Query Generator and Visualizer – A Reverse Query Mapping System

Geoff McGinnis
Computer Science and Engineering
The Ohio State University
mcginnis.1930@osu.edu

Manjari Akella
Computer Science and Engineering
The Ohio State University
akella.4@osu.edu

ABSTRACT

In this paper we discuss our recent project which is a reverse query mapping and visualization system. This motivation behind this project is to provide a tool for beginner developers who do not have much exposure to SQL and DBMSs to get a hands-on introduction. In this program the user selects the data rows, columns, or cells they want to observe, and the application returns a query which can be used to retrieve that data. The user then also has the option to view a visual representation of the generated query, further explaining how the query would work. We feel these features combined provide for a solid foundation for learning about SQL and how data interacts with itself and the user. This has the potential to become an ongoing project with potential for new features to be added, but this paper describes the project as it stands within the scope of this project timeline.

Keywords – query generator, visualizer, apriori, selection, binning, where clause.

1. INTRODUCTION

Query generators and visualizers allow users to analyze data and learn about a database with out any prior knowledge of it. These applications essentially 'reverse engineer' a query and then break it down in a visual manner. The idea is to allow intuitive querying and to have a user select what they want to do using a simplified visual interface. The user picks and selects the tables, columns, predicates etc. and the system then converts this input from the user into a standard executable SQL query. The aim of our query generator is to simplify things even more. The user can basically know nothing about SQL and still be in a position to generate queries. The goal is to generate the simplest query, given the subset of data we have, as input to the system by the user. The query generated will always be valid and will return the data specified by the user when executed.

The visualizer part of the system shows the relationship between tables and how exactly a query is solved in the form of a pictorial representation. This way the user can easily execute complex and powerful queries without much effort using a simple point and click interface. An example of the visual output of such a generator is shown in Figure 1[1].

The following sections of this paper discuss this project in greater detail. Section 2 talks more about the project. Section 3 describes the motivation behind the idea. The design of the project; including behind the scenes algorithms are described in Section 4. Section 5 describes the performance and accuracy of the project. Section 6 discusses refinements and possible future work. Section 7 contains the conclusion.

```
Likes(person, drink)
Frequents(person, bar)
Serves(bar, drink, cost)
```

(a)

```
SELECT   *
FROM     Frequents, Serves
WHERE    Frequents.bar = Serves.bar
```

(b)

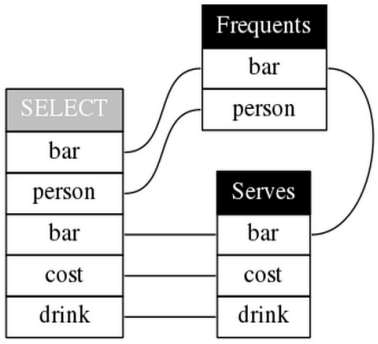

Figure 1: Visualization given schema (Serves (bar, drink, cost), Frequents (person, bar), Likes (person, drink))

2. PROJECT DESCRIPTION

The goal of our project was to try and generate the simplest possible query whose result is specified as input by the user. Though there exist many query generators [1][2][3], our search to find an existing system which takes as input a subset of data was futile, which is where we got the idea for our design. Our system takes as input collection of cells from a table and generates a valid SQL query which will generate the specified result. This reverse mapping process allows the user to see how an outcome can be established on a particular dataset.

One of the he most challenging parts of the project was establishing the relationships between the chosen data. Once these relationships have been established, we try to represent these in the form of predicates and finally establish the query. The generated query describes the data selected by the user, while also revealing details about the database itself.

3. MOTIVATION

The motivation behind creating this system is to educate users on the basics of relational databases and SQL queries. We feel that this will be a valuable teaching tool for beginner database developers who are new to RDBMSs and query scripting languages in general. The user begins with the end goal in mind as they select the specific data they want first, and then are able to see how that end goal is reached by learning an exact query that can be used to achieve that result. This system will teach the user about joining tables as well, as they will have to first consider for themselves what tables contain the data they want and how those tables are connected. The visualization feature of this application will help the user understand specifically what components of each table are interacting in the query. We also designed this project to be adaptable to different database schemas and user choices, as well as expandable for new features in the future.

4. PROJECT DESIGN

This application is implemented as a stand-alone program. We have been utilizing C# and Visual Studio to develop both the front-end user interface as well as the back end functionality. The available libraries and features of these tools simplified the development and allowed us to focus on design. Currently for testing, we are using a small sample database, containing 2 tables and about a 1000 entries as the data source for this application. We feel this is roughly the database size that would work well with this sort of project: there are enough rows and columns that the user has a diverse collection of data to choose from, but it is also not too big that the user can't find the data they want. One important aspect of our vision for this project is that the query generation does not rely on the schema design of the database, but works exclusively off the data contained within. Thus, our project makes little use of primary and foreign keys, designated column data types, or table interactions.

The following sections discuss some of the key features and algorithms of this project, and how we have implemented them.

4.1 User Interface

The user interface for our project was created using Windows Forms and is set up two sections. The left section contains the control buttons for generating the query and opening the visualization window. Below these buttons is the text field which will eventually contain the generated query once the user selects some data. This field is editable, so once a query is generated the user can make changes to it if they please.

The right side of the user interface contains the a tab view with each tab representing the tables of the imported database. Each tab contains an spreadsheet-like data table with each column and row of the corresponding table show (this data table is populated using a SELECT * FROM <table> query). These cells are selectable by individually CTRL clicking, clicking and dragging a selection of cells, or by selecting entire rows together. These selections are maintained as the user toggles between tabs, so that the user has the ability to select form multiple tables.

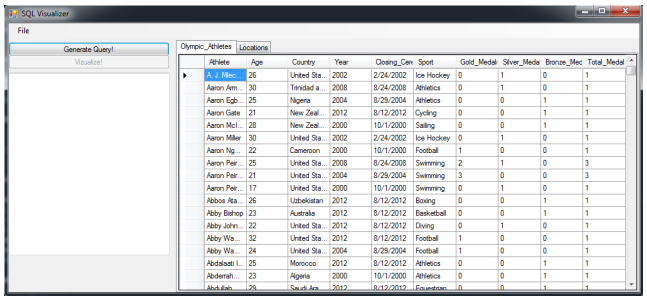

Figure 2: User Interface

The visualization window is another feature of this user interface, and is discussed in greater detail in section 4.5

4.2 Building the Queries

We have decided to make use of the QueryBuilder class from the open source project found in [4] to assist in building and formatting each query. This class features separate sub-classes for each query component and values that can make up a query. This allows us to add and remove query components dynamically at run-time based on the values received from the user. We were able to modify this class to suit are needs and fit in well with the rest of our system.

First, the program determines what tables are involved in the data selection. The table (tab) currently showing is added to the builder (assuming it has a selection within it), followed by any additional tables where data is selected. If multiple tables have selections, the next step is to determine the attribute to join on. This is described more in depth in Section 4.4. It is important to not here that there may not be any common attribute to be joined on, in which case the generator will display a message indicating this error. Next the generator determines what columns are involved in the selection. Every row selected does not need to select the same number of columns, but each column that has at least one cell in it selected will be included in the query. Finally, the WHERE clause is generated, which takes place in two parts. The following section will discuss this further. Once all the elements are added to the query builder, the finalized query is displayed in the text field on the user interface. Figure 3 below shows a flow chart for this process.

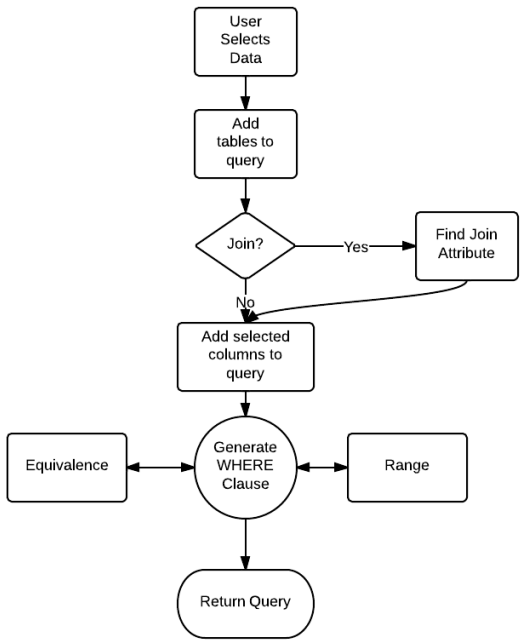

Figure 3: Process for building the query.

4.3 Generating the WHERE Clause

Finding commonalities in the data selected by the user is an interesting problem in this project. As stated before, we did not want to heavily rely on the schema design of the database, but rather on the data contents itself. With this in mind, we have decided to break up the problem into two sub categories, Equivalence and Ranges, which make for a fairly rich WHERE clause. The algorithms used are discussed in the following subsections.

One comment to make on this design is that is does not always generate the simplest WHERE clause possible, but the query is always valid and correct. We had considered doing some query optimization, but since this is intended to be a teaching tool we decided that doing so was not completely necessary.

4.3.1 Equivalence

Finding column values that all of the selected cells have in common can be looked at as a very small scale data mining problem. In this case, the sample population is small with each row representing a n-ary transaction (n being the number of columns selected), and the associations we are looking for are two-tuples of *(column_name, cell_value)*. With this in mind, it is clear that a modified version of the Apriori algorithm from [5] would be useful in finding these itemsets. Specifically, we are looking for any column value which can be found in all the selected rows. For now, the algorithm uses a minimum support of 1 (the value must be in all rows, i.e. WHERE age = 15), but we do plan on implementing the use of lesser support values to combine and make for a more interesting WHERE predicate (i.e. WHERE age = 15 OR age = 16). The modified Apriori algorithm used works as follows:

```
1.  supports  = dict {column_name : dict {value : support}}
2.  candidates = list [(column_name, value)]
3.  min_support = 1
4.  //generate the supports for all possible values
5.  foreach (value in SELECTED_CELLS)
6.      //increment the support for this column value pair
7.      supports[value.COLUMN][value.CONTENTS]++
8.  //find all frequent values
9.  foreach( col in supports)
10.     foreach(value in col)
11.         if (supports[col][val] / total_rows >= min_support)
12.             candidates = candidates + (col, val)
13.  return candidates
```

It can be seen that this is really just an adaptation of the first part of the Apriori algorithm, but we believe that this is sufficient to obtain the useful data we need. This algorithm is still a work in progress as the functionality of this feature is continuously expanding.

4.3.2 Ranges

Finding ranges of data is also useful when generating queries. For instance a user might select ages 15-20, and this needs to be represented appropriately in the WHERE clause. Our application makes use of a binning algorithm to narrow down columns of integer values into distinguishing max and min values. The following algorithm is a sample of how we determine these value ranges.

```
1.  num_of_bins = Ceiling(2 * values.count 1/3)
2.  values = [intger_column.selectec_values]
3.  bin_size = (values.max -values.min) / num_of_bins
4.  bins = new integer[num_of_bins]
5.  foreach(val in values)
6.      index = (val-min) / num_of_bins
7.      bins[index]++
8.  foreach(bin in bins)
9.      if (bin.Contains_Many_Values())
10.         lower = min + (index * bin_size)
11.         upper = min + ((index + 1) * bin_size)
12.         [combine adjacent bins, update upper and lower]
13.         return tuple(lower, upper)
```

One important feature of any binning algorithm is determining a correct number of bins to utilize. The number of bins used greatly impacts the quality of ranges returned by an algorithm like the one above. We decided to utilize the Rice Rule, a modification of the Sturgis Rule, [6] which determines the number of bins as a function of the cube root of the total number of items to be placed. An argument can be made as to weather or not this is the most effective method, but for this application we felt it was sufficient.

A tuple data structure was used to return the range as a pair of upper and lower limits. If the upper or lower value is contained in the columns selected data set, then the '>=' or '<=' operator is used, other wise the '>' and '<' are used. Figure 4 shows an instance of both equivalence and range queries.

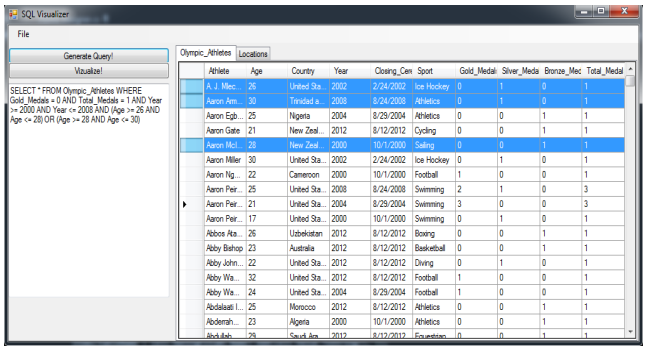

Figure 4: Range and Equivalence

4.4 Joining Tables

Another feature of this application is that the user may select data from multiple database tables, and use them both in the generated query. If cells from multiple tables are selected, we are able check if joining is at all feasible (Is there a join attribute?). Since the naming conventions differ in multiple tables for the same attribute, identifying join attributes needs something more along the lines of scheme matching. However that is outside the scope of this project and we have restricted our consideration only to cases where semantically same attributes are named the same. The algorithm below is a sample of how this application selects a join attribute and then generates a query for a two table join.

```
1.  selections = map{table_name:[selected_columns]}
2.  foreach(table in db)
3.      selections += {table: table.selected_cols}
4.  if (selections.count >1)
5.      foreach(table in selections)
6.          candidates = intersection(table, next_table)
7.          if (candidates.count > 0)
8.              query.add_join(table, next_table, candidates)
```

At this time the system is only capable of creating equijoin. We believe; logic similar to the one used in Section 4.2.2 can be used for other types of joins, including range joins. Figure 5 shows an instance of a join query whereas Figure 6 shows an instance where join is invalid.

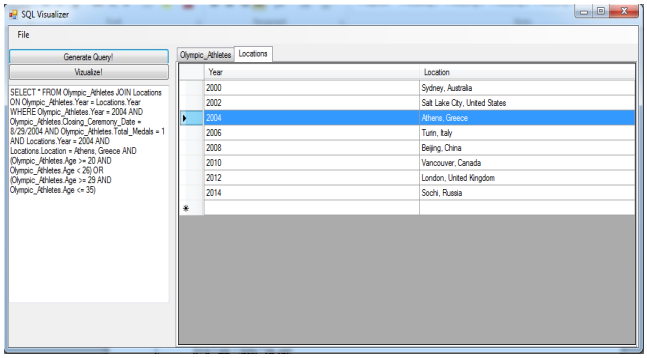

Figure 5: Join

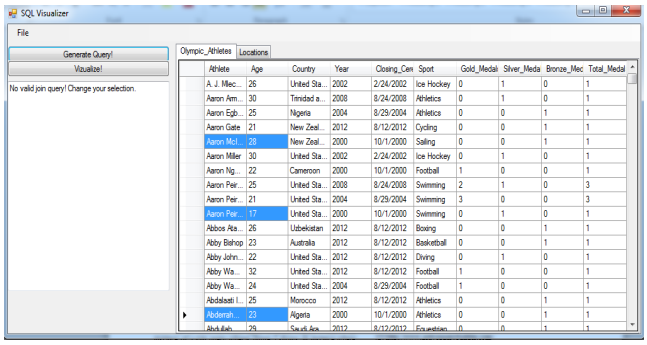

Figure 6: Invalid Join

4.5 Visualization

Another feature of this project is the visualization option of each of the generated queries. By selecting the "Visualize" button after generating a query, a second window appears which contains a graphical representation of the query generated. The visualization is formatted similarly to Figure 1 above. Each table involved in the query is represented as it's own rectangle list, with the table name as the header. Each column for the respective tables are listed with the table the rectangle as separate entries. Finally the table which are joined are connected by a line, with an indication to what properties the join is done on.

Since the current implementation of this project only allows no more than two tables to be joined, there is at most only two table records in the visualization. We feel this design and configuration is very expansible and would do well with an increased about of joining tables.

The visualization components were designed using XAML and Windows forms to allow for effective data binding and graphics formatting. Figure 7 below shows the visualization window.

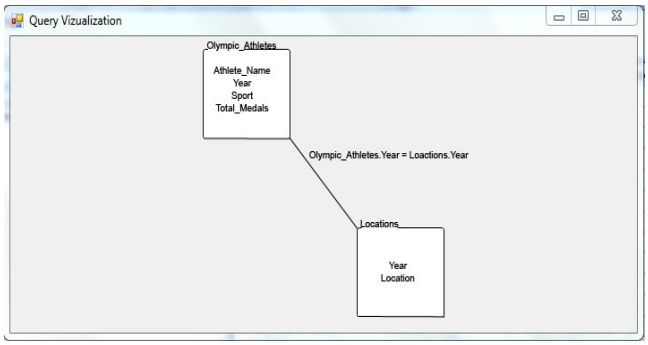

Figure 7: The Visualization Window

5. ACCURACY AND EFFICIENCY

One major consideration when designing this project was to ensure that every query generated was a correct and valid SQL query. We fell that we have achieved this as testing has shown that any combination of selected data will produce a valid query, which would generate the same data if executed. One downfall of our design, however, is that sometimes the query generated is not limited to just the selected data, but also may generate other results if executed. In other words, the generated query always is a good description of the data selected, but it may also describe data not selected.

Our application is able to process data and generate a query without any noticeable delay in processing or execution. Even though we have not collected any data on execution time, we feel that this program is efficient since no slow ups are noticeable to the user. This is largely in part of the considerably small amount of data that the user is able or willing to select. If this application was implemented on top of an enterprise level database, and the selection process was automated to allow for a much larger user selection, then performance effects would be more relevant.

6. FUTURE WORK

While this project can be considered a valuable educational tool in its current state, we feel there are additional features that would enhance it's potential that we considered outside of the scope of this initial project. First, we see the need for improvements to the specificity of the queries. That is, restricting the query to one that only generates the selected data, not more, which is sometimes the case. This would involve including schema elements in each query generation, most likely with the help of the primary and foreign keys. It would also be advantageous to allow the user to import their own database, that way the data is more relevant to the user. This would take a greater consideration to make sure different schemas and DB implementations do not act differently in the application.

Also it could be made a goal to simplify the queries as much as possible. If the user was working with our project in a corporate environment, they likely would desire the simplest possible query for time and cost savings. For the purpose of this project we feel the returned queries are simple enough, but acknowledge there is room for improvement.

Currently our project only allows the joining of two tables, but we realize many practical query scenarios involve joining much more than that. This would also make the visualization more interesting (and valuable) as there could be many tables interacting with each other.

Finally, our current project does not allow for a lot of user query customization. Some ideas we developed included options for the user to select, via the user interface, some of the query features to use – such as inner joins, group-by elements, or even nested queries – to make for more interesting and powerful queries. An option to execute the generated query could also enhance the learning element of the project.

7. CONCLUSION

Throughout this project we had the goal in mind of making an simple to use application to 'reverse engineer' an SQL query based on a selection of data. This started out with a very large project scope, and we sought to narrow it down to the important aspects of the idea: generating correct queries, a simple to use implementation that did not rely on the database's schema, and a useful graphical visualization feature. This set of goals is significantly smaller than our initial plans, but we feel through them we have created a well rounded and useful application. We took great consideration into the implementation details, and really thought about what we would want as a user. Through these efforts we learned a lot about what an SQL engine does (we were effectively writing one in reverse) as well as how quality data associations can enhance any query. We know SQL and relational databases are not an easy concept to pick up on, but we hope that our application will

8. ACKOWLEDGEMENTS

We would like to thank Dr. Arnab Nandi who oversaw this project and provided helpful feedback and ideas when we needed. Another thanks to Jonathan Wood for authoring a QueryBuilder class in [4] which we were able to modify to be used in this project. Finally we would like to thank the friends and classmates who we bounced ideas off of during this project.

9. REFERENCES

[1] http://queryviz.com/index/QueryInterpretation.png

[2] http://www.activequerybuilder.com/

[3] http://www.razorsql.com/docs/query_builder.html

[4]http://www.blackbeltcoder.com/Articles/strings/a-sql-querybuilder-class

[5]http://rakesh.agrawal-family.com/papers/vldb94apriori.pdf

[6]http://www.stat.rice.edu/~scottdw/stat550/HW/hw3/c03.pdf