Tech Report: Design and Evaluation of an SQL-Based Dialect for Spoken Querying

Kyle Luoma
University of California, San Diego
La Jolla, California
kluoma@ucsd.edu

Arun Kumar
University of California, San Diego
La Jolla, California
arunkk@eng.ucsd.edu

ABSTRACT

As automatic speech recognition matures, there is growing interest in speech-driven and multimodal database querying. In this exploratory work we study the potential of a speech-first dialect of SQL for more natural spoken querying with correctness guarantees. We desire minimal deviation from SQL, less structural rigidity, and an unambiguous context free grammar. We call our dialect SpeakQL. We devise a series of features to satisfy the desiderata and build a SpeakQL-to-SQL translator. We evaluate SpeakQL's ease of use against SQL for spoken querying with an A/B user study. The quantitative results show that despite being slightly more verbose, SpeakQL is not statistically significantly slower or faster than SQL for dictation, but most participants find SpeakQL easier for more complex queries. The qualitative feedback suggest an affinity for some of our new features and an overall user experience that validates SpeakQL's potential to make spoken querying easier.

1 INTRODUCTION

The database community has long explored new interfaces to make databases easier to access and query in various kinds of settings for both data professionals and lay users. Typing SQL remains the main form of usage for data professionals but many lines of research have explored other new modalities: visually-oriented, touchscreen-oriented (e.g., [12]), speech-oriented (e.g., [30]), and natural language interfaces or NLIs (e.g., [10, 14, 29, 33]). In particular, our recent work on SpeakQL [30] showed that modern automatic speech recognition (ASR) tools have matured enough to combine the benefits of dictating regular SQL in conjunction with touchscreen capabilities. Such speech-driven querying was shown to make query specification significantly faster in tablet environments, which could offer more flexibility for anywhere-anytime querying for data analysts and other data professionals.
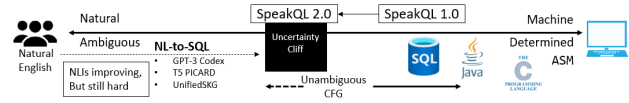
Figure 1: Bridging the gap between naturalness and determinism in programming languages; extension of figure from [30].

In this paper, we go further to ask a more radical exploratory research question: *If we are to design a structured query language for the speech-first era, how should it look?* At first blush, it may sound odd to still study structured querying in the era of ChatGPT [6, 23, 26] and advances in NLIs [14]. Why bother with anything more than mere English "prompting"? First off, there is a difference in motivation: NLIs mainly target lay users, while our focus is on the swathe of data professionals who are *already familiar* with SQL and use it regularly. Second, AI models still do not offer the strong guarantee of "correct-by-construction" that SQL-like languages do– we know *exactly what* we get in response to a query. In contrast, AI models suffer the "hallucination" problem [11] that can cause insidious errors that may be hard to catch, especially on arbitrary database schemas and more complex queries. Overall, we believe exploiting ASR to make SQL-style querying easier could help data professionals.

Desiderata. We start by describing some desirable properties of a spoken structured query language motivated by balancing *usability for specification* and *practicality in design.* (1) Minimal deviation from SQL to ensure it is easy to pick up for people who know SQL already. (2) Less rigid than SQL and more natural English-style flow in structure for the speech modality. (3) Unambiguous context free grammar (CFG) to ensure a valid spoken query can be translated to a *semantically equivalent* SQL query to enable use of an RDBMS as is for query execution.

Our Approach. In this paper, we design and evaluate a *new dialect of SQL* for spoken querying in response to the above desiderata. We call our dialect SpeakQL 2.0 or just *the SpeakQL dialect.* Figure 1 illustrates where SpeakQL 2.0 falls in the spectrum of naturalness and determinism. Unlike NLIs that do not offer correctness guarantees, SpeakQL 2.0 has an unambiguous CFG. But it is less rigid than regular SQL although it is not multimodal (no touchscreen component) like what SpeakQL 1.0 proposed [30]. We design the SpeakQL dialect as an extension of the ANSI SQL grammar. It has several features to help increase the "naturalness" of speaking queries in the style of "stream of thought" instead of specifying everything all at once.

Motivating Applications. Before explaining the dialect's features, we describe some motivating application scenarios for spoken structured querying with a dialect such as SpeakQL.

- *On-the-go ad hoc database access.* Data-driven operations are now common in many domains. Organizations with field-based assets who operate in remote or austere environments such as the military [9] and the oil-and-gas industry [20] may face barriers to purely typing-based or touch-based data access on the field. Such barriers could be due to lack of workspace for keyboards, personal protective equipment requirements impeding typing/touching, or on-the-go working conditions such as a mobile headquarters. Spoken querying boosts capabilities in such settings for on-the-go ad hoc database access. As an example, suppose a military cyber-defense team in the field that must wear protective equipment detects anomalous behavior on some client devices. Pre-built (canned) analytics dashboards may not address all their unexpected querying needs. NLIs run the risk of erroneous query translation. But dictating a precise structured query can empower the team to access and analyze relevant network traffic data without compromising team security, agility, and query fidelity.

- *Assistive technology for people with motor impairment.* The U.S. Bureau of Labor Statistics data shows that in 2020 in the IT and engineering sectors, 18% of nonfatal injuries/illness involving days away from work were in the upper extremities, viz., shoulders, arms, hands and wrists [1]. For such people, as well as for many people with arm disabilities, typing, clicking, or touchscreens may not be viable as a modality but speech can be a powerful modality. Thus, spoken structured querying can help data professionals with such disabilities or injuries.

SpeakQL Design and Features. The SpeakQL dialect has 4 new features with increasing sophistication, grouped into 2 categories: smaller local changes to the grammar's production rules and deeper structural changes with more complex rules. Each feature is *optional*, which means regular SQL syntax is also valid in SpeakQL. Section 3 dives into their details with examples but we summarize their rationale here. The first category has the following two features. (1) *English synonyms* for some SQL keywords such as SELECT and FROM. (2) *Natural functions* to omit speaking of special character symbols such as commas and parentheses in some contexts. These two extensions make SpeakQL sound less like code and more like English (compared to SQL). So, they enhance the "naturalness" of the spoken query.

The second category has the following two features. (3) *Clause reordering*. Specifically, the SELECT, FROM, and WHERE clauses can be spoken in any order. Query modifier clauses such as GROUP BY and ORDER BY too can be reordered. (4) *Unbundling* of complex queries into per-table decomposed queries. This allows users to reason about one table at a time instead of "all at once" like in SQL. This is inspired by function-stitching style programming seen for Python Pandas and Spark DataFrame. It can reduce the amount of schema context to keep in mind when speaking, albeit at the cost of raising verbosity and query token lengths. Overall, these two features reduce query rigidity and offer more freedom for "stream of thought" querying.

User Study-based Evaluation. In this paper, we focus primarily on evaluating the *usability* of the SpeakQL dialect for spoken querying compared to regular SQL. We leave more extensive comparisons with other querying modalities or integration with multimodal interfaces to future work. We implemented the SpeakQL dialect and conducted a within-subjects A/B user study. We had 22 participants, all UCSD students familiar with SQL and relational databases. They were given a 6-table university database schema and asked to speak 12 queries, half designed to be simple and half, complex. The user study was conducted over a span of 4 months. Performance was measured in terms of both the time required to plan and specify a full query in response to an English prompt posed and the number of attempts till a fully correct query.

Overall, we find no statistically significant differences in the total query specification time for SQL vs. the SpeakQL dialect. This suggests that SpeakQL's extra verbosity was compensated for by lower thinking effort/time. Indeed, on average SpeakQL sees slightly lower median specification times for planning the complex queries. We also find that participants get better at speaking SpeakQL as they become more familiar with its features. The mass of qualitative textual feedback from the post-participation surveys also offer numerous interesting insights into the strengths and current weaknesses of SpeakQL. We see both positive and negative feedback on both the dialect and its individual features. But in aggregate, participants reported that SpeakQL made it "much easier" or "somewhat easier" to use than SQL between half to four-fifths of the time depending on the feature. Natural functions and unbundled queries were the most highly liked and used features, while synonyms were (perhaps surprisingly) deemed not that useful. In all, our user study results and surveys suggest that the SpeakQL dialect is indeed user-friendly. We hope it spurs more research conversations in the community on more design iterations, additional features, and ultimately, fully fledged spoken querying for databases in more contexts.

To summarize, this paper makes the following technical contributions:

- To the best of our knowledge, this is the first paper to systematically study and evaluate an extension to SQL tailored for spoken querying.

- We present the SpeakQL dialect with four new features that raise naturalness and reduce rigidity compared to SQL, while preserving correct-by-construction guarantees with a context free grammar.

- We describe the implementation of the SpeakQL dialect, including its grammar rules and our translator to convert any SpeakQL query to regular SQL to ensure it can be used for existing RDBMSs.

- We present an extensive user study-based evaluation of SpeakQL vs. SQL for spoken querying. Our empirical findings, both quantitative and qualitative, suggest the usability of such a dialect and also offer avenues for more research to improve it.

2 BACKGROUND

2.1 The Structured Query Language (SQL)

Origins and Purpose. Introduced in 1974 [8], SQL is nearing its 50th birthday. Despite (or perhaps because of) its age, it remains the de facto standard language for database quyering. SQL (originally named SEQUEL) was intended for both application programmers and a non-programmer target audience of business professionals and other laypersons requiring data access from relational databases [7]. Its initial design and following updates were informed by a user-centric approach, and it is perhaps one of the first programming languages for which human-computer interaction considerations were deliberately studied [27, 28]. Due to its high popularity, declarative nature, targeted purpose, structured syntax, and human-centric design, SQL is a natural starting point for a spoken query language.

Syntax Grammar and SQL Variants. Numerous variants of SQL syntax exist from multiple vendors and open source projects. While each variant tends to contain implementation-specific features targeted at specific RDBMSs, most (if not all) generally adhere to the ISO/IEC 9075-1:2016 information technology standard for SQL [13]. Seven SQL grammars are available under various open source licenses on the ANTLR parser Github repository including: Hive, MySQL, PL-SQL, PostgreSQL, SQLite, Trino, and T-SQL [2].

Human Factors. Human factors evaluations were conducted as part of the SEQUEL development effort. Usability experiments comprised of teaching SEQUEL to programmer and non-programmer college students. The study yielded several results, including the recommendation to make SEQUEL a layered system consisting of three layers representing increasing levels of complexity, and the recommendation to replace complicated correlation and computed variable syntax with the join feature, which most SQL users are familiar with now.

Reisner also discovered that sources of minor errors when converting English statements into SEQUEL queries included ending errors, spelling errors, and synonym errors. These discoveries resulted in the recommendation to incorporate spelling correction, introduce a synonym dictionary to the language syntax, and a create stem-matching procedure as user aids that would enable users to use keywords with various forms of conjugation. [27] In a later study, Reisner also confirmed that query complexity has a directly proportional effect on the likelihood of error occurrence during query formulation [28].

A more recent SQL usability study revealed that user tendency toward invalid syntax synonyms, omission of punctuation, and the NL-like nature of some SQL keywords can be sources of programming errors among novice users. Table joins, aliases, and subqueries were also identified as significant sources of programming errors [19].

2.2 Natural and Controlled Natural Languages

Natural Language Interfaces. Natural language interfaces (NLIs), such as the recently popular ChatGPT [6, 23, 26], show that NL-based chatbots can be a viable tool for many purposes, including NL-to-SQL querying, wherein users express their query intent in regular English. NL-to-SQL is still an active area of research [6, 14, 29, 33], and it has strong potential to lower the barrier to entry to lay users (people without SQL knowledge). But NLIs still suffer from three issues in technical applications such as database querying that data professionals may be wary of: *ambiguities*, which can confound user goals; *out-of-vocabulary terms*, common in database schemas and predicate content, can hinder accurate translation; and *lack of correctness guarantees*, compounded by the "hallucination" problem of generative NLP models. That said, recent research suggests that NLI with more restricted grammar and/or structure can improve user experience for technically complex tasks [21].

Controlled Natural Languages. Controlled or restricted NL are based on an NL such as English but more restrictive in their lexicon, syntax, and semantics. They retain a majority of its base NL properties and are defined explicitly [15]. Most PLs (including SQL) are not controlled NLs because their syntax deviates too much from the NL and have many statements that do not exist in the NL. Controlled NLs have been evaluated against linear keyword languages such as SQL and found to be easier for novice users for performing data retrieval tasks [31]. In contrast to NL-to-SQL described previously, a controlled NL does offer the benefit of being "correct-by-construction" at the cost of being less flexible.

Naturalness. Naturalness of a controlled NL can be evaluated by how close an expression in it is to its base NL. This is evaluated in terms of both *readability* and *understandability*. These criteria can range from completely unnatural, where the controlled NL uses symbols, characters, and unnatural keywords, to languages with natural sentences where the controlled NL can yield expressions that appear as if they were written in the base languages [15].

2.3 SpeakQL

A subset of the authors designed the SpeakQL speech+touch multi-modal querying interface in prior work [30]. It was aimed at data professionals such as data analysts, nurse informaticists, and DBAs who desired ad-hoc on-the-go querying in settings without a regular computer but with mobile devices such as a tablet. That paper's user study showed that the SpeakQL interface reduced query specification times vs. typing SQL in such settings by 2.7x on average. But conversations with such data professionals in that work revealed a key functionality gap: people with SQL knowledge may want to do a quick record retrieval or analytics query in an ad-hoc setting where even touchscreens are unviable for query specification, let alone keyboards, but voice is feasible. Speech-driven querying can also help people with disabilities or temporary injuries and perhaps also augment conversational assistants such as Alexa, Siri, etc. to aid in database querying. That provided the basis for this exploratory work on a spoken SQL dialect.

3 OUR SPEAKQL DIALECT

We now present our prototype speech-first dialect extension of SQL. We overload the prior art interface name to call our dialect SpeakQL too. It has four new features, all optional for usage:

- Keyword Synonyms and Optional Syntax
- Natural Functions
- Query Expression Ordering
- Complex Query Unbundling

SQL Keyword	SpeakQL Synonyms
SELECT	Select, Find, Retrieve, Get, Show Me, Display, Present, What Is, What Is The, What Are, What Are The
FROM	From, From table, From Tables, In Table, In Tables
' , ' (Comma)	' , ' (Comma), And
JOIN	Join, Join Table, Join With Table, By Joining, By Joining Table, By Joining With Table, Joined With, Joined With Table

Table 1: Synonyms in SpeakQL for SQL keywords.

SpeakQL can be considered a controlled NL based on English that extends SQL. Note that SQL is a constructed language rather than a controlled NL. This distinction arises because the objective of the SpeakQL dialect is to increase naturalness of specifying queries, achieved via the introduction of English grammar features and the reduction of special character (non-alphabet) symbol usage in SpeakQL queries.

We defined the SpeakQL grammar by extending a big part of the MySQL grammar from the Antlr4 repository [2]. We added additional production rules within existing SQL rules to realize our features. This means that SpeakQL is a *superset* of that chosen SQL subset. That is, a query constructed using the regular SQL syntax and keywords is a valid SpeakQL query too. So, users can "fall back" on regular SQL if they desire to.

Due to space constraints, we only provide intuitive explanations and examples of SpeakQL features in this paper. More rigorous grammar details can be found in our full technical report [16].

3.1 Keyword Synonyms and Optional Syntax

This is a simple feature designed to increase the naturalness of an SQL query by enabling more sentence-like expressions. The intuition driving the development of these features is that speech patterns may be more amenable to use NL-like behavior, e.g., omitting syntax such as symbols and punctuation but including concepts such as determinatives (e.g., THE) and prepositions (e.g., OF).

3.1.1 **Synonyms**. This feature is motivated in-part by early human-computer interface research performed on SQL users [27] that recognized the benefit of syntax synonyms and optional word stemming, as well as more recent observations on tendencies toward synonyms [19]. We introduce SQL keyword-equivalent synonyms for the most common DML syntax keywords: SELECT, FROM, and JOIN, as well as for the comma as a column- or table-delimiter within the SELECT and FROM clauses, respectively.

3.1.2 **Optional Syntax**. SpeakQL allows for the use of optional THE and TABLE keywords when dictating a table expression. This permits expressions such as *SELECT everything FROM THE courseoffering TABLE*, which can be more natural for speaking than the SQL equivalent *SELECT everything FROM courseoffering*. Introducing the THE keyword as a determinant clarifies the context of the subject of the SpeakQL sentence, which is the *courseoffering* table. Appending

the TABLE keyword to the expression provides further context and clarity that the referenced table is a tangible object within the database. While neither keyword changes the expression's meaning, their usage improves the naturalness of the SELECT statement, thus potentially improving the dictation experience.

3.1.3 **Examples**.
```
SQL: SELECT area, wheelchairspaces
     FROM room WHERE floor = 2;
```

SpeakQL. **Show me** area and wheelchairspaces **in the** room **table** *where floor equals 2*

```
SQL: SELECT COUNT(id) FROM course;
```

SpeakQL:. **What is the** count parenthesis id parenthesis **in the** course table

3.2 Natural Functions

The SQL subset we support includes aggregator functions such as SUM, AVG, and COUNT. In the prior work on the SpeakQL interface, users had to verbalize the parentheses symbols, which reduces the naturalness of dictation. While parentheses are often essential for disambiguation, for the SpeakQL dialect we identified a set of function references in which parentheses could be omitted safely without affecting the query's semantic meaning. Specifically, our SpeakQL dialect permits the expression of functions naturally, that is without verbalizing parenthesis, for functions that have a single constant or column as an argument. This feature also permits the optional syntax keywords THE and OF to surround the function name, resulting in more NL-like sentence expressions.

However, if the query intent involves the inclusion of an expression as a function argument, the verbalization of parenthesis remains a requirement. This allows the SpeakQL dialect to retain SQL's capability to pass mathematical, comparative, and subquery expressions as function arguments within the boundaries of dictated parentheses, ensuring we avoid ambiguity such as cases in which neighboring SELECT clause elements get misinterpreted as function arguments by the translator.

3.2.1 **Examples**.
```
SQL: SELECT COUNT(id) FROM course;
```

SpeakQL. **Get the count of** id **from the** course **table**

```
SQL: SELECT AVG(units), COUNT(title)
     FROM course
```

SpeakQL:. **Find the average** units **and the count of** title **in the** course **table**

3.3 Query Clause Ordering

This feature allows for optionally reordering the SELECT, FROM/JOIN, and WHERE clauses, as well as the GROUP BY, HAVING, ORDER BY, and LIMIT clauses. That is, these clauses may appear in any order within a SpeakQL statement.

3.3.1 **SELECT-FROM-WHERE Ordering**. Intuitively, we recognize that there are alternate paths to forming a SQL query. In some instances, the SELECT, FROM, WHERE ordering of SQL syntax
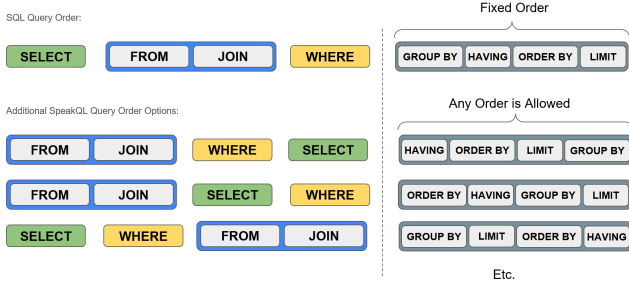
Figure 2: Alternate Ordering

is the order a user may find most-useful when dictating a query. In other cases, users may wish to reason about table sources and joins prior to defining columns and functions; alternatively, users may wish to establish restrictions using where predicates before defining other aspects of a query. Additionally, alternate ordering provides "second chances" for query recovery. For example, if a user is dictating a query and defines the SELECT and WHERE expressions, forgetting to state the table source, they may resolve this omission by dictating the FROM clause at the end of the query rather than starting over from the beginning.

3.3.2 **Modifier Ordering**. The *modifier* optional ordering feature (in this paper modifiers refer to the GROUP BY, HAVING, ORDER BY, and LIMIT expressions) is partially motivated by the observation that queries that require multiple modifier statements such as GROUP BY and HAVING tend to be more complex than simple single table queries or queries with a single modifier such as GROUP BY [4]. SQL syntax requires that these expressions occur in the strict order GROUP BY, HAVING, ORDER BY, and LIMIT. If these expressions appear out of order within a query, it is invalid and requires correction. While this is not a significant problem for typed queries, as they can easily be rearranged in a text editor, if such an error is introduced during the spoken querying process, more sophisticaed error correction is required and the query speaker must likely re-dictate the entire query.

3.3.3 **Examples**.

```
SQL: SELECT DISTINCT termperiod FROM term
     WHERE year = 2022;
```

SpeakQL. **From the** *term* **table show me distinct** *termperiod* **where** *year equals 2022*

```
SQL: SELECT facultyname, ondays
     FROM courseoffering
     WHERE capacity > 20
     ORDER BY facultyname LIMIT 10;
```

SpeakQL:. **In the** *courseoffering* **table where** *capacity is greater than 20* **find** *facultyname* **and** *ondays* **limit** *10* **order by** *facultyname*

3.4 Query Unbundling

This features aims to make it easier to formulate more complex queries joining multiple tables. In SQL, the SELECT clause requires the user to specify *all* columns, scalars, and functions from across all

tables in one go, followed by naming *all* table sources, subqueries, and joins in one go, followed by expressing *all* constraints in the form of WHERE predicates. Only after all that can the user add modifiers such as LIMIT, ORDER BY, HAVING, and GROUP BY. Basically, it is a "global" approach to using the full database's schema in query construction. It forces the retention of a lot of schema details in the speaker's working memory for the entire duration of the query dictation, e.g., all non-aggregate columns that appeared in the SELECT clause must reappear in the GROUP BY clause at the end. While this may not be a big deal for typing, it can be a hindrance for ease of spoken querying.

Query unbundling aims to directly reduce this cognitive load based on the "stream of thought" philosophy inspired by functional programming APIs such as Python Pandas and Spark DataFrames. Basically, this is a "local-first" approach to using the database schema in query construction. It is closer to the logical query plan produced behind the scenes for SQL queries. We extend the grammar to permit the expression of *unbundled SELECT* queries that specify columns, table source, and WHERE predicates for *one relation at a time*. These unbundled relation can then be joined together using separate *join-with* clauses where the speaker defines the join predicate(s). Each unbundled query is delimited by the AND THEN, THEN, or NEXT keywords. Modifier clauses can be specified together in a single expression or separately using multiple modifier clauses. The order of the *unbundled SELECT*, *join-with*, and *modifier* clauses remains optional, retaining the additional flexibility for dictation that SpeakQL offers.

3.4.1 **Unbundled Query Parts**. There are 3 types of unbundled query parts: a single-relation select-project, a join clause that specifies the join criteria between two single-relation queries, and a modifier clause that enables specification of GROUP BY, HAVING, LIMIT, and ORDER BY.

Within a single-relation select-project clause, a relation can be defined as a table reference or a subquery. Projections are expressed within the SELECT clause; and selections relating to the query's relation are defined within the WHERE clause as usual. As with non-unbundled SpeakQL queries, the table expression, SELECT expression, and WHERE expression may be dictated in any order. Table items for unbundled query parts that contain joins are consolidated in the output SQL query as a combination of a FROM clause for a single table and join expressions for the remaining tables. Otherwise, table items for unbundled queries where join conditions between tables are defined within a single-table part's WHERE clause are consolidated in the output SQL query's WHERE clause in the same fashion. We present two examples.

Unbundled SpeakQL query: SELECT a and b FROM table R AND THEN GET c and d FROM S WHERE R.id = S.id

```
SQL: SELECT R.a, R.b, S.c, S.d FROM S, R WHERE R.id=S.id;
```

Unbundled SpeakQL query: SELECT a and b FROM table R AND THEN GET c and d FROM S AND THEN JOIN R WITH S on R.id=S.id

```
SQL: SELECT R.a, R.b, S.c, S.d FROM S JOIN R on R.id=S.id;
```

All 4 queries above are logically equivalent, representable using the relational algebra expression shown below. The first line lists
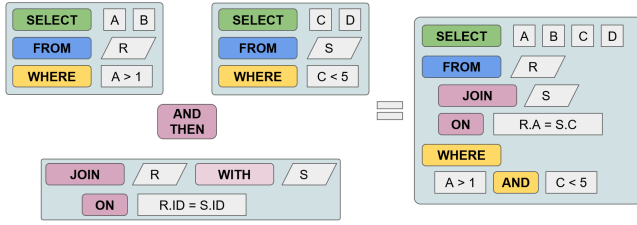
Figure 3: Query Unbundling

the individual unbundled queries in SpeakQL that perform the individual projections on R and S first before adding the equi-join part. (NB: We overload π for non-deduplicating project in bag semantics for brevity sake.)

$$[\pi_{a,b}(R)]\&\&[\pi_{c,d}(S)]\&\&[R \bowtie_{P.id=S.id} S]$$
$$\mapsto \pi_{R.a,R.b,S.c,S.d}(R \bowtie_{R.id=S.id} S) \qquad (1)$$

The above expression introduces two non-standard symbols to represent unbundled queries: square brackets [and] encase individual unbundled query parts, while the double ampersand && is the delimiter of the unbundled query parts, representing the AND THEN keyword or equivalent synonyms in SpeakQL. No other operations, say, relational algebra operations, are permitted in between the individual unbundled parts.

3.4.2 ***Translating an Unbundled Query to SQL****.* Our SpeakQL-to-SQL translator consolidates the unbundled query parts to produce a holistic valid SQL query. Due to space constraints, we present its precise algorithm in the appendix. We describe its basic workflow succinctly next.

First, column projections, function calls, and function arguments from all single-relation select-project unbundled parts are consolidated together into a single SELECT clause for the output SQL query. To avoid ambiguity in column names from different tables that are identical strings, every column reference in either a SELECT clause or WHERE clause that is not already in table-prefixed format (i.e., table.column) is prepended with its source table during the translation. This is shown in the examples above in which the translated SQL converts reference of column, say, "a" in SpeakQL to "R.a" in SQL. If the column reference already prefixes the table, our translator leaves it as is.

Second, selection predicates in the WHERE clauses of individual single-relation select-project query parts are consolidated into a single WHERE clause in the output SQL query using a boolean AND. A complex situation arises for a cross-table selection predicate within a boolean expression. In such cases, we only support *conjunctive* relationships between single-relation components of the overall boolean expression. That is, if a single-relation select-project query part contains multiple selections, these selections are encapsulated in parentheses prior to consolidation within the output SQL query. We currently do not support cross-table disjunctive predicates due to the additional complexity they add for speaking and anecdotally such cases being rarer in practice. As such, users always have the option of falling back on regular SQL syntax in SpeakQL for such cases.

Detailed Example. Figure 3 illustrates how an unbundled SpeakQL query with two separate single-table WHERE clauses gets consolidated into a single SQL query. The relational algebra representation for both queries is given below.

$$[\pi_{a,b}(\sigma_{a>1}(R))]\&\&[\pi_{c,d}(\sigma_{c<5}(S))]\&\&[R \bowtie_{R.id=S.id} S]$$
$$\mapsto \pi_{R.a,R.b,S.c,S.d}(\sigma_{((a>1)\wedge(c<5))}(R \bowtie_{R.id=S.id} S)) \qquad (2)$$

3.4.3 ***Selections Without Projections****.* One tricky situation in unbundling arises when a selection predicate (WHERE clause) is applied to a relation from which no columns are returned, i.e., it has no SELECT clause. This can lead to an "odd" impulse for a query speaker when nothing is retrieved from a table despite it having its own unbundled query part. To handle this situation, we introduce the NOTHING keyword as a special token within the *selectExpression* parser rule in the SpeakQL grammar.

SpeakQL:. FROM TABLE R SHOW ME a AND THEN GET NOTHING FROM S WHERE b = 2 and R.id=S.id

```
SQL: SELECT a FROM R, S WHERE R.b=2 and R.id=S.id
```

3.4.4 ***Automatic Group By Aggregation****.* To help reduce possible errors from incorrect GROUP BY clauses, we introduce the AUTOMATIC or AUTOMATICALLY keywords. The expression GROUP BY AUTOMATICALLY lets the translator infer the output SQL query's GROUP BY clause directly from the SELECT clauses of the unbundled query parts.

SpeakQL:. SELECT a AND THE SUM OF b FROM TABLE R AND THEN GET c and d FROM S AND THEN JOIN R WITH S on R.id=S.id AND THEN GROUP BY AUTOMATICALLY

```
SQL: SELECT R.a, SUM(R.b), S.c, S.d FROM R JOIN S
on R.id=S.id GROUP BY R.a, S.c, S.d
```

3.4.5 ***Verbosity vs. Brevity****.* In general, unbundled queries may contain more tokens than their corresponding SQL query, which means they may take slightly more time than reading the pure SQL translation. So, the higher naturalness comes at the cost of higher verbosity. This is an explicit tradeoff we made based on the rationale that perhaps less "think time" may be needed for the "local-first" unbundled SpeakQL query than the "global" SQL query. We view this tradeoff as reasonable to both reduce the chance of semantic errors and raise the user's overall query dictation experience.

Longer Detailed Example. The English prompt is as follows: "What is the average room seating capacity of rooms in buildings where the course with id 'CSE 232' has ever been offered?" The precise schema of this database is in the appendix.

SpeakQL. **Get** *buildingname* **from** *building*
and then from the *room* **table where** *floor equals 3* **show me the average** *capacity*
and then get *nothing* **from** *courseoffering* **where** *courseid equals quote CSE232 quote*
then join *courseoffering* **with** *room* **on** *courseoffering dot roomid equals room dot id*
next join *room* **with** *building* **on** *room dot buildingid equals building dot id*
and then group by *automatically*

```
SQL: SELECT buildingname, AVG(capacity)
     FROM courseoffering
     JOIN room ON courseoffering.roomid = room.id
     JOIN building ON room.buildingid = building.id
     WHERE courseoffering.courseid = 'CSE232'
     GROUP BY buildingname;
```

4 IMPLEMENTATION

4.1 Language Specification

SpeakQL grammar is defined using Antlr's modified Backus-Nair form (BNF) [24]. The grammar is an extension of the MySQL grammar available on the Antlr GitHub repository [2]. We use a subset of rules nested beneath the *querySpecification* parser rule, which are rules within the *dmlStatement* rule set. While the majority of the MySQL grammar within this subset remains unmodified, we make some significant adjustments within the *selectStatement* and *querySpecification* parser rules and their descendants.

4.1.1 Grammar Extension Strategy. The general strategy for SpeakQL rule definition within the grammar is to add intermediate rules between terminal token nodes and their parent SQL parser rules in order to modularize the grammar in a way that enables abstract syntax tree (AST) manipulation by the SpeakQL-to-SQL translator. This strategy results in deeper ASTs, and has the potential to force trades between modularity and performance. Because SpeakQL is targeted at speech-based database interaction, near-realtime performance is an important design factor. To counteract performance degradation resulting from deeper ASTs, we pruned the source MySQL grammar to include only DML parse rules. Within the DML parse rules, we further pruned out rules that were unlikely to be used during speech dictation.

4.1.2 Grammar. Figure 4 is an excerpt of the SpeakQL unbundling expression grammar in a modified BNF format. The *'?'* symbol appended to a rule indicates that the rule is optional. The *'*'* symbol indicates that 0 to many occurences of a rule are allowed. Symbols or tokens surrounded by double quotes indicate that the content enclosed within the quotes is a terminal symbol. The grammar contains a subset of child and descendant rules relevant to the unbundling feature. *selExpr*, and *whereExpr* grammar descendants are described in additional figures in the appendix.The root query specification rule abbreivated as *querySpec* is the point at which queries that use the unbundling feature diverge.

Unbundled queries may lead with any of the three unbundled query part type rules (*multiJoinExpr, selModExpr,* or *unbdQryOrdSpc*) followed by the *exprDelim* rule that contains the *and then, then,* and *next* terminal tokens. Although unbundled and non-unbundled SpeakQL queries may share the parser rules that describe select expressions and where expressions, unbundled queries possess distinct from expression rules. The *tabExprNoJoin* and *fromClsNoJoin*, as the name suggests, omit any possible join expressions. This serves to ensure that individual unbundled query parts reference only one table, leaving the join operations to the *multiJoinExpr* rule and its descendants.

<qrySpec> ::=	<qryOrdSpc> <selModExpr> | (<multiJoinExpr exprDelim>)? (<selModExpr> <exprDelim>)? <ubndQryOrdSpc> (<exprDelim> (<ubndQryOrdSpc> | <multiJoinExpr>)) (<exprDelim> <selModExpr>)?
<ubndQryOrdSpc> ::=	<selExpr> <whereExpr>? <tabExprNoJoin> | <selExpr> <tabExprNoJoin> <whereExpr>? | <tabExprNoJoin> <selExpr> <whereExpr>? | <tabExprNoJoin> <whereExpr>? <selExpr>
<tabExprNoJoin> ::=	<frmClsNoJoin>
<frmClsNoJoin> ::=	<frmKW> <theKW>? <tblSrcNoJoin> <tblKW>?
<tblSrcNoJoin> ::=	<tblSrcItm> | "(" <tblSrcItm> ")"
<mltiJnExpr> ::=	<mltiJnPrt> (<exprDelim> <mltiJnPrt>)*
<mltiJnPrt> ::=	<mltiInnrJn> | <mltiOutJn> | <mltiNatJn>
<exprDelim> ::=	"and then" | "then" | "next"
<mltiInnrJn> ::=	<inJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")?
<mltiOutJn> ::=	<joinDir> <outJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")
<mltiNatJn> ::=	<natJoinKW> (<joinDir> <outJoinKW>)? <joinKW> <tblSrcItm> <withKW> <tblSrcItm>
<withKW> ::=	"with" | "with table" | "and"

Figure 4: SpeakQL Unbundled Query Grammar Excerpt
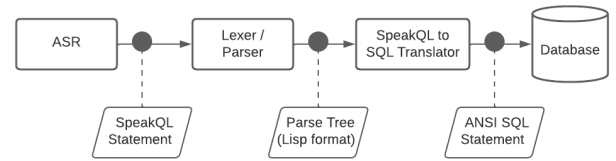
Figure 5: Translation Data Flow

4.2 SpeakQL to SQL Translation

The SpeakQL-to-SQL translator is a Java- and Python-based system that takes a valid SpeakQL as input, and returns a semantically equivalent SQL query (See Figure 5). The translator accomplishes this task through a series of intermediate steps described in the appendix.

4.2.1 Synonym Replacement. During synonym replacement, the translator performs a scan search of all active nodes in the AST for rules that are a member of the keyword and delimiter categories for which SpeakQL synonyms exist. This includes *selectKeyword,*

fromKeyword, *selectElementDelimiter*, and others. When a keyword or delimiter rule is identified during the search, the translator performs an update on the rule node which replaces the terminal token representative of a SpeakQL synonym with its corresponding SQL keyword. Optional keywords such as THE and TABLE are identified in the same manner and removed from the AST.

4.2.2 Expression Reordering. This step includes both the reordering of SELECT, FROM, WHERE and GROUP BY, ORDER BY, HAVING, LIMIT expressions. The translator takes advantage of AST structures that guarantee that all children of the *queryOrderSpecification*, *unbundledQueryOrderSpecification*, and *selectModifierExpression* parser rule nodes are rules that require evaluation and reordering. Specifically, it is guaranteed that the query order expression parser rules will only contain the children *selectExpression*, *whereExpression*, and *tableExpression*. We are also certain that the *selectModifierExpression* has no more than four children of type *selectModifierItem*, each of which may have a single child of type *groupByClause*, *havingClause*, *orderByClause*, or *limitClause*. With these guarantees, the translator simply collects and reorders the AST's parser rules' children into the correct SQL order.

4.2.3 Natural Function Transformation. Translating natural functions to valid SQL function expressions involves the location and removal of the optional keyword parser rules and the insertion of parentheses for all occurrences of the *noParenAggregateWindowedFunction* parse rule in the AST.

4.2.4 Query Bundling. Query bundling is the series of tasks the translator performs to combine a set of unbundled queries into a valid SQL statement. Query bundling is accomplished through a series of up to five steps. The number of steps varies by query and is dependent on the presence of WHERE expressions, JOIN expressions, and aggregate function calls. Unbundling steps are performed in the order:

(1) Infer group by expression (optional)
(2) Bundle select elements
(3) Bundle where statements (optional)
(4) Bundle join parts (optional)
(5) Bundle tables (optional)

A more in-depth description of unbundling feature translation and individual unbundling sub-process steps are described in the appendix.

5 USER STUDY

To assess the usefulness of our prototype features, we performed a comparative A/B user study of the SpeakQL prototype language extension features. The study's primary goal was to identify if SpeakQL *dialect* features and their naturalness had an effect on a user's ability to efficiently dictate a syntactically correct query. This objective is in contrast to the user studies previously conducted on the SpeakQL 1.0 system which, while orthogonal to our work on the SpeakQL 2.0 dialect, had different objectives. The SpeakQL 1.0 user studies were a comparison of querying with typed SQL on a table with the SQL dictation using the SpeakQL 1.0 end-to-end speech + touch system. On the other hand, this SpeakQL 2.0 dialect study employs a Wizard of Oz strategy to simulate a speech-based

interface in order to compare dialects: SQL (the dialect used by SpeakQL 1.0) and SpeakQL 2.0.

5.1 Study Objectives

5.1.1 Research Questions. The study is informed by three research questions:

Q1 - Effectiveness of alternate syntax: To what extent, if any, do syntax synonyms and symbol reduction improve user experience during speech-based querying?

Q2 - Effectiveness of alternate structure: To what extent, if any, does relaxing structure through alternate ordering reduce the tendency for errors during speech-based querying?

Q3 - Effectiveness of unbundling: To what extent, if any, does unbundling a complex query into smaller relation-focused parts reduce the tendency for error during speech-based querying?

5.1.2 Hypotheses.

H1 - Synonyms and Reduction in Special Characters Improve User Experience. Introducing syntax synonyms and removing the requirement to speak special characters like 'comma', 'parenthesis', or 'asterisk', makes speaking a query feel more natural and improves user experience, and reduces the number of errors and time taken to form simple queries.

H2 - Alternate Ordering Reduces Errors. Relaxing the ordering requirement for various subexpressions reduces the likelihood of ordering-based syntax errors and reduces the amount of time and number of attempts required to dictate a simple or complex query.

H3 - Unbundling Unburdens Working Memory. Breaking a complex, multi-table query into a series of individual single-table queries reduces the query speaker's working memory burden and reduces the likelihood of errors when constructing a query using speech recognition software. This should reduce the number of attempts required to form a correct complex query containing multiple relations.

5.2 Study Protocol

5.2.1 Participants. We primarily targeted UCSD academic programs that provided instruction relating to analytics, and computer and data science. Because the SpeakQL dialect is an extension of the SQL syntax, we sought out participants who were already experienced with SQL in order to reduce the time required for training. Prospective participants were asked to complete a short SQL skills assessment. Those who passed the screening questionnaire were then invited to participate in the experiment. Participants were offered up to 30 USD for their participation. They received a flat rate of six dollars for participation, and an additional two dollars per question answered using both SQL and SpeakQL for up to 12 questions.

5.2.2 Study Design.

Context. Study sessions were conducted online using a video conferencing program and a custom-built browser-based user interface. The user interface enabled participants to view a database schema, receive query prompts, dictate queries through their device's microphone, and review a live speech-to-text transcription
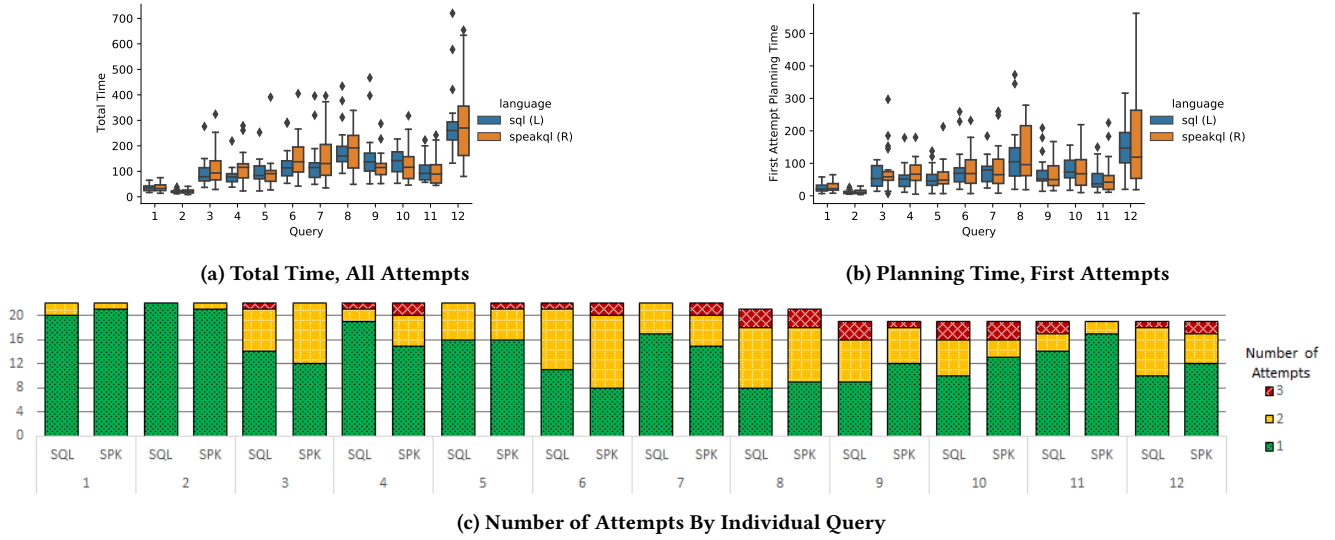
(a) Total Time, All Attempts

(b) Planning Time, First Attempts

(c) Number of Attempts By Individual Query

Figure 6: Performance Comparisons

of query dictation. The research facilitator employed a seperate 'Wizard of Oz' control panel to evaluate attempt correcntess, to provide feedback, to answer SQL- and SpeakQL-related questions, and to manage the overall progression of the study session.

Quantitative Analysis. We evaluated performance using time-based and attempt count dependent variables. Specifically, we measured total completion time per attempt, total completion time for all attempts, planning time (time from viewing query prompt to starting recording), and the number of attempts required to form a correct query.

Independent variables were based on query prompt attributes and feature usage. Prompty attributes were based on the expected complexity of a correct query required to provide an answer to the query prompt. Feature usage determination was based on post-participation transcription of user recordings and subsequent analysis. A more in-depth description of dependent and independent variables is available in the appendix.

Determining Feature Usage. We analyzed feature usage in each query attempt by performing speech-to-text transcription, using Whisper [25] audio and speech processing, on attempt recordings. In order to determine feature usage within the transcripts, we applied syntax-focused heuristics. For example, to determine if unbundling was used by a participant to answer a question, we performed a search for at least one AND THEN within the transcript and coded the attempt based on the presence or absence of the keywords.

Measuring Query Complexity. Query complexity was determined using a series of weighted criteria that include the number of tables (relations) in a query, the number of projections (columns, functions and constants within the select expression), number of functions, number of selections (where expressions), number of joins, and number of modifiers (group by, having, order by and limit). The process used to derive both raw and standardized query complexity scores is available in the appendix.

Managing the Learning Effect. We applied a latin squares approach using counterbalancing to counteract the learning effect [18] inherent in within-subjects studies of two or more treatments. Specifically, participants were divided into two groups: a SpeakQL-to-SQL group and a SQL-to-SpeakQL group, where participants in one group answered all 12 questions in one dialect and then switched to the opposite dialect and answered the same 12 questions again.

5.2.3 Study Session Overview. Sessions were designed to last 90 minutes. Participants were first directed to view a 22 minute training video that provided an overview of SpeakQL features and the schema associated with the query prompts delivered during the session. They were then given an opportunity to ask clarifying questions related to the video. The study facilitator then provided an overview of the query dictation process using the web interface. Participants answered a total of up to 15 questions using both dialects (three practice and 12 non-practice). Participants answered all questions using the first dialect (SpeakQL or SQL), and then repeated the same questions in the second dialect. The research facilitator was available to provide assistance with schema details, as well as both SpeakQL and SQL syntax. Additionally, participants were permitted to take notes during their planning, but were not permitted to use the notes created during the first dialect attempts for their second dialect attempts. Participants were encouraged, but not required, to use as many SpeakQL features as possible.

6 RESULTS AND DISCUSSION

6.1 Participants

Our recruiting efforts yielded contacts from 35 prospective participants. Of the 35 initial respondents, 30 completed the SQL skills assessment survey, and 29 were offered an opportunity to participate. 23 participants ultimately participated in the study. Data from one participant was omitted from analysis due to a violation of the

note transfer policy. Participants were reimbursed up to 30 USD depending upon how many questions they answered in both dialects. 19 Participants completed all 12 questions; and three participants answered at least seven questions in both dialects before opting to terminate the event.

6.2 Hypotheses Evaluation

Hypotheses one and two were feature-focused tests intended to analyze the effect of specific features on dictation performance. Because SpeakQL feature usage was optional, not all participants used all SpeakQL features in all query attempts. We also observed that feature usage was not consistent between participant groups (SpeakQl-first vs. SQL-first), and that participants in the SQL-first group were less likely to use SpeakQL features as often as participants who used SpeakQL as their first dialect. Because of this out-of-balance voluntary usage rate for some features, we were unable to make any significant observations of feature usage effects on dependent variables.

6.2.1 Planning Time. Planning time was a component of H1 and H2 evaluation. We analyzed total number of attempts required to achieve a correct query as well as the amount of time it took for a participant to plan their first attempt. We opted to measure first attempt planning time as opposed to the planning time for the final attempt because our observation was that for second and third attempts, participants generally performed no additional planning. Hence, the time participants used for question evaluation, schema analysis, and query planning was reflected within the first attempt planning time.

We found that the median planning time to prepare for simple queries was greater for SpeakQL attempts than for SQL attempts (38.5 seconds vs. 31.5 seconds), but the difference was not statistically significant (P=0.14). For complex queries, we observed that the median planning time was shorter for SpeakQL than for SQL (66.0 seconds vs. 72.0 seconds), but again the difference was not statistically significant (P=0.295). We also evaluated performance at the individual query level, and found no statistically significant difference in median planning times for SpeakQL and SQL attempts for any query, with P values ranging between 0.07 and 0.48. P values were derived using the Mann-Whitney U Test.

6.2.2 Number of Attempts. The number of attempts required to form a correct query were components of all three hypotheses. We did not observe any statistically significant differences in either mean or median numbers of attempts required. Figure 6c is a distribution plot, by query, of the number of attempts required for each dialect. We note that as queries become more complex, more second and third attempts are required. However, as the session progressed, first attempt answers increased suggesting that participants gained a familiarity with the query dictation process that counteracted the increasing query complexity. Additionally, we observe a slight improvement advantage where first attempt answers were more common using SpeakQL for queries 9 - 12.

6.3 Feature Usage and Participant Feedback

Observed and Self-Reported Feature Usage. Analysis of feature usage revealed varying levels of popularity among SpeakQL
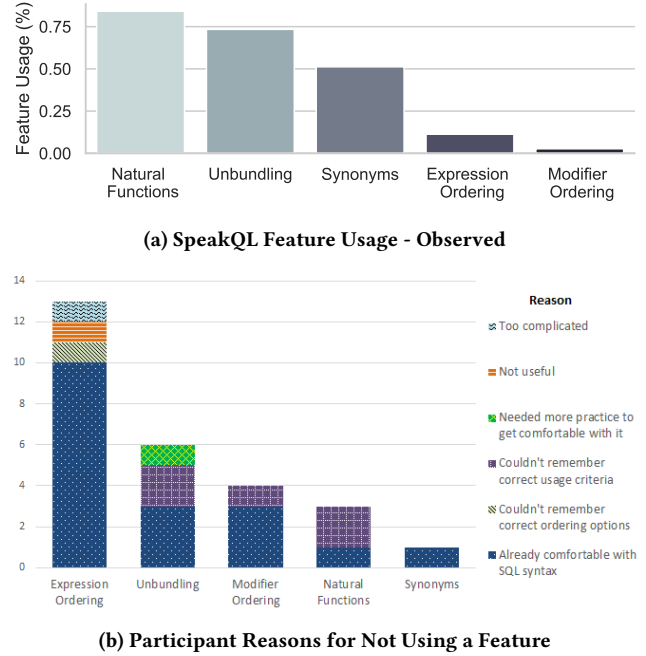

(a) SpeakQL Feature Usage - Observed

(b) Participant Reasons for Not Using a Feature

Figure 7: Feature Usage Observations and Avoidance Reasons

features. SpeakQL feature usage was optional for participants; and some participants relied more heavily on standard SQL syntax than others to complete the study's SpeakQL queries. Figure 7a shows a significant disparity in popularity among the four prototype features. Natural functions were the most-popular SpeakQL feature, whereas expression and modifier ordering were the least popuar features. Self-reported feature usage data (See Figure 7b) is consistent with these findings.

For participants who used a feature, we asked for their perception of its usefulness (Figure 8). Responses from these questions uncover some nuance with regards to the less popular synonym feature: participants preferred punctuation and join synonyms, but did not particularly care for FROM and SELECT synonyms. Responses to other features appear consistent with feature usage observations. Participants who avoided a feature provided at least one reason for avoidance (See Figure 7b); and the predominant reason was due to their high comfort level with SQL syntax.

6.3.1 Survey Feedback.

Thematic Analysis. Using thematic analysis of participant survey feedback, we categorized participant survey feedback and sentiment into positive, negative, and improvement suggestion categories. 13 participants provided some sort of positive feedback, nine provided some sort of negative feedback, and five provided some sort of improvement suggestions. Table 2 provides a count of participants who made comments that were coded within positive, negative, and improvement idea sub-categories.

Positive Comments. Eight participants made positive comments that were not targeted at specific features. These included general impressions of their experience using SpeakQL, a comment that
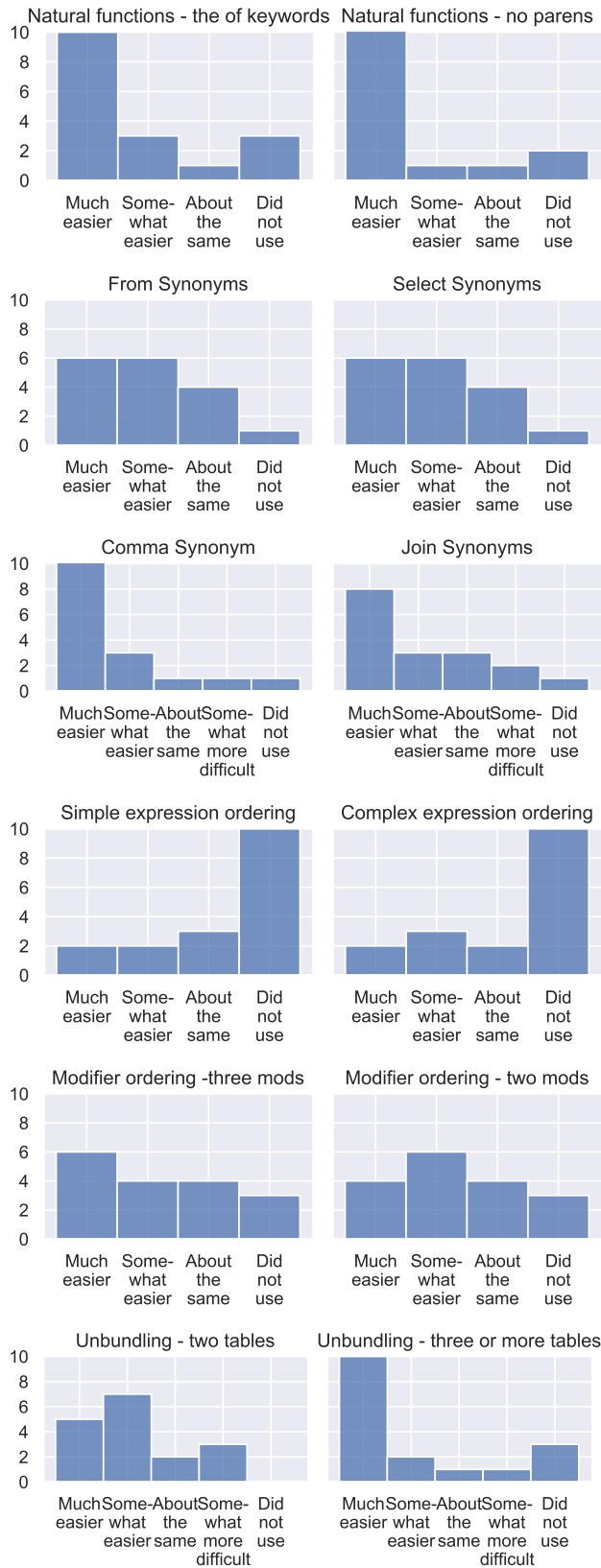
Figure 8: SpeakQL Feature Usefulness Compared to SQL

using SpeakQL was easier than using SQL for dictation, and four comments that SpeakQL had a natural feel and was easy to use.

"SpeakQL definitely makes dictation of queries more natural without worrying a lot about the syntax (which would include the orders) and even the parentheses." -Participant 10 *What I like the most is*

that it is almost like thinking out loud. You just think about what you want to do, and say the query, which makes it way more convenient. -Participant 16

Unbundling received the most feature-specific positive feedback with positive remarks from seven participants. Participants indicated that unbundling required less planning time, feels faster, enables focus on one table at a time, makes complex querying or unbundling easier, and is generally easy or useful.

"Since I did not have to worry about ambiguous column names, I could write the bundles separately faster and join them all later on." -Participant 19

Four participants reported that natural functions were useful and easier to use when dictating than SQL functions requiring parentheses. One participant reported that they found modifier ordering useful.

Negative Comments*.* The most common negative comments were related to syntax difficulty, with nine participants reporting dislike of various aspects of SpeakQL syntax. Four participants reported difficulty with using unbundling for more complex queries. Three indicated that using the unbundling join syntax or non-unbundled queries that used join synonyms was difficult. One participant expressed that the 'group by automatically' feature was not intuitive and felt less natural.

Five participants made negative comments relating to the naturalness of the SpeakQL features. One indicated that SpeakQL gave them a false impression of naturalness that made it difficult to discern when a natural-like statement was a valid SpeakQL query. Some participants provided more specific input including that there were too many synonyms, they were unsure of how expressive SpeakQL was, the language had "too much flexibility", synonyms were unbalanced between expression types, and that it was too nuanced.

"[I] Was afraid of saying the incorrect synonym. Do we really need all these synonyms? How many do we need? Too many synonyms might create the perception of natural language which will cause them to create incorrect queries." -Participant 1 Seven participants

indicated that they did not have enough time to gain familiarity with SpeakQL features and needed more practice.

"Though complex query bundling was easier, it takes a lot of time to go from creating individual table queries, and then joins and then grouping them, and so because I was querying the columns from all the tables together first and then making all the joins in one go, I felt it a little inconvenient. However, I missed the fact that I can simply query 2 tables and join and then go to the next table would have been easier." -Participant 2

Improvement Suggestions*.* Participants provided several suggestions for syntax improvements including allowing for the use of THE before *table* (e.g. *from the table building get buildingname*),

	# Participants
– Positive –	**13**
General Positive Impressions	8
Positive Unbundling Impressions	7
Positive Natural Function Impressions	4
Positive Ordering Impressions	1
– Negative –	**9**
Syntax Difficulty	6
Unfamiliar Language, Needs Practice	7
Faux Natural Language	5
– Improvement Ideas –	**5**
Minimize Punctuation	4
Syntax Improvements	3

Table 2: Thematic Category and Code Frequencies

making generation of the group by expression automatic (not requiring the *group by automatically* expression), and adding *retrieve* as an additional select synonym. Four participants also indicated a general dislike of dictating punctuation symbols including quotes, commas, and parentheses. Their suggestions were focused on reducing the need to verbalize these symbols, particularly quotes, during dictation.

6.4 Discussion of Results

Though our results were not conclusive in terms of statistical power, we find it encouraging that despite SpeakQL being a brand new dialect, there was no degradation in performance when participants used SpeakQL. In fact, we see hints that performance improved vs. SQL performance as the experiment progressed both in terms of planning time and number of attempts (Figure 6c).

Synonyms. Although the simplest to employ, synonyms were only the third-most popular feature when observing actual usage data, and attracted the most negative feedback. Users indicated that synonyms caused uncertainty about the validity of a query's syntax.

Optional Ordering. Optional ordering features were widely unused by all participants. The majority of participants' feedback indicated that their familiarity with SQL was the driving reason behind choosing not to use this feature. However, we elect to retain this feature in its current state as it was not perceived negatively by participants.

Natural Functions and Reduction in Punctuation. The natural function feature was most-popular and universally liked by participants. This feeback leads us to consider seeking additional symbol reduction opportunities including eliminating the need to encapsulate string literals inside quotes when forming where expressions.

Unbundling. In addition to hints at usefulness within the session performance data, we also received majority positive survey feedback relating to unbundling usage. Several users seemed enthusiastic about how unbundling enabled them to think differently about query formulation in a way that made dictating queries easier.

Summary. Considering our feature-based observations, we will likely reduce the keyword synonyms in the dialect, retaining only

synonyms associated with punctuation and symbols. Optional ordering, natural functions, and unbundling were assessed neutrally or positively and should remain as dialect features.

7 RELATED WORK

Natural Language Interfaces. While we envision potential integration of SpeakQL features into NLI systems, the primary focus of SpeakQL is to create a speech-friendly dialect; whereas the purpose of NL-to-SQL and chat-based interfaces are generally to facilitate better typing-based interactions. We discuss NLIs in more detail in section 2.2.

Speech-Based Database Interaction. EchoQuery [17] is a stateful query-by-voice system that allows users to interact with a relational database in a conversational manner. It is a system implementation demo, whereas this paper presents a novel dialect assessed by way of an A/B user study. EchoQuery also employs a unique speech-friendly dialect; but its syntax represents a subset of full SQL expressiveness (e.g. joins are not explicitly defined, disjunctive predicates are not possible) whereas SpeakQL extends SQL and retains its expressive power.

Our prior work on SpeakQL 1.0 [30] is a Speech + touch query system that allows users to modify query intent using a novel SQL keyboard. SpeakQL 1.0 employs a subset of SQL as the query interface language whereas SpeakQL 2.0 (this paper) evaluates a dialect prototype for its potential to replace standard SQL as SpeakQL's interface dialect.

CiceroDB [32] is a system that leverages Google Assistant to infer user query intent from natural language. CiceroDB's main contribution is toward the verbal articulation of large result sets using computer-generated voice. While orthogonal to SpeakQL work, the main focus of our SpeakQL paper is on dialect evaluation whereas the focus of CiceroDB is on result set articulation.

Spoken Programming as Assistive Technology. Spoken programming systems have been considered and evaluated as assistive technologies for programmers with motor impairments. Prior experiments were unconvincing due to weak speech recognition capabilities [5]. Improved ASR has re-kindled interest; and usability interviews with motor impaired programmers suggest a vision for a natural language-based programming system that can overcome challenges relating to dictating symbols and variables [22]. The SpeakQL dialect user study survey responses support similar conclusions–that there is a desire for natural-like dialects that reduce the need for symbol dictation.

8 CONCLUSIONS AND FUTURE WORK

We continue to work toward a natural yet still unambiguous dialect optimized for speech-based database interaction. Our evaluation of the our dialect prototype in this paper offers promising direction for future refinement and integration of SpeakQL into a speech-based database interaction system. We envision this system to be stateful and interactive, allowing users to converse with the system to clarify and expand upon their initial queries. We also plan to investigate the suitability of SpeakQL as an intermediate representation of natural language-to-SQL transformations, where SpeakQL is employed as a human-in-the-loop feedback mechanism for correctness evaluation.

REFERENCES

[1] U.S. Bureau of Labor Statistics 2020. *Survey of Occupational Injuries and Illnesses Data.* U.S. Bureau of Labor Statistics. Retrieved November 14, 2022 from https://www.bls.gov/iif/nonfatal-injuries-and-illnesses-tables.htm

[2] 2022. grammars-v4. https://github.com/antlr/grammars-v4.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[4] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. 2015. A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (Vilnius, Lithuania) *(ITiCSE '15).* Association for Computing Machinery, New York, NY, USA, 201–206. https://doi.org/10.1145/2729094.2742620

[5] A. Begel and S.L. Graham. 2006. An Assessment of a Speech-Based Programming Environment. In *Visual Languages and Human-Centric Computing (VL/HCC'06).* 116–120. https://doi.org/10.1109/VLHCC.2006.9

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165

[7] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. 1976. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM J. Res. Dev.* 20, 6 (Nov. 1976), 560–575. https://doi.org/10.1147/rd.206.0560

[8] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Ann Arbor, Michigan) *(SIGFIDET '74).* Association for Computing Machinery, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515

[9] LTC Jason Crist. 2021. From the Server to the Battlefield, How data Scientists are Key to Winning Future Conflicts. https://www.army.mil/article/249036/from_the_server_to_the_battlefield_how_data_scientists_are_key_to_winning_future_conflicts. Accessed: 2023-01-20.

[10] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021.* Association for Computational Linguistics, Punta Cana, Dominican Republic, 2030–2042. https://doi.org/10.18653/v1/2021.findings-emnlp.174

[11] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2022. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* (nov 2022). https://doi.org/10.1145/3571730 Just Accepted.

[12] Lilong Jiang, Michael Mandel, and Arnab Nandi. 2013. GestureQuery: A Multitouch Database Query Interface. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1342–1345. https://doi.org/10.14778/2536274.2536311

[13] Brad Kelechava. 2020. The SQL standard - ISO/IEC 9075:2016 (ANSI X3.135). https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/#gref

[14] Hyeonji Kim, Byeong Hoon So, Wook Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment* 13 (6 2020), 1737–1750. Issue 10. https://doi.org/10.14778/3401960.3401970

[15] Tobias Kuhn. 2014. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics* 40, 1 (03 2014), 121–170. https://doi.org/10.1162/COLI_a_00168 arXiv:https://direct.mit.edu/coli/article-pdf/40/1/121/1812691/coli_a_00168.pdf

[16] Kyle Luoma and Arun Kumar. Accessed January 31, 2023. *Tech Report: Design and Evaluation of an SQL-Based Dialect for Spoken Querying.* https://adalabucsd.github.io/papers/TR_2023_SpeakQL_Dialect.pdf.

[17] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. 2016. Making the case for query-by-voice with echoquery. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 26-June-2016, 2129–2132. https://doi.org/10.1145/2882903.2899394

[18] I. Scott MacKenzie. 2013. *Human-Computer Interaction: An Empirical Research Perspective* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. 2022. Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study. *ACM Inroads* 13, 1 (feb 2022), 52–65. https://doi.org/10.1145/3514214

[20] Mehdi Mohammadpoor and Farshid Torabi. 2020. Big Data analytics in oil and gas industry: An emerging trend. *Petroleum* 6, 4 (2020), 321–328. https://doi.org/10.1016/j.petlm.2018.11.001 SI: Artificial Intelligence (AI), Knowledge-based Systems (KBS), and Machine Learning (ML).

[21] Jesse Mu and Advait Sarkar. 2019. Do We Need Natural Language? Exploring Restricted Language Interfaces for Complex Domains. In *37th Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI '19 Extended Abstracts).* ACM. https://www.microsoft.com/en-us/research/publication/do-we-need-natural-language-exploring-restricted-language-interfaces-for-complex-domains/

[22] Sadia Nowrin, Patricia OrdóñEz, and Keith Vertanen. 2022. Exploring Motor-Impaired Programmers' Use of Speech Recognition. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility* (Athens, Greece) *(ASSETS '22).* Association for Computing Machinery, New York, NY, USA, Article 78, 4 pages. https://doi.org/10.1145/3517428.3550392

[23] OpenAI. [2022]. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/.

[24] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '14).* Association for Computing Machinery, New York, NY, USA, 579–598. https://doi.org/10.1145/2660193.2660202

[25] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust Speech Recognition via Large-Scale Weak Supervision. https://doi.org/10.48550/ARXIV.2212.04356

[26] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. *OpenAI* (2018).

[27] P. Reisner. 1977. Use of Psychological Experimentation as an Aid to Development of a Query Language. *IEEE Transactions on Software Engineering* SE-3, 3 (May 1977), 218–229. https://doi.org/10.1109/TSE.1977.231131

[28] Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin. 1975. Human Factors Evaluation of Two Data Base Query Languages: Square and Sequel. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition* (Anaheim, California) *(AFIPS '75).* Association for Computing Machinery, New York, NY, USA, 447–452. https://doi.org/10.1145/1499949.1500036

[29] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 9895–9901. https://aclanthology.org/2021.emnlp-main.779

[30] Vraj Shah, Side Li, Arun Kumar, and Lawrence Saul. 2020. SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2020), 2363–2374. https://doi.org/10.1145/3318464.3389777 School: UCSD.

[31] Kil Soo Suh and A. Milton Jenkins. 1992. A Comparison of Linear Keyword and Restricted Natural Language Data Base Interfaces for Novice Users. *Information Systems Research* 3, 3 (1992), 252–272. https://doi.org/10.1287/isre.3.3.252 arXiv:https://doi.org/10.1287/isre.3.3.252

[32] Immanuel Trummer. 2020. Demonstrating the Voice-Based Exploration of Large Data Sets with CiceroDB-Zero. *Proc. VLDB Endow.* 13, 12 (sep 2020), 2869–2872. https://doi.org/10.14778/3415478.3415496

[33] Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. UnifiedSKG: Unifying and Multi-Tasking Structured Knowledge Grounding with Text-to-Text Language Models. https://doi.org/10.48550/ARXIV.2201.05966

A BACKGROUND

A.1 SQL Grammar

All SQL grammars include syntax rules for both *data definition language* (DDL) and *data manipulation language* (DML) statements. DDL statements are intended to enable specification of data structure; and DML statements enable data access and update functions [3]. Data analysts, informaticists, and other data consumers generally make use of DML statements to fulfill data requirements. DDL statements are generally expressed by database administrators and software developers responsible for designing, implementing, and maintaining data models within database management systems.

B DIALECT

B.1 Synonyms

Keyword synonyms are listed in Figure 9.

B.2 Natural Functions

Natural function syntax and examples are portrayed in Figure 10.

B.3 Unbundling

B.3.1 Additional Unbundling Example.

Query Prompt. Find the titles of all courses offered in terms with the year 2022.

SpeakQL. **join the** term **table with the** room **table on** term dot id equals courseoffering dot termid
and then join the courseoffering **table with the** course **table on** courseoffering dot courseid equals course dot id
and then show me title **in the** course **table**
and then from room **where** floor equals 3 **select average** capacity
and then get nothing **from** term **where** year equals 2022

```
SQL: Select title from course
    join courseoffering
        on course.id = courseoffering.courseid
    join term
        on courseoffering.termid = term.id
    where year = 2022;
```

C IMPLEMENTATION

C.1 Grammar

Figures 19 and 18 are excerpts of SpeakQL and SQL select statement grammar in a modified BNF format. The '?' symbol appended to a rule indicates that the rule is optional. The '*' symbol indicates that 0 to many occurences of a rule are allowed. Symbols or tokens surrounded by double quotes indicate that the content enclosed within the quotes is a terminal symbol. Both figures contain only a subset of child and descendant rules and both are limited to two levels of depth below the *selectStatement* rule, and certain SQL rules including *union*, *window*, and *into* are omitted for the sake of brevity. Figure 20 is an example of a child expression to the SpeakQL select statement grammar defined in figure 19 and is provided as an example implementation of the extension strategy that facilitates SpeakQL to SQL translation. Additional examples are available in this paper's corresponding technical report.

C.2 SpeakQL to SQL Translation

The translator performs translation through a series of steps. The first step takes a valid SpeakQL query as input, where correctness is determined upstream of this task, and sends it to the Java-based SpeakQL parser via an HTTP post request. The parser responds with a Lisp-formatted abstract syntax tree AST that the Python-based translator transforms into an editable AST. The translator performs operations on the SpeakQL AST in order to transform it into a valid SQL query which include finding and replacing SpeakQL synonyms with SQL syntax, reordering expressions, reordering modifier items, and transforming unbundled queries into a single SQL select statement. The unbundling transformation step includes additional substeps including inference of a group by expression, consolidation of select elements (i.e. columns, functions, and scalars), consolidation of where statements, consolidation of join expressions, and finally consolidation of any table sources not included within join expressions.

C.3 Parser Performance

A comparison of the SQL and SpeakQL grammar implementations reveals that SpeakQL feature implementation results in deeper trees with additional branches. Naturally, this increase in complexity will result in a corresponding increase in time required to parse a query. A performance comparison between three parsers, a MySQL grammar-based parser, a full SpeakQL parser that extends the entire MySQL grammar, and a simple SpeakQL parser that extends only a DML subset of the MySQL grammar was conducted to validate assumptions about parser performance improvements for smaller grammars.

We measured parse time for each parser using a set of 8,946 SQL queries (see Figure 11). The full SpeakQL parser, on average, exhibited a 5.9x *slowdown* compared to the MySQL parser, suggesting that the modularity-based rule extensions negatively impact parser performance. On the other hand, the simple SpeakQL parser extending only a DML subset of MySQL rules exhibited a 2.6x *speedup* compared to the MySQL parser and a 15.1x *speedup* compared to the full SpeakQL parser. This performance increase is achieved by sacrificing certain SQL features including *window* and *union*. This rule omission is mitigated using a fallback strategy where SpeakQL queries identified by the translator as containing syntax not covered by the Simple SpeakQL parser are passed back to the full SpeakQL parser for processing.

C.4 SpeakQL to SQL Translation

Translating Unbundled Queries. The translator uses the same approach for each bundling step, which is to identify the first expression parser rule node in the AST for a given expression and use it as a migration target for additional expression parser rule nodes that may exist in the AST. For example, if a query contains two separate unbundled select-from-where queries connected with a join query, the translator will designate the first of the two *selectExpression* parser rules in the AST as *selectExpression'* and will migrate the *selectElements* that are children of the second *selectExpression* rule node to become children of *selectExpression'*. Parser rule node migration is accomplished by appending the migrated parser rule

Syntax Keyword Synonyms and Phrases

SQL Keyword: **SELECT**

Retrieval Words: **SELECT** **FIND** **RETRIEVE** **GET**

Presentation Words: **SHOW ME** **DISPLAY** **PRESENT**

Question Words: **WHAT IS** **WHAT IS THE** **WHAT ARE** **WHAT ARE THE**

SQL Keyword: **FROM**

Synonyms: **FROM** **FROM TABLE** **FROM TABLES** **IN TABLE** **IN TABLES**

Optional Phrases: **FROM THE** **IN THE** / TableName / **TABLE**

Syntax Keyword Synonyms and Phrases

SQL Symbol: **, (Comma)**

Two Options: **, (Comma)** **AND**

SQL Keyword: **JOIN**

Present: **JOIN** **JOIN TABLE** **JOIN WITH TABLE**

Progressive: **BY JOINING** **BY JOINING TABLE** **BY JOINING WITH TABLE**

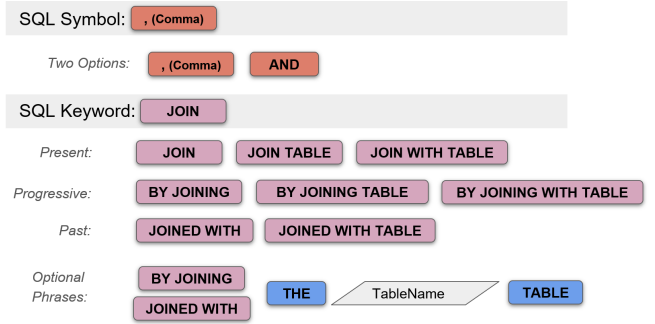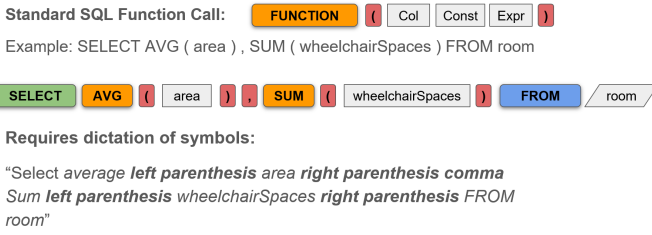*Past:* **JOINED WITH** **JOINED WITH TABLE**

Optional Phrases: **BY JOINING** **JOINED WITH** **THE** / TableName / **TABLE**

Figure 9: SpeakQL Synonyms

SQL Functions

Standard SQL Function Call: **FUNCTION** **(** Col Const Expr **)**

Example: SELECT AVG (area) , SUM (wheelchairSpaces) FROM room

SELECT **AVG** **(** area **)** **,** **SUM** **(** wheelchairSpaces **)** **FROM** / room

Requires dictation of symbols:

"Select *average* **left parenthesis** *area* **right parenthesis comma** *Sum* **left parenthesis** *wheelchairSpaces* **right parenthesis** *FROM room*"

SpeakQL Natural Functions

Parentheses symbols are not required for functions with only a single column or constant as a parameter.

FUNCTION Col Const   **FUNCTION** **(** Expr **)**

'THE' and 'OF' keywords may surround the function keyword.

THE **COUNT** **OF** id   **THE** **AVERAGE** area   **SUM** **OF** units

Parentheses symbols are still required if the function contains an expression

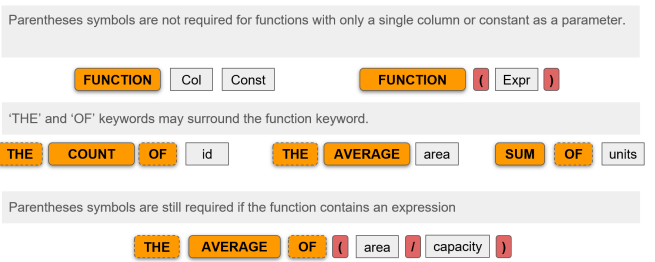**THE** **AVERAGE** **OF** **(** area **/** capacity **)**

Figure 10: Natural Functions
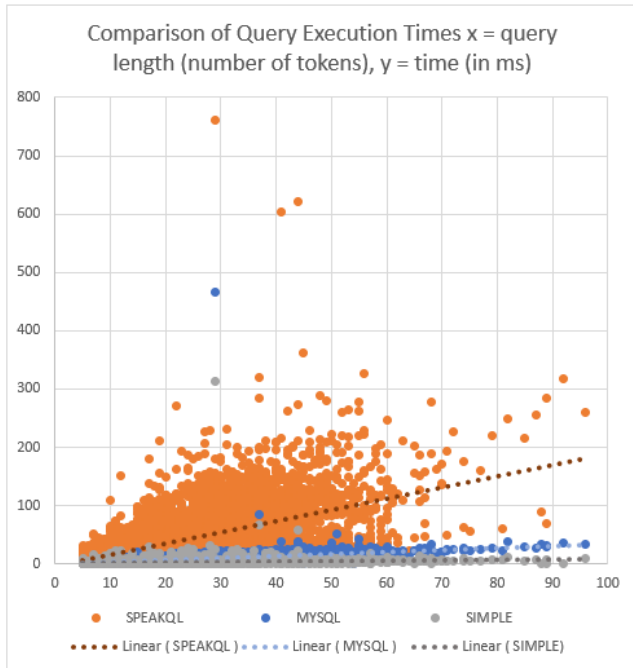
Figure 11: Grammar Parser Performance Comparison

Algorithm 1 Find and Replace Synonyms

$speakQlAst \leftarrow parseSpeakQL(query)$
$synonymKwRuleset \leftarrow \{selectKW, fromKW, joinKW, ...\}$
$syntaxSugarRs \leftarrow \{theKW, tableKW, ofKW, isKW\}$
$synonymHashMap \leftarrow \{(selectKW : SELECT), (fromKW : FROM), ...\}$
for all *nodes* such that $node \in speakQlAst$ **do**
  if $node.rule \in synonymKwRuleset$ **then**
    $node.replaceKW(synonymHashMap[node.rule])$
  end if
  if $node.rule \in syntaxSugarRs$ **then**
    $speakQlAst.removeNode(node)$
  end if
end for

---

node's ID to the migration target rule node's child list and removing the migrated parser rule node's ID from the migration source rule node's child list.

Inferring the group by expression. A feature of the SpeakQL translator is its ability to infer a group by expression using the presence of aggregator functions and column references within multiple *selectExpression* parser rules. A user can instruct the translator to perform group by expression inference using the *group by automatically* keyword within either a single-table SpeakQL query or a multi-table unbundled query.

Algorithm 2 Reorder Select, From and Where Expressions

$speakQlAst \leftarrow parseSpeakQL(query)$
for all $ruleNodes$ such that $ruleNode \in speakQlAst$ **do**
  if $ruleNode == qryOrdSpec$ **then**
    $expr \leftarrow ruleNode.children$
    $reorderedExpr \leftarrow [\varnothing]$
    $reorderedExpr.append(expr.selExpr)$
    $reorderedExpr.append(expr.tabExpr)$
    $reorderedExpr.append(expr.whereExpr)$
    $ruleNode.children \leftarrow reorderedExpr$
  end if
end for

Algorithm 3 Reorder Select Modifier Items

$speakQlAst \leftarrow parseSpeakQL(query)$
for all $ruleNodes$ such that $ruleNode \in speakQlAst$ **do**
  if $ruleNode == selModExpr$ **then**
    $reorderedItms \leftarrow [\varnothing]$
    $grpByCls \leftarrow ruleNode.getChildByRule(grpByCls)$
    $havingCls \leftarrow ruleNode.getChildByRule(hvingCls)$
    $ordrByCls \leftarrow ruleNode.getChildByRule(ordrByCls)$
    $limitCls \leftarrow ruleNode.getChildByRule(limitCls)$
    $reorderedItms.append(grpByCls)$
    $reorderedItms.append(havingCls)$
    $reorderedItms.append(ordrByCls)$
    $reorderedItms.append(limitCls)$
    $selModExpr.children \leftarrow reorderedItms$
  end if
end for

Algorithm 4 Natural Function Transformation

$speakQlAst \leftarrow parseSpeakQL(query)$
$syntaxSugarRs \leftarrow \{theKW, ofKW\}$
for all $ruleNode \in speakQlAst.nodesWithName(natFun)$ **do**
  for all $child \in ruleNode.children$ **do**
    if $child.rule \in syntaxSugarRs$ **then**
      $speakQlAst.removenode(child)$
    end if
  end for
  $functionArgs \leftarrow ruleNode.getChildWithName(funArgs)$
  $speakQlAst.surroundNodeWithParens(functionArgs)$
end for

Algorithm 5 Transform Unbundled Query (High Level)

$speakQlAst \leftarrow parseSpeakQL(query)$
Require: $\exists unbndQryOrdSpc \in speakQlAst$
$speakQlAst.inferGroupBy()$
$speakQlAst.bundleSelectElements()$
$speakQlAst.bundleWhereStatements()$
$speakQlAst.bundleJoinParts()$
$speakQlAst.bundleTables()$
for all $ruleNodes$ such that $ruleNode \in speakQlAst$ **do**
  if $ruleNode \in \{qryOrdSpec, multQryOrdSpec, exprDelim\}$
  then
    $speakQlAst.removeNode(ruleNode)$
  end if
end for

Algorithm 6 Unbundled Function: Infer GroupBy

$speakQlAst \leftarrow parseSpeakQL(query)$
Require: $\exists grpByCls \in speakQlAst \wedge \exists autoMtcKW \in speakQlAst$
$autoKws \leftarrow speakQlAst.nodesWithName(autoMtcKW)$
for all $autoKw \in autoKws$ **do**
  $speakQlAst.removeNode(autoKw)$
end for
$gbNode \leftarrow speakQlAst.nodeWithName(grpByCls)$
$elmtsByTbl \leftarrow speakQlAst.getAllTablesAndElements()$
$gbItms \leftarrow [\varnothing]$
for all $selElmt \in elmtsBtTbl$ **do**
  if $selElmt$ is $column$ **then**
    $gbItems.append(selElmt)$
    $gbItems.append(delim)$
  end if
end for
$gbNode.children \leftarrow gbItms$

When the translator encounters the *group by automatically* instruction, it proceeds as depicted in algorithm 6. It makes use of AST helper functions *nodeWithName* which returns a list of all nodes with a given rule name, and *getAllTablesAndElements* which returns a Python dictionary ({ *table name* : [*selectElements*]}) that contains a list of select elements and function calls for each table referenced within the SpeakQL query. The translator uses the existing *groupByExpression* parser rule node that contains the *auomaticGroupByKeyword* parser rule and replaces the *automaticGroupbyKeyword* rule with a *groupByItems* parse rule. The *groupByItems* parse rule node serves as the parent node for individual *groupByItem*

and *groupByItemDelimiter* nodes generated from non-function *selectElement* nodes registered in the table-element python dictionary. Because the *groupByItems* parse rule already exists beneath the *selectModifierExpression* parse rule, no further AST transformations are required; and the group by inference operation is complete after the generation of the *groupByItems* parse rule and its children *groupByItem* and *groupByItemDelimiter* parse rules.

Bundling select elements. Select element bundling is a required step for any SpeakQL query that makes use of the unbundling feature. Because any SpeakQL query that uses unbundling must have at least two *selectExpression* parse rules, select element bundling is a mandatory step of the bundling process.

As shown in algorithm 7 as *selElmtsRlNd'*, which abbreviates *selectElementsRuleNode*, the translator uses the first *selectElements* parser rule as a migration target for all other select elements within the SpeakQL query. Also depicted in the same algorithm is a preemptive method for avoiding ambiguity where the translator prepends each migrated select element with its associated table, resulting in a dotted id in the format *tableName.columnName*. This step is

Algorithm 7 Unbundled Function: bundleSelectElements

$speakQlAst \leftarrow parseSpeakQL(query)$
for all $ruleNode \in speakQlAst.nodesWithName(nothingElmt)$
do
   $ruleNode.rename(selElmt)$
end for
$elmtsByTbl \leftarrow speakQlAst.getAllTablesAndElements()$
$selElmtsRlNd' \leftarrow speakQlAst.nodesWithName(selElmts)[0]$
$newChildren \leftarrow [\varnothing]$
for all $tblItm \in elmtsByTbl$ **do**
   for all $selectElmt \in elmtsByTbl[tblItm]$ **do**
     $newChildren.append(tblItm.selectElmt)$
   end for
end for
$selElmtsRlNd'.children \leftarrow newChildren$
for all $ruleNode \in speakQlAst.nodesWithName(selElmts)$
such that $ruleNode! = selElmtsRlNd'$ **do**
   $speakQlAst.removeNode(ruleNode)$
end for

Algorithm 8 Unbundled Function: bundleWhereStatements

$speakQlAst \leftarrow parseSpeakQL(query)$
Require: $\exists whereExpr \in speakQlAst$
$unbdlExprs \leftarrow speakQlAst.nodesWithName(unbdlQryOrdSpc)$

$unbdlExpr' \leftarrow unbdlExprs[0]$
$whereExpr' \leftarrow unbdlExpr'.nodeWithName(whereExpr)$
for all $exprNode \in unbdlExprs[1:]$ **do**
   $tblName \leftarrow exprNode.nodeWithName(tblName)$
   $exprWhereExpr \leftarrow exprNode.nodeWithName(whereExpr)$

   $exprWhereKw \leftarrow exprWhereExpr.nodeWithName(whereKW)$

   $exprWhereExpr.removeNode(exprWhereKw)$
end for

---

performed for both standalone column references and column references as arguments within function expressions. After iterating through each *selectElementExpression* parser rule and migrating their associated *selectElements* children to *selectElementsRuleNode'*, a cleanup operation removes them from the AST leaving a single *selectElementsExpression* in the AST that contains all column and function references present in the original SpeakQL unbundled query.

Bundling where expressions. The where expression bundling step is optional, and is only invoked if at least one bundled query expression contains a *whereExpression* parse rule. When performing where expression bundling, the SpeakQL translator makes the following assumptions:

- All cross-table predicates are conjunctive (and)
- Multiple predicate expressions within a single unbundled query should be encased in parentheses before consolidation

Given these assumptions, where expression bundling proceeds as depicted in algorithm 8.

Similar to the select element bundling step, the first occurence of a *whereExpression* parse rule, *whereExpression'*, within the query AST and designates it as the target for additional *whereExpression* parse rule migration. Because where expressions are optional within *selectStatement* parser rules, it is possible that *whereExpression'* is not a member of the first *selectStatement* parse rule, *selectStatement'*, in the query. Because of this possibility, the translator may perform an additional migration step where it migrates *whereExpression'* to become a child of *selectStatement'*.

The translator takes advantage of the recursive nature of the where expression grammar (see figure **??**) by designating where expressions that exist in *selectStatement* parse rules that are not *selectStatement'* as children of *whereExpression'*. During this migration process, the translator employs helper functions to surround migrated expressions with parentheses and deliminating the migrated expressions with *andKeyword* parser rules. The end result

of this process is a single cross-table conjunctive *whereExpression* parser rule that contains all other *whereExpression* parser rules specified in other unbundled queries within the SpeakQL query.

Bundling join parts. Join part bundling collects all *multiJoinExpression* parse rules and aggregates them in an arbitrary order to form a valid SQL join expression. Join part bundling is an optional feature because an alternate form of relation joining where all table sources are specified within *fromExpression* parse rules, and join conditions between each table source are specified within *whereExpression* predicates (e.g. *from table one, table two where one.id = two.id*), is also possible. Currently, because join order in the resulting SQL query is arbitrary within the bounds of its implementation rules, join part bundling is limited to allowing only inner joins. Queries that require outer join expressions may still be expressed using other SpeakQL features; but may not employ the unbundling feature. Outer join capability is a future SpeakQL feature development objective.

The join part bundling process begins when the translator creates a list of all *multiJoinExpression* parse rule nodes within the AST. Given this list, it performs iterative analysis on each expression to determine if a subquery exists as a table item within any of the *multiJoinExpression* nodes, and if so, substitutes the subquery with a subquery masking rule and alias in the join expression. The translator then checks the left and right table reference parse rules within the join expression to determine if an alias exists for each table, or if the table is referenced in the join expression by its alias. If an alias association is encountered, it creates *asKeyword* and *multiJoinTableAlias* parse rules and adds them as children to the target *joinExpression* parser rule that represents the objective SQL-correct join expression.

After analyzing all *multiJoinExpression* and making modifications toward SQL-correct syntax, the translator begins the join consolidation process by designating the first table referenced in the queries first *selectStatement'* as the base table upon which the SQL-correct join statement will be built. In order to ensure valid join expression chaining, the translator determines a table join order that ensures that each subsequent table added to a join expression references a table in its join condition that already exists within the objective join expression. In other words, while building

the SQL-correct join expression, a *multiJoinExpression* will not be added to the target *joinExpression* until all tables referenced in its join condition have been added to the target *joinExpression*. The process will continue iterating over remaining *multiJoinExpressions* until the table existence constraint can be satisfied and all expressions have been added to the objective SQL-correct expression, or it is determined that *multiJoinExpressions* exist in the SpeakQL query that cannot be chained and the translator responds with an error condition. The translator then performs an additional safety check to ensure that all tables referenced in the query's *unbundledQueryOrderSpecification* parser rules are also referenced in a corresponding *multiJoinExpression* parser rule, and throws an error if a violation is encountered.

After the join bundling process is completed, all join expressions consolidated within a single *multiJoinExpression* are migrated to the first *tableExpressionNoJoin* parse rule. The *tableExpressionNoJoin* parse rule name is then updated to *tableExpression* and becomes a valid SQL from + join expression; and join bundling is complete.

Bundling table expressions. The final step of the bundling process involves collection of all *tableExpressionNoJoin* parse rule nodes within the AST. This is only required in cases where an unbundled query contains multiple table references but has no associated *multiJoinExpression* parser rules. In this case the first *tableExpressionNoJoin*, designated as *tableExpressionNoJoin'*, is established as the expression migration target, and additional *tableExpressionNoJoin* rules that exist as children of additional *selectStatement* rules are added to *tableExpressionNoJoin'*. If a where expression exists in any *(selectStatement)* that defines a join condition between two tables within the query, the expression consolidation occurs during the previously described where expression bundling step. If no where condition is defined between two tables within the query, the translator still performs table expression bundling, and the result is a SQL statement that produces a cartesian product of the two tables.

Syntax tree cleanup. After all relevant bundling steps are complete, the translator performs syntax tree cleanup by removing parse rules from which their child sub expressions have been migrated. Upon completion of tree cleanup, the SpeakQL to SQL translation process is complete, and serialization of the abstract syntax tree's terminal nodes results in a valid SQL statement equivalent to the input SpeakQL query.

D USER STUDY

D.1 Initial Design and Protocol

D.1.1 Quantitative Analysis. We evaluated performance using the dependent variables:

- Total-Time-to-Completion: Lower is better. Total time (in seconds) from when the participant is presented with the question prompt to when they press the submit button for their final query submission.
- First Attempt Planning Time: Lower is better. Time (in seconds) from when the participant is presented with the question for the first time to when they press the record button.

<selExpr> ::=	<selKW> <selSpec>* <selElmts>
<selKW> ::=	"select" | "find" | "retrieve" | "get" | "show me" | "display" | "present" | "what is" | "what is the" | "what are" | "what are the"
<tabExpr> ::=	<frmKW> <tblSrcs>
<frmKW> ::=	"from" | "from table" | "from tables" | "in table" | "in tables"
<joinKW> ::=	"join" | "join table" | "by joining" | "by joining table" | "joined with" | "join with" | "joined with table" | "join with table" | "by joining with table"
<onKW> ::=	"on"
<theKW> ::=	"the"
<tblKW> ::=	"table"

Figure 12: SpeakQL Synonym Keywords Grammar Excerpt

<qrySpec> ::=	<qryOrdSpec> <selModExpr> | (<multiJoinExpr exprDelim>)? (<selModExpr> <exprDelim>)? <ubndQryOrdSpc> (<exprDelim> (<ubndQryOrdSpc> | <multiJoinExpr>)) (<exprDelim> <selModExpr>)?
<ubndQryOrdSpc> ::=	<selExpr> <whereExpr>? <tabExprNoJoin> | <selExpr> <tabExprNoJoin> <whereExpr>? | <tabExprNoJoin> <selExpr> <whereExpr>? | <tabExprNoJoin> <whereExpr>? <selExpr>
<tabExprNoJoin> ::=	<frmClsNoJoin>
<frmClsNoJoin> ::=	<frmKW> <theKW>? <tblSrcNoJoin> <tblKW>?
<tblSrcNoJoin> ::=	<tblSrcItm> | "(" <tblSrcItm> ")"
<mltiJnExpr> ::=	<mltiJnPrt> (<exprDelim> <mltiJnPrt>)*
<mltiJnPrt> ::=	<mltiInnrJn> | <mltiOutJn> | <mltiNatJn>
<exprDelim> ::=	"and then" | "then" | "next"
<mltiInnrJn> ::=	<inJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")?
<mltiOutJn> ::=	<joinDir> <outJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")
<mltiNatJn> ::=	<natJoinKW> (<joinDir> <outJoinKW>)? <joinKW> <tblSrcItm> <withKW> <tblSrcItm>
<withKW> ::=	"with" | "with table" | "and"

Figure 13: SpeakQL Unbundled Query Grammar Excerpt

<whereExpr> ::=	<whereKW> <expr>
<whereKW> ::=	"where"
<expr> ::=	("not" | "!") <expr>
	| <expr> <logicOp> <expr>
	| <predicate> "is" "not"? ("true" | "false" | "unknown")
	| <predicate>
<logicOp> ::=	"and" | "xor" | "or"
<predicate> ::=	<predicate> "not"? "is" "in" <leftParen> <selStmt> <rightParen>
	| <predicate> "is" "not" "null"
	| <predicate> <compareOp> <predicate>
	| <predicate> <compareOp> ("all" | "any" | "some") <leftParen> <selStmt> <rightParen>
	| <predicate> "not"? "between" <predicate> "and" <predicate>
	| <predicate> "not"? "like" <predicate>
	| (expressionAtom)

Figure 14: SpeakQL Where Expression Grammar Excerpt

<tabExpr> ::=	<frmKW> <tblSrcs>
<frmKW> ::=	"from" | "from table" | "from tables" | "in table" | "in tables"
<tblSrcs> ::=	<theKW>? <tblSrc> <tabKW>? (<delim> <theKW>? <tblSrc> <tblKW>?)*
<tblSrc> ::=	<tblSrcItm> <joinPart>*
	| "(" <tblSrcItm> <joinPart> ")"
<tblSrcItm> ::=	"(" <slctStmt> ")"
	| <tblName> <tblAlias>?
	| "(" <tblSrcs> ")"
<joinPart> ::=	<inJoin> | <outJoin> | <natJoin>
<inJoin> ::=	<inJoinKW>? <joinKW> <tblSrcItm> (<onKW> <expr>)
<inJoinKW> ::=	"inner" | "cross"
<outJoin> ::=	<joinDir> <outJoinKW>? <joinKW> <tblSrcItm> (<onKW> <expr>)
<joinDir> ::=	"left" | right
<outJoinKW> ::=	"outer"
<natJoin> ::=	<natJoinKW> (<joinDir> <outJoinKW>)? <joinKW> <tblSrcItm>
<natJoinKW> ::=	"natural"
<joinKW> ::=	"join" | "join table" | "by joining" | "by joining table" | "joined with" | "join with" | "joined with table" | "join with table" | "by joining with table"
<onKW> ::=	"on"
<theKW> ::=	"the"
<tblKW> ::=	"table"

Figure 15: SpeakQL Table (From) Expression Grammar Excerpt

<selModExpr> ::=	<selModItm>? <selModItm>? <selModItm>? <selModItm>?
<selModItm> ::=	<grpByCls> | <havingCls> | <orderByCls> | <limitCls>
<grpByCls> ::=	<grpByKW> <grpByItm> (<delim> <grpByItm>)* ("with" "rollup")?
<grpByKW> ::=	"group by" | "group"
<grpByItm> ::=	<grpByExpr> <order> | <autoMtcKW>
<autoMtcKW> ::=	"automatic" | "automatically"
<grpByExpr> ::=	"not" <grpByExpr>
	| <predicate> "is" "not"? ("true" | "false" | "unknown")
	| "predicate"
<delim> ::=	"," | "and"
<havingCls> ::=	<havingKW> <expr>
<havingKW> ::=	"having"
<ordrByCls> ::=	"order by" <ordrByExpr> (<delim> <ordrByExpr>)
<ordrByExpr> ::=	<expr> <order>?
<order> ::=	<ascKW> | <descKW>
<ascKW> ::=	"asc" | "ascending"
<descKW> ::=	'desc' | 'descending'
<limitCls> ::=	'limit' <limitClsAtm>
<limitClsAtm> ::=	(decimalLiteral | simpleId)

Figure 16: SpeakQL Select Modifier Expression Grammar Excerpt

<funCall> ::=	<aggrFun> | <noParAggrFun> | <nonAggrWinFun> | <sclrFunNam>
<aggrFun> ::=	<theKW>? ("avg" | "average" | "max" | "min" | "sum") <ofKW>? "(" ("all" | "distinct")? **<funArg>** ")"
	| <theKW>? "count" <ofKW>? "(" ("*" | "all"? **<funArg>** | "distinct" **<funArg>**) ")"
	("std" | "std dev" | "std dev pop" | "std dev samp" | "var pop" | "var sample" | "variance") "(" "all"? **<funArg>** ")"
	"group concat" "(" "distinct"? **<funArg>** ("order by" <ordByExp> (<delim> <ordByExp>)*) ")"
<funArg> ::=	(<constant> | <colName> | <funCall> | <expr>) (<delim> (<constant> | <colName> | <funCall> | <expr>))*
<noParAggrFun> ::=	<theKW>? ("avg" | "average" | "max" | "min" | "sum") <ofKW>? ("all" | "distinct")? **(<constant> | <colName>)**
	| <theKW>? "count" <ofKW>? ("*" | "all"? <funArg> | "distinct" **(<constant> | <colName>)**)

Figure 17: SpeakQL Natural Function Expression Grammar Excerpt

\<selStmt> ::=	\<qrySpec>
\<qrySpec> ::=	"select" \<selSpec>* \<selElmts> \<fromCls>? \<grpByCls>? \<havingCls>? \<ordrByCls>? \<limitCls>?
\<fromCls> ::=	("from" \<tblSources>?) ("where" \<expression>)?
\<grpByCls> ::=	"group by" \<grpByItm> ("," \<grpByItm>)?
\<havingCls> ::=	"having" \<expression>
\<ordrByCls> ::=	"order by" \<ordrByExpr> ("," ordrByExpr)
\<ordrByExpr> ::=	\<expression> ("asc" | "desc")
\<limitCls> ::=	"limit" (\<limitClsAtom> ",")? \<limitClsAtom>
	| "limit" \<limitClsAtom> "offset" \<limitClsAtom>

Figure 18: SQL Select Statement Grammar Excerpt

\<selStmt> ::=	\<qrySpec>
\<qrySpec> ::=	\<qryOrdSpec> \<selModExpr>
	| (\<multiJoinExpr exprDelim)? (\<selModExpr> \<exprDelim>)? \<ubndQryOrdSpc> (\<exprDelim> (\<ubndQryOrdSpc> | \<multiJoinExpr)) (\<exprDelim> \<selModExpr>)?
\<qryOrdSpec> ::=	\<selExpr> \<whereExpr>? \<tabExpr>
	| \<selExpr> \<tabExpr> \<whereExpr>?
	| \<tabExpr> \<selExpr> \<whereExpr>?
	| \<tabExpr> \<whereExpr>? \<selExpr>
\<ubndQryOrdSpc> ::=	\<selExpr> \<whereExpr>? \<tabExprNoJoin>
	| \<selExpr> \<tabExprNoJoin> \<whereExpr>?
	| \<tabExprNoJoin> \<selExpr> \<whereExpr>?
	| \<tabExprNoJoin> \<whereExpr>? \<selExpr>

Figure 19: SpeakQL Select Statement Grammar Excerpt

\<selExpr> ::=	\<selKW> \<selSpec>* \<selElmts>
	| \<selKW> \<nothingElmt>
\<selKW> ::=	"select" | "find" | "retrieve" | "get" | "show me" | "display" | "present" | "what is" | "what is the" | "what are" | "what are the"
\<selSpec> ::=	"all" | "distinct" | "distinctrow
\<selElmts> ::=	("*" | \<selElmt>) (\<delim> \<selElmt>)*
\<delim> ::=	"," | "and"
\<selElmt> ::=	"*" | \<colName> | \<funCall>
\<nothingElmt> ::=	\<nothingKW>
\<nothingKW> ::=	"nothing"

Figure 20: SpeakQL Select Expression Grammar Excerpt

- Dictation (recording) Time: Lower is better. Time (in seconds) from when the participant presses the record button to when they press the stop button.
- Number of Attempts: Lower is better. Discrete (1, 2, 3) inicating how many attempts the participant made to form a correct query.

And the independent variables:

- Query Complexity: Continuous positive real number derived from the presence and quantity of columns, tables, joins, functions and predicates.
- Query is Complex: Discrete (0, 1) simpler method for indicating if a query is complex. Based on complexity threshold defined below.
- Query Attributes: Number of projections, number of tables, number of selections, number of joins, and number of modifiers required to answer the question.
- Features Used: One-hot encoded vector representing the SpeakQL features actually used by a participant for a given SpeakQL query.

D.1.2 Determining Query Complexity. Query complexity was defined using a set of weights associated with query attributes including number of tables and joins, number of column references, number of function calls, and number of modifiers (i.d. limit, order by, etc.).

$$WeightedScore = \sum_{n=1}^{n} AttrCt_n * AttrWt_n$$

$$Standardize(WeightedScore) = \frac{WeightedScore - \mu_{Scores}}{\sigma_{Scores}}$$

$$Complexity = Standardize(WeightedScore)$$

Where $AttrWt$ is a vector length n of weights (see Table 3) associated with values in vector $AttrCt$ of length n and the standardization function computes a value between -1 and 1 using the complexity score of a query minus the mean divided by the standard deviation of all query complexity scores.

D.2 Interface

D.2.1 Data Collection. Data generated by the study included time to plan (time from updating the query prompt until pressing record), time to record (time from pressing record to stopping the recording), number of attempts per query, dictation transcript, and audio recordings of each dication attempt. All data was captured within participants' browser sessions and uploaded to a web server and database for processing and analysis during the session. In an effort to add explanatory power to our quantitative results, we asked participants to complete a voluntary post-participation survey.

D.3 Session Sequence

The objective time to complete a study session was 90 minutes. Participants were informed prior-to beginning that it was possible that a session may exceed 90 minutes, and that they could opt to end the session prior to completing all queries.

The objective session schedule:

- 0-23: Training video
- 23-25: Q&A + UI orientation
- 25-35: Practice
- 35-61: Q 1-12 Dialect 1
- 61-64: Intermission
- 64-90: Q 1-12 Dialect 2
- 90+: Debrief and survey

We employed a latin squares counterbalancing strategy to mitigate learning effect by randomly assigning participants to one of two groups (i.e. SpeakQL first, then SQL or SQL first, then SpeakQL).

D.3.1 Training. Participants received SpeakQL dialect training by way of a 22 minute video that introduced the four SpeakQL features under evaluation as well as the schema used for all study questions.

The video is hosted at:

https://www.youtube.com/watch?v=JEtoFtjNqew.

D.4 Schema

The database schema referenced in all study query prompts was designed to represent a likely data model used for scheduling classes. We elected to employ a simple university catalogue model because our target participant population was university students who would be familiar with the concepts represented in the schema.
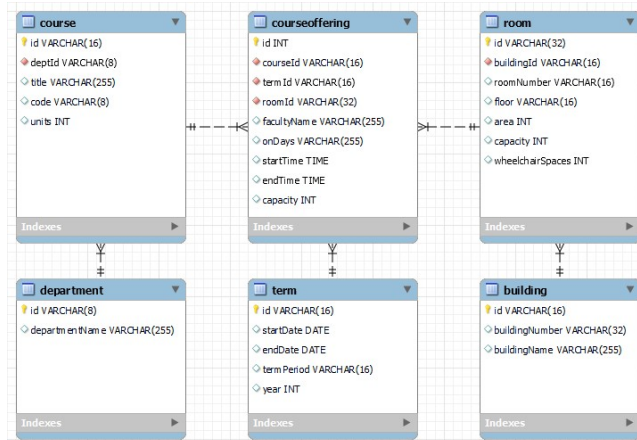

Figure 21: User Study Schema

The schema (Figure 21) consists if six tables:

- department(id, departmentName)
- course(id, deptId, title, code, units)
- courseoffering(id, courseId, termId, roomId, facultyName, onDays, startTime, endTime, capacity)
- room(id, buildingId, roomNumber, floor, area, capacity, wheelchairSpaces)
- term(id, startDate, endDate, termPeriod, year)
- building(id, buildingNumber, buildingName)

The university data model resembles a snowflake schema where the *courseoffering* table joins together *course*, *term*, and *room* to represent a course offering during a specific term (e.g. Fall 2021), and the room and building in which it is offered.

D.5 Queries

The query prompts, possible correct answers, and complexity (raw weighted scores and standardized scores) can be viewed in tables 8 and 9.

Attribute	Weight
Number of Tables	0.8
Number of Columns	0.25
Number of Functions	1.0
Number of Selections (Where Predicates)	0.8
Number of Joins	1.2
Number of Modifiers	1.0

Table 3: Query Complexity Attribute Weights

E USER STUDY DATA

E.1 Statistical Analysis

Based on the non-parametric properties of our results, we used the Mann-Whitney U test to evaluate statistical significance of the experiment.

For three dependent variables *first planning time*, *total time*, *number of attempts*, we separate the data into simple and complex query results in table 4.

	P Val	SQL Median	SpeakQL Median
First Planning Time - Simple	0.144	31.5	38.5
First Planning Time - Complex	0.295	72.0	66.0
Recording Time - Simple	0.214	13.5	15.5
Recording Time - Complex	0.122	32.0	34.0
Total Time - Simple	0.222	55.5	55.8
Total Time - Complex	0.336	137	131

Table 4: Mann Whitney U Test Results for Simple and Complex Queries

We provide a query-level view of the same dependent variables in tables ??, 6, and 7.

Query	P Val	SQL Median	SpeakQL Median
1	0.247740	20.5	22.0
2	0.348923	10.5	11.0
3	0.336262	53.0	58.5
4	0.069565	51.5	66.5
5	0.380095	45.5	48.5
6	0.340583	69.5	68.5
7	0.481270	79.0	65.0
8	0.420244	105.0	96.0
9	0.175026	52.0	50.0
10	0.470904	73.0	68.0
11	0.330556	37.0	42.0
12	0.330713	147.0	119.0

Table 5: Mann Whitney U Test Results by Query - First Planning Time

Figure 22: Simple and Complex Query Planning Time

Query	P Val	SQL Median	SpeakQL Median
1	0.495315	34.5	32.5
2	0.420752	20.0	20.5
3	0.259258	78.5	93.5
4	0.012856	78.0	115.5
5	0.319349	83.5	91.0
6	0.140103	113.5	137.5
7	0.148052	115.0	130.5
8	0.329868	160.0	192.0
9	0.046561	137.0	115.0
10	0.206796	142.0	116.0
11	0.206694	92.0	89.0
12	0.476718	260.0	270.0

Table 6: Mann Whitney U Test Results by Query - Total Time

Query	P Val	SQL Median	SpeakQL Median
1	0.002310	10.0	7.5
2	0.001484	10.0	6.0
3	0.179456	16.5	19.0
4	0.147531	24.0	24.0
5	0.302412	21.0	20.0
6	0.318916	26.0	25.0
7	0.012078	29.0	38.5
8	0.224763	32.0	34.0
9	0.385004	34.0	34.0
10	0.488340	33.0	35.0
11	0.447685	30.0	32.0
12	0.047982	66.0	79.0

Table 7: Mann Whitney U Test Results by Query - Recording Time

For recording time, the statistically significant differences in the first two queries are likely due to the use of natural functions,

as both query prompts require a function reference to derive the correct answer. Recording time for query 12 is likewise significant because the query requires at least four join expressions to form a correct answer. This means that the verbosity of unbundling results in longer recording times for this query.

We also evaluated performance in terms of differences and percent differences for each SpeakQL-SQL pair of queries performed by each participant (see Figure 23). That is, we calculated the difference in time between a participant's SpeakQL attempt and SQL attempt for each query and generated a raw difference as well as a percent difference statistic for each.

E.2 Learning Effect

We employed a latin squares method with counterbalanced groups. Group one answered all 12 queries using SpeakQL first, then answered the same 12 queries using SQL. Group two answered using SQL first, then SpeakQL.
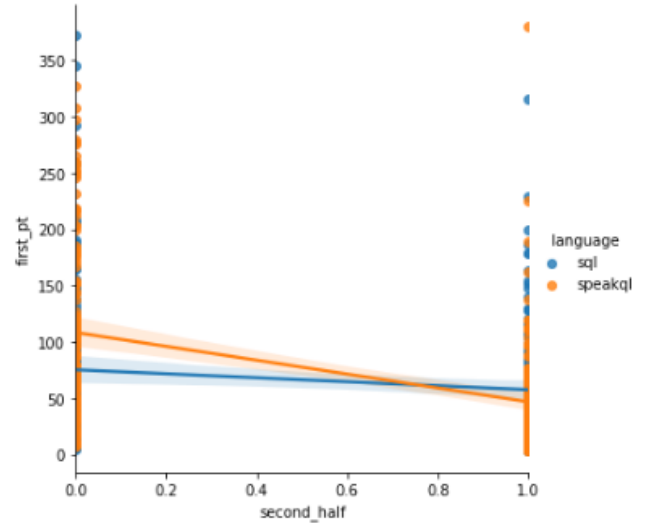
We analyzed the dependent variable results to assess any learning effect that may have occurred in the results and observe convergent learning between the first and second half of the session for both dialects, suggesting that some dialect-specific asymmetric learning occurred [18] that affected planning time. The intersection of SpeakQL and SQL lines in Figure 24 suggest that participants who were in the group that used SpeakQL in the second half of the experiment benefitted from some learning factor present in the first half of the experiment that participants who were in the opposite group did not. Though it is uncertain exactly why this effect occurred, we believe that participant familiarity with SQL allowed them to become more comfortable with the general query dictation process using the study interface, which may have lead to a more rapid adoption of the process. Conversely, participants who were in the SpeakQL first group may have had a harder time becoming more comfortable with the general dictation process while they were also grappling with the use of the relatively unfamiliar SpeakQL dialect features.

Another possible explanation of the asymmetric learning observed in Figure 24 is the tendency for participants in the SQL first group to be more avoidant of SpeakQL features. In these avoidance situations, participants would have been more likely to benefit
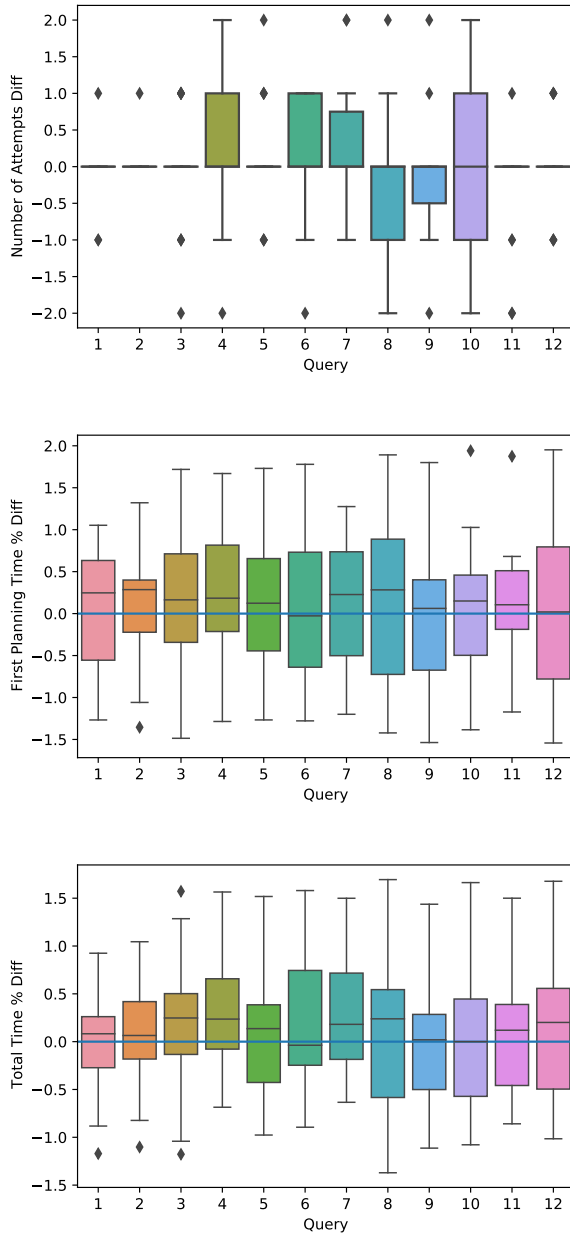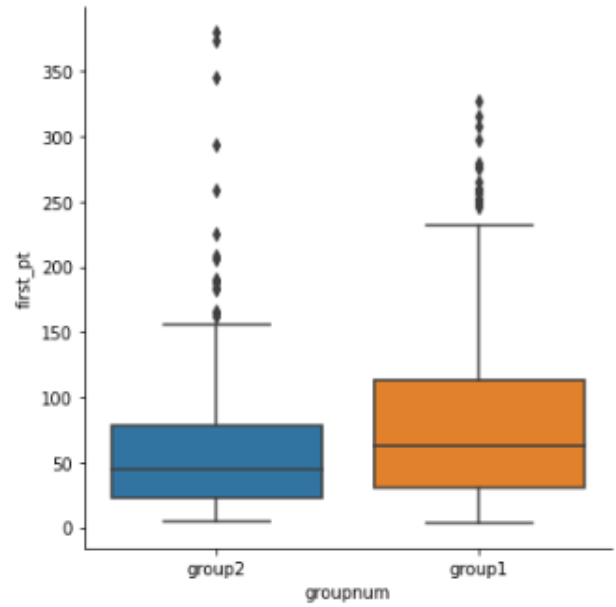
Figure 23: Participant SpeakQL-SQL attempt differences by query

Figure 24: Learning Effect - SpeakQL Planning Time Performance Improvement

Figure 25: Imbalance Between SpeakQL->SQL (Group 1) and SQL->SpeakQL (Group 2) first planning time performance. Group 1 median: 62.0; Group 2 median: 44.5; Mann-Whitney U Test P Value: 0.00007

from the practice gained in the first half of the session because their SpeakQL queries would have more closely resembled a standard SQL query.
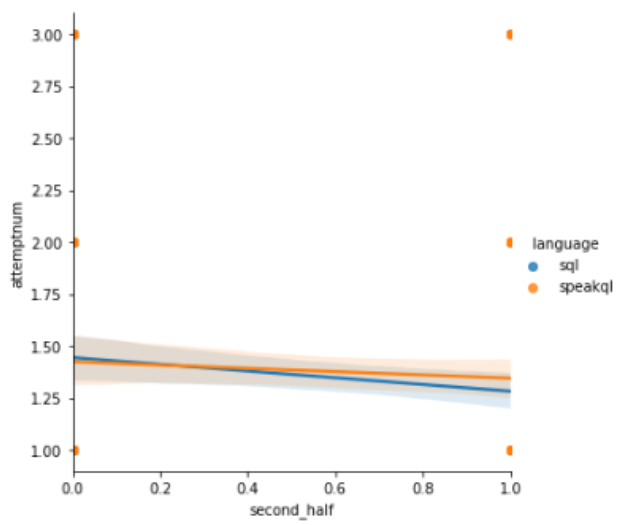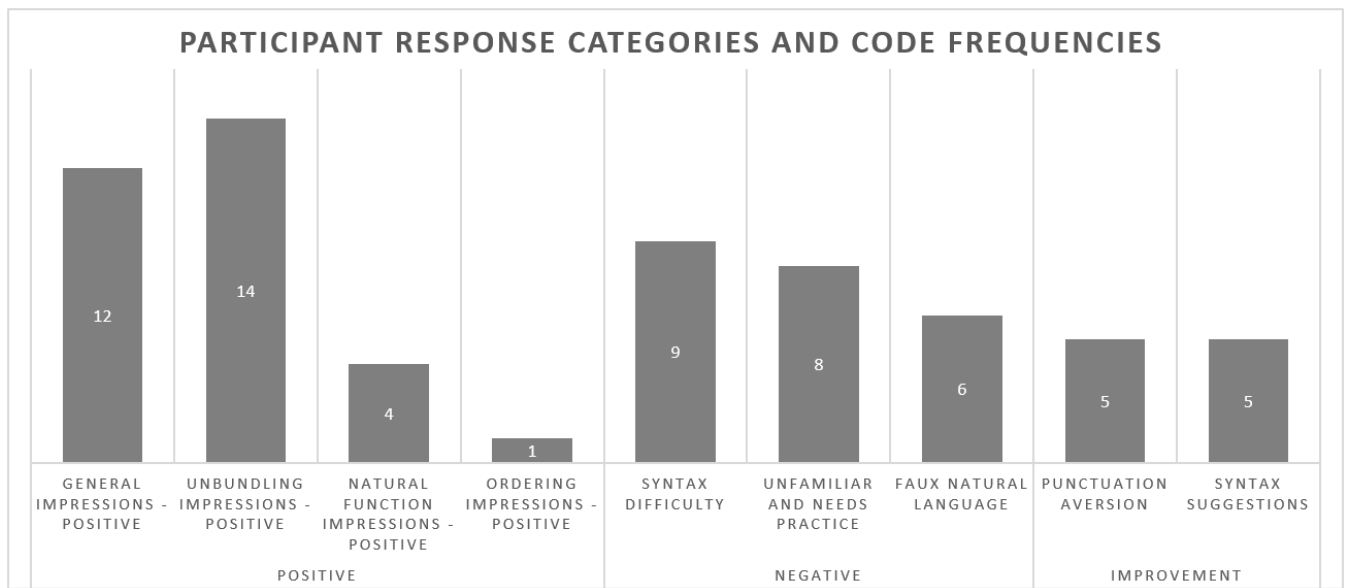
For number of attempts (see Figure 26) we observe little-to-no asymmetric learning as indicated by the nearly parallel lines that represent dialect performance in the first and second half of the experiments.

Figure 26: Learning Effect - SpeakQL Attempt Number Performance Improvement

F SURVEY FEEDBACK

Figure 27: Thematic Analysis

F.1 Survey Feedback With Coding

This section contains all survey free-text comments by participant. Coding annotations were added during analysis and are preceded by the # symbol.

F.1.1 Participant 1. **Was there anything else that you liked about your experience using the SpeakQL dialect?**
The localized predicates and projections made it easier to start speaking stuff rather than just think for too long;

#userstudy-unbundling-less-planning-needed

whereas for SQL had to think and collect items for different tables.

#userstudy-sql-planning-burden

For SpeakQL could just look at a table and collect items and then move onto another table.

#userstudy-unbundling-one-table-at-a-time

The count of/ the average of is much easier than count(),avg() when we are talking.

#userstudy-natural-functions-easier-when-talking

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?
All the synonyms was daunting, too many synonyms.
#userstudy-too-many-synonyms

Was afraid of saying the incorrect synonym. Do we really need all these synonyms? How many do we need? Too many synonyms might create the perception of natural language which will cause them to create incorrect queries.

#userstudy-false-natural-language-problem

When there are more than two tables to join, the unbundling syntax is hard to use.

#userstudy-complex-unbundling-syntax-difficult

F.1.2 Participant 2. **Do you have any suggestions for other expression reordering features that would make dictating queries easier?**
I am curious if we can reorder the groupby, having and order by statements just like the initial select.

#userstudy-suggest-modifier-reordering

Do you have any suggestions for other function dictation features that would make dictating queries easier?
None, I felt this was really good.

#userstudy-natural-functions-easier-when-talking

Do you have any suggestions for other complex query unbundling features that would make dictating queries easier?
Though complex query bundling was easier,

#userstudy-complex-unbundling-easier

it takes a lot of time to go from creating individual table queries, and then joins and then grouping them, and so because I was querying the columns from all the tables together first and then making all the joins in one go, I felt it a little inconvenient.

#userstudy-complex-unbundling-syntax-difficult

However, I missed the fact that I can simply query 2 tables and join and then go to the next table would have been easier. Definitely would suggest to pass this instruction to the next participant.

#userstudy-unbundling-order-options-good
#userstudy-need-practice

What changes could make you more likely to use the modifier ordering feature?
I didn't realize this existed. One of my previous suggestions is the same as this. So I am glad that SpeakQL has this feature as well.

#userstudy-modifier-reording-good

Was there anything else that you liked about your experience using the SpeakQL dialect?
It is easier to create queries if construction can happen from Natural Language,

#userstudy-general-positive-feedback

I really like this project. And also [the research facilitator] was patient and I felt the entire flow was good.

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?
Yes, using the JOIN WITH
#userstudy-join-synonyms-difficult
and mentioning the QUOTES when comparing string fields.
#userstudy-dislike-punctuation-dictation -dictation

F.1.3 Participant 3. **Was there anything else that you liked about your experience using the SpeakQL dialect?**
Much cooler way of using SQL
#userstudy-general-positive-feedback

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?
At times speaking the exact words like comma was a bit confusing, but comes easy with practice

#userstudy-dislike-punctuation-dictation
#userstudy-need-practice

F.1.4 Participant 4. #userstudy-nofeedback

F.1.5 Participant 5. **What changes could make you more likely to use the natural function feature?**
I think it's a very useful feature
#userstudy-natural-functions-useful
and I missed using it because of unfamiliarity with SpeakQL and strong familiarity/usage in SQL
#userstudy-sql-familiarity-less-speakql-use

F.1.6 Participant 7. #userstudy-nofeedback

F.1.7 Participant 8. **Do you have any suggestions for other keywords that would make dictating queries easier?**
Sometimes I tend to say "THE TABLE course" for example, instead of "THE course TABLE".

#userstudy-suggestion-synonym-the-table

What changes could make you more likely to use the expression ordering feature?
Not really, I guess it's just I'm used to the SQL syntax.
#userstudy-sql-familiarity-less-speakql-use

Do you have any suggestions for other function dictation features that would make dictating queries easier?
Omit left/right for cases without nested parenthesis, which I'd say happens much more often? But maybe this provides too little benefit.

#userstudy-dislike-punctuation-dictation
#userstudy-suggestion-punctuation-no-leftright-paren

Do you have any suggestions for other complex query unbundling features that would make dictating queries easier?

Is it possible to make "GROUP BY automatically" automatically?
#userstudy-suggestion-syntax-groupby-automatically-automatic

Do you have any suggestions for other expression modifier-related features that would make dictating queries easier?

Maybe move having clauses into where clauses, especially in unbundled queries, and then auto-detect it and move to a SQL HAVING clause, if possible.
#userstudy-suggestion-syntax-unbundling-having-with-where

Was there anything else that you liked about your experience using the SpeakQL dialect?

No particular things, but I guess just in general the feel of using it in spoken context is much easier and natural.
#userstudy-general-positive-feedback
#userstudy-natural-feel-easier

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

I tend to get used to language syntax, especially well-designed languages, and thus I don't like having too much flexibility in the case of JOIN for example. But I guess this was done for NLP purposes, so I'm not objecting it 100
#userstudy-too-much-flexibility
#userstudy-join-synonyms-difficult

F.1.8 Participant 6. **Do you have any suggestions for other keywords that would make dictating queries easier?**

Select synonyms - Retrieve
#userstudy-suggestion-synonym-retrieve

What changes could make you more likely to use the expression ordering feature?

If i use it more frequently that i have optional order i might start using it
#userstudy-need-practice
or atleast don't have to worry about ordering especially in bigger queries
#userstudy-ordering-might-use-with-practice

Do you have any suggestions for other function dictation features that would make dictating queries easier?

It seems intuitionally correct. So, good work capturing that
#userstudy-natural-functions-useful

Do you have any suggestions for other complex query unbundling features that would make dictating queries easier?

Although it was helpful everytime to use 'And then' but i found it repetitive so not sure how to include this feedback to suggestion
#userstudy-unbundling-and-then-repetitive
#userstudy-complex-unbundling-syntax-difficult

Do you have any suggestions for other expression modifier-related features that would make dictating queries easier?

Automatically is great option but felt less intuitional when speaking naturally
#userstudy-automatically-less-intuitive-less-natural

Was there anything else that you liked about your experience using the SpeakQL dialect?

It's certainly great step towards being intuitional speaking-wise in the basic SQL utlities.
#userstudy-general-positive-feedback

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

Not sure if things like quotes, distinct , other joins (left,right,full outer) are handled already. From experience pvov, Quotes if not handled is a bit of a major issue if not handled already.
#userstudy-unsure-of-expressiveness

F.1.9 Participant 10. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

SpeakQL definitely makes dictation of queries more natural without worrying a lot about the syntax (which would include the orders) and even the parentheses.
#userstudy-general-positive-feedback
#userstudy-natural-feel-easier

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

Found the usage of quote unnecessary, it would be great if SpeakQL takes care of this as well.
#userstudy-dislike-punctuation-dictation

F.1.10 Participant 12. #userstudy-nofeedback

F.1.11 Participant 11. **Do you have any suggestions for other keywords that would make dictating queries easier?**

The join synonyms can use some structural changes. Coming from SQL to SpeakQL, the join statement took me the longest to get used to.
#userstudy-join-synonyms-difficult

Was there anything else that you liked about your experience using the SpeakQL dialect?

I liked how it is highly structured similar to the SQL syntax, many features such as unbundling definitely improved the ease of usage compare to SQL.
#userstudy-structure-good
#userstudy-unbundling-ease-of-use
#userstudy-easier-than-sql
#userstudy-easy-to-use

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

It takes quite a bit to get used to.
#userstudy-need-practice
Some statement seems a little out of place compare to the commonly used statement (ex: SELECT has many synonyms I can use, while LIMIT, GROUP BY doesn't, so I struggled to find the correct wording)
#userstudy-synonym-imbalance

F.1.12 Participant 13. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

It is easy to use
#userstudy-easy-to-use
and provides an efficient unbundling feature to break down complex queries.
#userstudy-unbundling-ease-of-use

F.1.13 Participant 14. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

Complex queries became easier
#userstudy-complex-querying-easier

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

Slight worries about the language nuances

#userstudy-language-nuance-concern

F.1.14 Participant 16. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

What I like the most is that it is almost like thinking out loud. You just think about what you want to do, and say the query, which makes it way more convenient.

#userstudy-thinking-out-loud

#userstudy-unbundling-less-planning-needed

The ungrouping feature is particularly useful.

#userstudy-unbundling-useful

F.1.15 Participant 17. #userstudy-nofeedback

F.1.16 Participant 19. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

I could really understand the need for the bundling feature while using the SpeakQL dialect when I was formulating queries with multiple joins and same columns in multiple tables.

#userstudy-unbundling-no-ambiguous-columns

Since I did not have to worry about ambiguous column names, I could write the bundles separately faster and join them all later on.

#userstudy-unbundling-feels-faster

Using SQL, I would need to be more aware of the aliases of the columns and the tables while writing the query.

#userstudy-complex-unbundling-easier

F.1.17 Participant 20. **Was there anything else that you liked about your experience using the SpeakQL dialect?**

Interesting to speak SQL

#userstudy-general-positive-feedback

Was there anything else that you DID NOT like about your experience using the SpeakQL dialect?

Unbundling is slightly difficult to interpret in the first time

#userstudy-need-practice

#userstudy-complex-unbundling-syntax-difficult

No.	Prompt	SQL Example	SpeakQL Example	Complexity
P-1	which term years are present in the database?	SELECT DISTINCT year FROM term	in the term table get distinct year	W: 1.05 S: -1.28
P-2	how many rooms have at least two wheelchair spaces?	select count(*) as "Number of Rooms"" from room where wheelchairSpaces >= 2"	find the count of roomNumber where wheelchairSpaces >= 2	W: 2.85 S: -0.69
P-3	during which terms is the course with id 'anth1' taught by faculty named 'Jane Doe'? Include the term year and term period in the result.	SELECT term.id, term.termPeriod, term.year FROM term INNER JOIN courseOffering ON term.id = courseOffering.termId INNER JOIN course ON courseOffering.courseId = course.id WHERE course.id = 'anth1' AND facultyName = 'Jane Doe'	get year and termperiod from the term table AND THEN get nothing from the course table where id = 'anth1 AND THEN get nothing from the courseoffering table where facultyname = 'Jane Doe' AND THEN join course with courseoffering on course.id = courseoffering.id AND THEN join courseoffering with the term table on term.it = courseoffering.termid	W: 8.50 S: 1.29
Q-1	how many buildings does UCSD have?	SELECT COUNT(*) FROM building	what is the count of buildingNumber in the building table	W: 2.05 S: -0.95
Q-2	How many deparments does UCSD have?	SELECT COUNT(*) FROM department	get the count of distinct departmentname from the department table	W: 2.05 S: -0.95
Q-3	make a list of courses offered by the CSE department that includes the course title and units columns. The CSE department ID is 'CSE'.	SELECT course.title, course.units FROM course WHERE course.deptId = 'CSE'	find title and units in the course table where deptId = 'CSE'	W: 2.10 S: -0.94
Q-4	Display the building names for buildings that have a room with roomnumber 2001.	SELECT building.buildingName FROM building JOIN room ON building.id = room.buildingId WHERE room.roomNumber = '2001'	from the room table get nothing where roomnumber = 2001 AND THEN from the buildingtable get buildingname AND THEN join room with building on room.buildingid = building.id	W: 3.85 S: -0.94
Q-5	How many buildings have a room with at least three wheelchair spaces?	SELECT COUNT(DISTINCT buildingId) FROM room WHERE wheelchairSpaces >= 3	what is the count of buildingname in the building table joined with the room table on room.buildingid = building.id where wheelchairspaces >= 3	W: 2.85 S: -0.69
Q-6	what is the sum total number of wheelchair spaces in each building? Show the building name and the number of spaces in each building in the result.	SELECT buildingName, SUM(wheelchairSpaces) AS totalWheelchairSpaces FROM building JOIN room ON building.id = room.buildingId GROUP BY buildingName ORDER BY totalWheelchairSpaces DESC	show me the sum of wheelchair spaces in the room table AND THEN in the building table get buildingname AND THEN join room with building on room.buildingid = building.id AND THEN group automatically	W: 5.05 S: 0.02
Q-7	Find the titles of all courses offered in terms with the year 2022.	SELECT title FROM course JOIN courseOffering ON course.id = courseOffering.courseId JOIN term ON courseOffering.termId = term.id WHERE year = 2022	get title from course AND THEN get nothing from term where year = 2022 AND THEN join course with courseoffering on course.id = courseoffering.id AND THEN join courseoffering with the term table on term.it = courseoffering.termid	W: 5.85 S: 0.28
Q-8	Find the five departments that have the the highest count of courses and display them in descending order. Include department name in the result.	SELECT departmentName, COUNT(courseId) FROM courseOffering JOIN department ON department.id = courseOffering.deptId GROUP BY departmentName ORDER BY COUNT(courseId) DESC LIMIT 5	what is the count of id in the course table AND THEN show me departmentName in department AND THEN join department with course on course.deptID = department.id AND THEN group automatically limit 5 order by departmentName descending	W: 6.05 S: 0.35 W: Weighted S: Standardized

Table 8: User Study Queries (Practice - Query 8)

No.	Prompt	SQL Example	SpeakQL Example	Complexity
Q-9	Generate a list of departments that offer more than 100 courses. Include the count of all courses associated with the department, and the department name in the result.	SELECT departmentName, COUNT(*) AS numCourses FROM department JOIN course ON department.id = course.deptId GROUP BY departmentName HAVING COUNT(*) > 100	from department get departmentname AND THEN from course get the count of id AND THEN group automatically having count of course.id >= 100 AND THEN join the course table with the department table on course.deptid = department.id	W: 6.05 S: 0.35
Q-10	Make a list that shows the building name, building number, and average room area for buildings that have rooms with an average area that is greater than 1000	SELECT building.buildingName, building.buildingNumber, AVG(room.area) FROM building JOIN room ON building.id = room.buildingId GROUP BY building.id HAVING AVG(room.area) > 1000	show me buildingName and buildingNumber in the building table AND THEN get nothing from room AND THEN group automatically having average room.area > 1000 AND THEN join building with room on building.id = room.buildingId	W: 6.30 S: 0.43
Q-11	list the three buildings with the highest average room area. Include the building name and building number in the result.	select buildingNumber, buildingName, avg(area) as avgArea from building join room on building.id = room.buildingId group by buildingNumber, buildingName order by avgArea desc limit 3	show me buildingname in the building table AND THEN show me the average area in the room table AND THEN join the room table with the building table on room.buildingid = building.id AND THEN order by average area descending limit 3 group automatically	W: 7.30 S: 0.75
Q-12	what are the titles of the classes being held in the building 'York Hall' in room with room number 2622 during the fall 2020 term? Include the course title in the result. NOTE: the termPeriod column in the term table contains the values 'fall', 'spring', 'winter', and 'summer'.	SELECT title FROM courseOffering INNER JOIN course ON courseOffering.courseId = course.id INNER JOIN room ON courseOffering.roomId = room.id INNER JOIN building ON room.buildingId = building.id INNER JOIN term ON courseOffering.termId = term.id WHERE building.buildingName = 'York Hall' AND room.roomNumber = '2622' AND term.termPeriod = 'Fall' AND term.year = 2020	from the course table show me title AND THEN show me nothing in the room table where roomnumber = 2622 AND THEN show me nothing in the building table where buildingName = 'York Hall' AND THEN join the course table with the courseoffering table on course.id = courseoffering.courseID AND THEN join courseoffering with the term table on term.id = courseoffering.termid AND THEN join the courseoffering table with the room table on courseoffering.roomid = room.id and then join room with building on room.buildingid = building.id	W: 12.25 S: 2.36 W: Weighted S: Standardized

Table 9: User Study Queries (Query 9 - 12)