MySQL Auto-Complete

Vivian Ellis and Athena Mason

December 4 2017

Abstract

MySQL Auto-Complete is a database design tool that assists users in building queries by making context-aware suggestions. We designed this application with the user in mind and built suggestions based upon the current state of the query.

1 Introduction

MySQL Auto-Complete is a database design tool that assists users in building queries by recommending context-aware suggestions. We assume the user has moderate knowledge of MySQL and understands how to build queries. Our intent was to provide the user with practical suggestions but not limit the suggestions to only generate when the user chooses from our available list. The second section talks about the problem description and our goals of this application. Section three talks about the user interface and the architecture behind our application. We briefly discuss our testing in section 4. Section five presents relevant work that inspired us. We then conclude with our future work in MySQL Auto-Complete.

2 Problem Description

MySQL Auto-Complete will allow the user connect to an available database and aid the user in maintaining and querying data. Developing queries can often be complex; MySQL Auto-Complete will assist users in the creation of MySQL commands by making context-sensitive suggestions for the completion of the query. The application will be aimed at users with a moderate knowledge of MySQL, with the goal of improving efficiency and accuracy.

2.1 Purpose

We had an interest in the application since we have both used MySQL in our study and work. We wanted to make an application that would satisfy our needs for making useful suggestions and building queries seamlessly. We do not require the user to know the schema of the current database they choose since the Auto-Complete application generates suggestions.

2.2 Goal and Objective

We want our application to be as robust as possible making suggestions in all clauses when applicable. Each clause has its own suggestions based upon other dependant clauses. This means that the user will have to understand the structure of MySQL statements to decide which suggestions will provide the desired results.

3 Architecture

3.1 Overview

Our application is built in C# using the .NET framework. We will first discuss the front end and how the user interacts with the program. Then we will move on to how the suggestions are being formulated.

3.2 User Interface

During the early stages of development we started out using a text box. The text box has a built-in Auto-

CompleteMode property. This allowed for text completion where the suggestions would come from an AutoCompleteCustomSource, a collection that would be filled from a string array. We quickly realized that the text box, while it did offer an auto complete property, did not have versatility. We wanted a seamless design which lets the user type freely in a multiline setting. The text box does have a multiline property. However, turning the multiline property on prevents AutoCompleteMode from functioning. We decided to revisit our options because using a text box would mean having to create an individual text box for each clause. This was not an optimal choice.

A rich text box permits advanced text entry with paragraph formatting so the user can enter their choice of clauses with no restrictions. Using a rich text box did not offer a built in AutoCompleteMode property. To get around this we used a list box, a drop down list that the user may select an item from. The items in the list box is a collection of strings, which is similar to that of the AutoCompleteCustomSource. We concluded that a rich text box and a list box would be the best way to display the suggestions and allow the user to choose from these suggestions. We set the list box location to the current position of the cursor in the rich text box. This way the list box will follow the cursor, and only appears when there are items in the list box to be displayed. We also track if there is a change in the position of the cursor via mouse click. This is useful for detecting the current clause the user is in, which is discussed further in section 3.7.

3.3 Capturing Key Press

Each key press within the rich text box is captured. There are many cases that need to be checked each time a key on the keyboard is pressed. A string, “keyword”, is added to each time we do not detect a space, backspace, return, parentheses matching, equals sign, a semicolon, or comma. The new key press will be appended to the “keyword”. From here the list box is filled depending upon the state of the “keyword”. If the “keyword” contains one char, we build the suggestions from a new list, otherwise we simply filter the existing list. A list copy holds all possible suggestions built from QueryParser.Suggest, talked about in section 3.8. The Suggest method in QueryParser returns a list of strings that fills the list copy. From here, more suggestions may be added depending on the current query the user is in. For example, if the user is in the SELECT or HAVING, all aggregate functions will be added to the list copy. There is also limited functionality where MySQL Auto-Complete suggests the subsequent clauses (e.g. if in the FROM clause, it will suggest WHERE), but currently there are hard-coded into the suggestions and do not take into account anything besides the current clause; e.g. if in the FROM clause, it does not determine whether a table has already been typed into the query. Once the list copy is updated with all relevant suggestions for the current clause, we then filter the strings in the list copy. We do this by checking if each string in list copy starts with the “keyword” and ensure it is not currently in the list. Each string that passes these conditions is then added to the list box of suggestions. This process repeats until the “keyword” is cleared out by a return, comma, semicolon, etc.

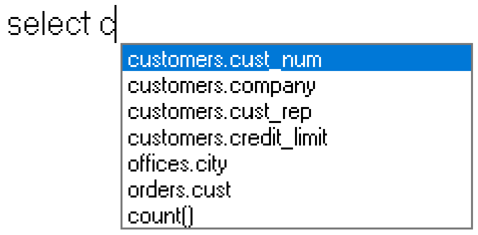
If the user enters a return, we know that there will be no suggestions to be made, so we clear out the “keyword” and all items currently in the list box and hide it from the user. This could be a problem if the user accidentally clicks return and then enters back into the line they were previously on. We do not keep track of previous list box suggestions; once they are cleared out they must be recalculated. This could slow down our suggestions if the list box contained a large number of suggestions.

If the user enters a space or equals sign we fill the suggestions in the list box similarly to when they add to the “keyword”. However, we add all suggestions and do not filter out unmatched suggestions. We are currently ignoring commas because we assume if the user types a comma they want to enter a list of attributes within the same clause.

Each backspace that the user presses deletes the last key press that was appended onto the “keyword”. Then, we look for the new “keyword” in the list copy and add any string that starts with “keyword” back into the list box of suggestions.

At this point if there are no suggestions we simply hide the empty list box, but keep the “keyword” intact in case we detect a backspace and have to add suggestions back in.

When in the SELECT clause we consider two options when filtering suggestions: the user is searching for a table that begins with the "keyword" or the user is searching for an attribute that starts with the "keyword". Let the keyword = 'c' and the selected database be sales. The user begins typing the query where no table is defined.



```
select c
customers.cust_num
customers.company
customers.cust_rep
customers.credit_limit
offices.city
orders.cust
count()
```

Figure 1: Select attribute with undefined table

Both the tables with columns that start with 'c' are included as well as tables that start with 'c' (refer to figure 1). See section 3.8 for an in depth discussion of how the suggestions are found.

On each semicolon we clear out all contents of the list box suggestions, list copy, and the "keyword". The query is then split. We scan the entire rich text box text and split at the semicolon and store each query into a string array. It is important at this step to delete any null or empty entries. Otherwise we may identify the incorrect query the user is in when executing the query.

3.4 Selecting Suggestion

When selecting an item from our list box of suggestions currently only double clicks are supported to choose a suggested word. We find where the cursor is currently located and insert the selected word into the rich text box at that location. After insertion we clear out all suggestions from the list box suggestions and the list copy. The list box is hidden and the "keyword" starts over.

3.5 Multiple Databases

Our Auto-Complete application supports multiple databases. The user has the freedom to select from an available database. The database is chosen via a drop down list that connects to MySQL and builds the DatabaseSchema, seen in section 3.6. We only make suggestions when a database is selected since our goal is to offer useful suggestions based upon the context. When running a query the user must be connected to an available database. Before execution, if there are multiple queries we preform a split and search for the query the cursor is currently in. To do this, we count the total number of queries by semicolon and then count the number of queries before the cursor, also by semicolon. Then we simply grab the current query from the string array of all queries.

Currently, our program is set up to support the USE statement. We suggest the USE statement if we detect no database is chosen and the "keyword" is 'u'. We fill the suggestions of the list box with available databases. Once a suggested database is chosen or the user manually types the database the application switches to the new database. If we detect the user is manually switching the database we count the number of USE clauses, splitting the rich text box by each semicolon. If the user has three USE clauses with queries in each:

```
use music;
select count(admin_id)
from administrators;

use bookstore;
select AuthorID
from author, bookauthor;
select AuthorID, ISBN
from author, bookauthor, books, orders;

use university;
select count(ID), name
from instructor join teaches using (ID)
group by ID;
```

We insert each query into a 2D string array, "allQueries" where the first string in each row, in column 0, is the database and each row is the group of

queries by database. Following is each query being implemented in those databases.
allQueries = {{"music","select count(admin_id) from administrators"},
{"bookstore","select AuthorID from author, bookauthor"},
{"university","select count(ID), name from instructor join teaches using (ID) group by ID"}}
There are several bugs with manually switching databases. While the application can detect the current database the suggestions are not based off the currently in use database. This feature is currently disabled and the user must select a database from the drop down menu to ensure accuracy. In the future work section, we discuss our plans to fix the USE statement.

3.6 Reading Schema

When a database is initially selected, MySQL Auto-Complete automatically reads in the schema from the database's Information Schema table. The BuildSchema(String database, MySQLConnection conn) method runs the query "SELECT TABLE_NAME, COLUMN_NAME, COLUMN_KEY, DATA_TYPE FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA = [database name] ORDER BY TABLE_NAME;" and saves the results to a DatabaseSchema object.

We created the DatabaseSchema object for the dual purposes of holding the current database schema, including table and column data, and the aliases in the current query. The aliases are further discussed in section 3.9. In this section, we will focus on the structure of the schema. The tables and columns are held in a Dictionary<String, List<Column>> object. The Dictionary object is essentially a hash table, holding key-value pairs. In C# Hashtable objects are weakly typed, while Dictionary objects are strongly typed. We wanted to ensure that data would be stored in pairs of <String, List<Column>>, so we used the Dictionary object. Like Hashtables, Dictionaries use a hash lookup and therefore it is highly efficient to access data stored in a Dictionary.

For each tuple read in, if the TABLE_NAME does not exist in the schema, it is added as a key to the Dictionary. The value of the key-value pair was a new List<Column> object, which begins as an empty list. Once the table has been added to the Dictionary, a new Column containing the COLUMN_NAME, TABLE_NAME, COLUMN_KEY, and DATA_TYPE is added to that table's list of Columns. This continues until all tables and columns have been added to the schema. From this point on, the DatabaseSchema object's schema Dictionary is referenced any time MySQL Auto-Complete needs to read any information about the database. Currently, MySQL Auto-Complete does not update the schema except when a new database is selected. Users would be able to run a CREATE TABLE query, but at this time it would not result in the schema being updated to include the new table. This is discussed further in section 5, future work.

Initially, we stored the schema as a Dictionary<String, Dictionary<String, String>>, and did not contain it within a DatabaseSchema object. This was problematic for several reasons. First, it obfuscated the purpose of the object for any other developer who might look at the code, as it is unclear what each of the strings represented. The first string (Dictionary< **String**, Dictionary<String, String>>) represented the table name, which is used as the key. This continues to be true in the current iteration of the program. The value within the first Dictionary (Dictionary<String, **Dictionary<String, String>**>) represented the column object, with the key in this dictionary representing the column name, and the value representing whether the column was a primary key, foreign key, or not a key.

This leads to the second issue with this method of storing the schema: it limited the information that could be stored in the column. Having the datatype as well as the column name and key status was necessary to make the JOIN suggestions useful, and storing this in the column Dictionary was not possible without doing something such as including another nested Dictionary, which would obscure the purpose of the object even further.

Finally, creating a DatabaseSchema object to hold the schema allowed us to create methods specifically for the manipulation of a database schema, rather than allowing any properly structured dictionary to be passed into the method.

The first two issues were resolved with the creation of a Column object. The Column contains the column name ("name"), the table the Column is stored in ("table"), the data type of the Column ("datatype"), and whether the column is a key ("key"). Each of these is stored as a string, with a null key value representing a column which is neither a primary key nor a foreign key. Having the table name inside the Column is slightly redundant, as all columns are stored as values within the schema Dictionary, with the table name as the key. However, the increased ability to manipulate columns directly, without having to reference the Dictionary they are stored in, made up for this slight redundancy.

The Columns are comparable objects and can be sorted based on their key value (primary, followed by foreign, followed by none), and then by name. The Column class also contains two Comparers, allowing Columns to be compared by name or by datatype. Columns can be converted to a string with or without an alias. The ToString() method returns a string in the form "table.column_name." The Column class also contains a ToString(String alias) method which returns a string in the form "alias.column_name." There are also two methods–GetColNames(List<Column> columns) and GetQualNames(List<Column> columns, DatabaseSchema schema)–which return the names from a list of columns. GetColNames simply returns the "name" string from the Column objects, while GetQualNames returns fully qualified column names ("table.column_name"), with the alias in the place of the table name if an alias exists. See section 3.9 for discussion of how the aliases are read from the query. These methods are instrumental to generating the Auto-Complete suggestions, which are discussed in section 3.8.

3.7 Identification of Clauses

The suggestions for MySQL Auto-Complete are context-sensitive and rely on being able to accurately identify the current query when multiple queries have been entered, as well as being able to accurately identify the current clause. See section 3.3 for discussion of how the current query is identified. This section will focus on the itentification of clauses within the query.

Initially, we began by assuming that the first word in any given line was the clause. However, it is not required in MySQL for users to put different clauses on different lines, and so we added functionality to detect the current clause regardless of whether there is a new line. The clause is identified by first finding the current position of the cursor within the query. This is done by finding the cursor position within the entire rich text box and looking backwards in the box until a semicolon is found. The difference between the current location and the index of the semicolon represents the position within the current query.

The current query, a list of MySQL keywords, and the position within the query are passed to a FindClause method within the QueryParser. The QueryParser is discussed in more detail in section 3.8. The current query is split into a substring which contains only the portion of the query which comes before the cursor. This substring is then split into an array of strings, such that each word in the query is its own string. The method works backward through the array, comparing each word to the list of keywords. Once a MySQL keyword is found, it is returned as the current clause. For the "order by" and "group by" clauses, we ignore the "by" in our search.

3.8 Auto-Complete Suggestions

The QueryParser class contains all the methods for reading the features of the current query and making suggestions for the list box. The only method within QueryParser which is referenced from outside the class is the Suggest method, which takes the current query, the current clause, the DatabaseSchema, and the position within the query as arguments. The Suggest method contains a switch statement, which calls the correct suggest method based on the current clause. The query is then parsed into its relevant sections in order to provide useful, context-sensitive suggestions. MySQL Auto-Complete offers suggestions for each of the following clauses: SELECT, FROM, WHERE, GROUP BY, ORDER BY, JOIN ON, HAVING, UPDATE, SET, DELETE, INSERT, and USE.

The SuggestSelect method first calls the ParseTables method, which identifies any tables already contained within the FROM clause of the query. It does this by splitting the FROM clause into individual words, and then checking each words against the keys in the schema Dictionary. Once it identifies the tables in the clause, it moves on to determining the fields to suggest. For example, a user may type the following query:

SELECT cust_num FROM customers

The user could insert their cursor within the "SELECT" clause, and MySQL Auto-Complete would determine that "customers" was included in the FROM clause of the query. It would then suggest only the columns in customers to the user. It does this by finding the "customers" key in the schema. The value associated with that key is a list of the Columns in that table. The list of Columns is then passed to GetQualColNames, which returns the fully qualified column names for each of the columns in the customers table. If there is no FROM clause, or there are no tables currently in the FROM clause, the SuggestSelect method instead returns a list of all the columns in all tables in the current database.

At the most basic level, the SuggestFrom method works similarly, except instead of determining the tables which are already in the query, it determines which fields are in the SELECT clause. If the field in the select clause is fully qualified, the method splits the field on the period, keeping everything before the period as the table name and everything after as the field name. If the table is in the database, it is included in the suggestions, and the field is no longer considered for suggestions. Once all tables in fully qualified field names have been added to the suggestions list, any remaining fields are checked against the values in the schema Dictionary. For each table, if the list of Columns contains the field, the table is added to the list of suggestions. SuggestFrom also automatically suggests a list of all the tables associated with the fields, in the form "table 1, table 2, ... table n." Currently, it only suggests a list of all tables, not every combination of tables. For example, if there are three tables, it will only suggest "table1, table2, table3," and will not suggest "table1, table2," "table1, table3," or "table2, table3." This may be something to add later, as discussed in section 5.

When the user types a valid table name in the FROM clause of a query, followed by a space, MySQL Auto-Complete suggests INNER JOIN, OUTER JOIN, or CROSS JOIN. If the user selects or types a JOIN, the FindQuery method identifies the user as being in the JOIN clause, and the SuggestJoin method is called rather than the SuggestFrom method. The SuggestJoin method works similarly to the SuggestFrom method, and both methods call the SuggestTable method to determine which tables to suggest. The difference is that in the SuggestJoin method, if the last word before the cursor in the query is a table, it suggests "on" rather than suggesting more tables. In addition, if there is only one table in the FROM clause, it suggests all the tables in the database to select from.

The SuggestOn method works differently depending on the current state of the query. There are two SuggestOn helper methods, and which one is called depends on whether the first field in the join had already been selected. If there is nothing in the ON clause, the method returns a list which contains suggestions in the form "table1.column=table2.column," as well as all of the fields included in the SELECT clause of the query. If the query already contains "table1.column =," the method simply returns suggestions in the form "table2.column."

If the first column has not been selected, for each column in table1, the SuggestOn method compares that column to each column in table2. Suggestions are prioritized in the following manner:

- Primary key/foreign key pairs, where both columns have the same data type
- Pairs where one column contains the name of the other (e.g. "cust_num" and "cust")
- Any other pairs of columns with the same data type

Suggestions for if the first column have been selected work similarly, except that rather than comparing all the columns in table2 to all the columns in table1, the table2 columns are only compared to the selected column.

The suggestions for other clauses are simpler and will only be discussed in brief. The method to determine the current tables and fields in the query is the same for all clauses. In the ORDER BY, WHERE, and HAVING clauses, suggestions simply consist of the current fields. In the GROUP BY clause, current fields are parsed the same way as in other clauses, except any fields containing parentheses (that is, aggregate functions) are discarded, and it only returns a list of non-aggregate fields currently in the query.

MySQL Auto-Complete also provides limited support for non-SELECT queries. The suggestions in the UPDATE clause are the same as in a FROM clase, minus anything related to joins. The suggstions for the SET clause are the same as in the SELECT clause. For INSERT INTO and DELETE commands, it simply provides a list of tables in the current database.

3.9 Aliases

MySQL Auto-Complete provides support for aliases throughout, although there have been situations where the aliases did not appear in suggestions for reasons unknown to us. These issues are sporadic and will need to be corrected in the future. Aliases are stored in two separate alias Dictionaries stored within the DatabaseSchema object. Table aliases are stored in a Dictionary<String, String>, where the first string represents the alias and the second string represents the table name. Column aliases are stored in a Dictionary<String, Column>, where the first string still represents the alias and the Column represents the column.

Aliases are detected any time the user switches clauses, either by typing a new clause name or by clicking elsewhere in the query. Aliases are identified as any word after "as," or any word in the SELECT, FROM, UPDATE, or SET clauses which is not in the database schema or in the list of MySQL keywords.

Throughout all other methods, whenever a method checks if a column or table is contained in the database, the column or table is checked against the appropriate alias Dictionary. If the word being passed through is an alias (that is, it is a key in the alias Dictionary), the column or table it is an alias for is returned. If it is not an alias, the word itself is checked against the schema Dictionary.

When fields are being returned, "field.ToString(schema.FindTableAlias(table))" is added to the list. The method FindTableAlias(table) checks the field against the Dictionary of table aliases. If the table has an alias, the alias will be returned. Otherwise, the table name will be returned. Either way, the result is passed to the ToString method, which returns "table.field," where table is the value returned by FindTableAlias. This is true in the SELECT, SET, WHERE, GROUP BY, and ORDER BY clauses.

When making suggestions in the ON portion of the JOIN, for each column being returned, the column name is checked against the column alias Dictionary, just like in the other clauses. However, if the first field has already been entered ("table1.column = "), an extra step is needed to return the suggestions. The table and column in "table1.column" are individually run against the table and column aliases Dictionaries. The result of this method (the Column being aliased) is then used to determine the other column in the manner described above.

4 Test

We performed limited testing on the latency of our program. The databases we used were small, and it is unknown how much scaling up the database would increase latency. As it stands, it took between 1 and 3 milliseconds for MySQL Auto-Complete to read in the schema of the three databases we tested. When running the program, we found that no suggestion took more than 24 ms to fill the list box, and on average the time was between 5-10 ms. Adding a small number of aliases (two table aliases and one column alias) did not noticeably increase the suggestion time, and adding the aliases did not cause the time to increase beyond the 24 ms maximum noted earlier.

Future testing could include testing the latency with larger databases including more tables, more columns, and/or more data. More complex queries could also be tested.

5 Future Work

For our future work we plan to implement more options for the user to build queries.

In section 3.5 we discussed how the user can implement multiple USE statements. To fix issues with the suggestions when manually switching databases, we have a few theories. Our application does not fill the DatabaseSchema on the USE case and does not recognize a database is being used. It is also is still picking up the USE as the current clause even once the user explicitly types SELECT. To begin fixing these issues we would like to add more conditions to track when the user has exited the USE statement and ensure the DatabaseSchema is being filled.

We would like to add a feature that will keep previous suggestions made, as well as if the user has selected one of our suggestions. This would allow us to further refine suggestions. This would also would eliminate the need to search and build suggestions each time he user enters a new keyword.

In addition, we would like to add the creation and manipulation of databases. This would assist in the creation of databases as follows,CREATE DATABASE <DATABASENAME>; Our application does not currently support suggestions for creation of tables. We would like to identify when the user has initiated a CREATE TABLE clause and suggest data types and column proprieties. Once both of these attributes are available to the user we could further support INSERT INTO <DATABASENAME> with suggested datatypes.

We would also like to suggest combinations of tables. Currently, when no attributes are defined in the SELECT clause only single tables are listed as suggestions. If a database contained n number of tables we would like to add as many as $\binom{n}{2} + \binom{n}{3} + ... + \binom{n}{n}$ suggestions.

6 Related Works

There are several existing programs which perform similar functions to MySQL Auto-Complete. They all seek to streamline the process of completing queries by using assistive features. However, they vary in their approach.

6.1 SnipSuggest

SnipSuggest is a middleware layer on top of a standard RDBMS. This software does not automatically provide suggestions as the user types. SnipSuggest maintains a continuous log of user queries and when requested by the user, it provides a recommendation for the selected clause. The suggested queries are based on the previously defined clauses, frequently referenced fields, and tables in the log.[1] While some details of this software's implementation may be of interest, the other applications are more directly comparable to our MySQL Auto-Complete project.

6.2 ApexSQL Complete and dbForge SQL Complete

ApexSQL Complete v2017.07 is a free tool for Microsoft SQL Server that provides automatic insertion of tables, fully-qualified field names, and one-click completion of fragments. These features are similar to those implemented in our project. Another free application compatible with Microsoft SQL Server Management Studio and Microsoft Visual Studio is dbForge SQL Complete. Like ApexSQL, it provides context-based code completion. It also includes additional features such as assisted refactoring, automatic formatting, and reusable SQL templates [2].

7 Conclusion

We have successfully built an application that completes the user's text as they type. This application fulfills our goal of offering diverse suggestions dependant upon the current location in the query, as well as what fields and tables have already been included.

The database schema is loaded in upon database selection, which allows us to reference information such as the table name and each column in the table, if the column is a primary or foreign key, and the data type of the column. Suggestions are filtered after each key press, ensuring that only the most relevant suggestions are offered to the user.

References

[1] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. Snipsuggest: Context-aware autocompletion for sql. *Proc. VLDB Endow.*, 4(1):22–33, October 2010.

[2] Devart. Download dbforge sql complete.