SpeakQL: Towards Speech-driven Multi-modal Querying

Dharmil Chandarana Vraj Shah Arun Kumar Lawrence Saul
University of California, San Diego
{dharmil,vps002,arunkk,saul}@eng.ucsd.edu

ABSTRACT

Natural language and touch-based interfaces are making data querying significantly easier. But typed SQL remains the gold standard for query sophistication although it is painful in many querying environments. Recent advancements in automatic speech recognition raise the tantalizing possibility of bridging this gap by enabling spoken SQL queries. In this work, we outline our vision of one such new query interface and system for regular SQL that is primarily speech-driven. We propose an end-to-end architecture for making spoken SQL querying effective and efficient and present initial empirical results to understand the feasibility of such an approach. We identify several open research questions and propose alternative solutions that we plan to explore.

1 INTRODUCTION

In the last few years, thanks to the advent of deep neural networks, large training datasets, and massive compute resources [7], automatic speech recognition (ASR) is beginning to match (in some cases, even surpass) human-level accuracy in some domains [4]. Naturally, the popularity of speech-based inputs is rising rapidly in several applications where typing was the primary mode of input, including text messaging, emails, and Web search. ASR is also a key enabler of new applications such as conversational personal assistants, e.g. Siri, Alexa, Cortana, and Google Home.

Our community has long studied low-barrier query interfaces to obviate the need to type SQL; they fall into two main categories. The first provide visual (both tabular [15] and drag-and-drop [3]) or touch (both gestural [12] and pen-based [6]) interfaces. The second provides a natural language interface (NLI), either typed [9] or bidirectional conversations [10]. Almost of of them translate under the covers to SQL but at the user level, they eliminate "SQL" from "type SQL" or both. But conspicuous by its absence is a robust speech-based interface for regular SQL that exploits modern ASR, i.e., one that eliminates only "type" from "type SQL." Building such an interface is the focus of this work.

At first blush, one might wonder, why dictate SQL? Why not just use NLIs or touch-based interfaces? From a research standpoint, the very motivation of SQL was to have a structured *English* query language, i.e., a *constrained* NLI, to enable non-CS users to perform sophisticated data querying. Thus, any understanding of ASR's benefits for data querying is incomplete without a proper understanding of its interplay with SQL. From a practical standpoint, many important users, including in the C-suite, in enterprise, Web, healthcare, and other domains are *already familiar* with SQL (even if only a subset of it) and use it routinely! A spoken SQL interface could help them speed up query specification, especially in constrained settings such as smartphones and tablets, where typing SQL would be painful but speech-driven applications are common. More fundamentally, there is a trade-off inherent in any new query interface: *how easy is it to use* vs. *how much query sophistication can it support well*, as illustrated in Figure 1. SQL remains the gold standard for query sophistication. While complex NLIs might supplant SQL in the future, they are beholden to the "AI-hard" natural language understanding (NLU) problem. For these reasons, we argue that more research is needed on exploiting ASR to make it easier to specify SQL or SQL-like queries, not just eliminating SQL from data query interfaces.

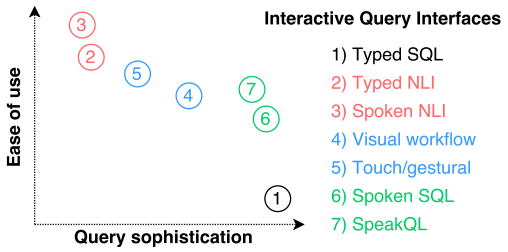

Figure 1: A qualitative comparison of the trade-off between ease of use and *currently* feasible query sophistication for different kinds of interactive query interfaces (top right is best). SpeakQL aims for almost full SQL sophistication, while improving ease of use using both speech and touch.

Relationship to Prior Work. There is some prior work on using ASR for data querying. The US military has explored the use of Dragon NaturallySpeaking for querying document databases [8]. Nuance has a healthcare-focused ASR engine integrated with some commercial RDBMSs [2]. But to the best of our knowledge, there is no general-purpose *open domain* spoken SQL query interface. The recent system EchoQuery is a conversational NLI designed as an Alexa skill [10]. While it could be useful for layman users, cascading of errors caused by NLU and ASR could restrict the query sophistication of such an approach. Moreover, the low information density of speech makes it an impractical mechanism to return large query results. Also, a recent user study by Baidu showed that even for simple text messaging, most users prefer using speech only for the first dictation; for error correction and refinement, they prefer touch [14]. Nevertheless, our work can be seen as an alternative approach within the "query by voice" paradigm mentioned in [10].

In contrast to the prior approaches, we aim to build an *open domain*, speech-driven, and *multi-modal* interface for regular SQL.

HILDA'17, Chicago, IL, USA

978-1-4503-5029-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3077257.3077264

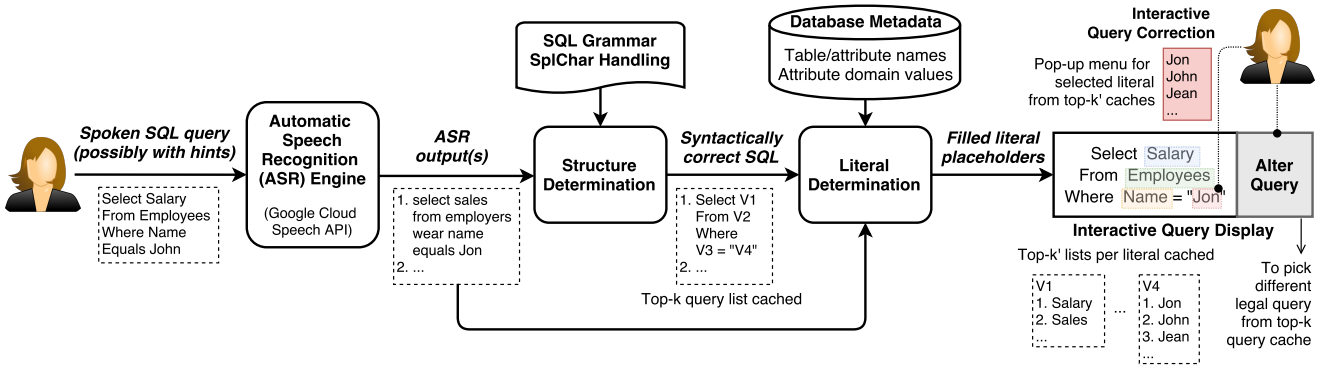

Figure 2: End-to-end Architecture of SpeakQL 1.0. For illustration purposes, we show how a simple spoken SQL query eventually gets converted to a query displayed on a screen, which the user can interactively refine.

We would like to enable users to specify an SQL query with speech but perform interactive query correction using a screen-based touch (or click) interface, with the query results displayed on screen as well. We call our interface and system, SpeakQL.

In the rest of this paper, we present our vision for SpeakQL, including our desiderata and an initial architecture that uses a cloud ASR service. We discuss our evaluation methodology and present an initial empirical study that reveals the key differences between general English recognition and SQL recognition. Our results reveal several challenging research questions and we consider alternative approaches to solving them by combining ideas from database systems, information retrieval (IR), linguistics, and applied ML.

2 PRELIMINARIES

2.1 Desiderata and Technical Challenge

Complementary to prior work on NLIs, visual, touch-based, and gestural interfaces, our desiderata for a new speech-driven multimodal query interface are as follows:

(1) Supports regular SQL with a given grammar.
(2) Exploits state of the art ASR for high accuracy.
(3) Open domain, i.e., supports queries in any domain with a potentially *infinite* vocabulary.
(4) Supports touch-based or click-based interactive query correction with a screen display.

Our desiderata are motivated by the unique way in which SQL differs from regular English speech: it is both *less* and *more* general! SQL is less general due to its unambiguous context free grammar (CFG), which could make syntactic analysis and parsing easier than regular English speech. But SQL is also more general because non-vocabulary tokens (from an ASR perspective) are far more likely in SQL due to the infinite variety of database instances across domains. For instance, it is unlikely that any ASR engine can exactly recognize a literal like `CUSTID_1729A`. We call this the *open domain* problem; addressing this problem is a core technical challenge in making speech-driven SQL effective. We believe human-in-the-loop query correction might be a necessary component of achieving this goal. Note that the open domain problem has not been solved even for spoken NLIs such as [10]. Thus, we believe some of the techniques developed as part of this work could potentially benefit spoken NLIs as well.

2.2 Architecture of SpeakQL 1.0

In our first design, we want to understand how good a modern off-the-shelf ASR tool is for our task and how to exploit it. So, we "outsource" the ASR part and focus on SQL-specific issues. To expand on an earlier observation, SQL contains only three kinds of tokens: English *keywords*, *special characters* ("SplChar"), and *literals* (table names, attribute names, and attribute values). Keywords and SplChars are from a small vocabulary present in the SQL grammar, while literals are from a potentially infinite vocabulary. But we observe that in practice, literals in SQL are typically from the set of values already present in the database instance being queried or a general numeric value. Exploiting our observations, we envision a four-component architecture, as Figure 2 shows. We briefly describe each component's role and their current baseline implementation.

ASR Wrapper. This component records the spoken SQL query as an audio file, invokes a modern ASR tool, and obtains the top ASR transcription outputs.

Current Implementation. We send the audio to Google's Cloud Speech API [1] and obtain a ranked list of outputs. Google's API also offers two interesting and useful options: *accents* and *hints*. We used the "en-US" option for the accent. 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. Empirically, we find that hints help improve baseline ASR accuracy and thus, decrease the downstream work.

Structure and Literal Determination. The structure determination component post-processes the ASR output(s) to determine a ranked list of syntactically correct SQL statements with placeholder variables for literals. It exploits the given SQL grammar and a set of SplChar handling rules (their textual and phonetic representations) to correct the query structure and fix keywords and SplChars. The literal determination component processes the syntactically correct SQL statements and "fills in the blanks" for the literal placeholders. It uses the raw ASR outputs as a surrogate for what was spoken for the literals. The open domain issue is addressed by pre-computing a set of "materialized views" that provide the table names, attribute names and types, and the domains of the attributes (sets of unique values, tokens in textual attributes,

and numeric ranges) from the database schema and instance being queried. This component then uses the raw ASR outpus and "joins" them with relevant metadata to obtain a ranked list of literals for each placeholder. *Decoupling* structure determination from literal determination is a crucial design decision that helps us attack the open domain problem. This is because correcting the syntactic structure is a relatively easier problem for which existing natural language processing (NLP) techniques can be applied, while effective literal determination is harder.

Current Implementation. We have built a simple baseline module using rule-based heuristics to correct SQL queries. Essentially, we create a dictionary of all keywords, SplChars, and schema literals (table and attribute names). We then match each token in the transcription output with a dictionary entry based on shortest edit distance on strings. We also create another baseline that augments this dictionary with instance literals (domain values of attributes). The intuition is that the larger dictionary could help resolve errors in literals that arise in predicates. Clearly, this naive dictionary-based approach has several issues, including not exploiting the SQL grammar and poor scalability of dictionary lookups. But it serves as the most "braindead" baseline; we explore more sophisticated approaches to mitigate both of these issues in Section 4.

Interactive Display. This is the component that the user interacts with. It displays a single SQL statement that represents the "best" transcription of the spoken SQL query and provides interactive touch/click-based mechanisms for query correction that are inspired by touch-based text messaging apps. Literals are highlighted and boxed, with the respective ranked lists cached by our system. If a displayed literal is incorrect, the user can touch its box and a pop-up menu will display the ranked lists of alternatives for that placeholder. In Figure 2, the user corrects "Jon" to "John." If the structure of the query itself is wrong, the user can select the "Alter Query" button to obtain a larger menu with the ranked list of alternative query structures. In the worst case, if our system fails to identify the correct query structure and/or literals, the user can delete and type into the query display box to correct one token, multiple tokens, or the whole query.

Current Implementation. In our preliminary implementation, we use a simple terminal-based front-end for displaying the correction results. We plan to develop a more sophisticated interface as outlined above in due course.

3 INITIAL EMPIRICAL STUDY

We start with an initial empirical study using our baseline implementation. This study will help us set up the workloads and an end-to-end evaluation infrastructure.

3.1 Experimental Setup

Since there are no publicly available datasets for spoken SQL recognition, we create our own dataset. We will shortly describe the database and query set but first, we explain our overall workflow.

(1) Recording Spoken SQL: To simulate a real-world scenario, we recorded the SQL queries on a popular smartphone (iPhone 6S). The audio clips were of varying lengths, between 4 to 24 seconds at a sampling rate of 48,000 Hz.

(2) ASR Transcription: The audio files are encoded in Base64 format, as required by Google's API, and sent as JSON requests with the language option "en-US", top 5 as the number of outputs, and the hints as mentioned earlier. The API returns the hypothesis text with top 5 alternatives and their confidence scores.

(3) Query Correction: We analyze the transcribed results using our baseline implementation. The details of our methodology and metrics will be discussed shortly.

Database and Query Set. We used two publicly available database schemas: the Employees Sample Database from MySQL[1] and the Northwind database schema from northwinddatabase.codeplex.com. We created a set of 25 SQL queries in the MySQL dialect; they are listed in Table 1. All queries are SELECT statements with no nesting; we focus on SELECT statements in this initial work, since they are perhaps the most common form of queries, and to improve tractability (although our architecture is applicable to any SQL query in general). The queries are designed to be of increasing difficulty for a speech-based querying engine. Thus, these queries have different fractions of keywords, SplChars, and literals, with even the literals being increasingly sophisticated.

ASR Hints. We provided all of the SQL keywords and SplChars that arise in our queries as hints to Google's API. As we show later, hints help improve baseline accuracy significantly. The following are the hints we provided:

- **Keywords**: Select, From, Where, And, Or, Order By, Limit, Between, Group By, In, Sum, Count, Max, Avg
- **SplChars**: = * < > ( ) % . , _

We post-process the ASR results and use different error metrics for evaluation. For brevity sake, we list only the most likely (top 1) transcription per query in Table 2.

3.2 Error Metrics

We explain why a standard ASR metric is not enough for SQL queries and then propose new SQL-oriented metrics.

Word Error Rate (WER). WER is the most common error metric in ASR [13]; it is the edit distance between the *reference text* (ground truth SQL query) and the *hypothesis text* (transcription output). Formally, given the number of "words" n in the reference text, number of substitutions s, number of deletions d, and number of insertions i, $WER = (s + d + i)/n$. Since SQL has three kinds of tokens (keywords, SplChars, and literals), all should be treated as words. Literals with spaces are treated as multiple tokens.[2]

WER does not capture important peculiarities of SQL that are crucial for query correction, especially the separation of structure and literal determination. Moreover, it is not always clear what a "word" is in an SQL query. For instance, consider the literal *oid_73fc* from Q21 in Table 1. Is it one word or eight words ({o,i,d,_,7,3,f,c})? From ASR's perspective, it is the latter. But from an SQL parser's perspective, it is the former! Thus, we make the evaluation process SQL-aware by proposing eight additional fine-grained metrics that intuitively separate the concerns of structure and literal determination. We start with some notation.

[1] dev.mysql.com/doc/employee/en. We transformed the attribute names to be more easily pronouncable.

[2] For the user's convenience, we omit quotation marks during dictation; so, they are excluded in the error metrics too.

ID	Query
Q01	SELECT * FROM Departments
Q02	SELECT * FROM Employees WHERE FirstName="Adam"
Q03	SELECT * FROM Employees WHERE FirstName="Mary"
Q04	SELECT * FROM Employees WHERE FirstName="Mary" AND Gender="M"
Q05	SELECT BirthDate, JoinDate FROM Employees WHERE LastName="Griswold"
Q06	SELECT * FROM Departments where DepartmentName = "Finance"
Q07	SELECT * FROM Employees WHERE salary>150,000
Q08	SELECT EmployeeNumber FROM titles WHERE Title="Senior Engineer"
Q09	SELECT EmployeeNumber FROM DepartmentManager WHERE DepatmentNumber="d001"
Q10	SELECT DepartmentNumber FROM DepartmentManager WHERE EmployeeNumber=110228
Q11	SELECT FirstName, LastName, DepartmentNumber FROM Employees, DepartmentManager WHERE Employees.EmployeeNumber = DepartmentManager.EmployeeNumber
Q12	SELECT FirstName, LastName, DepartmentName FROM Employees, DepartmentManager, Departments WHERE Employees.EmployeeNumber = DepartmentManager.EmployeeNumber AND DepartmentManager.DepartmentNumber = Departments.DepartmentNumber
Q13	SELECT * FROM Employees WHERE FirstName="Adam" ORDER BY LastName
Q14	SELECT JoinDate FROM Employees WHERE FirstName="Domenick" OR FirstName="Bojan" LIMIT 10
Q15	SELECT AVG(Salary) FROM Employees,Salaries WHERE Employees.FirstName="Patricio" AND Employees.EmployeeNumber = Salaries.EmployeeNumber
Q16	SELECT COUNT(*) FROM Salaries WHERE Salary BETWEEN 70,000 AND 80,000
Q17	SELECT Title, SUM(Salary) FROM Salaries, Titles WHERE Salaries.EmployeeNumber = Titles.EmployeeNumber GROUP BY Title
Q18	SELECT * FROM Employees WHERE BirthDate = "1953-09-02"
Q19	SELECT * FROM DepartmentEmployees WHERE DepartmentNumber IN(d001,d002)
Q20	SELECT * FROM Employees WHERE BirthDate LIKE "1953%"
Q21	SELECT Order_Dt, Shipped_Dt FROM Orders WHERE OrderID="oid_73fc"
Q22	SELECT Supp_ID FROM Products WHERE ProductName="Belkin BE44-80 Surge Protector"
Q23	SELECT MAX(QuantityPerUnit), ProductName FROM Products WHERE UnitPrice=4.99
Q24	SELECT ShipVia, Freight FROM Orders, Customers WHERE Customer.Region="us_west_1"
Q25	SELECT ContactTitle, ContactName FROM Customers WHERE Address="1035 Niguel Ln"

Table 1: The ground truth for the spoken SQL queries.

Given a query text, we tokenize it to obtain a multiset of tokens; let A denote this multiset for the reference SQL query text and B, for a given a hypothesis query text (with or without correction). Tokens that are not textually identical are treated as different tokens. For example, "=" and "equals" are treated as different tokens as are "department" and "departments." While this might appear harsh, we think it is necessary because an SQL parser might report an error otherwise, which has to be fixed by our system or the user. We define A_{KW}, A_{SC}, and A_{LI} as the subsets of A corresponding to keywords, SplChars, and literals respectively. Subsets of B are defined similarly. Keywords and SplChars are detected using the vocabulary of the SQL grammar; all other tokens are treated as literals. We are now ready to define our new metrics.

(1) Keyword Recall Rate. $KRR = \frac{|A_{KW} \cap B_{KW}|}{|A_{KW}|}$
(2) SplChar Recall Rate. $SRR = \frac{|A_{SC} \cap B_{SC}|}{|A_{KW}|}$
(3) Literal Recall Rate. $LRR = \frac{|A_{LI} \cap B_{LI}|}{|A_{LI}|}$
(4) Word Recall Rate. $WRR = \frac{|A \cap B|}{|A|}$

Similarly, we define four precision-oriented metrics: Word Precision Rate, $WPR = \frac{|A \cap B|}{|B|}$, Keyword Precision Rate, and so on. Note that while these new metrics delineate the accuracy of recognizing the different kinds of tokens, they do not capture errors in the ordering of the tokens.

3.3 Results

For brevity sake, we only report top 1 results (we briefly discuss top 5 results later). We compare "before correction" accuracy against two naive dictionary-based correction baselines: a "small" dictionary with only keywords, SplChars, and schema literals, and a "large" dictionary that also includes instance literals. Figure 3 plots the cumulative distribution functions of the error metrics for the queries in Table 2. Table 3 reports the mean values of the same.

We see that the recall rates are already quite high for keywords and SplChars, even without correction. This is not surprising because they were provided as hints. For literals, however, the recall rates are quite low (mean of 0.43). The dictionary-based correction schemes lead to only marginal improvements (mean of 0.47). Thus, the overall WRR is also low, primarily because of literals. The precision results present an interesting contrast: the overall literal precision rates go up significantly but the larger dictionary performs counterintuitively *worse* than the smaller one! Even more surprisingly, keyword precision rates go *down* after correction! We confirmed that this is because the dictionary-based methods introduce spurious keywords and literals based on the edit distances. This underscores the need for more sophisticated similarity search approaches (more in Section 4). Overall, these initial results validate our earlier claim: the open domain problem is a key bottleneck for spoken SQL recognition (and this issue will remain for spoken NLIs too). By delineating structure and literal determination, our work lays out a promising architecture and framework to systematically improve spoken SQL recognition and data querying.

Importance of Hints. We now turn off hints to assess how much they helped before correction. Table 4 lists the mean error metrics with and without hints. For additional insights, we present the results for both top 1 outputs and "best of" top 5 outputs. We see that hints improve both recall and precision rates for keywords and SplChars (especially the latter). LRR, as expected, is not affected significantly. Thus, it is beneficial to exploit hints, if possible. As

ID	Query
Q01	select * from Department
Q02	select * from employees their first name = Adam
Q03	select * from employees their first name = Mary
Q04	select * from employees where first name = Mary and gender = m
Q05	select birth date, you ended from employees where last name = driftwood
Q06	select * from department where Department name = Finance
Q07	select * from employees where salary greater than 150000
Q08	select employee number from titles where title = senior engineer
Q09	select employee number from department manager where department number = LY001
Q10	select department number from department managers where employee number = 1 10228
Q11	select first name, last name, and apartment number from employees, department manager where employees not employ a number = department manager. Employee number
Q12	select first name, last name, Department name from employees, department managers, department where employees. Employee number = department manager. Employee number and department manager. Department number department store Department number
Q13	select * from employees where first name = Adam order by last name
Q14	select join date from employees where first name = Dominic or first name = version limit 10
Q15	select AVG (salary) from employees, salaries where employees. First name = by Tricia and employees. Employee number = salaries. Employee number
Q16	let's count (* plus (from Telus where Saturday between 70000 and 80
Q17	select Title, sum (salary growth) from salaries, titles van salaries. Employee number = titles. Employee number Group by title
Q18	select * from employees where birthdate = 1953 - 09 - 02
Q19	select * from Department employees where department number in (bc20 one, d002 plus (
Q20	select * from employees where birthday like 1953 %
Q21	select order _ Duty, ship _ Duty from order where are there any = or ID _ 7 3 f c
Q22	select Su Bebe _ ID from products where product name = Belkin be for 4-8 zero search protector
Q23	select Max (quantity per unit), product name from products that you need price = 4.99
Q24	select Supply, free from orders, customers that customer. Region = u.s. _ vest _ one
Q25	select contact title, contact name from customers where address = 1035 Niguel Ln

Table 2: Top 1 transcription results from the ASR engine (with hints provided).

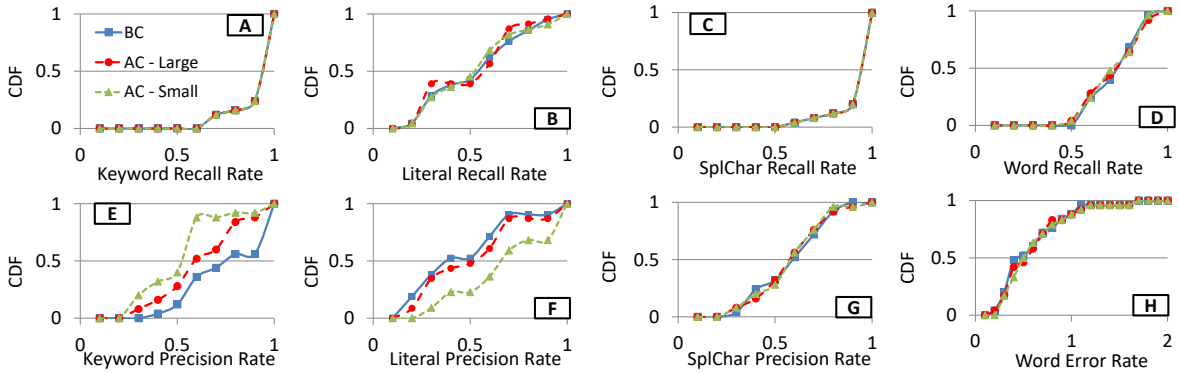

Figure 3: Cumulative distribution of accuracy metrics for the top 1 results from Table 2. Before correction (BC), After Correction with large dictionary (AC - Large) and small Dictionary (AC - Small).

Metric	Top 1			BoTop-5		
	BC	ACS	ACL	BC	ACS	ACL
KRR	0.94	0.94	0.94	0.98	0.98	0.98
SRR	0.94	0.94	0.94	0.97	0.97	0.97
LRR	0.43	0.45	0.47	0.51	0.49	0.50
WRR	0.71	0.71	0.72	0.76	0.75	0.75
KPR	0.76	0.50	0.62	0.78	0.55	0.66
SPR	0.89	0.89	0.89	0.92	0.92	0.92
LPR	0.38	0.59	0.45	0.45	0.60	0.49
WPR	0.58	0.58	0.58	0.62	0.62	0.62
WER	0.58	0.58	0.57	0.53	0.53	0.52

Table 3: Mean error metrics: Before Correction (BC), After Correction using small dictionary (ACS) and using large dictionary (ACL). "BoTop" stands for "Best of Top."

Metric	KRR	SRR	LRR	WRR	WPR	WER
Top 1 wo	0.78	0.44	0.44	0.57	0.46	0.73
BoTop 5 wo	0.79	0.44	0.52	0.60	0.48	0.70
Top 1 w	0.94	0.94	0.43	0.71	0.58	0.58
BoTop 5 w	0.98	0.97	0.51	0.76	0.62	0.53

Table 4: Mean error metrics for (uncorrected) transcribed queries with ("w") and without ("wo") hints.

an interesting aside, even when we turned off the hints, Google's API managed to correctly transcribe "," and "." as symbols for some queries! But in most cases, hints were necessary to avoid low SRR. These results also suggest that providing some literals as hints might help improve LRR. While this is feasible for schema literals, instance literals might become a bottleneck due to their sheer number. We plan to study this issue further in future work.

Latency. In our current baseline implementation, Google's API turned out to be the latency bottleneck. The round-trip times for obtaining transcription results were between 8s to 46s, with a mean of 24s. Our pre- and post-processing took less than 5% of the overall time for most queries when using the small dictionary for correction. While this is clearly not "real time," we expect that Google will improve their cloud API's latency over time. However, we are also exploring more local ASR alternatives (see Section 4). In contrast, when using the large dictionary, the time was roughly equally split between the cloud API round-trip and query correction for most queries. This suggests that it is crucial to design and use better index structures for similarity search to speed up literal determination (see Section 4).

4 RESEARCH QUESTIONS AND PLAN

We now discuss key open research questions and our plan to improve SpeakQL based on our initial empirical findings.

Structure Determination. To make this component effective, we need to be able to parse erroneous SQL statements, while masking out literals. Tackling this challenge requires combining text processing, SQL grammar, and NLP. First, we need to *tag literals* and mask them as terminal variable placeholders. This is a binary classification problem; we plan to explore both rule-based and ML classifiers, say, a conditional random field (CRF). Going further, we need to exploit the SQL grammar (at least its SELECT statement subset for starters). Since directly applying an SQL parser will fail when there are errors in the transcriptions, we plan to adapt the approach of [5] and create a language model that exploits the syntactic structure of SQL. This would likely require augmenting SQL's CFG to devise a probabilistic context free grammar (PCFG) [11] that would let us obtain the most likely parse trees of a given erroneous SQL statement. Eventually, we expect to combine literal tagging with the PCFG for holistic structure determination.

Literal Determination. As our baseline dictionary-based approach showed, we need indexes of materialized views of the given database instance to improve the effectiveness of literal determination. Such indexes should improve both accuracy and runtimes. Tackling this challenge requires combining database indexing, IR, linguistics, and HCI. First, string similarity measures will likely fall short of what we really want: *phonetic* similarity search. We plan to map literals to a standard IPA-based phonetic representation. While IPA dictionaries exist for regular English words, out-of-vocabulary tokens are a challenge, e.g., `CUSTID_1729A` from before. Second, to obtain near-real time latency, our indexes should be optimized in layout and hardware usage for top k similarity queries. We plan to benchmark existing indexes from the database and IR literatures and possibly adapt them. Finally, incorporating user feedback in the top k listings of literal placeholders and re-optimizing the rankings on the fly, say, as the user is typing, could help improve accuracy, while still reducing user effort.

Metrics, Data, and Methodology. To the best of our knowledge, there is almost no prior research on spoken SQL. Thus, many open questions remain even on methodology and evaluation metrics. Neither WER nor our new SQL-oriented quality metrics quantify structural errors precisely. Thus, we plan to devise such new error metrics that exploit the SQL grammar. We also plan to create human efficiency metrics such as the total time to obtain a fully correct query. There are also no known public benchmark labeled datasets for spoken SQL. We plan to create such a labeled dataset but how to do so is itself a non-trivial problem. It is probably cruel to get students to dictate millions of queries. We plan to explore various options and evaluate their trade-offs: crowdsourcing (a la ImageNet for image recognition), semi-synthesis of queries by combining the SQL grammar with human-spoken query fragments, and speech synthesis tools such as Amazon Polly. Finally, once we build SpeakQL 1.0, we plan to conduct a user study to compare the user experience both subjectively (surveys) and objectively (number of clicks, number of edits, time to correct query, etc.)

Towards Fully Local Execution. While Google's speech API is a promising start, the latency is currently unacceptably high. Thus, we plan to explore alternatives that enable fully local execution. One option is Apple's device-local speech recognition API for iOS environments. Another option is to adapt an open-source ASR engine for spoken SQL. While the acoustic model can be retained, we would likely need to plug in a new language model that is tailored to SQL queries. Down the line, we also plan to explore a fully end-to-end spoken SQL recognition engine using a deep recurrent neural network, say, by adapting Baidu's DeepSpeech2 [4]. This is predicated upon the availability of a large training dataset as mentioned in the previous paragraph.

Spoken Predicates and Spoken SQL Dialect. Eventually, we envision supporting spoken SQL over both structured and speech data, say, using the `LIKE` predicate. With local execution, the availability of acoustic features could help support *spoken predicates* that bypass text altogether. We also plan to explore how to make SQL more natural for spoken querying, while still preserving its unambiguous CFG-based sophistication. For example, we could replace some special characters with new intuitive idioms and add new keywords to demarcate complex literals, e.g., `BEGIN/END LITERAL`. By slightly adapting the SQL grammar, we will essentially create a new speech-first SQL dialect!

REFERENCES

[1] Google Cloud Speech API. cloud.google.com/speech.
[2] Nuance MagicSpeech. australia.nuance.com/products/speechmagic/index.htm.
[3] Oracle SQL Developer. oracle.com/technetwork/issue-archive/2008/08-mar/o28sql-100636.html.
[4] D. Amodei et al. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *ICML*, 2016.
[5] C. Chelba and F. Jelinek. Exploiting Syntactic Structure for Language Modeling. In *ACL*, 2008.
[6] A. Crotty et al. Vizdom: Interactive Analytics through Pen and Touch. In *VLDB Demo*, 2014.
[7] G. Hinton et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *Signal Processing Magazine*, 2012.
[8] S. Lajoie et al. Application of Spoken and Natural Language Technologies to Lotus Notes Based Messaging and Communication, 2002. dtic.mil/dtic/tr/fulltext/u2/a402014.pdf.
[9] F. Li et al. Constructing an Interactive Natural Language Interface for Relational Databases. In *VLDB*, 2015.
[10] G. Lyons et al. Making the Case for Query-by-Voice with EchoQuery. In *SIGMOD Demo*, 2016.
[11] T. Matsuzaki et al. Probabilistic CFG with Latent Annotations. In *ACL*, 2005.
[12] A. Nandi et al. Gestural Query Specification. In *VLDB*, 2014.
[13] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition.* Prentice-Hall, Inc., 1993.
[14] S. Ruan et al. Speech Is 3x Faster than Typing for English and Mandarin Text Entry on Mobile Devices. *CoRR*, abs/1608.07323.
[15] M. M. Zloof. Query by Example. In *National Computer Conference and Exposition*, 1975.