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. Structured data querying is ubiquitous in the enterprise, healthcare, and other domains. Typing queries in SQL is the gold standard for such querying. But typing SQL is painful or even impossible in the above device environments, which restricts when and how users can 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. Our target is users with some familiarity of SQL. 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 schema. Our experiments show that SpeakQL can automatically correct a large fraction of the errors in ASR transcriptions. User studies show that SpeakQL can help users specify SQL queries faster with a speedup of average 2.7x and up to 6.7x relative to raw typing. It also saves a significant amount of user effort with a speedup of average 10x and up to 60x relative to raw typing. This work is a step towards a larger vision of making structured data querying more speech-friendly.**

I. Introduction

Speech-based inputs have seen widespread adoption in many applications on emerging device environments such as smartphones, tablets, and even personal conversational assistants such as Alexa. Inspired by this recent success of speech-driven interfaces, in this work, we consider an important fundamental question: *How should one design a speech-driven system to query structured data?*

Recent works have studied new querying modalities like visual [1], [2], touch-based [3], [4], and natural language interfaces (NLIs) [5], [6], for constrained querying environments such as tablets, smartphones, and conversational assistants. The commands given by the user are translated to SQL. Hence, this would allow users to query without specifying 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?* Structured data querying is widely used in domains such as enterprise, Web, and healthcare. 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 consumers of the data such as business analysts, nurse informaticists and managers. From conversations with such users, we find that they desire anytime and anywhere access to their data, but their SQL knowledge is ignored by visual or NLI research. Even SQL expert DBAs sometime desire off-hour on-the-go access to their data. We believe a spoken structured querying interface can bridge such crucial gaps in querying capabilities. We illustrate the potential benefits of spoken querying system based on our conversations with case studies below.

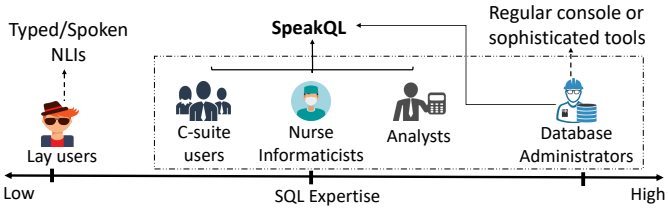

Fig. 1: Contrasting SpeakQL's goals with current NLIs and other sophisticated tools in terms of SQL expertise of users.

Case Study 1. A DBA is interested in database (DB) monitoring, identifying system issues, and maximizing its uptime. He gets an email alert when a DB goes down or some exception occurs. To resolve the issue, he composes one or two SQL queries such as: `SELECT * FROM logs WHERE eventname='exception'`. This can be done easily when he has a console on a workstation/laptop. But DB issues happen around the clock. The spoken query system via his phone/tablet can enable him to provide a quicker support.

Case Study 2. A nurse informaticist at an in-home healthcare service company is familiar with basic SQL. He often visits the home of elderly people who are unable to visit hospitals for treatments or emergencies. He wants to query the patient's medical history to aid in diagnosis. He has no access to patient's database records outside of his desk. Thus, he carries physical files with sensitive patient data. Spoken query system on his tablet/phone can potentially help him speed up the retrieval of relevant patient's information instead of scanning through the files.

Case Study 3. An analyst at an e-commerce company meets with their business partners in the marketing department on

a daily basis. The marketing folks are thinking of a new campaign to raise sales in a particular region. They want to know the total sales in that region in the past 5 years. But the analyst has to go back to his computer to compute this and then report it to the team. Spoken query system can help such meetings become more productive.

Lessons from case studies. We find that many users in industry want anytime and anywhere access to their data on mobile platforms such as smartphones and tablets. However, typing SQL is really painful in such constrained settings. Having a spoken SQL interface can help them speed up their query specification process.

Why speak SQL? To answer our earlier question about why dictate structured queries and not just use NLIs, 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. On the other hand, NLIs are primarily aimed at lay users and not professionals who work with structured data. Moreover, NLIs may not offer any guarantees on the correctness of results. But with structured queries, users know the results will match their query. Thus, instead of forcing all users to only use NLIs, we study how to leverage ASR for SQL and make spoken querying more "natural" without losing SQL's benefits. In this work, as the first major step in our vision, we build a spoken querying system for a subset of SQL. We call this system SpeakQL. Since current NLIs are increasingly relying on more keywords and structured interactions [5], [7], we believe our lessons can potentially also help improve NLIs in future.

Desiderata. Complementary to existing visual interfaces, NLIs, and touch-based interfaces, we desire the following. (1) Support regular SQL with a tractable subset of the CFG, although our architecture and methods will be applicable to any SQL query in general. (2) Leverage an existing modern state of the art Automatic Speech Recognition (ASR) technology instead of reinventing the wheel. (3) Support queries in any application domain over any database schema. (4) Support multimodal interactive query correction using touch (or clicks) and speech with 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: First, ASR introduces myriad forms of errors when transcribing. For instance, errors can arise due to homophony, e.g., "`sum`" vs. "`some`". Second, 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. Finally, 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 placeholder 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 more correction and speech-friendly respectively. 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 offer 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.

Experimental Evaluation. Since there is no known publicly available dataset of spoken SQL queries, we create the first such dataset using real-world database schemas. Using several accuracy metrics such as precision, recall and edit distance, we show that SpeakQL can correct large proportions of errors in the transcriptions produced by a modern ASR. For example, we noticed an average lift of 21% in Word Recall Rate. Empirically, we show that SpeakQL can achieve real-time latency and can potentially reduce a significant amount of user effort in correcting the query. Through user studies, we show that SpeakQL allows users to compose queries significantly faster, achieving a speedup of average 2.7x and upto 6.7x relative to raw typing. Moreover, with SpeakQL, users require less effort in specifying the query, achieving a speedup of average 10x and upto 60x relative to the raw typing.

Overall, the contributions of this paper are as follows:

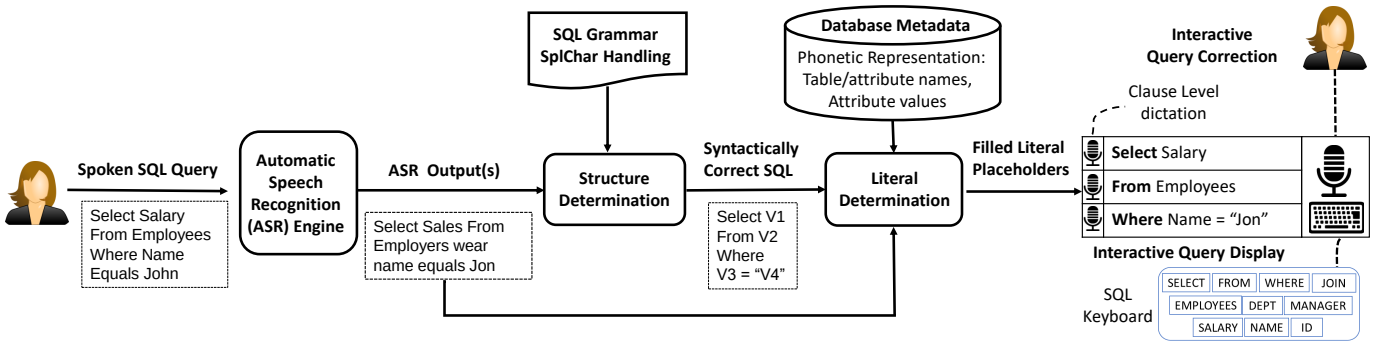

Fig. 2: End-to-end Architecture of SpeakQL. We show an example of a simple spoken SQL query, and how it gets converted to a query displayed on a screen, which the user can correct interactively.

- 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 correct a significant portion of errors in ASR transcription. Moreover, our user studies shows that SpeakQL helps users reduce time and effort in specifying SQL query significantly.

II. 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 [8]. 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. Due to homophones, the ASR might convert Literals into Keywords or SplChars and vice versa. For example, SQL keyword `sum` detected as "some." Even a single-token transcription might be completely wrong because the token is simply not present in the 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 structure of a SQL query. 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 and we present the complete four-component end-to-end system in Figure 2. *This decoupling of structure and literal determination is a critical design decision that helps us tackle the unbounded vocabulary problem*. The entire system has four major components as described below.

ASR Engine. This component processes the recorded spoken SQL query to obtain a transcription output. A modern speech recognition system consists of two major components: acoustic model and the language model. The acoustic model captures the representation of sounds for words, and a 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 Section 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. It exploits CFG of our currently supported subset of SQL to generate all possible ground truth structures. A ground truth structure is a syntactically correct SQL string obtained from our SQL grammar by applying the production rules recursively. The closest matching structure is retrieved by doing a similarity search based on edit distances with the ground truth structures. 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.

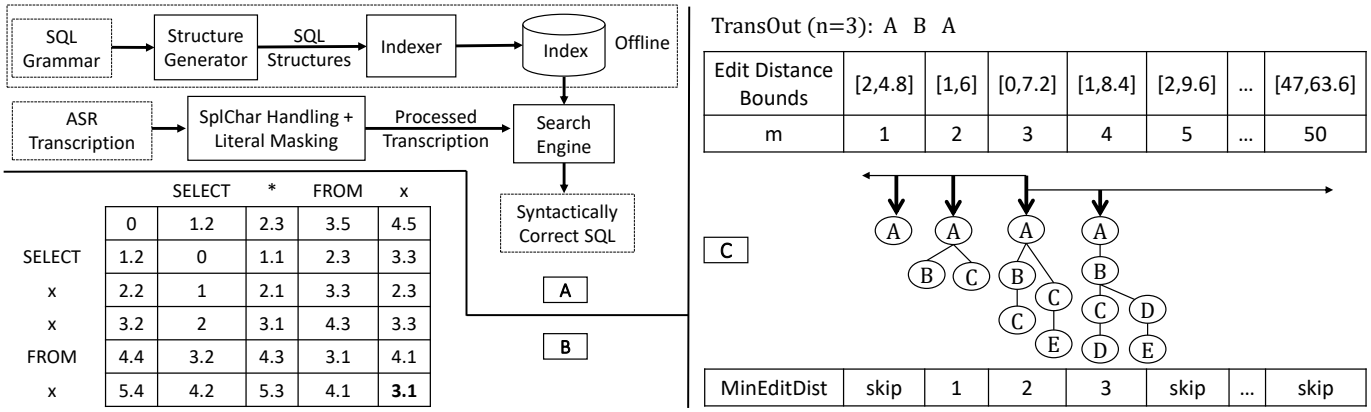

Fig. 3: (A) Structure Determination Component Architecture (B) Dynamic Programming Memo that computes edit distance between the `MaskOut` and `GrndTrth` (C) Bidirectional Bounds (BDB) Example

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. This component displays a single SQL statement that is the best possible transcription generated by our system. Even with our query determination algorithms, it might turn out that some of the tokens in the transcription are incorrect, especially for Literals not present in ASR vocabulary ("out-of-vocabulary" Literals). Thus, we support user-in-the-loop interactive query correction through speech or touch/click-based mechanisms. The user can either choose to dictate/re-dictate queries at the clause level or make use of a novel SQL keyboard tailored to reduce their effort in correcting the displayed query. The database's schema will always be displayed to help users pose their queries. Section 5 explains the interface in depth.

III. Structure Determination

We now discuss the technical challenges of the structure determination component and present our algorithms to tackle them. The goal of this component is to get a syntactically correct SQL statement given transcription output from ASR as input. Figure 3(A) presents the architecture of this component. Next, we describe each sub-component.

Supported SQL Subset. In this first work on this problem, for tractability sake, we restrict ourselves to only a subset of regular SQL Data Manipulation Language (DML) that is meaningful and practically useful for spoken data retrieval and analysis. Our currently supported subset includes most Select-Project-Join-Aggregation (SPJA) queries along with `LIMIT` and `ORDER BY`, without any limits on the number of joins or aggregates, as well as on predicates. We do not currently support nested queries or queries belonging to Data Definition Language (DDL). We use the production rules of `SELECT` statements of standard SQL in BNF (Backus-Naur Form) [8]. Due to space constraints, we provide the grammar in our technical report [9]. We find that our 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 reinforcement learning-based NLI Seq2SQL [10] allows queries over only one table and with only one aggregate.

A. SplChar Handling and Literal Masking

We create a dictionary of the supported SQL Keywords and SQL SplChars, namely, `KeywordDict` and `SplCharDict`. The two dictionaries are given below:

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

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

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

B. Structure Generator

This component would apply the production rules of SQL grammar recursively to generate a sequence of tokens. This sequence of tokens is a string representing a SQL ground truth structure. Since the number of tokens that can be generated is potentially infinite, we restrict the number of tokens in the string to a maximum of 50. This leads to generation of roughly 1.6M ground truth structures. Our basic idea is to compare `MaskOut` with these set of generated ground truth structures and select the ground truth string that has minimum edit distance value. *Thus, the knowledge of the grammar let*

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 that can be present in the ASR output 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.

C. Indexer

Clearly, comparing the transcription output with every ground truth string is infeasible 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 of those can 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, such structured trees can 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 about the length of strings in the trie as explained in the 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. Thus, we have 50 disjoint tries in all.

D. Search Engine

Given a `MaskOut`, the search engine aims to find the closest matching structure by comparing against the ground truth strings from the index. This comparison is based on edit distance, a popular way to quantify similarity between two strings by counting the number of operations to transform one string into another. There are many variants of edit distance differing in the set of operations involved. We use a weighted longest common subsequence edit distance [11], 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 the ASR engine 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 weight W_S to SplChars, and lowest weight W_L to Literals. We choose $W_K = 1.2, W_S = 1.1$ 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, these fixed weights suffice for our purpose. Figure 3(B) shows the matrix with edit distance between `MaskOut` string `SELECT x x FROM x` and ground truth (`GrndTrth`) string `SELECT * FROM x` computed using the dynamic programming approach.

Algorithmus 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
```

We introduce some notation to explain our weighted edit distance. 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 programming algorithm to compute this matrix. We observe that computing `dp(i,j)` requires using only the previous column (`DpPrvCol`) and current column (`DpCurCol`) of the matrix. Moreover, if for a node n, `min(DpCurCol) > MinEditDist`, then we can stop exploring it further. Given this behavior of the dynamic program, we now present an optimization that can reduce the computational cost of searching over our index.

Bidirectional Bounds (BDB). 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 result:

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, Figure 3(C) shows an example. `TransOut` is a string of length 3: `A B A`. Ground truth strings are indexed from keys 1 to 50 by their length (m). The first row denotes the range of possible edit distances with `TransOut`. We first go in the direction of decreasing m from $m = 3$ to $m = 1$. We start comparisons of strings in the Trie for $m = 3$ with `TransOut`. We find that the `MinEditDist` is 2 with string `A B C`. With $m = 2$, the lower bound on edit distance is 1, which is less than `MinEditDist` found so far. Hence, we explore the trie for $m = 2$. We find that the `MinEditDist` is 1 with string `A B`. With `m=1`, the lower bound on edit distance is 2, which is more than `MinEditDist` found so far. Hence, there's no

way we can find a ground truth string in the Trie that can deliver MinEditDist. Thus, we skip its exploration. In the next pass, we go from $m = 4$ to $m = 50$. When $m > 4$, we find that MinEditDist $>$ 1. Hence, we skip all the tries for values of m from 5 to 50.

Overall Search Algorithm. The central idea of this algorithm is to skip searches on tries that were pruned out 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 programming algorithm to calculate edit distance with TransOut, and build a column of the memo as shown in Figure 3(B). 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. The worst case time complexity of the algorithm 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 [9]. We propose two additional accuracy-latency tradeoff algorithms that further reduce runtime by trading off some accuracy. We present them in our technical report [9]. Note that, we do not use these techniques by default in SpeakQL, but users can choose to enable them, if they want even lower latency.

IV. Literal Determination

The central goal of the Literal Determination 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 the Literal Determination component because the most prominent information that it can use to identify a literal for any placeholder variable is the raw transcription output obtained from ASR. 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. These observations about how ASR fails to generate correct transcription helps us to identify 2 crucial design choices for the Literal Determination.

(1) In contrast to string-based similarity search, a similarity search on a pre-computed *phonetic representation* of the existing Literals in the database can help us disambiguate the words from TransOut that sounds 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

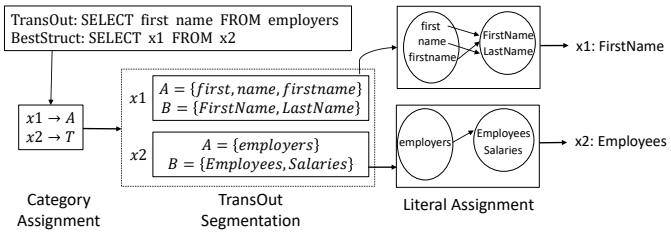

Fig. 4: Literal Determination component example

(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) The Literal Determination component has to be made aware of the splitting of tokens into sub-tokens, so that it can decide when and how to merge these sub-tokens. Figure 4 shows the workflow of the Literal Determination component with TableNames= {Employees (EMPLYS), Salaries (SLRS)} and AttributeNames= { FirstName (FRSTNM), LastName (LSTNM)}.
The inputs given to the Literal Determination component are TransOut and best structure (BestStruct) obtained from the Structure Determination component. 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 variable. 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 representation of the two sets A and B. This algorithm is described in depth in our technical report [9]. The worst-case time complexity of the algorithm is $O(n^2m)$, where n is the length of TransOut and m is the domain size of Literals. The space complexity is $O(n^2+m)$.

A. Category Assignment

We constrain the space of possible Literals to consider for any given placeholder variable in BestStruct. Each placeholder variable can be a table name (category type = T), an attribute name (category type = A) or an attribute value (category type = V). Using SQL grammar, we assign a category type to the placeholder variable. 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.

B. Transcription Output Segmentation

In the previous step, we identified a set of possible Literals that can take up the value of a placeholder variable. But still, we have not found the exact literal to "fill in". This

requires using raw TransOut to identify transcribed tokens for Literals. In this step, we segment TransOut such that only relevant tokens are retrieved to be compared against items in set B. For a placeholder variable in BestStruct, we first identify a window in TransOut, denoting where the literal in TransOut 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. The maximum possible length of the substring is given by a *WindowSize* parameter. *WindowSize* needs to be tuned based on how many sub-tokens ASR splits a single token into. For example, a token like CUSTID_1729A can be split into 7 sub-tokens ({CUSTID, _, 1, 7, 2, 9, A}). Thus, setting *WindowSize* of 7 lets our algorithm merge all sub-tokens into one. This also presents a tradeoff with the running time, i.e., increasing *WindowSize* would lead to a higher accuracy but also an increase in latency. From our initial experiments with the dataset mentioned in Section 6.1, we found that ASR often splits many schema Literals into 2 or 3 subtokens. Thus, we set *WindowSize* = 3. In our running example, for placeholder variable x1, set A is given by {first, name, firstname}, while set B is the set of attribute names.

C. Literal Assignment

As 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. One straightforward approach is to do an all pairs comparison to retrieve the item in set B that gives the minimum edit distance with any item in set A. However, this approach does not necessarily give the correct desired literal. We observe that ASR is likely to break apart a large token into a series of sub-tokens with some sub-tokens erroneously transcribed. Hence, there can exist a literal that has minimum edit distance with a correctly transcribed sub-token but not necessarily overall. Moreover, resolving ties with this approach is non-trivial and requires more tedious heuristics. Two examples below illustrates this issue.

Example 1. Set A = {FRONT (FRNT), DATE (TT), FRONTDATE (FRNTTT)} and set B = {FROMDATE (FRMTT), TODATE (TTT)}. The ground truth literal is FROMDATE. But, the minimum edit distance occurs between pair DATE and TODATE. Hence, the approach of returning a literal in set B that gives minimum edit distance with any item in set A would simply not work.

Example 2. Set A = {RUM (RM), DATE (TT), RUMDATE (RMTT)} and set B = {FROMDATE (FRMTT), TODATE (TTT)}. The ground truth literal is again FROMDATE. However, both FROMDATE and TODATE give minimum edit distance of 1 with RUMDATE and DATE respectively. To resolve this tie, we can use an additional information that RUM has less edit distance with FROMDATE, than with TODATE. Hence, now we have 2 out of 3 items in set A for which edit distance with FROMDATE is less than edit distance with TODATE. This would help us retrieve FROMDATE with a greater confidence. Inspired by this, we propose the following literal voting algorithm.
(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 that wins the maximum number of votes. Literals with the next highest votes will be the second returned literal, and similarly we fetch top k Literals for each placeholder variable. If there exist ties in votes, we resolve it in lexicographical order. In our running example, the returned literal for the placeholder variable x1 is FirstName, while for x2 is Employees.

V. Interface

Figure 5(A) shows the SpeakQL interface. 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. +/− buttons allows for easy insertion/deletion of Keywords and Literals from the query. The notable </> button allows for a quick insertion or removal of SplChars. If the displayed literal is incorrect, the user can touch its box and a drop-down menu will display the ranked lists of alternatives for that placeholder.

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. 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-driven or touch-driven interactive query correction.

VI. Experimental Evaluation

In this section, we present a thorough empirical evaluation of our system. We start with our procedure to generate dataset of spoken SQL queries. This procedure applies to any arbitrary database schema and is scalable. We then define the accuracy and runtime metrics used for evaluation and evaluate SpeakQL end-to-end on these metrics. In addition, we dive deeper into evaluation of each of our components. Finally, we present our

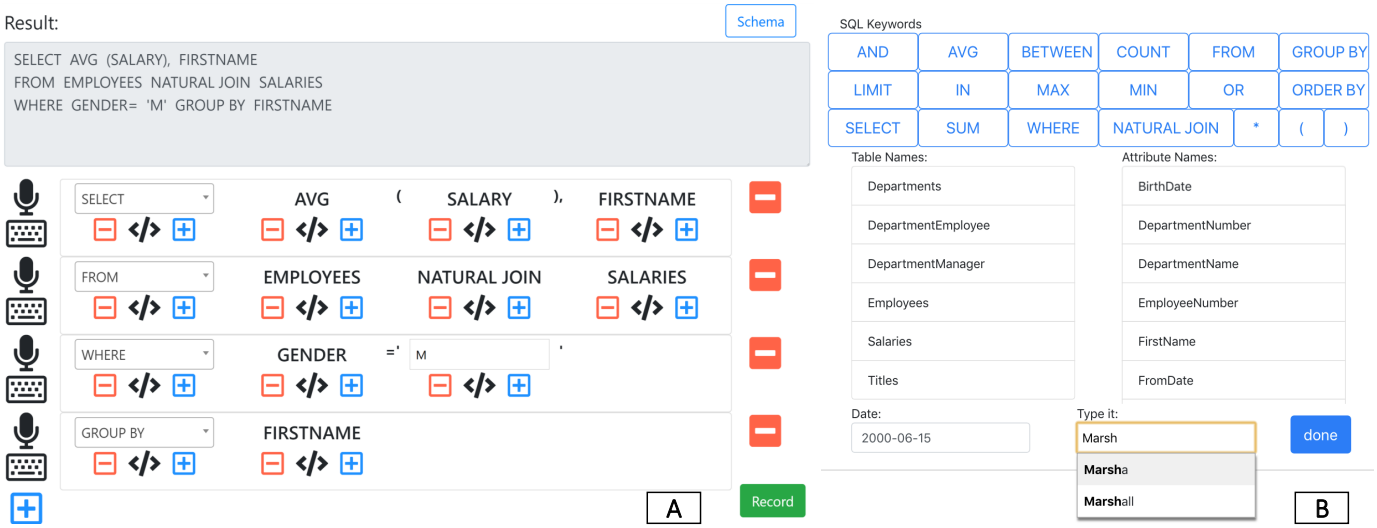

Fig. 5: SpeakQL Interface. (A) The Interactive Display showing the dictated query after being processed by the SpeakQL engine, as well as the touch-based editing functionalities and clause-level redictation capabilty. (B) Our simple SQL keyboard designed for touch-based editing of the rendered query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

findings from the actual user studies that show that SpeakQL can help reduce the query specification time significantly.

A. Data

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

1. We use two publicly available database schemas: Employees Sample Database from MySQL [12] and the Yelp Dataset [13]. We get the `table names`, `attribute names`, and `attribute values` in each database.
2. Use our unambiguous SQL subset context free grammar described in Section 3.2 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`$\} \in$ `tablenames`; $\{$`x1,x3`$\} \in$ `attributenames`; $\{$`x4`$\} \in$ `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 purpose and 500 for testing purpose) from the Employees database and 500 SQL queries from the Yelp database (for testing purpose). 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 speeech 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 value Literals. We sample and hear a few queries to verify this. Especially for dates, we found that Polly auto converts textual 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.

This procedure for the generation of data 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 infinite number of times, the procedure is scalable. *All our datasets are available for download on our project webpage [9].*

B. 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 error metrics defined in [14]: 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 [11]. For evaluating latency, we simply use the running time in seconds.

C. End-to-End Evaluation

Experimental Setup. All experiments were run on a commodity laptop with 16GB RAM and Windows 10. We use Cloudlab [15] for running backend server for the user studies.

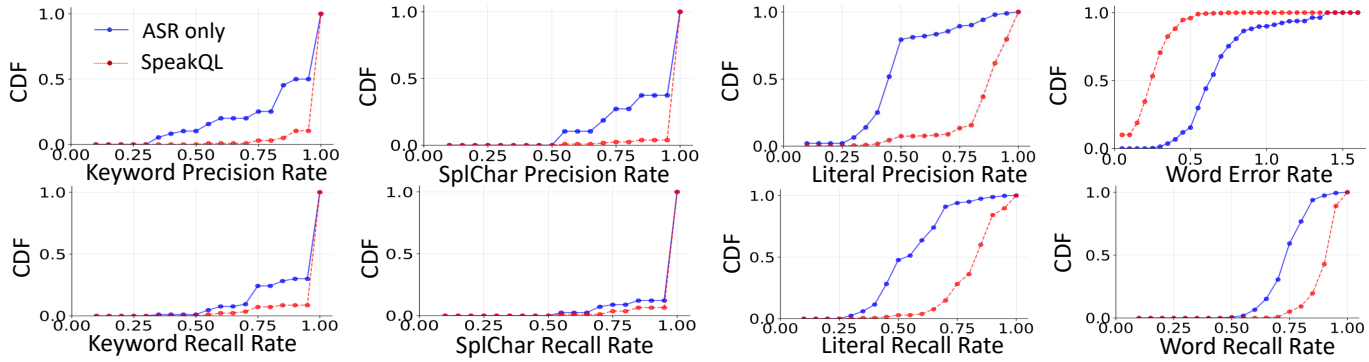

Fig. 6: Cumulative distribution of accuracy metrics for top 1 results from Employees test dataset.

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

We use a custom OpenStack profile running Ubuntu 16.04 with 256GB RAM. For the frontend, we use the chrome browser of Samsung Tablet that has 2GB RAM and 1.6GHz Processor. For ASR, we use Azure's Custom Speech Service because it allows us to customize the language model of the speech recognizer. Hence, training a language model with the dataset of SQL queries would allow us to capture the vocabulary of the underlying application. Employees training dataset of 750 queries is used to train the language model. Due to space constraints, we present the comparison of Azure's Custom Speech Service with other ASR tools in our tech report [9].

Results. Figure 6 plots the cumulative distribution functions (CDF) of the error metrics for the queries in Employees test dataset. Table I reports the mean error metrics for the same. For additional insights, we present the results for both top 1 outputs and "best of" top 5 outputs both from the structure determination and literal determination. We can see that the recall rates are already high for Keywords (mean of roughly 0.92) and SplChars (mean of roughly 0.96) using just the ASR. For Literals, however, the recall rate is quite low (mean of 0.53). 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 datasets. In addition, on Yelp dataset as well, the precision and recall are considerably high. Since the ASR is customized on the training data from the Employees schema, SpeakQL is more

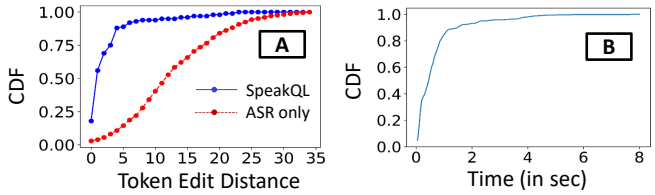

Fig. 7: (A) Evaluation of SpeakQL on Token Edit Distance (B) Runtime of SpeakQL.

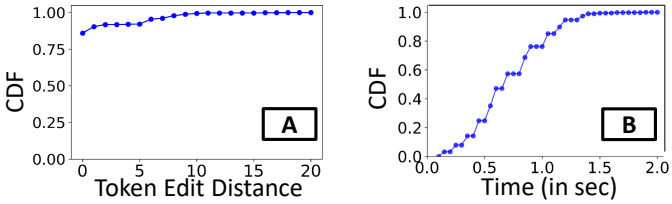

Fig. 8: Structure Determination component evaluation. (A) Token Edit Distance (B) Latency.

likely to correctly detect the Employees schema Literals than any other schema literal. Hence, on the Yelp test dataset, the fraction of relevant tokens successfully retrieved is less. This leads to a lower recall rate (mean of 0.64) for Literals.

Figure 7(A) shows the CDF of TED for the Employees test set. Here, TED is a surrogate for the amount of effort (touches or clicks) that user needs to put in when correcting a query. Higher TED means more user effort. We saw that only 23% of the queries were successfully transcribed fully. The reasons for this would become clear in our evaluation of the literal determination component later. However, almost 90% of the queries have TED of less than 6. Hence, from the user end, correcting most of the queries would require only a handful of touches or clicks. This underscores the importance of having an interactive system that allows for correction. The CDF of the latency of SpeakQL is given in Figure 7(B). We notice that, for almost 90% of the queries in the Employees test set, the final transcription output can be obtained well within 2 seconds. Only 1% of the queries took more than 5 seconds.

D. Structure Determination Drill Down

In this evaluation, we would like to see how correct is the structure returned by this component relative to the ground truth structure. Figure 8(A) shows the CDF of the token edit distances for the queries in the Employees test set. We see that this component delivers the correct structure (TED of 0)

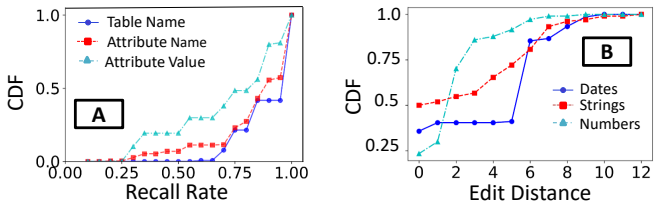

Fig. 9: (A) CDF plot of Recall Rates for different types of Literals (B) CDF plot of edit distances for different types of attribute values from ground truth. Strings are evaluated with phonetic edit distance, while Dates and Numbers are evaluated with character level edit distance.

for about 86% of the queries in the test set. Almost 99% of the structures are within a TED of 10. Figure 8(B) shows the CDF of time taken by this component. We see that for almost 80% of the queries, the structure was determined in less than 1 second. Moreover, the latency is less than 1.5 second for almost 99% of the queries.

E. Literal Determination Drill Down

First, we would like to see what fraction of the relevant Literals are retrieved by our literal determination component. Figure 9(A) presents the CDF plot of recall rates for table names, attribute names and attribute values. We notice that the 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, the recall rate is low (mean of 0.68). To see why exactly this is the case, we break down attribute value into different types.

The attribute values can either be a date, a real number or a string value. Figure 9(B) shows CDF of the edit distance for different types of attribute values from the ground truth. This is to show how much editing effort can potentially be required by the user to correct an attribute value. For example, almost 50% of the attribute values of type string were correctly retrieved with a phonetic edit distance of 0. Thus, no effort in terms of correcting would be required by the user. On the other hand, we found that only 35% of dates were returned perfectly as intended. About 85% of the dates can be found within an edit distance of 6. This is because dictating dates requires successfully transcribing 3 tokens: day, month and year. We notice ASR either omitting or wrongly transcribing one of the 3 tokens, leading to an increase in user effort to correct dates. This is what motivated our scroller design for dates in our SQL Keyboard. In addition, we notice only 23% of the numbers were detected exactly as spoken. This is because ASR messes up when transcribing a number spoken with pauses. For example, “forty five thousand three hundred ten” is transcribed as “45000 310”. We notice from the plot that editing dates would require maximum units of effort (mean edit distance of 3.9), followed by strings (mean edit distance of 2.8) and finally numbers (mean edit distance of 2.2).

F. User Study

Preliminary User Study. In our first pilot user study, we recruited 15 participants without vetting them for their SQL knowledge. Thus, many participants had little experience composing SQL queries. Each participant composed 12 SQL queries on Employees database, where only English description of the query was given. We compared two conditions for specifying the query with a within-subjects design. In the first condition, the participant had access to an old SpeakQL interface that allowed them to dictate the SQL query and perform interactive correction using only a drag and drop based touch interface. In the second condition, the participant typed the SQL query from scratch with no access to our interface. We record the end-to-end time taken and evaluate our system using 144 data points (16 participants, 12 queries, some of the queries were not finished). We noticed a speedup of just 1.2x, when using SpeakQL interface in comparison with raw typing. Many participants spent a lot of time figuring out the correct query due to their poor knowledge of where and how to use SQL Keywords. As a result, the users dictated the entire query twice or thrice and then used a drag and drop based interface to correct the query.

Lessons Learned. This pilot user study helped us to learn several important lessons: (1) We did not vet the participants to ensure they are representative of our target userbase. Unlike NLIs, this version of SpeakQL is *not* aimed at lay users but rather data professionals that are already familiar with SQL and the schemas they likely query regularly. Thus, we need more vetting of participants. (2) We found that it was difficult for users to compose the entire query in head and dictate in one go. Research in cognitive science also tells us that the human working memory can retain a phrase or a context for maximum of only 10 seconds [16], [17]. Although SQL was designed for typing, users often think of the query at the clause level, since it has shorter contexts. Thus, supporting clause-level dictation could make the interface more speech-friendly. (3) Editing tokens in place required users to spend considerable amount of time in drag and drop effort. Hence, supporting SQL keyboard that allows users to quickly insert or delete any incorrectly placed or transcribed token using just a touch make the interface more correction friendly. In addition, the SQL keyboard would allow a user to correct a query (or sub-query) out of the order.

Learning from our previous experience, we revamped our interface to support: clause level dictation of queries, allowing insertion and deletion of tokens through a novel SQL keyboard and auto-completion of Literals, as mentioned in Section 5. We again conduct another user study with 15 participants where the recruitment was conducted through a short SQL quiz. Each participant was first made familiar with our interface through an introductory video [18]. Again, the participant composed 12 queries (say, from q_1 to q_{12}) to a browser-based interface on a hand-held tablet. Natural language description of the query along with the database schema was provided to the participant. We use the same two-condition within-subjects design as the first user study.

The queries were divided into two segments: *simple* and *complex*. We define *simple* queries as those with less than 20

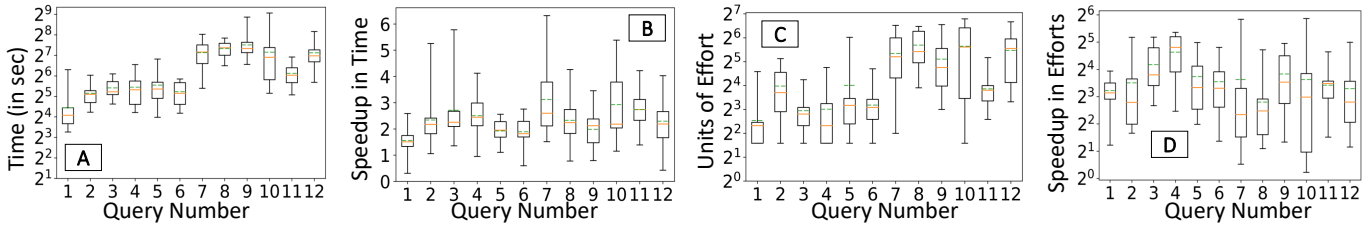

Fig. 10: *Simple* queries are marked from 1 to 6 and the rest are *complex*. (A) Time to completion for queries composed with SpeakQL (B) Speedup in time for queries composed with SpeakQL vs raw typing (C) Units of efforts for composed queries when using SpeakQL interface (D) Speedup in units of efforts for queries composed with SpeakQL vs raw typing.

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. We think 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).

When speaking the query, the SpeakQL interface will record the audio of the participant upon their pressing of "Record" button. The dictated query's transcription that is automatically corrected by SpeakQL will be displayed on the interface screen upon their press of the "Stop" button. If the query spoken is correct, then the response will be automatically logged, and the participant moves on to the next query. If not, then the participant will be instructed by the interface to correct the query. This process is continued until the participant gets the query right. We repeat this for the typing mode. We record the time to complete the query for both the conditions, i.e., when the participant is dictating and when the participant is typing. Also, we log every interaction of the user with our system, i.e., the number of corrections through touch/clicks and number of re-dictation attempts. We evaluate our system using 180 data points (15 participants, 12 queries).

Results. Figure 10 plots the time to completion, speedup in time to completion(i.e., time to completion of typing vs time to completion of SpeakQL), units of efforts and speedup in units of efforts for the 12 queries. The queries from 1-6 are *simple* queries, 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. From plots (B) and (D), we see that SpeakQL leads to significantly higher speedup in time to completion and units of effort than raw typing SQL queries. In addition, from plots (A) and (C), we notice that time to completion and units of effort for the *complex* queries is considerably higher than the *simple* ones. In terms of time to completion, the average speedup is 2.4x for the *simple* queries and 2.9x for the *complex* ones. On units of efforts, the average speedup is 12x for the *simple* queries and 7.5x for the *complex* ones. This is because SpeakQL is more likely to get a spoken *simple* query correct. Hence, it requires only minor editing efforts. We observe that for *simple* and *complex* queries, the fraction of time spent in dictation is an average of 43% and 32% respectively of total time, while that time spent on SQL Keyboard is an average of 17% and 47% respectively. The remaining time is spent idle either checking the query or looking at the schema. We present these results in our technical report [9].

Hypothesis Tests. We conducted 5 hypothesis tests with different measured quantities from the user study. The first 3 tests focus on the time to complete a query, the time spent editing a query, and the total units of efforts, with the null hypothesis being that these quantities in the SpeakQL condition are not significantly lower than than the typing condition. The last 2 tests check if the speedups offered by SpeakQL over typing on the *complex* queries are higher than the *simple* queries in terms of time to completion, and lower than the *simple* queries in terms of units of effort. Overall, we found that the null hypotheses were rejected at the $\alpha = 0.05$ level. The p-values were between 0 and 0.008. In order to control for multiple hypothesis tests, we also perform bonferroni correction by setting $\alpha = 0.01$. We again observe that all null hypotheses are rejected at the new α level.

VII. 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 [19]. Several systems such as Google's Search by Voice [20], [21] and Microsoft's Model M [22] 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 [1], [2], [23]. Tabular tools such as [2] allow users to query by example, wherein the user specifies results in terms of templates. [1] allows users to create drag-and-drop based interfaces. Keyword-search based interfaces such as [23] help users formulate SQL queries by giving query suggestions. More recently, non-keyboard based touch interfaces [3], [4], [24]–[26] have received attention

because of the potentially lower user effort to provide input. In particular, [4] allows user to query a database using a series of gestures, [3] is a pen-based human-in-the-loop interactive analytics system, and Tableau [26] offers touch-based visual analytics products. 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. Previous works [6], [10], [27], [28] have studied natural language interfaces for databases in order to allow layman users to ask questions in natural lanuage such as English. NLIs are orthogonal to this paper's focus. Seq2SQL exploits reinforcement learning to translate natural language questions to SQL queries [10]. However, the queries can be composed only over one table and with a maximum of only one aggregate. Inspired from regular human to human conversations, Echoquery [6] 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 [16], [17] on users when a large query result is returned; a screen mitigates such issues, e.g., as in the Echo Show. Moreover, a recent user study [29] on a text messaging app conducted by Baidu, showed that the input rate is significantly faster when users uses speech to perform only the first dictation, and then errors are corrected and refined through touch.

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 [10], [30]–[33]. The recent trend seen in the NLP community of incorporating structural knowledge into the modeling offers a form of validation for our approach of directly exploiting the rich structure of SQL using its grammar.

VIII. Conclusions and Future Work

We present the first end-to-end multimodal system for speech-driven querying of structured data using SQL. We find that raw ASR transcriptions are not usable directly because of myriad errors introduced by out-of-vocabulary tokens, homophones and other issues. This motivates our unique architectural design decision of separating the structure determination component from literal determination component. In order to empirically evaluate our system, we create the first dataset of spoken SQL queries. Furthermore, we present a scalable procedure to generate such data that applies to any arbitrary database schema. 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 speedup their SQL query specification process and as a result, saves a significant amount of their time and effort. As for future work, we would like to modify SQL to make it more amenable for spoken querying. From our empirical evaluation, we saw that Literals are the 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.

References

[1] "Oracle SQL Developer," Accessed December 18, 2018, blogs.oracle.com/smb/what-is-visual-builder-and-why-is-it-important-for-your-business.
[2] M. M. Zloof, "Query by Example," in *National Computer Conference and Exposition*, 1975.
[3] A. Crotty *et al.*, "Vizdom: Interactive analytics through pen and touch," *PVLDB*, 2015.
[4] A. Nandi *et al.*, "Gestural query specification," *PVLDB*, 2013.
[5] F. Li *et al.*, "Constructing an interactive natural language interface for relational databases," *PVLDB*, 2014.
[6] G. Lyons *et al.*, "Making the case for query-by-voice with echoquery," in *SIGMOD*, 2016.
[7] "Alexa commands," Accessed December 18, 2018, https://www.cnet.com/how-to/amazon-echo-the-complete-list-of-alexa-commands.
[8] "SQL grammar," Accessed December 18, 2018, http://forcedotcom.github.io/phoenix.
[9] V. Shah *et al.*, "SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data (Technical Report)," https://adalabucsd.github.io/speakql.html.
[10] V. Zhong *et al.*, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint*, 2017.
[11] S. B. Needleman *et al.*, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, 1970.
[12] "Mysql employees sample database," Accessed December 18, 2018, available from https://dev.mysql.com/doc/employee/en/.
[13] "Yelp database," Accessed December 18, 2018, available from https://www.kaggle.com/yelp-dataset/yelp-dataset.
[14] D. Chandarana, V. Shah, A. Kumar, and L. Saul, "Speakql: Towards speech-driven multi-modal querying," in *HILDA*. ACM, 2017.
[15] R. Ricci *et al.*, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications," *; login:: the magazine of USENIX & SAGE*, 2014.
[16] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psych. review*, 1956.
[17] T. Raman, "Audio system for technical readings," Cornell University, Tech. Rep., 1994.
[18] "SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data (Tutorial Video)," https://vimeo.com/295693078.
[19] "Nuance's dragon speech recognition," Accessed December 18, 2018, https://www.nuance.com/dragon.html.
[20] J. Schalkwyk *et al.*, "your word is my command: Google search by voice: a case study," in *Advances in speech recognition*, 2010.
[21] J. Shan *et al.*, "Search by voice in mandarin chinese," in *ISCA*, 2010.
[22] G. Zweig *et al.*, "Personalizing model for voice-search," in *ISCA*, 2011.
[23] J. Fan *et al.*, "Interactive sql query suggestion: Making databases user-friendly," in *ICDE*, 2011.
[24] A. Nandi, "Querying without keyboards." in *CIDR*, 2013.
[25] S. Idreos *et al.*, "dbtouch: Analytics at your fingertips." in *CIDR*, 2013.
[26] P. Terlecki *et al.*, "On improving user response times in tableau," in *SIGMOD*, 2015.
[27] D. Saha *et al.*, "Athena: an ontology-driven system for natural language querying over relational data stores," *PVLDB*, 2016.
[28] A.-M. Popescu *et al.*, "Towards a theory of natural language interfaces to databases," in *IUI*, 2003.
[29] S. Ruan *et al.*, "Speech Is 3x Faster than Typing for English and Mandarin Text Entry on Mobile Devices," *CoRR*, 2016.
[30] C. Dyer *et al.*, "Recurrent neural network grammars," *CoRR*, 2016.
[31] T. Cohn *et al.*, "Incorporating structural alignment biases into an attentional neural translation model," *arXiv preprint*, 2016.
[32] D. Marcheggiani *et al.*, "Encoding sentences with graph convolutional networks for semantic role labeling," *arXiv preprint*, 2017.
[33] Y. Kim *et al.*, "Structured attention networks," *arXiv preprint*, 2017.

APPENDIX

A. SQL Grammar

The production rules of the supported SQL grammar are shown in Box 1.

Box 1: SQL Grammar Production Rules

```
Q → S F | S F W
S → SEL LST | SEL L C | SEL SEL_OP BP L EP | SEL SEL_OP BP L EP C | SEL CNT BP ST EP | SEL CNT BP ST EP C
C → COM L | C COM L | COM SEL_OP BP L EP | C COM SEL_OP BP L EP
CF → COM L | CF COM L
F → FRO L | FRO L CF
W → WHE WD | WHE AGG
WD → EXP | EXP AN WD | EXP OR WD
EXP → L OP L | WDD OP L | WDD OP WDD | L OP WDD
WDD → L DO L
AGG → WD CLS L | WD CLS WDD | WD LMT L | L BTW L AN L | L NT BTW L AN L | L IN BP L EP | L IN BP L CS EP
CS → COM L | CS COM L
CLS → ODB1 ODB2 | GRP1 ODB2
LST → L | ST
COM → ','
SEL → 'SELECT'
FRO → 'FROM'
ST → '*'
L → 'x'
OP → '=' | '<' | >'
AN → 'AND'
OR → 'OR'
NT → 'NOT'
BTW → 'BETWEEN'
WHE → 'WHERE'
DO → '.'
ODB1 → 'ORDER'
ODB2 → 'BY'
GRP1 → 'GROUP'
LMT → 'LIMIT'
SEL_OP → 'AVG' | 'SUM' | 'MAX' | 'MIN' | 'COUNT'
CNT → 'COUNT'
BP → '('
EP → ')'
IN → 'IN'
```

B. Structure Determination

1) Overall Algorithm: The central idea of this algorithm is to skip searches on tries that were pruned out by our bidirectional bounds in Proposition 1. For the tries that are not pruned, we recursively traverse every children of the root node using `SearchRecursively` procedure. At every node, we use the dynamic programming algorithm (Box 2) to calculate edit distance with `TransOut`, and build a column of the memo as shown in Figure 3(B). 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 (`node.sentence`). This algorithm does not affect accuracy, i.e., it returns the same string as searching over all the tries.

2) Accuracy-Latency Tradeoff Techniques: We propose two additional algorithms that uniquely exploit the way SQL strings are stored in the tries. This helps us to reduce runtime further by trading off some accuracy.

Diversity-Aware Pruning (DAP): We observe that many paths from root to leaf in a trie differ in only one token that is either from the keyword set {`AVG`, `COUNT`, `SUM`, `MAX`, `MIN`}, {`AND`, `OR`} or the SplChar set $\{=, <, >\}$. We call the union of 3 sets, a *prime superset*. Instead of exploring all the branches, we can instead skip many branches that differ only in one token from the prime superset. Based on this observation, we propose the following technique. Given a trie T and transcription output `TransOut`, if a node *n* has p children nodes ($C_i, 1 \leq i \leq p$) belonging to any set in the prime superset, and C_k gives the minimum edit distance with `TransOut`, while the other siblings C_i, $i \neq k$ does not give minimum edit distance, then skip exploration of all the descendants of C_i, $i \neq k$. i.e., for every children C_i of a node in the prime superset, the node to explore is $\texttt{argmin}_i(\texttt{DpCurCol}_{C_i}(\texttt{lastrow}))$

However, this optimization can skip the branch leading to the minimum edit distance, resulting in a decrease in accuracy. This optimization is introduced to have more diversity in the returned top k structures. Rather than

Box 2: Structure Determination Algorithm

```
Let k = Max Tokens possible in GrndTrth (50)
LowerBound = Array of size k; MinEditDist = INT_MAX
m = CountTokens(TransOut); result[MinEditDist] = ""
for i from 1 to k do
    LowerBound[i] = |m-i|*W_L
end for
for j from m to 0 do
    if MinEditDist < LowerBound[j] then j--
    else SearchTrie(j)
    end if
end for
for j from m to k do
    if MinEditDist < LowerBound[j] then j++
    else SearchTrie(j)
    end if
end for
return result[MinEditDist]

procedure SearchTrie(j)
    TrieRoot = RetrieveStrings(j):
    DpPrvCol = [1,2,...,m]
    for token in TrieRoot.children do
        SearchRecursively(TrieRoot.children[token],token,DpPrvCol)
    end for
end procedure

procedure SearchRecursively(node,token,DpPrvCol):
    rows = CountTokens(MaskOut) + 1
    DpCurCol = [DpPrvCol[0]+1]
    for row from 1 to rows do
        if MaskOut[row-1] == token then
            DpCurCol.append(DpPrvCol[row-1])
        else
            if DpPrvCol[row] < DpCurCol[row-1] then
                insertCost = DpPrvCol[row] + W_token
                DpCurCol.append(insertCost)
            else
                deleteCost = DpCurCol[row-1] + W_token
                DpCurCol.append(deleteCost)
            end if
        end if
        if node is leaf and DpCurCol[rows] < MinEditDist then
            MinEditDist = DpCurCol[rows]
            result[MinEditDist] = node.sentence
        end if
        if min(DpCurCol) ≤ MinEditDist then
            for token in node.children do
             SearchRecursively(node.children[token],token,DpCurCol)
            end for
        end if
    end for
end procedure
```

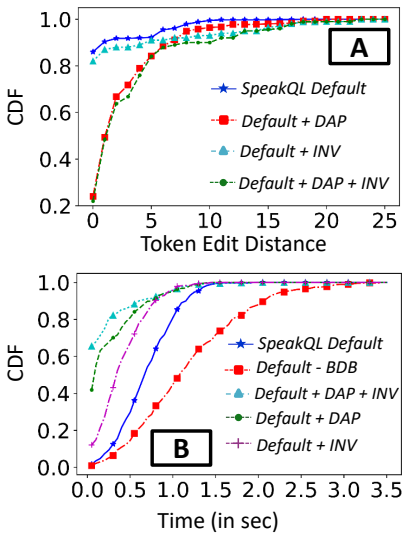

Fig. 11: Ablation study of Structure Determination Component

overloading the system to find the most correct structure, the system can find an approximately correct structure which misses only in certain keywords like {AND, OR} or {AVG, COUNT, SUM, MAX, MIN} or SplChars like {=, <, >}. Doing this would require more amount of user effort in order to correct the query. However, if the user effort is just 1-5 touches/clicks, then this is justifiable.

Inverted Indexes (INV): We can utilize the knowledge about the different keywords occurring in TransOut in order to build an inverted index of all the unique keywords (except SELECT, FROM, WHERE) appearing in the ground truth strings. Hence, for each keyword, we directly retrieve a list of strings in which it appears. When multiple keywords exist in the TransOut, we select the index that has the minimum number of strings corresponding to it. This can reduce the computation time, as we only retrieve a fraction of relevant strings. Although, this optimization leads to runtime efficiency, it heavily relies on the fact that ASR engine is very unlikely to misrecognize SQL Keywords. As any ASR engine cannot be perfect, we anticipate a drop in accuracy.

3) Proofs:

Algorithm 2 Time and Space Complexity Analysis. Let n be the length of TransOut, p be the number of nodes in the largest trie, and k be the number of tries (value fixed to 50). In the worst case, the algorithm would traverse each node of every trie and would compute DpCurCol for each and every node. Hence, the worst-case time complexity of the algorithm is bounded by $O(pkn)$. The maximum possible space required by the algorithm is equivalent to number of tries times number of nodes in the largest trie. Hence, space complexity is $O(pk)$.

Box 3: The Literal Determination Algorithm

```
procedure LiteralFinder(TransOut,BestStruct):
    RunningIndex = 0; FilledOut = BestStruct
    for every placeholder x_j in BestStruct do
        while TransOut(RunningIndex) ∈ (KeywordDict or SplCharDict) do
            RunningIndex++
        end while
        BeginIndex(x_j) = RunningIndex
        EndIndex(x_j) = RightmostNonLiteral(RightNonLiteral(x_j))
        A, positions = EnumerateStrings(BeginIndex(x_j), EndIndex(x_j)))
        B = RetrieveCategory(x_j)
        literal, k = LiteralAssignment(A,B,positions)
        FilledOut(x_j) = literal
        RunningIndex = k+1
    end for
    return FilledOut
end procedure

procedure EnumerateStrings(BeginIndex, EndIndex):
    results = {};positions = {}; i = 0
    while i ≠ EndIndex do
        j = i; k=0; curstr = ""
        while TransOut(j) ∉ (KeywordDict or SplCharDict) AND (j < EndIndex) AND (k < WindowSize) do
            curstr = curstr + TransOut(j)
            results.append(PhoneticRep(curstr))
            positions.append(j)
            j++;k++
        end while
        i++
    end while
    return results, positions
end procedure

procedure LiteralAssignment(A, B, positions):
    for every item b in B do
        Initialize count(b) = 0; location(b) = -1
    end for
    for every item a in A do
        set(a) = φ; minEditDist = ∞
        for every item b in B do
            if EditDist(a,b) < minEditDist then
                set(a) = φ; set(a).add(b)
                minEditDist = EditDist(a,b)
            else if EditDist(a,b) == minEditDist then
                set(a).add(b)
            end if
        end for
        for every item b in set(a) do
            count(b)++
            location(b) = max(location(b),positions(a))
        end for
    end for
    literal = argmax_{b∈B} (count(b))
    k = location(b)
    return literal, k
end procedure
```

4) Ablation Study: In this analysis, we would like to understand how effective these different optimization techniques are in reducing latency and how much they affect accuracy, for the structure determination component. Figure 11 (A) shows the CDF of TED. Note that the BDB is an accuracy preserving optimization. When we consider all the optimizations (SpeakQL Default + DAP + INV), we notice a significant amount of drop in accuracy. Number of queries having TED of 0 drops from 86% to 21%. Also, the TED to deliver 99% of the correct structures increases to 23 (from 10, which was observed with SpeakQL Default). This is expected because DAP does not explore all the branches containing special characters such as {=, <, >} and keywords such as {AVG, SUM, MAX, MIN, COUNT} and {AND, OR}. On the

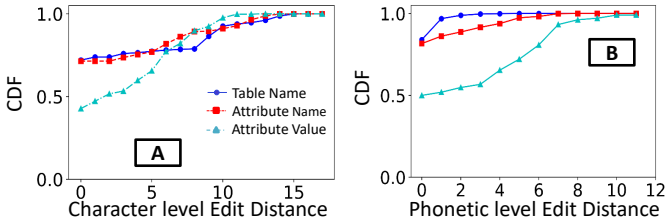

Fig. 12: Evaluation of Character level Edit distance vs Phonetic level Edit Distance

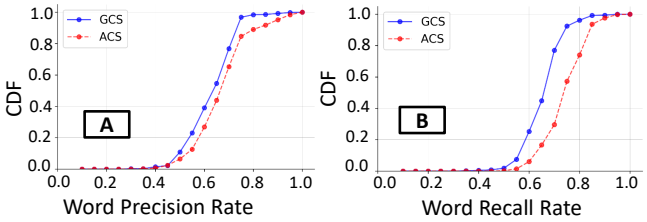

Fig. 13: Cumulative Distribution of accuracy metrics for top 1 results

	KPR	SPR	LPR	KRR	SRR	LRR
GCS	0.78	0.94	0.39	0.85	0.97	0.4
ACS	0.84	0.87	0.49	0.92	0.96	0.53

Fig. 14: Mean error metrics: Precision and Recall

other hand, INV leads to only a minor drop in accuracy. This is because the ASR is good enough not doing a mistake of recognizing a literal as a keyword or a special character. If that happens, then INV is expected to mess up and lead to a larger drop in accuracy. 11 (B) plots the CDF of running time. When using only the prefix tries with BDB turned off, we notice that an increase in running time by almost 2x. Thus, BDB helps in saving the running time by a factor of 2. DAP leads to runtime gains of rougly 3.5x (compared to SpeakQL Default) with almost 40% of the queries finishing under 0.1 seconds. While, with INV runtime gain is of roughly 1.7x. Thus, DAP and INV can be used to further reduce the runtime, but would also lead to some amount of drop in accuracy.

C. Literal Determination

1) Overall Algorithm: The Literal Determination component has to be made aware of the splitting of tokens into sub-tokens, so that it can decide when and how to merge these sub-tokens. Figure 4 shows the workflow of the Literal Determination component with `TableNames = {Employees (EMPLYS), Salaries (SLRS)}` and `AttributeNames = {FirstName (FRSTNM), LastName (LSTNM)}`. The inputs given to the Literal Determination component are `TransOut` and best structure (`BestStruct`) obtained from the Structure Determination component. 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 variable. 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 representation of the two sets A and B. Box 3 describes this in depth.

2) Evaluation of Phonetic Edit distance: We evaluate how much a similarity search on a pre-computed phonetic representation of existing Literals in the database help relative to a string-based similarity search. That is, we intend to compare character-level edit distance on phonetic representation with character-level edit distance on the string. Phonetic representation helps us to provide a more condensed representation of a literal. From Figure 12, we see that it requires less phonetic distance compared to the character level edit distance in order to obtain the correct token. For example, the correct literal exists within a character-level edit distance of 17 from the transcribed literal. But if we rely on the phonetic character-level edit distance, the correct Literals can be found within edit distance of only 11. In addition, we see that, almost 70% of the table names and attribute names have edit distance of 0, while the character-level edit distance on the phonetic representation is 0 for almost 80% of them. Thus, phonetic representation can help in retrieving the extra 10% of the table names and attribute names that were not found when doing the edit distance on the full word representation. Overall, we find phonetic representation helps in achieving higher accuracy.

D. ASR

We compare Google's Cloud Speech Service (GCS) vs Azure's Custom Speech Service (ACS) on 500 test queries belonging to the Employees database. Due to space constraints, we only plot the CDF of word precision and recall rates in the Figure 13. We notice an improvement in word precision rate from mean of 0.62 for GCS to 0.67 for ACS and an improvement in word recall rate from mean of 0.65 for GCS to 0.73. For Google's Cloud Speech Service as shown in Table 14, we noticed that the precision and recall rates for Keywords and SplChars are high. This is because GCS allows them to be provided as hints to the API. Hints are tokens that might be present in the audio; they help the ASR engine pick between alternate transcriptions. For example, if "=" is given as a hint, we might get the "=" symbol instead of "equals" as text. Despite this, Azure's Custom Speech Service fare significantly well in recognizing Keywords and also Literals .

E. User Study

We drill down deeper to see how a user is interacting with SpeakQL. Figure 15 shows the % time of the total end-to-

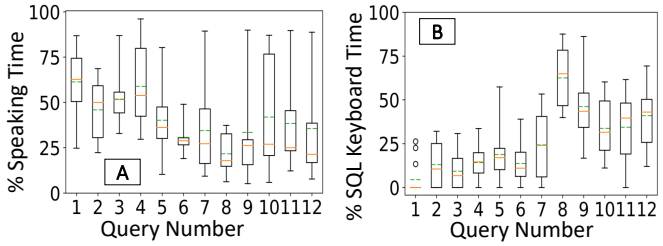

Fig. 15: *Simple* queries are marked from 1 to 6 and the rest are *complex*. (A) Fraction of time spent in dictating the query relative to the total end-to-end time (B) Fraction of time spent in using the SQL keyboard relative to the total end-to-end time.

end time that went into Speaking out the query (plot A) and using the SQL Keyboard (plot B). We notice that for the simple queries, SpeakQL is able to get most of dictated queries correct. Hence, a user spends most of their time in just dictating the query by either speaking out the entire query or speaking out at the clause level. Thus time spent in performing corrections using SQL keyboard is negligible or almost none. While for the complex queries, the trend is exactly the opposite. Users prefer to use the SQL Keyboard than speech when composing complex queries.