**Interactive SQL Query Generation**

CSE 544 Final Report

Adrian Sham (adrsham) and Lindsey Nguyen (nhlien93)

*University of Washington, Seattle*

**Abstract**

In this paper, we examine an application that helps a user generate complex SQL queries using feedback provided by the user. In order to provide this feedback, the user directly modifies the results of a SQL query and lets the application decide on a possible new query based on the modifications. We discuss some of the implementation details of the application and also some of the challenges faced during development. We will also discuss some the current work in the area of assisting users with writing SQL queries and some of the different approaches to solving this problem.

**1 Introduction**

The proliferation of relational database systems in almost all aspects of computing has increased the importance of learning SQL for managing data. Various work such as web programming and scientific research of data require the use of SQL for accessing databases. This creates a need for people who are not computer scientists or database administrators to interact with databases, such as researchers or web application developers. As such, for users who are less familiar with SQL, writing complex SQL queries may be difficult.

In general, most users write SQL queries by starting with the simplest form of the query, such as SELECT \* FROM table. Then, after taking a look at the returned table, users will start thinking about refining the results to suit their needs, such as by adding filters, joins and projections to the query.

After adding the refinements such as filters, joins and projections, the user will usually rerun the query to see if there are any errors and whether they are closer to what they have in mind as the desired result. This process is usually repeated multiple times until the user has found the correct query to do what he/she wants.

For users who are more familiar with Microsoft Office software such as Excel, they may decide to just import the entire table and work with the data using Pivot tables. The first of several issues with this approach is that the user usually does not need the entire dataset, and if the dataset is large, downloading the entire dataset would be slow over the network. Also, if the dataset is large, trying to process it locally may slow down the user’s machine. Finally, it is usually more efficient to do data processing like Aggregation in the database itself and only return the processed result.

Recognizing this problem, we decided we would try to work on an application that would help the user write more complex queries in an intuitive way. Instead of having the user rerun queries and see if the result is closer to what they had in mind, why not let the user directly modify the result to tell the application what the user wants? This way, instead of doing several iterations of modify and run, the application would propose the SQL query and hopefully save the user some time.

We hope that by providing this application, users will feel comfortable with writing complex queries and shift away from downloading entire datasets into software like Excel. This would move more data processing work into the database and likely help save the user some time.

**2 Approach**

As stated previously, the main goal of this project is to find a way to make writing complex SQL queries easier for regular users. In order to achieve this goal, we propose creating an application that helps the user update complex SQL queries by attempting to use the modified results of a SQL query to generate a new query. We will start out with supporting very simple SQL statements and expand on the application depending on how long and difficult it is.

The overall experience we are trying to provide the user can be summarized in the following sequence of actions:

1. Provide the application with a simple query with no sub queries, this initial query is provided by the user and can be simple. An example would be SELECT \* FROM students

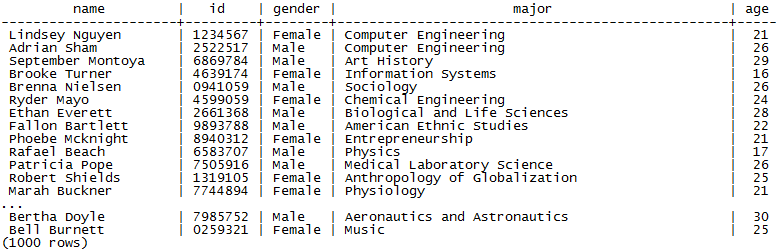
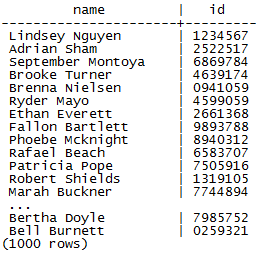
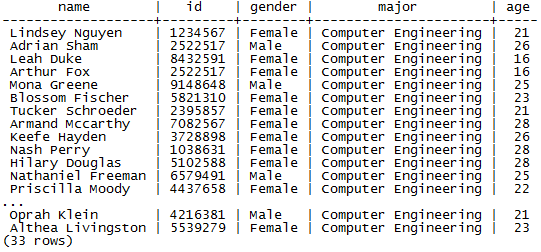
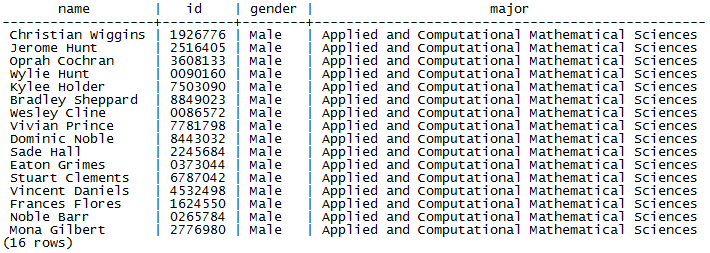


Table 1: Table of results when doing a SELECT \* on the Students table.

(a) Changing select statement (b) Changing where clauses



(c) Changing both select statement and where clauses

Table 2: Manually modified query used as input to our program so that an SQL statement can be generated

1. The application then runs the query on an existing Postgres database
2. The application outputs the result of the query
3. The user modifies the result such that it is what the actual user wants and inputs this new dataset back into the application
4. Based on the modified result, the application will try to generate the corresponding query that will output these exact results

For our project we will be using a generated dataset that contains one table called Students. This table will have fields including name (text), id (int), gender (text), major (text), and age (int), and will have a total of 1000 tuples.

Using the query “SELECT \* FROM Students” which results are displayed in Table 1, the user will be able to input a modified data table such as the one in Table 2a to get the resulting query:

*SELECT name, id FROM Students;*

This is an example of changing the projection operator of a query plan or the select statement inside a SQL query.

The user could also input another modified data table as shown in Table 2b which would generate the resulting query:

*SELECT \* FROM Students WHERE major=’Computer Engineering’;*

This is an example of changing the select operator of a query plan or the where clause inside a SQL query.

Another good example of what our program can do is change both the select statement and where clause both in one as shown in Table 2c. The result would be the query:

*SELECT name, id, gender, major FROM students WHERE gender = ‘Male’ AND*

*major = ‘Applied and Computational Mathematical Sciences’;*

In the simple examples above we show that a SQL statement can be generated using the changes in the resulting dataset. Our goal is to use the original SQL statement, convert them into a relation algebra query plan, and then use the different operators and its order to create the final SQL statement.

For our implementation, we started out with the Project Operator which was relatively simple and mainly used to test our application setup and connection with Postgres. Next we only looked at the WHERE clause of an SQL statement, concentrating on the Select Operator. For this research paper, those two operators were the only ones accomplished, however if future work is done on this project, other operators such as joins and group bys will be implemented as well.

**3 Implementation**

We have implemented our query generator application in about 2,000 lines of Java. Java was selected as the implementation language because of portability and functionality. The system runs by connecting to a local Postgres database where user information of that database is given to the application as arguments. Java also provided a well-documented API for accessing and processing data stored in a database.

When the program starts up, a command prompt comes up and takes in console input to decide what commands to run. There are two commands that our program supports, select and update. Select is basically our version of PSQL that only allows select queries through. Update is where the bulk of our time went into perfecting. This is where a SQL statement and table of the modified result gets inputted so that the application can generate a SQL statement. When running these commands, the user has the option to either just output the results (either the query results or the modified SQL statement) to the console or to a specified file location.

Using Postgres as our backend database, we created a Table object with which to hold the database query results in main memory so that it can be used and manipulated to find the file input’s query. Code was written so that we could generate this table object both from a text file of the correct format and from a ResultSet which is what queries are returned in from the basic SQL package in java.

Currently, as mentioned before, our program only supports basic SQL statements that only have the basic SELECT, FROM, and WHERE clauses where the WHERE clauses only do selection and not joins.

**3.1 Select Command**

Our program is able to do the basic command of returning query results back either through the console or having it saved to a file. To implement this command, we used the basic SQL package in Java and passed the commands through to Postgres. When results get returned from Postgres, the results get parsed into a String of a correct format so that if the user uses the file as an input to the application, the application will be able to successfully parse it.

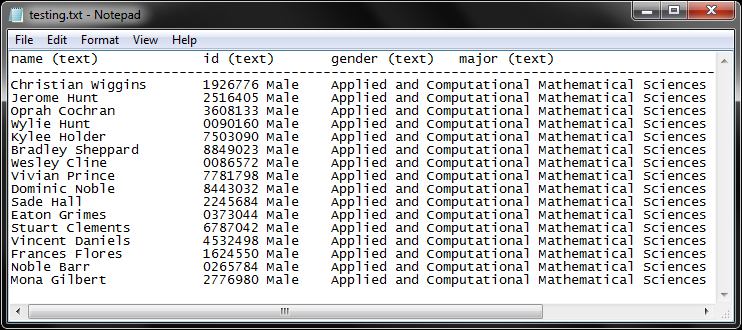


Figure 1: Image of file with the correct format to input into our program

Very little work was put into this command. The only real benefit to having this was so that if the user wanted to have a file to be able to manipulate before inputting into our update command, they would be able to see how the file was formatted before inputting.

**3.2 Update Command**

The Update Command, which has been mentioned multiple times in this paper, is where our query generator runs. This command takes in two, possibly three inputs. The first input is the original SQL statement which we will call the original query from now on. Next input is the filepath to the file that holds the data that we hope to get a SQL statement for (Figure 1). The last input is another filepath for if we want the SQL statement returned to be saved to a file. This last parameter is optional.

When the application parses the SQL query, it is stored as a ZQuery [1] object. This gives us a more sophisticated representation of the information contained within the SQL query for use in later stages of the application. As the application is building the modified SQL query, it is also stored as a ZQuery object.

After taking in the necessary inputs, the file and the query results are both read into a String and then parsed into a Table object. A Table object contains a table descriptor, and a list of tuples. The table descriptor has a list of the field descriptors which include the type and name. The tuples contain the actual fields which can only be either a string or integer. This was done to simplify our implementation.

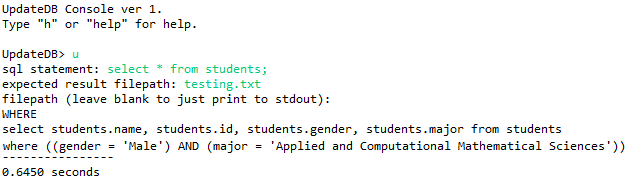


Figure 2: Image of program being run to create result from Figure 1

Table objects are then used as a comparison to find out the changes from one to the other, starting with projection then selection.

**3.2.1 Projection Operator**

To get the modified SELECT list in the query, we tried two different ways. The first was just a brute force method that tried all possible combinations and then a smarter way that based it on the columns of the original queries SELECT list. Using the SELECT list of the original query, we first check to see if the two tables have the same columns. To do this we checked to see if either table contains the other table. (The contains method for a Table object was written to check if the list of tuples were within the other list of tuples. This means containing checks row wise not column wise). If they are the same as the original queries SELECT list, that SELECT list would be used for the new query we are trying to generate.

If the columns do not match then there are two options. Either the original has more columns in which we would need to figure out which to get rid of or the original has fewer columns so we would have to figure out which columns to add. To do either of these, we must first label the current columns of the input file table with what their column names would be by checking to see if a column in the original table is contained in or contains a column in the modified input file. This is due to the fact that the names of the columns in the input file may be an alias. If the original column doesn’t have a certain column in the modified one then our program will check the actual database for any missing columns.

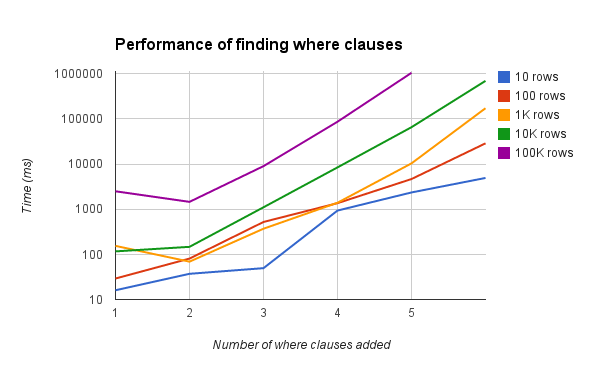


Figure 5: Performance of finding where clauses

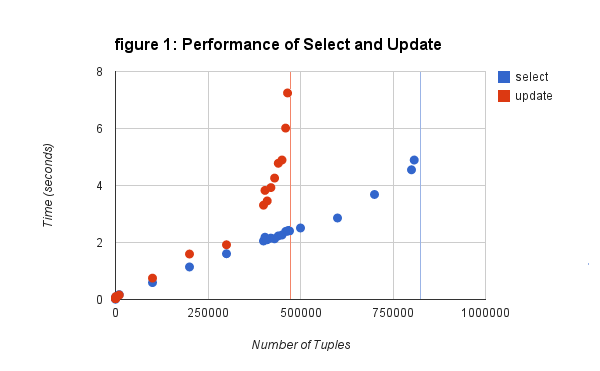


Figure 3: Performance of Select and Update with memory bounds

**3.2.2 Selection Operator**

To edit the WHERE clauses we also tried two different ways but found that for WHERE clauses it is easiest to start with no WHERE clauses and based on the input file’s table, generate a list of all possible WHERE clauses and choose all possible combinations starting with no WHERE clauses and slowly increasing. Although this process does not have a great big O complexity, it guarantees that any possible SQL statement that results in the inputted file will be found. We used the typical n choose k formula using a BitSet to save memory to implement this section.

To find all of the possible WHERE clauses, we create a map that goes from name of each column to a HashSet containing all distinct fields of that column. Then we look through the map and create a WHERE clause for each column to field pair, combining them with an equals operator. For the case where the field is an integer, we also add <= and >= operators. Once we have our total possible list of where clauses, it makes it simple to find which WHERE clauses we actually want.

**4 Evaluation**

In this section of the paper, we will try to evaluate the performance of our application. For our evaluation, we will talk about 3 main tasks that we wanted to test for our application. The first was the size of the possible database query. Because all of our query results were held in main memory, large databases currently will cause out of memory issues, therefore we wanted to test and see what the upper bound of that memory was. Next we evaluated the time it took to add and remove projections compared to the original query. Lastly we found the time it took for where clauses in an update to run, first starting with no WHERE clauses and then going up to 5 different WHERE clauses.

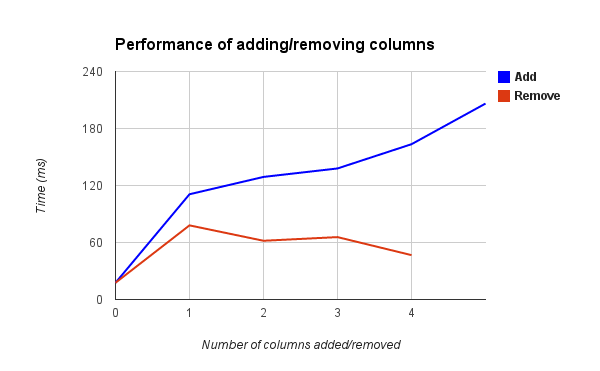


Figure 4: Performance of adding/removing columns

**4.1 Memory Upper Bound**

To test when our program will run out of memory, we tried running on a simple self generated dataset that contained one table called Students with two columns, name and id. As we increased the size of the dataset (Exponentially until we found the upper bound), we also measured times it took to run the SQL command ‘SELECT \* FROM Students’ and the time it took to update the query from a single tuple with the correct projection to the full list again. The results are displayed in Figure 3. As seen, the select command can go to 807350 tuples with two columns while the update command can only go to 465000 tuples. Figure 3 shows the performance of these runs.

**4.2 Update for Projection Changes**

For our next evaluation, we tested adding and removing projections the in final update by creating a data table with 10 columns and 1000 tuples. For our original query, we provide the SQL statement to have five of the columns projected and then we test by having 1-10 columns for the input file. The graph is shown in Figure 4.

**4.3 Update for Selection (Where clause) Changes**

For our last performance evaluation, we tested adding WHERE clauses to a table called mock\_data that had 2 columns named name and id. The name column has Strings generated by Java Secure Random and id is the row index. In order to force the application to generate a specific number of WHERE clauses, we provided the application with an original query of SELECT \* from mock\_data and a modified table containing a subset of the original table. An example of a query generated with 2 WHERE clauses is SELECT \* FROM mock\_data WHERE id=1 AND id=2. Figure 5 shows the performance as more WHERE clauses gets added for five different table sizes including 10, 100, 1K, 10K and 100K rows.

**5 Challenges**

Some of the challenges we encountered that we did not expect is just the time it took to get to the interesting part of the project, which is the query generation part. Although we made use of an existing Postgresql database, it took a while to get the parsing of user input and the modified query result file correct. Also, due to our decision to take the results of a query and store it as a String in memory for parsing, we had issues when trying to work with results that are large.

There were also a lot of subtle details we needed to worry about when parsing SQL queries, such as whether a column name is the alias or the original name and the various possible ways a valid SQL query can be written. We also had to assume a lot of situations won’t occur in order to simplify our application, such as assuming the user won’t make mistakes when modifying the query results, since the user would be directly modifying a text file. The application also cannot support subqueries and large query results.

While trying to generate a query for modified results seems relatively straight forward on paper, it has definitely been more challenging than we initially thought, especially when we were trying to come up with algorithms to do Projection and Selection.

**6 Related Work**

In our research, we have not been able to find a work that matches exactly what we are trying to accomplish. However, there are a few research papers trying to solve the problem of making SQL query writing easier for users, and are therefore relevant to us. It is interesting to see how they approached the problem and the methods they used.

One paper we found interesting is the paper about Interactive SQL query suggestions [2]. In this paper the authors attempt to make SQL query writing easier for inexperienced users by helping users formulate queries with *queryable templates* to model the structures of SQL queries. These templates are then ranked by their relevance to the keyword query provided by the user, such as “count database author”. Compared to our approach, the paper makes use of keywords provided by the user to generate queries, making it easier for someone totally unfamiliar with SQL to write queries. Our approach asks to user to modify the result to get their desired query and asks the user to provide an initial query, thereby assuming a little bit of SQL knowledge.

Another way to make query writing easier is to use a graphical user interface, such as the one presented in the paper Querying without Keyboards [3]. With the proliferation of next-generation computing devices such as tablets and smartphones, users are increasingly interacting with software without a keyboard. Therefore, it would make sense that some users would be more comfortable with writing queries using a GUI instead of a keyboard interface. In the QWiK database system proposer by the author, if the user brings the tiles for two relations close together, the interface would begin suggesting possible query actions and arrange attributes from both tables such that they are amenable to joining together. This would likely seem more intuitive to users compared to typing a specific SQL statement.

**7 Conclusion**

We have presented an application that tries to make SQL query writing easier for the user. This application allows the user to generate new SQL queries by having the user modify the result of an existing SQL query, and passing it into the application. The application that we created is a very crude implementation of the idea we came up with given the amount of time we had available. However, we hope that this application will help give other researchers some ideas and motivation to work on this problem.

Our application can successfully generate SQL queries with modified SELECT and WHERE clauses, however, the performance of our application degrades significantly with large tables. Although currently only the SELECT and WHERE clauses are handled, we believe that given time this application could help users come up with SQL statements that we would not have come up with ease originally. As such, we hope to be able to increase the performance and functionality of this application in the future.

**References**

[1] ZQL: a Java SQL parser. n.d. Web. 15 Mar. 2015.

[2] Fan, Ju, Guoliang Li, and Lizhu Zhou. "Interactive SQL query suggestion: Making databases user-friendly." In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 351-362. IEEE, 2011.

[3] Nandi, Arnab. "Querying Without Keyboards." In *CIDR*. 2013.