Meet Cyrus – The Query by Voice Mobile Assistant for the Tutoring and Formative Assessment of SQL Learners*

Author
Department
University
email

Author
Department
University
email

ABSTRACT

Being declarative, SQL stands a better chance at being the programming language for conceptual computing next to natural language programming. This paper examines the possibility of using SQL as a back-end for natural language database programming. Distinctly from keyword based SQL querying, in our approach, keyword dependence and SQL's table structure constraints are significantly less pronounced. We present a natural language voice query interface for mobile devices, *Cyrus*, to relational databases of arbitrary structure. We show that Cyrus supports a large type of query classes, sufficient for an entry level database class. Cyrus is application independent, allows test database adaptation, and not limited to specific sets of keywords or natural language sentence structures. Its cooperative error reporting is more intuitive, and iOS based mobile platform is also more accessible compared to most contemporary mobile and voice enabled systems.

CCS CONCEPTS

•Human-centered computing →Natural language interfaces; *Graphical user interfaces; Auditory feedback;* HCI theory, concepts and models; Empirical studies in HCI; **•Computing methodologies →Information extraction;** *Speech recognition; Knowledge representation and reasoning;* **•Theory of computation →***Database query languages (principles);*

KEYWORDS

Query by Voice; SQL tutoring; query mapping; formative assessment; self-paced learning; mobile learning system

1 INTRODUCTION

Although declarative programming is arguably more intuitive, students still find the transition from imperative languages to SQL hugely difficult. They often struggle to form complex queries that are rich in semantic nuances, especially those involving nested subqueries or GROUP BY functions [2]. To help students learn SQL better, researchers have been trying to develop various teaching systems and learning strategies, and support them with powerful online [12] and desktop learning tools [3]. Despite substantial recognition of its importance, research in developing teaching tools for SQL, especially smart tools have been scant [11, 12]. Most of the contemporary systems still focus on teaching syntactic aspects of SQL and do not go far in teaching the semantic and conceptual underpinnings of declarative programming using SQL with the few exceptions of the automated SQL exercise grading [6] and guided teaching [8] systems.

Recent introduction of voice services such as Amazon Alexa, Apple Siri and Google Assistant leveraged extensive research in understanding and mapping natural language queries (NLQ) to SQL [9] to substantially simplify access to data and to help users who otherwise would not use the vast amount of knowledge to improve their lives [13]. Internet of things (IoT) and smart devices research are also encouraging people to use various digital home services and mobile phone applications using voice recognition and speech processing for automated information gathering from databases. These successful voice technologies remain largely unexplored in educational systems for teaching SQL to first-time learners. We believe that integrating voice or natural language based query capabilities in online systems will make learning platforms powerful. If such a system can be coupled with automated tutoring and grading functions, it can serve as a smart and comprehensive learning environment for SQL.

1.1 Voice Assistant as a Teaching Aid

In the recent years, there has been a significant shift in the size, technology, speed and cost of hardware devices for the better. Today, the ubiquity of smart and mobile devices with large memories offers us the opportunity to rethink how educational aids may be designed for the new century. It is estimated that there are about 700 million Apple iPhones in use worldwide and this number could reach to 1 billion in a few years. With the unprecedented accessibility afforded by such devices, it is important to consider their large-scale applications in education, in particular, since a large number of them are in the hands of students. It is thus natural to explore the opportunity to help students learn using mobile devices, and especially using effortless, convenient and ad hoc voice technology to support uninhibited exploration.

A substantial segment of users are also already familiar with voice-interfaces and with prominent virtual assistants such as Siri. It is thus conceivable that a voice enabled SQL tutoring and assessment tool along the lines of the systems such as PhotoMAT [14] or [4] could support anytime online learning in a hands free manner,

*Research supported in part by a STEM Center grant, and a National Science Foundation grant DRL YYYYYYY.

ACM-ITiCSE '18, July 2–4, 2018, Larnaca, Cyprus.

ISBN 978-1-4503-4722-8/17/08…$15.00
DOI: http://dx.doi.org/10.1145/XXXXXXX.YYYYYYY

and with smart response read out options, students could self-assess their SQL composition abilities over known test databases. Naturally, platforms such as iOS with preexisting APIs for NLP, speech-to-text, and speech synthesis are ideal due to their first-party support, large user-base, and familiarity of voice-interface systems. With the combination of widespread device availability, inexpensive hardware cost (relative to specialized hardware), and stable and heavily field-tested firmware, it is a prudent choice to utilize this platform to implement such a system.

1.2 Advantages of a Voice Assistant

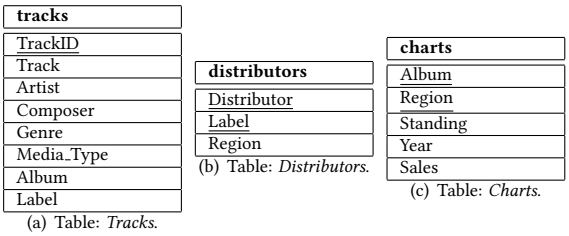
While the usefulness of voice interactions for interfacing with applications and databases are somewhat well understood, its use as a teaching aid requires some justifications mainly because not many research have used this technology for teaching systems. It is argued that voice interfacing to databases in general provides three main advantages [10] – hands-free access, personalizable vocabulary and dialogue-based querying. Given a set of learning objectives of SQL query constructs and classes, all a student possibly wants is to see whether the system generated SQL expression for a query in English matches with his mental formulation of the query in SQL, and produces identical response. From this standpoint, hands-free access to database querying engines and personalizable vocabulary certainly could play major roles in a mobile SQL tutoring and assessment system, which are the major focus of this research.

2 RELATED RESEARCH

Although text interfaces to databases [9] have been explored relatively more than voice interfaces [7], their use for teaching SQL is less explored [12]. While we can point to SQL teaching systems such as [3, 6, 8, 11, 12], we are unable to discuss them in this paper for the want of space except EchoQuery [10] as this framework has much in common with our approach in Cyrus. In EchoQuery, users are able to communicate with the database using voice at any time and queries can be asked incrementally, in steps, in a context of prior queries and stateful dialogue-based refinement are supported along with clarification if queries are incomplete or ambiguous. Finally, EchoQuery allows for a personalized vocabulary on a per-user basis, which makes the system robust. Although Cyrus is not as adept at continuous querying, Cyrus does have a more vigorous translation system for matching user queries over heterogeneous schemes and handle numbers in NLQ better. Cyrus can also map column/table names that span multiple words with special delimiters such as "Track ID" to "Track_id" or "TrackId" by maintaining a history stack with context.

3 CYRUS USER INTERFACE

Cyrus is a smart tutoring and assessment system for SQL designed on the mobile iOS platform. The voice enabled interface translates natural English language queries on a test database into executable SQL queries. A successful translation shows the translated query and the computed results for the student to review and validate her mental model of the query. Since Cyrus generates editable SQL queries that can be executed optionally, students have the opportunity to master the language through iterative refinement. As a formative assessor or grader, Cyrus also compares system mapped queries with students' translated queries over a test database. In the role of a grader, to be deemed correct, a student SQL query must compute identical views as does the system generated query. The test queries are crafted by an instructor as database assignments with escalating difficulty levels. In the remainder of this paper, we use a music database as shown (with the primary keys underlined for each table) in figure 1 to exemplify the features of Cyrus and for other illustrative purposes. While we sketch how Cyrus maps NLQ to SQL in section 4.3 and use examples to discuss the functionalities of Cyrus on intuitive grounds, we defer the full treatment of the NLQ to SQL mapping algorithm to an extended paper.



(a) Table: *Tracks*. (b) Table: *Distributors*. (c) Table: *Charts*.

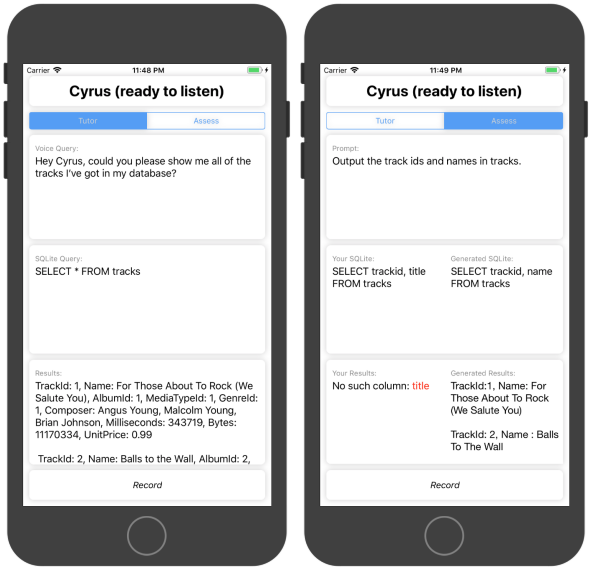
Figure 1: Scheme of *Music* Database.

3.1 Using Cyrus

Cyrus has two main modes, as a tutor, it allows students to choose an example database, and in assessment mode, it additionally allows to choose a difficulty level. In tutoring mode, it simply accepts voice query in natural English, maps the query to SQL, executes it, and shows the computed response and the SQL query it executed to produce the result. Students are allowed to edit the SQL query or write their own, bypassing the voice interface, for execution. In assessment mode though, it shows only the test queries in English at the difficulty level chosen by the student to transcribe in SQL. However, in assessment mode, the voice interface is disabled leaving only the text interface for the students to write the SQL queries without system assisted translation from NLQ. A score is shown at the end of the session informing the student how well she did according to the grading scheme of the instructor. Figure 2 shows the Cyrus interface in both tutoring and assessment modes.

3.2 Supported Query Classes

In the current Cyrus prototype, our focus has been to support a sufficient number of SQL query classes for an entry level database class, and allow multiple natural language renditions of the queries to support variability and personalization. Students with different levels of spoken English skills (e.g., especially students with non-English native languages) are expected to express differently in free form languages, although the corresponding SQL renditions are usually limited. Although in formative and summative assessment mode, written queries will be framed by the course instructors, in tutoring mode, the students are expected to frame their own queries just by looking at the database schema and the instance. Thus, the expressions of the queries in English will vary from student to student necessitating a more flexible natural language input support. In the sections below, we use the music database scheme



(a) Cyrus in tutoring mode. (b) Cyrus in assessment mode.

Figure 2: Cyrus interface.

of figure 1 to illustrate the functionalities of Cyrus and discuss the supported query classes on intuitive grounds. We note that while Cyrus is able to map most queries, in the event it is presented with a query for which it cannot find a mapping, Cyrus will ask to restate the query differently in order to retry.

3.2.1 Wildcard Queries. Consider, for example, the query *"List all music tracks"* over the music database. Obviously, the information for all music albums are in the *tracks* table, and a wildcard query will return the expected response. But, unlike other systems, this query can be asked in multiple different ways in Cyrus, including the following widely different forms, and all will still map to the wildcard SQL query below.

SELECT *
FROM *tracks*;

(1) *"Hey Cyrus, could you please show me all of the tracks I've got in my database?"*
(2) *"What are the songs in the database?"*
(3) *"List all of my tracks for me."*
(4) *"Tracks, please."*

In Cyrus, indeed an exceedingly large number of verbal syntax are allowed so long sufficient keywords are used that can be successfully related to a table name in the chosen database. In particular, query 2 above did not use the keyword *tracks*, yet Cyrus is able to relate the *tracks* table to the songs keyword as a personalization feature leveraging a synonym/vocabulary database.

3.2.2 Projection Queries. Often, specific field of tuples or columns are of interest and need to be extracted. Cyrus supports such extractions as projection queries. However, this innocuous extension of wildcard queries is relatively harder to map and its simplicity is largely deceptive. Consider the following natural language queries. These queries map to the SQL query

SELECT *TrackId, Track*
FROM *tracks*;

(5) *"Show track id and name from the tracks table."*
(6) *"Output the track ids and names in tracks."*
(7) *"List the number and title of the songs."*
(8) *"Track id, name, tracks."*

In these queries, Cyrus primarily looks for a table name to match followed by matching column names, uniquely. Note that, the synonyms and similarity functions it uses makes the matching process complicated. Conceptually, it uses a maximizing function based on Levenshtein Distance (discussed in section 4.3.2) to find a table and attribute list pair that uniquely matches the terms in the English query. In particular, query 7 above still maps to the SQL query above because *number* matches with *TrackId*, *title* with *Track* and *songs* with *tracks*, and the combined similarity of the pair $\langle songs, \{number, title\}\rangle$ is much greater for the pair $\langle tracks, \{TrackId, Track\}\rangle$ than any other pairs.

3.2.3 Selection Queries. Queries satisfying constraints are also supported in Cyrus. Consider the SQL query

SELECT *
FROM *tracks*
WHERE *TrackId=1479* AND *Composer='Jimi Hendrix'*;

which lists the track details for the song with the *TrackID* 1479 for composer *Jimi Hendrix*, in the table *tracks*. This query, as usual, can be asked in any of the following forms, among many other.

(9) *"Get tracks composer Jimi Hendrix where the track id is 1479."*
(10) *"Print Jimi Hendrix composed songs with the number 1479."*
(11) *"Show me the tracks composed by Jimi Hendrix if the track id is 1479."*
(12) *"Tracks composer Jimi Hendrix 1479 track id."*

Cyrus maps these voice commands to the selection query above using techniques similar to the projection queries. Projection of course can be easily combined with a selection query, e.g., the SQL query below is a mapping from the NLQ that follows.

SELECT *Name, MediaTypeId, Genre*
FROM *tracks*
WHERE *TrackId=1479* AND *Composer='Jimi Hendrix'*;

(13) *"Print track name, media type and genre of Jimi Hendrix composed songs whenever the number is 1479."*

3.2.4 Join or Multi-Relational Queries. Join queries, often called multi-relational or SPJ (select-project-join) queries, are in fact the most general and common type of query classes in relational databases. Such queries require linking more than one relations to form a large table on which the selection conditions and projections are applied to find responses. For example, consider the queries

(14) *"List all the artists and their albums distributed by Redeye Distribution in USA that charted top 5 in USA in 2017."*
(15) *"List top 5 ranked 2017 albums and artists distributed by Redeye Distribution in USA."*

Cyrus constructs the join query below over the scheme in figure 1 for either of the two NLQs above.

```
SELECT Album, Artist
FROM tracks NATURAL JOIN distributors NATURAL JOIN charts
WHERE Distributor='Redeye Distribution' AND Year=2017
  AND Rank ≤ 5;
```

From our example database, we compute the response as *Reflection* by *Brian Eno* and *Take Me Apart* by *Kelela* which ranked higher than 6 in multiple charts in 2017 under the label *Warp*, which can be computed only after joining these three tables. Although *Redeye* distributed other albums such as *Death Peak* by *Clark, Shake the Shudder* by *!!!*, and *London 03.06.17* by *Aphex Twin*, they did not make the top 5 chart anywhere in 2017.

3.2.5 Division Queries. Consider the queries below.

(16) *"List artists who recorded albums under all the labels artist Gone is Gone has ever recorded."*

(17) *"List all the artists who have recorded albums at least with all the labels who recorded Gone is Gone too."*

In such queries, we are interested to find association of one object with a set of objects that cannot be computed using one simple join query – called the *division* queries. While it can be computed in several steps without using the concept of division, Cyrus maps these queries to the nested SQL query below utilizing SQL's tuple variable feature. Recognizing and mapping such a query is one of the most difficult ones in NLQ to SQL translation.

```
SELECT Artist
FROM tracks AS t
WHERE (SELECT Label
       FROM tracks
       WHERE Artist=t)
      CONTAINS
      (SELECT Label
       FROM tracks
       WHERE Artist=
         'Gone is Gone')
```

3.2.6 Aggregate Queries with Sub-Group Filtering. In contrast, aggregate queries insist on creating groups on which aggregate operations such as **sum**, **avg**, *max*, **min**, etc. are computed and filter conditions applied. For example, consider the query

(18) *"Print the artists who sold more than 2 million copies of their albums in USA in 2017."*

This query requires filtering out first only those albums sold in 2017 followed by creating a subroup (GROUP BY) of albums with their sales record to find the total sales and then only selecting those artists who has more than 2 million in sales within this filtered group (HAVING clause). On our example database, Cyrus translates this NLQ to the following SQL query.

```
SELECT Artist, SUM(Sales)
FROM tracks NATURAL JOIN charts
WHERE Year=2017
GROUP BY Artist
HAVING SUM(Sales) > 2,000,000
```

4 CYRUS SYSTEM ARCHITECTURE AND IMPLEMENTATION

Voice processing apps are popular on mobile platforms and mobile app operating systems such as Apple iOS, Microsoft Phone OS, HP WebOS and Google Android are supporting an increasing number of such tools for app developers. These tool support includes voice-to-text, speech analysis, and text-to-speech libraries. One of the leading intelligent voice processing system is Apple's Siri on iOS platform. In iOS in particular, Apple supports NSLinguisticTagger, AVSpeechSynthesizer, and SFSpeechRecognizer classes in its Foundation and Speech frameworks suit that are relatively more mature and stable. NSLinguisticTagger is a uniform interface that can be used for a variety of natural language processing functions such as segmenting natural language text into paragraphs, sentences, or words, and tag information about those segments, such as part of speech, lexical class, lemma, script, and language. The AVSpeechSynthesizer class, on the other hand, can be used to produce synthesized speech from text, while the SFSpeechRecognizer class help recognize a speech of a locale. Availability of these enabling tools was the primary motivation for choosing iOS platform for Cyrus along with the fact that Apple iPhone is the most widely used mobile smart device.

At a high level, the Cyrus query processing pipeline proceeds in five distinct steps - voice (query) acquisition, voice to text transcription, text parsing, text to SQL mapping, and SQL query processing. In Cyrus, we have assembled the pipeline leveraging iOS speech library classes embedded within our query processing system. We describe the process below in some detail but defer a full discussion to an extended version of this article.

4.1 Voice Query Acquisition and Transcription

In iOS platform, SFSpeechRecognizer requires the use of permission through its requestAuthorization function for app permission to perform speech recognition, followed by a monitoring session using SFSpeechRecognitionTask via the AVAudioSession to create a new live recognition request using SFSpeechAudioBufferRecognitionRequest. This recognition task then keeps track of the user's speech and utilize a result handler to generate the most accurate text transcription possible until the recognition request has been completed. While these functions instantiate to local language by default, we have used English initialization to limit variability.

4.2 Parsing Natural Language Queries

Once the text transcription is received, semantic understanding of the text query can begin using natural language processing. We preprocess the text using a process called *lemmatization* that standardizes the text query. For this purpose, we use the powerful NSLinguisticTagger class which is able to perform language identification, tokenization, lemmatization, part of speech (PoS) tagging, and named entity recognition tasks with proper instantiations. As mentioned earlier, we instantiate our language to English and proceed with the remaining steps in parsing the text. After tokenization, a lemmatization step is performed to reduce inflectional forms and often derivationally related forms of words to a common base form. For example, words such as *am, are,* and *is* are replaced with *be*, a stem form of a word token, to help transcribe sentences such as *the boy's cars are different colors* to *the boy car be differ color*. Lemmatization help in situations when inflectional and derivational forms force us to search for too many words and establish their semantic meanings making understanding difficult although the words involved differ in their flavor.

Before we initiate PoS tagger in NSLinguisticTagger to obtain a tagged string Q_p, we include tagger option *.joinNames* to collapse

"San Francisco" to "SanFrancisco" to be able to treat this token as a singular entity instead of two separate words, and enumerate the tags within the string using a name type and lexical class scheme, to aid semantic matching of the tokens with the database scheme in our next step.

4.3 Text to SQL Mapping and Query Processing

Mapping the PoS tagged and enriched text query is a complex process. In Cyrus, we follow a heuristic approach for the identification of table names for the FROM clause, attribute lists in the SELECT clause and Boolean conditions in the WHERE clause of all SQL queries. This approach also helps us avoid complex grammatical and semantic analysis of the text query that may not bear fruit at the end anyway. But by doing so we risk failure even on a otherwise mappable semantically correct query.

4.3.1 Levenshtein Distance for String Matching. To match entities referenced in the English query with the table names in the database, and to map possible properties of entities to attribute names of tables, we leverage the popular Levenshtein distance for term matching, which basically is a edit distance function with some interesting properties, including triangle inequality. Mathematically, Levenshtein distance between two strings a and b is a function of the form

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+ \\ \qquad 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $i = |a|$ and $j = |b|$ are lengths of strings a and b, and $1_{(a_i \neq b_j)}$ is the indicator function which equals to 0 when $a_i = b_j$ and equal to 1 otherwise, and $lev_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b. Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same. As an example, consider the strings "eaten" and "sitting." We can transform or edit "eaten" in five steps,

$$\text{eaten} \xrightarrow{1} \text{saten} \xrightarrow{2} \text{siten} \xrightarrow{3} \text{sittn} \xrightarrow{4} \text{sittin} \xrightarrow{5} \text{sitting}$$

by first substituting "s" for "a", then substituting "i" for "a", then substituting "t" for "e", then inserting "i" before "n", and finally inserting "g" at the end.

to make it "sitting," i.e., $lev_{eaten,sitting}(|eaten|, |sitting|) = 5$. We use a special case of $lev_{a,b}(i,j) = 0$ when the strings a and b are identical or one is a substring of the other.

4.3.2 Table and Column Name Recognition. Since the input to Cyrus is a voice query, we have at least two input data types that we are able to leverage, the sound and the text transcription. Assuming that the voice to text translator was flawless, we can combine analytical tools for both to make sense of the English query in order to map it to an equivalent SQL query. We illustrate the process using the example query 11 in section 3.2.3. For this query (superscript shows the word's sequential position in the sentence),

Show[1] *me*[2] *the*[3] *tracks*[4] *composed*[5] *by*[6] *Jimi*[7] *Hendrix*[8] *if*[9] *the*[10] *track*[11] *id*[12] *is*[13] *1479*[14]

the NSLinguisticTagger will generate the PoS tags and the universal and enhanced dependencies shown in figure 3.

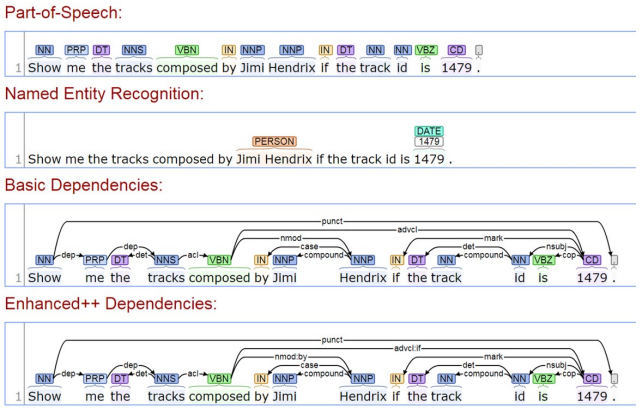


Figure 3: Parsing query 11 using NSLinguisticTagger.

To identify the table names, we assign preference to the terms for matching in the enhanced dependency graph and start with the nodes that have the least in-degrees and are not stop words [1], i.e., we consider only *Show*[1], *tracks*[4], compounds of *Jimi*[7] and *Hendrix*[8] (*Jimi Hendrix*), and *track*[11] and *id*[12] (*track id*), for which the in-degrees are respectively 0, 1, 2 and 2. We discard *Show* because we assume it as an database command and also because it does not match with any database table names "closely." Closeness of a term with another term is determined using a combination of measures such as $\sigma(a,b) = 1 - \frac{homo(a,b)+\lambda(a,b)+\psi(a,b)}{3}$, where $homo(a,b)$, $\psi(a,b)$, and $\lambda(a,b)$ respectively are homonym, substring similarity, and Levenshtein similarity functions with ranges $[0,1]$. Function $\psi(a,b)$ is defined as $\frac{sub(a,b)}{|a|}$ such that $sub(a,b)$ returns the length of maximum continuous substring of a of b, and $\lambda(a,b)$ is defined as the ratio $\frac{max(|a|,|b|)-lev_{a,b}(|a|,|b|)}{max(|a|,|b|)}$, where higher the value of $\sigma(a,b)$, closer is term a to b. Let us call these terms the set T_n while we call the set of all terms in the sentence without stop words T_p. We compute $\sigma(a,b)$ for all terms $a \in T_n$, and for all terms $b \in S$ where S is the set of table names in database D, and annotate each term a with their similarity score σ. We discard all terms $a' \in T_n$ from T_n for which there exists another term $a \in T_n$ such that $\sigma(a,b) \gg \sigma(a',c)$. For example, for query 11, we discard the compound term *Jimi Hendrix*, but retain *track id* along with *tracks*, because we have a table in D named *Tracks*. We call this new set T_c, and call the set of table names in D that matched with the terms in T_c, T_d. We maintain a list L_t of triples $\langle a, r, \sigma(a,r)\rangle$ such that $a \in T_d$ and $r \in S$, for every r that met the filter condition $\sigma(a,r) \gg \sigma(a',c)$.

In our next step, for every triple $m \in L_t$ of the form $\langle a, r, \psi\rangle$, we compute a similarity score Ψ as follows.

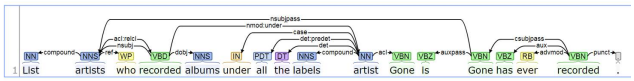
$$\Psi(a,r) = \sigma(a,r) + \Sigma_{i=1}^{|T_p|} \mu(a_i, b_i) - \pi(A)$$

where, $\mu(a_i, b_i)$ is a Stable Marriage [5] matching function such that pairwise matching is maximized for the terms in $a_i \in T_p$ (i.e.,

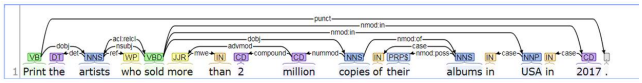
the candidates) and the attributes of the table $r(R)$, i.e., $b_i \in R$. The one to one function μ assigns $\sigma(a_i, b_i)$ the highest matching score possible such that for no other $b_j \in R$, $\sigma(a_i, b_j) > \sigma(a_i, b_i)$. The terms that could not be matched are assigned 0. If the set of terms $a_i \in T_p$ that could not be matched is A, then $\pi(A) = \frac{|A|}{|T_p|-1}$ is a penalty function that compensates for the missing candidates not finding a matching partner. Once we compute $\Psi(a, r)$ for every $m \in L_t$, we choose distinct r's for which $\Psi(a, r)$ is maximum and insert pairs $\langle r, T_p \setminus A \rangle$ in a list F_t of tables names r for the SQL query we intend to construct using attributes $B = T_p \setminus A$, which concludes the process of table and column name identification process.

4.4 Query Construction

In general, construction of an SQL query from an NLQ is a complex and involved task. Especially, reputed PoS parsers such as Stanford CoreNLP parser too pose significant challenges in the analysis process. For example, as shown in figure 4(a), CoreNLP parser does not recognize the singer *Gone is Gone* as a named entity for the division query 16. Instead, it identifies the artist as three separate verb incarnations. In our heuristic driven mapping process, we keep in mind that terms such as "all" and "every" along with their synonyms play an important role in division queries, and that CoreNLP parser or NSLinguisticTagger do not always identify all possible named entities, we thus proceed to analyze the dependency graph cautiously.



(a) Division query 16.



(b) Aggregate/Sum query 18.

Figure 4: Dependency parsing.

Our table and column name identification process will isolate *tracks* as the table name, and *Artist*, *Album*, and *Label* as the attributes of interest for table *Tracks* during the initial step. For the division query 16, we look for a set of objects that are contained in another – a set of *labels*. We do so because the noun "artists" record "albums", and there is a second term "recorded" at the end of Q_p, we look to see if other artists are linked. We find in the dependence graph that "recorded" is linked to "Gone" through a verb "has" and a modifier "ever", and they are after the second noun "artist", we conclude that the set of terms between "artist" and "has" is a named entity if the NSLinguisticTagger did not do so already, and we analyze it further. We then discover that the second "artist" is qualified – "all the labels", and so we construct the subquery

SELECT *Label*
FROM *tracks*
WHERE *Artist= 'Gone is Gone'*

The *nsubjpass* link from *Gone is Gone* back to the first noun "artist" helps us construct the container subquery and tie it up with the driver query completing the division query.

SELECT *Label*
FROM *tracks*
WHERE *Artist= t*

Aggregate queries such as query 18 are constructed similarly by analyzing the dependency graph in figure 4(b), and realizing that it is also a join query over the attributes *Artist, Sale, Region, Album,* and *Year* in the tables *Tracks* and *Charts*. We note that our aggregate query mapping algorithm shares similarity with the approach presented in [15], without the restrictions they impose on the NLQ.

5 CONCLUSION AND FUTURE RESEARCH

There are many approaches to advancing NLP-based voice querying. By having an interactive spoken dialogue interface for querying relational databases, we were able to build a functional learning management system for SQL that has the potential to support the classes of queries novice SQL learners encounter frequently. In our first edition of Cyrus, our focus was to build a concept system to demonstrate its feasibility as a voice assistant. In our testing, we have observed that Cyrus was successful in mapping queries 1 through 18 properly and executed them flawlessly on our test database. The division and aggregate queries remain under refinement and further adjustment as they are currently minimally functional. Despite its relatively weaker ability to map all division and aggregate queries, the progress Cyrus embodies is nonetheless intellectually satisfying which we plan to address in future. The complete mapping algorithm, which we have omitted in this paper for the want of space, along with a first database course classroom user study will be published elsewhere.

REFERENCES

[1] English stop words list - Github. https://tinyurl.com/y86xbn9e, November 2016. Accessed: January 18, 2018.

[2] A. Ahadi, J. C. Prior, V. Behbood, and R. Lister. A quantitative study of the relative difficulty for novices of writing seven different types of SQL queries. In *ACM ITiCSE, Vilnius, Lithuania, July 4-8, 2015*, pages 201–206, 2015.

[3] P. Brusilovsky, S. A. Sosnovsky, M. Yudelson, D. H. Lee, V. Zadorozhny, and X. Zhou. Learning SQL programming with interactive tools: From integration to personalization. *TOCE*, 9(4):19:1–19:15, 2010.

[4] C. Chen and M. Chen. Mobile formative assessment tool based on data mining techniques for supporting web-based learning. *Computers & Education*, 52(1):256–273, 2009.

[5] B. Genc, M. Siala, B. O'Sullivan, and G. Simonin. Finding robust solutions to stable marriage. In *IJCAI, Melbourne, Australia, August 19-25, 2017*, pages 631–637.

[6] C. Kleiner, C. Tebbe, and F. Heine. Automated grading and tutoring of SQL statements to improve student learning. In *Koli Calling, Koli, Finland, November 14-17, 2013*, pages 161–168, 2013.

[7] S. Kumar, A. Kumar, P. Mitra, and G. Sundaram. System and methods for converting speech to SQL. *CoRR*, abs/1308.3106, 2013.

[8] D. Lavbic, T. Matek, and A. Zrnec. Recommender system for learning SQL using hints. *Interactive Learning Environments*, 25(8):1048–1064, 2017.

[9] F. Li and H. V. Jagadish. Understanding natural language queries over relational databases. *SIGMOD Record*, 45(1):6–13, 2016.

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

[11] A. Mitrovic and S. Ohlsson. Implementing CBM: SQL-tutor after fifteen years. *I. J. Artificial Intelligence in Education*, 26(1):150–159, 2016.

[12] J. R. Prior. AsseSQL: an online, browser-based SQL skills assessment tool. In *Innovation and Technology in Computer Science Education Conference Uppsala, Sweden, June 23-25, 2014*, page 327, 2014.

[13] A. Reis, D. Paulino, H. Paredes, and J. Barroso. Using intelligent personal assistants to strengthen the elderlies' social bonds - A preliminary evaluation of amazon alexa, google assistant, microsoft cortana, and apple siri. In *UAHCI, Vancouver, BC, Canada, July 9-14, 2017*, pages 593–602, 2017.

[14] T. Shelley, C. Dasgupta, A. Silva, L. Lyons, and T. Moher. PhotoMAT: A mobile tool for aiding in student construction of research questions and data analysis. *Technology, Knowledge and Learning*, 20(1):85–92, 2015.

[15] Z. Zeng, M. Lee, and T. W. Ling. Answering keyword queries involving aggregates and GROUPBY on relational databases. In *EDBT, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 161–172, 2016.