# 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.

## 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 [23, 56], touch-based [30, 42], typed natural language interfaces (NLIs) [37], and even bidirectional conversations [38]. 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 Appendix.

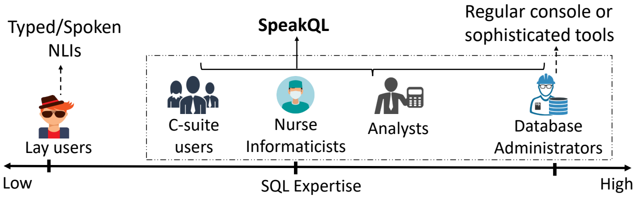
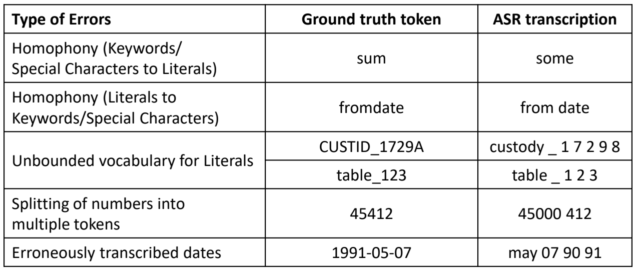


Figure 1: Contrasting SpeakQL's goals with current NLIs and other sophisticated tools in terms of SQL expertise of users.

**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 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 Appendix.

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



| Type of Errors | Ground truth token | ASR transcription |
|---|---|---|
| Homophony (Keywords/<br>Special Characters to Literals) | sum | some |
| Homophony (Literals to<br>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<br>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).

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[18, 37], 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.

**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 [55] and Spider [54]. 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

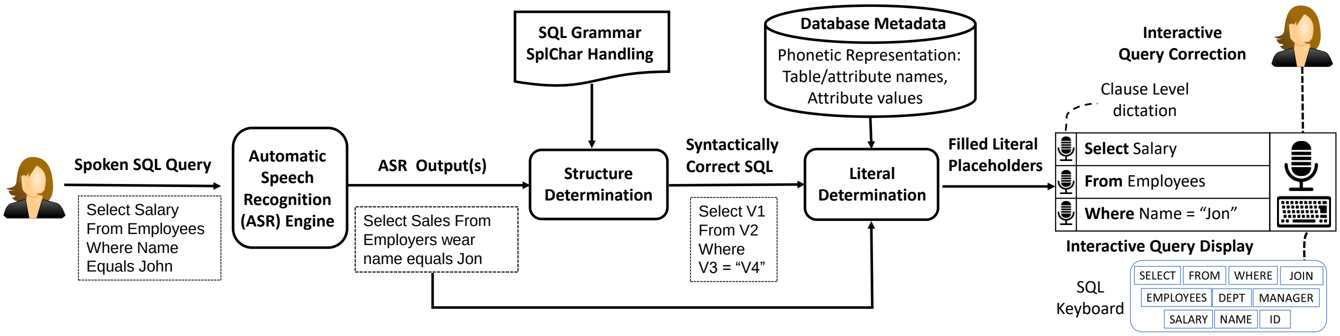


Figure 2: End-to-end Architecture of SpeakQL[28]. We show an example of a simple spoken SQL query, and how it gets converted to a query displayed on a screen, which the user can correct interactively.

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 [24]. 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 [28].

**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 10 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

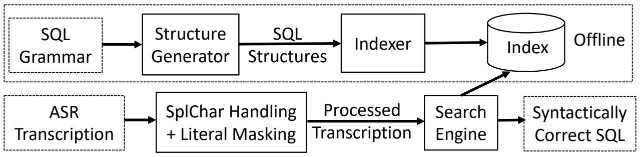


Figure 3: Structure Determination component's architecture.

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 DDL. We use the production rules of SELECT statements of standard SQL in Backus-Naur Form [24]. 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 [34, 55] allows queries over only one table and with only one aggregate. In addition, the task on Spider dataset does not involve generating condition values [54]. We provide the full grammar in the Appendix.

## 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 [43], 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.

**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

**Algorithmus 1** Dynamic Programming Algorithm

```
if token in KeywordDict then $W_{token}$ = $W_K$
else if token in SplCharDict then $W_{token}$ = $W_S$
else $W_{token}$ = $W_L$
dp(i,0) = i for 0 $\leq$ i $\leq$ n; dp(0,j) = j for 0 $\leq$ j $\leq$ m
if a(i) == b(j) then dp(i,j) = dp(i-1,j-1) = DpPrvCol(row-1)
else dp(i,j) = min($W_{token}$+dp(i-1,j), $W_{token}$+dp(i,j-1))
DpPrvCol(row) = dp(i,j-1)
DpCurCol(row-1) = dp(i-1,j)
insertCost = DpPrvCol(row) + $W_{token}$
deleteCost = DpCurCol(row-1) + $W_{token}$
```

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 Appendix.

**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 the appendix. We also study two additional accuracy-latency tradeoff algorithms that further reduce runtime by trading off some accuracy, which can be found in the appendix. 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 pre-computed *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.

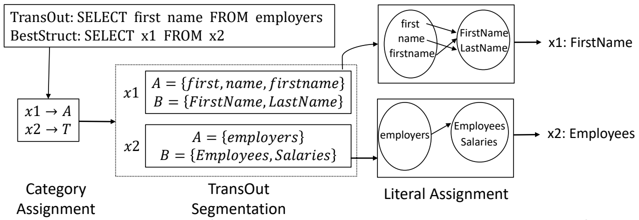


Figure 4: Literal Determination component example

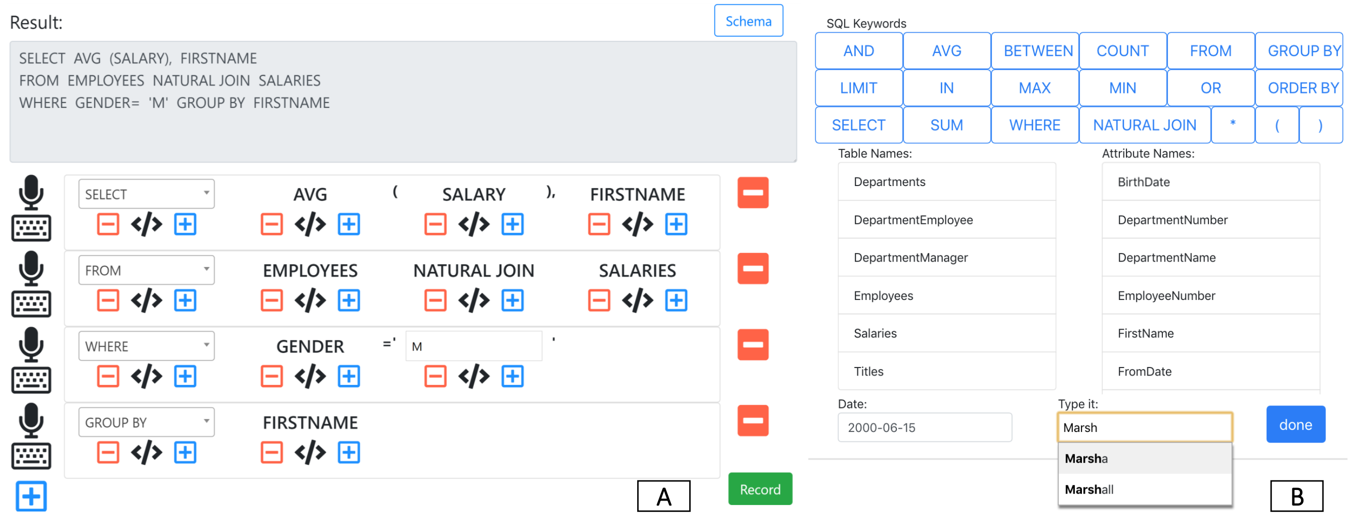
**2. Handling out-of-vocabulary tokens.** Literal Determination has to be made aware of the splitting of tokens (out-of-vocabulary from ASR's perspective) into sub-tokens so that it can decide when and how to merge them. Figure 4 shows the workflow of this component with `TableNames`= {`Employees (EMPLYS)`,`Salaries (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 the appendix. 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 $O(n^2 + m)$.

## 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


Result:

```
SELECT  AVG  (SALARY),  FIRSTNAME
FROM  EMPLOYEES  NATURAL JOIN  SALARIES
WHERE  GENDER=  'M'  GROUP BY  FIRSTNAME
```

Figure 5: SpeakQL Interface[48]. (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.

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`.

# 5. INTERFACE

Figure 5(A) shows our interface. We demonstrated our interface at [48]. 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 [54] and WikiSQL [55] 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 [27, 26]. While for spoken

| 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.

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 the appendix.

**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 [21] and the Yelp Dataset [25]. We get the `table names`, `attribute names`, and `attribute values` in each database.
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 [6].

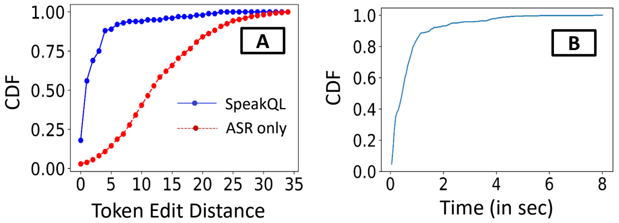


Figure 6: (A) Evaluation of SpeakQL on Token Edit Distance (B) Runtime of SpeakQL.

## 6.2 Metrics

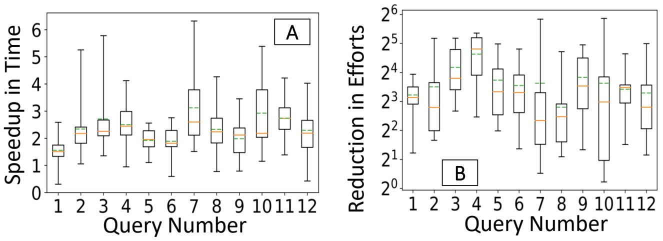
For evaluating accuracy, we first tokenize a query text to obtain a multiset of tokens (Keywords, SplChars, and Literals). We then compare the multiset A of the reference query (ground truth SQL query) with the multiset B of the hypothesis query (transcription output from SpeakQL). We use the following 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.

## 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 [45].

**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 appendix. 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



| 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.

TED of less 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 the appendix.

**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 [7]. 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).

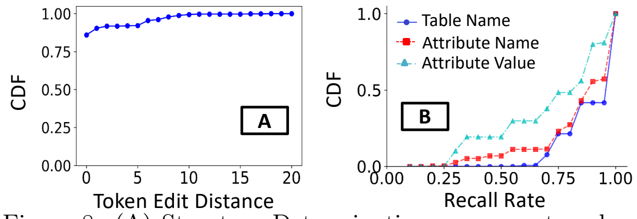


Figure 8: (A) Structure Determination component evaluation on Token Edit Distance (TED) (B) Literal Determination component evaluation. CDF of Recall Rates for different Literal types.

**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 the appendix.

## 6.5 Component-level Drill Down

**Structure Determination Evaluation.** We evaluate the structures returned by this component relative to the ground truth structure. Figure 14(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 appendix.

**Literal Determination Evaluation.** Figure 14(B) presents the CDF of recall rates for table names, attribute names, and 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 appendix.

## 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 [54] and WikiSQL [55]. 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 appendix.

# 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 [22]. Several systems such as Google's Search by Voice [47, 49] and Microsoft's Model M [57] 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 [56, 23, 32]. Tabular tools such as [56] allow users to query by example, [23] allows users to create drag-and-drop based interfaces, and keyword-search based interfaces such as [32] help users formulate SQL queries by giving query suggestions. More recently, non-keyboard based touch interfaces [41, 42, 30, 35, 51] 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 [46, 50, 52, 53, 54, 33, 34, 55, 38]. NLIs are orthogonal to this paper's focus. Inspired from regular human to human conversations, Echoquery [38] 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 [40, 44] 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 [55, 31, 29, 39, 36]. 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.

# 9. REFERENCES

[1] Blended Learning for SQL. https://link.springer.com/chapter/10.1007%2F978-3-319-59360-9_30.

[2] Oracle Business Intelligence Cloud Service Mobile. https://docs.oracle.com/en/cloud/paas/bi-cloud/bilug/viewing-content-mobile-devices.html.

[3] Oracle Business Intelligence Mobile. http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/bi-mobile-1993624.pdf.

[4] Oracle Sales Cloud Mobile. http://docs.media.bitpipe.com/io_11x/io_116852/item_1033327/Oracle%20Sales%20Cloud_%20Smartphones_Tablets_Datasheet.pdf.

[5] Power BI Mobile. https://www.clearpeaks.com/mobile-bi-using-power-bi/.

[6] Speakql project webpage. https://adalabucsd.github.io/speakql.

[7] SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data (Tutorial Video). https://youtu.be/KsgNo-CkE8Y.

[8] SQL Server Mobile Reporting. https://docs.microsoft.com/en-us/sql/reporting-services/mobile-reports/create-mobile-reports-with-sql-server-mobile-report-publisher?view=sql-server-ver15.

[9] SQL Server Mobile Reporting Tutorial. https://www.mssqltips.com/sqlservertutorial/4100/ssrs-mobile-reports-tutorial-outline/.

[10] SQL Server Voice Dictation. `https://www.reddit.com/r/SQLServer/comments/7nogtn/best_voice_dictation_software_for_sql_server/`.

[11] Tableau Forum asking for Alexa Integration. `https://community.tableau.com/thread/226682`.

[12] Tableau Forum asking for Alexa Integration. `https://community.tableau.com/thread/255574`.

[13] Tableau Forum asking for voice integration. `https://community.tableau.com/thread/270545`.

[14] Tableau Forum Voice Commands. `https://community.tableau.com/thread/276723`.

[15] Tableau Forum Voice Commands. `https://community.tableau.com/message/786209#786209`.

[16] Tableau Voice-based Dashboard. `https://medium.com/@bhavasarhemang/tableau-with-voice-commands-9d9d9664853f`.

[17] Talking to Tableau Dashboard. `https://www.datablick.com/blog/2017/12/29/talking-to-tableau-via-react`.

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

[19] Amazon polly speech synthesis api, Accessed April 12, 2020. Available from `https://aws.amazon.com/polly`.

[20] Custom speech service, Accessed April 12, 2020. `https://azure.microsoft.com/en-us/services/cognitive-services/custom-speech-service/`.

[21] Mysql employees sample database, Accessed April 12, 2020. Available from `https://dev.mysql.com/doc/employee/en/`.

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

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

[24] SQL grammar, Accessed April 12, 2020. `http://forcedotcom.github.io/phoenix`.

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

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

[27] F. Basik, B. Hättasch, A. Ilkhechi, A. Usta, S. Ramaswamy, P. Utama, N. Weir, C. Binnig, and U. Cetintemel. DBPal: A Learned NL-Interface for Databases. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1765–1768, New York, NY, USA, 2018. ACM.

[28] D. Chandarana, V. Shah, A. Kumar, and L. Saul. SpeakQL: Towards Speech-driven Multi-modal Querying. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, pages 1–6, 2017.

[29] T. Cohn, C. D. V. Hoang, E. Vymolova, K. Yao, C. Dyer, and G. Haffari. Incorporating Structural Alignment Biases into an Attentional Neural Translation Model. *arXiv preprint arXiv:1601.01085*, 2016.

[30] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive Analytics Through Pen and Touch. *Proceedings of VLDB Endowment*, 8(12):2024–2027, 2015.

[31] C. Dyer, A. Kuncoro, M. Ballesteros, and N. A. Smith. Recurrent Neural Network Grammars. *arXiv preprint arXiv:1602.07776*, 2016.

[32] J. Fan, G. Li, and L. Zhou. Interactive SQL Query Suggestion: Making Databases User-friendly. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 351–362. IEEE, 2011.

[33] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. *arXiv preprint arXiv:1905.08205*, 2019.

[34] W. Hwang, J. Yim, S. Park, and M. Seo. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. *arXiv preprint arXiv:1902.01069*, 2019.

[35] S. Idreos and E. Liarou. 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, 2013.

[36] Y. Kim, C. Denton, L. Hoang, and A. M. Rush. Structured Attention Networks. *arXiv preprint arXiv:1702.00887*, 2017.

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

[38] G. Lyons, V. Tran, C. Binnig, U. Cetintemel, and T. Kraska. Making the Case for Query-by-Voice with EchoQuery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2129–2132. ACM, 2016.

[39] D. Marcheggiani and I. Titov. 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*, pages 1506–1515. Association for Computational Linguistics, 2017.

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

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

[42] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *Proceedings of the VLDB Endowment*, 7(4):289–300, 2013.

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

[44] T. V. Raman. *Audio System for Technical Readings*, volume 1410 of *Lecture Notes in Computer Science.* Springer, 1998.

[45] R. Ricci, E. Eide, and C. Team. Introducing CloudLab: Scientific Infrastructure for Advancing

Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[46] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. ATHENA: an Ontology-driven System for Natural Language Querying over Relational Data Stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016.

[47] J. Schalkwyk, D. Beeferman, F. Beaufays, B. Byrne, C. Chelba, M. Cohen, M. Kamvar, and B. Strope. "Your word is my command": Google Search by Voice: A Case Study. In *Advances in speech recognition*, pages 61–90. Springer, 2010.

[48] V. Shah, S. Li, K. Yang, A. Kumar, and L. Saul. Demonstration of SpeakQL: Speech-driven Multimodal Querying of Structured Data. In *Proceedings of the 2019 International Conference on Management of Data*, pages 2001–2004, 2019.

[49] J. Shan, G. Wu, Z. Hu, X. Tang, M. Jansche, and P. J. Moreno. Search by Voice in Mandarin Chinese. In *Eleventh Annual Conference of the International Speech Communication Association*, pages 354–357, 2010.

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

[51] P. Terlecki, F. Xu, M. Shaw, V. Kim, and R. Wesley. On Improving User Response Times in Tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1695–1706. ACM, 2015.

[52] X. Xu, C. Liu, and D. Song. SqlNet: Generating Structured Queries from Natural Language without Reinforcement Learning. *arXiv preprint arXiv:1711.04436*, 2017.

[53] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. SQLizer: Query Synthesis from Natural Language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63:1–63:26, 2017.

[54] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev. 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*, pages 3911–3921, 2018.

[55] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103*, 2017.

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

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

# APPENDIX

## A. INTERVIEW STUDY

To better understand the querying practices of data professionals, we conducted an interview study with 26 SQL users from 17 different sectors. We performed this study in two rounds. In the first round, we interviewed SQL users to get information about the style of queries they compose, the kinds of challenges they encounter, and if they are interested in using a speech-driven multimodal interface. In the second round, we contacted the interested users and gave them access to our system to get feedback on the utility of our system.

**Part A.** We first interviewed 26 SQL users from 17 different sectors such as finance, healthcare, insurance, retail, social housing, telecom, etc. We recruited interviewees by emailing working professionals in the business school and medical school of UC San Diego, and by contacting individuals on LinkedIn. The interviewees had many different job titles such as Data Analyst, Business Intelligence Analyst, Risk Analyst, Executive, Management Information Analyst, Data Architect, etc. We asked them many questions to understand the following:

- *What all kinds of queries do they compose?*
- *What all kinds of query complexity do they have to deal with?*
- *What sort of environment do they use to retrieve/analyze their data?*
- *What kinds of challenges do they encounter when composing an SQL query on their environment?*
- *Do they see a speech-driven multimodal interface enhancing their data querying abilities?*

The list of all questions along with their responses is presented in Appendix H. We summarize the takeaways below.

1. Almost all the users are well-versed with SQL and they compose SQL queries routinely. 21 out of 26 users rated their SQL expertise level at 4 or 5 out of 5. 17 of them actually compose SQL queries a lot almost everyday and 25 of them compose queries at least a few times a week.

2. Most users (24 out of 26) compose ad hoc queries over arbitrary tables instead of following fixed template of queries. Thus, from our interactions with the users, we realize that dashboards that supports fixed query templates are not enough for them.

3. While all users perform data retrieval and analysis queries, only a handful of them (5 out of 26) compose complex queries such as the ones from TPC-H benchmark. We find that 3 of these 5 users are non-DevOps DBAs, which are anyways not the primary target audience for our system.

4. We find that non-trivial fraction of users (11 out of 26) typically query over 1 to 3 tables and they encounter up to 50 tokens in their query. This makes querying on constrained environment ideal for many scenarios. We do not claim to support all possible type of querying scenarios.

5. For two-third of the users, a typical query either involves a join over multiple tables or aggregation over

multiple joined table. Only fifth of the users typically perform complex nested queries. We are happy to see that our currently supported subset of SQL can already satisfy the common cases of many users.

6. Overall, a non-trivial fraction of users (11 out of 26) was really interested in trying out interface. We admit that there is heavy polarization on the responses received. For instance, non-DevOps DBAs find tablet/phone environments too small and are not inclined towards using a speech-based system. As we explained in Figure 1 of the paper, such ultra-sophisticated SQL users are not the primary target of our system.

**Part B.** In the second round, we followed up with the interested users, gave them access to our system, and asked them to use our system. We asked them to fill-up another google form questionnaire to receive feedback of our system. The list of all questions along with their responses is presented in Appendix I. We summarize the takeaways below.

1. We asked users to rate their difficulty with typing SQL queries vs using the SpeakQL interface on a scale from 1 to 10, where 1 being "most difficult/painful" and 10 being "very easy and seamless" on the device. **We find that the median rating of the 7 users who used tablet/smartphone device improved from 4 (score when typing) to 8 (score with SpeakQL).** We consider this as a very significant jump.

2. Overall, we only see 3 cases where the rating score with SpeakQL is less than the score with typing. However, 2 of these users were querying on desktop/laptop device to interface (contrary to our stated request) where typing is more convenient (recall that deskptop/laptop are not our target device environments). For the 1 other user, the drop was from 8 to 7, which is minor.

3. Almost all the users found the ability to speak out the query useful. We quote some of the use case scenarios where users foresee our speech-driven multimodal interface on smartphone/tablet enhancing their querying abilities below:

   *"It would make pulling ad-hoc data during meetings simpler"*

   *"Don't use tablets/smartphones to query data mostly due to the pain of writing the sentences. Usually query process related data from PPM DBs"*

   *"When I am outside of my usual working environment and want to see data."*

   *"Training in front of a class"*

   *"90% of the time I can tell what the issue is (and make decisions) from my phone. I have my laptop with me for the other 10%. it would be very useful to run some queries based on different alerts I get throughout the day."*

Overall, from the interview study we find that there is a non-trivial user base who is really interested in exploring such speech-driven interfaces.

**Part C.** We went through websites of existing industrial tools like Oracle BI, Tableau, SQL Server, and Power BI and we find that there is recurring interest in industry in the following two directions:

Box 1: SQL Grammar Production Rules

```
 1: Q → S F | S F W
 2: 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
 3: C → COM L | C COM L | COM SEL_OP BP L EP | C COM SEL_OP BP
    L EP
 4: CF → COM L | CF COM L
 5: F → FRO L | FRO L CF
 6: W → WHE WD | WHE AGG
 7: WD → EXP | EXP AN WD | EXP OR WD
 8: EXP → L OP L | WDD OP L | WDD OP WDD | L OP WDD
 9: WDD → L DO L
10: 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
11: CS → COM L | CS COM L
12: CLS → ODB1 ODB2 | GRP1 ODB2
13: LST → L | ST
14: COM → ‘,’
15: SEL → ‘SELECT’
16: FRO → ‘FROM’
17: ST → ‘*’
18: L → ‘x’
19: OP → ‘=’ | ‘<’ | >’
20: AN → ‘AND’
21: OR → ‘OR’
22: NT → ‘NOT’
23: BTW → ‘BETWEEN’
24: WHE → ‘WHERE’
25: DO → ‘.’
26: ODB1 → ‘ORDER’
27: ODB2 → ‘BY’
28: GRP1 → ‘GROUP’
29: LMT → ‘LIMIT’
30: SEL_OP → ‘AVG’ | ‘SUM’ | ‘MAX’ | ‘MIN’ | ‘COUNT’
31: CNT → ‘COUNT’
32: BP → ‘(’
33: EP → ‘)’
34: IN → ‘IN’
```

1. Creating apps on smartphones/tablets to allow users to interact with their data on-the-go with Oracle BI [2, 4, 3], PowerBI [5], and SQL Server [8, 9]. In addition, in the blended learning literature, it has been found that allowing students to learn SQL right from their phones/tablets improves their SQL learning ability significantly [1].

2. Request for features to query with voice-based commands on Tableau [14, 15, 16, 17, 13], SQL Server [10], and even request for integration with Alexa as a Tableau assistant [11, 12]. Although these tools are orthogonal to our focus, we think that the interest of industry in designing on-the-go speech-driven tools that are SQL-aware is encouraging for an exploratory research project like ours.

# B. COMPARISON AGAINST SPOKEN NLI

Table 3 shows a qualitative comparison of a spoken SQL system against a spoken NLI system on different issues that we identified based on our conversations. Although a spoken SQL system expects knowledge of SQL and database schema being queried, it offers four major advantages that many data professionals find useful. (1) SQL is unambiguous due to its CFG. In contrast, NLIs can fail to understand query intent because the ambiguities introduced by natural language are hard to fix. (2) With SQL, users always know that the results will match their query. However, NLIs may find it hard to offer such firm guarantees on result correctness. (3) NLIs often rely on lexical databases or word embeddings to map tokens in the posed query with schema literals. Thus, for non-dictionary tokens (e.g., $CUSTID\_1729A$), NLIs may fail to obtain a correct mapping. In contrast, SQL's syntactic and semantic properties makes non-dictionary tokens easier to support. (4) ASR can introduce errors in the transcription and can cause the errors to propagate further in NLIs. For instance, even a simple wrong transcription of the token "and" to "in" can change the meaning of the query completely and can cause the NLI to fail. A spoken SQL querying system can leverage the rich structure of SQL and the database schema to potentially correct such errors in transcription and thus can deliver much higher accuracy, as confirmed by our initial comparison (Appendix F.9).

| | **ASR + SQL** | **ASR + NLI** |
|---|---|---|
| Prior knowledge | Subset of SQL & Schema | Tables queried |
| Ambiguity of query intent | None | Possible |
| Result Correctness guarantee | Always | Unclear how to support |
| Non-dictionary schema literal | Can be supported (user knows schemas) | Unclear how to support |
| Resolving ASR errors | Can exploit SQL grammar & schema | Unclear how to support |
| Main difficulty metric for queries | Token length | Number of join paths, aggregates, and nesting |

Table 3: Contrasting Spoken NLI systems against Spoken SQL systems on different attributes.

# C. SQL GRAMMAR

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

# D. STRUCTURE DETERMINATION

## D.1 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 9. 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.

| | | SELECT | * | FROM | x |
|---|---|---|---|---|---|
| | 0 | 1.2 | 2.3 | 3.5 | 4.5 |
| SELECT | 1.2 | 0 | 1.1 | 2.3 | 3.3 |
| x | 2.2 | 1 | 2.1 | 3.3 | 2.3 |
| x | 3.2 | 2 | 3.1 | 4.3 | 3.3 |
| FROM | 4.4 | 3.2 | 4.3 | 3.1 | 4.1 |
| x | 5.4 | 4.2 | 5.3 | 4.1 | **3.1** |

Figure 9: Dynamic programming memo that computes edit distance between the `MaskOut` and `GrndTrth`.

## D.2 Bidirectional Bounds (BDB) Example

To illustrate how our bounds could be useful, Figure 10 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 is 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.

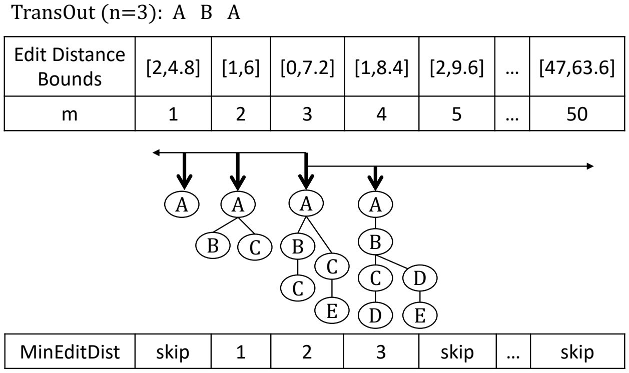
Box 2: Structure Determination Algorithm

```
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) $\leq$ MinEditDist then
         for token in node.children do
          SearchRecursively(node.children[token],token,DpCurCol)
         end for
      end if
   end for
end procedure
```

## D.3 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



TransOut (n=3): A B A

| Edit Distance Bounds | [2,4.8] | [1,6] | [0,7.2] | [1,8.4] | [2,9.6] | … | [47,63.6] |
|---|---|---|---|---|---|---|---|
| m | 1 | 2 | 3 | 4 | 5 | … | 50 |

| MinEditDist | skip | 1 | 2 | 3 | skip | … | skip |
|---|---|---|---|---|---|---|---|

Figure 10: Bidirectional Bounds example.

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 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.

## D.4 Proofs

**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) $\in$ (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 $\neq$ EndIndex do
        j = i; k=0; curstr = ""
        while TransOut(j) $\notin$ (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) = $\phi$; minEditDist = $\infty$
        for every item b in B do
            if EditDist(a,b) < minEditDist then
                set(a) = $\phi$; 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 = $argmax_{b \in B}$ (count(b))
    k = location(b)
    return literal, k
end procedure
```

# E. LITERAL DETERMINATION

## E.1 Algorithm

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

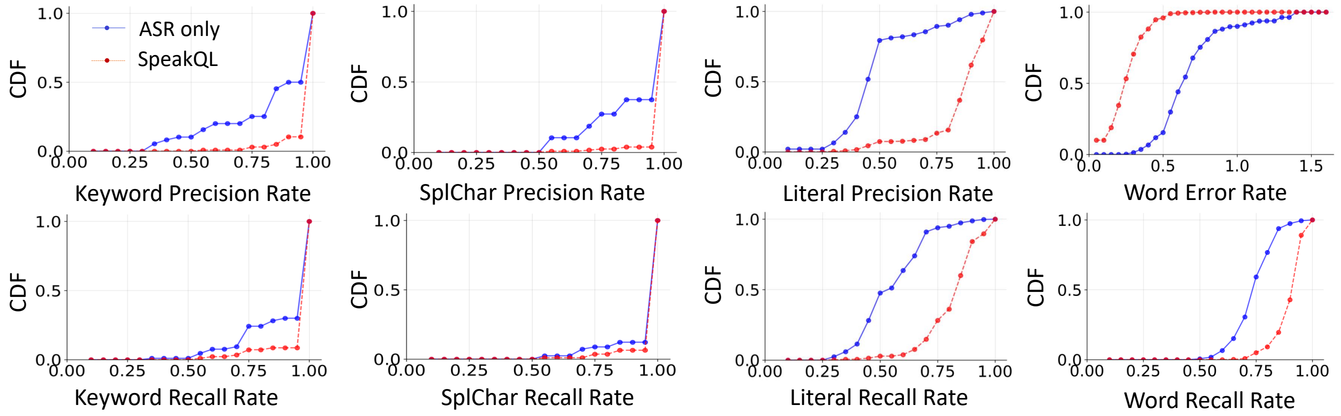


Figure 11: Cumulative distribution of accuracy metrics for top 1 results from Employees test dataset.

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.

## E.2 Literal Assignment Examples

As the final step of the literal determination algorithm, 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. The two examples below illustrate 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.

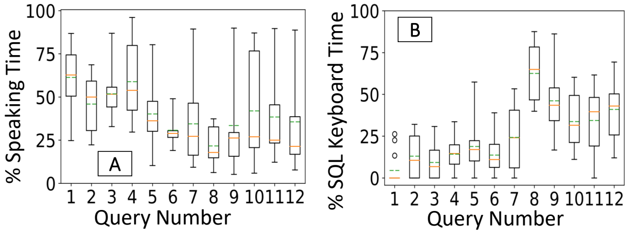


Figure 12: *Simple* queries are marked from 1 to 6 and the rest are *complex.* (A) Fraction of time spent in dictating the query relative to the total end-to-end time (B) Fraction of time spent in using the SQL keyboard relative to the total end-to-end time.

# F. END-TO-END EVALUATION

## F.1 End-to-End system Accuracy

Figure 11 plots the CDF of the error metrics for the queries on the Employees database.

## F.2 User Study

**Preliminary User Study.** In our first pilot user study, we recruited 15 participants; each composed 12 SQL queries on the Employees database. Only English description of each 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 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 in comparison with raw typing. We realized this was because participants were recruited without vetting them for their SQL knowledge. Thus, many participants had little experience composing SQL queries. As a result, many 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 taught us several key 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

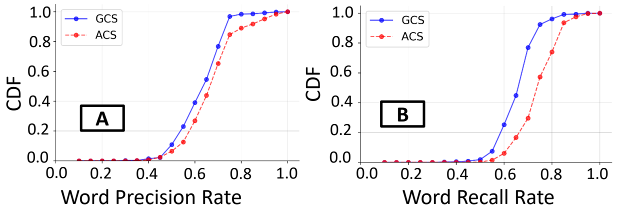


Figure 13: Cumulative Distribution of accuracy metrics for top 1 results

| | 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 |

Table 4: Mean error metrics: Precision and Recall

data professionals that are already familiar with SQL and the schemas they likely query regularly. Thus, we need more vetting. (2) It was difficult for users to compose the entire query in their 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 a maximum of only 10 seconds [40, 44]. 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 on drag and drop effort. Hence, supporting an SQL keyboard that allows users to quickly insert or delete any incorrectly placed or transcribed token using just a touch could make our interface more correction friendly. In addition, an SQL keyboard will allow a user to correct a query (or sub-query) out of order.

**User Study Drill Down.** We drill down deeper to see how a user is interacting with SpeakQL. Figure 12 shows the % time of the total end-to-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.

## F.3 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. We 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 4, 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 .

## F.4 Structure Determination Latency

Figure 14 shows the CDF of time taken by this component. We see that the latency is less than 1.5 s for almost 99% of queries.

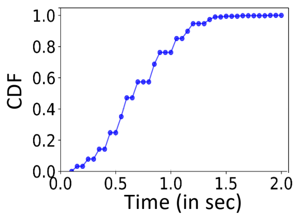


Figure 14: Structure Determination component evaluation on Latency.

## F.5 Structure Determination 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 15 (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 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. Figure 15 (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.

## F.6 Literal Determination drill down

Figure 16(A) presents the CDF of recall rates for table names, attribute names, and 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 break down attribute value into different types.

Attribute values can either be a date, a real number, or a string value. Figure 16(B) shows CDF of edit distance for these different types with the ground truth. This shows how much editing effort the user needs to correct an attribute value. For example, almost 50% of the attribute

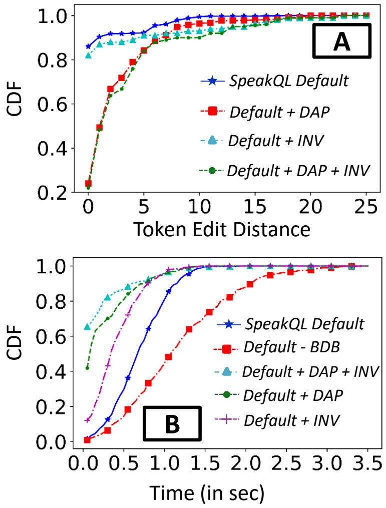


Figure 15: Ablation study of Structure Determination Component

values of type string were correctly retrieved with a phonetic edit distance of 0. On the other hand, only 35% of dates were returned perfectly. Dictating dates requires successfully transcribing 3 tokens: day, month, and year; ASR either omits or wrongly transcribes one of these 3 tokens, leading to an increase in correction for dates. Finally, only 23% of numbers were detected exactly. This is because ASR messes up when transcribing a number spoken with pauses. e.g., “forty five thousand three hundred ten” is transcribed as “45000 310.”

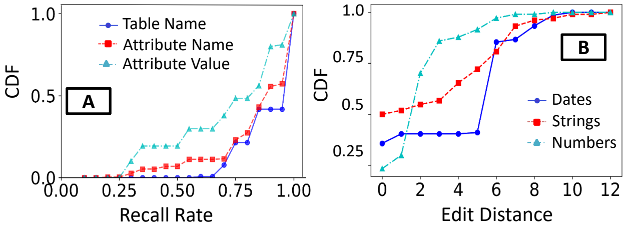


Figure 16: (A) CDF of Recall Rates for different Literal types (B) CDF of edit distances for different attribute value types. Edit distance evaluation for strings is phonetic, while for Dates and Numbers is character-level.

## F.7 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 17, 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.

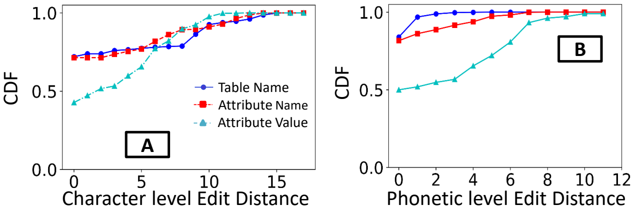


Figure 17: Evaluation of Character level Edit distance vs Phonetic level Edit Distance

## F.8 Evaluation with nested queries

To support one-level nested queries in SpeakQL, we employed a heuristic to detect if there exist a nested query inside a query. If detected then we replace the substring containing the nested query with a special placeholder variable. We then apply our structure and literal determination independently to both the original and the nested query. Note that our dataset of spoken SQL queries does not contain any nested queries. Since the major difficulty metric for our system is not nesting but the number of tokens in the query, this does not affect our evaluation. However, for the sake of completeness, we evaluate SpeakQL on one-level nested queries from Spider dataset [54] and present results in Figure 18.

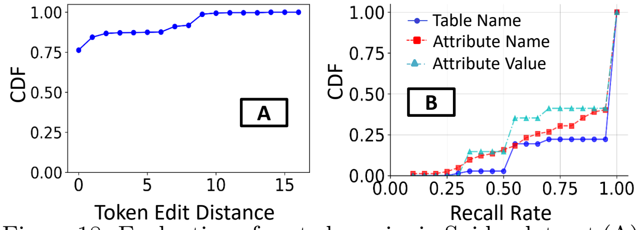
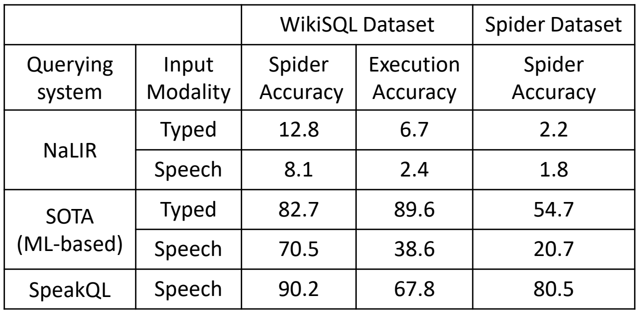


Figure 18: Evaluation of nested queries in Spider dataset (A) Structure Determination component with TED (B) Literal Determination component with LRR.

## F.9 Comparison against NLIs

In Section 1, we discussed that ASR can introduce a variety of errors in the transcription and can cause an NLI to fail more when queries are spoken as to when queries are typed. In this section, we show that the accuracy decreases significantly when queries are speech-based than typing-based for NLIs. In addition, we show that by leveraging “structure” in SQL, SpeakQL can help bridge the gap in accuracy significantly.

**Datasets and NLIs.** We use two large-scale human-annotated datasets containing pairs of typed natural language queries and typed SQL queries for evaluation: WikiSQL [55], consisting of 15878 pairs and Spider [54], consisting of 1034 pairs in the test set. For NLIs, we choose NaLIR [37] as baseline and state-of-the-art (SOTA) ML-based approaches for semantic parsing (natural language to SQL task): SQLova [34] on WikiSQL and IRNet [33] on Spider. To the best of



| | | WikiSQL Dataset | | Spider Dataset |
|---|---|---|---|---|
| Querying system | Input Modality | Spider Accuracy | Execution Accuracy | Spider Accuracy |
| NaLIR | Typed | 12.8 | 6.7 | 2.2 |
| | Speech | 8.1 | 2.4 | 1.8 |
| SOTA (ML-based) | Typed | 82.7 | 89.6 | 54.7 |
| | Speech | 70.5 | 38.6 | 20.7 |
| SpeakQL | Speech | 90.2 | 67.8 | 80.5 |

Table 5: Comparison of SpeakQL against NLIs.

our knowledge, there does not exist any general-purpose open-source spoken NLI for evaluation. Thus, we adapt existing typed NLI for speech-based inputs. We use Amazon Polly [19] speech synthesis API to obtain both spoken natural language and spoken SQL queries from the typed queries. Next, we use Azure's Speech Services' API [20] to obtain typed transcriptions. We then run the transcriptions through different NLIs.

**Evaluation Metric.** We use 2 different metrics for accuracy: (1) Spider's exact match accuracy score [54]. This evaluation script decomposes the query into different clauses such as *SELECT*, *WHERE*, and so on. The predicted query is correct only if the set of components match with those of ground truth. (2) Execution accuracy of the query. The predicted query is correct only if the results returned by the predicted query and the ground query match exactly. Note that the current text-to-SQL parsing task on the Spider dataset does not involve generating values in the SQL conditions. Thus, we can not evaluate execution accuracy on Spider dataset.

**Results with NLIs when queries are typed.** We present a comparison of SpeakQL against NLIs in Table 5. To present a fair comparison with other approaches, we evaluate NaLIR in the non-interactive setting. We find that NaLIR often fails when the query is posed as a question in contrast to when posed as a statement. Thus, we convert all queries with questions to statements for evaluating only NaLIR. For example, "what is x?" is converted to "show me x." We find that the accuracy of NaLIR on both datasets is quite low. This is because NaLIR is unable to resolve ambiguities automatically but rather relies on user interactions for such resolution. On the other hand, SQLova and IRNet achieve the best accuracy on respective datasets with typed natural language queries.

**Results with NLIs when queries are spoken.** Interestingly, with spoken natural language queries as input, the accuracy drops significantly in comparison with typed queries. For instance, the speech input on SQLova model leads to a whooping 50% drop in the execution accuracy as compared with the typed input. Wrong transcription of even a single token (such as "and" transcribed to "in") changes the meaning of the query completely and "fools" the model into wrong semantic parsing. We also conduct another experiment on the Spider dataset where commas in the natural language query are spoken out. We find that speaking out such structural element in the query raises the Spider accuracy score of IRNet model by 2%. The lift in overall accuracy is not high enough since there are only 5% queries with commas. We find that speaking commas helped 34 out of 45 queries to deliver the correct SQL, which is significant.

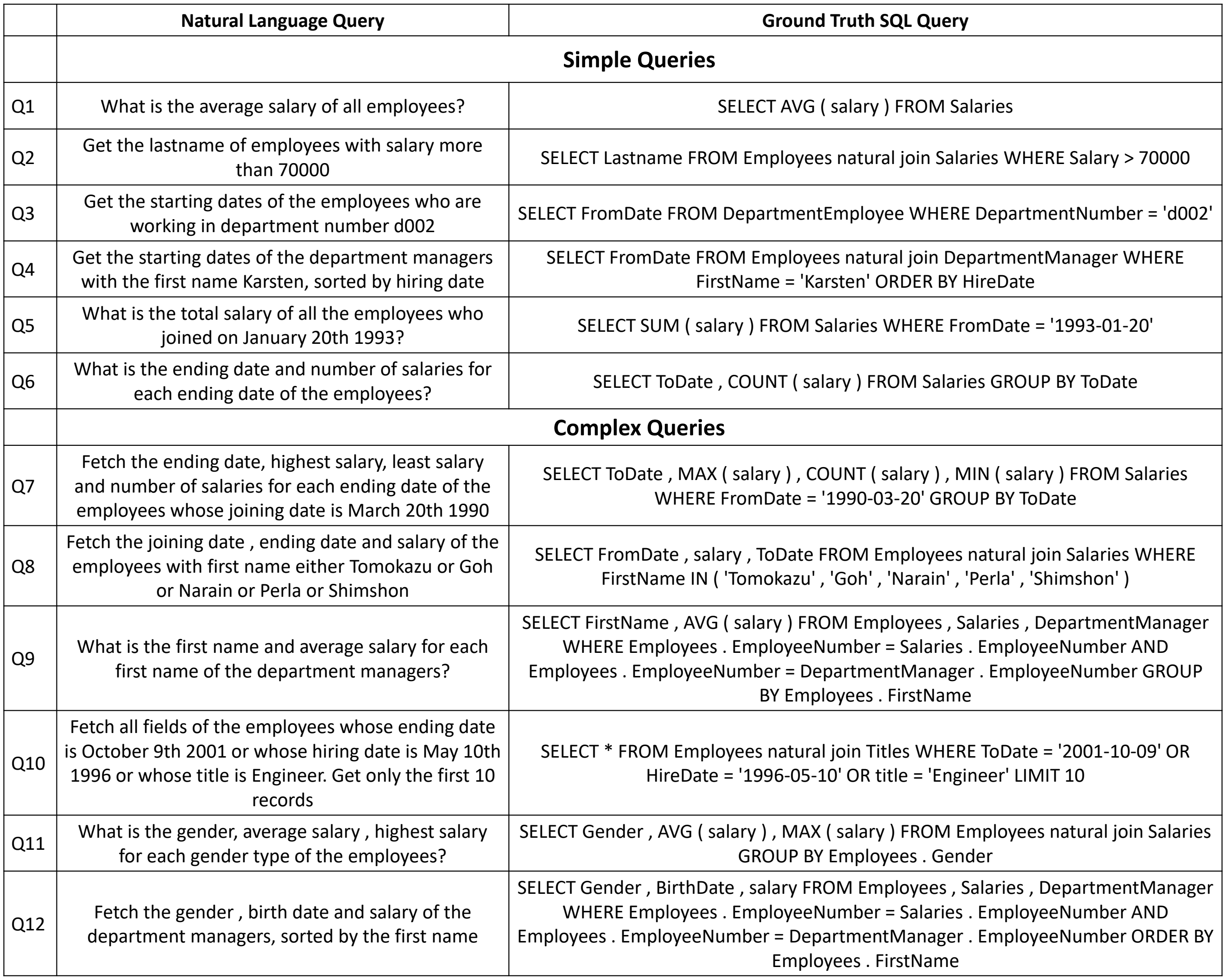
**Results with SpeakQL.** SpeakQL accepts spoken SQL queries as input where all special characters are dictated. We find that SpeakQL achieves significantly higher accuracy than the SOTA NLIs with speech inputs. For instance, we observe that the lift in Spider accuracy score is about 60% on Spider dataset. However, on the WikiSQL dataset, the execution accuracy of typed SOTA NLI is better than SpeakQL. This is because many literal values in the queries are long with special characters such as '#21/#07 SS-Green Light Racing', leading to ASR transcription errors and therefore causing SpeakQL to fail. In such cases, our human-in-the-loop interface can help users to interactively correct such errors. We also saw that the accuracy of SpeakQL largely depends upon the number of tokens in the query. A longer query can have more transcription errors than a shorter one and thus correction becomes harder.

## G. USER STUDY QUERY SET

Table 6 shows the query set used in the user studies.

## H. GOOGLE FORM QUESTIONNAIRE RESPONSES ROUND 1

## I. GOOGLE FORM QUESTIONNAIRE RESPONSES ROUND 2



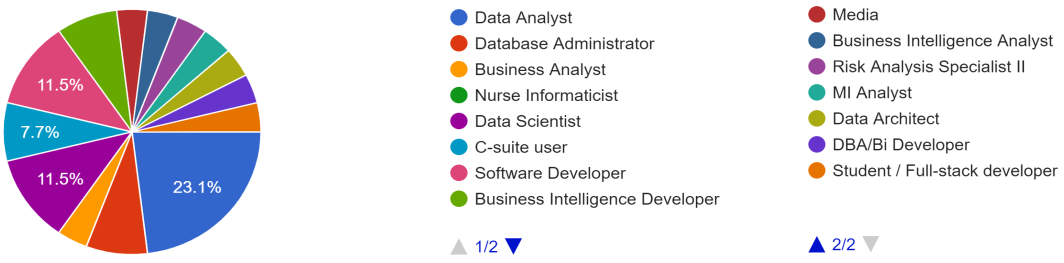
| | Natural Language Query | Ground Truth SQL Query |
|---|---|---|
| | **Simple Queries** | |
| Q1 | What is the average salary of all employees? | SELECT AVG ( salary ) FROM Salaries |
| Q2 | Get the lastname of employees with salary more than 70000 | SELECT Lastname FROM Employees natural join Salaries WHERE Salary > 70000 |
| Q3 | Get the starting dates of the employees who are working in department number d002 | SELECT FromDate FROM DepartmentEmployee WHERE DepartmentNumber = 'd002' |
| Q4 | Get the starting dates of the department managers with the first name Karsten, sorted by hiring date | SELECT FromDate FROM Employees natural join DepartmentManager WHERE FirstName = 'Karsten' ORDER BY HireDate |
| Q5 | What is the total salary of all the employees who joined on January 20th 1993? | SELECT SUM ( salary ) FROM Salaries WHERE FromDate = '1993-01-20' |
| Q6 | What is the ending date and number of salaries for each ending date of the employees? | SELECT ToDate , COUNT ( salary ) FROM Salaries GROUP BY ToDate |
| | **Complex Queries** | |
| Q7 | Fetch the ending date, highest salary, least salary and number of salaries for each ending date of the employees whose joining date is March 20th 1990 | SELECT ToDate , MAX ( salary ) , COUNT ( salary ) , MIN ( salary ) FROM Salaries WHERE FromDate = '1990-03-20' GROUP BY ToDate |
| Q8 | Fetch the joining date , ending date and salary of the employees with first name either Tomokazu or Goh or Narain or Perla or Shimshon | SELECT FromDate , salary , ToDate FROM Employees natural join Salaries WHERE FirstName IN ( 'Tomokazu' , 'Goh' , 'Narain' , 'Perla' , 'Shimshon' ) |
| Q9 | What is the first name and average salary for each first name of the department managers? | SELECT FirstName , AVG ( salary ) FROM Employees , Salaries , DepartmentManager WHERE Employees . EmployeeNumber = Salaries . EmployeeNumber AND Employees . EmployeeNumber = DepartmentManager . EmployeeNumber GROUP BY Employees . FirstName |
| Q10 | Fetch all fields of the employees whose ending date is October 9th 2001 or whose hiring date is May 10th 1996 or whose title is Engineer. Get only the first 10 records | SELECT * FROM Employees natural join Titles WHERE ToDate = '2001-10-09' OR HireDate = '1996-05-10' OR title = 'Engineer' LIMIT 10 |
| Q11 | What is the gender, average salary , highest salary for each gender type of the employees? | SELECT Gender , AVG ( salary ) , MAX ( salary ) FROM Employees natural join Salaries GROUP BY Employees . Gender |
| Q12 | Fetch the gender , birth date and salary of the department managers, sorted by the first name | SELECT Gender , BirthDate , salary FROM Employees , Salaries , DepartmentManager WHERE Employees . EmployeeNumber = Salaries . EmployeeNumber AND Employees . EmployeeNumber = DepartmentManager . EmployeeNumber ORDER BY Employees . FirstName |

Table 6: Query set used for user study. Queries from 1 to 6 are *simple* and the rest are *complex*.

# Round 1

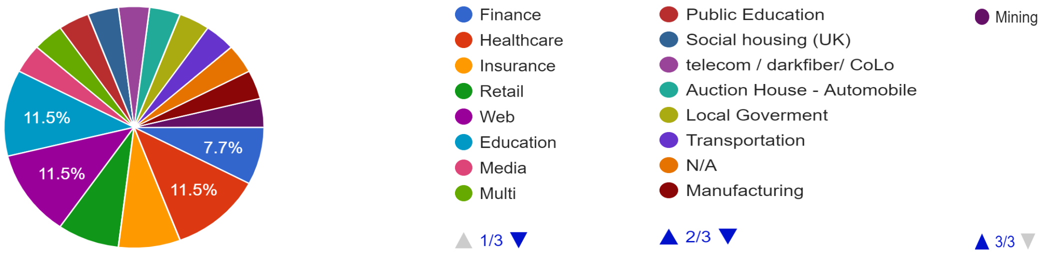
What is your job title?

26 responses



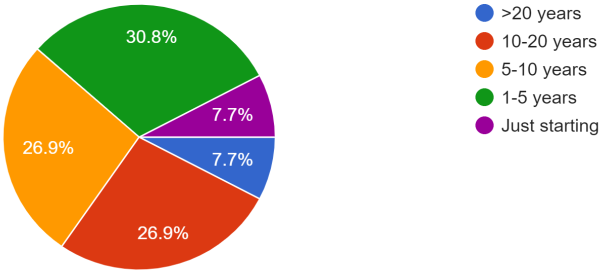
Which sector does your organization belong to?

26 responses



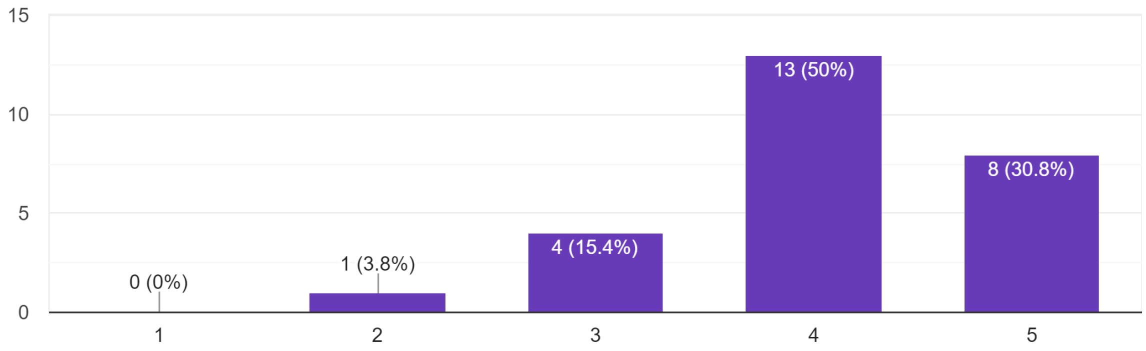
Your experience in the current role?

26 responses



What is your expertise level with SQL?

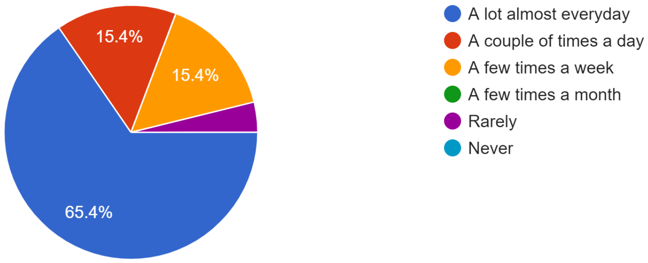
26 responses



Round 1

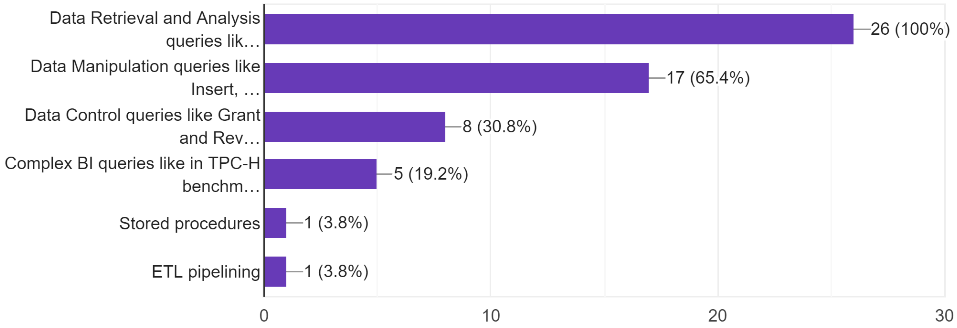
How often do you query your database with SQL?

26 responses



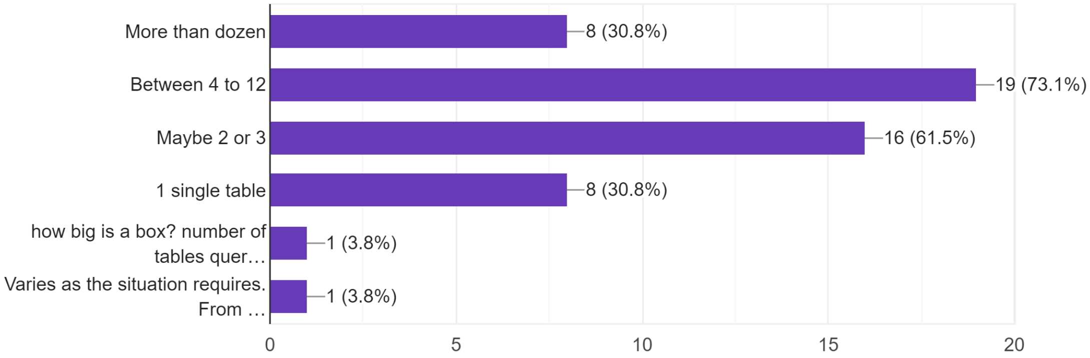
What all kinds of queries do you compose?

26 responses



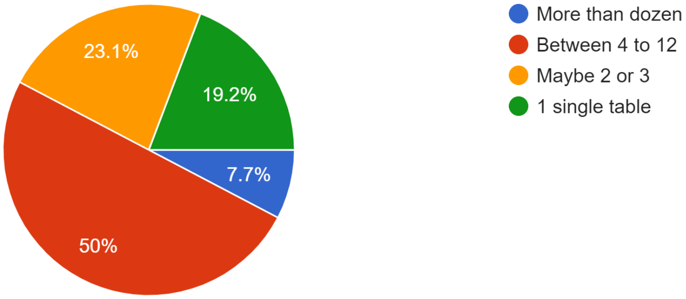
How many tables do you query at a time?

26 responses



Which one of the numbers of tables is perhaps the most frequent in queries you compose?

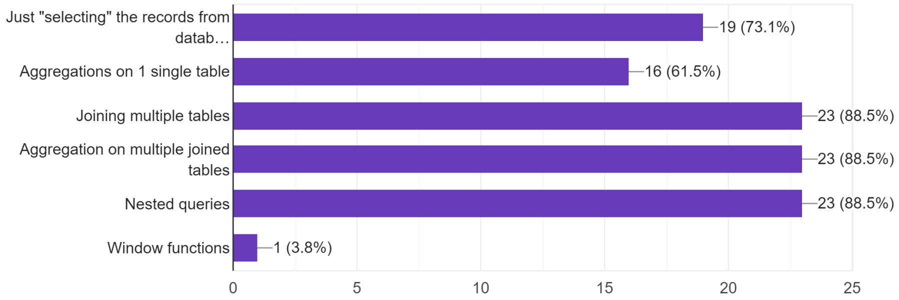
26 responses



Round 1

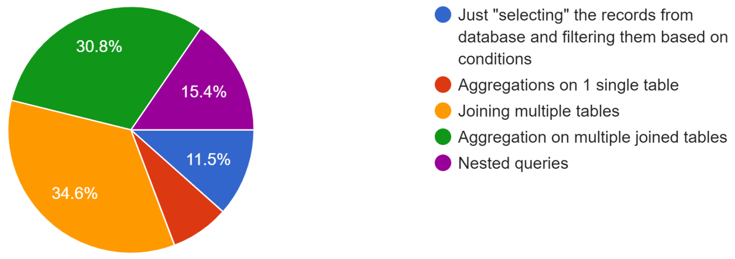
If you perform data retrieval and analysis, what all kinds of query complexity do your queries represent?

26 responses



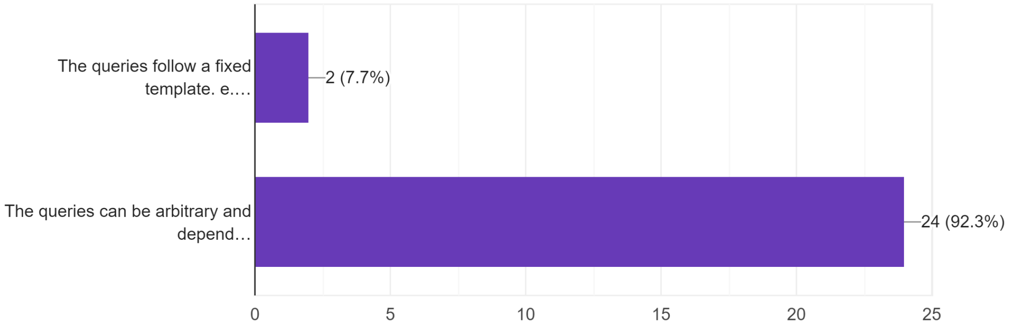
Which one of the query complexity is perhaps the most frequent one you compose?

26 responses



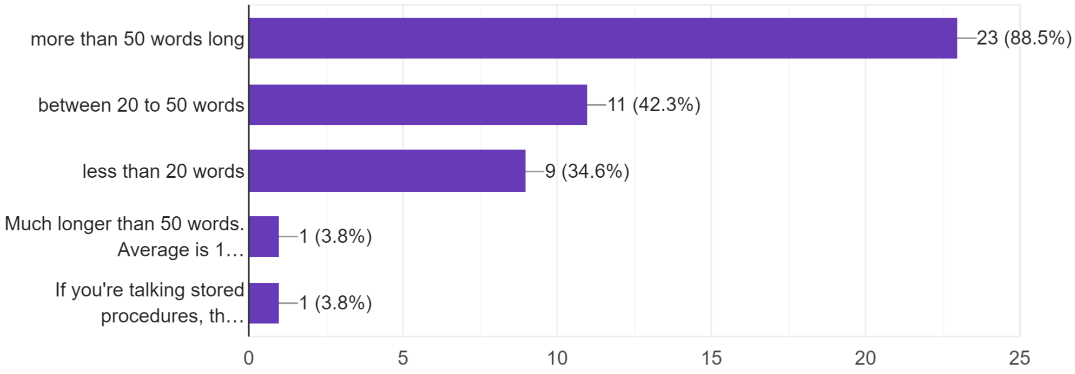
Nature of queries you compose

26 responses



What all lengths of queries do you compose?

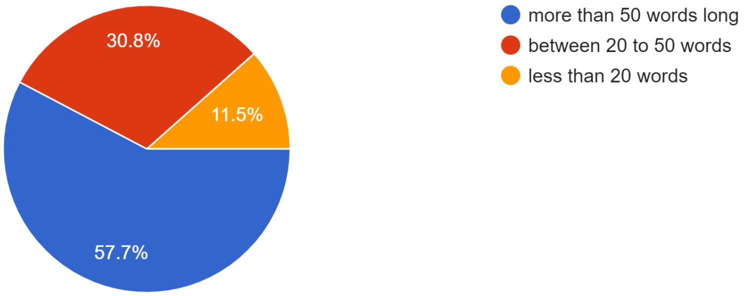
26 responses



Which one of the query lengths is perhaps the most frequent you compose?

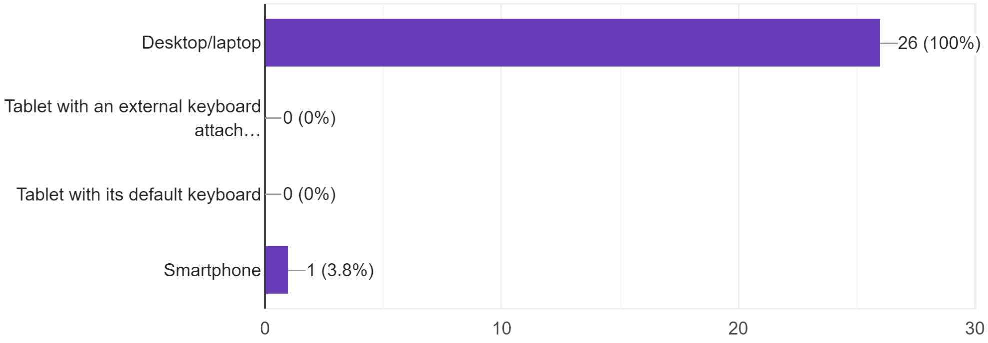
26 responses

**Round 1**



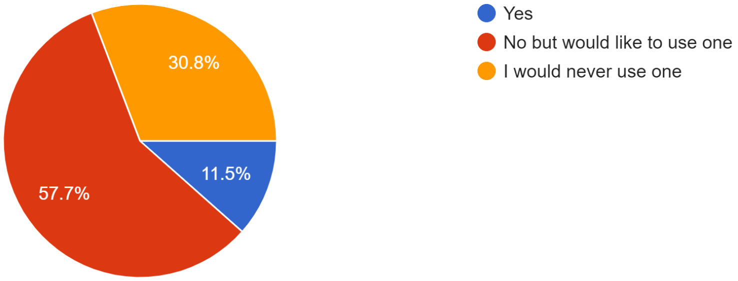
What kind of device do you use to query?

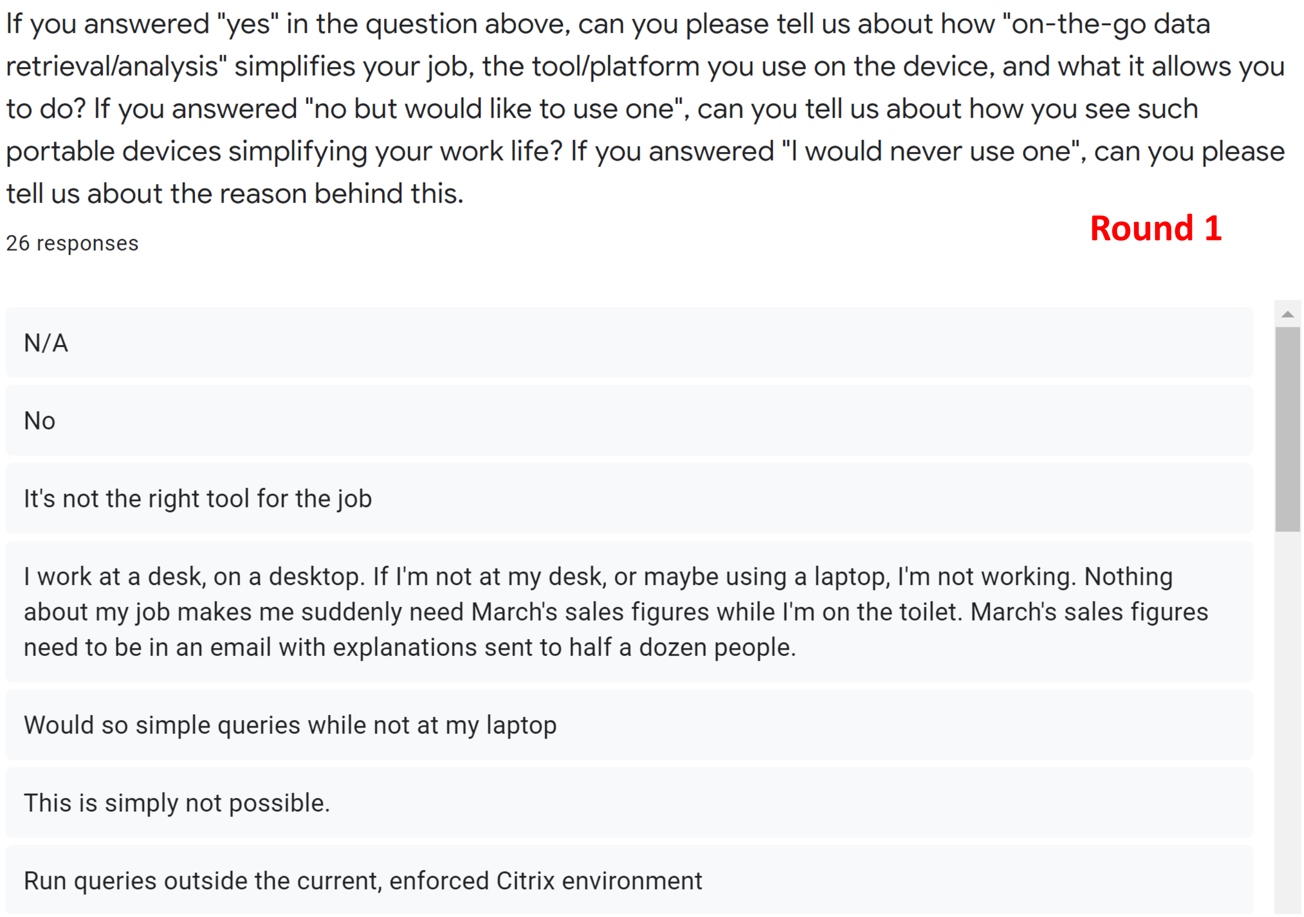
26 responses



Do you use or have considered using tablets/smartphones for data retrieval and analysis?

26 responses





If you answered "yes" in the question above, can you please tell us about how "on-the-go data retrieval/analysis" simplifies your job, the tool/platform you use on the device, and what it allows you to do? If you answered "no but would like to use one", can you tell us about how you see such portable devices simplifying your work life? If you answered "I would never use one", can you please tell us about the reason behind this.

**Round 1**

26 responses

N/A

No

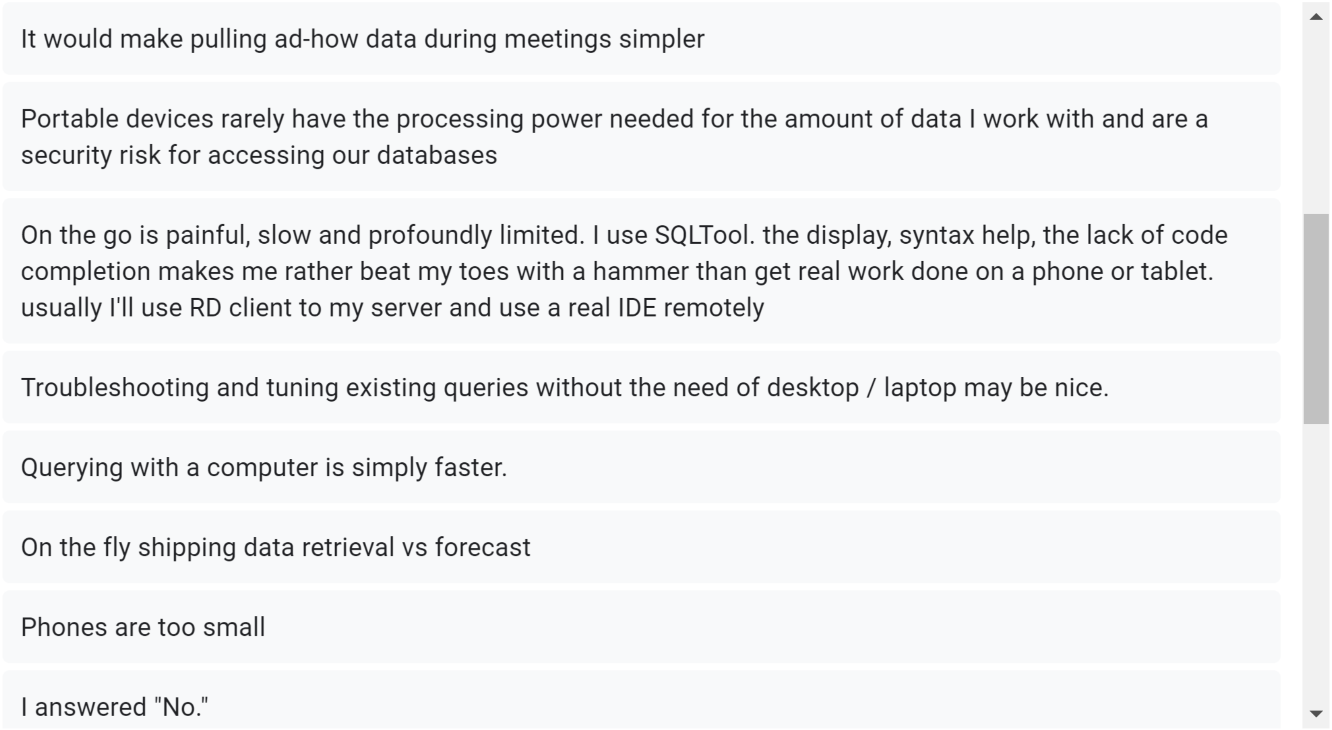
It's not the right tool for the job

I work at a desk, on a desktop. If I'm not at my desk, or maybe using a laptop, I'm not working. Nothing about my job makes me suddenly need March's sales figures while I'm on the toilet. March's sales figures need to be in an email with explanations sent to half a dozen people.

Would so simple queries while not at my laptop

This is simply not possible.

Run queries outside the current, enforced Citrix environment



It would make pulling ad-how data during meetings simpler

Portable devices rarely have the processing power needed for the amount of data I work with and are a security risk for accessing our databases

On the go is painful, slow and profoundly limited. I use SQLTool. the display, syntax help, the lack of code completion makes me rather beat my toes with a hammer than get real work done on a phone or tablet. usually I'll use RD client to my server and use a real IDE remotely

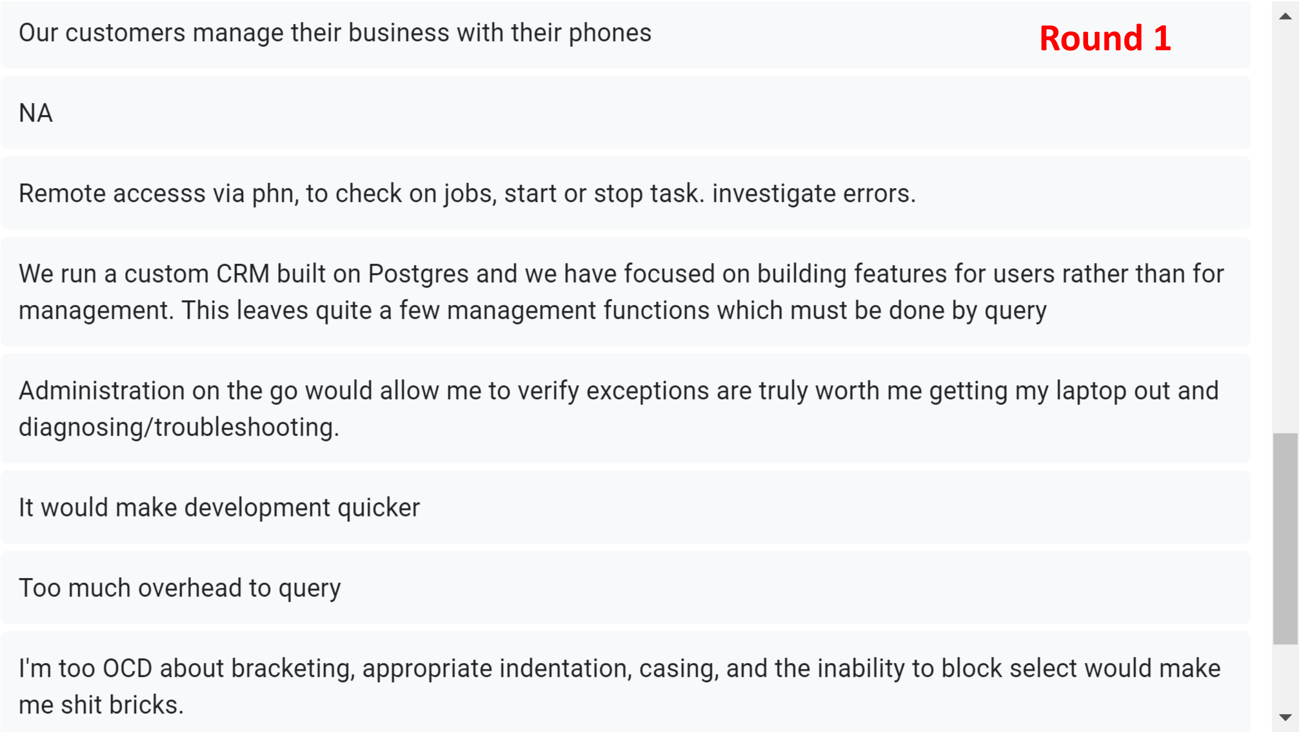
Troubleshooting and tuning existing queries without the need of desktop / laptop may be nice.

Querying with a computer is simply faster.

On the fly shipping data retrieval vs forecast

Phones are too small

I answered "No."



Round 1

Our customers manage their business with their phones

NA

Remote accesss via phn, to check on jobs, start or stop task. investigate errors.

We run a custom CRM built on Postgres and we have focused on building features for users rather than for management. This leaves quite a few management functions which must be done by query

Administration on the go would allow me to verify exceptions are truly worth me getting my laptop out and diagnosing/troubleshooting.

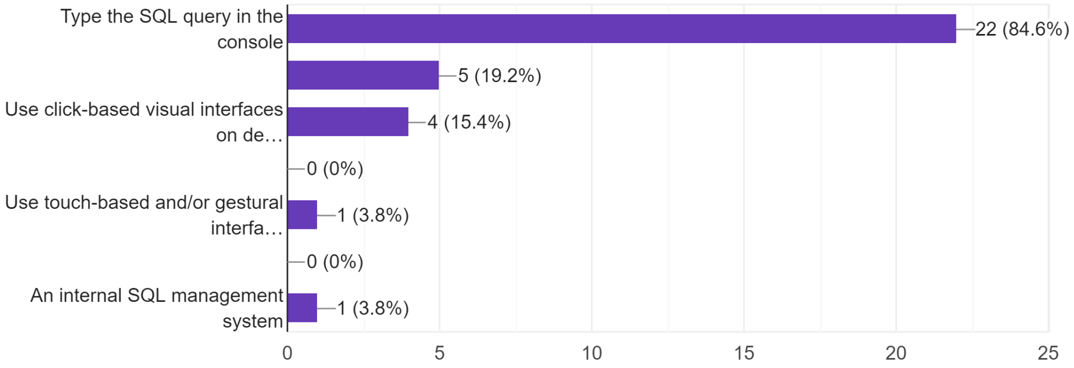
It would make development quicker

Too much overhead to query

I'm too OCD about bracketing, appropriate indentation, casing, and the inability to block select would make me shit bricks.

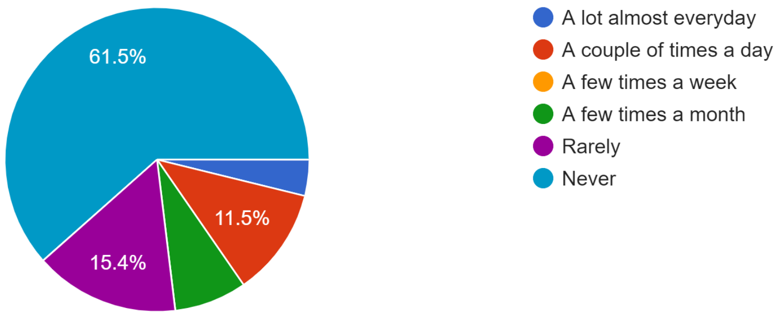
How do you query your dataset?

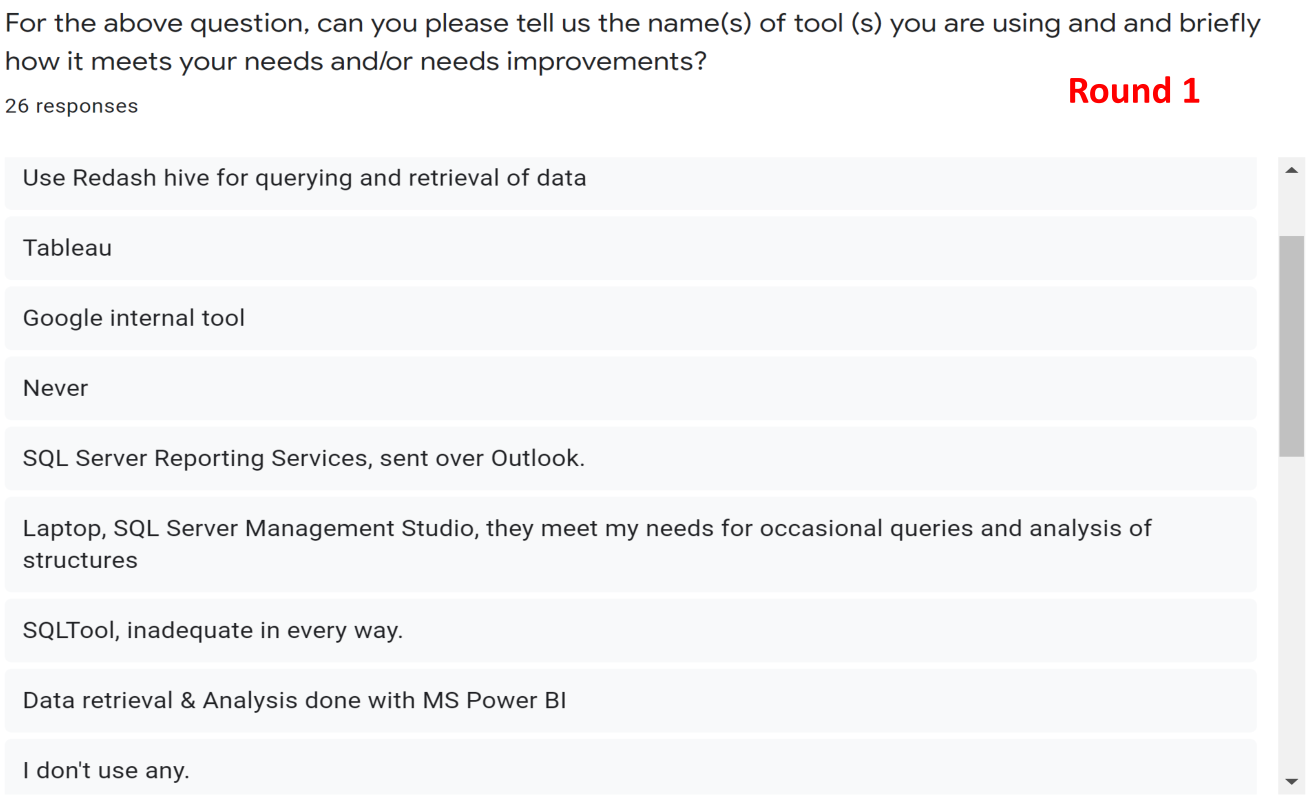
26 responses



How often do you retrieve/analyze your data on a tablet/smartphone environment?

26 responses





For the above question, can you please tell us the name(s) of tool (s) you are using and and briefly how it meets your needs and/or needs improvements?

26 responses

**Round 1**

Use Redash hive for querying and retrieval of data

Tableau

Google internal tool

Never

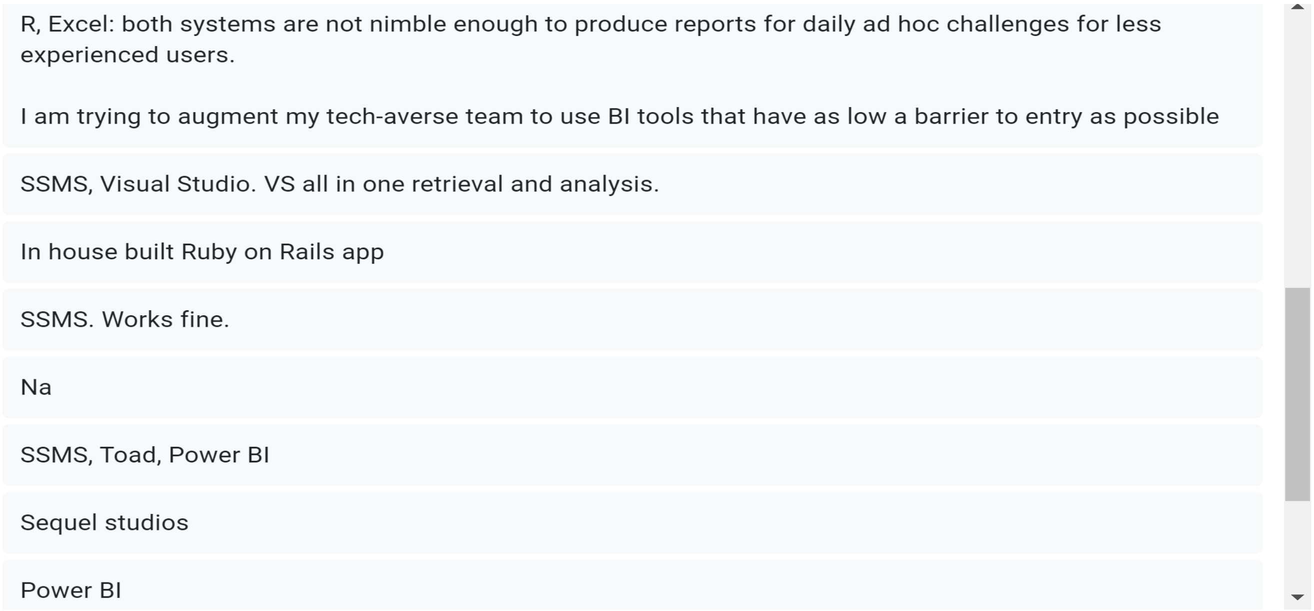
SQL Server Reporting Services, sent over Outlook.

Laptop, SQL Server Management Studio, they meet my needs for occasional queries and analysis of structures

SQLTool, inadequate in every way.

Data retrieval & Analysis done with MS Power BI

I don't use any.



R, Excel: both systems are not nimble enough to produce reports for daily ad hoc challenges for less experienced users.

I am trying to augment my tech-averse team to use BI tools that have as low a barrier to entry as possible

SSMS, Visual Studio. VS all in one retrieval and analysis.

In house built Ruby on Rails app

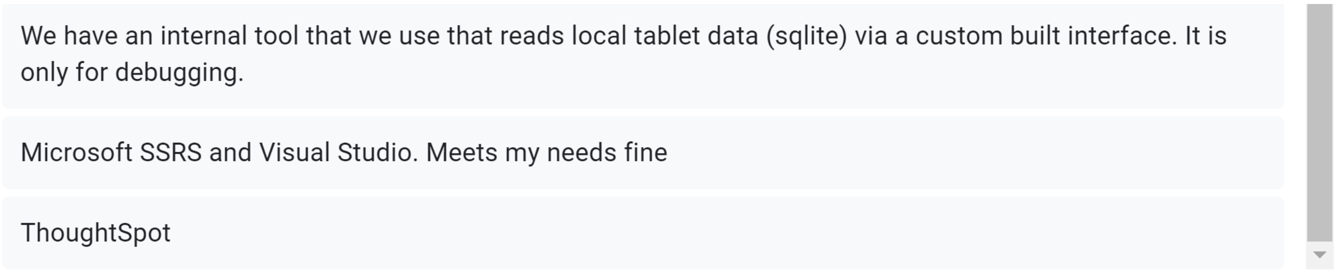
SSMS. Works fine.

Na

SSMS, Toad, Power BI

Sequel studios

Power BI



We have an internal tool that we use that reads local tablet data (sqlite) via a custom built interface. It is only for debugging.

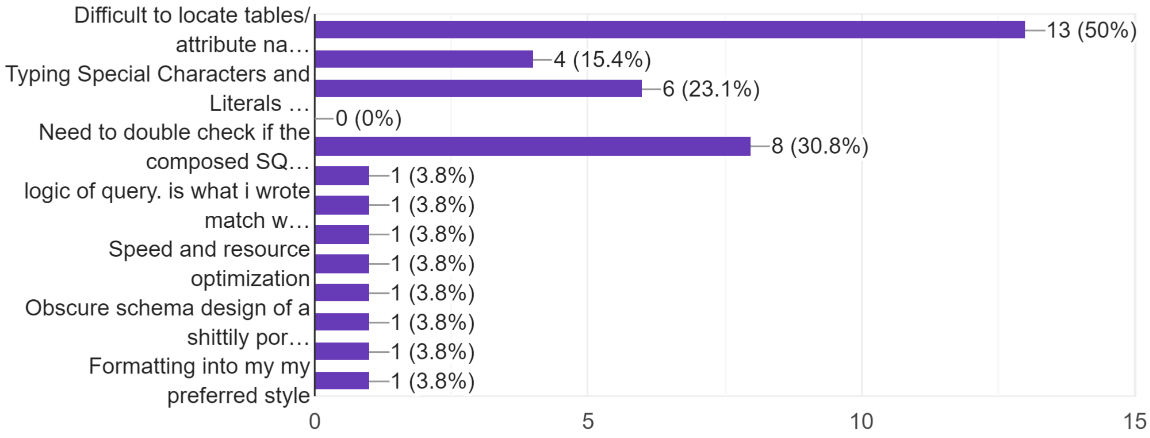
Microsoft SSRS and Visual Studio. Meets my needs fine

ThoughtSpot

What kinds of challenges do you sometimes encounter when composing an SQL query?

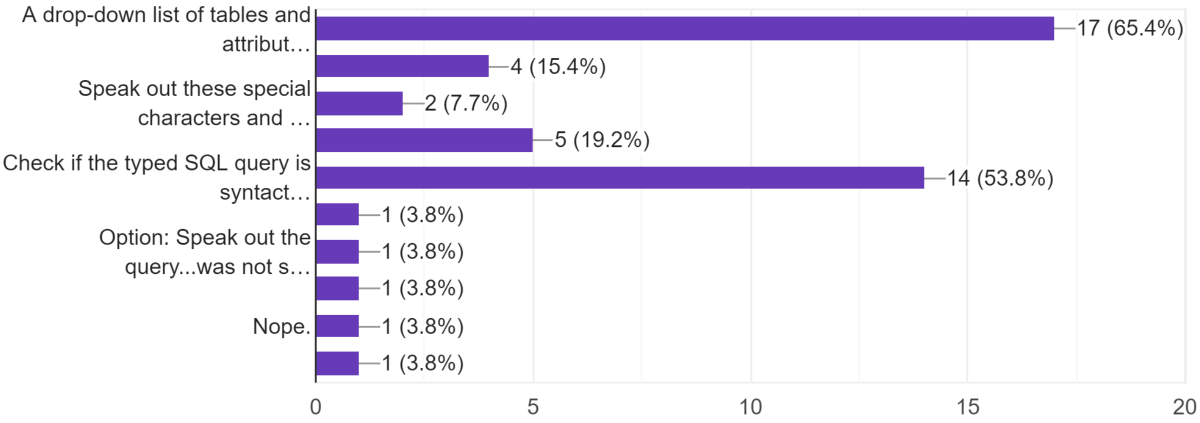
26 responses

**Round 1**



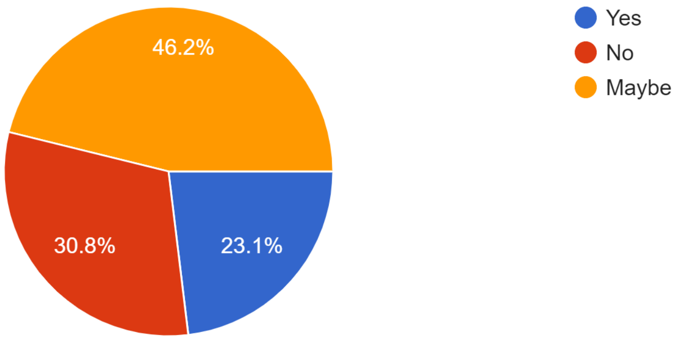
For the challenges above, do you think that the following solutions help?

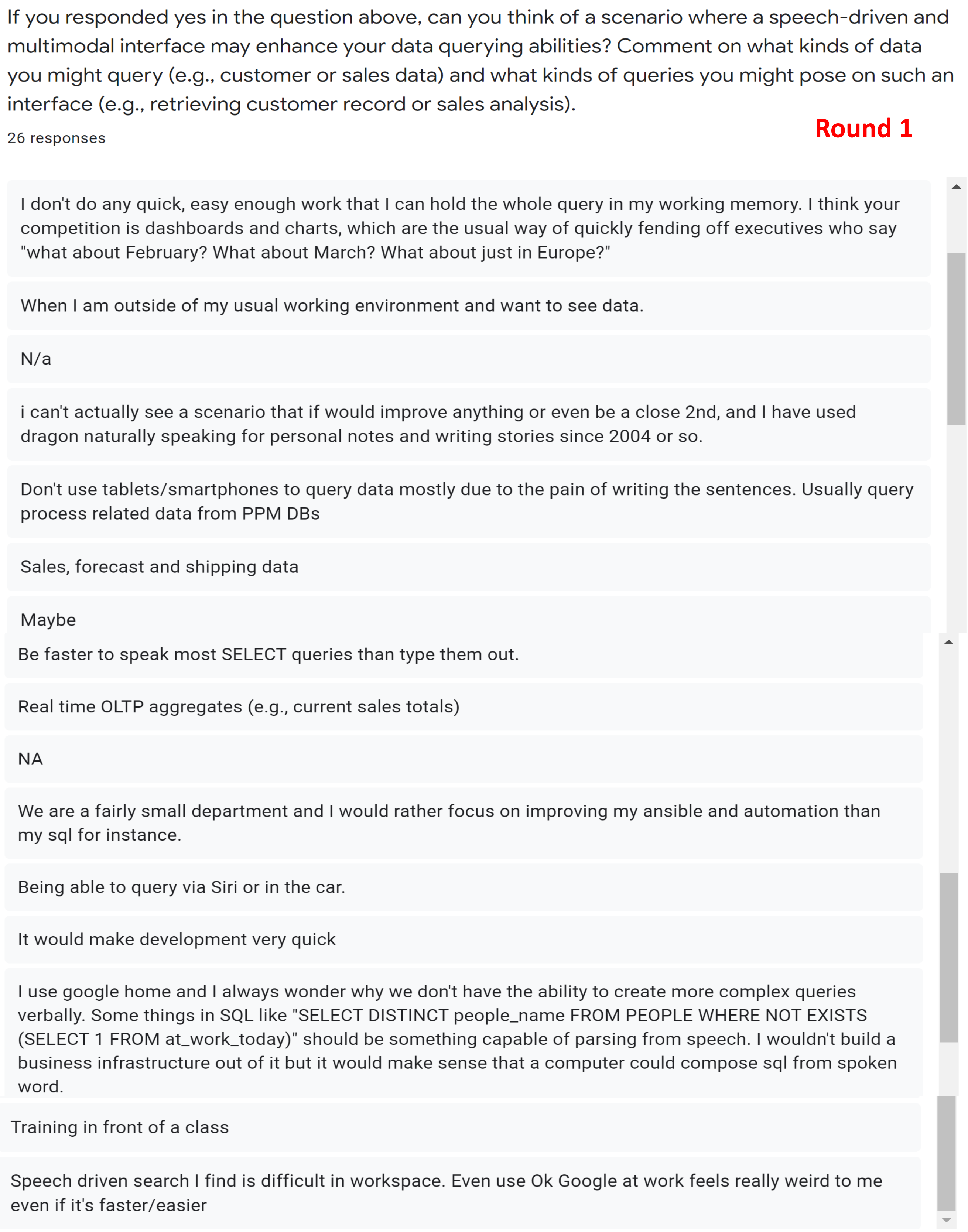
26 responses



Would you be interested in a speech-first SQL-aware querying tool on handheld devices (e.g., tablets/smartphones) where you can dictate your q...nteractive correction using touch and/or speech?

26 responses





If you responded yes in the question above, can you think of a scenario where a speech-driven and multimodal interface may enhance your data querying abilities? Comment on what kinds of data you might query (e.g., customer or sales data) and what kinds of queries you might pose on such an interface (e.g., retrieving customer record or sales analysis).

26 responses

**Round 1**

I don't do any quick, easy enough work that I can hold the whole query in my working memory. I think your competition is dashboards and charts, which are the usual way of quickly fending off executives who say "what about February? What about March? What about just in Europe?"

When I am outside of my usual working environment and want to see data.

N/a

i can't actually see a scenario that if would improve anything or even be a close 2nd, and I have used dragon naturally speaking for personal notes and writing stories since 2004 or so.

Don't use tablets/smartphones to query data mostly due to the pain of writing the sentences. Usually query process related data from PPM DBs

Sales, forecast and shipping data

Maybe

Be faster to speak most SELECT queries than type them out.

Real time OLTP aggregates (e.g., current sales totals)

NA

We are a fairly small department and I would rather focus on improving my ansible and automation than my sql for instance.

Being able to query via Siri or in the car.

It would make development very quick

I use google home and I always wonder why we don't have the ability to create more complex queries verbally. Some things in SQL like "SELECT DISTINCT people_name FROM PEOPLE WHERE NOT EXISTS (SELECT 1 FROM at_work_today)" should be something capable of parsing from speech. I wouldn't build a business infrastructure out of it but it would make sense that a computer could compose sql from spoken word.

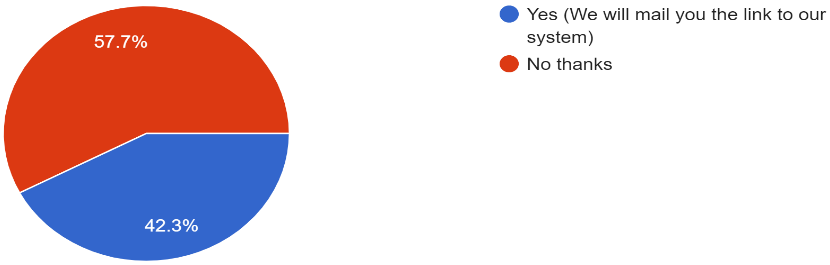
Training in front of a class

Speech driven search I find is difficult in workspace. Even use Ok Google at work feels really weird to me even if it's faster/easier

We have a working prototype of such a speech-driven multimodal querying interface. Our initial research shows it can reduce SQL query specificati...g. Are you interested in trying out our interface?

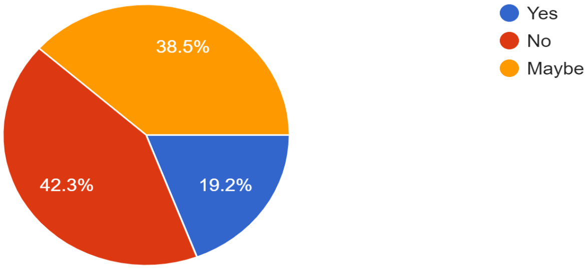
26 responses

**Round 1**



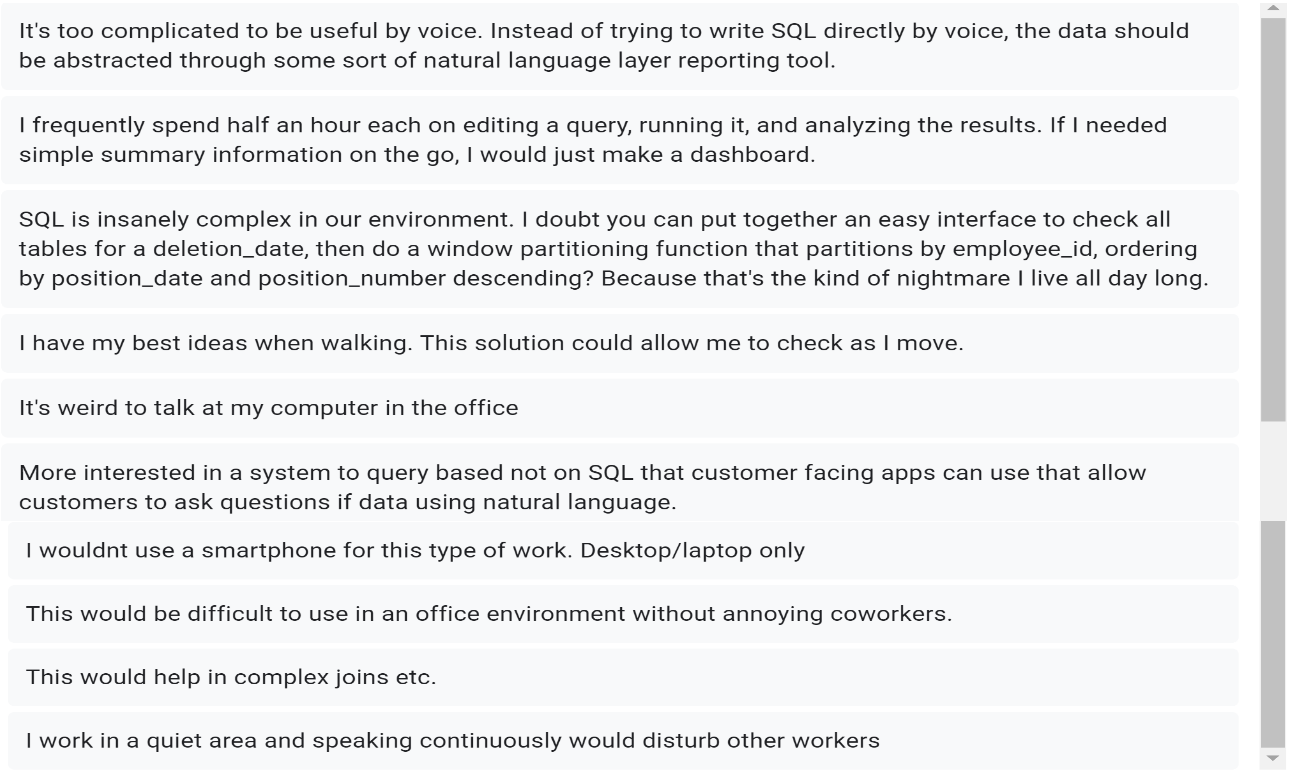
Would you be interested in a totally hands-free extension to such speech-driven query interfaces (i.e., no touch/typing) that lets you correct errors with voice conversations alone?

26 responses



If you said No, please explain briefly why a hands-free option does not appeal to you. If you said Yes or Maybe, can you think of a scenario where such a hands-free interface can enhance your data querying abilities? Again, comment on the kinds of data and queries you might pose.

10 responses



It's too complicated to be useful by voice. Instead of trying to write SQL directly by voice, the data should be abstracted through some sort of natural language layer reporting tool.

I frequently spend half an hour each on editing a query, running it, and analyzing the results. If I needed simple summary information on the go, I would just make a dashboard.

SQL is insanely complex in our environment. I doubt you can put together an easy interface to check all tables for a deletion_date, then do a window partitioning function that partitions by employee_id, ordering by position_date and position_number descending? Because that's the kind of nightmare I live all day long.

I have my best ideas when walking. This solution could allow me to check as I move.

It's weird to talk at my computer in the office

More interested in a system to query based not on SQL that customer facing apps can use that allow customers to ask questions if data using natural language.

I wouldnt use a smartphone for this type of work. Desktop/laptop only

This would be difficult to use in an office environment without annoying coworkers.

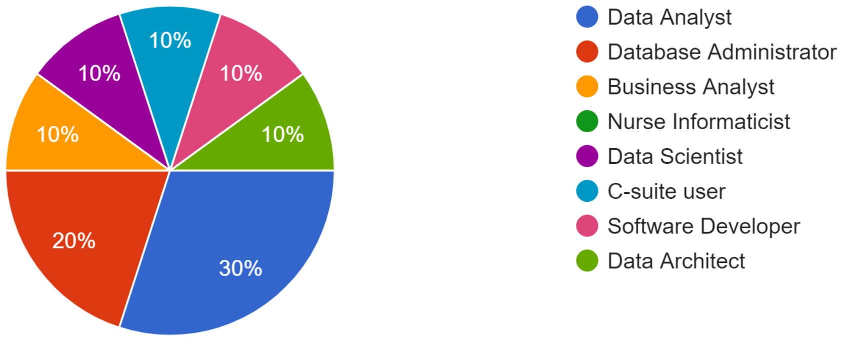
This would help in complex joins etc.

I work in a quiet area and speaking continuously would disturb other workers

What is your job title?

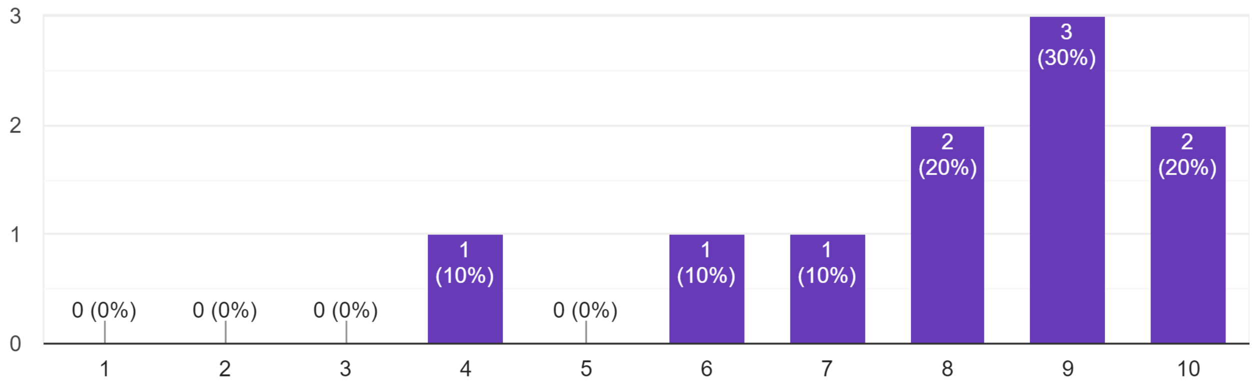
10 responses

**Round 2**



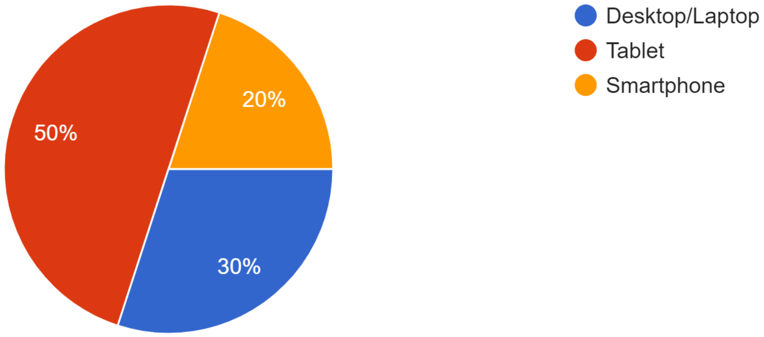
(1) On a rating scale from 1 to 10, how do you rate your SQL expertise (1 being “do not know SQL at all” and 10 being "professional-level")?

10 responses



What device did you use to try out our system?

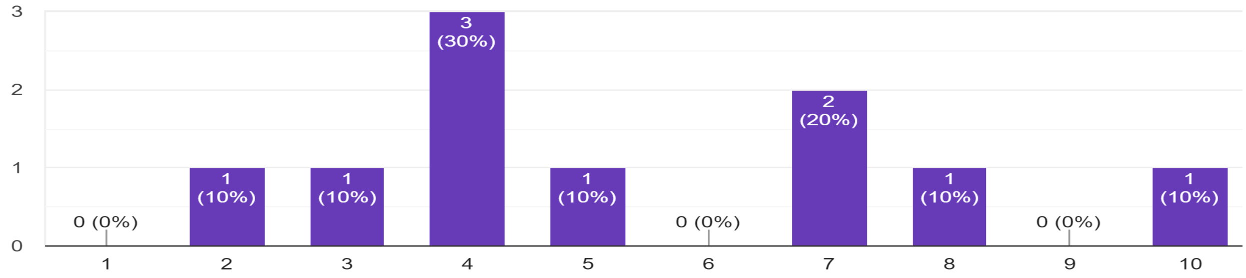
10 responses



(2) On a rating scale from 1 to 10, how do you rate your difficulty/experience with typing the SQL queries on the device (1 being "most difficult/painful" and 10 being "very easy and seamless")?

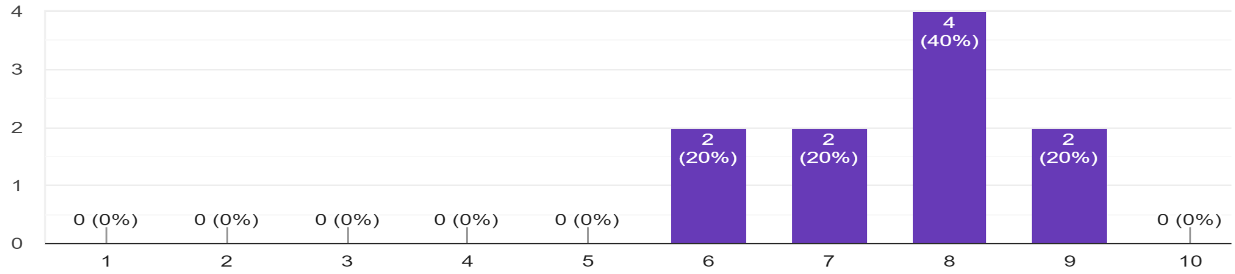
10 responses

**Round 2**



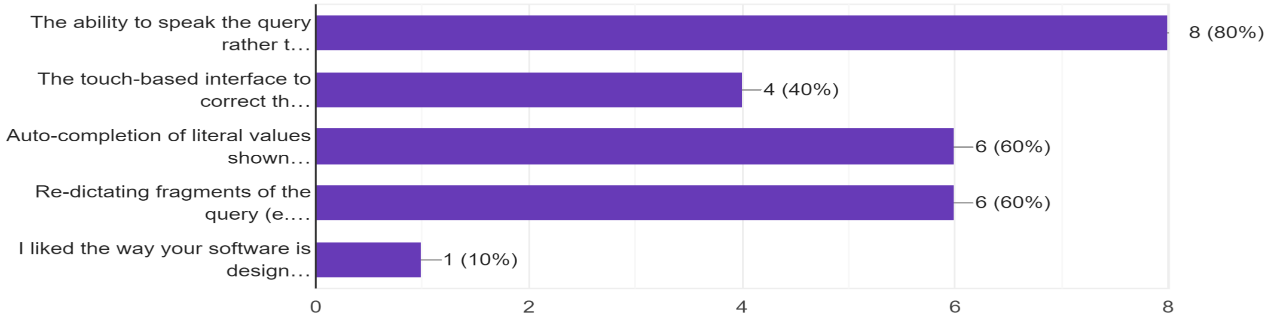
(3) On a rating scale from 1 to 10, how do you rate the difficulty/experience with dictating the queries and using the SpeakQL interface on the de.../painful" and 10 being "very easy and seamless")?

10 responses



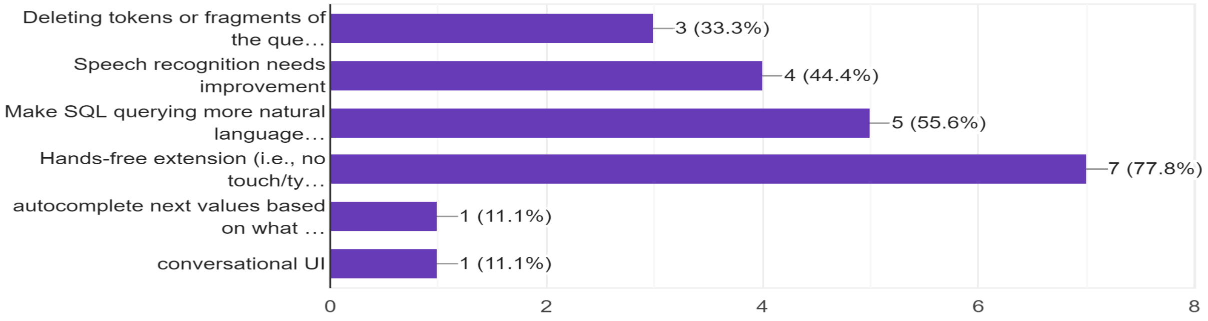
(4) What aspects of SpeakQL did you like the most and/or find the most useful (check all options that apply)?

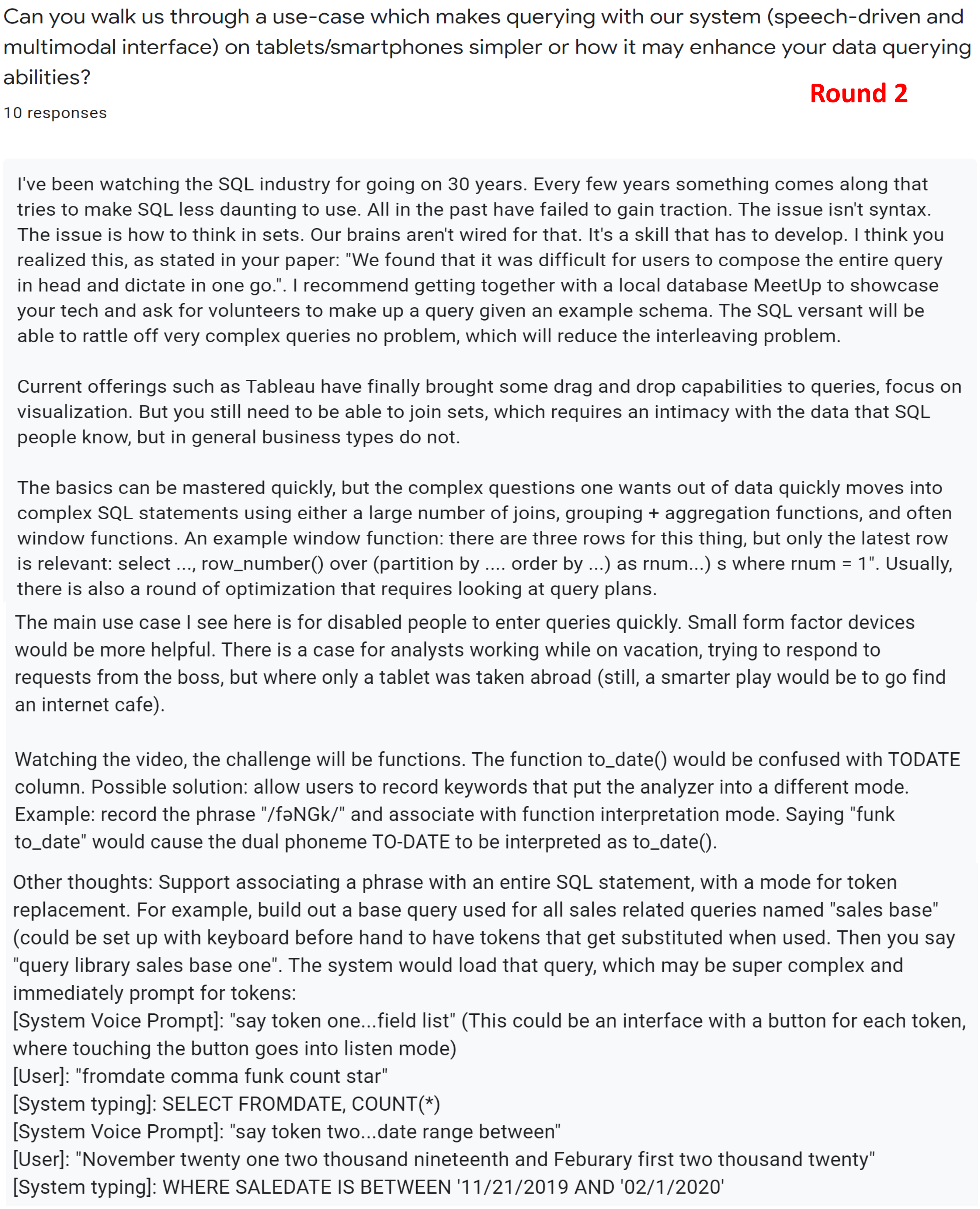
10 responses



(5) What new aspects do you desire to see in SpeakQL as we improve the system (check all options that apply)?

9 responses





Can you walk us through a use-case which makes querying with our system (speech-driven and multimodal interface) on tablets/smartphones simpler or how it may enhance your data querying abilities?

**Round 2**

10 responses

I've been watching the SQL industry for going on 30 years. Every few years something comes along that tries to make SQL less daunting to use. All in the past have failed to gain traction. The issue isn't syntax. The issue is how to think in sets. Our brains aren't wired for that. It's a skill that has to develop. I think you realized this, as stated in your paper: "We found that it was difficult for users to compose the entire query in head and dictate in one go.". I recommend getting together with a local database MeetUp to showcase your tech and ask for volunteers to make up a query given an example schema. The SQL versant will be able to rattle off very complex queries no problem, which will reduce the interleaving problem.

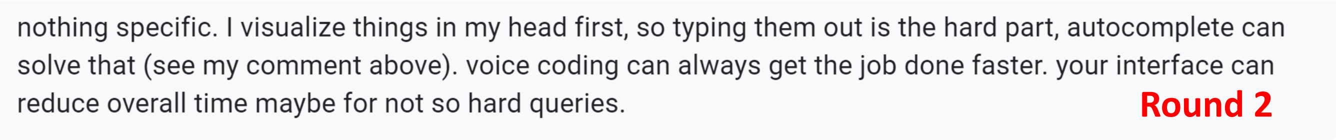
Current offerings such as Tableau have finally brought some drag and drop capabilities to queries, focus on visualization. But you still need to be able to join sets, which requires an intimacy with the data that SQL people know, but in general business types do not.

The basics can be mastered quickly, but the complex questions one wants out of data quickly moves into complex SQL statements using either a large number of joins, grouping + aggregation functions, and often window functions. An example window function: there are three rows for this thing, but only the latest row is relevant: select ..., row_number() over (partition by .... order by ...) as rnum...) s where rnum = 1". Usually, there is also a round of optimization that requires looking at query plans.

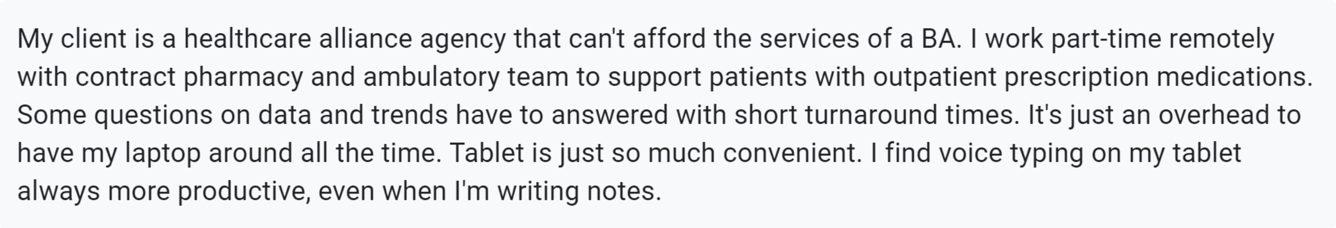
The main use case I see here is for disabled people to enter queries quickly. Small form factor devices would be more helpful. There is a case for analysts working while on vacation, trying to respond to requests from the boss, but where only a tablet was taken abroad (still, a smarter play would be to go find an internet cafe).

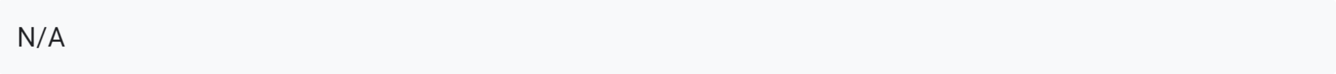
Watching the video, the challenge will be functions. The function to_date() would be confused with TODATE column. Possible solution: allow users to record keywords that put the analyzer into a different mode. Example: record the phrase "/fəNGk/" and associate with function interpretation mode. Saying "funk to_date" would cause the dual phoneme TO-DATE to be interpreted as to_date().

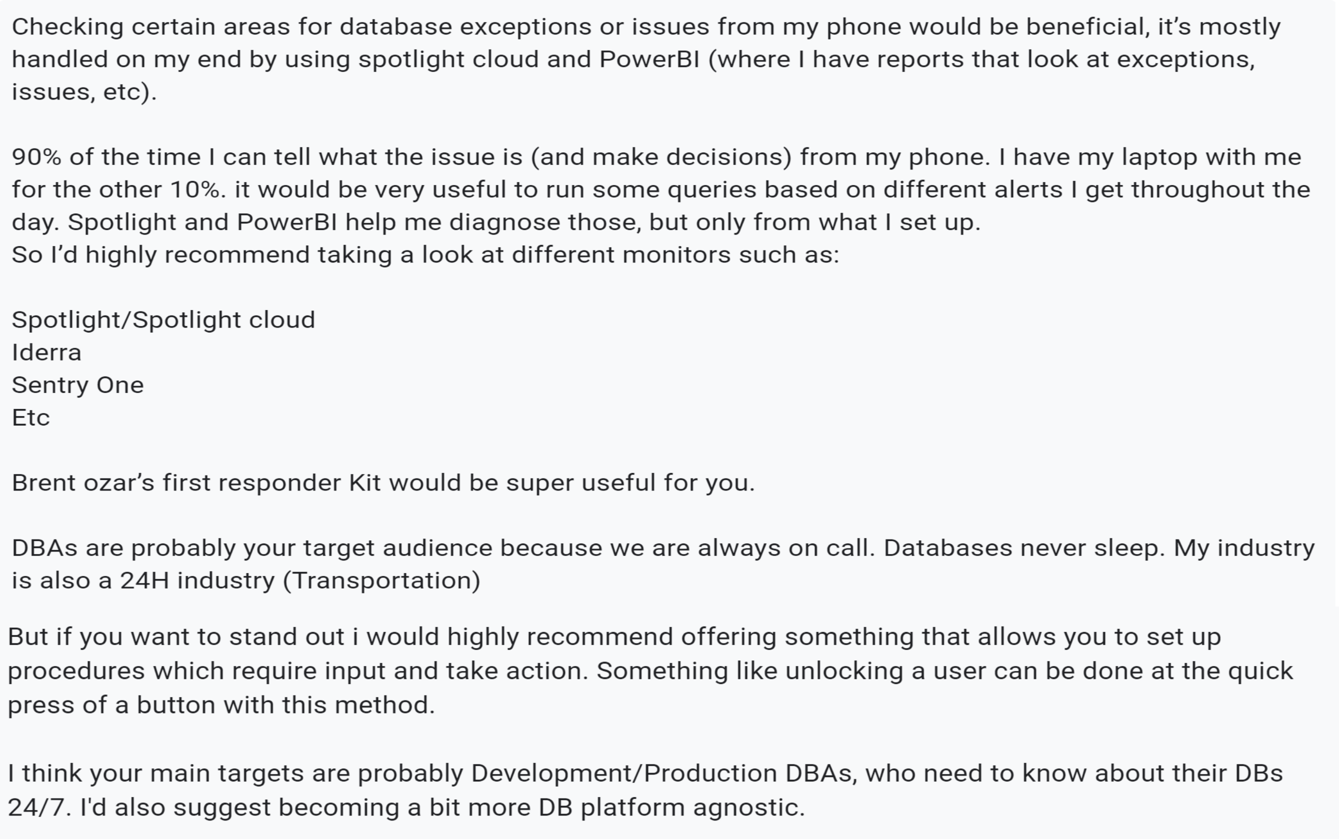
Other thoughts: Support associating a phrase with an entire SQL statement, with a mode for token replacement. For example, build out a base query used for all sales related queries named "sales base" (could be set up with keyboard before hand to have tokens that get substituted when used. Then you say "query library sales base one". The system would load that query, which may be super complex and immediately prompt for tokens:
[System Voice Prompt]: "say token one...field list" (This could be an interface with a button for each token, where touching the button goes into listen mode)
[User]: "fromdate comma funk count star"
[System typing]: SELECT FROMDATE, COUNT(*)
[System Voice Prompt]: "say token two...date range between"
[User]: "November twenty one two thousand nineteenth and Feburary first two thousand twenty"
[System typing]: WHERE SALEDATE IS BETWEEN '11/21/2019 AND '02/1/2020'


nothing specific. I visualize things in my head first, so typing them out is the hard part, autocomplete can solve that (see my comment above). voice coding can always get the job done faster. your interface can reduce overall time maybe for not so hard queries.

**Round 2**


My client is a healthcare alliance agency that can't afford the services of a BA. I work part-time remotely with contract pharmacy and ambulatory team to support patients with outpatient prescription medications. Some questions on data and trends have to answered with short turnaround times. It's just an overhead to have my laptop around all the time. Tablet is just so much convenient. I find voice typing on my tablet always more productive, even when I'm writing notes.


N/A


Checking certain areas for database exceptions or issues from my phone would be beneficial, it’s mostly handled on my end by using spotlight cloud and PowerBI (where I have reports that look at exceptions, issues, etc).

90% of the time I can tell what the issue is (and make decisions) from my phone. I have my laptop with me for the other 10%. it would be very useful to run some queries based on different alerts I get throughout the day. Spotlight and PowerBI help me diagnose those, but only from what I set up.
So I’d highly recommend taking a look at different monitors such as:

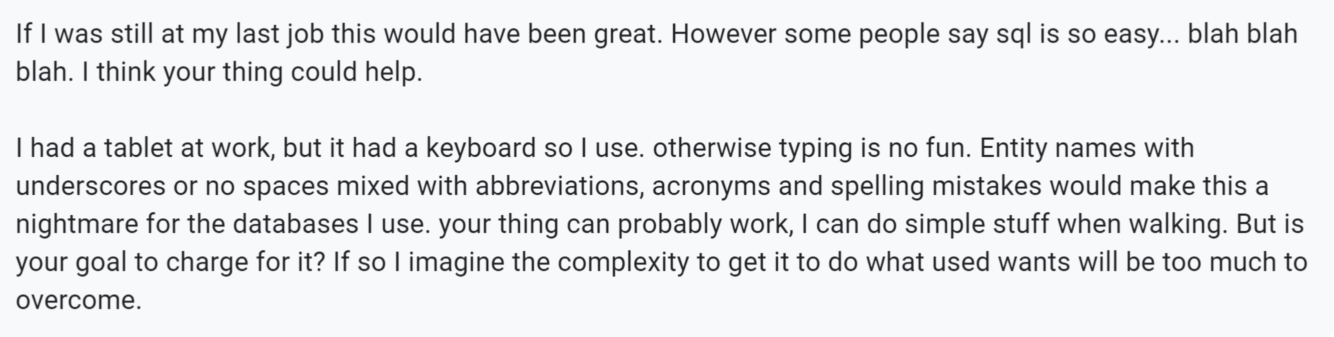
Spotlight/Spotlight cloud
Iderra
Sentry One
Etc

Brent ozar’s first responder Kit would be super useful for you.

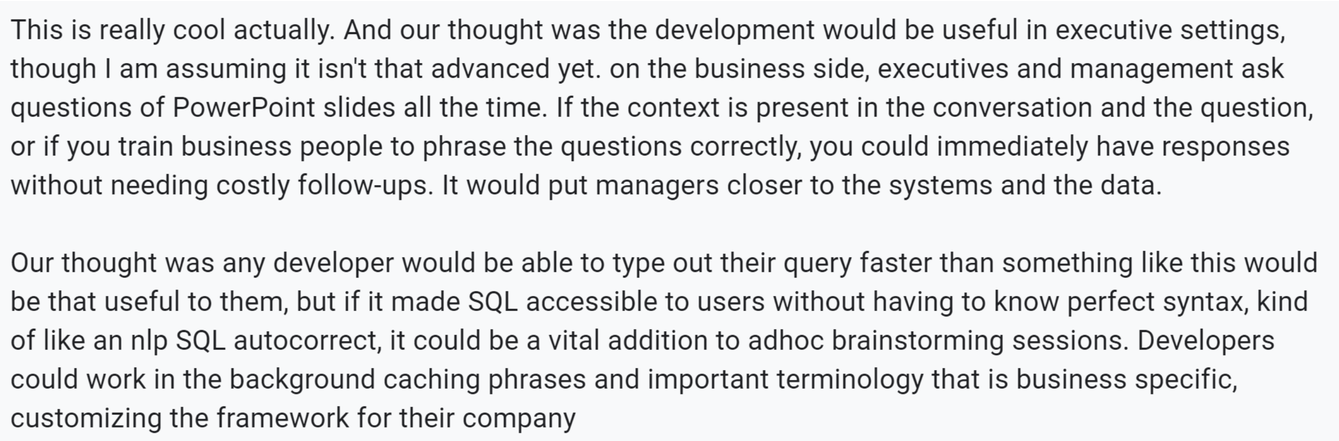
DBAs are probably your target audience because we are always on call. Databases never sleep. My industry is also a 24H industry (Transportation)

But if you want to stand out i would highly recommend offering something that allows you to set up procedures which require input and take action. Something like unlocking a user can be done at the quick press of a button with this method.

I think your main targets are probably Development/Production DBAs, who need to know about their DBs 24/7. I'd also suggest becoming a bit more DB platform agnostic.

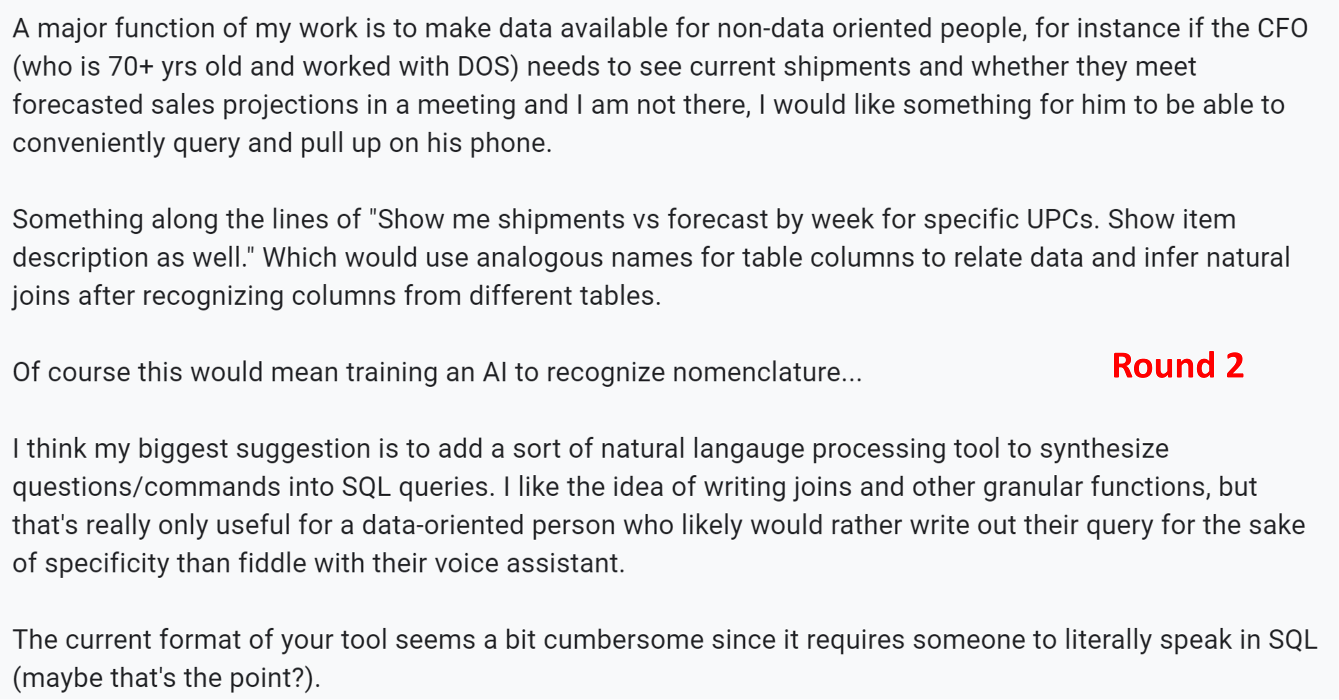

If I was still at my last job this would have been great. However some people say sql is so easy... blah blah blah. I think your thing could help.

I had a tablet at work, but it had a keyboard so I use. otherwise typing is no fun. Entity names with underscores or no spaces mixed with abbreviations, acronyms and spelling mistakes would make this a nightmare for the databases I use. your thing can probably work, I can do simple stuff when walking. But is your goal to charge for it? If so I imagine the complexity to get it to do what used wants will be too much to overcome.



This is really cool actually. And our thought was the development would be useful in executive settings, though I am assuming it isn't that advanced yet. on the business side, executives and management ask questions of PowerPoint slides all the time. If the context is present in the conversation and the question, or if you train business people to phrase the questions correctly, you could immediately have responses without needing costly follow-ups. It would put managers closer to the systems and the data.

Our thought was any developer would be able to type out their query faster than something like this would be that useful to them, but if it made SQL accessible to users without having to know perfect syntax, kind of like an nlp SQL autocorrect, it could be a vital addition to adhoc brainstorming sessions. Developers could work in the background caching phrases and important terminology that is business specific, customizing the framework for their company



A major function of my work is to make data available for non-data oriented people, for instance if the CFO (who is 70+ yrs old and worked with DOS) needs to see current shipments and whether they meet forecasted sales projections in a meeting and I am not there, I would like something for him to be able to conveniently query and pull up on his phone.

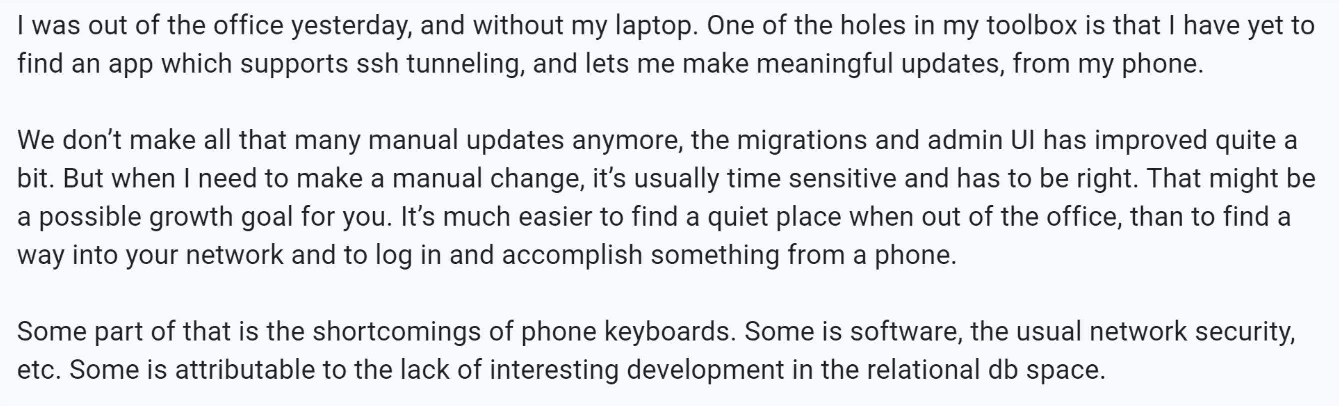
Something along the lines of "Show me shipments vs forecast by week for specific UPCs. Show item description as well." Which would use analogous names for table columns to relate data and infer natural joins after recognizing columns from different tables.

Of course this would mean training an AI to recognize nomenclature...

**Round 2**

I think my biggest suggestion is to add a sort of natural langauge processing tool to synthesize questions/commands into SQL queries. I like the idea of writing joins and other granular functions, but that's really only useful for a data-oriented person who likely would rather write out their query for the sake of specificity than fiddle with their voice assistant.

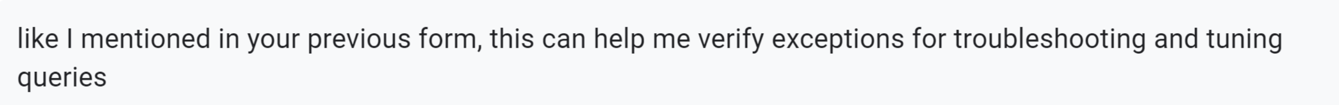
The current format of your tool seems a bit cumbersome since it requires someone to literally speak in SQL (maybe that's the point?).



I was out of the office yesterday, and without my laptop. One of the holes in my toolbox is that I have yet to find an app which supports ssh tunneling, and lets me make meaningful updates, from my phone.

We don’t make all that many manual updates anymore, the migrations and admin UI has improved quite a bit. But when I need to make a manual change, it’s usually time sensitive and has to be right. That might be a possible growth goal for you. It’s much easier to find a quiet place when out of the office, than to find a way into your network and to log in and accomplish something from a phone.

Some part of that is the shortcomings of phone keyboards. Some is software, the usual network security, etc. Some is attributable to the lack of interesting development in the relational db space.



like I mentioned in your previous form, this can help me verify exceptions for troubleshooting and tuning queries