SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data

Vraj Shah, Side Li, Arun Kumar, Lawrence Saul
University of California, San Diego
{vps002,s7li,arunkk,saul}@eng.ucsd.edu

ABSTRACT

Speech-driven querying is becoming popular in new device environments such as smartphones, tablets, and even conversational assistants. However, such querying is largely restricted to natural language. Typed SQL remains the gold standard for sophisticated structured querying although it is painful in many environments, which restricts when and how users consume their data. In this work, we propose to bridge this gap by designing a speech-driven querying system and interface for structured data we call SpeakQL. We support a practically useful subset of regular SQL and allow users to query in any domain with novel touch/speech based human-in-the-loop correction mechanisms. Automatic speech recognition (ASR) introduces myriad forms of errors in transcriptions, presenting us with a technical challenge. We exploit our observations of SQL's properties, its grammar, and the queried database to build a modular architecture. We present the first dataset of spoken SQL queries and a generic approach to generate them for any arbitrary schema. Our experiments show that SpeakQL can automatically correct a large fraction of errors in ASR transcriptions. User studies show that SpeakQL can help users specify SQL queries significantly faster with a speedup of average 2.7x and up to 6.7x compared to typing on a tablet device. SpeakQL also reduces the user effort in specifying queries by a factor of average 10x and up to 60x compared to raw typing effort.

ACM Reference Format:
Vraj Shah, Side Li, Arun Kumar, Lawrence Saul. 2020. SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3318464.3389777

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389777

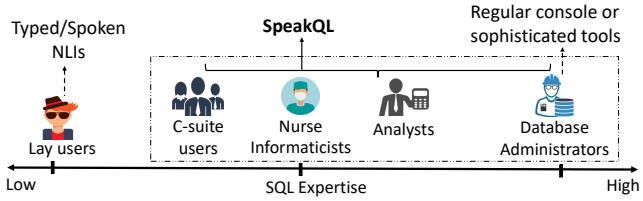


Figure 1: Contrasting SpeakQL's goals with current NLIs and other sophisticated tools in terms of SQL expertise of users.

1 INTRODUCTION

Structured data querying is practiced by users in many domains such as enterprise, Web, and healthcare. Typing queries in SQL is the gold standard for such querying. Many works have looked into creating new query interfaces that lowers the barrier to type SQL. They offer new types of querying modalities such as visual [4, 40], touch-based [13, 25], typed natural language interfaces (NLIs) [20], and even bidirectional conversations [21]. This allows users to query on constrained environments such as tablets, smartphones, and even conversational assistants without specifying any SQL. However, what is missing from the prior work is a speech-driven interface for regular SQL or other structured querying.

One might ask: *Why dictate structured queries and not just use NLIs or visual tools?* Many prior works assume there exist only two kinds of users: SQL wizards such as database administrators (DBAs), who use consoles or other sophisticated tools, or non-technical lay users, who use NLIs. This is a false dichotomy. As Figure 1 shows, there are many users who are comfortable with basic SQL and are mostly read-only data consumers such as business analysts, nurse informaticists, and managers. The SQL knowledge of such users is ignored by visual or NLI research. We conduct an interview study with 26 SQL users belonging to 17 different sectors to understand how a spoken structured querying interface can bridge such crucial gaps in querying capabilities. We summarize the key lessons from the study below and discuss it in depth in the technical report [31].

Lessons from interview studies. We find that most users in industry compose ad hoc queries over arbitrary tables and desire unambiguous response to their queries. In addition, consumers such as analysts and informaticists often desire anytime and anywhere access to their data, say via mobile

Type of Errors	Ground truth token	ASR transcription
Homophony (Keywords/ Special Characters to Literals)	sum	some
Homophony (Literals to Keywords/Special Characters)	fromdate	from date
Unbounded vocabulary for Literals	CUSTID_1729A	custody _ 1 7 2 9 8
	table_123	table _ 1 2 3
Splitting of numbers into multiple tokens	45412	45000 412
Erroneously transcribed dates	1991-05-07	may 07 90 91

Table 1: Illustration of different types of errors made by Automatic Speech Recognition engine (ASR).

platforms such as tablets and smartphones. Even SQL expert DevOps DBAs sometime desire off-hour on-the-go access to their data. However, as quoted by many users, typing SQL is really painful in such constrained settings. Having a speech-driven SQL interface that leverages both *speech* and potentially also the *touch* capabilities of such platforms can help speed up their query specification.

Comparison against NLI. One might still wonder: *Why can not these users query their data using spoken NLIs rather than dictating SQL?* SQL offers advantages that many data professionals find useful. SQL is already a structured English query language. Key to its appeal is query sophistication, lack of ambiguity due to its context-free grammar (CFG), and succinctness. In contrast, NLIs are primarily aimed at lay users and not necessarily professionals who manage structured data. We discuss this comparison in the tech report [31].

Thus, instead of forcing all users to only use NLIs, we pursue an exploratory research agenda that is complementary to NLI research and existing touch-based or visual interfaces. We study how to make spoken querying effective and efficient without losing SQL's benefits. In this work, we build a speech-driven querying system for a subset of SQL which we call SpeakQL. Since current NLIs are increasingly relying on keywords and structured interactions[1, 20], we believe our lessons can potentially also improve NLIs in future.

Desiderata. (1) Support regular SQL with a tractable subset of the CFG, although our architecture and methods should be applicable to any SQL query in general. (2) Leverage an existing modern state-of-the-art ASR technology instead of reinventing the wheel. (3) Support any database schema in any application domain. (4) Support speech-first query specification and speech-driven and potentially touch-driven query correction on a screen display. Overall, we desire an open-domain, speech-driven, and multimodal querying system for regular SQL wherein users can dictate the query and perform interactive correction using touch and/or speech.

Technical Challenges. Unlike regular English speech, SQL speech gives rise to interesting novel challenges: (1) ASR introduces myriad forms of errors when transcribing that confound different elements of the query, as illustrated by several examples in Table 1. (2) It is impossible for ASR to recognize tokens not present in its vocabulary. Such "out-of-vocabulary" tokens are more likely in SQL than natural English because SQL queries can have infinite varieties of literals, e.g. `CUSTID_1729A`. A single token from SQL's perspective might get split by ASR into many tokens. We call this the *unbounded vocabulary problem*, and it is a central technical challenge for SpeakQL. Note that this problem has not been solved even for spoken NLIs such as Alexa, which typically responds "I'm sorry, I don't understand the question" every time an out-of-vocabulary token arises. Thus, we believe addressing this problem may benefit spoken NLIs too. (3) Achieving real-time efficiency for an interactive interface is yet another technical challenge.

System Architecture. To tackle the above challenges, we make a crucial design decision: decompose the problem of correcting ASR transcription errors into two tasks: *structure determination* and *literal determination*. Structure determination delivers a syntactically correct SQL structure where literals are masked out with placeholder variables. Literal determination identifies the literals for the variables. This architectural decoupling lets us effectively tackle the unbounded vocabulary problem. If the transcription generated by SpeakQL is still incorrect, users can correct it interactively with speech/touch-based mechanisms in our novel interface.

Technical Contributions. Our key technical contributions are as follows. (1) For structure determination, we exploit the rich structure of SQL using its CFG to generate many possible SQL structures and index them with tries. We propose a similarity search algorithm with a SQL-specific weighted edit distance metric to identify the closest structure. (2) For literal determination, we exploit our characterization of ASR's errors on SQL queries to create a literal voting algorithm that uses the phonetic information about database instances being queried to fill in the correct literals. (3) We create an interactive query interface with a novel "SQL Keyboard" and a clause-level dictation functionality to make our interface multimodal and more amenable to speech- and touch-friendly corrections. For instance, to reduce cognitive load of users when dictating a longer query, we allow users to specify queries at a clause-level.

Overall, the key novelty of our system lies in synthesizing and innovating upon techniques from disparate literatures such as database systems, natural language processing, information retrieval, and human-computer interaction to build an end-to-end system that satisfies our desiderata. We adapt these techniques to the context of spoken SQL based on the syntactic and semantic properties of SQL queries.

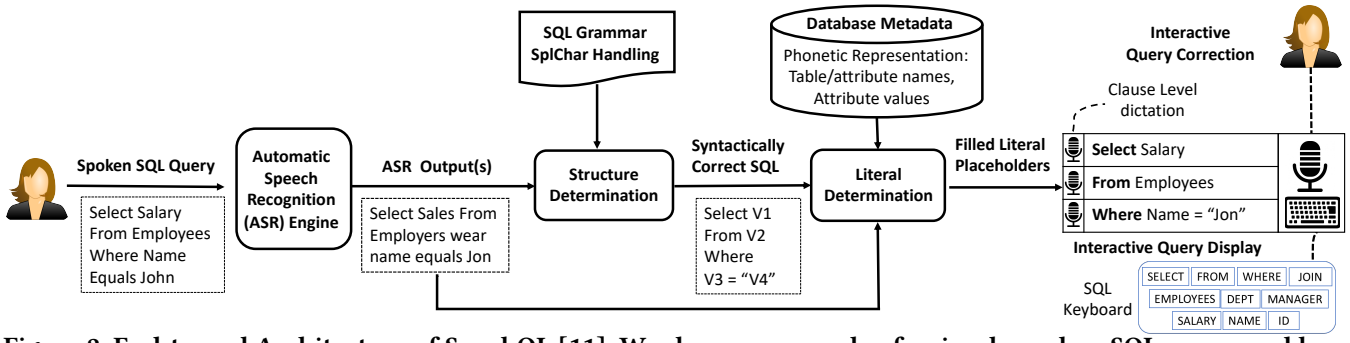


Figure 2: End-to-end Architecture of SpeakQL [11]. We show an example of a simple spoken SQL query, and how it gets converted to a query displayed on a screen, which the user can correct interactively.

Experimental Evaluation. We first explain why the existing datasets are not enough for spoken querying and we create the first dataset of spoken SQL queries using real-world database schemas. Using several accuracy metrics we show that SpeakQL can automatically correct large proportions of errors in the ASR transcriptions. For example, we see a substantial average lift of 21% in Word Recall Rate. SpeakQL achieves almost real-time latency and through user studies, we show that SpeakQL allows users to compose queries significantly faster, achieving a speedup of average 2.7x and up to 6.7x compared to typing on a tablet. Moreover, the user touch effort to specify and/or correct the query goes down by a factor of average 10x and up to 60x compared to raw typing. We then evaluate SpeakQL against state-of-the-art NLIs with typed and speech inputs on two large-scale datasets containing pairs of natural language and SQL queries: WikiSQL [39] and Spider [38]. Our evaluation shows that SpeakQL achieves significantly higher accuracy than state-of-the-art NLIs adapted for speech input. e.g., lift of 50% in execution accuracy on WikiSQL.

Overall, the contributions of this paper are as follows:

- To the best of our knowledge, this is the first paper to present an end-to-end speech-driven system for making spoken SQL querying effective and efficient.
- We propose a similarity search algorithm based on weighted edit distances and a literal voting algorithm based on phonetic representation for effective structure and literal determination, respectively.
- We propose a novel interface using SQL Keyboard and clause-level dictation functionality that makes correction and speech-driven querying easier in touch environments.
- We present the first public dataset of spoken SQL queries. Our data generation process is scalable and applies to any arbitrary database schema.
- We demonstrate through quantitative evaluation on real-word database schemas that SpeakQL can automatically correct a large portion of errors in ASR transcriptions. Moreover, our user studies shows that SpeakQL helps significantly reduce user time and effort in SQL specification.

2 SYSTEM ARCHITECTURE

Modern ASR engines powered by deep neural networks have become the state-of-the-art for any industrial strength application. Hence, to avoid replicating the engineering efforts in creating a SQL-specific ASR, we exploit an existing ASR technology. This decision allows us to focus on issues concerning only SQL as described below.

First, unlike regular English, there are only three types of tokens that arises in SQL: *Keywords*, *Special Characters* ("SplChar"), and *Literals*. SQL Keywords (such as *SELECT, FROM* etc.) and SplChars (such as * = etc.) have a finite set of elements that occurs only from the SQL grammar [7]. A literal can either be a table name, an attribute name or an attribute value. Table names and attribute names have a finite vocabulary but the attribute value can be any value from the database or any generic value. Hence, the domain size of the Literals would likely be infinite.

Second, the ASR engine can fail in several interesting ways when transcribing as shown in Table 1. Due to homophones, ASR might convert Literals into Keywords or SplChars and vice versa. Even a single-token transcription might be completely wrong because the token is simply not present in ASR's vocabulary. Worse still, ASR might split a token like `CUSTID_1729A` into a series of tokens in the transcription output, possibly intermixed with Keywords and SplChars.

These observations related to SQL suggest that a correctly recognized set of Keywords and SplChars can help us deliver the correct SQL structure. Correct structure combined with the correct Literals can give us the correct valid query. Based on this observation, we make an important architectural design decision to decouple structure determination from literal determination. *This decoupling is a critical design decision that helps us tackle the unbounded vocabulary problem.* We present the complete four-component end-to-end system in Figure 2 and the components are described below. We presented an initial version of this architecture in [11].

ASR Engine. This component processes the recorded spoken SQL query to obtain a transcription output. ASR consists of two major components: acoustic model and language

model. The acoustic model captures the representation of sounds for words, and the language model captures both vocabulary and the sequence of utterances that the application is likely to use. We utilize Azure's Custom Speech Service to create a custom language model by training on the dataset of spoken SQL queries (explained in 6.1). For the acoustic model, we use Microsoft's state-of-the-art search and dictation model. For the dictated query in Figure 2, the result returned by ASR engine could be `select sales from employers wear first name equals Jon`.

Structure Determination. This component processes the ASR output to obtain a syntactically correct SQL statement with numbered placeholder variables for Literals, while Keywords and SplChars are fixed. We propose a similarity search algorithm with a SQL-specific weighted edit distance metric that leverages SQL's CFG to deliver a syntactically correct SQL structure. In our running example, the detected structure is `Select x1 From x2 Where x3 = x4`. Here, the Keywords and SplChars are retained, while the Literals are shown as placeholder items `x1`, `x2`, `x3` and `x4`. We dive into Structure Determination in depth in Section 3.

Literal Determination. The Literal Determination component finds a ranked list of Literals for each placeholder variable using both the raw ASR output and a pre-computed phonetic representation of the database being queried. For example, variable `x1` is replaced as a top k list of attribute names. Phonetically, among all the attribute names, `Salary` is the closest to `Sales`, and thus, `x1` would be bound to `Salary`. This component is explained in depth in Section 4.

Interactive Display. We present a single SQL statement to the user. Even with our query correction techniques, some tokens in the transcription may be incorrect, especially for Literals not in the ASR vocabulary ("out-of-vocabulary" Literals). Thus, we support user-in-the-loop interactive query correction through speech or touch-based mechanisms. The user can re-dictate queries at the clause level or make use of a novel SQL keyboard tailored to reduce their correction effort. Section 5 explains the interface in depth.

3 STRUCTURE DETERMINATION

We now discuss the challenges of structure determination and present our algorithms to tackle them. The goal of this component is to get a syntactically correct SQL statement given ASR transcription. Figure 3 presents its architecture.

Supported SQL Subset. We currently support a subset of regular SQL DML that is meaningful and practically useful for spoken data retrieval and analysis. This subset includes Select-Project-Join-Aggregation (SPJA) queries along with `LIMIT` and `ORDER BY`, one level nested queries, *without any limits* on the number of joins or aggregates, as well as on predicates. We do not currently support queries belonging to SQL

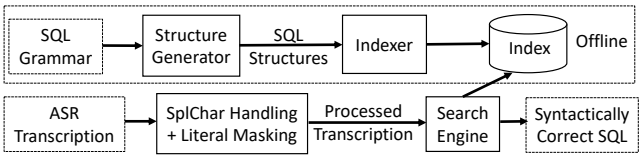


Figure 3: Structure Determination component's architecture.

DDL. We use the production rules of `SELECT` statements of standard SQL in Backus-Naur Form [7]. This subset already allows many structurally sophisticated retrieval and analysis queries that may arise in speech-driven environments. That said, we do plan to systematically expand our subset to offer more SQL functionalities in future work. In contrast, note that some NLIs impose much more stringent structural restrictions. For instance, the state-of-the-art NLI on the WikiSQL dataset [17, 39] allows queries over only one table and with only one aggregate. In addition, the task on Spider dataset does not involve generating condition values [38]. We provide the full grammar in the technical report [31].

3.1 SplChar Handling and Literal Masking

We create a dictionary of the supported SQL Keywords and SplChars, namely, `KeywordDict` and `SplCharDict` as below:

`KeywordDict`: *Select, From, Where, Order By, Group By, Natural Join, And, Or, Not, Limit, Between, In, Sum, Count, Max, Avg, Min*

`SplCharDict`: `* = < > ( ) . ,`

ASR often fails to correctly transcribe SplChars and produces the output in words. For example, < becomes "*less than*". Thus, we replace the substrings in the transcription output (`TransOut`) with the corresponding SplChars. Then, we mask out all tokens in the transcribed text that are not in `KeywordDict` or `SplCharDict` with a placeholder variable. In our running example, the masked out transcription output (`MaskOut`) is `SELECT x1 FROM x2 x3 x4 = x5`.

3.2 Structure Generator

This offline component uses the production rules in the grammar recursively to generate a sequence of tokens, which is a string representing a SQL ground truth structure. Since the number of tokens that can be generated is infinite, we restrict the string to a maximum of 50 tokens. This leads to generation of roughly 1.6M ground truth structures. Our basic idea is to compare `MaskOut` with these generated ground truth structures and select the one with minimum edit distance. *Thus, the knowledge of the grammar lets us effectively invert the traditional approach of parsing strings to extract structure.* We found that parsing is an overkill for our setting, since the grammar for spoken queries is more compact than the full grammar of SQL. Furthermore, the myriad forms of errors ASR introduces (Table 1) means deterministic parsing will

almost always fail. Early on, we also tried a probabilistic CFG and probabilistic parsing but it turned out to be impractical because configuring all the probabilities correctly is tricky and parsing was slower.

3.3 Indexer

Comparing `TransOut` with every ground truth string will be too slow as we want our system to have a real-time latency. Thus, we index the generated ground truth strings such that only a small subset needs to be retrieved by the Search Engine to be compared against `TransOut`. A challenge is that the number of strings to index is large. *But we observe that there is a lot of redundancy, since many strings share prefixes.* This observation leads us to consider a trie structure to index all strings. A path from root to leaf node represents a string from the ground truth structures. Every node in the path represents a token in the string. Thus, tries not only save memory but can also save computations with respect to common prefixes. The computations can be saved further by making the search engine more aware of the length of strings in the trie as we will explain in Section 3.4. Hence, packing all strings into a single trie leads to a higher latency. Since latency is a major concern for us, we trade off memory to reduce latency by storing many tries, one per structure length. We have 50 disjoint tries in all.

3.4 Search Engine

Given `MaskOut`, the search engine aims to find the closest matching structure by comparing against the ground truth strings from the index based on edit distance. There are many variants of edit distance that differs in the set of operations involved. We use a weighted longest common subsequence edit distance [26], which allows only insertion and deletion operations at the token level.

Typically, all operations in an edit distance function are equally weighted. But we introduce a twist in our setting based on a key observation of ASR outputs. We find that ASR is far more likely to correctly recognize Keywords than Literals, with SplChars falling in the middle. Thus, we assign different weights to these three kinds of tokens. We assign the highest weight W_K to Keywords, next highest W_S to SplChars, and lowest W_L to Literals. We set $W_K = 1.2, W_S = 1.1$ and $W_L = 1$. One could set these weights differently by training an ML model, but we find that the exact weight values are not that important; it is the ordering that matters. Thus, the fixed weights suffice for our purpose.

Denote the source string as $a = a_1a_2...a_n$ and target string as $b = b_1b_2...b_m$. Let dp denote a matrix with $m + 1$ columns and $n + 1$ rows, and $dp(i, j)$ be the edit distance between the prefix $a_1a_2...a_i$ and $b_1b_2...b_j$. Algorithm 1 shows the dynamic program to compute this matrix. We observe that computing `dp(i,j)` requires only the previous column (`DpPrvCol`) and current column (`DpCurCol`). Moreover, if for a node n, `min(DpCurCol)` > `MinEditDist`, then we can stop exploring it further. We now present an optimization that can reduce the computational cost of searching over our index.

Algorithm 1 Dynamic Programming Algorithm

```
1: if token in KeywordDict then W_token = W_K
2: else if token in SplCharDict then W_token = W_S
3: else W_token = W_L
4: dp(i,0) = i for 0 ≤ i ≤ n; dp(0,j) = j for 0 ≤ j ≤ m
5: if a(i) == b(j) then dp(i,j) = dp(i-1,j-1) = DpPrvCol(row-1)
6: else dp(i,j) = min(W_token+dp(i-1,j), W_token+dp(i,j-1))
7: DpPrvCol(row) = dp(i,j-1)
8: DpCurCol(row-1) = dp(i-1,j)
9: insertCost = DpPrvCol(row) + W_token
10: deleteCost = DpCurCol(row-1) + W_token
```

Bidirectional Bounds. Recall that our index has many tries, which means searching could become slow if we do it naively. Thus, we now present a simple optimization that prunes out most of the tries without altering the search output. Our intuition is to bound the edit distance from both below and above. Given two strings of length m and n (without loss of generality, $m > n$), the lowest edit distance is obtained with $m - n$ deletes. Similarly, highest edit distance is obtained with m deletes and n inserts. This leads us to the following:

Proposition 1. Given two query structures with m and n tokens, their edit distance d satisfies the following bounds: $|m - n| \cdot W_L \leq d \leq |m + n| \cdot W_K$.

Here, the lower bound denotes the best case scenario with $|m - n|$ deletes and minimum possible weight of W_L. The upper bound denotes the worst case scenario with m deletes, n inserts and maximum possible weight of W_K. To illustrate how our bounds could be useful, we present an illustrative example in the technical report [31].

Overall Search Algorithm. Our main idea is to skip searches on tries that are pruned by our bidirectional bounds in Proposition 1. For the tries that are not pruned, we recursively traverse every children of the root node. At every node, we use the dynamic program to calculate edit distance with `TransOut`. When we reach a leaf node and see that the edit distance with current node is less than `MinEditDist`, then we update `MinEditDist` and the corresponding structure. This algorithm does not affect accuracy, i.e., it returns the same string as searching over all the tries. Its worst case time complexity is $O(pkn)$, where n is the length of the `TransOut`, p is the number of nodes in the largest trie, and k is the number of tries. The space complexity is $O(pk)$. The complete search procedure along with the proofs of the complexity analysis can be found in our technical report. We also study two additional accuracy-latency tradeoff algorithms that further reduce runtime by trading off some accuracy, which can be found in our tech report [31]. Note that we do not use the approximation techniques by default in SpeakQL, but users can choose to enable them, if they want even lower latency.

4 LITERAL DETERMINATION

The goal of this component is to "fill in" the values for the placeholder variables in the syntactically correct SQL structure delivered by the Structure Determination component. Literals can be table names, attribute names, or attribute values. Table names and attribute names are from a finite domain determined by the database schema but the vocabulary size of attribute values can be infinite. This presents a challenge to this component because the most prominent information that it can use to identify a literal for any placeholder variable is the raw ASR transcription output. This transcription is typically erroneous and unusable directly because ASR can either split the out-of-vocabulary tokens into a series of tokens, incorrectly transcribe it, or simply not transcribe it at all. Even for in-vocabulary tokens, ASR is bound to make mistakes due to homophones (see Table 1). These observations about how ASR fails helps us to identify two crucial design decisions for Literal Determination.

1. Leveraging phonetic representation. In contrast to string-based similarity search, a similarity search on a precomputed *phonetic representation* of the existing Literals in the database can help us disambiguate the words from `TransOut` that sound similar. This motivates us to exploit a phonetic algorithm called Metaphone that utilizes 16 consonant sounds describing a large number of sounds used in many English words. We use it to build a dictionary for indexing the table names, attribute names, and attribute values (only strings, excluding numbers or dates) based on their English pronounciation. For example, phonetic representations of table names `Employees` and `Salaries` are given by `EMPLYS` and `SLRS` respectively.

2. Handling out-of-vocabulary tokens. Literal Determination has to be made aware of the splitting of tokens (out-of-vocabulary from ASR's perspective) into sub-tokens so that it can decide when and how to merge them. Figure 4 shows the workflow of this component with `TableNames`= {`Employees (EMPLYS)`,`Salaries (SLRS)`} and `AttributeNames` ={`FirstName(FRSTNM), LastName(LSTNM)`}. The inputs are `TransOut` and best structure (`BestStruct`) obtained from the Structure Determination. As output, we want to map a literal each to every placeholder variable in `BestStruct`. To do so, we first identify the type of the placeholder variable (table name, attribute name, or attribute value). This lets us reduce the number of Literals to consider for a placeholder. We denote the set containing relevant Literals for a placeholder variable by set B. Next, we use `TransOut` to identify what exactly was spoken for Literals. We segment `TransOut` to identify a set of possible tokens to consider and form set A. Finally, we identify the most phonetically similar literal by computing edit distance between the phonetic representations of the two sets A and B. The algorithm pseudocode can be found in our technical report [31]. Its worst-case time complexity is $O(n^2m)$, where n is the length of `TransOut` and m is the domain size of Literals. The space complexity is $O(n^2 + m)$.

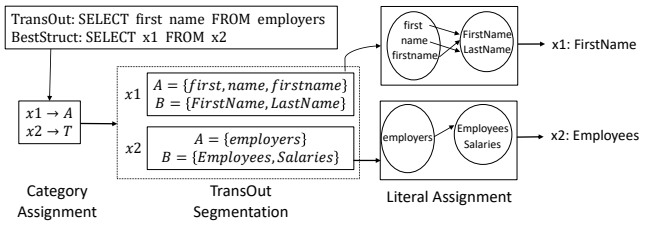


Figure 4: Literal Determination component example

4.1 Category Assignment

We constrain the space of possible Literals to consider for any given placeholder variable in `BestStruct`. Each variable can be a table name (type = `T`), an attribute name (category type = `A`) or an attribute value (type = `V`). We assign a category type to the placeholder variable using SQL grammar. In Figure 4, the category assigned to `x2` is type `T`, and `x1` is type `A`. Given a placeholder variable in `BestStruct`, we retrieve the phonetic representation of the relevant Literals. For example, if the placeholder variable is of type `T`, then the set B of phonetic representations for all the table names is returned.

4.2 Transcription Output Segmentation

We now determine the exact literal to "fill in" a placeholder. This requires using `TransOut` to identify transcribed tokens for Literals. We segment `TransOut` such that only relevant tokens are retrieved to be compared against set B items. For a placeholder in `BestStruct`, we first identify a window in `TransOut` where the literal is likely to be found. In our example, the window for `x1` starts at token `first` and ends at token `name`. We then enumerate all the possible substrings (phonetic representation) of Literals occurring in the window in set A. For variable `x1`, A = {`first,name,firstname`} and B is the set of attribute names.

4.3 Literal Assignment

As the final step, we retrieve the most likely literal for a placeholder variable by comparing the enumerated strings in set A and relevant Literals in set B. The comparison is based on the character level edit distance of the strings in phonetic representation. Our algorithm is given below.
(1) For an item a in set A, compute pairwise edit distance with every item in set B. (2) Pick an item $b \in B$ that has least edit distance. Hence, a has so-called "voted" for b. (3) Repeat this process of voting for every item $a \in A$.

We return the literal with the maximum number of votes. We fetch top k Literals overall for each placeholder variable. The ties in votes are resolved in lexicographical order. In our running example, the returned literal for the variable `x1` is `FirstName`, while for `x2` is `Employees`. We present many examples of this step in our tech report [31].

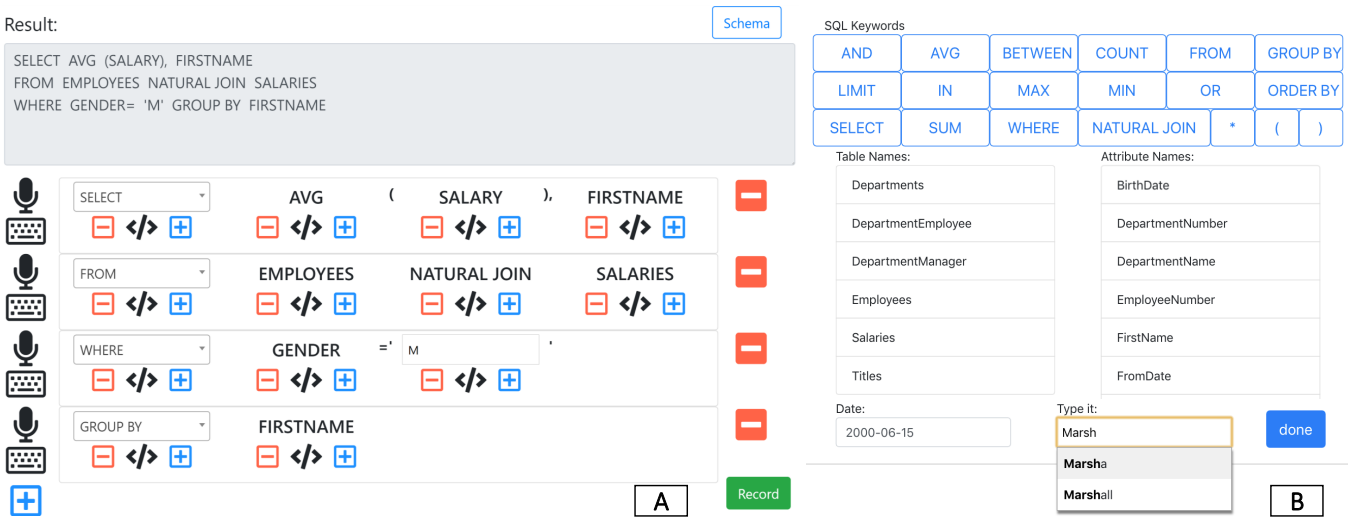


Figure 5: SpeakQL Interface [32]. (A) The Interactive Display showing the dictated query after being processed by the SpeakQL engine, as well as the touch-based editing functionalities and clause-level redictation capabilty. (B) Our simple SQL keyboard designed for touch-based editing of the rendered query string.

5 INTERFACE

Figure 5(A) shows our interface. We demonstrated our interface at [32]. This interface allows users to dictate SQL query and interactively correct it, if the transcribed query is erroneous. Such interactive query correction can be performed using both touch/click and speech. The "Record" button at the bottom right allows the user to dictate the entire query in one go. At the same time, the interface allows the user to dictate or correct (through re-dictation) the queries at the clause level (using record button to the left of each clause). For example, the user can choose to dictate only the `SELECT` clause or `WHERE` clause. We find from user study (Section 6.4) that this clause-level functionality helps users in reducing their cognitive load while speaking significantly. Such a design makes our interface more speech-friendly.

Figure 5(B) shows the novel "SQL Keyboard" that consists of entire lists of SQL Keywords, table names, and attribute names. Since attribute values (including dates) can be potentially infinite, they cannot be seen in a list view. But the user can type with the help of an auto complete feature. Dates can be specified easily with the help of a scrollable date picker. Our keyboard design allows for a quick in-place editing of stray incorrect tokens, present anywhere in the SQL query string. We find from user study (Section 6.4) that such a design makes our interface more correction-friendly. In the worst case, if our system fails to identify the correct query structure and/or Literals, the user can type one token, multiple tokens, or the whole query from scratch in the query display box, or redictate the clauses or the whole query again. Thus, overall, SpeakQL's novel multimodal query interface allows users to easily mix speech-driven query specification with speech/touch interactive query correction.

6 EXPERIMENTAL EVALUATION

We now present a thorough empirical evaluation of SpeakQL. We first present the new dataset of spoken SQL queries. We define the accuracy metrics and evaluate SpeakQL end-to-end on them. We then present our findings from actual user studies with SpeakQL. Next, we dive deeper into evaluating SpeakQL's components. Finally, we compare SpeakQL against NLIs on two existing large-scale datasets.

6.1 New Dataset for Spoken SQL

Why are existing datasets not enough? The existing large-scale datasets created for NLIs such as Spider [38] and WikiSQL [39] are not directly tailored towards evaluating a spoken querying system. This is because the major difficulty metric for spoken querying and typed querying are different. The difficulty for typed NLI lies in inferring join paths and building nested queries [9, 10]. While for spoken querying, the difficulty metric is the number of tokens in the query. For instance, even a natural language query with 50 tokens can be very simple for a typed NLI but not necessarily for a spoken NLI. Conversely, a short query with many joins may be simple for SpeakQL but very hard for an NLI. We confirm this observation by comparing SpeakQL against state-of-the-art NLIs on Spider and WikiSQL datasets in tech report [31].

Procedure to generate dataset on arbitrary schema. To the best of our knowledge, there are no publicly available datasets for spoken SQL queries. Hence, we create our own dataset using a scalable procedure described below.

1. We use two publicly available database schemas: Employees Sample Database from MySQL [2] and the Yelp Dataset [8]. We get the `table names`, `attribute names`, and `attribute values` in each database.

Metric	Top 1			Top 5		
	Employees		Yelp	Employees		Yelp
	Train	*Test*	*Test*	*Train*	*Test*	*Test*
KPR	0.99	0.98	0.94	0.99	0.99	0.98
SPR	0.99	0.98	0.98	0.99	0.99	0.99
LPR	0.92	0.85	0.72	0.97	0.93	0.81
WPR	0.95	0.91	0.81	0.98	0.96	0.9
KRR	0.99	0.97	0.95	0.99	0.99	0.99
SRR	0.98	0.98	0.98	0.99	0.99	0.99
LRR	0.88	0.8	0.64	0.95	0.91	0.69
WRR	0.92	0.88	0.78	0.96	0.95	0.82

Table 2: End-to-end mean accuracy metrics on real data for query string corrected by SpeakQL.

2. Use our SQL subset's CFG to generate a random structure (e.g., `SELECT x1 FROM x2 WHERE x3 = x4`).
3. Identify the category type of each literal placeholder variables from section 4.1 (e.g `{x2}` ∈ `tablenames`; `{x1,x3}` ∈ `attributenames`; `{x4}` ∈ `attributevalues`).
4. Replace the placeholder variables with the literal belonging to its respective category type randomly. We first bind the `table names`, followed by the `attribute names`, and finally, `attribute values`.
5. Repeat the steps 2, 3 and 4 until we get a dataset of 1250 SQL queries (750 for training and 500 for testing) from Employees and 500 SQL queries from the Yelp dataset (for testing). We use the 750 training queries from the Employees database to customize our ASR engine, Azure's Custom Speech API. We are also interested in testing the generalizabilty of our approach to new database schemas. Hence, we do not include queries from Yelp database for customizing the API.
6. Use Amazon Polly speech synthesis API to generate spoken SQL queries from these queries in text. Amazon Polly offers voices of 8 different US English speakers with naturally sounding voices. We found that voice output is of high quality even for Literals. We sampled and heard a few queries to verify this. Especially for dates, we found that Polly auto converts format 'month-date-year' to spoken dates. Polly also allow us to vary several aspects of speech, such as pronunciation, volume, pitch, and speed rate of spoken queries.

Note that our procedure for data generation applies to any arbitrary schema where `tablenames`, `attributenames` and `attributevalues` are user-pluggable. Since the steps 2, 3 and 4 of the above procedure can be repeated for infinitely many times, the procedure is scalable. We make all our dataset publicly available on our project webpage [5].

6.2 Metrics

For evaluating accuracy, we first tokenize a query text to obtain a multiset of tokens (Keywords, SplChars, and Literals). We then compare the multiset A of the reference query (ground truth SQL query) with the multiset B of the hypothesis query (transcription output from SpeakQL). We use the following error metrics: Keyword Precision Rate (*KPR*), SplChar Precision Rate (*SPR*), Literals Precision Rate (*LPR*), Word Precision Rate (*WPR*), Keyword Recall Rate (*KRR*), SplChar Recall Rate (*SRR*), Literals Recall Rate (*LRR*) and Word Recall Rate (*WRR*). For example, $WPR = \frac{|A \cap B|}{|B|}$, $WRR = \frac{|A \cap B|}{|A|}$, and the rest are defined similarly. Any incorrectly transcribed token will result in loss of accuracy and will force users to spend time and effort correcting it. Thus, we are also interested in finding out how far the output generated by SpeakQL is from the ground truth. For this purpose, we include one more accuracy metric: Token Edit Distance (TED), which allows for only insertion and deletion of tokens between the reference query and the hypothesis query. The latency is evaluated with running time in seconds.

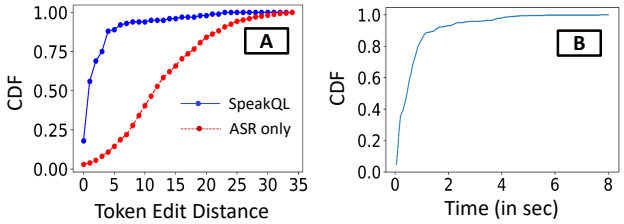


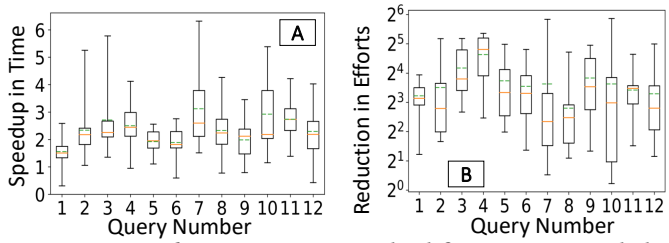
Figure 6: (A) Evaluation of SpeakQL on Token Edit Distance (B) Runtime of SpeakQL.

6.3 End-to-End Evaluation

Experimental Setup. All objective experiments were run on a commodity laptop with 16GB RAM and Windows 10. We use Cloudlab OpenStack profile with Ubuntu16.04 and 256GB RAM for running backend server during user studies [28].

Results. Table 2 reports the mean accuracy metrics for queries on the Employees and Yelp database. For additional insights, we present both top 1 outputs and "best of" top 5 outputs. We present the CDF of the accuracy metrics in the tech report [31]. We see that with SpeakQL, we achieve almost maximum possible precision and recall (mean of roughly 0.98) for Keywords and SplChars on both Employees train and test dataset. Even for Literals, the accuracy improves significantly on both databases compared to ASR. In addition, on Yelp, the precision and recall are considerably high. Since ASR is customized on the training data from Employees, SpeakQL is more likely to correctly detect its schema Literals than for other schemas. Hence, on Yelp, the fraction of relevant tokens successfully retrieved is less. This leads to a lower recall rate (mean of 0.64) for Literals.

Figure 6(A) shows the CDF of TED on the Employees test set. TED is a surrogate for the amount of effort (touches) that the user needs when correcting a query. Higher TED means more user effort. Almost 90% of the queries have TED of less



C

	Simple Queries						Complex Queries					
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12
Median time to completion	17	34.4	36.4	40	40.5	31.6	143	165.3	160.9	120	62.5	119.7
Median units of effort	5	13	7	5	8	8	37	43	19	49	14	43

Figure 7: *Simple* queries are marked from 1 to 6 and the rest are *complex*. (A) Speedup in time to completion for queries using SpeakQL vs raw typing (B) Reduction in units of efforts for queries composed with SpeakQL vs raw typing (C) Median time to completion and units of effort for queries composed with SpeakQL.

than 6. Hence, from the user end, correcting most queries require only a handful of touches. Figure 6(B) shows the CDF of latency of SpeakQL. We notice that for almost 90% of the queries on Employees, the runtime is well within 2 seconds and only 1% of the queries took more than 5 seconds.

6.4 User Study

Setup. We choose a tablet device with 2GB RAM and 1.6GHz processor for the user study. We prioritize users and queries on the tablet to get more confidence for our results. Thus, we leave study with phones to future work. Since typing on phones can be even harder, the study with a tablet would give a lower bound on the benefits of our system. We conducted a preliminary user study that helped us learn several key lessons in making our interface more speech-friendly and correction-friendly. We describe the pilot study and the lessons learned in our technical report [31].

Actual User Study. We conduct user study with 15 participants where the recruitment was conducted through a short SQL quiz. Each participant is first made familiar with our interface through an introductory video [6]. Each participant composes 12 queries (q_1 to q_{12}) on a browser-based SpeakQL interface on the tablet given the natural language description of the query along with the schema. We compare two conditions for specifying the query with a within-subjects design. In the first condition, the participant has access to our SpeakQL interface that allows them to dictate the SQL query and perform interactive correction. In the second condition, the participant types the SQL query from scratch with no access to our interface. We record the time to complete the query for both the conditions. Also, we log every interaction of the user with our system, i.e., the number of corrections and re-dictation attempts. We evaluate our system using 180 data points (15 participants, 12 queries).

Study Design. The queries were divided into two segments: *simple* and *complex*. We define *simple* queries as those with less than 20 tokens; the rest are considered *complex*. Thus, composing a *complex* query imposes a higher cognitive load relative to a simple query. Participant p_1 was asked to speak query q_1 first and type q_1 next. p_1 will then type q_2 first and dictate q_2 next. We alternate this order across the 12 queries. Similarly, this order is alternated across participants, i.e., p_2 will type query q_1 first and dictate q_1 next. This design lets us account for the interleaving of thinking and speaking/typing when constructing SQL queries and reduce the bias caused by a reduced thinking time when re-specifying the same query in a different condition (typing or speaking).

Results. Figure 7 shows the median time to completion with SpeakQL, median units of efforts spent on our interface, speedup in time to completion (i.e., time to completion of typing vs time to completion of SpeakQL), and reduction in efforts for the 12 queries. The queries from 1-6 are *simple* and the rest are *complex*. Units of effort is defined as number of touches/clicks (including keyboard strokes) or dictation/re-dictation attempts made when composing a query. The main takeaways are given below.
(1) Plot A shows that SpeakQL leads to significantly higher speedup in time to completion compared to raw typing. The speedup is higher for *complex* queries (average of 2.9x) than the *simple* ones (average of 2.4x).
(2) Plot B shows that SpeakQL leads to significantly less units of effort than raw typing. The average reduction factor is 12x and 7.5x for *simple* and *complex* queries respectively.
(3) From table C we notice that the median time to completion and units of effort for the *complex* queries is considerably higher than the *simple* ones, which is expected.

Hypothesis Tests. Hypothesis tests shows that the time to complete a query, the time spent editing a query, and the total units of efforts with SpeakQL is statistically significantly lower than than the typing condition. We discuss the tests in depth in our technical report [31].

6.5 Component-level Drill Down

Structure Determination Evaluation. We evaluate the structures returned by this component relative to the ground truth structure. Figure 8(A) shows the CDF of TED for the Employees test set queries. The correct structure is delivered for about 86% of the queries. We report the CDF of this component's latency in the tech report [31].
Literal Determination Evaluation. Figure 8(B) presents the CDF of recall rates for table names, attribute names, and

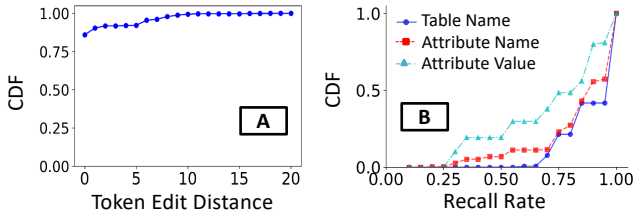


Figure 8: (A) Structure Determination component evaluation on Token Edit Distance (TED) (B) Literal Determination component evaluation. CDF of Recall Rates for different Literal types.

attribute values. We see that recall rates for table names and attribute names are considerably high, with a mean of 0.90 and 0.83, respectively. But for attribute values, recall rate is low (mean of 0.68). To see why this is the case, we present the CDF of edit distance for different type of attribute values with the ground truth in the technical report [31].

6.6 Comparison with NLIs

We compare SpeakQL against state-of-the-art NLIs with typed and speech inputs on two large-scale human-annotated datasets containing pairs of natural language and SQL queries: Spider [38] and WikiSQL [39]. We find that the accuracy of NLIs decreases significantly when queries are speech-based than typing-based due to a variety of errors in the transcription. Moreover, we observe that SpeakQL achieves significantly higher accuracy than the state-of-the-art NLIs with speech inputs. For instance, lift of 50% in execution accuracy on WikiSQL. We present the complete evaluation and additional insights in the tech report [31].

7 RELATED WORK

Speech-driven Querying Systems. Speech recognition for data querying has been explored in some prior systems. Nuance's Dragon Naturally Speaking allows users to query using spoken commands to retrieve the text content of a document [3]. Several systems such as Google's Search by Voice [30, 33] and Microsoft's Model M [41] have explored the possibility of searching by voice. Conversational assistants such as Alexa, Google Home, Cortana, and Siri allow users to query over only an application-specific knowledge bases and not over an arbitrary database. In contrast, SpeakQL allows users to interact with structured data using spoken queries over any arbitrary database schema.

Other Non-typing Query Interfaces. Query Interfaces that help non-technical users explore relational databases have been studied for several decades. There has been a stream of research on visual interfaces [4, 15, 40]. Tabular tools such as [40] allow users to query by example, [4] allows users to create drag-and-drop based interfaces, and keyword-search based interfaces such as [15] help users formulate SQL queries by giving query suggestions. More recently, non-keyboard based touch interfaces [13, 18, 24, 25, 35] have received attention because of the potentially lower user effort to provide input. At the user level, almost all of these query interfaces obviate the need to type SQL. This rich body of prior work inspired our touch-based multimodal interface for query correction that augments spoken input. But unlike these tools, our first version of SpeakQL does not aim to obviate SQL but rather embraces and exploits its persistent popularity among data professionals.

Natural Language Interfaces. There is a long line of work on NLIs for databases in order to allow layman users to ask questions in natural lanuage [16, 17, 21, 29, 34, 36–39]. NLIs are orthogonal to this paper's focus. Inspired from regular human to human conversations, Echoquery [21] is designed as a conversational NLI in form of an Alexa skill. Although, this system certainly enables non-experts to query data easily and directly, ASR can cause a series of errors and would restrict users from specifying "hard" queries. In addition, such a system might impose a higher cognitive load [23, 27] on users when a large query result is returned; a screen mitigates such issues, e.g., as in the Echo Show.

Natural Language Processing (NLP). Recent work in NLP community has emphasized the fact that incorporating linguistic structure can help prune the space of generated queries and thus help in avoiding the NLU problem [12, 14, 19, 22, 39]. This recent trend of incorporating structural knowledge into the modeling offers a form of validation for our approach of directly exploiting rich structure of SQL using its grammar.

8 CONCLUSIONS AND FUTURE WORK

We pursue an exploratory research direction on speech-driven query interfaces that is complementary to NLIs and visual interfaces. Inspired by our conversations with diverse data querying professionals, we build the first end-to-end multimodal querying system for a practical subset of SQL that combines speech and touch interactions. Our empirical findings suggest that SpeakQL achieves significant improvements over ASR on all accuracy metrics. Through user studies, we show that our system helps users to speed up their SQL query specification process. As for future work, we would like to modify SQL itself to make it more speech-friendly. Our empirical results show that Literals are the biggest bottleneck for accuracy. Hence, we plan to rewrite our SQL subset's CFG in a manner that focuses more on literals and de-emphasizes structure.

Acknowledgments. This material is based upon work supported in part by the National Science Foundation under Grant No. IIS-1816701. We thank the members of UC San Diego's Database Lab and Ndapa Nakashole for their feedback on this work.

REFERENCES

[1] Accessed April 12, 2020. Alexa commands. https://www.cnet.com/how-to/amazon-echo-the-complete-list-of-alexa-commands.

[2] Accessed April 12, 2020. MySQL Employees Sample Database. Available from https://dev.mysql.com/doc/employee/en/.

[3] Accessed April 12, 2020. Nuance's Dragon Speech Recognition. https://www.nuance.com/dragon.html.

[4] Accessed April 12, 2020. Oracle SQL Developer. blogs.oracle.com/smb/what-is-visual-builder-and-why-is-it-important-for-your-business.

[5] Accessed April 12, 2020. *SpeakQL Project Webpage.* https://adalabucsd.github.io/speakql.

[6] Accessed April 12, 2020. *SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data (Tutorial Video).* https://youtu.be/KsgNo-CkE8Y.

[7] Accessed April 12, 2020. SQL Grammar. http://forcedotcom.github.io/phoenix.

[8] Accessed April 12, 2020. Yelp Database. Available from https://www.kaggle.com/yelp-dataset/yelp-dataset.

[9] Christopher Baik, HV Jagadish, and Yunyao Li. 2019. Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 374–385.

[10] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. DBPal: A Learned NL-Interface for Databases. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1765–1768. https://doi.org/10.1145/3183713.3193562

[11] Dharmil Chandarana, Vraj Shah, Arun Kumar, and Lawrence Saul. 2017. SpeakQL: Towards Speech-driven Multi-modal Querying. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*. ACM, 11:1–11:6. https://doi.org/10.1145/3077257.3077264

[12] Trevor Cohn, Cong Duy Vu Hoang, Ekaterina Vymolova, Kaisheng Yao, Chris Dyer, and Gholamreza Haffari. 2016. Incorporating Structural Alignment Biases into an Attentional Neural Translation Model. *arXiv preprint arXiv:1601.01085* (2016).

[13] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive Analytics Through Pen and Touch. *Proceedings of VLDB Endowment* 8, 12 (2015), 2024–2027. https://doi.org/10.14778/2824032.2824127

[14] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent Neural Network Grammars. *arXiv preprint arXiv:1602.07776* (2016).

[15] Ju Fan, Guoliang Li, and Lizhu Zhou. 2011. Interactive SQL Query Suggestion: Making Databases User-friendly. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 351–362.

[16] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. *arXiv preprint arXiv:1905.08205* (2019).

[17] Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. *arXiv preprint arXiv:1902.01069* (2019).

[18] Stratos Idreos and Erietta Liarou. 2013. dbTouch: Analytics at your Fingertips. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper13.pdf

[19] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. 2017. Structured Attention Networks. *arXiv preprint arXiv:1702.00887* (2017).

[20] Fei Li and HV Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proceedings of the VLDB Endowment* 8, 1 (2014), 73–84.

[21] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. 2016. Making the Case for Query-by-Voice with EchoQuery. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2129–2132.

[22] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*. Association for Computational Linguistics, 1506–1515. https://doi.org/10.18653/v1/d17-1159

[23] George A Miller. 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological review* 63, 2 (1956), 81.

[24] Arnab Nandi. 2013. Querying Without Keyboards. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*.

[25] Arnab Nandi, Lilong Jiang, and Michael Mandel. 2013. Gestural Query Specification. *Proceedings of the VLDB Endowment* 7, 4 (2013), 289–300.

[26] Saul B Needleman and Christian D Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.

[27] T. V. Raman. 1998. *Audio System for Technical Readings*. Lecture Notes in Computer Science, Vol. 1410. Springer. https://doi.org/10.1007/BFb0054977

[28] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[29] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. 2016. ATHENA: an Ontology-driven System for Natural Language Querying over Relational Data Stores. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1209–1220.

[30] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. 2010. "Your word is my command": Google Search by Voice: A Case Study. In *Advances in speech recognition*. Springer, 61–90.

[31] Vraj Shah, Side Li, Arun Kumar, and Lawrence Saul. Accessed April 12, 2020. *SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data.* https://adalabucsd.github.io/papers/TR_2020_SpeakQL.pdf.

[32] Vraj Shah, Side Li, Kevin Yang, Arun Kumar, and Lawrence Saul. 2019. Demonstration of SpeakQL: Speech-driven Multimodal Querying of Structured Data. In *Proceedings of the 2019 International Conference on Management of Data*. 2001–2004.

[33] Jiulong Shan, Genqing Wu, Zhihong Hu, Xiliu Tang, Martin Jansche, and Pedro J Moreno. 2010. Search by Voice in Mandarin Chinese. In *Eleventh Annual Conference of the International Speech Communication Association*. 354–357.

[34] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from Unstructured Keywords as Queries to Structured Databases as Answers. *The VLDB Journal—The International Journal on Very Large Data Bases* 17, 1 (2008), 117–149.

[35] Pawel Terlecki, Fei Xu, Marianne Shaw, Valeri Kim, and Richard Wesley. 2015. On Improving User Response Times in Tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1695–1706.

[36] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SqlNet: Generating Structured Queries from Natural Language without Reinforcement Learning. *arXiv preprint arXiv:1711.04436* (2017).

[37] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 63:1–63:26.

[38] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. 3911–3921.

[39] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103* (2017).

[40] Moshé M. Zloof. 1975. Query by Example. In *National Computer Conference and Exposition.*

[41] Geoffrey Zweig and Shuangyu Chang. 2011. Personalizing Model M for Voice-Search. In *Twelfth Annual Conference of the International Speech Communication Association*. ISCA, 609–612.