Making the case for Query-by-Voice with EchoQuery

Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, Tim Kraska
Brown University

ABSTRACT

Recent advances in automatic speech recognition and natural language processing have led to a new generation of robust voice-based interfaces. Yet, there is very little work on using voice-based interfaces to query database systems. In fact, one might even wonder who in her right mind would want to query a database system using voice commands!

With this demonstration, we make the case for querying database systems using a voice-based interface, a new querying and interaction paradigm we call *Query-by-Voice* (*QbV*). We will demonstrate the practicality and utility of *QbV* for relational DBMSs using a using a proof-of-concept system called *EchoQuery*. To achieve a smooth and intuitive interaction, the query interface of *EchoQuery* is inspired by casual human-to-human conversations. Our demo will show that voice-based interfaces present an intuitive means of querying and consuming data in a database. It will also highlight the unique advantages of *QbV* over the more traditional approaches, text-based or visual interfaces, for applications where context switching is too expensive, too risky or even not possible at all.

1. INTRODUCTION

Motivation: Recent advances in automatic speech recognition and natural language processing enabled a new generation of robust voice-based interfaces, such as Apple's Siri [3], Google Voice Action [4], and Amazon's Alexa Voice Service [1]. When used together with relational database systems, voice-based interfaces provide an intuitive way to query and consume data. A major advantage over classical query interfaces or even touch-based visual interfaces, such as Tableau [11] or Vizdom [7], is that voice-based interfaces are completely hands-free.

As such, voice-based interfaces are superior in many situations where a context switch is either impossible or could cause a major distraction. As an example, one such scenario is when a doctor is in the middle of a surgery and instantly needs some information about the patient to make a critical decision. Instead of disrupting the workflow by using a visual interface to search the patient database, the doctor could simply use a voice-based interface directly to formulate a question like "What is the blood type of patient X?" or even more complex questions such as "What is the maximum dose of anesthetic Y for patient Z given his current conditions?".

The medium of spoken-dialogue, although weak in information density, is strong in encouraging interactive use and engaging users into a "conversation" with the database system. Moreover, a voice-based interface also has also other benefits towards the general accessibility of information. Finally, current database querying tools are inaccessible for people with disabilities that prevent them from using screens, keyboards, or even gestures. Having a voice-based interface for database querying enables people with disabilities to query data directly without having to use cumbersome and inefficient workarounds such as a footmice, eyetracking, or virtual keyboards in order to use SQL or any other query interface. Thus, we argue and demonstrate that a new querying paradigm, which we call *Query-by-Voice* (*QbV*), can be used profitably in such circumstances.

Contributions: In this demo, we present an end-to-end prototype system called *EchoQuery* that implements the *QbV* paradigm, by offering a voice-based and hands-free interface to a relational database. Natural language interfaces to databases (NLIDBs) have been studied for several decades in the past [6] and recently gained more attention again [8, 9]. Different from our system, early NLIDBs have been very limited since they mainly focused on constructing interfaces for individual domains and not general purpose interfaces for exploring arbitrary data sets. More recent work [8, 9] to construct NLIDBs also provides a general purpose natural-language interface to query data sets independent from a certain domain. However, this work mainly focuses on the translation process from natural language (e.g., English) to SQL rather than building a full end-to-end *QbV*-system which is the scope of *EchoQuery*.

Another major difference from existing work is that the query interface of *EchoQuery* is inspired by regular human-to-human conversations in order to be natural for the user. The main features of the voice-based interface of *EchoQuery* are:

- **Hands-free Access**: *EchoQuery* does not require the user to press a button or start an application using a gesture or a mouse-click. Instead, users can interact with the database by solely using their voice at any time.
- **Dialogue-based Querying**: While traditional database systems provide a one-shot (i.e., stateless) query interface, natural language conversations are incremental (i.e., stateful) in nature. To that end, *EchoQuery* provides a stateful dialogue-based query interface between the user and the database where (1) users can start a conversation with an initial query and refine that query incrementally over time, and (2) *EchoQuery* can seek for clarification if the query is incomplete or has some ambiguities that need to be resolved.

- **Personalizable Vocabulary**: Domain experts often use their own terms to formulate queries, which might be different from the schema elements (i.e., table and column names) of a database. Learning the terminology of a user and its translation to the underlying schema is similar to the problem of constructing a schema mapping in data integration. *EchoQuery* constructs these mappings incrementally on a per-user basis by issuing clarification questions using its dialogue-based query interface.

In its current version, *EchoQuery* does not strive to be a system which attempts to translate arbitrary natural language statements into queries. Instead, *EchoQuery* exposes its query interface as a natural language based version (i.e., a spoken version) of SQL with a purposefully limited grammar. However, as we will see, this query interface allows non-trivial database queries.

Outline: The rest of this paper is organized as follows. Section 2 gives an overview of the architecture of *EchoQuery*. Section 3 then discusses details of *EchoQuery*'s interface. Finally, Section 4 presents the scenario that we plan to show in our demo. We recommend that the reviewers watch our demo movie to get an idea of *EchoQuery*.[1]

2. SYSTEM ARCHITECTURE

EchoQuery is built using a middleware approach over an existing relational database that provides a SQL interface. While the middleware implements our novel concepts for a voice-based database interface (i.e., hands-free access, dialog-based querying, personalizable vocabulary), the database system is used to efficiently store and query the data.

Figure 1 shows the system architecture of *EchoQuery*. As a frontend to the user, we currently use the *Echo*, a wireless speaker and voice command device from Amazon, and *Alexa* [2], the voice command service that powers the Echo. The main purpose of Alexa is twofold: Firstly, Alexa turns the voice input of the user into an audio request and sends it to the *Alexa Voice Service (AVS)* via an HTTP-request. Secondly Alexa also replays the audio response that is returned from AVS via an HTTP-response.

Moreover, the AVS is also the entry point to *EchoQuery*. AVS implements the speech recognition part of *EchoQuery* and maps voice commands to different intents. An intent in *EchoQuery* defines how the voice command of the user is handled. At the moment, *EchoQuery* implements three different intents: a *Query Intent*, a *Refine Intent*, and a *Clarify Intent*. In the following, we briefly discuss the tasks of each intent. A detailed discussion of each intent can be found later in Section 3.

Query Intent: This intent is used by *EchoQuery* to start a conversation with the user. In order to process an initial query request, the *Query Intent* first adds missing information that is not specified by the user (e.g., it infers missing join predicates or table names that are not specified when selecting columns). Afterwards, the completed query request is translated into an executable SQL query using the *Query Translator*.

Once the SQL query is executed the *Result Translator* takes the result table and renders a (textual) query response that is replayed to the user. It is important to note that the

[1]`https://vimeo.com/vqtran/echoquery`

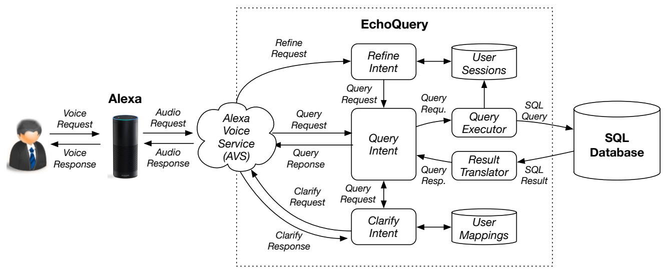

Figure 1: ***EchoQuery*** **Architecture**

result of the SQL query is not directly mapped into a textual query response one-by-one because the result might contain too many rows/columns to be efficiently consumable by the user. Therefore, the *Result Translator* might apply transformations that reduce the level of detail in the result (e.g., it returns only the count of rows as an answer instead of the result table content). If the user wants to know further details about the result, the user can use the *Refine Intent* to extract more details about the result or even to force *EchoQuery* to replay the full result.

Refine Intent: This intent is used by *EchoQuery* to answer a follow-up query in a conversation between the user and the database. In general, there are two purposes for which this intent is used for: (1) Extract some particular details of the result of the previous query (as discussed before). (2) Modify some parts of the previous query (e.g., apply or remove filter predicates, etc.). The *Refine Intent* therefore stores the last query of each user session.

Clarify Intent: If some information to actually execute a query request can not be automatically inferred or some information in the query request is ambiguous, the *Query Intent* calls the *Clarify Intent*. An example for this is, when the user refers to a column *name* that exists in multiple tables of the database schema. In that case, the *Clarify Intent* asks the user for more information to disambiguate the query request using a voice-based conversation (i.e., a clarify request is sent to Alexa for which the user has to provide a clarify response).

A second function of the clarify intent is that it learns the user terminology during the course of a conversation. Therefore, after asking the user for clarification, *EchoQuery* stores the mapping of the user terminology to schema elements (column or table names). That way, follow-up queries can use these stored mappings and execute the user queries without the need to ask the user for further clarification.

3. QUERY INTERFACE

EchoQuery's interaction model is composed of different *intents*, which implement the conversation between the user and our system.

3.1 Query Intent

The Query Intent is the primary intent that is typically used whenever a user starts a new conversation with *EchoQuery*. The input for this intent is called a query request.

Basic Query Requests: In its current version, *EchoQuery* supports simple non-aggregation and aggregation query requests using one of the aggregation type (COUNT, SUM, AVG, MAX, MIN). In the current version of *EchoQuery*, basic query request can only contain one result column but allows multiple where clauses, and also multiple group-by columns. To add or remove additional result columns, the Refine Intent has to be used. We felt that this is more natural to the user when using a voice-based interface where results are replayed to the user using an audio channel.

The most basic query request is of the following form:

> *Get the {Column}(s) of {Table}(s)?*
> *What is the {Aggregation} {Column}(s) of {Table}(s)?*

Here {Column}(s) {Table}(s) are elements of the database schema, while {Aggregation} refers to one of the spoken variants of the standard aggregation functions; i.e.,"total", "average", "minimum", and "maximum". COUNT aggregation queries, however, are formulated in a different way as they contain no column reference:

> *How many {Table}(s) are there?*
> *What's the number of {Table}(s)?*

There are many more variations of query requests that are not listed here. Given that, we can already see that even limited, the spoken SQL dialect of *EchoQuery* is quite intuitive. In the following, we consider how where and group-by clauses can be formulated in a query request.

Where Clauses: Where clauses can be added to a basic query request by appending:

> *... where {Table}(s) {Column}(s) {Comparator} {Value}?*

Here {Column}(s) and {Table}(s) refer to elements of the database schema, while {Comparator} and {Value} refer to the comparison operator ("is", "is equal to", "is greater than", "is less than", etc.), and the value to compare the column against, respectively. When referring to a column the *{Table}(s)* phrase is optional in all query requests. For example, two equivalent query requests that use a simple where clause (one with and one without the table name) are:

> *How many orders are there where customer name is Sally?*
> *How many orders are there where name is Sally?*

Multiple where clauses are also supported, and can be added by adding "and" between clauses. At the moment, we only support conjunctive predicates for the where clause. We decided to not support arbitrary predicates in our initial version of *EchoQuery* since the semantics involving precedence of order of conjunctions and disjunctions would be hard to grasp for a non-expert user.

Moreover, it is important to note that join clauses between different tables used in the query request are not specified at all in our spoken SQL dialect. Instead, join paths are inferred using foreign-key relationships in the database schema. In the example above, *EchoQuery* uses the foreign-key relationship of the *Orders* table that refers to the *Customer* table. In case multiple join paths exist, *EchoQuery* asks the user for feedback using the Clarify Intent.

Group-by Clauses: Group-by clauses can be added to a query by adding one of the following variants to the beginning or end of a query request:

> *Grouped by {Table}(s) {Column}(s), ...*
> *For each {Table}(s) {Column}(s), ...*
> *..., grouped by {Table}(s) {Column}(s).*
> *..., for each {Table}(s) {Column}(s).*

For example, the following two queries are equivalent:

> *For each supplier name, what's the average price of orders?*
> *What is the average price of orders, grouped by supplier name?*

3.2 Refine Intent

After the user has formulated a query request using the Query Intent and thus opened a session with *EchoQuery*, the user can issue a refine request that triggers the Refine Intent to modify the last query of the session. Currently, possible modifications are to add/remove a result column and to add/change/remove where or group-by clauses.

Result Columns: The refine requests to add or remove a result column are of the following form:

> *Add {Table}(s) {Column}(s)*
> *Drop {Table}(s) {Column}(s)*

Similar requests can be also used to add or remove aggregation functions.

Where Clauses: A refine request can also be used to modify the where clause of the last query. To trigger the Refine Intent to modify the where clause, the user can use one of the following refine requests:

> *And {Predicate}*
> *What if {Predicate}*
> *Drop {Predicate}*

While the And refine request is used to add a simple predicate to the current where clause, the What if request modifies a simple predicate with the given column name. Finally, the Drop request removes a simple predicate from the current where clause. An example of a refine request where the user adds an additional where clause is:

> *And where supplier name is Amazon.*

Group-by Clauses: Similar refine requests can also be used to modify the group-by clause:

> *Also group by {Column}(s)*
> *Instead group by {Clause}*
> *Drop group by {Column}(s)*

An example of refine request where the user adds an additional group-by attribute thus is:

> *Also group by supplier name.*

A complete interaction with *EchoQuery* (EQ) using the Query Intent and Refine Intent might look like:

> *User: How many orders are there?*
> *EQ: There are six thousand rows in the orders table.*

User: And where customer name is Sally?
EQ: There are three hundred rows in the orders table where customer name is Sally.
User: Also group by supplier name?
EQ: There are one hundred rows in the orders table where customer name is equal to Sally for the suppliers name Adobe, one hundred fifty rows for Amazon, and fifty rows for Apple.

Another class of refine requests is used to extract more information about the previous result. This class of refine requests is useful, in case the last query result contains too many rows or columns to be consumable by the user using the voice-based result and *EchoQuery* thus only replays the number of rows as a voice-based result to the user. In the current version, this case is determined by some configurable and static thresholds. In future work, we plan to study results from perceptual science, of how much information a user can process efficiently using its auditory perception system, to make that decision.

In order to request more details about the previous result, the user can issue a refine request that starts with the phrase For the last result, ... Moreover, to be executable the user has to append a normal query request that asks for specific details. In effect, this is a query over the previous query's result. An example for this refine request is:

For the last result, how many rows are there where name is Sally

Future work may include other ways of modifying a query using a refine request (e.g., to formulate arbitrary predicates). However, we can already see how using both the Query Intent and Refine Intent already results in quite a fluent and natural interaction between the user and *EchoQuery*.

3.3 Clarify Intent

This last intent illustrates an important capability of *EchoQuery*, namely asking questions back to the user in case a query or refine request can not be directly answered. This intent is applied in many different situations in *EchoQuery*.

In the following, we discuss the case, where *EchoQuery* runs into a case where the user references a column without specifying which table it should be from, and our system is unable to infer which table the user is referencing to as one example. That is, there is more than one table with the same column name. A full discussion of all different cases which need clarification is beyond the scope of this demo.

In that case, instead of returning an error and having the user to start all over again with issuing a possibly lengthy Query Intent, *EchoQuery* asks the user for clarification:

By {ColumnUsage}, do you mean {Table1} {Column1}, ... , or {TableN} {ColumnN}?

{ColumnUsage} here quotes the user's usage of the column to remind them which part of their query request *EchoQuery* is referring to and {Table1} {Column1}, ... , or {TableN} {ColumnN} lists the possible choices to disambiguate the initial query request. The user then responds with a clarify response to tell *EchoQuery* what they actually meant. In the example before, this could be in the form of:

I meant {Table}(s) {Column}(s)

Once the user responds with this clarify response, the ambiguous query request is updated with this specific table. If there are more ambiguities in the query then *EchoQuery* will continue to ask further questions until they are resolved, otherwise the system continues as before. To avoid asking the same clarify requests over and over *EchoQuery* stores the answers of the users as mappings for future use.

In the following, we show an example of a complete dialogue involving a clarify request and response:

User: How many orders are there where name is Sally?
EQ: By where name is equal to Sally, are you referring to parts name, suppliers name, or customers name?
User: I meant customers name.
EQ: There are three hundred rows in the orders table where customer name is Sally.

4. DEMO SCENARIO

In our demo, we show all the before-mentioned features of *EchoQuery* and its query interface using the setup as depicted in Figure 1. For demoing our prototype of *EchoQuery*, we provide two databases that are preloaded: (1) A variant of the star schema benchmark database [10] where we replace the generated random values (e.g., random customer names) with more meaningful values. (2) The Multiparameter Intelligent Monitoring in Intensive Care (MIMIC II) dataset [5], which contains clinical data for ICU patients.

To get a good sense of how *EchoQuery* operates, we recommend seeing our demo video for the star schema benchmark database: `https://vimeo.com/vqtran/echoquery`.

5. REFERENCES

[1] Alexa Voice Service. `https://developer.amazon.com/appsandservices/solutions/alexa/alexa-voice-service`. Accessed: 2016-01-15.

[2] Amazon Echo / Alexa. `http://www.amazon.com/echo`. Accessed: 2016-01-15.

[3] Apple Siri. `http://www.apple.com/ios/siri/`. Accessed: 2016-01-15.

[4] Google Voice Action. `https://developers.google.com/voice-actions/`. Accessed: 2016-01-15.

[5] MIMIC II Dataset. `http://mimic.physionet.org`. Accessed: 2016-01-15.

[6] I. Androutsopoulos et al. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[7] A. Crotty et al. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2035, 2015.

[8] F. Li et al. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[9] F. Li et al. Nalir: an interactive natural language interface for querying relational databases. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 709–712, 2014.

[10] P. O'Neil et al. Star Schema Benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`, 2007.

[11] P. Terlecki et al. On improving user response times in tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1695–1706, 2015.