Synthesizing SQL from Natural Language

Anish Doshi
UC Berkeley
Berkeley, USA
apdoshi@berkeley.edu

Jianglai Dai
UC Berkeley
Berkeley, USA
jldai@berkeley.edu

Min-Chi Chiang
UC Berkeley
Berkeley, USA
mencher@berkeley.edu

ABSTRACT

Structured Query Language (SQL) is ubiquitous throughout the data science community. However, SQL is syntactically complex, difficult to remember, and does not always align with users' mental models. Thus, we build our tool on top of SQLectron, an open-source database client, and enable the tool to translate the natural language (NL) queries into formal SQL. Users would be able to express the SQL queries with various natural language combinations, such as texting, dragging tables, and clicking entries, and they can run the query result with our generated SQL query. To evaluate the intuitiveness of our tool, we conduct a formative study to observe the user behaviors. We defined the research question as how users express natural language specifications when trying to synthesize a SQL query. Meanwhile, we chose two protocols, probing and concurrent think-aloud, for the study. From the qualitative result, we discovered several use cases, such as preferring to start a query with verbs and preferring to use text compared to selecting entries on tables, and further iterated the platform based on the evaluation result.

Author Keywords

programming language, natural language programming, research software, programming-aided tool

MOTIVATION

SQL, Structured Query Language, is ubiquitous throughout the data science community. The use of databases has become universal in the 21st century. From storing medical records, to processing streams of sensor data, to indexing education records, databases are used in every domain to store, process, index, and secure data for later use. Given the importance of databases, it should not come as a surprise that SQL is one of the most important skills of data scientists in the 21st century, based on industry reports from Coursera, Indeed, and KDnuggets [3, 7, 9]. Given that the data science profession is growing at an annual rate of 31% year-over-year, which is much faster than average according to the U.S. Bureau of Labor Statistics, SQL is a must-have skill for anyone to compete in the 21st-century job market [12, 13]. The data science profession is also of paramount importance to companies around the world as evident from 2020 The Future of Jobs Report from the World Economic Forum, which surveyed employers across 15 industry sectors in 26 of the world's advanced and emerging economies to give insight on employment trends in the next five years. As shown in Figure 1, the job report concluded that the data scientist is expected to have the highest increase in demand in the job market. The reason is that COVID-19 has accelerated the digital transformation of businesses as a necessary survival strategy instead of the previously held belief that it is merely an optional marginal improvement in operations [5]. Given these massive tailwinds to upskill to data roles, it is necessary for many aspiring data scientists to learn SQL in the coming years.

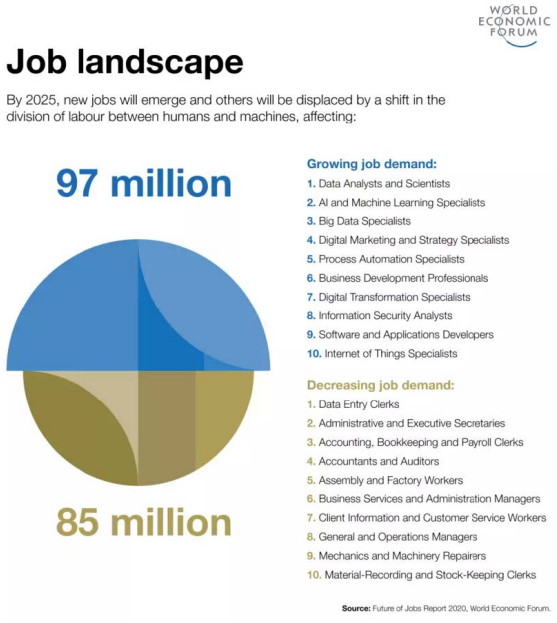


Figure 1. World Job Landscape 2020 - 2025

However, SQL has the following problems that often make data scientists have a hard time writing a runnable SQL statement. First, SQL is syntactically complex. Especially when a SQL statement involves join, aggregation, subquery, and string comparison, it becomes difficult for programmers to structure SQL statements and place keywords in the correct positions. Second, SQL syntax is difficult to remember. When writing SQL queries, most programmers would always need to keep switching back and forth between the web browser and

UIST'20, October 20–23, 2020, Minneapolis, MN, USA

ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

database IDE to search for SQL syntax, which undoubtedly reduces the SQL typing efficiency. Third, SQL does not always align with users' mental models. According to our previous need-finding study, we found our participants would sometimes be unable to write correct aggregation queries since it requires them to specify filtering conditions in both the HAVING clause and WHERE clause, and they're unable to distinguish the difference between these two clauses. The same situation also applies to the JOIN operation with the ON clause and WHERE clause.

SQL as a language is very limited in its capacity to port to other domains compared to general-purpose languages like Python that are used in many different settings like web development, machine learning, and many other fields. It would not be unreasonable to ask, "Given these headwinds on the declining value of SQL in the future job market, why even learn it at all?" We ask the same question and believe that it is not only a valid question to ask but also regard SQL as a huge roadblock in transitioning our current workforce to the 21st century. Given industry demand, current issues with SQL, and recent breakthroughs in code generation or program synthesis, we see an opportunity to create a platform that allows data scientists to bypass SQL and use English instead of interacting with their databases. The difficulty with using English, however, is that it lacks structure. Programming languages can be deterministically and safely understood by computers. English, however, lacks this inherent structure. Without a way to safely understand the intent of English phrases, interacting with databases may seem impossible for the large population of the workforce without knowledge of SQL.

The advent of deep learning has offered another option. The last few decades have seen tremendous breakthroughs, such as the Transformer model architecture and new GPU architectures specifically designed to train them at an ever-increasing scale. Extensive research using these breakthroughs has resulted in neural networks trained to translate English to SQL (NL2SQL) [16, 21]. One noteworthy example of this is the recent OpenAI model called Codex, a billion-parameter GPT-3 model fine-tuned on GitHub code, for the sole task of generating Python code from English that produced state-of-the-art results on code generation. Amazingly, although it was designed to generate Python code, it is able to generate other languages such as SQL as well due to the enormity of the pretrained corpus of the GPT-3 model [15].

However, we note the lack of usable applications that make use of this technology. Commercial database applications such as SAP Hana, Snowflake, and Microsoft SQL Server provide battle-tested interfaces to perform operations on databases and author SQL queries within editors, but as of yet have not taken advantage of the advances in NL2SQL. Of course, translation from NL to SQL is still not perfect, but we argue that providing an interface for someone to iteratively use the outputs of such models still dramatically improves anyone's ability to interact with databases. For veterans of SQL, having suggestions for automated tasks they find themselves performing reduces the time taken to write new queries. For newcomers, offering SQL suggestions from English commands allows people to start interacting with databases in ways they could not before.

This platform not only makes data scientists more productive as they wrangle SQL less, but also breaks down the barrier of entry to the profession. In effect, this platform will not only alleviate the current shortage of data scientists around the world, but also democratize the data science profession to the world. Our team developed an interactive application that allows users to type, speak, and select data as input to the NL2SQL synthesizer. Initial qualitative feedback received from our participants has been overwhelmingly positive, and we also discovered the robustness and usefulness of NL2SQL tools during the study.

RELATED WORK

Our goal is to build an interface for a user to enter in a natural language query (e.g. Fetch the latest 10 invoices), and having the system display a synthesized SQL query that matches their intent, (e.g. SELECT * from Invoices ORDER BY date DESC LIMIT 10). Research into this task has mainly progressed from the academic community, with collaboration from corporate research teams at companies like Google, Facebook, and Salesforce.

Synthesizing SQL from Natural Language

SQL is suitable for automated synthesis because it is a formal language - its composite tokens (keywords like SELECT, delimiters like, and other characters) come from a fixed alphabet, and its structure is guaranteed to follow a definite grammar. In fact, many synthesis strategies for SQL focus on formally searching over this grammar. The challenge, however, is understanding how the search can be guided by ambiguously specified natural language (which does not strictly follow grammar). In recent years, a multi-step deep learning approach has proven to be the most effective way to address this issue. The natural language query is fed into a neural network encoder, which often makes use of neural attention to generate a representation that captures the semantic intent of the query. Next, the schema of the database, i.e., table names, column names, and optionally types + foreign key constraints, is also encoded, either in the same sequence as the query or with another model. TAPAS [8] uses a pre-trained language model, BERT, to jointly encode both schema and NL query. Another approach is to use a graphical attention model, as done by RAT-SQL [22]. Finally, a neural network outputs the SQL query token by token, with some approaches instead of using "slot-filling" or an intermediate language representation [6].

Other areas of research focus on converting speech to SQL query. SpeakQL [18] adopts Automated Script Recognition (ASR) and Structure and Literal Determination components to parse speech query to runnable query. On the other hand, SpeechSQLNet [19] integrates a speech encoder, Graphical Neural Network, and transformer to construct speech-to-SQL systems.

Recently, OpenAI's Codex model, trained on GitHub, provided an alternative approach to SQL specific models: by training a large language model on a huge corpus of code

sourced from GitHub, OpenAI showed that numerous code synthesis tasks can be performed simply by submitting a natural language description of the task into the model [1, 2]. These methods are typically trained and evaluated on benchmark datasets containing thousands of manually curated natural language query/SQL query pairs. The two most notable such datasets are Spider and CoSQL [23, 24]; with the latter benchmark more oriented towards conversational natural language to SQL methods.

Providing an Interface for Users

Research on the user interface/experience (UI/UX) aspect of SQL synthesis is surprisingly sparse, and mainly originates from the corporate research community. Salesforce's Photon [25], released in 2020, integrates a neural semantic parser, human-in-the-loop question corrector, a SQL executor, and a response generator as a pipeline to generate the result, and provides a conversational interface that mimics a "text chat" with an interactive SQL synthesizer. The schema of the user interface has somehow enabled interactions between software and users and further fosters the iterations and revision of user input. Since the NL model often cannot provide the most accurate answer at the first time, this sort of interaction would help users retrieve the expected SQL query after several rounds of iteration.

NLSQL [11] follows a similar chatbot schema but design the chatbot as a digital assistant that would not only provide the table result to the user. Instead, the chatbot would sometimes clarify what users really want by asking the follow-up question and providing a variety of result display methods, e.g., Excel, Chat, Bubble. Meanwhile, it has successfully extended the platform to support several major database management systems, such as MySQL, Microsoft SQL Server, and PostgreSQL. Meanwhile, it provides a more powerful display interface. For example, if you specify "Show me a chart of trends that shows how many people start booking hotels that cost more than 300 dollars", NLSQL would automatically organize the result data and generate the chart for the users.

Microsoft's DIY [10], released in 2021, includes a tool to let users post-process neurally synthesized SQL queries. The uniqueness of this work is that it provides a sample data view, explainer view, and answer views. These views thoroughly provide the detail of how a natural language is converted and how data is generated based on the tokens from input statements. However, effectiveness and natural experiences for users are still required to be validated.

Since the natural language model is not robust enough to provide the most accurate result to the user at once, we believe enabling interactions between software and users is critical. Inspired by prior works, we intend to build a platform that would be able to provide real-time feedback to users based on their input, and enable interactions between software and users to let users iterate their input once they're unsatisfied with the result. Meanwhile, to develop our tool with human-computer interaction in mind, as Sarah Chasins encourages in the class "PL and HCI: Better Together" [17], we aim to provide a tool with flexibility and intuitiveness.

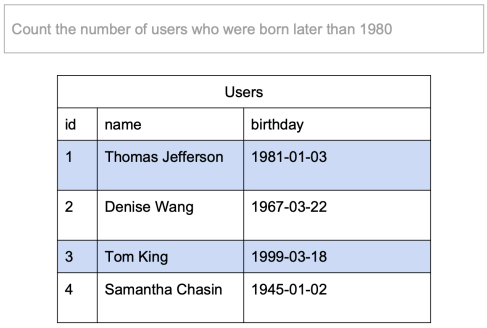


Figure 2. Example of data aggregation

MOTIVATING TASKS

Since the robustness of our tool is that users are able to perform complex operations without writing complex queries, we design the following complex and complicated tasks for users to demonstrate the robustness of our tool.

Data Aggregation

The sample problem is to perform data aggregation to find the number of users who were born later than 1980.

At the start of the interaction, IDE would provide users with several tables. These tables' views would contain the table name, columns, and several records. Thus, the IDE might display the Users table with a birthday column, which records formatted as "YYYY-MM-DD." The user would realize that it's possible to filter out users who were born later than 1980 based on their birthdays.

However, with existing SQL IDE, users might need to look for what columns they can reference to build the relationships between "user" and "birthday". Later on, users need to perform date conversion and data aggregation to filter out those who were born later than 1980 and count the total number of records. Since these tasks involve converting the birthday column from "YYYY-MM-DD" to only "YYYY" and performing data aggregation, programmers might not be able to come up with how to write these complex queries with SQL syntaxes. We assume they probably need to spend the majority of time searching for the syntax on Google, and frequently switching back and forth between web browsers and DB IDEs.

As shown in Figure 2, the benefit of our tool is that when programmers try to approach the problem, they would only need to type "Count the number of users who were born later than 1980". Our tool would then start synthesizing the queries that match the programmer's needs.

Table Joining

The sample problem is to perform table joins to find out all the orders for the user whose first name is "Thomas".

With existing tools, the user would first need to search over all tables for the ones most likely to correspond to the concept of Orders. Later on, the user needs to find how to concatenate these two tables on which foreign key. Besides, the user is also required to filter out all the users whose first name

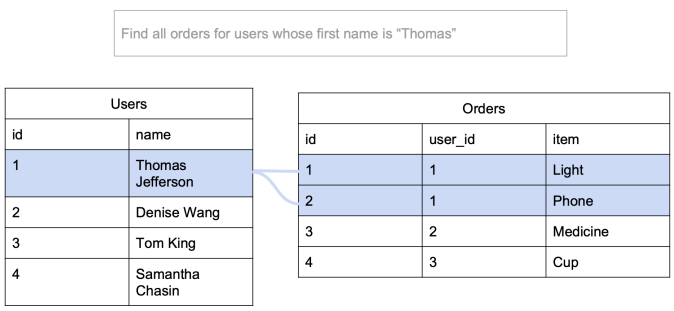


Figure 3. Example of table joining

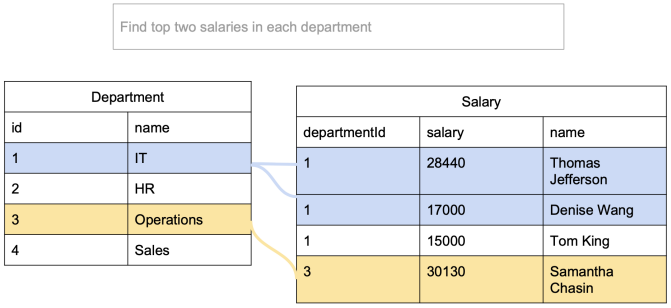


Figure 4. Example of subquery

is not equal to "Thomas." This operation requires writing regular expressions. However, it's hard to formulate queries that are able to perform table joining and string comparison simultaneously. First, table relationships might cross multiple tables (e.g., users table ←→ payments table ←→ orders table), and thus it's sometimes difficult to find how to join two tables. Second, filtering out users whose first name is equivalent to "Thomas" requires writing some regex expressions, but most programmers are not familiar with regex expressions. Thus, programmers would probably need to look for a Regex document when writing a query.

However, our tool would provide the visualization showing the relationship between table users and table orders. As shown in Figure 3, the users know they're able to combine records from two tables. The programmer only needed to type "Find all orders for users whose first name is "Thomas," and then our tool would synthesize the query.

Subquery

The sample problem is to write an SQL query to find the employees who are top 3 high earners in each of the departments.

With existing tools, the user would first need to figure out the table containing the salary information and the one containing the department information. Later, the user would realize that they need to go through records one by one and count the number of salaries which is bigger than the current salary record. This operation would require users to write SQL subquery to

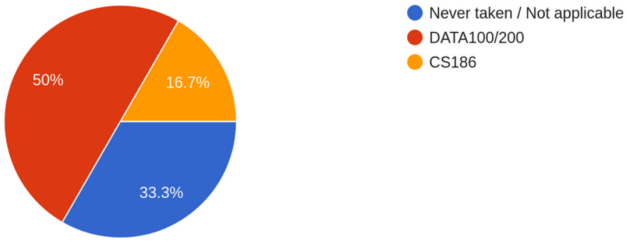


Figure 5. Classes taken by participants

finish the operation. Meanwhile, the user is also required to merge the salary table with the department table to retrieve department information. However, based on the observation from our prior need-finding study, we realized users are not able to successfully construct queries with subquery, not to mention successfully write runnable multi-level queries with table join.

As shown in Figure 4, after users realize the relationship between these tables, they just need to type "Find top three salaries in each department" in our tool, then our tool would start synthesizing the query that performs subquery and table join.

USER STUDY

Our formative user study is based on the research question: **In what ways do users express natural language specifications when trying to synthesize a SQL query?** Through the user study, we hope to gain insights into the way people are used to specifying SQL queries in natural language, after they are given our NL2SQL tool. Also, we hope to learn what kind of combination of input (e.g., typing, clicking and dragging) is intuitive for users. In addition, We want to test the software and improve both the ML model and the user interface based on user feedback.

We designed a participant recruiting survey and posted the link on several groups and channels. Based on our research question, qualified participants are supposed to have basic understanding of relational databases (e.g., the concepts of table and column), familiarity with basic SQL syntax and certain level of proficiency in English. The experience with SQL language is required, since we assume that those who are already familiar with SQL syntax would be able to tell whether generated queries are semantically correct, and be able to write English queries that is more likely to meet the standard of the NL2SQL model. In our survey, we asked volunteers how many lines of SQL code had they written, and whether they had enrolled in DATA100/200, CS186 or any equivalent class teaching SQL language. We also selected participants based on their availability. The most updated demography data have shown in Figure 5, 6 and 7. As shown in Figure 5 and 6, we can see 83% of participants have taken college-level courses. Meanwhile, from Figure 7, we can observe that 83% of participants have written more than 30 lines of codes.

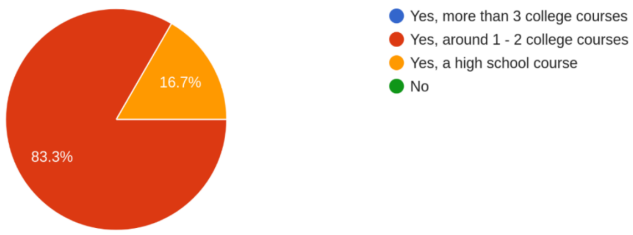


Figure 6. Number of related classes taken by participants

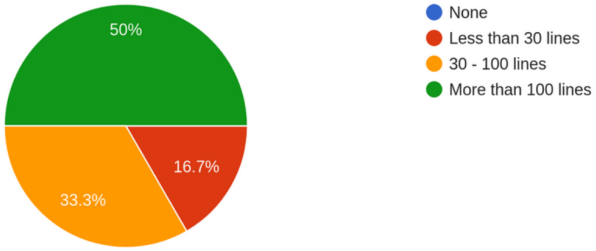


Figure 7. Lines of SQL code participants had written

We designed our user study as a qualitative one, since we are supposed to learn participants' thoughts and insights while using our tool to come up with solutions for their tasks. A qualitative study would also be helpful for us to identify existing design problems that may affect user experience. In order to collect qualitative data during the study, the major approaches we adopt is probing and concurrent think-aloud. Specifically, we give participants a walk-through of the user interface, ask participants to think aloud in their task execution, and conduct a short post-session user interview with each of them.

At the beginning of each session, we gave a brief introduction about the NL2SQL tool and the user interface to the participant. We also walked through the UI to show the participant how to input query, synthesize query suggestions, select best option and execute it. In order not to introduce bias, we used a standard SQL query rather than English sentences as the input, based on which the NL2SQL model would only output itself as the suggested query.

However, our user study is also semi-evaluative in the sense that we are presenting them with a functioning database client and UI, and collecting feedback about it. Although we are primarily interested in examining the space of natural language specifications, we also qualitatively evaluate how well certain elements of our tool's design work. While we don't adopt a concrete set of criteria, we aim to reduce the amount of noticed frustration users have. In the future, we hope to run a more thorough evaluative study with set tasks, and monitor more specific criteria to judge how well different aspects of our tool work.

After the walk-through, participants are supposed to complete tasks to fetch information from the given database. Instead of designing specific tasks in advance, we decided to ask participants to come up with tasks by them, since it is likely for participants to input natural language by imitating the description of the tasks. Basically, we just asked them what information they would like to get from the given database. We would keep reminding the participants if the tasks they proposed are either too simple or prohibitively complicated.

We adopted a hybrid mode with concurrent probing and retrospective probing to as the interaction between moderators and participants. It is necessary to offer help and answer questions asked by participants while they are exploring or using our tool. There is likely to be some confusion during their interactions with the software, since it might be the first time for the participants to use a tool to convert natural language to valid SQL queries. That being said, we wouldn't give participants any guidance of input, which obviously introduces biases into the study, or interrupt them in their task execution on purpose except there is a clear need to moderate the study session.

In addition to probing, concurrent think-aloud is another major approach for our user study. The participants are asked to verbalize their thoughts during the process of interacting with our NL2SQL prototype.

With thinking aloud concurrently rather than retrospectively, the participants would not have to describe their experience after finishing their task, of which the reliability highly depends on the participants' ability to summarize.

Besides, since our research question is specifically about understanding the interleaving between natural language and hard specifications, we can use verbalized thoughts as a proxy for the natural language specifications someone might have. If we notice a participant is saying "I'm trying to find out how to filter this table for rows that have this user id" while clicking on a record, we can infer that row selection and the natural language is a good input to the synthesizer.

Moreover, we found that thinking aloud is actually quite helpful for participants to feel more confident to try out different ideas and approaches while figuring out solutions for their tasks. Since we're giving users synthetic tasks, not observing them doing tasks they already normally do, the probing approach risks making the user feel less confident about trying different ideas.

RESULT ANALYSIS

In addition to mock-up sessions, we conducted six 1-on-1 formative user study sessions in total. We took notes from participants' behaviors, conducted retrospective interviews with them, and recorded the screen as well as the participants. Based on these methods, we gained insights into both the common input specifications of users and possible improvements for our tool.

We found that it is quite intuitive for participants to describe their queries based on a verb. Here is a few real examples that the participants came up with during our study:

- *Get me all data with name Thomas*
- *Fetch all data in column B*
- *Find the artist of the album named XXX*

- `List all rows in albums`

Though different people may choose different words, the basic structure of sentences barely changed. Participants usually started with a verb that is similar to a common method name in some programming language, then they wrote the target object and gave description about the target. It is likely that they are still thinking based on the structure of SQL query, given that all of our participants had some experience in SQL language.

During our study, some of the participants were able to directly thinking and writing queries in natural language, after being given the task. We found this was likely to be correlated with participants' level of proficiency and confidence in English. While handling those correct English sentences, our tool performed pretty well. For instance, it was always able to identify the need to join tables and suggest correct columns to join on, even if there was no keyword like "join" in the original sentences. From our observation, this was beyond many participants' expectation of the capability of the tool.

While some participants were able to describe queries in natural language right away, there were also participants who were more willing to convert their queries to natural language step by step. Here is an example of an iteration process of three steps that was came up by a participant:

1. `SELECT * FROM albums WHERE artistId = 2`
2. `List all rows in albums where artistId = 2`
3. `List all albums whose artist id equals to 2`

One of the interesting findings was that, for non-native speakers, describing queries in natural language is somewhat similar to writing SQL queries. While writing queries in English, they were sometimes pondering the grammar, just like thinking about the syntax while writing in SQL. This iteration process is also probably a result of the fact that the participants were not sure about the capability of our tool. Thus, they would like to try something out first before directly starting to write queries based on what on their mind.

We noticed that some users preferred to fully understand the query before executing it; whereas others eagerly executed queries to judge their correctness. In both cases (i.e., either before or after execution), we noticed that editing the query afterwards is a fundamental aspect of using our tool. Interestingly, after clicking the suggestion, 4 out of 6 of our participants did not immediately move to editing it, but instead tried to visually understand elements of the query or its execution result. We did provide some gentle prompting to suggest editing, with the question of whether they would edit it to provide another *natural language command*, or *revert to editing the SQL*. We found that three of our 6 participants relied on the NL editing process described above, whereas the other 3 of them simply edited the SQL directly.

Using Google was a key aspect of everyone's process. When figuring out how to write a SQL query, many people would google commands. We noted that the flow of Googling was very similar to using an NL2SQL tool, since users were usually trying to search for things in the domain of "how to write a query that does ...". Unfortunately, we did not monitor the queries Googled, but a future formative study should take those into account, since those are precise examples of intent that we might want to equip our synthesizer to handle.

The most surprising thing was there was not a single participant who managed to interact by clicking. By our design, we expected users to click table intuitively and use words like "this column" or "this value". Actually, we had found this problem during mock-up sessions. we decided to explicitly tell participants, during the UI walk-through, that they were able to interact with displayed table through clicking, without actually showing them how. We thought, by this way, they would be able to explore this feature by themselves later in execution of tasks.

However, no one did that. Some participants claimed that they were thinking the displayed table was just the result of executing SQL queries, so there was no point in interacting with the result. Others did try to click but they gave up after they found there was no significant effect after either right or left click. And all of them never thought of using keywords like "this value" "this column" as we designed.

In addition to the insights into the research question, we also noticed some possible improvements and iterated our tool based on the feedback collected from the formative user study.

From users' perspective, it could be very confusing to see suggestions from NL2SQL model consisting of some invalid queries. To prevent this issue, we updated our backend to validate the syntax correctness of any synthesized results before returning them. Based on the problems found during mock-up sessions, we got rid of a button of "record speech" to focus better on our research question. Moreover, we redesigned the button layout to build a cleaner user interface for formative user study.

During studies, many participants suggested that some autocompletion features could be the next step for development. From users' perspective, it would be a huge improvement regardless of whether it is able to suggest natural language directly or just token names in the table schema. Thus, we improved the editor interface by providing several suggestions on what types of commands users can execute.

TOOL DESIGN AND IMPLEMENTATION

We implemented our tool on top of *SQLectron* [20], an open-source database client. SQLectron is itself built in ElectronJS [4], a cross-platform desktop application toolkit that combines a nodejs with a Chromium based rendering engine. Our choice of building our tool on top of SQLectron was motivated by the following factors:

- **Location switching is ok:** From our background user studies, we found that the most users were generally comfortable entering a new interface to author queries. As mentioned above, we uncovered a diversity of ways in which people actually had to *execute* queries (data analysts were executing queries on DB applications; data science students were submitting Jupyter notebooks; software engineers were plac-

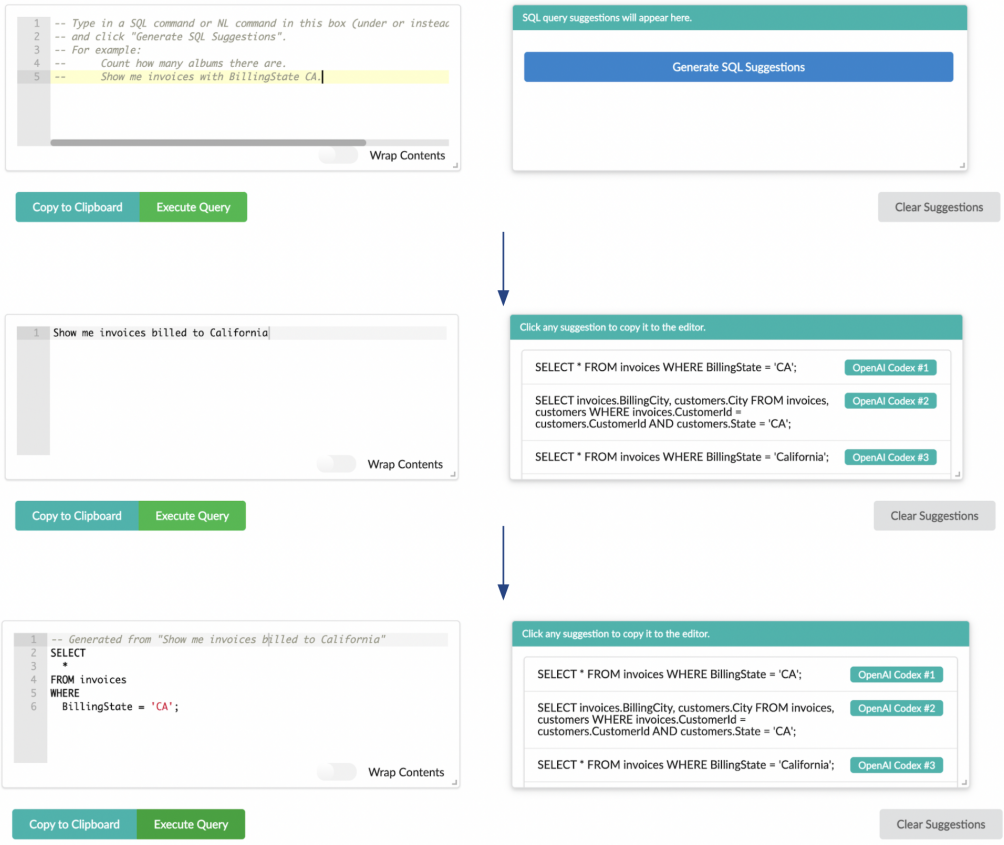
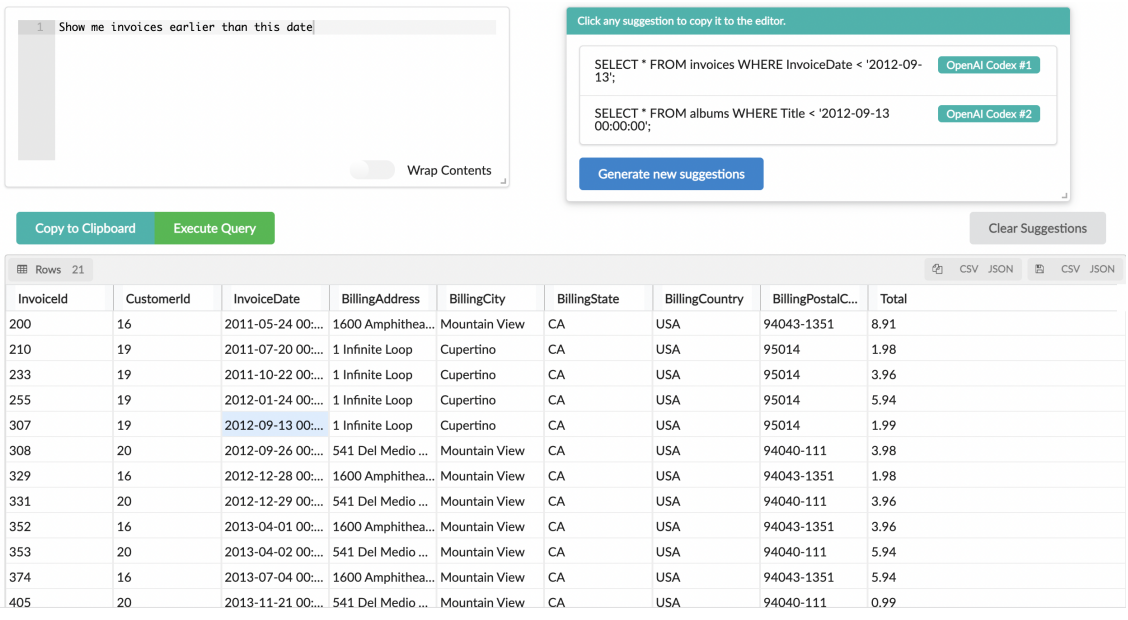


Figure 8. The flow of generating SQL suggestions from an NL command. 1) The user starts with a box with several suggested *natural language* commands present in comments. 2) The user then types out a command, presses Generate Suggestions, and waits for suggestions to be generated. 3) The user can then select a suggestion, which will be copied back into the editor where it can be directly executed.



InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalC...	Total
200	16	2011-05-24 00:...	1600 Amphithea...	Mountain View	CA	USA	94043-1351	8.91
210	19	2011-07-20 00:...	1 Infinite Loop	Cupertino	CA	USA	95014	1.98
233	19	2011-10-22 00:...	1 Infinite Loop	Cupertino	CA	USA	95014	3.96
255	19	2012-01-24 00:...	1 Infinite Loop	Cupertino	CA	USA	95014	5.94
307	19	2012-09-13 00:...	1 Infinite Loop	Cupertino	CA	USA	95014	1.99
308	20	2012-09-26 00:...	541 Del Medio ...	Mountain View	CA	USA	94040-111	3.98
329	16	2012-12-28 00:...	1600 Amphithea...	Mountain View	CA	USA	94043-1351	1.98
331	20	2012-12-29 00:...	541 Del Medio ...	Mountain View	CA	USA	94040-111	3.96
352	16	2013-04-01 00:...	1600 Amphithea...	Mountain View	CA	USA	94043-1351	3.96
353	20	2013-04-02 00:...	541 Del Medio ...	Mountain View	CA	USA	94040-111	5.94
374	16	2013-07-04 00:...	1600 Amphithea...	Mountain View	CA	USA	94043-1351	5.94
405	20	2013-11-21 00:...	541 Del Medio ...	Mountain View	CA	USA	94040-111	0.99

Figure 9. Combining data selection with NL commands. With the user having selected a particular `InvoiceDate` value from the table, they are then able to

ing query calls in code). However, to actually *author* the queries, users seem not to mind switching between environments. We hypothesize that the most common authoring pattern - switching between Google and the users' existing environment - might have primed users to be comfortable with context switching while authoring.

- **Batteries are included:** Out of the box, SQLectron already provides many common functions of a database client. The entrypoint into the client is an interface to connect to a variety databases (including PostgreSQL and SQLite, which were commonly used by our users). After successfully connecting, a main view pops up with three main components: a *sidebar*, a *query editor*, and an *execution results view*. The *sidebar* lists the current set of tables, each tables' columns, and their associated datatypes, in a dropdown based menu. Queries can be written in a *mini editor* that includes keyboard navigation, syntax highlighting, database schema based auto-completion, and edit history. Finally, queries can be executed, and their results (or an error message) will then show up in a tabular *execution results view*. By building on top of this client, we provide users with novel interactive NL2SQL functionality while also letting them navigate and query databases in ways that mimic other DB applications.
- **Hackable:** The implementation of SQLectron is easily extendable. The frontend is built with React and semantic UI, and written in TypeScript. We augment this with a Python based backend, which launches a REST api server that the frontend can connect with over HTTP.

Design

We now describe the design of our tool. We made many changes based on the formative and evaluative elements of our user study, and we highlight the most important changes in blue.

The UI of our tool is shown. Having connected to a database, our tool extends the SQLectron UI with a resizable *suggestions* box to the right of the editor. Within the editor, we also provide suggestions for natural language commands to try.

- **Users we had interviewed did not know. where to start.** Trying to phrase a command for an unknown tool might be more confusing than writing SQL itself. For users that do not use English as a native language, we describe this exact comparison. Even within the native English speakers we interviewed, one user expressed that having interacted with natural language command tools before, he found them "very buggy unless you phrase things in a particular way". Unfortunately, the limitations on our NL algorithms more or less support this argument. However, we do know that certain structures of natural language queries are good inputs for our synthesizer, in the sense that NL2SQL models *usually* perform well on them. For example, "Count how many X there are" is a command for which our main neural model consistently produces the correct SELECT query.

  Thus, our editor provides example natural language commands to try, which are *customized* for the schema of currently selected database. To generate such suggestions, we search for columns of a preset list of types: numeric columns, dates, locations (detected by searching for substrings like "State"), and categorical columns (ones whose output range is much less than the number of rows). Additionally, we also collect per-table examples for common operations like counting and fetching all records. We then suggest commands based on templates which the model performs well on. Our current implementation randomly selects 2 such suggestions to display.

  An example is shown in the figure 8. Given a SQLite database, we choose one table level command for counting the number of `albums`, and a column-specific command for filtering invoices by a location column, `BillingState`. Note that including the table name is also important for the column specific NL.

  We hope to continue improving the method of generating suggestions, especially for more sophisticated operations like joins and groupbys. Trying to provide too many suggestions, or a fully featured "nl suggestion editor" might be too much, as that limits the user to express their intent in structured ways that might not be necessary for our neural synthesizer. However, if we can give simple examples of what sorts of commands the user might try, the users might feel a lot more comfortable expressing their true intent in our tool.

When the user presses the "Generate Suggestions" button in the center of this box, we take the content in the editor box on the left, and pass it to our NL2SQL backend. Our backend algorithm formats the input for a neural model, which generates 4 candidate queries. These queries are deduplicated, and passed to a postprocessing function. The postprocessing step first tries to fix common errors in the outputs of neural models, and then actually *executes* the query on the database.

- **We noticed from our formative studies that users found incorrect query suggestions extremely confusing.** We had initially hypothesized that neural SQL output that might be incorrect could be simply flagged as such before suggesting. However, SQL queries that caused an error tended to cause frustration and block the user in debugging, so instead, we now guarantee that a given SQL query will actually be valid before surfacing it.

While generation is in process, the suggestions box shows a loader. When the generated queries are ready and sent back to the UI, they are then displayed as a list. Clicking an individual query in this list copies it back into the lefthand editor, with a comment annotation describing the source of the suggestion. The copied suggestion is also *indented* for visual clarity - constituent keywords of the SQL query like `SELECT`, `FROM`, `WHERE` appear on their own lines.

- **Users found it difficult to visually parse the suggested queries.** Indenting them once their copied can help with this, but an ideal solution would also make the queries easier to understand while they're still in the suggestion box. Syntax highlighting, and maybe an alternative way of listing them where we *paginate* suggestions (only one query shows up at a time) are both ideas to evaluate.

As with the original tool, the user presses "Execute Query" to run the query on the database. Given the importance of the editing process, but the dichotomy between editing the SQL directly and editing NL, our current design *compromises* between making both smooth:

- After being copied, generated SQL can be directly edited, and re-executed. Users do not need to press any buttons or perform any other action in the editor, besides editing their SQL string and pressing Execute Query.
- After being copied, the SQL suggestion includes a *comment annotation* of the form `- Generated from "<NL command>"`. The <NL command> text can be directly edited as well. If the user then re-presses "Generate Suggestions" in the righthand suggestions box, we treat the string within those quotes as the input NL command, and regenerate suggestions for it. However, since it is still within a comment, this editing does not interfere with the validity of the SQL.

The goals of the user after generating and editing a query are user/context/situation dependent. However, our tool focuses on three main classes of actions the user might want to do once they have a usable query:

- **They might want to copy the SQL query to another application.** Given our knowledge that users are comfortable context switching into our tool to author queries, but are actually trying to embed the queries within other applications, this is *probably* the most common use case. To support this, we include a Copy to Clipboard button placed directly under the editor window that simply copies the editor content (including comments) to the clipboard.
- **They might want to copy the results of executing the query (the rows generated from the SELECT query) to another application.** To enable this, our execution results table of data can also then be exported (either copied, or saved) to CSV or JSON. The text within the table is also highlightable and copyable. While not necessarily the format the user might want, providing these formats as sensible defaults should enable them to move into other editing tools for postprocessing their data.
- **They might want to keep authoring queries while saving their work so far**. To support this case, the overall application includes a tab view at the top, to enable people to write new queries without losing their work on previous ones. In this way, our application maintains a manually maintained edit history. Our interface also includes a button to save the SQL directly to a local ".sql" file (it will create a new one with a numbered different name if that file already exists).

WHAT SHOULD I KNOW ABOUT YOUR PROJECT?

Conclusion

We developed a tool to improve the experience of authoring SQL queries, by allowing users to specify commands in natural language. By building on top of SQLectron, we are able to embed an NL2SQL interface within a flexible DB client application. To improve our interface and motivate further improvement of the neural synthesizer, we investigated the research question of what natural language commands users would intuitively use within our tool. We combined this with qualitative evaluation of our tool's interface. Our findings suggest that our synthesizer tool could indeed help people trying to author queries, and we detail a new tool design that takes into account the evaluative feedback we received.

Future work

In the future, we aim to continue improving the neural synthesizer, while also moving towards more rigorous evaluative studies for the UI. One of the challenges with evaluative studies of natural language interfaces is that tasks themselves are often specified in natural language, which means that just the task *specification* could be a possible direct input to the synthesizer. This can either overly bias the results in favor of NL2SQL, or, it can potentially lead a user down the wrong path if the task specification NL is not a good input to the synthesizer. To get around this, for evaluative studies we might try a similar approach of prompting the user for potential tasks they might want to perform, and choosing one based on if it meets certain criteria. As mentioned in our previous evaluation assignments, we also will aim to use *counterbalancing* treatments to reduce the effect of variables like general tool familiarity. Finally, we will aim to do evaluations in more real world/expert contexts, where we monitor someone using this tool for a longer task that they might "actually" be performing.

Another input specifications research question

One of the side research questions that we had initially proposed, was whether combining *data selection* with the natural language command could be either useful or more expressive than the natural language command alone.

Design

The functionality we implemented for this allows users to select (and unselect) cells, columns, and rows of the execution result table. With something selected, they can then type in a natural language command that *refers* to the current selection, with the keyword `this`. They can then press generate suggestions as normal.

Synthesis

To resolve references to the data selection, our backend does the following. First, it regex searches for instances of "this <keyword>" in the input NL command. If any instances are found, it then checks if <keyword> is one of value, cell, or row. In these cases, we replace the phrase "this <keyword>" with a preset phrase that incorporates the selected data/column/table. For example, "this value" could become "FirstName equal to Jane"; "this column" could become "column FirstName"; "this row" could become "the row where CustomerId equals 1". We chose such template phrases based on what our neural model performed well on. Additionally, the keyword can be a fuzzy reference to the title of a column or table - e.g. if we detect "this date", and the currently selection is in a column called "BillingDate", we do a replacement "BillingDate equal to <the selected date>".

An example is shown in Figure 9. The user typed out `Show me all invoices earlier than this date`

with the value "2012-09-13 00:00:00" selected; they then requested suggestions. The algorithm would first convert the command to `Show me all invoices earlier than InvoiceDate equal to 2012-09-13 00:00:00` before feeding it to the neural models. The first suggested SQL query will correctly accomplish this task.

Preliminary Evaluation

Unfortunately, we found that none of our users use this feature, even when explicitly described. In most cases, the tasks they were performing (fetching, aggregation, joining) simply did not need this sort of functionality. We realize that the data selection would probably be most effective with *filtering*, but this was not a common task. Even when such a selection might be useful in terms of time taken (e.g., when the cell value is very hard to type out), it might have been more natural to simply suggest commands that include that value directly. For example, if a user selects a particular InvoiceDate cell value, we might suggest a SQL command that directly filters out values before that date.

Nevertheless, we believe there is still value in some combination of selection and NL commands. If we can, for example, combine data selection with *verbal* queries (which we also abandoned before!), the user might be able to do tasks by demonstration. For example, they might say "Join this column", select one column, navigate to another table, say "with this one", and select another one there. This sequence of actions is difficult to do with typing, since the users cursor would need to navigate back and forth between the editor and the table view. As described in *Evaluating user interface systems research* [14], if we can improve the *ease of combination* of different specification types, we might be able to positively evaluate this tool.

REFERENCES

[1] 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. (2020). DOI: http://dx.doi.org/10.48550/ARXIV.2005.14165

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). DOI: http://dx.doi.org/10.48550/ARXIV.2107.03374

[3] Coursera. 2022. What is a Data Scientist? Salary, Skills, and How to Become One. (2022). https://www.coursera.org/articles/what-is-a-data-scientist.

[4] ElectronJS. 2022. ElectronJS. (2022). https://www.electronjs.org.

[5] World Economic Forum. 2020. The Future of Jobs Report 2020. (2020). https://www.weforum.org/reports/the-future-of-jobs-report-2020.

[6] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John H. Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. *CoRR* abs/2109.05153 (2021). https://arxiv.org/abs/2109.05153

[7] Jeff Hale. 2018. The Most in Demand Skills for Data Scientists. (2018). https://www.kdnuggets.com/2018/11/most-demand-skills-data-scientists.html.

[8] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TAPAS: Weakly Supervised Table Parsing via Pre-training. *CoRR* abs/2004.02349 (2020). https://arxiv.org/abs/2004.02349

[9] Indeed. 2021. Top 12 Skills You Need to Become a Data Scientist. (2021). https://www.indeed.com/career-advice/resumes-cover-letters/skills-for-a-data-scientist.

[10] Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. (2021). DOI: http://dx.doi.org/10.1145/3397481.3450667

[11] NLSQL. 2022. NLSQL AI Bot. (2022). https://www.nlsql.com.

[12] U.S. Bureau of Labor Statistics. 2020. Computer and mathematical occupations. (2020). https://www.bls.gov/ooh/about/data-for-occupations-not-covered-in-detail.htm.

[13] U.S. Bureau of Labor Statistics. 2021. Occupational Employment and Wages, May 2021;15-2051 Data Scientists. (2021). https://www.bls.gov/oes/current/oes152051.htm.

[14] Dan R. Olsen. 2007. Evaluating user interface systems research. *20th annual ACM symposium on User interface software and technology* (2007).

[15] OpenAI. 2021. OpenAI Codex. (2021). https://openai.com/blog/openai-codex/.

[16] Dave Salvator. 2022. H100 Transformer Engine Supercharges AI Training, Delivering Up to 6x Higher Performance Without Losing Accuracy. (2022). https://blogs.nvidia.com/blog/2022/03/22/h100-transformer-engine/.

[17] Joshua Sunshine Sarah E. Chasins, Elena L. Glassman. 2021. PL and HCI: Better Together. (2021). https://cacm.acm.org/magazines/2021/8/254314-pl-and-hci/fulltext.

[18] Vraj Shah, Side Li, Arun Kumar, and Lawrence Saul. 2020. SpeakQL: Towards Speech-Driven Multimodal Querying of Structured Data. (2020), 2363–2374. DOI: http://dx.doi.org/10.1145/3318464.3389777

[19] Yuanfeng Song, Raymond Chi-Wing Wong, Xuefang Zhao, and Di Jiang. 2022. Speech-to-SQL: Towards Speech-driven SQL Query Generation From Natural Language Question. (2022). DOI: http://dx.doi.org/10.48550/ARXIV.2201.01209

[20] SQLectron. 2022. SQLectron. (2022). https://sqlectron.github.io.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. (2017), 6000–6010.

[22] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. *CoRR* abs/1911.04942 (2019). http://arxiv.org/abs/1911.04942

[23] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter S Lasecki, and Dragomir Radev. 2019. CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases. (2019). DOI: http://dx.doi.org/10.48550/ARXIV.1909.05378

[24] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. (2018). DOI: http://dx.doi.org/10.48550/ARXIV.1809.08887

[25] Jichuan Zeng, Xi Victoria Lin, Caiming Xiong, Richard Socher, Michael R. Lyu, Irwin King, and Steven C. H. Hoi. 2020. Photon: A Robust Cross-Domain Text-to-SQL System. (2020). DOI: http://dx.doi.org/10.48550/ARXIV.2007.15280