Naïve SQLizer - Naïve SQL Query Synthesis from Natural Language Queries

CS516 (Programming Languages and Compilers II) Final Project

Subha Srikanth (ss1826)

May 1, 2018

Abstract

The purpose of this paper is to expand upon the ideas, implementations, and results of two algorithms to synthesize queries in the SQL database language from queries given in natural language by a user. This synthesis is entirely done in Javascript with a front-end website (http://sqlizer.subhasrikanth23.dx.am/) written in HTML and PHP to support and demonstrate the algorithms implemented. The source code can be found at https://github.com/subhasrikanth23/Naive-SQLizer.

The algorithms that have been implemented in this project are a particularly naïve concatenation and a skeleton generation, which leads to a substitution of tokens into the skeleton. After testing, it was found that the skeleton substitution yielded more accurate results for queries while the naïve concatenation failed to generate SQL queries for any inputs that required a proper search through the given BarBeerDrinker schema.

1 Problem Definition/Background

Synthesizing SQL (Structured Query Language) queries from a natural language input has been a well-known problem for many years. SQL is a language built to query databases and retrieve information based on the criteria provided by the query. For example, a query to retrieve all records from a table "people" where its "name" attribute is "Smith" is written in SQL as:

*SELECT * FROM people WHERE name = "Smith"*

For people with little knowledge of SQL and its syntax, constructing queries can be somewhat of a daunting task, particularly when involving more complex queries in relation to the example above or nested and sub-queries.

Therefore, the goal of this project is to create a user-friendly interface or tool that takes in a natural language input string from the user and parses it into tokens for the synthesizer to take and transform into a query in SQL syntax.

2 Past Related Work

The main body of work done on this domain by Yaghmazadeh et al. involves creating a program sketch using natural language techniques before entering an iterative refinement loop to correct the program by alternating between quantitative type inhabitation and automated sketch repair [1].

Other previous existing works using natural language are easier for the end-user, but more difficult for the underlying synthesizer, as natural language is ambiguous. Because of this, most natural language-based synthesizers are not schema-agnostic. However, Yaghmazadeh et al. combine the natural language approach with repair techniques to provide a schema-agnostic and fully automated synthesizer to generate SQL queries.

The first step of their synthesis process takes the input natural language query and transforms it using standard semantic parsing techniques into a query sketch, or a skeleton. Because this only yields the outline of the query, the semantic parsing techniques do not need to be trained on the database schema.

From there, their technique performs a type-directed sketch completion, upon which the holes in the sketch are filled in by likely completions based on the database schema and contents. Each possible completion is assigned a confidence score.

At this point, the sketch enters the refinement phase using repair techniques. The top k most likely query sketches have been generated and, for each sketch in the generated set of sketches, a completion with a certain confidence score has been synthesized. If no such completion can be found, then the sketch is automatically repaired once the technique finds the root cause of the failure.

Finally, after their parse-synthesize-repair process is complete, the SQLizer technique returns the final top k most likely queries based on the ranking of their computed confidence scores to the user.

3 Back End Details

3.1 Schema

The default schema implemented into the Naïve SQLizer program is the BarBeerDrinker schema, with the tables and attributes indicated below.

Table	Attributes
Bars	name, address (abbreviation: addr), license
Beers	name, manufacturer (abbreviation: manf)
Drinkers	name, addr, phone number (abbreviation: phone)
Sells	bar, beer, price
Likes	drinker, beer
Frequents	drinker, bar

Table 1: Default BarBeerDrinker schema of tables and their respective attributes

The tables are stored as an array while the attributes are stored as a dictionary with the keys as the tables and the values as arrays of the attributes of each respective table.

3.2 SQL Syntax Keywords

There are four keywords that appear in every SQL query that requires some form of search (i.e., a query that isn't simply "get all values from a table", which is equivalent to "SELECT * FROM [table name]" in SQL), and so we use those as a "base case" in the synthesis. These keywords are "SELECT","FROM", "WHERE", and "=".

In future implementations, additional keywords such as "AND","GROUP BY", and "ORDER BY" can be added into the synthesis process provided that the generated tokens [Section 4] indicate that these keywords will exist in the final query. Additionally, aggregate functions such as "AVG", "MAX/MIN", "COUNT", etc. can also be included if the natural language input implies that they are necessary in the final SQL query.

4 Natural Language Query Tokenization

The first step of synthesizing a SQL query from a natural language input sentence is to extract meaningful words - or rather, tokens - from the natural language input query. Using a Javascript-based natural language processing library named Compromise [2], each word of the input string is assigned a tag [Table 2]. These tags can range from the part of speech (adjective, verb, noun, etc.) to the tense it is in (past, present, or future).

The tagged words are then grouped into three categories (stored as arrays), as follows:

- Values: All words tagged as quotations
- Fields: All words tagged as nouns
- Keywords: All words tagged as adjectives or verbs

Get me all drinkers with the name "Smith"	
Word	**Tags**
Get	"TitleCase", "Infinitive", "PresentTense", "Verb", "VerbPhase"
me	"Pronoun", "Noun"
all	"Adjective"
drinkers	"Noun", "Plural"
with	"Preposition"
the	"Determiner"
name	"Noun", "FirstName", "Person", "Singular", "ProperNoun"
"Smith"	"LastName", "Person", "Singular", "Noun", "StartQuotation", "Quotation", "EndQuotation", "StraightDoubleQuotes", "ProperNoun"

Table 2: Tags for each word of the sample input query

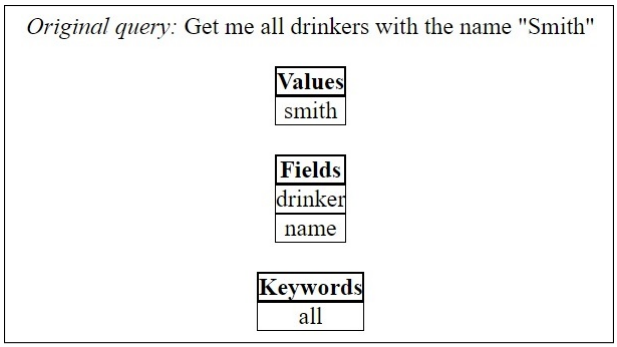

Figure 1: Tokenization of a sample input query, where the words are individually tagged and then categorized based on their tags into values, fields, and keywords

Using this method of categorization, we assume that all values that the user is searching for will be enclosed in quotations. Figure 1 above shows an example where the natural language query is processed and the words are separated into values, fields, and keywords based on the previously-specified tags.

5 SQL Query Synthesis

5.1 Naïve Token Concatenation

This method relies on naïvely concatenating SQL syntax keywords along with the tokens created by the tokenization algorithm described in Section 4 to the string that is to be returned as the query.

The query is first initialized as an empty string before the keyword "SELECT" is automatically concatenated to it, as every SQL query begins with "SELECT." If the keyword "all" is present in the tokens, then "*" will be concatenated to the end of the query. Otherwise, the first field in the "fields" category of tokens will be substituted in its place. "FROM" is then concatenated before a table is concatenated depending on if it exists in the "fields" tokens. If multiple tables are found in the tokens, then the first unused table is used.

From there, "WHERE" is concatenated, followed by a field, "=", and then a value. If more than one field and value are remaining, an "AND" clause is added and repeated until all tokens have been used. The first two letters of each table name are used as aliases to refer to each table, and therefore is also added to the front of each field name in the "WHERE" and "AND" clauses to specify the tables that the fields are referencing. Finally, a semicolon is concatenated to the end of the query to complete it.

As each token is used, it is added to an array of used values, fields, or keywords respectively to keep track of those that have been previously concatenated to the query to avoid repetitive use. Upon the next concatenation, the token is checked to confirm if it has been used previously or not. In the former case, it is skipped in favor of the next token in the list.

Because this method consists of mainly hard-coded concatenations/substitutions, it is not a particularly viable method of SQL query synthesis for more complex queries.

5.2 Skeleton Generation and Token Substitution

To generate a SQL query skeleton, a recursive algorithm was implemented such that the depth of recursion depends on the user feedback given after the first query generation [Section 6]. By default, the depth is initialized to 0, thereby invoking the base case as the skeleton.

The base case is given as "SELECT ? FROM ? WHERE ? = ?", where "?"s indicate unknowns that will be later replaced by the tokens generated by the tokenization algorithm described in Section 4. As the depth is increased or decreased, so is the complexity of the query. For example, if the depth is 1, then the skeleton becomes "SELECT ? FROM ?, (SELECT ? FROM ? WHERE ? = ?) WHERE ? = ?", thereby creating nested queries or sub-queries as needed. Similarly, decreasing the depth will remove a nested query or sub-query from the skeleton.

The skeleton is then passed into the substitution algorithm, at which point the skeleton is split into an array of strings based on the space delimiter. With this array, the indices of all "?"s are found and stored in a separate array to iterate through each unknown location and substitute a certain token in depending on the value stored in the previous index of the skeleton array.

For example, if "SELECT" was the syntax keyword before the current "?" being evaluated, then "*" would need to be substituted into the "?". Similarly, if "FROM" was the previous syntax keyword, then a table name would need to be substituted. If "WHERE" was the previous syntax keyword, then an attribute of the table used in the "?" following "FROM" would need to be substituted in. Finally, if "=" was the previous syntax keyword, then a value would need to be substituted in.

To decide which tables need to be used, the algorithm searches the fields for any field resembling any of the tables in the original array storing all table names in the schema. To do this, each field must be made plural, as all table names in the schema are plural, using a Javascript package called Pluralize [3]. If any of the pluralized fields match a table name, then the table name is added to an array of "tables to use." Similarly, any keywords (once they have been pluralized) that match a table name are added to the "tables to use" array as well.

Unlike the naïve algorithm, the skeleton substitution algorithm forgoes aliases, as they are not intuitive to the end user and provide no help in the actual SQL query. For complex queries that access the same table multiple times, aliases may need to play a factor. However, for the purpose of this project, they have been excluded for the sake of reduced complexity in the final resulting query.

The algorithm can then repeat the process for any attributes found in the fields or keywords that match the attributes found in the original dictionary storing the keys as table names and the values as arrays of attributes for each table. These found attributes are added to a dictionary of "attributes to use." When substituting tables and attributes into the unknowns, only the "tables to use" and "attributes to use" will be involved.

From here, the algorithm enters a loop until all of the indices of "?"s have been reached and a token has been substituted in each "?"'s place. Each used token is removed from its respective array once it has been used to avoid repetition.

Finally, once all possible tokens have been exhausted, the query is returned to the user to show its current state before it is checked for any remaining "?"s or "undefined"s, in which case, the algorithm additionally outputs that the final query "cannot be synthesized." Otherwise, the final query is returned to the user for any further repairs [Section 6].

6 User Feedback-Based Repair

The user is given the option, post-synthesis, to indicate if the query is not complex enough or too complex. Depending on the option chosen, the following response will occur:

- Not complex enough: The depth will be incremented by 1
- Too complex: The depth will be decremented by 1

The "too complex" option does not appear if the depth is already 0, as the depth can never be less than 0. Once the appropriate change is made to the depth, then the synthesis is repeated.

Because the depth changes as a result of the user's choice, the recursion in the skeleton generation will also change, thereby either adding or removing a nested query to the skeleton.

7 Results

7.1 Overview

Overall, the naïve implementation of the SQL query generation falls short of the mark, mainly because many values and choices were hard-coded, and there is no real intuition behind the algorithm outside of deciding which value to concatenate onto the query string.

While simple queries are successfully generated, more complex queries are incorrectly generated or incomplete and it is clear that this algorithm is not feasible to implement for multiple kinds of queries.

In contrast, the implementation of substitution into a skeleton is naturally more intuitive, as it searches through the tokens and determines which token is appropriate to substitute into which location based on the keyword preceding each unknown.

While it stands to reason that the currently-implemented skeleton substitution algorithm may not generate overly-complex queries correctly, in addition to being unable to cope with queries that require aggregate functions and methods such as grouping and ordering, these are features that can be added in future improvements.

7.2 Sample Results

The results below [Tables 3, 4] are some sample inputs and their respective SQL queries generated by each algorithm (naïve and skeleton substitution) after the user has modified the depth of the query to their satisfaction. For example, in the first two queries' cases, no depth changes were needed, while in the third, the query depth needed to be increased by one.

The final columns in both tables indicate if the output query is correct or incorrect, and if incorrect, why the algorithm failed. Success or failure in this implementation is determined by visual inspection of the query, but in future implementations, can be determined by testing the query on an input set of data corresponding to the BarBeerDrinker schema.

Input	**Final Naïve Output**	**Success/Failure**
Get me all drinkers with the name "Smith"	SELECT * FROM drinkers dr, WHERE dr.name = "smith";	Success
Get me all bars that sell "Corona"	SELECT * FROM bars ba, WHERE	Failure (breaks after WHERE clause)
Get me all drinkers with the name "Smith" who enjoy "Bud Light"	SELECT * FROM drinkers dr, WHERE dr.name = "smith";	Failure (not complex enough)

Table 3: Results of sample queries given to naïve algorithm

Input	**Final Skeleton Substitution Output**	**Success/Failure**
Get me all drinkers with the name "Smith"	SELECT * FROM drinkers WHERE name = "smith";	Success
Get me all bars that sell "Corona"	SELECT * FROM sells WHERE beer = "corona";	Success
Get me all drinkers with the name "Smith" who enjoy "Bud Light"	SELECT * FROM (SELECT * FROM likes WHERE beer = "bud light") WHERE drinker = "smith";	Success

Table 4: Results of sample queries given to skeleton substitution algorithm

8 Future Work

Ideally, the following improvements can be implemented upon the current Naïve SQLizer program:

- Add a repair option for the user to indicate that the depth of the query is correct, but that the fields/values have been placed in the wrong order
  - In this case, the algorithm should rearrange the fields and values without repeating what it had done in the past (i.e., memorizing the wrong solutions so that it does not return any solutions that had been previously established as incorrect)
- Make the project schema-agnostic by having the user input the tables and attributes either directly through the website or as a CSV file before these tables and attributes are passed in as a dictionary to replace the default BarBeerDrinker schema, or keep the default schema if the user chooses to
- Use a neural network with a large training dataset of input-output examples (input being a natural language sentence and output being a possible skeleton) to generate the skeleton automatically based on previous input-output examples
  - On this note, potentially improve skeleton generation efficiency by implementing an automated program sketch generation and completion synthesis similar to the past work done by Yaghmazadeh et al.
- Improve upon the substitution into the "?" after the "SELECT" syntax keyword so that it isn't automatically "*" for all queries, but rather reliant upon the tokens specified in the input
- Add "AND", "GROUP BY", and "ORDER BY" clauses, in addition to aggregate functions

9 Final Takeaway Lessons

The main takeaway from this project was not only what was accomplished, but also how the algorithm can be improved. Currently, the user determines how the query is formulated, which is not ideal since the user does not always know what depth the query should be or how the order of the substituted fields should be. By studying previous works that automate the process described in this paper, one idea that stood out was the concept of automating skeleton generation and post-synthesis repair without the need for user feedback.

In addition, working with a natural language processing library was another experience that led to a better understanding of how a natural language sentence is parsed by a program compared to how we as humans naturally understand the same sentence. Manipulating the program to sort words into the correct category of tokens based on the natural language tags they were assigned was one of the larger challenges that was interesting to solve, particularly when this had to also be somewhat intuitive to the user to allow for user-friendliness.

References

[1] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, Dec 2017. doi: 10.1145/3133887. URL https://www.cs.utexas.edu/~ypwang/pubs/oopsla17.pdf.

[2] Spencer Kelly. Compromise. URL https://github.com/spencermountain/compromise/wiki.

[3] Blake Embry. Pluralize. URL https://github.com/blakeembrey/pluralize.