

CROWDSOURCING A REAL-WORLD ON-LINE QUERY BY HUMMING SYSTEM

Arefin Huq

Northwestern University
EECS Department
2133 Sheridan Road
Evanston, IL 60208, USA
fig@arefin.net

Mark Cartwright

Northwestern University
EECS Department
2133 Sheridan Road
Evanston, IL 60208, USA
mcartwright@u.northwestern.edu

Bryan Pardo

Northwestern University
EECS Department
2133 Sheridan Road
Evanston, IL 60208, USA
pardo@northwestern.edu

ABSTRACT

Systems able to find a song based on a sung, hummed, or whistled melody are called Query-By-Humming (QBH) systems. Tunebot is an online QBH web service and iPhone app that connects users to the desired recording on Amazon.com or iTunes. Tunebot's searchable database is composed of thousands of user-contributed melodies. Melodies are collected from user queries, sung contributions and through contributions from on-line play of an associated iPhone Karaoke game: Karaoke Callout. In this paper we describe the architecture and workings of the paired systems, as well as issues involved in building a real-world, working music search engine from user-contributed data.

INTRODUCTION

Music audio is one of the most popular categories of multimedia content. Examples include the song repositories of Apple's popular iTunes (www.apple.com/itunes), the indie-music site CD Baby (www.cdbaby.com) and Amazon (amazon.com). These music collections are indexed by such metadata as title, composer, and performer. Finding the desired recording with this indexing scheme can be a problem for those who do not know the metadata for the desired piece.

If the user has access to a recording of the desired audio (e.g. it is currently playing on the radio), then an audio fingerprinting system, such as Musiwave [1] or Shazam [2] can be used. Such systems require the query example be a (possibly degraded) copy of the exact recording desired. This makes audio fingerprinting unsuitable for any situation where the user is unable to provide a portion of the exact recording sought (e.g. the song ended on the radio before a search could begin).

Another approach is to identify a song based on entering its lyrics into a standard text-based search engine. This is a relatively mature field with successful commercial search engines (e.g. Google) already available. It is not, however, applicable to pieces of music that have no lyrics, or in situations where the user remem-

bers the melody but not the words.

In this work, we concentrate on the situation where the user queries a system by singing or humming some portion of the song ("What is the name of the song that goes like this?"). Song identification systems that take sung or hummed input are known as query-by-humming (QBH) systems [3-4]. These are an example of melodic search engines. Melodic search engines (including QBH and rhythmic search) have received much attention in recent years [5-14] and use a melodic fragment as a query, entered as musical notation, through a virtual piano keyboard or sung into a microphone.

Most published research in QBH has focused on the matching algorithms and distance measures for melodies. While this is important, there are other technical and scientific challenges that must be surmounted to build an effective QBH system ready for real-world deployment. Example issues include: creation of a large database of relevant search keys, handling large numbers of users, speeding search as the database goes from hundreds to hundreds of thousands of melodies, and updating the database and matching algorithms after deployment in a seamless way.

Our solutions to these problems are embodied in *Tunebot*, an online QBH web service that connects users to the desired recording on Amazon.com or iTunes. Tunebot's searchable database is composed of thousands of user-contributed melodies. Melodies are collected from user queries, sung contributions and through contributions from on-line play of an associated Karaoke game: *Karaoke Callout*. In this paper we describe the architecture and workings of the paired systems, as well as issues involved in building a real-world, working music search engine from user-contributed data.

TUNEBOT

We embody our solutions to the problems of real-world QBH in an on-line web service called Tunebot (tunebot.org). Tunebot lets the user search for music by singing a bit of it (with or without lyrics) as a query.

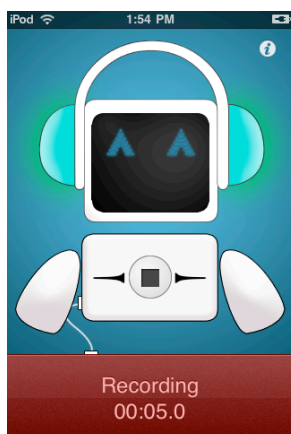
The system does not require hand-coded search keys, since it automatically updates the database with new search keys derived from user queries and contribu-

tions. To speed data collection and encourage collaborative participation from the public, we integrate Tunebot with an online social music game (Karaoke Callout) that encourages collaborative tagging of audio with new search keys [15]. Karaoke Callout is a Game With A Purpose [16] that helps build our knowledge base of songs. Users may also register with our website and freely contribute sung examples in a manner similar to the OpenMind initiative [17].

User Interaction

Tunebot is available as a web service and is currently in beta testing as an iPhone application. The user interaction in both the web and iPhone versions is identical: 1) Sing, 2) Choose. The user simply sings a portion of the desired song to Tunebot. The system returns a ranked list of songs. Each song is playable by a simple click. While the song is playing, the system presents a dialog box asking if this is the correct piece of music. If the user clicks “yes,” the query is stored in our database as a searchable example for that song. The user is then connected to either Amazon.com or iTunes where the music may be purchased. Figure 1 illustrates this interaction on the iPhone version of Tunebot. Figure 2 illustrates the Flash-based web interface for Tunebot.

1. Sing



2. Choose

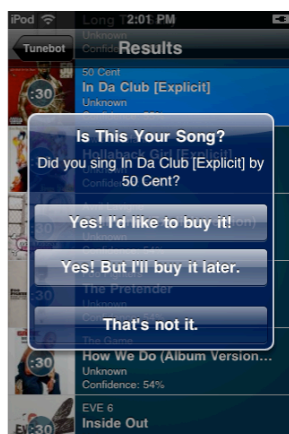


Figure 1. Screen shots of the iPhone interface for Tunebot.

Searchable Database Construction

Creating searchable keys that can be queried by singing is non-trivial. Hand-keying a database with thousands or millions of documents consumes prohibitive amounts of effort, as does updating and vetting such a database as music is created and tastes change. Thus, it is important to develop good methods to create and vet perceptually relevant search keys for music audio that allow the creation of a large music database indexed by musical content. This database must be expandable after deployment so the system may search for new music introduced as time goes by. For a system to scale, this must be done with minimal training and minimal oversight by human operators.

We do not currently use existing MIDI files or extraction of melodies from the original polyphonic audio. Automated transcription of polyphonic commercial recordings is still not sufficiently robust to provide good searchable melodies. The symbolically encoded databases available to us do not provide the coverage of modern pop, and rock tunes that our users tend to search for. Further, as user tastes change and new songs are released, a real working system must have the ability to constantly add songs to the database after deployment. We address these issues by turning to the users of the system for contributions.

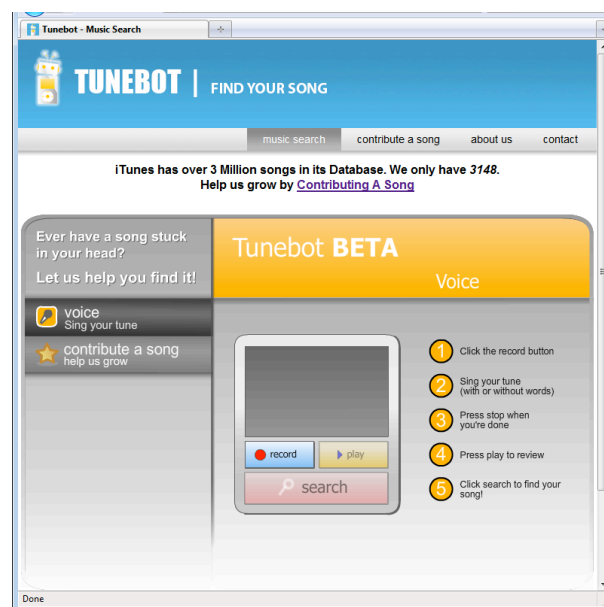


Figure 2. Screen shot of the Flash-based web interface for Tunebot.

The database for Tunebot uses searchable melodic keys derived from *a cappella* performances contributed by users as a result of playing Karaoke Callout, through use of the Tunebot search engine, and by logging in as a contributor and singing melodies to the system. Search keys are encoded as described in the section *Matching and Encoding*.

As of this writing, the Tunebot database contains roughly 11,000 examples for over 3,100 songs. Nearly 900 songs have 5 or more examples associated with them, and over 100 songs have 10 or more examples. The database is constantly growing as users contribute new songs and new examples for existing songs. To illustrate the rate of growth of the database, 5,053 examples representing 1,017 new songs were added to the database in the period from January 1, 2010 to April 15, 2010. At the current rate of growth, the size of the database should more than double by the end of this year compared to its size at the end of 2009.

System Overview

The Tunebot architecture is divided into three parts: (1) the client, (2) the server-side front-end, and (3) the

server-side back-end. These components are shown in Figure 3.

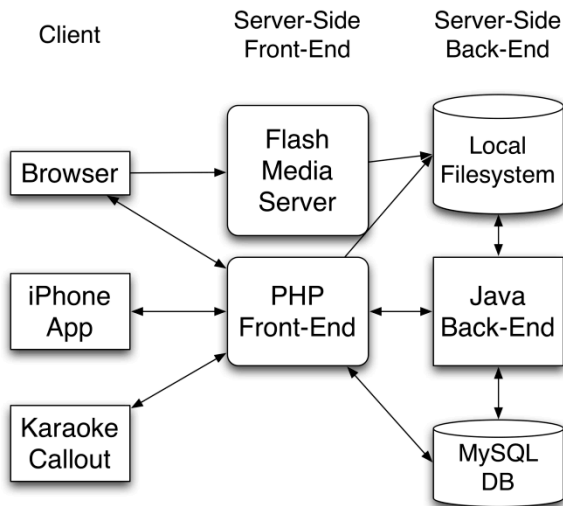


Figure 3. An overview of the Tunebot system.

The client-side is most typically a web browser. In this scenario, a Flash plug-in runs on the client side in the browser to record the audio of user queries and contributions and send it to the server.

The server-side front-end consists of two parts: (1) a set of PHP scripts served by an Apache Web Server, and (2) the Flash Media Server. The server-side front-end is responsible for presenting the user interface to search for and contribute songs, managing user information, and passing requests and audio files to the back-end. The iPhone client under development does not interact with the Flash Media Server, instead communicating with the server only through a PHP front end, as illustrated in Figure 3.

The server-side back-end is built around a Java servlet, running in Apache Tomcat. The back-end implements the matching algorithm and computes similarity rankings of submitted queries. Both the front and back ends interact directly with the SQL database on the server.

Encoding Melodies

Before a melodic comparison takes place, our transcriber estimates the fundamental frequency of the singing every 20 milliseconds. The note segmenter then divides this series of estimates into notes [18]. We encode all queries and all melodies in the database as sequences (strings) of note intervals. Each note interval is represented by a pair of values: the pitch interval (PI) between adjacent notes (measured in units of musical half-steps) and the log of the ratio between the length of a note and the length of the following note (LIR). Note lengths are defined to be inter-onset-intervals. We use note intervals encoded in this way because they are transposition invariant (melodies that differ only in key appear the same) and tempo invariant (melodies that differ only in tempo appear the same). We represent a

melody X as a string of note intervals. The encoding of a sung example into note intervals is illustrated in Figure 4.

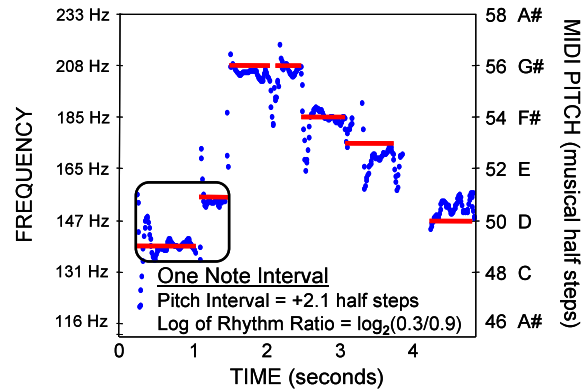


Figure 4. Pitch tracking and encoding of a sung example. Dots are pitch estimates. Horizontal lines are segmented notes. One note interval is shown in the rounded square.

Measuring Distance Between Melodies

Equation 1 defines a simple metric between note intervals x and y , with pitch intervals x_p and y_p and LIRs x_l and y_l .

$$d(x, y) = a|x_l - y_l| + b|x_p - y_p| \quad (1)$$

Here, a and b are non-negative weights chosen to optimize performance on a set of example queries for a given database of songs. Of course, when searching in a melodic database, one is not comparing individual note intervals, but full melodies. To compare melodic strings, we use edit distance [19].

The edit distance between two strings is the cost of the least expensive way of transforming one string into the other. Here, transformation cost (a.k.a. match cost) depends on the comparison function for the individual string elements described in Equation 1. We have a fixed insertion/deletion cost of one, effectively forcing the other parameters to be in these units.

This simple approach, when paired with a differential melodic encoding like our note-interval representation (this encoding is crucial to the use of such a simple note metric), has been shown to produce comparable search performance to more complex distance measures, without the need to optimize many parameters [4].

Each song in the database is represented by one or more sung melodies (search keys). A song's ranking in the search results is determined by the distance between the query and the nearest search key for that song.

Direct comparison of the query to every melody in the database becomes prohibitively slow as the size of the collection increases. If the comparison function for string elements is a metric (like Equation 1) then edit distance can also be made a metric [19]. Placing database melodies in a metric space allows efficient search

of a melodic database using vantage point trees [20, 21].

Vetting the Database

When a user queries for a particular song (e.g. “Lola”), we consider a search successful if the correct target song is returned as one of the top answers. The closer the target gets to number one, the better the system performance. When a single search fails, it may be difficult to tell exactly why. The query may be poorly formed (singing “Hey Jude” when searching for “Lola”), the search method may be ineffective for a particular user (perhaps a user model needs optimization), or the individual search key may not correspond well with what a typical person would sing (storing only the verse when people sing only the chorus). Maintaining a database of past queries and their associated targets makes it possible to distinguish between cases and react appropriately.

Each row in Table 1 corresponds to a query made to a search engine. Here, “Query Audio” is the recorded singing, “Target Title” is the title of the correct target in the database and “Target Rank” is the rank of the correct target in the results returned by the system. In this example, every query by User 1 failed to place in the top ten. This is an indication that the search engine is not optimized properly for this user. Note also that every query for “Hey Jude” failed to place within the top fifty, regardless of user. This indicates a mismatch between the target and the kinds of query example users provide. This is in direct contrast to both “Que Sera Sera” and “Lola,” each of which has one query whose correct target was ranked first.


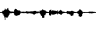


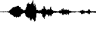


User	Query Audio	Target Title	Target Rank
1		Hey Jude	190
1		Que Sera Sera	39
1		Lola	21
2		Hey Jude	233
2		Lola	1
3		Hey Jude	59
3		Que Sera Sera	1

Table 1. Examples in a database

Our searchable database is composed of sung examples, keyed to correct song titles. This lets us automatically vet our search keys by using them as example queries. Those targets with below-average search results can then be tagged for search key updating. Such a database also allows for principled, automatic improvement of our similarity measures, as described in the section *System Optimization*.

System Optimization

Recall that searchable keys in the database are generated from past queries, sung contributions and examples of singing from Karaoke Callout. Each sung example is a potential new search key. The effectiveness of this new key can be measured by rerunning saved queries against this new key. This can be repeated using a key based on each query (or even on the union of all queries) and the best new key may then replace or augment the original search key for a particular song. This allows automatic, constant updating and improvement of the database without need for expert intervention.

A primary measure our system optimizes is mean reciprocal right rank (MRRR), shown in Equation 2. The right rank of a query is the rank of the correct song for the query. We refer to the correct song as the *target*. The mean right rank for a trial is the average right rank for all queries in the set.

$$MRRR = \frac{\sum_{n=1}^{\text{numberOf Queries}} \frac{1}{\text{rightRank}_n}}{\text{numberOf Queries}} \quad (2)$$

We use MRRR because it gives more useful feedback than the simple mean right rank. Consider the following example. System A returns right ranks of 1, 1, 199, and 199 for four queries. System B returns 103, 102, 98, and 97. We prefer a system that ranks the correct target 1st half of the time to one that ranks it around 100th every time. Mean right rank returns a value of 100 for both systems. MRRR returns 0.5 for system A and 0.01 for system B.

When vetting search keys, one need only measure reciprocal right rank for each search key in the database. When this falls below a given value, it becomes a candidate for removal or replacement, as described above.

Similarly, we use MRRR as the measure of the effectiveness of a melodic similarity measure. We currently use the simple edit-distance melody metric described in a previous section because it allows the application of vantage-point trees to speed search. This metric, however, does have tunable parameters that let us weigh the relative importance of pitch and rhythm in melody matching. Our system allows re-tuning of the weight of such parameters after deployment, as the composition of the database and the user queries shift over time [18].

The relative importance of rhythm and pitch are characterized by the parameters a and b , respectively, in Equation 1. The *rhythm weight* is a , and b is referred to as the *pitch weight*. We cannot know *a priori* what values should be given to these parameters, so these values must be determined empirically. It also seems natural to wonder if different values would be appropriate for different individuals, depending on how accurate a given individual’s singing is with regard to rhythm or pitch.

To explore these issues we first determined generally applicable values for these parameters by optimizing MRRR with respect to these parameters over a subset of the database. This process yielded an optimal value of 0.5 for rhythm weight and 0.005 for pitch weight. (These values only have meaning relative to the corresponding units used in Equation 1.) These values were set as the default parameter values of the system.

Next we collected a large number of labeled queries for a set of four heavy users of the system (more than 512 queries per user) and computed MRRR over a wide range of rhythm weight and pitch weight values. This served two purposes: to validate our choice of default parameter values, and to determine the importance of tuning these parameters per user. Note that this second set of queries, and the users who provided them, were not part of the initial optimization of the parameter values and so this constitutes a proper validation. Table 2 contains an illustrative excerpt of the analysis.

User	Best Rhythm Weight	Best Pitch Weight	Best Individual MRRR	% MRRR change from best global settings
1	0.400	0.00450	0.4760	+2.1%*
2	0.475	0.00450	0.4412	+0.9%*
3	0.525	0.00425	0.4065	+2.4%*
4	0.475	0.00500	0.3771	+2.4%*

Table 2. Optimal pitch and rhythm weights. Here, * means *not statistically significant*.

Each row of the table shows the result of optimizing MRRR with respect to rhythm weight and pitch weight for the given user. In each case the optimization was done over 512 labeled queries using a grid search with 17 points in each dimension and a granularity of 0.025 for rhythm weight and 0.00025 for pitch weight. This gave a total of 289 parameter value pairs tried for each user. The % change in MRRR is measured with respect to the MRRR achieved using the default rhythm and pitch weights learned from an earlier set of singers and examples.

The values shown for MRRR are based on an early 2010 snapshot of our constantly-growing database of real-world, user-contributed sung examples. For this experiment, all contributions from the singer for whom we optimize the values were removed prior to testing, as were all anonymous contributions to the database. This was done to ensure no contributions by the singer in question were used as searchable targets. Therefore the size of the test database depends on the number of contributions by the singer in question. The MRRR reported for User 1 was based on the largest resulting data set (5302 contributions representing 1919 unique songs). The data set for User 3 was the smallest (4556 contributions representing 1730 unique songs).

Several observations are possible from this table. The parameter values that result from optimizing per user are fairly close to the defaults learned from a large set of earlier singers. The optimal rhythm weight is within one grid point in three of four cases and the optimal pitch weight is within two grid points in three of four cases. More importantly, the improvement in MRRR from optimizing these parameters is quite small. In fact, it is less than 3% of the MRRR for each singer when using the default global parameter values learned from another set of singers. This difference is not statistically significant when taking into account the variance of MRRR on a random sample of 512 queries.

On the basis of this data and on similar analysis of other users, we are confident that our empirically determined global defaults for rhythm and pitch weights are valid and robust across a wide range of users, in the context of the current algorithm and the current composition of the database. Given the robustness of the default settings it appears that personalization in this parameter space is not necessary. However, our system contains several other parameter spaces and algorithmic choices where the importance of personalization has not yet been explored.

KARAOKE CALLOUT

In order to bootstrap the creation of a paired singing-example/target database and encourage user participation, we take a page from recent work in participatory and collaborative tagging. Particular inspirations include Open Mind [17] the ESP Game [16] and Karaoke Revolution (a popular video game released by Konami for the Sony PlayStation 2).

These examples have inspired us to cast system training in the form of a prototype interactive, client-server karaoke game: *Karaoke Callout*. This game closes the loop between system use and system improvement by providing correct song labels for sung examples, so that we can automatically vet and update a musical search engine.

An initial prototype of this game was originally developed for Symbian-OS phones [15]. Since creating the initial Symbian prototype, we have developed a new iPhone version of the game that is in beta testing with a small group of users. Those interested in becoming testers or in receiving notification of the final release of the game are encouraged to contact the authors of this paper.

The Karaoke Callout Game Interaction

The flow of Karaoke Callout proceeds as follows. Player 1 selects a song from our constantly growing database and sings it into the phone. While singing, the player is provided the lyrics to the song (Figure 5, step 1). If the player has the selected song in their iPod music library, then they have the option to sing along as the original recording plays.

Once the player is done singing, the audio is sent to the Tunebot music search engine, which rates the quality of the singing by measuring how closely it resembles the nearest melodic key for that song in the server database, sending a score back to the user (Figure 5, step 2)

Player 1 may then challenge another person to beat their score. If that person is a registered Karaoke Callout user, Player 1 needs to only provide the callout recipient's username, and they will be notified of the challenge via a push notification on their phone (Figure 5, step 3). If Player 1 wishes to invite a new person to play, they can select any email address (their phone contact list is provided as a convenience) and a mail will be sent to that person explaining how to install and play Karaoke Callout.

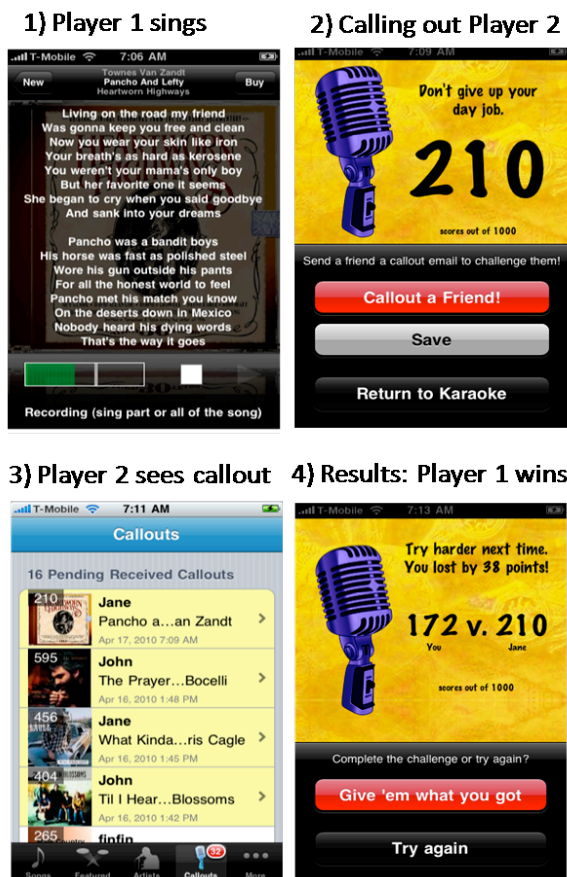


Figure 5. Screen shots of the iPhone interface for Karaoke Callout.

To accept the challenge, the callout recipient (Player 2) sings the song, attempting to better the performance of the challenger. The players are then notified of the results (Figure 5, step 4). This process may then be repeated, with either party selecting a new song with which to "call out" the other party. Over the course of an interaction, numerous examples of each party's singing are created and stored in our database.

Karaoke Callout System Architecture

The game server (see Figure 6) is divided into three main components. The first of these is the Karaoke

Server (written in PHP), which handles communication with the clients, queries the Singing Scorer (our music search engine) and stores sung examples in the database. The final component is a SQL database of user accounts, audio queries, scores, and challenges. In addition to our server, the Apple Push Notification Service is also in the loop in order to communicate with the users when the game is not running. The Singing Scorer is modular and separate from the Karaoke Server, allowing each component to be updated independently. This is key for implementing automated system personalization and learning, as the Singing Scorer is the search engine that we wish to optimize (Tunebot).

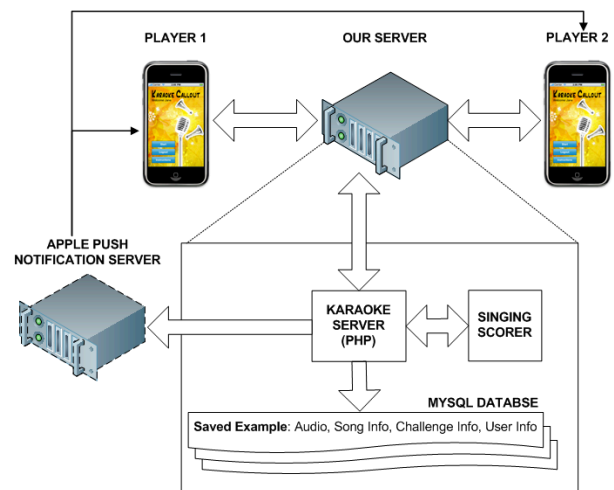


Figure 6. An overview of the KaraokeCallout system architecture.

USAGE STATISTICS

An ongoing goal of the Tunebot project has been to create a live, real-world system available to the general public, containing a growing set of songs that are of interest to a wide audience, and developed using data that represents the queries that real users generate. The usage statistics that follow were collected courtesy of Google Analytics.

In the period from January 15, 2010 to April 15, 2010, the Tunebot website had 15,421 unique visitors from 118 countries and territories. While more than three-quarters of these visits are from the United States and Canada, nearly 2,500 are from Europe and another 1,000 are from the rest of the world. The site receives between 100 and 200 hits on a typical day, most of which are new visitors. Figure 7 shows a breakdown of visitors by country of origin for the top ten countries.

Tunebot currently has more than 70 users who have chosen to register so that they may contribute songs to the system. It is clear the vast majority of users currently use the system anonymously to perform queries. We expect that broad dissemination of Karaoke Callout should increase the proportion of registered users.

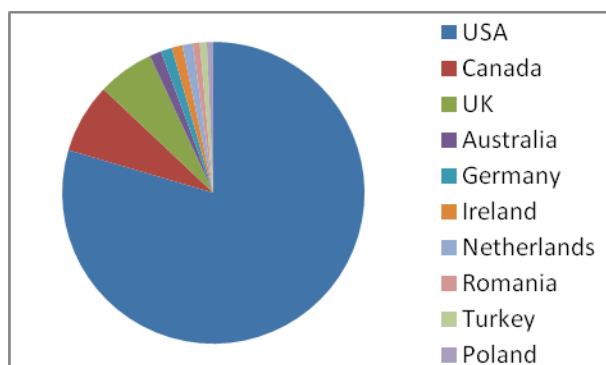


Figure 7. Proportion of visitors to Tunebot, by country of origin. Data collected over the period January 15 to April 15, 2010 (out of a total 15,421 unique visitors).

ANALYSIS AND CHALLENGES

Because we are developing a real-world system, some of our efforts have been directed at dealing with the practical issues that arise in implementing such a system, including robustness, scalability, efficiency, and system responsiveness.

The median length of a user query is around 18 seconds of audio, and our system currently takes about 5 seconds to return results from the time the query is received. For comparison, the longest query received to date is around 48 seconds long, and our system currently takes about 13 seconds to return a response to that query. The turnaround time is a function of several factors, including the size of the database and the length of the query, both in terms of the overall duration of the audio and the number of notes the user has sung. In the current implementation of the matching algorithm the running time is $O(kn)$, where k is the length of the query in notes and n is the number of keys in the database. While query lengths are not likely to change in the future, the size of the database is expected to grow dramatically over time. Algorithmic optimizations such as vantage point trees (discussed earlier) are one way to deal with the increasing query turnaround time. Another possibility, which we have implemented in our development environment but not yet in the production system, is to distribute the search algorithm across processors and compute matches in parallel. The potential for parallelization to speed up QBH is illustrated in [22].

Profiling analysis of our system has shown that a major portion of the query processing time is currently spent converting the raw audio of the query to the internal key representation, even though this phase of the algorithm does not dominate asymptotically. Future work includes exploring algorithmic and code-level optimizations to improve the running time of this portion of the algorithm.

A separate but related area of work has been to improve the scalability of our system in response to growing and fluctuating demand. This requires that the Tunebot serv-

ice run on multiple machines concurrently, while maintaining a synchronized view of the database (so that, for example, a newly contributed song will be visible immediately to the user who contributed it). Cloud computing is an appealing solution to provide online services in a scalable and distributed fashion. We have developed a working prototype of Tunebot that is deployed as a virtual machine image on the Amazon Web Services cloud infrastructure.

MOVING FORWARD

We expect this work will lead to new insight into the mappings between human perception, human music production and machine-measurable features of music, as well as leading to new approaches to automatically tagging large databases of multimedia content, new approaches to individualized search engines for improved results and new approaches to speed multimedia search.

ACKNOWLEDGEMENTS

We would like to thank the National Science Foundation for funding to do this research. This work was supported by NSF Grant number IIS-0812314.

REFERENCES

- [1] Haitsma, J. and T. Kalker. A Highly Robust Audio Fingerprinting System. in ISMIR 2002. 2002. Paris, France.
- [2] Wang, A. An Industrial Strength Audio Search Algorithm. in 4th International Conference on Music Information Retrieval (ISMIR 2003). 2003. Baltimore, Maryland, USA.
- [3] Typke, R., F. Wiering, and R.C. Velthkamp. A Survey of Music Information Retrieval Systems. in ISMIR 2005: 6th International Conference on Music Information Retrieval. 2005. London, England.
- [4] Dannenberg, R., W. Birmingham, B. Pardo, N. Hu, C. Meek, and G. Tzanetakis, A Comparative Evaluation of Search Techniques for Query-by-Humming Using the MUSART Testbed. Journal of the American Society for Information Science and Technology, 2007: p. in press.
- [5] Hewlett, W.B. and E. Selfridge-Field, eds. Melodic Similarity: Concepts, Procedures, and Applications. Computing in Musicology. Vol. 11. 1998, MIT Press: Cambridge, MA.
- [6] Hu, N., R. Dannenberg, and A. Lewis. A Probabilistic Model of Melodic Similarity. in International Computer Music Conference (ICMC). 2002. Goteborg, Sweden: The International Computer Music Association.

- [7] McNab, R.J., L.A. Smith, D. Bainbridge, and I.H. Witten, The New Zealand Digital Library MELody inDEX. D-Lib Magazine, 1997. May Issue.
- [8] Uitdenbogerd, A. and J. Zobel. Melodic Matching Techniques for Large Music Databases. in Seventh ACM International Conference on Multimedia. 1999. Orlando, FL.
- [9] Kornstadt, A., Themefinder: A Web-based Melodic Search Tool, in Melodic Similarity Concepts, Procedures, and Applications,, W. Hewlett and E. Selfridge-Field, Editors. 1998, MIT Press: Cambridge, MA.
- [10] Gillet, O. and G. Richard, Drum Loops Retrieval from Spoken Queries. Journal of Intelligent Information Systems, 2005. 24(2-3): p. 159-177.
- [11] Salamon, J. and M. Rohrmeier, A Quantitative Evaluation of a Two Stage Retrieval Approach for a Melodic Query by Example System, Proceedings of the 10th International Society of Music Information Retrieval Conference (ISMIR 2009), Kobe, Japan, 26-30 October 2009,
- [12] Meek, C. and W. Birmingham, A Comprehensive Trainable Error model for sung music queries. Journal of Artificial Intelligence Research, 2004. 22: p. 57-91.
- [13] Pauws, S. CubyHum: A Fully Operational Query by Humming System. in ISMIR 2002. 2002. Paris, France.
- [14] Unal, E., S.S. Narayanan, H. Shih, E. Chew, and C.J. Kuo. Creating Data Resources for Designing User-centric Front-ends for Query by Humming Systems. in Multimedia Information Retrieval. 2003.
- [15] Shamma, D. and B. Pardo. Karaoke Callout: using social and collaborative cell phone networking for new entertainment modalities and data collection, in Proceedings of ACM Multimedia Workshop on Audio and Music Computing for Multimedia (AMCMM 2006). 2006. Santa Barbara, CA, USA.
- [16] von Ahn, L. and L. Dabbish. Labeling Images with a Computer Game. in CHI 2004. 2004. Vienna, Austria.
- [17] Singh, P., The public acquisition of commonsense knowledge, in Proceedings of AAAI Spring Symposium on Acquiring (and Using) Linguistic (and World) Knowledge for Information Access. 2002, Palo Alto, CA.
- [18] Little, D., Raffensperger, D, and B. Pardo, A Query by Humming System that Learns from Experience, Proceedings of the 8th International Conference on Music Information Retrieval, 2007 Vienna,Austria
- [19] Wagner, R. and M. Fischer, The string-to-string correction problem. Journal of the ACM, 1974. 21(1): p. 168-173.
- [20] Chavez, E., G. Navarro, and J.L. Marroquin, Searching in Metric Spaces. ACM Computing Surveys, 2001. 33(3): p. 273-321.
- [21] Skalak, M., J. Han, B. Pardo, Speeding Melody Search with Vantage Point Trees, Proceedings of the International Society of Music Information Retrieval Conference(ISMIR 2008), Philadelphia, PA, USA, September 14-18, 2008.
- [22] Ferraro, P., P. Hanna, L. Imbert, T. Izard, Accelerating Query-by-Humming on GPU, Proceedings of the International Society for Music Information Retrieval, 2009.