A NATURAL LANGUAGE INTERACTIVE INTERFACE FOR SQL QUERY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We contribute `nalini`, an natural-language based interactive interface for SQL query generation. Motivated by a lack of usability of existing systems, `nalini` was built with the intention of using it for complex query generation. The interface allows users to use natural language fragments with minimal structure; users separately describe each desired column in the output table and can optionally describe additional filters. We evaluated `nalini` with a first-use study with five participants, where participants were asked to generate queries from the TPC-H decision support benchmark. Our study showed that users were able to use `nalini` to generate complex queries, including TPC-H queries which cannot be generated by any of the leading natural-language-to-SQL tools, and points to promising areas of future work.

1 INTRODUCTION

Ever since the relational data model was pioneered decades ago (Codd, 1970), querying data in relational databases has become an increasingly common operation, with SQL emerging as the standard query language (Li & Jagadish, 2014).

In today's data-driven world, many new SQL users are non-experts (Li & Jagadish, 2014). SQL queries can be difficult to write; the process typically requires expert knowledge of both the query language and the specific data being queried. Perhaps the most obvious barrier to writing an accurate SQL query is the requisite knowledge of the data: its format, its exact schema, and the precise relationships necessary to create join paths. Also, as is necessary for any language, working with SQL requires that the programmer understands its syntax and semantics. Although there is a standard version of SQL, no database vendor is fully compatible with the standard. Numerous dialects of SQL are used today, each of which has its own syntax variations regarding dates and times, string representation, column aliasing, aggregate operations, and case sensitivity (Khalil, 2022). Even an experienced SQL user may find it challenging to write a query in a new environment.

An important goal of the database community is thus to enable non-experts to easily write accurate, executable queries based on their specifications. There has been a lot of work in this area, including using a combination of rule-based and deep-learning natural language approaches and programming-by-example approaches. Some approaches include an interactive component as well. While the advances in NLP-to-SQL research have steadily brought us closer to a fully-automated solution, there is still a lot of work to be done. A 2020 study of 12 state-of-the-art NLP-to-SQL methods showed that although natural language methods have performed well on specific databases or against specific benchmarks, their performance significantly degrades when applied to complex queries (Kim et al., 2020). These systems, including Templar (Baik et al., 2019), NSP (Guo et al., 2018), SQLNet (Xu et al., 2017), TypeSQL (Yu et al., 2018a), SyntaxSQLNet (Yu et al., 2018b), GNN (Chen et al., 2021), IRNet (Guo et al., 2019), and NaLIR (Li & Jagadish, 2014), use a variety of deep learning approaches. However, they all have the same fundamental assumption that a SQL select statement can be uttered in a single sentence. All of these sytems showed 0% accuracy when tested against the TPC-H benchmark. In reality, many select statements are written as part of data pipelines and generate entire tables, with complex business logic applied to many separate columns. These specifications are not natural to express in one-sentence queries.

On the other hand, programming-by-example interfaces, such as SCYTHE (Wang et al., 2017) and PATSQL (Takenouchi et al., 2021) have demonstrated better performance on complex queries, but are not as easy to intuitively use. For a non-technical user, it can be extremely cumbersome to create input/output tables to communicate the intention. Thus, we motivate the development of a tool that combines the effectiveness of program-synthesis based work with the ease-of-use of natural-language based solutions.

1.1 CONTRIBUTIONS

In this work, we develop `nalini`, a natural language interactive interface for SQL query generation. In contrast to other natural language based SQL tools, `nalini` does not answer a single question posed by the user. Instead, the interface has a query generation panel where users provide individual natural language descriptions of each column in their desired output table as well as optional filter descriptions.

We evaluated `nalini`'s efficacy through a first-use study with 5 participants. The results of the study demonstrate `nalini`'s success as a proof of concept. The novel minimally-structured format of the natural language input, along with the use of meaningful error messages enabled `nalini` to achieve the following desired outcomes:

- **Participants interact using natural language.** After seeing a few examples of how `nalini` could be used, all participants interacted with `nalini` using a combination of English words and phrases, database-specific table and column names, and mathematical expressions, without requiring specific directions or documentation.
- **Participants completely and correctly generated SQL queries.** Unlike any of the existing NLP-based solutions, with a limited scope and human interactive loops, users were able to successfully generate complex queries from the TPC-H decision support benchmark.

2 NALINI

2.1 SAMPLE USE CASE

In this section, we provide a high-level overview of our technique and its implementation in `nalini` through a simple motivating example. The example is loosely based on Query 7 from the TPC-H decision support benchmark (tpc), and the data follows the TPC-H schema and constraints. See Figure 1 for the full schema.

Our user, Michael, is a regional manager at large paper supply corporation who doesn't have a lot of experience with SQL. His boss expressed concern that in 1995, the company's supply chain was particularly inefficient because of their international shipping methods. In particular, Michael's boss wants him to look into the amount of international shipping that was done via trucks.

Michael decides to look into the historical data. Rather than exporting data into a spreadsheet to do a one-time analysis, Michael explicitly wants to create a new table in his business intelligence pipeline, so that other analysts and stakeholders can trace the inputs of his analysis and use his table for further analysis. Michael sets out to create a table that calculates, for the calendar year 1995, the total gross discounted revenue derived from sales that involved items being shipped from a supplier in one country to a customer a different country, and that were shipped via trucks.

Previously, he might have requested the help of a data scientist, perhaps based out of a different office location, to generate and execute the SQL query for him. Instead, he opens up `nalini` to write the query on his own (see Figure 4).

Michael knows that he wants his output table to have three columns: `customer_nation`, `supplier_nation`, and `revenue`. He clicks the "Add Column" button in the Query Generation panel twice to create a total of three columns. For the first column, he types `customer_nation` into the "column name" input and types "customer nation name" into the column description area beside it. Similarly, he types `supplier_nation` into the second "column name" input and types "supplier nation name" into the second column description area. He names his third column `revenue`, and then realizes that he actually isn't sure how to calculate gross revenue.

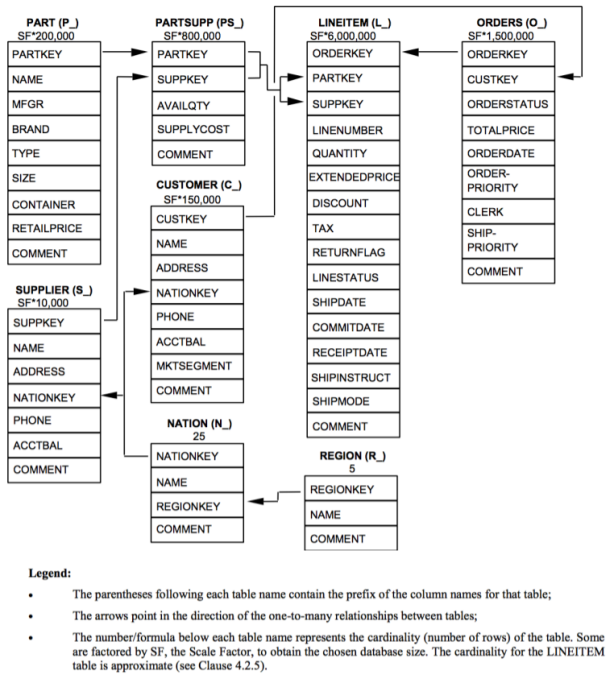

Figure 1: The TPC-H schema, as specified in the original benchmark (tpc). This figure was created by the Transaction Processing Performance Council.

After checking with a salesperson at his branch, Pam, he confirms that to calculate gross discounted revenue, he needs to multiply the sale price of each lineitem by `1` minus the discount percentage and then add up the discounted revenues for each lineitem to get the gross discounted revenue. Scrolling through the columns of the `lineitem` table, he sees that there are in fact columns named `l_extendedprice` and `l_discount` which correspond to the sale price and discount percentage, respectively. He types `total lineitem extendedprice * (1-discount)` into the description area for the `revenue` column.

At present, Michael's description will include items shipped within a single nation, so Michael clicks the "Add filter" button and types `customer nation name is not supplier nation name` into the filter description field. He also needs to consider only lineitems that were shipped in 1995, so he adds another filter with the description `ship date must be in 1995`. Finally, he realizes he needs to check that the lineitems whose revenue he is summing were shipped via truck. He recalls that there is a column that would say "`TRUCK`" if the shipping method was truck-based delivery, but he isn't sure what table that row is in. It would be time-consuming for him to check. Michael knows that he can add his new filter condition as a standalone filter, like his date filter, or as part of one of the column descriptions. In this case, he adds a description of that filter to his description of the revenue column so that it now reads `total lineitem extendedprice * (1-discount) for lineitems with shipping method "TRUCK"`.

Michael clicks the "Generate SQL!" button under his specifications, and eagerly awaits his completed query. However, `nalini` does not generate a query - instead, under the text field where he specified the date filter, it reads *Unable to resolve: "ship date must be in 1995"*. Michael edits his filter description to be `ship date on or after 1/1/1995 and ship date before 1/1/1996` and clicks "Generate SQL!" once again.

This time, his completed query, detailed in Figure 2, appears in the UI. Michael examines the generated query.

The `SELECT` clause has three columns as expected; the `customer_nation` and `supplier_nation` columns are selected as the `n_name` columns from two different instantiations of the `nation` table, `nation1` and `nation2`. These two columns also appear in the `GROUP`

```
SELECT nation2.n_name AS customer_nation,
       nation1.n_name AS supplier_nation,
       SUM((l_extendedprice * (1 - l_discount))) AS revenue
FROM supplier
    JOIN nation nation1 ON s_nationkey = nation1.n_nationkey
    JOIN lineitem ON s_suppkey = l_suppkey
    JOIN orders ON l_orderkey = o_orderkey
    JOIN customer ON o_custkey = c_custkey
    JOIN nation nation2 ON c_nationkey = nation2.n_nationkey
WHERE l_shipmode = 'TRUCK'
    AND (l_shipdate >= date '1995-1-1')
    AND (l_shipdate <= date '1995-12-31')
    AND NOT ((nation2.n_name = nation1.n_name))
GROUP BY customer_nation,
         supplier_nation
```

Figure 2: Demonstration: Generated Query

`BY` clause, as the revenue is an aggregate over these two values. The `revenue` column is calculated using the correct arithmetic expression based on values from the `lineitem` table.

The `FROM` clause includes many joins; although perhaps not written the way an experience data scientist would have written it, it provides the correct outcome. The clause starts with the `supplier` table, which is joined with both the `nation` table to get the supplier nation and with the `lineitem` table to get all lineitems for all sales that were shipped from a supplier in the supplier nation. The revenue is calculated based on these lineitems after the filters have been applied. The `lineitem` table is then joined with the `order` table to get the corresponding order, which is joined with the `customer` table, which is then joined with the `nation` table to finally get the customer nation for each lineitem.

The `WHERE` clause contains a filter stating that `l_shipmode` must be equal to `TRUCK` - after a quick inspection of the `lineitem` table, Michael verifies that this was, in fact, the column he was looking for. There are two filters to ensure that the date is in 1995, as well as a final filter to ensure that the customer and supplier nations do not have the same name. Note that in practice, the final filter could have been replaced with an equivalent but more efficient check that the two keys are different.

After quickly verifying the SQL, Michael types `itnl_truck_shipping` into the ”table name” field and clicks the ”Execute Query” to create a new table in the database based on this query. He scrolls to the bottom of the web page to view his new table.

2.2 Overview

Now that we've seen an example of `nalini` in action, we'll describe our novel technique and implementation. At a high level, the system consists of two components: the interactive web interface, which the user interacts with, and the engine, which is implemented as an API.

The web interface is the bridge between the user and the database. Most of the supported database interactions are simple functions that rely only on basic SQL query execution. Once the web interface is initially configured to connect to a database, the user can use the interface to view previews of all database tables. If the table was created using a SQL `SELECT` query, the user can edit the backing SQL and re-execute the query. The user may also delete tables and create new tables by writing new SQL queries from scratch.

The web interface also allows the user to interact with the `nalini` engine. The engine API takes as input a list of one or more *column specifications* and a list of zero or more *filter specifications*. Every *column specification* consists of an optional *column name* which, if specified, must consist only of alphanumeric characters and underscores, and required *column description*, which can be any plaintext string. Every *filter specification* contains only a plaintext string description. Note that in order to use the query generation engine, the user is expected to view and modify information in

the "Table Relationships" section of the table to keep the database schema up-to-date as tables are generated and modified.

The engine API generates a SQL query from the input through a series of stages detailed in Section 2.3. If the engine is unable to synthesize a query, the web interface surfaces the uninterpretable phrases to the user. Otherwise, the web interface displays the complete query, which the user can make edits to (if desired) and execute. The remainder of the chapter explains the algorithms and implementation of `nalini` in further detail.

2.3 Query Synthesis Engine

The query synthesis engine consists of several independent components which work together to generate a complete SQL query. In this section, we explain the high-level steps of the process and then go into detail about each of the constituent functions using our motivating example for guidance. The algorithm for the general synthesis approach can be found in Algorithm 1.

Algorithm 1 General synthesis methodology

procedure SQLSynthesize(C,F,Γ, γ)
  Input: natural language column descriptions C, natural language filter descriptions F, type environment Γ, confidence threshold γ
  Output: the top-ranked synthesized SQL query
  ▷ Sketch Generation
  Column_Sketches := SemanticParse(C)
  Filter_Sketches := SemanticParse(F)
  ▷ Dependency Resolution
  Resolved_Hints, *Uninterpretable_Hints* := ResolveDependencies(*Column_Sketches*, *Filter_Sketches*, Γ, γ)
  ▷ Error Propagation
  if Length(*Uninterpretable_Hints)* > 0 **then**
    return *Uninterpretable_Hints*
  ▷ Table Graph Synthesis
  Table_Graph, *Dependency_Lookup* := SynthesizeJoins(*Column_Sketches*, *Filter_Sketches*, *Resolved_Hints*, Γ)
  ▷ SQL Rendering
  SQL_Query := RenderSQL(*Column_Sketches*, *Filter_Sketches*, *Table_Graph*, *Dependency_Lookup*)
  return *SQL_Query*

The input to the query synthesis engine consists of a set of columns C, each of which consists of a column description and optional column name, a set of filters F, each of which consists of a filter description, a type environment Γ, which contains information about the database and its schema, and a confidence threshold γ. The confidence threshold γ is used as a cut-off for each natural language hint to determine if it can be resolved to a database reference, or if the user needs to provide additional or more clear information.

The first step of the algorithm is to run the natural language descriptions through `nalini`'s semantic parser, which we detail further in Section 2.3.1. The parser, which we built from scratch using the SEMPRE framework (Berant et al., 2013) uses standard semantic parsing techniques to translate English descriptions into *sketches* of SQL columns and filters respectively. The column and filter sketches specify the shapes of the output SQL fragments (e.g. as a tree of operations) rather than specifying a complete SQL fragment. Where the eventual query will contain references to database columns or values, the parsed sketch will contain a hole annotated with the corresponding fragment of the English description. As a result, the semantic parser can operate without any knowledge of the database schema or values. The database-agnostic nature of the semantic parser is extremely valuable, as it means that the semantic parser does not have to be fine-tuned or retrained every time the user wishes to query a new database.

Once all column and filter descriptions have been parsed into column and filter sketches, respectively, our technique employs program synthesis to fill in all of the holes with proper references

to database columns (see Section 2.3.2). Our program synthesis does not make use of types. For each hole, in order to choose the best-fitting dependency completion out of the many possible completions, our approach defines *confidence scores* based on the schema of the database. Given that `nalini` is intended to be a simple, lightweight proof of concept, the confidence scores do not make any use of the actual contents of the database tables.

In the case that no likely dependency match was found for one or more holes, the `nalini` engine will return a response at this point in the process. The response contains the original input as well as annotations indicating which fragments of the descriptions could not be interpreted as references to database columns. Upon receiving a response with this structure, the web interface will display it to the user so that the user can modify their inputs and try to generate their query again.

If all holes can be interpreted as database references, the engine will then move on to the next step, which is to synthesize the tree representing the `FROM` and `JOIN` clauses of the SQL query. We detail this algorithm in Section 2.3.3. At a high level, the tree of table relationships is synthesized by running each dependency through a depth-first search on schema edges and then running a simple algorithm to combine the dependencies from all columns and filters into one tree.

The last step of the synthesis process is to actually render the SQL query from all of the synthesized data structures, which we detail in Section 2.3.4. This is a deterministic process where the syntax of the SQL dialect actually comes into play. The engine then returns the completed query.

There is one notable caveat we have not yet mentioned: the semantic parser framework we used to build `nalini`'s parser relies on an unconfigurable tokenizer that converts all input phrases to lowercase and splits on some tokens, even if the value is enclosed by quotation marks. For example, the phrase `"Customer#0001"` will be tokenized as `["customer", "#", "0001"]` rather than remaining together as a single string. To bypass this limitation, we include a pre-processing step before calling the semantic parser to save the original input queries, and convert exact quotes back to their original forms before displaying them back to the user as uninterpretable hints or in a rendered SQL query.

2.3.1 SEMANTIC PARSING

Inspired by the technique used to build SQLIZER (Yaghmazadeh et al., 2017), `nalini` relies on a custom semantic parser to map natural English phrases to sketches of SQL query fragments. This strategy enables the generation of high-quality intermediate representations of the user's desired output columns and filters without any database-specific schema information or training data. The SQL column and filter sketches generated by our semantic parser, like all outputs of semantic parsers, are *logical forms*, or unambiguous statements in a domain-specific language (DSL) which follow a context-free grammar.

In order to map a sequence of tokens to a logical form, a semantic parser must have a context-free grammar and a designated *root* non-terminal symbol. The parser specifies a list of rules that can be used to derive non-terminals from the input token sequence, and can define any number of intermediate non-terminal symbols to use as part of those rules. The process of tokenization poses its own challenges; for instance, a multi-word phrase such as "January 1st 1995" should be recognized as a single token representing a date. To handle named entity recognition during tokenization, as well as enable more sophisticated rule-generation based on part-of-speech tags and other attributes, most semantic parsers make use of a linguistic processing module. The other major objective of a semantic parser is to distinguish between the many possible logical forms that can be derived from a single natural language utterance. Typically, this is done via statistical methods that assign a score, or probability, to each candidate logical form based on a set of (utterance, logical form) pairs used as training examples. The parser either returns the top value by likelihood or a ranked list of possible candidates.

We implemented `nalini`'s semantic parser using an existing toolkit for building semantic parsers called SEMPRE (Berant et al., 2013), which was also used to build SQLIZER (Yaghmazadeh et al., 2017). We drew on Stanford CoreNLP library (Manning et al., 2014) for named entity recognition. Our implementation contains 113 rules, each of which is quite simple; we did not leverage part-of-speech tagging or any of the other pre-trained CoreNLP models while writing rules.

columnSketch := expr (, source)? (, filterSet)?
filterSketch := filterSet (, source)?
source := ??*h*
filterSet := expr (, expr)*
expr := value — unaryOp, expr — binaryOp, expr, expr
value := number — string — date — ?*h*
columnValue := COLUMNNAME (, TABLENAME)?
date := day — month — year
unaryOp := `NOT` — aggOp
aggOp := `SUM` | `COUNT` | `AVG` | `MIN` | `MAX`
binaryOp := $+ \mid - \mid \times \mid \div \mid = \mid > \mid \geq \mid \leq \mid <$ | `AND` | `OR`

Figure 3: Grammar of logical forms produced by `nalini`'s semantic parser. Here, *h* denotes a hint; ??*h* denotes to a table hint while ?*h* denotes a column hint.

To assign likelihoods to each possible derivation, the SEMPRE framework maps each derivation to a to a feature vector θ of approximately 40 dimensions, where each feature is an indication of how well the derivation applies to the utterance. For example, one feature corresponds to the number of grammar rules used in the derivation. Another set of features is used to indicate how many skipped words with each part of speech were not used in the derivation (skipping a transitive verb like "is" may not be problematic, but failing to include a noun like "lineitem" may lead to a less accurate derivation). There are also features that encode the relative simplicity of the denotation.

For a given utterance, each possible derivation is given a score proportional to their likelihood by taking the dot product of the feature vector θ and a fixed weight vector w. In the SEMPRE framework, the weight vector w is calculated by maximizing an objective function which rewards correct output based on a set of (utterance, derivation) training examples (x_i, y_i). Our implementation uses a small set of only fifteen hand-generated training examples to provide examples of order of operations in arithmetic, preferences for parsing phrases as dates and quotes, and usage of overloaded/ambiguous keywords such as "of" and "over."

In `nalini`'s semantic parser, input sequences are English natural language phrases corresponding to either column or filter descriptions. The logical forms correspond to SQL column and filter sketches, respectively, where the structure of arithmetic, boolean, and aggregate operators is defined and holes are left to represent references to columns in the database.

The grammar defining the logical form representations is given in Figure 3. In the grammar of our DSL, the rule for $columnSketch$ defines logical forms mapped from column descriptions, while the rule for $filterSketch$ defines logical forms mapped from filter descriptions.

Recall that in our running example, the user provides three column descriptions. The first two, `customer nation name` and `supplier nation name` are parsed as `(columnSketch(expr(value(colHint[customer nation name]))))` and `(columnSketch(expr(value(colHint[customer nation name]))))` respectively.

The third column description, `total lineitem extendedprice * (1-discount) for lineitems with shipping method "TRUCK"`, is parsed as a column sketch with

both a $source$ and a $filterSet$ as follows:

```
(columnSketch
 (expr(SUM(
  *(colHint[lineitem extendedprice])(−(1)(colHint[discount])))))
 (source(tableHint[lineitems])
 (filterSet[(=(colHint[shipping method])("TRUCK"))])
)
```

Note that if the phrase `with shipping method "TRUCK"` had been replaced with `if type of shipping is "TRUCK"` or `that have shipment vehicle "TRUCK"`, we would have ended up with the same exact logical form with the exception of the hint text (which would be `type of shipping` or `shipment vehicle`, respectively).

When parsing a filter, we define the root non-terminal to be $filterSketch$, so for example `customer nation name is not supplier nation name` parses to:

```
filterSketch((filterSet[
 (=(colHint[customer nation name])(colHint[supplier nation name]))
]))
```

and `ship date on or after 1/1/1995 and ship date before 1/1/1996` parses to:

```
(filterSketch(filterSet[
 (≥(colHint[ship date])(date(1)(1)(1995))),
 (<(colHint[ship date])(date(1)(1)(1996)))
]))
```

Notice that both a column description and filter description can always parse to a single hint string, the parser will always return a valid derivation, even if it is not particularly meaningful.

2.3.2 HINT RESOLUTION

The next stage of the query generation process is *sketch completion*, or the process of filling in the holes left in all of the column and filter sketches. Given the column and filter sketches and the type environment Γ, which encodes the database schema, we fill each hole with a with the most likely reference to a column in our database table. We define a reference to a column as having three parts: c, the column name, t, the table the column can be found in, and optionally, j, another table from which we should arrive at t via joins. In a valid column reference, c must be the name of a column in the schema, t must be the name of a table in the schema, j is either the name of a table in the schema or `None`. Additionally, c must be a column of table t, and j cannot be equal to t. Note that our approach is based on the assumption that for any two tables x and y in our database, there exists at least one way to join x to y using *equi-joins*. We define an *equi-join* to be a join where every join condition must consist of one or more equality checks. Because of our general assumption, we do not have to impose an additional constraint that tables t and j must be connected via *equi-joins*.

For example, the column reference `(n_name, nation, customer)` refers to the `n_name` column of the `nation` table, as do the column references `(n_name, nation, supplier)` and `(n_name, nation, None)`—the difference is that in the first two column references, the `nation` table is interpreted as a table joined to the `customer` and `supplier` tables respectively, and in the third column reference, there is no such constraint. In Section 2.3.3, we detail how these column references are used to put together a complete representation of a SQL query. In this section, we explain our algorithm and heuristics for finding the most probable column reference for each hole in the column and filter sketches.

Overall Algorithm: At a high level, for each sketch, we combine knowledge of the database schema, namely the names of the tables and their columns, with the natural language hints provided

in the input, to generate a confidence score for each valid column reference. Algorithm 3 details this procedure for filling holes.

Algorithm 2 Hint resolution

```
1:  procedure RESOLVEDEPENDENCIES(cs, fs, Γ, γ)
2:      Input: column sketches cs, filter sketches fs, type environment Γ, confidence threshold γ
3:      Output: Resolved_Hints, Uninterpretable_Hints
4:      resolved_hints = {}
5:      uninterpretable_hints = {}
6:      function UPDATEHINTS(hole_ptr, top_col)
7:          if top_col.SCORE ≥ γ then
8:              resolved_hints[hole_ptr] = top_col
9:          else
10:             uninterpretable_hints[hole_ptr] = top_col
11:     for column_sketch in cs do
12:         col_src = cs.SOURCE if cs.SOURCE else None
13:         for (hole_ptr, hint_str) in GETHOLES(column_sketch) do
14:             UPDATEHINTS(hole_ptr, TOPCOL(hint_str, col_src, Γ))
15:         for filter_sketch in cs.FILTER do
16:             for (hole_ptr, hint_str) in GETHOLES(filter_sketch) do
17:                 UPDATEHINTS(hole_ptr, TOPCOL(hint_str, col_src, Γ))
18:     for filter_sketch in fs do
19:         for (hole_ptr, hint_str) in GETHOLES(filter_sketch) do
20:             fil_src = fs.SOURCE if fs.SOURCE else None
21:             UPDATEHINTS(hole_ptr, TOPCOL(hint_str, fil_src, Γ))
22:     return resolved_hints, uninterpretable_hints
```

Recall that our input is separated into a list of column sketches and a list of filter sketches. We iterate through each hole independently, whether it is within a column expression, column-associated filter, or standalone filter, and use both the column hint and the optional `source` hint to find the best column by calling the TOPCOL function. For each hole, the TOPCOL function returns the most likely column as well as a score $p \in [0, 1]$; if $p < \gamma$, the column reference is not considered sufficiently likely, and the hint is marked as uninterpretable. After attempting to resolve the holes in all sketches generated by the user's natural language inputs, this phase of query synthesis either returns the list of uninterpretable hints, which will halt execution of the synthesis engine, or will pass the resolved hints to the next phase.

Top Column: The heuristic used to determine the most likely column reference and its score takes three inputs: the column hint (required) col_hint, the source hint (optional) source_hint, and the type environment Γ. Additionally, it makes use of two manually-tuned constants α and β which are used to weight the relative important of matching the table hint and the join hint respectively.

This algorithm is enumerative in nature. First, the database schema is used to generate a complete list of all possible column references A_n of the form (j, t, c) and (None, t, c); this list must only be generated once per query. Then, using a heuristic, a score is calculated for each possibility, and the column reference with the highest score is returned along with its score.

The heuristic is simple: the natural language hints are used to enumerate all likely interpretations as column references H_i, and then a similarity function GETSIMILARITY, based on word embedding distance, is used to assign similarity scores between each interpretation H_m and each column reference A_n. The score of the column reference A_n is equal to the maximum score GETSIMILARITY(H_m, A_n) over all n.

To enumerate the likely interpretations from a column hint, the hint is separated into tokens and consecutive tokens are interpreted as columns, tables, and join tables while preserving the hint order; if a source hint is provided as input to the function, additional possible interpretations are considered.

The similarity between two strings is calculated as the cosine difference of the vector embeddings of both strings in the OpenAI `ada` engine. The `ada` engine is the simplest of four large language models released by OpenAI under the name GPT-3, with 300M parameters. We are only leveraging

the underlying word embeddings generated in training, rather than leveraging its full capabilities as a language generation engine. We chose `ada` over other similar language models which also handle multi-word phrases, such as `BERT` primarily because its API was free and convenient to use.

2.3.3 TABLE RELATIONSHIP DISCOVERY

Once all of the holes in column and filter sketches have been filled with the best candidate for column resolution, the query generation engine synthesizes a tree representing the sources of those columns, and their relationship to each other. This subroutine takes as input the column sketches cs, filter sketches fs, resolved hints $resolved\_hints$, and the type environment Γ and uses a simple shortest path algorithm repeatedly to build up a best guess of the table graph.

Based on the database schema, the query engine first creates a graph representation of all of the tables in the database. Each table is a node. Edges are created when a column in one table has values that can be *equi-joined* to values in a column of a different table. For instance, in the TPC-H database, there are columns named $p\_partkey$, $ps\_partkey$, and $l\_partkey$ in the `part`, `partsupp`, and `supp` tables respectively, all of which contain values that are used as primary keys for parts, so there are edges generated between each pair from the group of three tables.

Then, the $resolved\_hints$ dictionary is traversed to determine which root node is most common among all of the column and filter dependencies. The root node of a column reference is the join table if there is a join table; otherwise, it is the table field. If there is a tie for the most common root node, one of the most common nodes is selected at random. Since all joins are inner joins, the starting node of the join tree can actually be an instance of any of the tables used in the final join tree–we use the maximum rule simply because it makes intuitive sense to start from the table which is referenced the most by the user. In our running example, we had nine holes, five of which were filled with references to the `lineitem` table, so we instantiate a copy of the `lineitem` table as our root node.

Then, we iterate through each of the resolved hints and generate paths to the root. Going back to our original example, lets say we have matched the hint `customer nation name` to the column reference (customer, nation, n_name). Since this column reference has a join hint, our first step is to run a DFS to determine the shortest path between `customer` and `nation`; this is a direct link between the customer and nation tables joined on $c\_nationkey = n\_nationkey$. Now that we know we want the n_name column to come from a `nation` table joined to a `customer` table, we need to synthesize the most likely (simplest) connection between our root `lineitem` table and the `customer` table. Presumably, this is the customer *of* the lineitem. We call DFS once again and determine that the shortest path joins from the `lineitem` table to the `orders` table and from the `orders` table to the `customer` table. We instantiate all three of the tables needed in this path (`orders`, `customer`, and `nation`), and update the dependency lookup table to contain a pointer to the new `nation` table.

We repeat the same process for the next column reference, (supplier, nation, name), leading to building up a new branch from `lineitem` to `supplier` and then to `nation`. Dependencies based on the root, such as `(None, lineitem, l_shipmode)` are pointed straight to the root. Note that when we have to join paths for the same exact phrases (”customer nation name” and ”supplier nation name”) again as part of the filters, we will create duplicate branches attached to the root. To apply the filters correctly, we introduce a final step of graph consolidation. In this step, redundant branches are merged, and their corresponding dependency lookups are updated. Note that this will merge multiple copies of the same table as long as they have the same parent, but that the two instances of the nation table representing the customer and supplier nations will stay since they come from different parent nodes.

2.3.4 SQL RENDERING

In the final step of the query generation process, the column and filter sketches, the table graph, and the dependency lookup dictionary are used to put together a single data structure representing an entire query. This data structure is then expressed as a query in SQL. While `nalini` is currently built to write PostgreSQL queries, the final step of query rendering can easily be rewritten to support any dialect of SQL.

Algorithm 3 Table Graph Creation

procedure SYNTHESIZEJOINS(cs, fs, $resolved\_hints$, Γ)
    Input: column sketches cs, filter sketches fs, resolved hints $resolved\_hints$, type environment Γ
    Output: $table\_graph$, $dependency\_lookup$
    edges = GETGRAPHFROMSCHEMA(Γ)
    root = DETERMINEROOTNODE($resolved\_hints$)
    table_graph = root
    for (column, table, join) `in` resolved_hints **do**
        branch_path = []
        if join **then**
            branch_path.append(DFS(join, table, edges))
        branch_path.append(DFS(table, root, edges))[1:]
        new_table_ptr = ATTACHBRANCH(table_graph, branch_path, root)
        UPDATEDEPLOOKUP(new_table_ptr, column)
    consolidated_graph, consolidated_lookup = CONSOLIDATE(table_graph, dep_lookup)
    return consolidated_graph, consolidated_lookup

The `SELECT` clause is built by filling in the column sketches with the dependencies in the lookup dictionary. At this stage, the `AS` keyword will be used to add the user-inputted column aliases to the `SELECT` statements.

The `FROM` clause, along with the `JOIN` statements, are generated directly from the table graph.

The filter sketches, with dependencies filled in, comprise the optional `WHERE` and `HAVING` clauses; which clause it belongs in is determined by the presence or absence of an aggregate function.

Finally, the `GROUP BY` clause is synthesized by checking which column expressions contain aggregate functions. If one or more synthesized column expressions contains an aggregate function, the remaining columns are added to the `GROUP BY` clause (unless all columns have an aggregate, in which case the `GROUP BY` clause is not rendered at all).

2.4 SCOPE

The implementation of our novel technique supports standard arithmetic operations, boolean operations, and aggregate functions, enabling `nalini` to generate a relatively wide variety of SQL queries. However, there are a few key assumptions made in order to bound the technical scope of this work.

First of all, we assume all joins are inner joins, and furthermore, all joins are *equi-joins*, as defined in 2.3.3. Also, the only supported operations on date and string types are comparisons.

In general, rather than supporting a wide variety of SQL syntax, we require the user to make use of simpler syntax and functions in order to replicate certain behavior. For example, `nalini` cannot directly generate nested queries, but a user can generate one or more intermediate tables to incrementally build toward their target table. Complex logic such as `CASE` and `EXISTS` statements are not directly supported, but filters that would typically be found in `WHERE` and `HAVING` clauses can be combined with multi-step logic to reach the same outcomes.

The grammar specifying the queries that `nalini` can render is formalized in Figure 5. Note that `nalini` is designed to work in cases where columns in different tables in the same database have identical names. For this reason, the grammar allows for the table name to be included as part of a column value. Although there is no formal constraint listed, database systems typically necessitate that table names are unique within a database.

2.5 IMPLEMENTATION

We implemented `nalini` primarily in Python and used Django, a high-level Python web framework. We configured `nalini` to use a local Postgres database. We wrote our custom semantic

parser as an extension to the SEMPRE semantic parsing framework: grammars and training examples are written in SEMPRE's domain-specific language, classes are written in Java, and scripts are executed via Ruby and Bash.

3 RELATED WORK

The work presented in this thesis is related to a broad range of prior work attributed to the natural language processing, programming languages, and database communities. In this section, we compare the technique implemented in `nalini` with related approaches.

3.1 NALINI VS SQLIZER

Because our work has a lot in common with SQLIZER (Yaghmazadeh et al., 2017), we start by differentiating the two systems. While both `nalini` and SQLIZER parse the natural language input using a database-agnostic semantic parser and then fill in the holes in the query sketch using information about the database, one key difference is the problem space that the two tools address. Whereas SQLIZER makes the assumption that every query description is one sentence long, `nalini` is designed to handle more complex queries by allowing the user to input individual descriptions for each necessary column and filter. SQLIZER is intended to be fully automatic, while `nalini` makes use of an interactive interface.

Both tools use the SEMPRE (Berant et al., 2013) semantic parsing framework to build the semantic parser. Note that since the code for SQLIZER is not available, the semantic parser for `nalini` was built from scratch. The SQLIZER parser is also much more sophisticated, making use of several pre-processing steps and part-of-speech tagging.

The two tools differ significantly in how they fill in holes in query sketches. Because SQLIZER supports a much richer SQL vocabulary, it relies on a more sophisticated synthesis algorithm. Not only does it make use of the actual database contents to assign confidence scores to various possible assignments, its algorithm also includes a fault-localization step. In this step, if a hole cannot be filled with high enough confidence, it is replaced with another fragment of a SQL query that evaluates to the same type. The entire synthesis algorithm is type-driven. In contrast, `nalini`'s algorithm can be much simpler since it is given much more information about the structure of the eventual query, since users explicitly specify their desired output columns and tables.

3.2 INTERACTIVE INTERFACES FOR QUERY GENERATION

The idea of using an interactive interface for query generation is not unique to `nalini`. A prominent SQL query generation system, NALIR (Li & Jagadish, 2014), used an interactive interface to supplement natural-language driven query synthesis. NALIR is also based on the fundamental assumption that the natural language query description provided by the user consists of a single sentence. It uses a dependency parser to translate the user's statement into a *query tree*, and then asks the user to refine the visually-rendered generated parse tree. This study showed that interaction with the user can be quite useful, especially in the context of a refinement loop.

In addition to natural language-based query generation, another field of study is query generation using programming-by-example (PBE). The programming-by-example paradigm asks the user to give the synthesis engine one or more example input/output pairs. In the context of database query generation, this lends itself naturally to use of an interactive interface. Both PATSQL (Takenouchi et al., 2021) and SCYTHE (Wang et al., 2017), which are recent PBE-based SQL-query generators, make use of a refinement loop where the user can tweak their input and the desired query as much as possible before exiting it. The demo for SCYTHE is not available. The user interface for `nalini`, which has a database preview on the left and a query generation panel on the right, is inspired by the web interface used in the PATSQL demo, although the query generation panel looks notably different. Our implementation for `nalini` also borrows from PATSQL the idea that all constants used in the final query must be supplied by the user. We chose not to explore programming-by-example in this work because it can be quite impractical for users to specify a concise, yet fully descriptive example when dealing with arithmetic operations and complex logic.

3.3 SKETCH-BASED SYNTHESIS

One of the most central ideas in our technique is Sketch-based synthesis, which was first explored in the SKETCH system (Solar-Lezama et al., 2005; 2006; 2008). In the SKETCH system, a *sketch*-based approach is defined by its two steps: first, the outline of a program is synthesized with holes in place of constants, and then the holes are instantiated with the appropriate constants. The concept of programming by sketching has evolved to include other types of holes; for our context, we use columns and table names. Our particular use of sketch-based synthesis was inspired by SQLizer (Yaghmazadeh et al., 2017), which generates query sketches from natural language, fills in the holes using type-driven synthesis, and then uses a fault-localization algorithm to refine the query until reaching a certain confidence threshold. Other tools for database query generation also rely a sketching approach, including NaLIR (Li & Jagadish, 2014), which uses natural language along with an interactive interface, and PATSQL (Takenouchi et al., 2021) and SCYTHE (Wang et al., 2017), which are programming-by-example interfaces.

3.4 LARGE LANGUAGE MODELS

There have been many deep-learning (Baik et al., 2019; Guo et al., 2018; Yu et al., 2018a;b; Chen et al., 2021; Guo et al., 2019) and rule-based (Li & Jagadish, 2014; Yaghmazadeh et al., 2017) attempts to develop text to SQL tools. All of the recent state-of-the-art methods face the fundamental problem of understanding word similarity, and take advantage of pre-trained embedding vectors of tokens to quantify it (Baik et al., 2019; Guo et al., 2018; Yu et al., 2018a;b; Chen et al., 2021; Guo et al., 2019; Li & Jagadish, 2014; Yaghmazadeh et al., 2017; Kim et al., 2020). One common set of pre-trained embedding vectors is Word2Vec (Mikolov et al., 2013), which is used by similar systems such as SQLIZER and NALIR. However, Word2Vec embeddings are not context-aware; for example, the word "running" in the phrase "running a mile" has the same embedding as the same word in the phrase "running for president." In recent years, there have been advances in context-aware word embeddings, and large language models such as ELMo (Peters et al., 2018), BERT (Devlin et al., 2019) and OpenAI's GPT-3 (Brown et al., 2020) have been made available to the public. Our work uses the smallest possible version of GPT-3's word embeddings, simply because it is the most recently released language model and because it is easily accessible via an API.

3.5 SEMANTIC PARSING

The concept of semantic parsing has been used in a wide variety of natural-language and programming-language related research in the past. In general, semantic parsing uses a formal, often domain-specific, language to generate logical forms which best represent input sentences. Much of the work done using both a neural semantic parser (Guo et al., 2018) and the SEMPRE framework (Berant et al., 2013) has made use of semantic parsing to query databases and knowledge engines. However, these approaches require that the parser is trained on the specific knowledge engine. We believe that our approach, like the approach taken by SQLIZER (Yaghmazadeh et al., 2017) is much more robust and extensible because the semantic parser only needs to be trained on the English language, rather than on a specific database or knowledge graph.

4 EVALUATION: FIRST-USE STUDY

To evaluate `nalini`, we conducted a first-use study with five users, all of whom are early-career software engineers and/or quantitative analysts with computer science degrees and industry programming experience. All participants had some prior exposure to SQL; the average self-reported SQL knowledge was 5.2 on a 7-point Likert scale ($\sigma = 1.3$). Participants reported having SQL exposure in a variety of different dialects (including PostgreSQL, MySQL, Transact-SQL, PL/SQL, SparkSQL, and PySpark) and environments (including Snowflake, Navicat, Sublime, Sequel Pro, DBeaver, DbVisualizer, Jupyter Notebooks, Microsoft SQL Server, and Palantir Foundry).

4.1 METHODS

Due to the ongoing COVID-19 pandemic, we conducted our studies via video conference with screen-sharing enabled. Each participant was given a private URL to access `nalini`, which had

Query	num columns	num filters	num src tables	arithmetic	string comparison	date filter	aggregation	multi-step
Q6	1	5	1	✓		✓	✓	
Q3	4	3	3	✓	✓	✓	✓	
Q1	8	1	1	✓		✓	✓	
Q5	2	5	6	✓	✓	✓	✓	
Q2	8	1	5	✓	✓		✓	✓

Table 1: High-level attributes of TPC-H queries participants were asked to write using `nalini` during the user study.

already been connected to a Postgres database that meets the specifications for the TPC-H decision support benchmark. For convenience, we limited the length of the generated tables.

We began each study with a 15-minute orientation to get each participant acquainted with the user interface as well as to contextualize the data they were working with. Then, working with the interviewer, each participant spent 15 minutes completing three short warm-up tasks. Through the warm-up tasks, the participant was gradually introduced to multi-column table generation, filters, join logic, and multi-step queries. We then asked participants to use `nalini` to write five queries. We chose queries from the TPC-H decision support benchmark because they are designed to include a wide variety of structures and are known to be difficult to support using existing NLP-to-SQL methods.

Users were asked to write queries meeting the specifications for Q6, Q3, Q1, Q5, and Q2 from the decision support benchmark; the ordering was based on how much text input was necessary to specify the queries to `nalini`. Attributes of the selected queries can be found in Table 1. For each query, we made slight modifications to the written specification to improve clarity and include parameters. For example, instead of the original phrasing of "all lineitems shipped in a given year" where the parameter YEAR is later specified to have the value 1994, we might write "all lineitems shipped in 1994." Given that `nalini` does not support sorting or complex string operations, we removed the sorting requirements for Q1, Q2, Q3, and Q6 and replaced the string operation in Q2 that required the 'like' operation with a similar filter that checked for an exact match. Note that Q4 was omitted because `nalini` does not yet directly support the SQL 'exists' operation, and Q2 already provided a good example of a multi-step query.

As they worked, participants were asked to explain their thought processes out loud. Participants took 45-60 minutes to complete the five queries, and then completed a quick exit survey.

4.2 Quantitative Results

4.2.1 Task Completion

In order to evaluate participants' success on the query writing tasks, we compared the final output table of each query they wrote with the output table of the golden query. We define table A to be equivalent to table B as long as the tables contain the same values; row ordering, column ordering, and column names may vary. We define a query q as equivalent to the gold query g if and only if the table produced by q is equivalent to the table produced by g.

The participants' performance on the query writing tasks is summarized in Table 2. All participants were able to use `nalini` to execute queries equivalent to Q6 and Q1, the two queries which require arithmetic expressions, date filters, and aggregates, and had only one source table. All participants generated a correct query equivalent to Q3, which requires arithmetic expressions, string comparison, date filters, and aggregates, and also joins between multiple tables. Four participants were able

Participant	Q6	Q3	Q1	Q5	Q2
P1	●	●	●	●	●
P2	●	●	●	●	●
P3	●	●	●	●	●
P4	●	●	●	●	●
P5	●	●	●	●	●

- ● Participant generated a table equivalent to the table produced by the gold query
- ● Participant generated a correct query but failed to execute it due to WiFi interruption
- ● Participant executed an incorrect query which differed from a correct query by one join condition
- ● Participant failed to generate the correct table independently but was able to after being assisted through the first stage of a multi-stage query.

Table 2: Summary of user task completion.

to successfully execute that query; one had a WiFi interruption and due to time constraints did not re-create the query and execute it against the database. Q5, which required more complex joins and filtering, proved to be slightly more difficult. Only one participant (P4) used `nalini` to generate a correct query. All other participants used inputs to `nalini` that generated a correct SQL query with the exception of a single missing join condition. Participants P1, P2, and P5 did not notice the error and executed the incorrect query, while P3 manually added the missing condition and was able to produce the correct result. Finally, three participants (P1, P3, and P5) were able to write queries equivalent to Q2. The goal of Q2 is to produce a table with information about each part and the supplier that supplies the part at the minimum cost. The gold query is a nested query which also includes aggregation, arithmetic, and string comparisons. In order to produce the desired result, P3 used two chained queries, one of which found the minimum-cost supplier for each part and one of which used joins to find all of the other necessary information. P1 and P5 both built two tables, one of which contained part information (including the minimum-cost supplier), and one of which contained supplier information, and then combined them to create a third table matching the desired output. Two participants, P2 and P4, were unable to independently generate the desired output table. However, once the two-step approach was explained to them, they were able to use `nalini` to build the first table, update the database schema, and build the final output table using their first table.

4.2.2 Post-Study Survey Results

On 7-point Likert scales, participants positively rated `nalini` overall ($\mu = 5.6$, $\sigma = 0.7$). Participants rated `nalini` positively for ease of use ($\mu = 5.6$, $\sigma = 0.9$) and felt that the columns-and-filters approach was intuitive ($\mu = 5.6$, $\sigma = 0.5$). When asked to rate the "naturalness" of their inputs (as opposed to "code-like" inputs), participants rated `nalini` as more natural ($\mu = 4.6$, $\sigma = 1.1$), and even with its bare-bones implementation and limited capabilities, participants said that they would use `nalini` again as opposed to writing raw SQL ($\mu = 4.2$, $\sigma = 1.8$).

4.3 Qualitative Results

Based on our observations, all participants quickly developed an understanding of what `nalini` is capable of and how to use it to write the desired queries. Participants developed a common flow: first, they read and understood the prompt. Some made additional notes in the shared document that had the prompt. Then, they wrote natural language descriptions in the `nalini` web interface, switching frequently between the text input area and scrolling through the table previews. The SQL query generation process was iterative, with participants tweaking their natural language inputs and generated queries until receiving the desired result and executing the query.

4.3.1 Use of Natural Language

We observed participants successfully using a wide variety of natural language constructs to generate their desired queries.

As an example, to specify the `revenue` column for Q6, one participant typed `sum(l_extendedprice * l_discount)`, which is already SQL syntax, one typed `total of lineitem price times lineitem discount`, which was much more natural, and one typed `sum lineitem extprice * discount`, which was a mix of both. Another participant also incorporated one of the necessary filters and typed `total (extendedprice * discount) if lineitem quantity less than 24`. Similar variation was observed for all queries.

We also observed many attempted natural language constructs that were not understood by our semantic parser; we detail them in 4.3.4.

4.3.2 INTERACTION LOOPS

When using `nalini` to generate a query, we observed three types of interaction loops. First, participants' natural language inputs often did not lead to a generated query on the first try, so they had to tweak their inputs. Then, when a query was generated, participants checked that the query was correct and either edited their inputs or the SQL query itself until getting the desired query. Lastly, though rare, after attempting to execute a SQL query, participants may have had to edit the SQL further or start over with natural language to get their desired output table.

Generating a SQL Query: All five participants naturally encountered phrases that were not recognized and relied on the error messages to tweak their inputs. For example, a participant had to clarify the column name p_mfgr when `part manufacturer` did not match any columns. The error messages were also useful when participants used phrases and syntax patterns that were not recognized by `nalini`'s semantic parser (see 4.3.4).

Refining the Generated Query: Once a SQL query was generated, participants frequently made changes before executing it. The most common issue was that a hint had been interpreted as the incorrect column. For example, both P2 and P3 ran into an issue where the phrase `nation` matched the c_nationkey column from the `customer` table, but the intention was actually to read the `n_name` column from the `nation` table. We observed that P2 made this change by changing their input string (in this case, `nation name` worked). P3, who had a bit more SQL experience, opted to change the SQL query directly. Another scenario that occurred for multiple participants (P1 and P2) was misinterpretation of the phrases `date` and `lineitem date` as corresponding to the `o_orderdate` column from the `orders` table and the `l_receiptdate` column from the `lineitem` table, respectively. Both participants adjusted their natural language inputs by typing the exact intended column name, `l_shipdate`, into the column description field.

There were other types of instances where participants directly modified SQL code, including minor tweaks (swapping comparators, fixing typos) and more careful manipulation (for Q5, which involved joins between six source tables, `nalini` generated an incomplete join clause for P1, P2, P3, and P5—P3 noticed the error and added an additional join clause themself).

Adjustments after Query Execution: Sometimes participants wanted to make changes after executing the query and went back to make changes to the underlying SQL. This proved time-consuming, since once executed, the web interface does not preserve the natural language used to generate the query.

In certain cases, users modified the SQL directly and re-executed the query. Sometimes, users opted to start from the beginning and type in all of the natural language inputs again. Another method of progressing after generating an incorrect table was the generation of a new table based on the incomplete generated table. For example, when P4 was generating Q2, they forgot to filter by region in the second step and created a new table using the natural language interface with the same columns and a filter applied.

Combining Interaction Loops: It is worth mentioning that while these cycles of iteration tended to happen sequentially as participants moved between stages of the query generation process, some users combined these interaction loops during their processes. In particular, P4 adopted a strategy of generating a SQL query every time they added one or two columns or filters, and P1, who was the most fluent in PostgreSQL specifically, tended to execute generated queries before inspecting them, using both the generated output and the query syntax together to verify correctness.

4.3.3 OPTIMIZATIONS AND SHORTCUTS

Once participants discovered phrases and syntax patterns that worked, they tended to reuse them. For example, once users saw that not all columns needed aliasing, they stopped adding unnecessary aliases. Once they found an abbreviation that worked for a particular column (e.g. `extprice` for `l_extendedprice`), they tended to reuse the abbreviation for future queries. Some participants began to directly copy/paste phrases from the prompt text into `nalini`. Those who did (P1, P2, and P4) continued doing so for queries after the first query they copy/pasted for. Similarly, some participants (P2, P3, P5) began to copy/paste column names and continued doing so after the first time.

4.3.4 COMMONLY ATTEMPTED CONSTRUCTS

There were several constructs that multiple participants attempted to use which are not yet supported by `nalini`. Between Q6 and Q3, all participants attempted to use the keyword "between" (e.g. `discount between 0.05 and 0.07`, `ship date between jan 1 1994 and dec 31 1994`) and found that it was not supported, opting instead to specify upper and lower bounds separately. Similarly, three out of five participants tried specifying date ranges using the "in" keyword (e.g. `ship date in 1994`). Two out of five participants tried to use "and" in a distributed manner (e.g. `discount more than 0.05 and less than 0.07`). Two out of five participants tried to use "both" in their natural language inputs (e.g. `customer nation and supplier nation both have name "ASIA"`). While working on Q1, two participants also referred to a previously described column, which represented the gross discounted revenue, when trying to describe a new column which represented gross discounted revenue after tax was applied.

4.3.5 A SHIFT IN FOCUS

Several participants commented on how using `nalini` differs from writing SQL queries by hand. P2 and P5 felt that using `nalini` allowed them to spend most of their time thinking about what they were trying to produce, and why, rather than spending time figuring out how to write SQL to fit their output. P4 made a comment about how *"It's actually so nice to not have to worry about typos and exact spelling."*

P1, P3, and P5 all noted that this required a shift in mindset, and that they kept trying to approach problems as though they were writing SQL queries. P1, P3, and P5 all asked questions at various points during the study about how to ensure that a certain join path was generated and then correct themselves to think about the output in terms of columns instead. Interestingly, P1 did actually find a way to force certain join paths by using multi-step query chaining to build up the join tree one step at a time in a way that they could control.

P1 stated that the query prompts, which were formatted as short multi-sentence paragraphs, were extremely comparable to the emails they receive on a daily basis from business analysts at their company, and that the columns-and-filters approach was much more aligned with the way that queries are specified than the one-sentence prompts required by other NLP-to-SQL tools.

4.3.6 COMPARISONS TO OTHER METHODS

In addition to their ability to complete the presented tasks, participants made several verbal comments that provide valuable insight into `nalini`'s strengths, limitations, and potential future improvements.

During the guided tutorial, all participants were enthusiastic and eager to test out the abilities to `nalini`, using several different text inputs to generate the same desired queries. All participants responded positively the first time they saw SQL syntax generated automatically (*"This is so cool!"*, *"This is awesome!"*).

As they completed some of the queries, participants commented on `nalini`'s utility. P1 said *"this can save so many hours of query writing, even, and perhaps especially, for folks with a solid understanding of databases already"*. In reference to `nalini` producing a complex join clause, P3 said *"this thing does all the heavy lifting, all the brutal stuff"*.

Participants also expressed ways in which nalini could be improved. Four of five participants noted that the runtime was too long, with P1 commenting that they would have changed their general strategy to include more incremental iteration if the synthesis engine ran faster. P2 and P3 expressed interest in UI improvements such as being able to click on a column name to indicate getting information form that column and using color to indicate which natural language phrases were mapped to which columns. P5 felt that a visualization of the joins that were constructed would be helpful.

Usability in the Workplace: As part of the exit survey, participants were asked about how nalini might fit into their existing processes at work.

P1 was particularly enthusiastic about being able to use it at their workplace–as a data scientist, often burdened with requests from other people, they could use nalini to make this process self-service or at least much faster. They also mentioned that they have to frequently switch between SQL dialects due to the multiple tools at their workplace and frequently spent time looking up syntax, which could be addressed by nalini.

P2, who had very little SQL experience relative to the other participants, said *"I literally would have had to look up the syntax of every single part of this query"* while working on Q5, and said that whenever they have had to use SQL at work in the past, it was for very simple projections, filters, aggregations that nalini is well suited to. P5 only said that their existing work processes were easy enough to use for software engineers, perhaps indicating that they would be unlikely to prefer using a tool like nalini.

P3 and P4 made note that joins are often confusing and difficult to write, with P3 stating that *"Nalini covers a lot of the most tedious yet basic SQL functionality (e.g. joins) that can waste developer time"*.

P1, P3, and P4 all emphasized that runtime would have to be improved in order for them to get the most benefit out of nalini, since they all currently write SQL in environments that allow rapid iteration.

P1 and P3 also said better support for nested queries would be necessary for it to be useful at the data analysis level, but that it is already useful for simple joins and cleaning steps that are key components of most ETL (extract, transform, load) tasks.

P3, P4, and P5 all commented on the fact that nalini could identify which table a column should come from even if there were multiple tables that had columns with the same name. P3 mentioned that in enterprise systems, many database interfaces include a "data dictionary" which maps column names to natural language descriptions of the column's contents, and suggested that since column names themselves are often esoteric and difficult to interpret, nalini could be more useful if it was integrated with that information

P2 indicated that most of the people they work with (who are bankers) copy data into excel to do these types of transformations rather than creating tables in their database, which leads to a lot of duplicated work and unnecessary file transfer. They said that something approachable like nalini that makes working in a database environment as easy as working in Excel could reduce a lot of unnecessary work.

5 CONCLUSION

We contribute nalini, an interactive interface for natural language-based SQL query generation. Our goal was to address the need for an easy-to-use, user-friendly way to write SQL code, especially given the lack of general usability of current natural language and programming-by-example tools. We built nalini using simple and straightforward implementations of many of its constituent components, including the semantic parser, the synthesis algorithm, the data-based word-similarity function, and the user interface, leaving a lot of space for further improvement. Our first-use study validates the hypothesis that nalini's approach centered on natural language descriptions of columns and filters is both intuitive for the user and descriptive enough for existing synthesis techniques to generate SQL queries. Iterative interaction with the user enables nalini to be effective without necessarily relying on state-of-the-art technology.

5.1 FUTURE WORK

Having built a minimal proof-of-concept to address these technological gaps, we have revealed several areas of interest which are ripe for future research.

5.1.1 UNDERSTANDING PARTICIPANTS

Fundamentally, a tool intended to make life easier for people can only be as good as our understanding of the people whose needs we aim to address. Our first-use study was conducted on a very niche set of people: young (22-25 year old) American technologists who have computer science degrees from nationally-recognized universities and whose work environments typically have access to cutting-edge technology. While they have enough in common with typical SQL developers to have valuable insights during a first-use study, there remains a larger question of what groups of people have the most to gain by using a tool like `nalini`, and what features are most important for them. For example, a fully-featured user interface may be much more valuable for non-technical users than for engineers who are used to debugging and are comfortable with command-line interactions, and English syntax patterns that are natural for developers in India or Nigeria may be completely unrecognized by a system built for Americans. Once the big picture questions of who `nalini` may be most useful for are answered, we can start to ask questions about technical details. What dialect(s) of SQL should be supported? What functions are most important to include? What language model is best-suited for matching hints to columns? What existing SQL-writing environment(s) should `nalini` be integrated in? We emphasize that `nalini` functions as a proof of concept, and that there are many factors concerning usability and accuracy that need to be addressed through broader study.

5.1.2 SUPPORTING A RICHER SQL VOCABULARY

Our user study was based on only a small subset of TPC-H queries because many of the more complex queries could not yet be supported. A natural area of further development for `nalini` is the scope of the SQL queries that can be created. The existing system is designed such that with minimal changes to the semantic parser and query rendering component, `nalini` should be able to support additional operations such as string operations, date operations, window functions, case statements, and the ”between” keyword.

Other operations, such as incorporating ordering and limiting, may require additional thought to the user interface as well. Perhaps it makes sense to add another free-form text inputs, similar to the filter text inputs, where the user can type phrases like ”sorted by revenue descending” and ”only show top ten” to indicate their intention.

A few potential extensions to the SQL language supported by `nalini` that would require more in-depth user studies and development to address: expanded support for joins, and native support for nested queries. Our study only addressed queries where the necessary joins are inner joins. Before building out a solution, it remains to be studied how a user might specify the need for a left, right, or outer join: does a natural language solution make sense? Is that easiest to indicate using the user interface? Should multiple resultant queries be presented to the user so that they can see the possibilities and choose for themselves?

Nested queries (and ”exists” clauses) pose similar questions. Our study only included one chained query, which showed that it is possible to use the chained-query approach, to build nested queries using the chaining approach, but not necessarily intuitive. Is there a distinction between types of problems where the existence of a nested query needs to be specified by the user and types of problems when it can be inferred and generated with only the provided information? Depending on the answer to that question, `nalini` not be well-suited to the problems in its current state; it could require a change in its fundamental assumptions about query structure.

5.1.3 DEVELOPING A MORE SOPHISTICATED SEMANTIC PARSER

The general approach of always allowing for a programmatic solution and gradually building automation and NLP capabilities is good for further development, as we can theoretically have a good understanding of the types of phrases users try to use with `nalini` and use it to inform future feature development. Even with the minimal SQL grammar supported by `nalini` at present, there

many possible improvements to the semantic parser can be improved to generate queries with the same structure from a much broader subset of natural language inputs.

For example, with minimal changes, the parser can support keywords that apply to non-consecutive words, such as "between", "both", and "and" and support keywords that are split into multiple parts such as "either/or", "same/as", and chained comparisons (e.g. `x < y < z`). Another future area of study is using part-of-speech tagging to allow for more complex constructs which could correspond to joins. For example, the phrase `people who drive a car that was in at least one accident in 2010 in a city with population more than 1 million` contains quite a few nested relationships. If part-of-speech tagging can be employed to get meaningful parses of such complex phrases, it makes way for the synthesis algorithm to be modified to support SQL rendering of those inputs as well.

Another area to explore is breaking the assumption that the semantic parser must be entirely database agnostic. Other natural language to SQL attempts (Yaghmazadeh et al., 2017; Li & Jagadish, 2014) have made use of pre-processing and tagging steps to label phrases that might match exact table names, column names, or database values. For example, without any knowledge of what columns the hints might correspond to it is hard to distinguish whether the phrase `[colHint] + [colHint]` should be parsed as an arithmetic operation or a string operation.

5.1.4 Increasing confidence in the synthesis algorithm

Our current synthesis algorithm is relatively simple; it independently fills all holes, and then independently uses each column dependency to contribute to the overall table graph. Other synthesis engines (Yaghmazadeh et al., 2017; Berant et al., 2013; Guo et al., 2018) have demonstrated improvements in synthesis accuracy by using more holistic synthesis algorithms.

For instance, instead of only using the top parsed value for each input, the synthesizer could explore the top-k parsed values for each input, adding robustness. While resolving hints, the synthesis engine could also take into account context. Using type analysis, it may be obvious that an argument to a function is more likely to be one column than another. Similarly, the way a hint is resolved could be influenced by knowledge of the source tables of other hints in the same column or filter. Intuitively, it may make sense to assign ratings to sets of possible dependency resolutions to choose the best combination, rather than choosing the best match for each hole independently. During our first-use study, we observed examples where users unsuccessfully tried to reference previous columns; perhaps it would be helpful if columns could have dependencies on each other.

5.1.5 Improving the user experience

The interactive interface used in `nalini` is bare-bones compared to most modern-day enterprise software in the data engineering space. Now that we have observed that `nalini`'s columns and filters approach can be intuitive, how can we continue to expand on the user experience to make it frictionless? There are quite a few aspects to consider. How can the system be designed to minimize the query engine's runtime? What existing coding environments should a tool like this be embedded in? What additional visual cues, such as color-coding table columns with the phrases they are linked to, would be useful to the user? Is there a visual way to represent joins and table graphs that we could explore?

References

Tpc-h decision support benchmark. `http://www.tpc.org/tpch/`.

Christopher Baik, H. V. Jagadish, and Yunyao Li. Bridging the semantic gap with sql query logs in natural language interfaces to databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 374–385, 2019. doi: 10.1109/ICDE.2019.00041.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL `https://aclanthology.org/D13-1160`.

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. Language models are few-shot learners, 2020.

Zhi Chen, Lu Chen, Yanbin Zhao, Ruisheng Cao, Zihan Xu, Su Zhu, and Kai Yu. ShadowGNN: Graph projection neural network for text-to-SQL parser. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5567–5577, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.441. URL https://aclanthology.org/2021.naacl-main.441.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and Ming Zhou. Question generation from sql queries improves neural semantic parsing, 2018.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation, 2019.

Mona Khalil. Sql server, postgresql, mysql... what's the difference? where do i start?, January 2022. URL https://www.datacamp.com/community/blog/sql-differences.

Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proc. VLDB Endow.*, 13(10):1737–1750, jun 2020. ISSN 2150-8097. doi: 10.14778/3401960.3401970. URL https://doi.org/10.14778/3401960.3401970.

Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, sep 2014. ISSN 2150-8097. doi: 10.14778/2735461.2735468. URL https://doi.org/10.14778/2735461.2735468.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, Baltimore, Maryland, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/P14-5010. URL https://aclanthology.org/P14-5010.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 281–294, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065045. URL https://doi.org/10.1145/1065010.1065045.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pp. 404–415, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pp. 136–148, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375599. URL https://doi.org/10.1145/1375581.1375599.

Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. Patsql. *Proceedings of the VLDB Endowment*, 14(11):1937–1949, Jul 2021. ISSN 2150-8097. doi: 10.14778/3476249.3476253. URL http://dx.doi.org/10.14778/3476249.3476253.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062365. URL https://doi.org/10.1145/3140587.3062365.

Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning, 2017.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133887. URL https://doi.org/10.1145/3133887.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation, 2018a.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task, 2018b.

A Appendix

A.1 Nalini User Interface

See Figure 4.

A.2 Grammar of SQL Queries

See Figure 5.

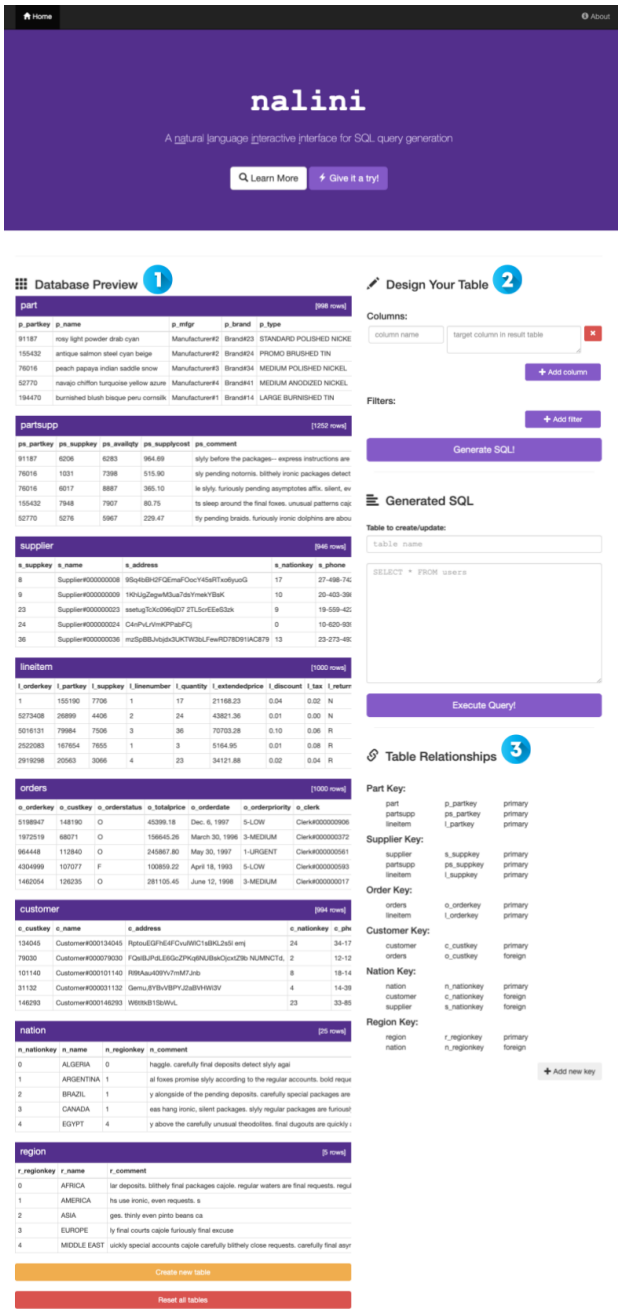

Figure 4: The starting screen of `nalini` with a sample database matching the TPC-H specifications loaded upon startup. 1 A preview of each table in the database is visible in the Database Preview section. 2 The Query Generation panel consists of a place to input column and filter descriptions, as well as a section for the generated SQL to appear. From there, the query can be executed to create or edit any table in the database. 3 The Table Relationships panel allows the user to view and edit keys that are shared amongst tables (and thus can be used for joins).

sqlSelectQuery := sqlColumns, sqlFilters, sqlJoins

sqlJoins := TABLENAME (, (TABLENAME, joinCondition))*
joinCondition := columnValue = columnValue

sqlColumns := sqlColumn (, sqlColumn)*
sqlColumn := expr (, COLUMNALIAS)?

sqlFilters := sqlFilter (, sqlFilter)*
sqlFilter := expr

expr := value — unaryOp, expr — binaryOp, expr, expr

value := number — string — date — columnValue
columnValue := COLUMNNAME (, TABLENAME)?
date := day — month — year

unaryOp := `NOT` — aggOp
aggOp := `SUM` | `COUNT` | `AVG` | `MIN` | `MAX`
binaryOp := $+ \mid - \mid \times \mid \div \mid = \mid > \mid \geq \mid \leq \mid <$ | `AND` | `OR`

Figure 5: Grammar of modified relational algebra produced by `nalini` query engine; each expression in this grammar can be directly rendered using SQL syntax.