DIY: Assessing the Correctness of Natural Language to SQL Systems

Arpit Narechania
Georgia Institute of Technology
Atlanta, USA
arpitnarechania@gatech.edu

Adam Fourney
Microsoft Research
Redmond, USA
adamfo@microsoft.com

Bongshin Lee
Microsoft Research
Redmond, USA
bongshin@microsoft.com

Gonzalo Ramos
Microsoft Research
Redmond, USA
goramos@microsoft.com

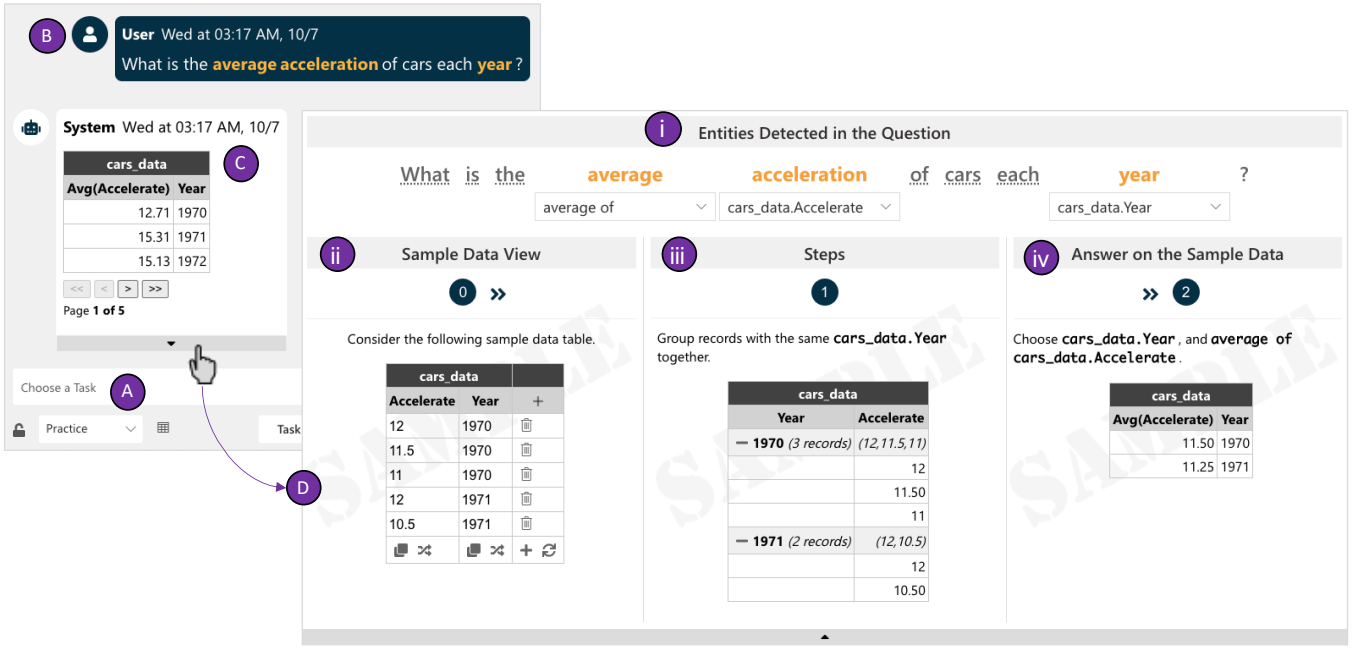


Figure 1: The *DIY* technique implemented in a QA shell. (A) Query input, (B) Annotated Question View shows the question with important tokens highlighted, (C) Answer on Production Database View shows the query result on the production database (DB), and (D) Debug View. (i) Detect Entities View shows the mappings between the question and the query, (ii) Sample Data View shows a *small-but-relevant* subset (sample testing DB) of the production DB, (iii) Explainer View provides step-by-step explanations of the query, and (iv) Answer on Sample Data View shows the query result on the sample testing DB.

ABSTRACT

Designing natural language interfaces for querying databases remains an important goal pursued by researchers in natural language processing, databases, and HCI. These systems receive natural language as input, translate it into a formal database query, and execute the query to compute a result. Because the responses from these systems are not always correct, it is important to provide people with mechanisms to assess the correctness of the generated query and computed result. However, this assessment can be challenging for people who lack expertise in query languages. We present *Debug-It-Yourself (DIY)*, an interactive technique that enables users to assess the responses from a state-of-the-art natural language to SQL (NL2SQL) system for correctness and, if possible, fix errors. DIY provides users with a sandbox where they can interact with (1) the mappings between the question and the generated query,

IUI '21, April 14–17, 2021, College Station, TX, USA

ACM ISBN 978-1-4503-8017-1/21/04…$15.00
https://doi.org/10.1145/3397481.3450667

(2) a *small-but-relevant* subset of the underlying database, and (3) a multi-modal explanation of the generated query. End-users can then employ a back-of-the-envelope calculation debugging strategy to evaluate the system's response. Through an exploratory study with 12 users, we investigate how DIY helps users assess the correctness of the system's answers and detect & fix errors. Our observations reveal the benefits of DIY while providing insights about end-user debugging strategies and underscore opportunities for further improving the user experience.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces**; **User interface management systems**; • **General and reference** → **Verification**.

KEYWORDS

natural language interface, human computer interaction, database systems

ACM Reference Format:
Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. In *26th International Conference on Intelligent User Interfaces (IUI '21), April 14–17, 2021, College Station, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3397481.3450667

1 INTRODUCTION

Current advances in machine learning make it possible for many systems to let their users express and fulfill their goals through natural language (NL) in what are known as natural language interfaces (NLIs). A particular family of these systems, NLIs for querying databases, have been studied by researchers in natural language processing [12, 17, 19], databases [3, 14, 15, 26, 29, 30, 40, 48], and HCI [11, 18, 21, 31, 32, 34, 37, 42]. Systems employing these NLIs receive a natural language (NL) question as input, translate it into a formal database query and execute the query on the underlying database to compute an answer. Existing systems present these responses using a combination of the computed answer, the generated query, any associated meta-data (e.g., mappings between the question and the generated query), easy-to-understand explanations of the aforementioned artifacts (using, for example, NL and visualizations), and UI control augmentations (e.g., drop-downs) that facilitate fixing errors and disambiguation.

These systems present challenges for users who may be familiar with the domain but are not fluent in the database query language. In particular, assessing the correctness of an answer that is output from an NLI can be challenging. For example, in a system that answers questions about a cars database (Figure 2a), a user asks a question *"Which car makers are American?"* The system first translates it into a SQL query (Figure 2b), and then runs it on the database to compute the answer—*Ford, Chrysler* (Figure 2c). An expert on cars might suspect the answer to be correct based on their knowledge, but it might not be so. In this case, the question contains *"American,"* which is syntactically similar to *"america"* of the `continents.Continent` column and semantically similar to *"usa"* of the `countries.CountryName` column. Just based on the computed answer, it is challenging for users to infer if this NL ambiguity was successfully resolved. Displaying the generated SQL query can help make these issues clear, but will only benefit users fluent in the query language.

In this paper, we present Debug-It-Yourself (DIY; Figures 1 and 4), an interactive technique that enables users without specialized knowledge of a query language (e.g., SQL) to assess the responses of a state-of-the-art NL2SQL system for correctness. Specifically, DIY lets users inspect for, isolate, and if possible, fix errors in the system's output. DIY's intended users include domain experts with limited databases experience and information workers who are not familiar with writing complex database queries.

Our work builds upon past research exploring how to explain queries and answers using natural language [9, 22, 23, 33, 35] and visualizations [1, 2, 8, 25], as well as past work that utilizes multimodal interactive widgets to communicate and resolve ambiguities [11, 26, 31, 32, 34]. Our approach differs from prior work by its use of the *data* itself as a means to explain the query and the query execution process. DIY presents users with a sandbox where they interact with (i) a *small-but-relevant* subset of the underlying production database, which we refer to as the sample testing database; (ii) mappings between the entities in the question and the generated query; and, (iii) multi-modal explanations of the execution of the generated query on the sample testing database.

Figure 3 shows DIY applied to our earlier scenario (Figure 2). DIY first identifies relevant tables and columns from the query and samples a few relevant records from the underlying production database (Figure 3a). The query is then broken into three subqueries that are sequentially executed on the sample testing database. Each subquery explains one or more SQL clauses: `FROM, JOIN` (Figure 3b), `WHERE` (Figure 3c), and `SELECT` (Figure 3d), respectively using NL and tabular visualizations. Figure 3d is also the final answer when the query is executed on the sample testing database.

The sample data and sandbox environment allow users to employ different back-of-the-envelope calculation debugging strategies to assess the correctness of the query. For example, users can experiment by modifying the sample testing database (e.g., edit a cell's value) that updates the subsequent steps including the answer. Experimenting with the sample testing database can build trust in the system's interpretation of the original question and its output on the production database. If a problem is detected, DIY also presents users with the means to fix errors and resolve ambiguities by allowing them to adjust the mappings between the question and the generated query. The adjusted query is automatically applied to the sample testing database and can eventually be applied to the production database.

We use DIY as a design probe in a user study with 12 participants from a large technology company to investigate how it helps users assess the system's answers and isolate and fix errors. Our observations reveal how DIY helped participants assess the correctness of a state-of-the-art NL2SQL system while providing insights about different debugging strategies. We discuss opportunities for improving the user experience, as well as the remaining challenges and open questions. Our contributions include:

(1) Generation of a *small but relevant* subset of the database to effectively demonstrate the execution of a SQL query.

continents

ContId	Continent
1	america
2	europe
3	asia
4	africa
5	australia
...	...

countries

CountryId	CountryName	Continent
1	usa	1
2	germany	2
3	france	2
4	italy	2
5	japan	3
...	...	...

car_makers

Id	Maker	CountryId
1	Citroen	3
2	Ford	1
3	Daimler	2
4	BMW	2
5	Chrysler	1
...	...	...

model_list

Id	Model	MakerId
1	citroen	1
2	plymouth	5
3	mercury	2
4	mercedes	3
5	bmw	4
...	...	...

car_names

Id	Make	ModelId
1	ds pallas	1
2	satellite	2
3	duster	2
4	zephyr	3
5	benz 300	4
...	...	...

cars_data

Id	Horsepower	Weight	Edispl	Accelerate	Year
1	115	3090	133	17.50	1970
2	150	3436	318	11	1970
3	95	2833	198	15.5	1973
4	85	3070	200	16.70	1978
5	77	3530	183	20.10	1979
...	...	...	...	...	...

(a) The cars database (production database); → depicts the Foreign Key - Primary Key relationships; [...] imply more rows.

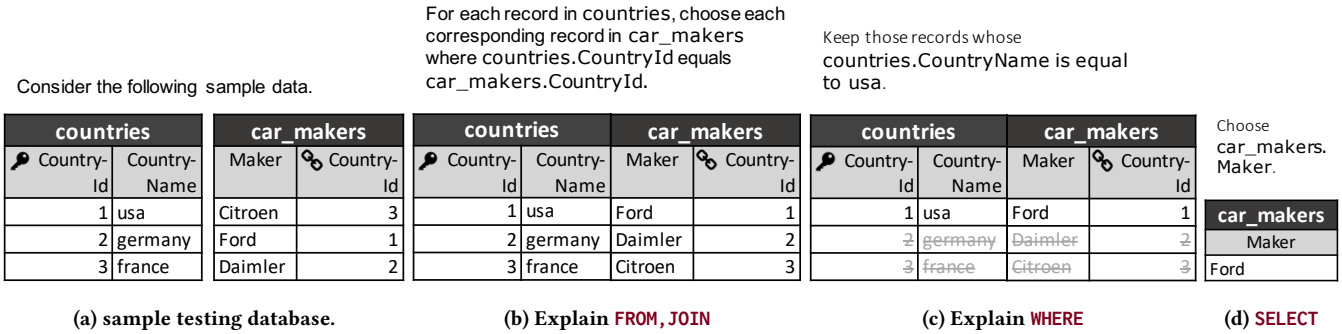
```
SELECT car_makers.Maker FROM car_makers
JOIN countries ON countries.CountryId=car_makers.CountryId
WHERE countries.CountryName='usa';
```

(b) Generated SQL query for the "Which car makers are American?" question

car_makers
Maker
Ford
Chrysler

(c) Answer on production database

Figure 2: An example natural language to SQL (NL2SQL) scenario.



(a) sample testing database. **(b) Explain `FROM,JOIN`** **(c) Explain `WHERE`** **(d) `SELECT`**

Figure 3: A query explained using the DIY technique.

(2) Generation of multi-step multi-modal explanations of the execution of a SQL query using natural language and tabular visualizations.
(3) Debug-It-Yourself (DIY), an interactive technique leveraging the above contributions for end-user debugging NL2SQL scenarios that helps users assess for correctness and potentially fix errors by themselves.
(4) A design probe that incorporates DIY and and its evaluation with 12 participants conducted to understand DIY's helpfulness in assessing the correctness of NL2SQL scenarios.

2 RELATED WORK

2.1 NL input and system output

Natural language is often ambiguous and underspecified, leading to interpretation errors. NaLIR [26] reveals these issues by mapping entities from the input query to the entities in the database schema and presenting them to the user using NL and dropdowns. Su et al.'s system [36] converts a Seq2Sql model API output into NL, augmented with GUI widgets that support error-fixing and disambiguation using fine-grained user interaction. DataTone [11] leverages mixed-initiative interaction through dropdown menus called "Ambiguity Widgets" to resolve ambiguities in the input query. Subsequent works leverage and extend these widgets: Eviza [31] handles quantitative magnitudes and time & space through range sliders; Evizeon [16] targets textual feedback, employing compact and interactive visualizations within the text; Orko [34] incorporates range sliders and interactive tooltips; and NL4DV [28] is a library that application developers can use to manage ambiguity in their applications. Inspired by all these works, DIY uses dropdown-menus to present ambiguities to the user for interactive disambiguation.

Visualizations have been used to explain a SQL query and its execution. QUEST [2] connects matching entities from a query's input to a database structure. QueryVis [25] automatically generates diagrams of SQL queries that capture their logical intent. Berant et al.'s cell-based provenance model explains the execution of a SQL query using provenance-based highlights on tabular visualizations (e.g., highlighting relevant cells that match a `WHERE` condition) [1]. DIY differentiates from these approaches by presenting multiple tabular visualizations that explain one SQL construct at a time (e.g., a `WHERE` clause), highlighting relevant elements along the way.

2.2 NL2SQL and SQL2NL

Semantic parsing of NL to SQL has recently surged in popularity thanks to the creation of dataset benchmarks such as WikiSQL [48], Spider [45], SParC [46], and CoSQL [44]. These dataset and associated benchmarks have led to the development of many deep learning models that address semantic parsing [4, 5, 7, 13, 20, 24, 27, 39, 41, 43, 47]. RAT-SQL [39] stands out by achieving state-of-the-art performance on the Spider dataset; hence we use it as our underlying NL2SQL engine to explore ways to help people assess the correctness of their NL2SQL interactions.

Research has also explored methods to translate SQL queries to natural language (SQL2NL). In the context of an NL2SQL system, this can be used to allow the "DMBS to talk back in the same language" as the users, allowing users to verify if their question was interpreted correctly [33]. Several SQL2NL strategies have been explored: Kokkalis et al. [22] and Elgohary et al. [9] employ a template-based approach while Su et al. [35] employ a grammar based approach. DIY leverages a template-based approach to explain the steps in the execution of an underlying SQL program.

In summary, our work leverages advances in NL2SQL and SQL2NL, and is further inspired by past interactive multi-modal techniques for resolving ambiguities. It builds on these ideas by applying them to a small but relevant sample testing database, providing a minimal example that demonstrates the query's behavior.

3 DIY: DEBUG-IT-YOURSELF

In this section we first enumerate our design goals, then present two example scenarios that illustrate how users can use DIY to assess and debug NL2SQL outputs. We then discuss the generation of sample testing databases, as well as generation of multi-modal explanations. Finally, we provide brief implementation details.

3.1 Design Goals

Our development of the DIY technique was driven by three key design goals. We compiled these goals based on a combination of reviewing design goals and system implementations of prior NLIs for visualization [11, 32] and databases [1, 26], and our own hypotheses of what will improve the overall user experience.

DG1. Explain the system's response to the question. DIY's users are not required to be fluent with database query language. Our goal is to design a system that explains how the system computes responses in an understandable manner for these users. This goal translates to explaining the query using a combination of natural language (NL) and visualizations.

DG2. Facilitate isolating errors. User-specified NL may include incorrect or partial references to the underlying data attributes which can lead to errors and ambiguities when translated into formal language (SQL). It was thus important to help users isolate these errors. Therefore, we convey the mappings between the entities in the question and the generated query to the user.

DG3. Facilitate fixing errors. In addition to isolating, our goal was to allow users to fix errors and resolve ambiguities upon discovery. This facilitates human-machine collaboration and avoids paraphrasing on the human's part. We therefore augment the question-query mappings with user interface controls that facilitate fixing errors and disambiguation.

3.2 Assessing and Debugging with DIY: Example Usage Scenarios

With the above design goals in mind, we designed DIY and embedded it into a QA shell (Figure 1). In the following two scenarios, we illustrate how DIY can help users assess the generated queries and answers for correctness and detect & fix errors.

3.2.1 Scenario 1, Fixing the Mapping: Chris, an automotive enthusiast without much knowledge on SQL, loads a *cars* database (Figure 2a) and asks *"What is the mean acceleration, minimum horsepower, and maximum displacement among all cars?"* (Figure 5). On reviewing the system's response, Chris notices that even though the system correctly identified the three superlatives (mean, minimum, and maximum) and attribute keywords (acceleration, horsepower, and displacement), it applied all superlative operators only to the *Horsepower* attribute. Convinced that the result is not correct, Chris expands the Debug View to repair the output. From the Detect Entities View, Chris notices the incorrect mappings, and selects the correct attributes from the respective drop-downs ([`cars_data.Accelerate` ▾] and [`cars_data.Edispl` ▾]). Based on these new mappings, the system automatically updates the sample data, and produces a new answer for inspection.

Next, Chris wants to verify if the superlatives were interpreted correctly. Chris sorts the records by `Horsepower` by clicking on the `Horsepower` column in the Sample Data View and verifies that the first (i.e., smallest) record matches the computed value in the Answer on the Sample Data View. After checking the *Max* operation in a similar way, Chris is convinced that the system now correctly performs the query. It is at this point that Chris notices a system alert indicating that the mapping changes have only been applied to the sample testing database, and that they must [Apply] or [Reject] them. Chris chooses to apply the changes and the Debug View closes. The answer on the Production Data updates accordingly.

3.2.2 Scenario 2, Checking the System Strategy: Being curious about the European automobile industry, Chris now asks the system, *"Which countries in Europe have more than 2 car manufacturers?"* (Figure 6). While checking the answer—*Germany, France*—they wonder based on prior knowledge why *Italy* was not included. To investigate this, Chris expands the Debug View and inspects the Detect Entities View. After confirming that the existing mappings are correct, Chris checks how the query is being executed on the sample testing database by reading through the four steps shown in the Explainer View: Step 1 joins the three tables; Step 2 removes non-European countries; Step 3 groups the rows by `countryId`; and Step 4 first counts the number of rows per `countryId` and then removes those groups that have less than or equal to 2 rows (indicated by ~~gray color and strike-through~~).

To verify the system's strategy, Chris decides to test it by manipulating the sample data. In the `car_makers` table of the sample testing database, they add a new row for the German car manufacturer "opel." On inspecting the updates to the subsequent steps, Chris confirms that "germany" now has three records, is no longer removed by step 4, and thus appears in the final answer for the sample data (step 5). Satisfied that Italy was likely excluded for having fewer than 3 records, Chris closes the Debug View [▴] without making or applying further changes.

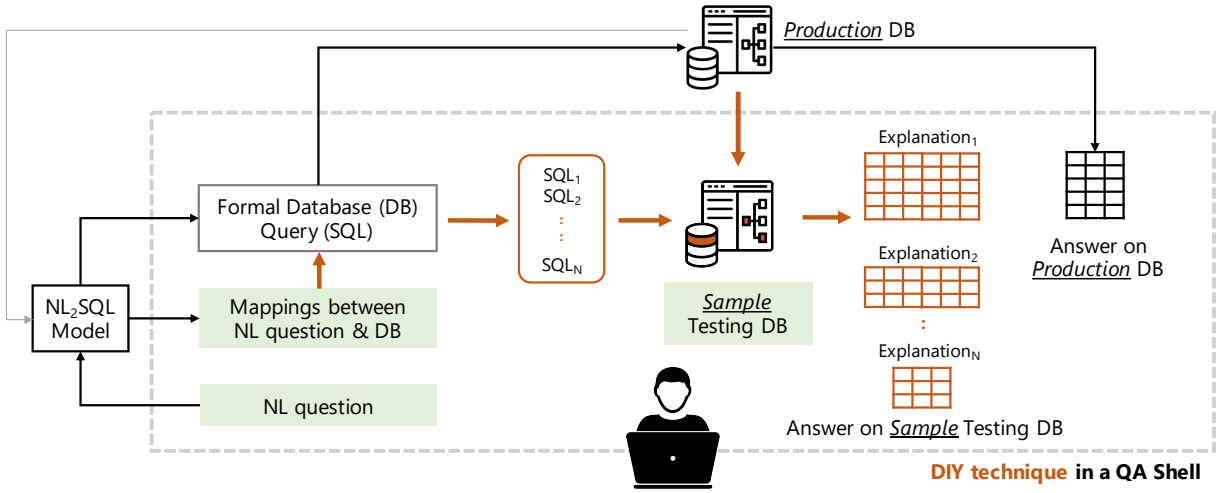


Figure 4: Overview of the DIY technique in a QA shell.

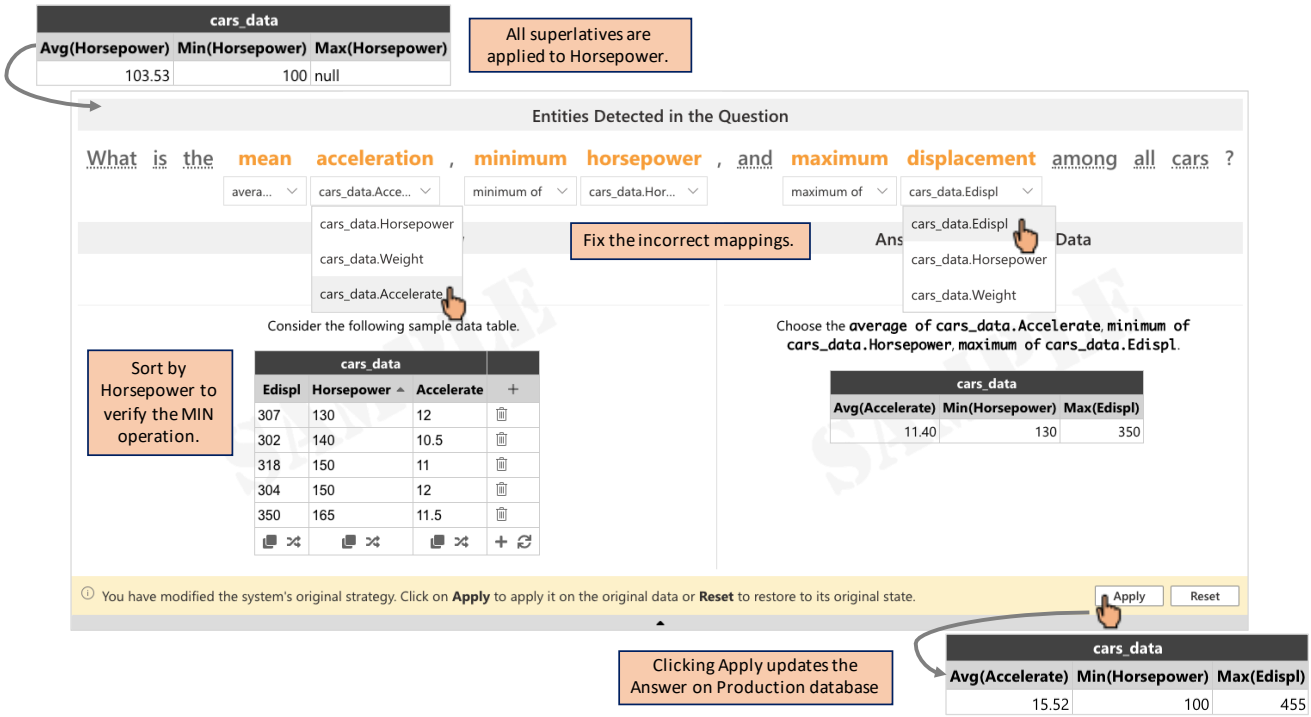


Figure 5: Scenario 1: DIY being used to correct a misclassified NL2SQL scenario.

3.3 Generating the Sample Data

DIY's key element and contribution is the use of a sample testing database to provide a sandbox for simplified inspection, testing and debugging. To generate the sample testing database, we clone the production database schema and show only those tables and columns that are part of the generated query. We then apply one of the following two strategies to populate each sample table with *five* records (we chose *five* based on feedback from pilot studies and UI design considerations with respect to visual clutter; currently this limit and sampling criteria are pre-configured).

3.3.1 Smart Constraints: To generate a *small-but-relevant* sample, the system first lists the entities and expressions in the query. Based on these, it identifies *smart constraints* that the data sampling algorithm must satisfy. For example, consider the SQL query: (`SELECT Id FROM cars_data WHERE Horsepower>200`). The expression (`WHERE Horsepower>200`) leads to a constraint requiring that *at*

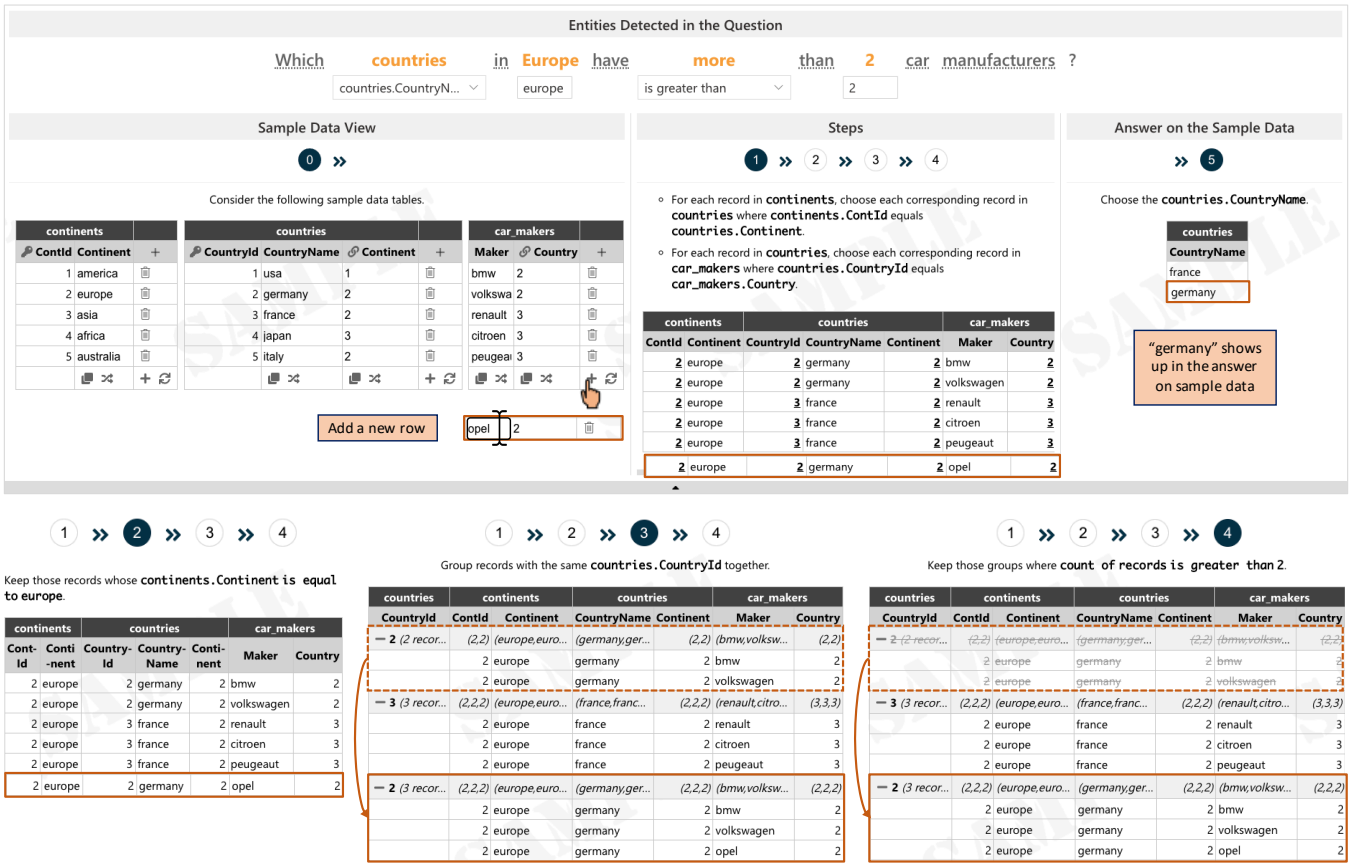


Figure 6: Scenario 2: DIY being used to understand and verify a complex NL2SQL scenario.

least one of the sample rows has *Horsepower>200* so that the final result set is non-empty. We also add a second constraint requiring that at least one row has *Horsepower≤200*. This ensures that both sides of the boundary condition are represented so that, when the relevant subqueries are executed on the sample testing database, the subanswers create before-after scenarios that help to visualize the effects of specific operations. Table 1 catalogs the *sample constraints* that we considered and implemented for various SQL constructs. We implemented those constraints that had primitive entities, for example, a simple `WHERE` clause, (`WHERE Horsepower>200`) comprises {"Horsepower", '>', 200}. On the other hand, both a subquery (e.g., `WHERE Horsepower>(SELECT AVG(Horsepower))`) and a `HAVING` construct (e.g., `HAVING AVG(Price)>2000`) require an additional computation step using a SQL engine. We did not implement these types of constraints.

3.3.2 Human-in-the-loop: For any generated query, it is not always possible to satisfy all *smart constraints*. This can be due to: (i) Practicality: Records that satisfy all constraints may not be common, and the database may not be structured to support efficient sampling of certain constraint combinations. In such cases, collecting five records may require a linear scan of the entire production database, and this may be too computationally costly to be practical in an interactive setting; (ii) Feasibility: satisfying certain constraints may be impossible. For example, the positive constraint for the question *"How many* ***car makers have their headquarters on Mars****?"* will be *car_makers.Headquarter="Mars"* which cannot be satisfied. In such scenarios, the system generates *partially-relevant* sample data. Users can then optionally modify the tables in the sample testing database to add records or to modify existing records to make them *more relevant*.

3.4 Generating Multi-modal Explanations

To break the SQL query into subqueries, we consider the order of execution of different SQL clauses. Each step generates a virtual table that is used as the input to the following step. If a certain clause is not specified in a query, the corresponding step is skipped. DIY considers only the forms of SQL queries output from the underlying NL2SQL model (Listing 1). This is a subset of all valid SQL queries (e.g., it excludes the clauses `TOP` and `WITH`).

3.4.1 Logical Order of Execution of a SQL query: As shown in Listing 1, the `FROM` clause and the subsequent `JOIN`s are executed first to determine the working set of data. Next, the `WHERE` constraints are applied to the individual rows, discarding the rows that do not satisfy the constraints. The remaining rows are then grouped based on common values as specified in the `GROUP BY` clause. If the query has a `HAVING` clause, it is then applied to the grouped rows — the

Table 1: Smart Constraints: A catalog of constraints to generate a sample testing database that can effectively explain the execution of the SQL query. IEU* stands for INTERSECT, EXCEPT, UNION SQL keywords. SoI stands for Status of Implementation.

SQL Entity	Constraint Operation	SoI
SELECT	Choose all columns mentioned in the SELECT clause.	✓
FROM	Choose all tables mentioned in the FROM clause.	✓
JOIN	Choose records from each *to-be-joined* table such that the *joined* state has at least one record.	✓
GROUP BY	Choose records such that the *grouped-by* columns have duplicate values.	✓
HAVING	Choose records such that the *grouped-by* state satisfies the HAVING expression(s).	×
WHERE	Choose records such that at least one satisfies the WHERE condition, and at least one fails.	✓
DISTINCT	Choose records such that the *grouped-by* column has duplicate values.	✓
LIMIT	Choose records such that the result set has enough records to apply the LIMIT operation.	✓
IEU*	Choose records such that the execution of the subqueries have intersecting subanswers.	×
Subquery	Choose records such that the execution of this subquery produces a non-empty final result set in the query.	×
Functions	Aggregation functions (COUNT, SUM)	×
Operators	Wildcards (*, %), LIKE	×

(6) SELECT (7) DISTINCT *select_list*
(1) FROM *left_table*
(2) *join_type* JOIN *right_table*
ON *join_condition*
(3) WHERE *where_condition*
(4) GROUP BY *group_by_list*
(5) HAVING *having_condition*
(8) ORDER BY *order_by_list*
(9) LIMIT *count* (10) OFFSET *count*
(12) *left_SQL* IEU* *right_SQL*
(11) *right_SQL*;

Listing 1: General form of a SQL query, with step numbers assigned according to the order in which the different clauses are logically processed. *left_SQL* & *right_SQL* represent SQL queries with Steps 1-10. IEU* stands for INTERSECT, EXCEPT, UNION.

(I) *left_SQL*
(a) SELECT * FROM JOIN;
(b) SELECT * FROM JOIN WHERE;
(c) SELECT * FROM JOIN WHERE GROUP BY;
(d) SELECT * FROM JOIN WHERE GROUP BY HAVING;
(e) SELECT *select_list* FROM JOIN WHERE GROUP BY HAVING;
(f) SELECT DISTINCT *select_list* FROM JOIN WHERE GROUP BY HAVING;
(g) SELECT DISTINCT *select_list* FROM JOIN WHERE GROUP BY HAVING ORDER BY;
(h) SELECT DISTINCT *select_list* FROM JOIN WHERE GROUP BY HAVING ORDER BY LIMIT OFFSET;
(II) *right_SQL*
(III) *left_SQL* IEU* *right_SQL*;

Listing 2: Sequence of subqueries generated by DIY at each step of the Explainer View for a general SQL query represented in Listing 1. The underlined text shows the difference with the previous subquery.

groups that do not satisfy the constraints are discarded. Next, the expressions in the SELECT clause are computed. This may include columns, or aggregation of functions, or subqueries. If a DISTINCT keyword is present, duplicate records are discarded. Likewise, if an ORDER BY clause is present, the rows are sorted accordingly. Finally, the rows that fall outside the range specified by LIMIT and OFFSET clauses are discarded, leaving the final result set.

3.4.2 NL explanations: We follow a heuristics-based approach to generate NL explanations for each step in the Explainer View. One notable aspect of these explanations is that they explain the *difference* between the current and the previous subquery. For example, consider two consecutive subqueries: (i) (SELECT * FROM cars_data) and (ii) (SELECT * FROM cars_data WHERE Horsepower>200). The generated NL explanation for subquery (i) is *"Choose all columns from the cars_data table."* and for subquery (ii) is, *"Keep those records whose* Horsepower *is more than 200."* We hypothesize that this approach can help users to not only understand each step but also enable them to detect and isolate specific errors. Table 2 shows the complete list of templates that are currently being used to explain each subquery.

3.4.3 Tabular Visualizations: An interactive tabular visualization complements each NL explanation displaying the result *after* the corresponding subquery is executed on the sample testing database. Table headers communicate the table names and column names of data values. Each table is treated as the input to the following step. For example, observe *steps 2, 3, and 4* in Figure 6. Step (2), explains the WHERE clause, and the rows that do not satisfy the corresponding constraints are ~~faded out and striked-through~~. Similarly, Step (3), explains the GROUP BY clause, and the grouped records are visually grouped and indicated accordingly.

3.5 Implementation Details

We use RAT-SQL [39], an NL2SQL semantic parsing framework with state-of-the-art performance as our backend. It uses *relation-aware self-attestation* and encodes the names of columns and tables, as well as the values of data, into a common dense representation.

We implemented the DIY technique as well as the QA shell using the ReactJS framework (https://reactjs.org) making API requests to the RAT-SQL model over HTTP REST. We instantiated and managed

Table 2: NL Explanation templates. Each template scales to multiple instances (e.g., two `WHERE` clauses) using punctuations (e.g., ',') and conjunctions (e.g., 'and').

SQL keyword	Natural Language Template
`FROM`	Choose columns from the {table} table.
`FROM` + `JOIN`	For each record in {$table_1$}, choose each corresponding record in {$table_2$} where {$column_1$} {operator} {$column_2$}.
`WHERE`	Keep those records whose {column} {operator} {value}.
`GROUP BY`	Group records with the same {column} together.
`HAVING`	Keep those groups where {aggregation} of {records/column} {operator} {value}.
`SELECT`	Choose the {column}.
`DISTINCT`	Keep unique records.
`ORDER BY`	Sort the records by {column} in the {orderType} order
`LIMIT`	Choose the first {N} record(s).
`INTERSECT`	Choose all records that are common to the answers of Step {M} and Step {N}.
`EXCEPT`	Choose all records from the answer of Step {M} that are not in the answer of Step {N}.
`UNION`	Combine all records from the answer of Step {M} and the answer of Step {N}.

databases in an instance of the SQLite database in the user's web browser using the sql.js library (https://sql.js.org).

4 USER STUDY

After developing the DIY prototype and receiving an approval from our ethics board, we conducted an exploratory user study and design probe with 12 participants. With this study, we aim to understand how users utilize DIY to assess the generated results for correctness, and detect and fix errors in NL2SQL scenarios. In the following sections we describe the participants, detail the high-level procedure, and present the specific study tasks. We then present and discuss our findings.

4.1 Participants, Procedure, and Tasks

We recruited 12 participants (4 female, 7 male, 1 preferred not to say). They worked for a large technology company in different roles including UX Designers, Design Researchers, Site Reliability Engineers, Data Scientists, Cloud Solution Architects, Program Managers, and Research Interns. We compensated each participant with a $25 Amazon Gift card.

Due to the COVID-19 pandemic, we leveraged numerous Internet collaboration tools to conduct the study remotely. Participants were asked to complete a brief online demographics questionnaire, and to connect with the experimenter using the Microsoft Teams teleconferencing software. Participants were then quickly briefed about the study, and were presented with a 5 minute tutorial video that demonstrated the features of DIY. Following the video, the experimenter provided participants access to the study environment by sharing the study computer's screen and granting input control. Participants were then asked to complete 8 tasks of varying difficulty using DIY, and to think out loud while interacting with the system. Participants were free to ask questions at any time, and the experimenter occasionally asked questions to probe participants' strategies. The study ended with a debriefing in which participants completed a system usability score (SUS) [6] questionnaire, discussed their overall experience with the system, and provided suggestions for improvements. The entire session took 90 minutes to complete. All sessions were screen-recorded, and transcripts were later generated using automated software.

The eight tasks were organized into sections according to complexity: 3 easy, 2 medium, and 3 hard tasks (Table 3). We determined the complexity based on the count and types of SQL clauses (e.g., `GROUP BY`, `INTERSECT`, `MIN()`) and the count and types of errors in the generated SQL query (e.g., wrong operator, missing column). For example, the query corresponding to **Task #6** in Table 3 is `hard` because it has two `JOIN`s and one each of: `WHERE`, `GROUP BY`, `HAVING`, and `SELECT`. This methodology was inspired from the one used by the Spider dataset benchmark [45]. Within each complexity level, half the tasks resulted in DIY presenting correct results which participants may nonetheless seek to verify; and half the tasks resulted in NL2SQL translation errors that needed to be corrected. We curated these tasks by asking different types of questions to the RAT-SQL model and inspecting the responses. We include an equal number of correct and incorrect responses, spanning a range of SQL complexities and error types.

4.2 Results

Our observations revealed the benefits of using sample data to help users assess the correctness of the system's responses, and both a range of debugging rationales and strategies across participants. We present these observations below, and discuss observations that could indicate where DIY may benefit from additional refinement.

4.2.1 General Reactions: Overall, participants liked DIY's approach of using sample data to explain the system's strategy. P10 commented, *"It is important to have this transparency and to show people how the system is working and to let them control it. This is a great example of that."* P8 commented, *"I think it's really cool and I think there are a lot of customers who would benefit from something like this."* P14 liked the multi-modal explanations, commenting *"I really liked your idea of the linear stuff...kind of a visual explanation of the of the query path."* Also, participants rated their experience with an average SUS of 65.42 (Figure 7): while this is encouraging, further refinement is possible.

4.2.2 Debugging Rationales: Our system's initial response highlighted important tokens in the question and the computed answer on the production database (Figure 1B,C). Participants inspected these first and then, based on their assessment, optionally chose to

Table 3: Eight tasks used in the study.

No.	Question	Error	Error Type	Complexity
1	What is the mean acceleration, minimum horsepower, and maximum displacement among all cars?	Yes	Wrong columns	Easy
2	What is the average acceleration of cars each year?	No	-	Easy
3	Which products are manufactured in Austin?	Yes	Wrong column	Easy
4	Which products by Sony are priced above 100?	No	-	Medium
5	Which car models are produced since 1980?	Yes	Wrong operator	Medium
6	Which countries in Europe have more than 2 car manufacturers?	No	-	Hard
7	Which continent has the most car makers? Also list the count.	Yes	Missing column	Hard
8	Which car models are lighter than 3500 or built by BMW?	No	-	Hard

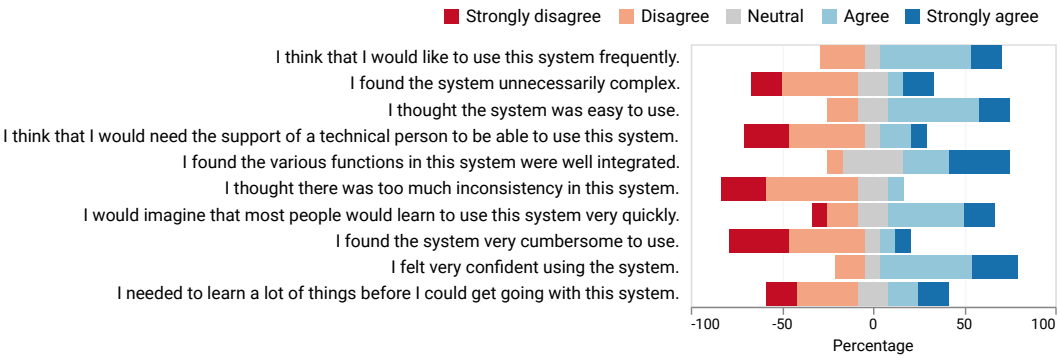


Figure 7: SUS Scores reported by participants.

expand the Debug View. We observed different rationales for electing to debug, including: (i) the option was available (*"Just because I can!"* – P8), (ii) they detected an error (everyone), (iii) they just wanted to double-check the answer or strategy (almost everyone, *"I want to verify the Average."* – P1), (iv) they did not have enough domain expertise to trust the answer on production database (*"I am not good with cars"* - P1), (v) they had some domain expertise that led them to suspect the answer (*"I can think of a couple more rows so I'm just gonna verify"* - P8). At times, participants did not utilize the Debug View because: (i) they had begun to trust the system (*"I will probably not verify, I trust the system by now."* – P1, *"Assuming the math is correct, this seems fine."* – P7), (ii) they were satisfied with the orange highlights in the Annotated Question View (*"it looks to me that it highlighted the right keywords"* – P5, P14). Many participants also expressed a need for additional context to interpret the answers, even if that context was not explicitly requested in the original question. For example, for the question "Which products by Sony cost more than $100?" the system's response returned only `Products.Name` values. P1, P7, and P17 wished to see more columns, including the manufacturer and price, to facilitate inspection. Likewise, P7 suggested adding rows that do not match the criteria, and striking them out using the same visual convention employed by the DIY Explainer View.

4.2.3 Strategies: With the Debug View, participants employed three broad strategies to verify the query. In the first category were participants who expressed concerns about modifying the sample data, and predominantly utilized inspection (e.g., of term mappings and explanations) to assess the correctness of the query. In the extreme cases, three participants (P1, P16, P17) did not modify the sample testing database at all. P1 commented, *"My judgement is based on the result I see, if I manipulate the data, I don't trust the result anymore, and I don't trust the system anymore."*

In the second category were participants who modified the sample data to explore counterfactual *what-if scenarios*. Specifically, participants modified sample data to (i) generate positive (or negative) scenarios (*"Now that I have been able to generate an affirmative case, I am more happy with this"* – P8) or (ii) to test specific boundary conditions (*"I was basically playing for boundary conditions"* – P7, *"I want to make sure I have tested the right boundary conditions"* – P15). We observed participants manipulate the sample testing database in several different ways. One participant chose to *delete irrelevant rows* from the sample data (*"I might as a matter of figuring this out remove everybody I don't care about"* - P8). One participant chose to *add a new test row* (*"as I did not want to manipulate existing data"* - P15). One participant sorted the sample data tables to verify the *MAX* and *MIN* superlatives in the question. Most participants edited specific cells in the sample testing database, e.g., *"ford"* to *"bmw"* to verify a `WHERE` clause, or change the `CountryId` from *1* to *2* to create a successful `JOIN`.

Finally, in the third category participants manipulated the mappings in the Detected Entity View to, for example, test boundary conditions. One participant modified the operator mappings "is greater than" to "is less than" to test a reverse scenario (P8). Another participant changed the attribute *Price* to *Revenue* to verify the query response updated accordingly. This strategy is interesting because modifying the mappings changes how the system interprets the user's original question. Accordingly, these affordances

were intended for *fixing errors* or *resolving ambiguities*. Some participants were aware of this and planned to revert to the original mapping after testing. Other participants were reminded by the notification at the bottom of the Debug View.

4.2.4 Areas for improvement: Finally, though participants were generally positive about the system, we identified several areas of improvement in both the DIY technique and the QA shell.

Confusion between production and sample data. The DIY technique at present presents two distinct answers for any given query: one for the production database, and a second for the sample testing database. At multiple points during the study, participants (P1, P8, P11, P14) exhibited confusion as to why the two answers did not match. P11 commented, *"OK, so it says monitor here (in the Answer on Sample Data View), which is what I was expecting. Why does it say CD drive, DVD drive (in the Answer on Production Data View)?"* They failed to recognize that the sample testing database is a very small subset of the production database.

Generating a smarter sample testing database. The sample data generation module (Section 3.3) identifies entities from the query and defines constraints that, if satisfied, would generate a relevant sample. As discussed earlier, practicality and feasibility related restrictions further constrain sampling. Participants pointed out this limitation when they encountered a sample testing database that they felt they need to modify further to enable relevant debugging. P1 commented, *"For me, to build trust with the system, I would want the system to be smart enough and return sample data relevant to the question."* They went on to suggest the human to be more involved in the generation of the sample data, *"I wonder if I could tell the system to return sample data post 1980 so then I can verify if the answer is indeed correct."*

Improving the multi-modal explanations. Some participants found it challenging to follow the explanations for certain SQL constructs. For example, P5 did not understand multi-table `JOIN` conditions. P7 worried that the use of (too many) *IDs* in the sample testing database and the `JOIN` condition resulted in added complexity. Some participants failed to interpret compound SQL clauses (e.g., `UNION`) as it was presented in a linear manner just like other `SQL` clauses. We will explore alternate representations for these SQL clauses (e.g., representing subqueries in a tree-like representation, and using animations to visualize multi-table join operations).

5 DISCUSSION AND FUTURE WORK

5.1 Notable Observations

5.1.1 Handling ambiguities between conversational and formal language: For one of the task questions, *"Which car models are produced since 1980?,"* the NL2SQL system mapped the token "since" to the "greater than" operator. In colloquial conversation, "since" often implies a "greater than or equal to" operation, and thus this mapping needed to be fixed. It was interesting that six participants (P1, P7, P11, P12, P15, P17) pointed out this ambiguity, commenting that it is sometimes up to the user's interpretation. Nevertheless, only P7 and P17 modified the mapping.

5.1.2 Leveraging manipulation to facilitate understanding: As mentioned earlier, some participants modified the mappings between the question and the database entities to either test a boundary condition (e.g., *is greater than* → *is less than*) or to observe a change (e.g., *Price* → *Revenue*). This was interesting as the participants deliberately modified what were already correct mappings. We envision this to be an opportunity to support data exploration within the Debug View. For example, consider a scenario wherein a user first asks for cars with Acceleration>100 and upon inspecting the answer, is interested in cars with Acceleration>200 instead. Answering this question in a QA system generally involves paraphrasing the original question.

5.2 Limitations and Future Work

5.2.1 System Limitations: While the system supports modifying existing mappings from question tokens to database columns or operators, it is more limited in what new mappings can be added. For example, unmapped tokens may only be mapped to columns previously implicated by the system's original interpretation of the question. Likewise, the sample data generation and the explanation generation modules currently do not support all SQL constructs. For example, neither module generates *smart constraints* or *multi-modal explanations* for window functions (e.g., `OVER`) or wildcard operators (e.g., `LIKE`, `%`), since the NL2SQL backend does not currently support these constructs.

5.2.2 Usability and Design: We have identified several improvements to the Debug View design. For example, in scenarios involving multiple tables and steps, the current horizontal layout was not easy to navigate. We are considering a switch to a vertical layout, or showing views only on demand. Likewise, we are exploring alternate layouts to better distinguish between the sample testing database and the production database. In addition, it would be useful to allow users to add columns to the Explainer View. This will enable the creation of new mappings to columns not previously implicated by the system's original interpretation of the question.

5.2.3 Teach SQL:. We believe that NL2SQL can be a powerful tool for teaching SQL and that DIY can be extended to better support this use case. Existing tools (e.g., *SQL Fiddle* [10], *Tryit Editor* [38] already provide users with a sandbox for executing SQL queries on existing data sets. Adding DIY could provide a natural language interface to help novices formulate SQL queries, and DIY's step-by-step multi-modal explanations may also offer pedagogical value.

6 CONCLUSION

Debug-It-Yourself (DIY) is an interactive technique that helps users to assess NL2SQL scenarios for correctness and, if possible, fix errors. DIY provides users with a sandbox where they can interact with (1) the mappings between the question and the generated query, (2) a *small-but-relevant* subset of the underlying database, and (3) multi-modal explanations of the generated query. In an exploratory user study, we investigated how DIY helps users assess the correctness of the system's answers, and we discussed how our observations revealed insights about different end-user debugging strategies, as well as the challenges in supporting such scenarios.

ACKNOWLEDGMENTS

We thank Ahmed Elgohary, Ahmed Hassan Awadallah, Ameesh Shah, Ben Zorn, Christian König, Chris Meek, Matthew Richardson, Oleksandr Polozov, Tao Yu, and Xiang Deng for their feedback.

REFERENCES

[1] J. Berant, D. Deutch, A. Globerson, T. Milo, and T. Wolfson. 2019. Explaining Queries Over Web Tables to Non-experts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1570–1573. https://doi.org/10.1109/ICDE.2019.00144

[2] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo Lado, Yannis Velegrakis, et al. 2013. QUEST: a keyword search system for relational data based on semantic and machine learning techniques. (2013).

[3] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. Soda: Generating sql for business users. *Proceedings of the VLDB Endowment* 5, 10 (2012), 932–943.

[4] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global Reasoning over Database Structures for Text-to-SQL Parsing. arXiv:1908.11214 [cs.CL]

[5] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. arXiv:1905.06241 [cs.CL]

[6] John Brooke. 2013. SUS: a retrospective. *Journal of usability studies* 8, 2 (2013), 29–40.

[7] DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2020. RYANSQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases. arXiv:2004.03125 [cs.CL]

[8] Jonathan Danaparamita and Wolfgang Gatterbauer. 2011. QueryViz: helping users understand SQL queries and their patterns. In *Proceedings of EDBT*. 558–561.

[9] Ahmed Elgohary, Saghar Hosseini, and Ahmed Hassan Awadallah. 2020. Speak to your Parser: Interactive Text-to-SQL with Natural Language Feedback. *arXiv:2005.02539* (2020).

[10] Jake Feasel. 2021. SQL Fiddle. http://sqlfiddle.com/, accessed 2021-01-01.

[11] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. 2015. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of ACM UIST*. 489–500.

[12] Barbara J Grosz, Douglas E Appelt, Paul A Martin, and Fernando CN Pereira. 1987. TEAM: an experiment in the design of transportable natural-language interfaces. *Artificial Intelligence* 32, 2 (1987), 173–243.

[13] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. arXiv:1905.08205 [cs.CL]

[14] Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-SQL: reinforce schema representation with context. *arXiv:1908.08113* (2019).

[15] Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TAPAS: Weakly Supervised Table Parsing via Pre-training. *arXiv:2004.02349* (2020).

[16] Enamul Hoque, Vidya Setlur, Melanie Tory, and Isaac Dykeman. 2018. Applying pragmatics principles for interaction with visual analytics. *IEEE TVCG* 24, 1 (2018), 309–318.

[17] Mohd Ibrahim and Rodina Ahmad. 2010. Class diagram extraction from textual requirements using natural language processing (NLP) techniques. In *2010 Second International Conference on Computer Research and Development*. IEEE, 200–204.

[18] Jan-Frederik Kassel and Michael Rohs. 2018. Valletto: A multi-modal Interface for Ubiquitous Visual Analytics. In *ACM CHI '18 Extended Abstracts*.

[19] Esther Kaufmann, Abraham Bernstein, and Lorenz Fischer. 2007. NLP-Reduce: A naive but domainindependent natural language interface for querying ontologies. In *4th European Semantic Web Conference ESWC*. 1–2.

[20] Amol Kelkar, Rohan Relan, Vaishali Bhardwaj, Saurabh Vaichal, and Peter Relan. 2020. Bertrand-DR: Improving Text-to-SQL using a Discriminative Re-ranker. arXiv:2002.00557 [cs.CL]

[21] Dae Hyun Kim, Enamul Hoque, and Maneesh Agrawala. 2020. Answering Questions about Charts and Generating Visual Explanations. In *Proceedings of ACM CHI*. :1–:13.

[22] Andreas Kokkalis, Panagiotis Vagenas, Alexandros Zervakis, Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2012. Logos: a system for translating queries into narratives. In *Proceedings of ACM SIGMOD*. 673–676.

[23] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. 2010. Explaining structured queries in natural language. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 333–344.

[24] Dongjun Lee. 2019. Clause-Wise and Recursive Decoding for Complex and Cross-Domain Text-to-SQL Generation. arXiv:1904.08835 [cs.CL]

[25] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, HV Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-based diagrams help users understand complicated SQL queries faster. In *Proceedings of ACM SIGMOD*. 2303–2318.

[26] Fei Li and Hosagrahar V Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *Proceedings of ACM SIGMOD*. 709–712.

[27] Kevin Lin, Ben Bogin, Mark Neumann, Jonathan Berant, and Matt Gardner. 2019. Grammar-based Neural Text-to-SQL Generation. arXiv:1905.13326 [cs.CL]

[28] A. Narechania, A. Srinivasan, and J. Stasko. 2021. NL4DV: A Toolkit for Generating Analytic Specifications for Data Visualization from Natural Language Queries. *IEEE TVCG* 27, 2 (2021), 369–379. https://doi.org/10.1109/TVCG.2020.3030378

[29] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of ACL IJCNLP*. 1470–1480.

[30] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of ACM IUI*. ACM, 149–157.

[31] Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. 2016. Eviza: A natural language interface for visual analysis. In *Proceedings of ACM UIST*. 365–377.

[32] Vidya Setlur, Melanie Tory, and Alex Djalali. 2019. Inferencing underspecified natural language utterances in visual analysis.. In *Proceedings of ACM IUI*. 40–51.

[33] Alkis Simitsis and Yannis Ioannidis. 2009. DBMSs should talk back too. *arXiv:0909.1786* (2009).

[34] Arjun Srinivasan and John Stasko. 2018. Orko: Facilitating multi-modal interaction for visual exploration and analysis of networks. *IEEE TVCG* 24, 1 (2018), 511–521.

[35] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. 2017. Building natural language interfaces to web apis. In *Proceedings of ACM CIKM*. 177–186.

[36] Yu Su, Ahmed Hassan Awadallah, Miaosen Wang, and Ryen W White. 2018. Natural language interfaces with fine-grained user interaction: A case study on web apis. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 855–864.

[37] Yiwen Sun, Jason Leigh, Andrew Johnson, and Sangyoon Lee. 2010. Articulate: A semi-automated model for translating natural language queries into meaningful visualizations. In *Proceedings of the International Symposium on Smart Graphics*. 184–195.

[38] W3Schools. [n.d.]. SQL Tryit Editor v1.6. https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all, accessed 2020-12-29.

[39] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of ACL*. Online, 7567–7578.

[40] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust Text-to-SQL generation with execution-guided decoding. *arXiv:1807.03100* (2018).

[41] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. arXiv:1711.04436 [cs.CL]

[42] Bowen Yu and Cláudio T Silva. 2019. FlowSense: A natural language interface for visual data exploration within a dataflow system. *IEEE TVCG* 26, 1 (2019), 1–11.

[43] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-DomainText-to-SQL Task. arXiv:1810.05237 [cs.CL]

[44] 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. arXiv:1909.05378 [cs.CL]

[45] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv:1809.08887* (2018).

[46] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. SParC: Cross-Domain Semantic Parsing in Context. arXiv:1906.02285 [cs.CL]

[47] Rui Zhang, Tao Yu, He Yang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-Based SQL Query Generation for Cross-Domain Context-Dependent Questions. arXiv:1909.00786 [cs.CL]

[48] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).