October 5, 2024

CS 842 Programming project proposal

Alvin Zhang

Motivation

This project is about building a compiler for a new programming language, “unordered SQL”. This builds upon SQL by allowing users to write the Select, From and Where clauses in whatever order they wish, including multiple times (eg. writing “select a from table select b” is equivalent to “select a, b from table”).

There are 2 primary benefits to such an extension:

- **Better modularity**: It is possible to copy a Where clause from one SQL statement to another, rather than having to worry about changing the “where” keyword into “and” (and vice versa). As a more complex example, it’s possible to create a complex query (such as a table join) by writing the queries on each table individually, then combining them with a Where clause.
- **Better autocomplete**: With the current setup, it is not possible to have proper autocomplete when writing the Select clause because the From clause has not been written yet. Alternatively, one can write the From clause first, then backtrack the cursor to finish the Select clause. However, both feel unnatural and unnecessary because there shouldn’t be any semantic difference in the ordering of the clauses.

It is worth noting that, while this makes it easier for someone to write a query, it makes it harder for someone to read it. Perhaps it is for this reason that the original creators opted to put the Select clause first: it follows a declarative programming approach and follows more naturally from English. However, this extension of the language has a direct and intuitive translation, allowing it to be used interchangeably with regular SQL. The ultimate goal is for all regular SQL to also be unordered SQL, and for all unordered SQL to have a simple transformation to regular SQL.

Approach

To ensure a successful project, it is broken up into 3 stages: Bare minimum, Prototype and Extensions. After the completion of this course, the lessons learned from building the prototype can be used to build a readily available product, such as a fork of DBeaver.

The Bare Minimum

This allows users to reorder the Select, From and Where clauses in whatever order they wish, including multiple times. No support is required for complex expressions or other clauses, such as Limit and Order by. For this project, an interpreted language with dynamic type checking such as Python is ideal for quick development, though it is ultimately decided by what AST parser is being used.

There are 2 milestones for this stage:

1. **Parse the language**: Find an open-source AST parser for SQL and modify the grammar to support an arbitrary number of the 3 clauses. If a suitable open-source AST parser cannot be found, a LALR(1) parser generator such as one used in CS 444 Compiler Construction can be used to create a bare-bones version.
2. **Output the SQL**: Print the corresponding SQL by processing the AST.

The Prototype

This adds the second primary benefit by providing rudimentary autocomplete abilities. To avoid the complications of integration, each milestone will be completed without worrying about integrating its functionality into the existing prototype. This may result in the prototype feeling “clunky” to use.

There are 5 milestones for this stage:

1. **Break the AST into clauses**: To simplify further development, and to aid in reducing the cost of regenerating the AST with every user update, the AST should be modified to read each clause separately. This may require banning the use of keywords, such as the use of “from” (without double quotes) as a column name.
2. **Generate semantic AST**: Augment the AST by analysing it and providing details such as which table a column is associated with. Ideally would also include a link to the location in the file (ie. row and column numbers).
3. **Query table metadata from database**: To retrieve table metadata from the database, a SQL connection must be set up.
4. **Perform autocomplete**: Create a function that, given a partially completed column name and a set of visible tables, generates a list of likely column names. This can also be done for keywords and table names, then combined to produce a holistic program.
5. **Integrate into a text editor**: Figure out a way to display and write the autocomplete into a text editor, such as VS Code. That is written in JS, but there are methods to call processes running in different languages, along with other alternative text editors.

The Extensions

These are features that are not required to demonstrate the main functionality, but nonetheless useful to have. Technically, the last milestone of the prototype stage is an extension, but from a practical standpoint it would be nice to properly integrate the prototype in said stage.

Here are a number of potential categories of milestones:

1. **Adding aliases**: Adding the ability to write “table1.column2” or “t1.column2”.
2. **Additional where expressions**: Additionally, it is possible to add expressions in the Select clause.
3. **Additional clauses**: Such as Limit, Order by, Having clauses
4. **Additional features not included in regular SQL**: Such as a Using clause, which indicates which database schema is being used.

5. **Additional configurations when outputting regular SQL**: For example, if there is a Using clause, do all table names need to be prepended with the database name?