Voice-based Data Exploration: Chatting with your Database

Prasetya Utama
Brown University

Nathaniel Weir
Brown University

Carsten Binnig
TU Darmstadt and Brown University

Ugur Çetintemel
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 paper, 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 show the practicality and utility of *QbV* for relational DBMSs using a using a proof-of-concept system called *EchoQuery*. There exists already work for building natural language interfaces for databases. A first major difference to this line of work is, that we use recent deep-learning models to allow for a robust translation from natural language to SQL. 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.

ACM Reference Format:
Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Çetintemel. 2017. Voice-based Data Exploration: Chatting with your Database. In *Proceedings of SCAI'17*. ACM, New York, NY, USA, 6 pages.

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 [12], Google Voice Action [6], and Amazon's Alexa Voice Service [3]. 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 [14] or Vizdom [5], 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.

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 other benefits towards the general accessibility of information. 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 footmice, eyetrackers, or virtual keyboards in order to use SQL or any other query interface. Thus, we argue that a new querying paradigm, which we call *Query-by-Voice* (*QbV*), can be used profitably in such circumstances.

Contributions: In this paper, 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. We recommend that readers watch our demo movie to get an idea of *EchoQuery*.[1]

Natural language interfaces to databases (NLIDBs) have been studied for several decades in the past [2] and recently gained more attention again [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 [9] to construct NLIDBs also provides a 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.
- **Conversational 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* help the user and suggest interesting data or it can

SCAI'17, ICTIR'17 Workshop on Search-Oriented Conversational AI, Amsterdam, Netherlands

[1] https://vimeo.com/151847274

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.

This paper is an extended version of a demo paper [10]. The main difference to the existing work is that in this paper we use recent deep-learning models to enable a more robust translation from natural language to SQL. 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. Furthermore, with our recent extensions towards using deep-learning, we are able to support more complex natural language queries as well.

Outline: The rest of this paper is organized as follows. Section 2 gives an overview of the initial architecture of *EchoQuery*. Section 3 then discusses details of *EchoQuery*'s conversational interface. Afterwards, Section 4 presents our recent extensions of how we trained the deep learning model to translate natural language queries to SQL and discusses the initial accuracy results compared to other state-of-the art approaches. Finally, Section 5 concludes and discusses future avenues.

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 front-end to the user, we currently use the *Echo*, a wireless speaker and voice command device from Amazon, and *Alexa* [1], 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.

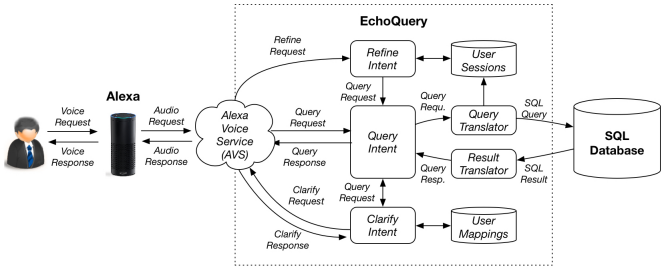

Figure 1: *EchoQuery* Architecture

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 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 needed 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.

In its current version, *EchoQuery* supports simple non-aggregation and aggregation query requests using one of the aggregation types (COUNT, SUM, AVG, MAX, MIN). In the current version of *EchoQuery*, a basic query request can contain either one or all result column(s), while allowing for multiple attributes in the where clause and also multiple group-by columns. To add or remove additional result columns, the Refine Intent must be used.

Basic Query Requests: The most basic query request is of the following form:

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

Here {Column}(s) and {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 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 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. 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 QUERY TRANSLATION

The main idea of the query translator in *EchoQuery* is that we model the translation form natural language to SQL as a language translation problem and use a state-of-the art sequence-to-sequence model [13]. A major challenge when using a sequence-to-sequence model for the translation from natural language to SQL is the problem of curating a large training set consisting of natural language to SQL pairs.

While existing work already has shown that a manually curated training corpus can be used to train a sequence-to-sequence model [7] for translating natural language to SQL, it imposes a high overhead for every new database that should be supported. Our approach is therefore different and does not require any manual effort to curate a training set. Instead, we use the database schema to fully automatically generate the training set. In the following, we describe the process of how we generate the training corpus and then show the accuracy results compared to other baselines.

4.1 Training Set Generation

The observation is that SQL, as opposed to natural language, has only a limited expressivity. We therefore use so-called query templates to instantiate different possible SQL queries over a given database schema:

Select {Att}(s) From {Table} Where {Filter}

In our training generator we use different types templates covering different types of queries from simple select-from-where queries up to more complex aggregate-grouping queries. At the moment, we do not yet support nested SQL queries. However, our templates could easily be extended to support nested queries as well.

For each of those templates, we also define a natural language counter-part templates for the direct translation such as:

Get the {Att}(s) of all {Table}(s) with {Filter}

Moreover, to make our translation model robust towards ways to paraphrase simple natural language queries, we additionally define different paraphrased templates for each SQL template to cover different paraphrasing techniques as discussed in [15] covering categories such as syntactical, lexical or morphological paraphrasing.

We instantiate these templates with the schema information of the database to generate the training set of natural language to SQL pairs that already covers a wide spectrum of possible query formulations that a user might use to query the database. An example of an instantiated SQL and natural language query pair looks like:

SELECT name FROM patient WHERE age=20 and
Get the name of all patients with age 20

One important step during the template instantiation is that we keep the balance for the number of query pairs that are instantiated per template. An imbalance of instances could result from templates with a different number of slots. The reason is that if we naively replace the slots of a query template with all possible combinations of slot instances (e.g., all attribute combinations of the schema), then instances that result from templates with more slots would dominate the training set and bias the translation model. We therefore, randomly sample from the possible instantations to get a good coverage of different queries but keep the number of instances per query template balanced.

In order to make the trained deep model more robust, we apply the following post-processing steps after the template instantiation phase: First, for each natural language query we lemmatise the words. Second, constants such as numbers or strings are replaced by special tokens. Finally, we further augment the dataset by randomly selecting words or phrases of a sentence and paraphrase them using PPDB [11] as the lexical resource. In future, we also plan to investigate the idea of using off-the-shelf part-of-speech tagger to enrich each word in a given query and make the model more robust towards syntactic paraphrasing..

4.2 Model Architecture and Training

We follow a similar architecture of sequence-to-sequence model from [13] for machine translation tasks. Our model maps the natural language input directly to SQL output queries, which could be seen as the target language. The model itself consists of two recurrent neural networks, namely, the encoder and decoder. The encoder network encodes a natural language input into its vector representation using word vectors.

As a network architecture, we use a bidirectional encoder/decoder architecture as proposed by [4]. Given the input sequence, the decoder learns to generate the resulting output sequence of the SQL query. Similar to [4], the decoder employs attention mechanism to allow the model to compensate for information loss that might happen during the sentence encoding. This mechanism empirically improves the performance of the model on longer input sequences.

Both the encoder and decoder comprise of three layers of gated recurrent units (GRUs). The dimension of the hidden state vectors as well as the the word embeddings is currently set to be 250 and 150, respectively. During training, we also apply a very aggressive dropout of 0.8 to avoid overfitting to our training corpus, which only consist of a small vocabulary (i.e., SQL keywords as well schema information of the given database).

During training, the model parameters are updated using Adam [8] optimization with a batch size of 64. All model parameters are initialized over a uniform distribution $\mathcal{U}(-0.1, 0.1)$. Upon testing, we applied a beam search mechanism of size 5 to compute the top-5 query translation candidates given a test input.

Table 1: Accuracy Results for our Benchmark

	Naive	Synt.	Lex.	Morph.
EchoQuery	100.0%	87.7%	68.42%	80.7%
Templ.-only [7]	7.0%	3.5%	1.7%	3.5%
NaLIR [9]	17.5%	8.7%	7.0%	14.0%

4.3 Evaluation Results

For evaluating our approach, we used a simple database schema with eight different attributes that models the name, age, disease, etc. of patients in a hospital. We used this database schema and the query templates as discussed before to instantiate our training set. In total, we used 89 different templates with pairs of SQL to NL queries. After instantiating the templates, we got a training set of approx. 450, 000 SQL and NL query pairs. After applying automatic paraphrasing using PPDB, we extended the training set to approx. 900, 000 SQL and NL query pairs.

For testing our model, we created a benchmark (test set) that covers different paraphrasing techniques for the user input The benchmark consists of 228 pairs of SQL and natural language queries for each of the following categories: naive, syntactical paraphrases, semantic paraphrases, and morphological paraphrases. The queries in the benchmark were not used as templates in the training set.

As first baseline, we used the template-only approach as presented in [7] that was used to extend the manual curated training set. The templates that were used covered only some simple natural language queries. We did not compare the performance to the extended manual curated training set but only to the training data generated from their templates since our aim in this paper to be able to build a query translator without any manual effort. As a second baseline, we used NaLIR, which is a state-of-the-art natural language interface for databases, that does not rely on a deep-learning model but uses a rule-based approach to map a semantic parse tree to a SQL query tree. Moreover, for NalIR we did not allow any user-feedback since user-feedback is an orthogonal dimension that could also be added to our approach; i.e., we wanted to show the baseline of a direct translation of natural language to SQL compared to our approach.

The results of the benchmark are shown in Table 1. We see that our approach outperforms the other approaches for each test category. Furthermore, when comparing the accuracy of our approach to the accuracy results reported in [7] for the manually curated training set, we reach approximately the same performance of 85% accuracy on average as well.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown the results of building a first system for querying database systems using a voice-based interface called

EchoQuery. We have show the practicality and utility of *QbV* for relational DBMSs using a using a proof-of-concept system called *EchoQuery*. 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. As a first difference to existing systems, we use recent deep-learning models to allow for a robust translation from natural language to SQL. Another major difference from existing work is that the query interface of *EchoQuery* is inspired by regular human-to-human conversations where the system can ask questions back to the user.

In future, we plan to extend our system to cover also more complex nested SQL queries as well as SQL queries with complex join paths. Other directions are to we also plan to leverage deep-learning techniques for the conversational part (i.e., where the system asks questions back to the user).

REFERENCES

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

[2] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

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

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[5] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2035, 2015.

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

[7] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 963–973, 2017.

[8] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[9] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[10] G. Lyons, V. Tran, C. Binnig, U. Çetintemel, and T. Kraska. Making the case for query-by-voice with echoquery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2129–2132, 2016.

[11] E. Pavlick and C. Callison-Burch. Simple PPDB: A paraphrase database for simplification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 2: Short Papers*, 2016.

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

[13] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[14] P. Terlecki, F. Xu, M. Shaw, V. Kim, and R. M. G. Wesley. On improving user response times in tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, SIGMOD '15, pages 1695–1706, 2015.

[15] M. Vila, M. A. Martí, and H. Rodríguez. Paraphrase concept and typology. A linguistically based and computationally oriented approach. *Procesamiento del Lenguaje Natural*, 46:83–90, 2011.