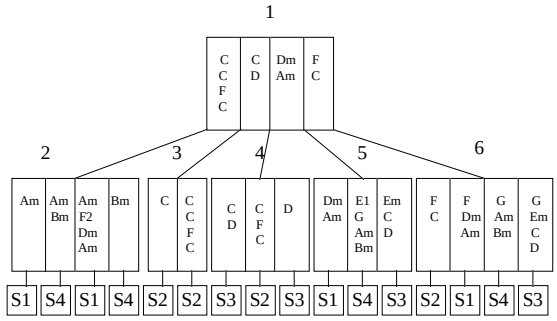

Figure 3.4 A B+tree implementation of a PAT-tree.

The B+tree implementation is excellent for "range searching". We do not have to traverse the whole subtree to find the answers. We just need to find the first substring, and sequentially check the substrings in the neighbor buckets. It saves a lot of disk accesses and seek time when we do a range search. For example, if we want to find the songs that have "F" *chord*. First we traverse to node one, and discover that "F" is prefix-matched to "F C". Then we directly go down to "F C" in node six, and sequentially prefix-match the substrings in buckets one and two. In this example, we get S1 and S2 as the answers.

4. Implementation of Music Database System

We have implemented a music database system based on our music representation model and indexing technique. Figure 4.1 shows a music query interface. It provides three ways for users to pose their music queries. For users who can play piano, they can play a piece of song using the MIDI keyboard. The MIDI messages generated by the MIDI keyboard will be transformed into a rhythm and a melody. For naive users, they can sing a piece of song using the microphone. Applying pitch tracking techniques (zero-passing or discrete cosine transforming) on the song, its rhythm and melody can be recognized. Users can also draw the staff of a song using the GUI. Since that it is difficult for a user to sing a piece of song that exactly matches the songs to be retrieved, certain errors are admittable. In our system, the number of notes in a music query can be more (insertion error) or less (deletion error) than that in a result song. The key of a music query can be higher or lower than that of a result song (transposition error). The tempo of a music query can be faster or slower than that of a result song (tempo scaling error). No matter which way the users select to pose a music query, the music query is transformed into the corresponding melody and rhythm for further processing. The melody of the music query is then transformed into a string of chords which is used to traverse the PAT-tree to find the answers.

Figure 4.1 The music query interface.

5. Conclusion

In this paper, we introduce the chord-representation model that flexibly specifies the music to allow the user's input faults. Besides, we provide the

"unstructured search" ability by using the PAT-tree as the indexing structure. There are still many problems that we have not studied. One of the problems is when the users mistakenly rise the pitch to a higher or lower key, the search operation will repeal. Basically, we can automatically key up the user's input to search the PAT-tree. However, we will spend a lot of time in searching the irrelevant songs , and get a lot of useless data. This is not a practical solution. We must alter the representation model to solve this problem.

Another puzzle is the effectiveness of the chord-set. As we mentioned earlier, the chord-set decides the search performance, and our chord-set is aimed at the pop-music. However, with classical music, this chord-set may not fit very well. We have to polish our chord-set to apply to the classical music. Besides, the analysis of the search performance is also an important future work.

The WAV and AU media types of the search tool need a pitch-converting interface to generate the information about notes. Basically there are two solutions, one is to input the note information manually. That is, we have to listen to the music and key in the information. It does not seem to be a good solution. The other solution is to input the note information automatically. That is, we use a pitch-converting device, software or hardware, to get the note information. It is an important research problem in audio signal processing.

References

[1] Baeza-Yates R. and G. H. Gonnet, A New Approach to Text Searching. *COMMUNICATIONS OF THE ACM*, Vol. 35, No. 10, October 1992.

[2] Chafe C. and D. Jaffe, Source Separation and Note Identification in Polyphonic Music. Technical Report No. STAN-M-34.

[3] Dannenberg R. B., Music Representation Issues,Techniques, and Systems. *Computer Music Journal*, 17:3, pp. 20-30, Fall 1993.

[4] Feiten B. and S. Gunzel, Automatic Indexing of a Sound Database Using Self-organizing Neural Nets. *Computer Music Journal*, 18:3, pp.53-65, Fall 1994.

[5] Frakes. W. B. and R. A. Baeza-Yates, *Information Retrieval*. Prentice Hall, 1992, chapter 5.

[6] Frakes. W. B. and R. A. Baeza-Yates, *Information Retrieval*. Prentice Hall, 1992, chapter 10.

[7] Gonnet G., Unstructured Data Bases or Very Efficient Text Searching. In *ACM PODS*, vol. 2, pp. 117-24, Atlanta, 1994.

[8] Katayose H. and S. Inokuchi, The Kansei Music System. *Computer Music Journal*, Vol. 13, No. 4, Winter 1989.

[9] Katayose H. and S. Inokuchi, The Kansei Music System '90. In *Proc. of ICMC GLASGOW* 1990.

[10] Kuhn W., A Real-Time Pitch Recognition Algorithm for Music Applications. *Computer Music Journal*, Vol. 14, No. 3, Fall 1990.

[11] Lent K., An Efficient Method for Pitch Shifting Digitally Sampled Sounds. *Computer Music Journal*, Vol. 13, No. 4, Winter 1989.

[12] Manber, U., and G. Myers, Suffix Arrays: A New Method for On-line String Searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 319-27, San Francisco, 1990.

[13] McCREIGHT E. M., A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*, Vol. 23, No. 2, April 1976, pp.262-272.

[14] Polansky L., Live Interactive Computer Music in HMSL, 1984-1992. *Computer Music Journal*, 18:2, pp. 59-57, Summer 1994.

[15] Sedgewick R., *Algorithms in C*. Addison-Wesley, 1990, pages 253-257.

[16] Wang J. T. L., G. W. Chirn, T. G. Marr, etal, Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results. *In Proc. of the 1994 ACM International Conference on Management of Data*, pages 115-124, 1994.

[17] Written I. H., L. C. Manzara, and D. Conklin, Comparing Human and Computational Models of Music Prediction. *Computer Music Journal*, 18:1, pp. 70-80, Spring 1994

[18] Wu S. and U. Manber, Fast Text Searching Allowing Errors. *COMMUNICATIONS OF THE ACM*, Vol. 35, No. 10, October 1992.

[19] Yan T. W. and H. Garcia-Molina, Index Structures for Information Filtering Under the Vector Space Model. In *Proc. of the 10th International Conference on Data Engineering*, pages 337-347, 1994.