SQuilL: SQL Visual Learning

Nicholas O. Fullerton

CST 499, California State University, Monterey Bay

June 13, 2016

Table of Contents

Executive Summary

The purpose of this paper is to propose development and deployment of a software application, SQuilL, the author intends to produce as his capstone project to serve as a learning and quick-reference aid to visualizing the effects of Structured Query Language (SQL) statements on one or more tables. Structured Query Language is a ubiquitous language used to communicate with databases. SQL databases can be conceptualized as collections of grid-like structures called "tables" and each row in a table represents a "record" (this is an oversimplification but will suffice for the sake of explanation). Each column in a table represents a common attribute of a set of records in a table. A basic and typical table named "Employees," for example, would have columns for attributes such as "emp_name," "pay_rate," "job_title," etc representing the employee's name (a string of letters), the employee's rate of pay (a number), and the employee's job title (another string of letters). Many tools exist for visualizing unique databases at the block-diagram (or "Entity-Relationship") level, but none exist to graphically display the effects of SQL statements on one or more tables. What the author proposes to address the lack of table-level SQL visualization tools is an education and reference application called "SQuilL," so-named because it is a word which contains the letters "S," "Q," and "L"; and when spoken sounds like the word "skill" – a play on words as the program is intended to help the user build and/or maintain SQL skills by seeing exactly how their SQL statements affect one or more tables. An ancillary note is that SQuilL was named after the purple Squill *(Scilla siberica)* flower which will be used as its logo.

SQuilL would be a desktop Graphical User Interface application featuring a workspace where database tables will be represented graphically as spreadsheet-like objects, with a text area for the user to enter SQL statements. The rows, columns, and/or cells in the displayed tables will

highlight based on the SQL statement entered by the user. For example, if a user has 2 tables and writes a SQL statement to provide a summary of a few fields from each, the appropriate rows/columns/cells in both tables will highlight. Rows/columns/cells to be deleted may be highlighted in a different fashion to show the user the impending effects of their statements on their tables.

The primary purpose of SQuilL is as an educational tool geared toward students, to assist visual learning of the effects of SQL statements – though it may also serve as a quick-reference to anybody who uses SQL. The tool would be important because it fills the niche of interactive SQL visualization at the table level, providing a more intuitive and complete learning experience for the SQL student or professional.

Ideological Goals

The primary ideological goal from the end-user perspective is to serve as a visual, interactive education and reference tool for all users of SQL, so this section will outline the ideological goals of SQuilL's development and deployment:

- SQuilL is meant to be as accessible as possible, which means that it will be released as Free and Open-Source Software under one (yet to be determined) of the many existing software licenses such as the BSD, GNU GPL, MIT, Apache, etc.
- SQuilL is meant to be lightweight in both logical size and with respect to resource consumption such that people with older hardware may run it without experiencing undesirable behavior.
- SQuilL is meant to allow for a user-interface which supports languages other than English, ensuring that there will be no language barriers as long as there is a willing

translator. As an educational and reference tool, it is not possible for SQuilL to be used directly for malevolent purposes.

- SQuilL's ideology subscribes to the adage "information wants to be free" and that dissemination of not only the application, but education and knowledge in general, serves the greater good of humanity.

Technical Goals

The following section will outline the technical goals required or recommended to implement the application. Not all of the following features will be promised as the core functionality will be first implemented, followed by extra features as time allows. The proposed technical features are as follows:

- The application will be a standalone desktop application developed using a modern, memory-managed language such as Java or C# to facilitate safe, rapid development.
- The application will provide a graphical user interface using spreadsheet-like objects to represent database tables, and will handle as many tables as the computer resources will allow.
- Users can name or rename tables and attributes, but the default naming convention will be tables named, "Table_1," "Table_2, etc.; with default row values to be determined and default column values as letters ("Column_1," "Column_2" etc.) as in spreadsheet software.
- The user input/output between statements and tables will be bidirectional, that is, the user can generate statements based on highlighting tables and subsections thereof; or the application can highlight tables and table subsections in response to statements entered by the user.

- The user will be warned when improper statements or operation are detected.
- Robust context (right-click) menus to allow for naming and manipulations of tables and other options will be implemented.
- Users will be allowed to save and load schemas (collections of tables) of their own design.
- Users will be allowed to connect to SQL databases and import database schemas into a format they can manipulate graphically, committing changes to the database from within SQuilL.

End-Users

SQuilL is intended to be used primarily by students of all levels who wish to use an interactive visualization of the effects of SQL statements at the row/column/cell level. Many students are primarily visual learners or grasp certain concepts only visually, and yet, no such interactive tool exists to aid understanding at the intra-table level. The tool could also be used by instructors to graphically show students certain concepts. SQuilL may also be used in an educational and reference context outside of academia – for example, an experienced programmer with no database experience could use SQuilL as a supplement to their on-the-job training; or a new graduate could use SQuilL to aid learning in their first entry-level position.

Project Justification

The concept of an application to visualize SQL database operations at the table level and as an educational/reference application was conceived by the author, approximately six months before the writing of this paper, and considered seriously as an idea for a capstone project shortly thereafter. The first thoughts regarding the project surfaced while the author was enrolled in a database class (CST 363) and lamented the fact that, while simple operations were trivial to

visualize, more complicated SQL queries such as JOINs across multiple tables were more difficult to visualize. A JOIN SQL statement combines elements of two tables, compound SQL statements can include or exclude multiple elements across more than two tables (RJMetrics, 2016). Cursory internet research reveals that there are plenty of SQL database visualization tools available. MySQL Workbench, for example, includes tools available for design and implementation at the schema level as well as allowing manipulations of data at the table level – as an example, consider the following quote from the MySQL workbench design web page, according to Oracle Corp (2016),

> *"MySQL Workbench provides capabilities for forward engineering of physical database designs. A visual data model can easily be transformed into a physical database on a target MySQLServer with just a few mouse clicks. All SQL code is automatically generated and runs right the first time, which eliminates the normal error-prone process of manually writing complex SQL code. MySQL Workbench also enables you to reverse engineer an existing database or packaged application to get better insight into its database design. Not only can MySQL Workbench forward and reverse engineer existing databases, but it can also import SQL scripts to build models and export models to DDL scripts that can be run at a later time."*

Very impressive, but that level of functionality is geared more towards seasoned professionals and does not provide the basic but specific table-level visualization needed. The author's own experience with MySQL Workbench, while overwhelmingly positive, involved a learning-curve unwieldy for casual education and required significant computational resources to run MySQL workbench – Significant resource requirements are what SQuilL attempts to avoid in its development and deployment.

Other tools such as the online SQL Joins Visualizer, exist to visualize SQL statements but do not use interactive table structures as their graphical aid. The vast majority of visualization tools available are either static non-interactive charts (as seen when reading a book) or interactive non-table graphical objects such as Venn diagrams as seen in the *MySQL Joins Visualizer* website (Vasilev, 2015). MySQL Query Analyzer does not examine operations on a per-table or per-column basis, but rather analyzes query statistics such as execution time and data transfer. Yet more applications also allow for creation and editing of tables and databases but fail to visualize per-table operations in the manner desired by the author. The specific niche of SQL query visualization desired but not yet discovered may be seen in the following example, a mock-up of how exactly the tables and Query selections may be visualized, coded in the C# language using Microsoft's Visual Studio 2012 as the Integrated Development Environment.

If a user enters a query to display only the employees' first and last names from the first table along with only the employees' pay rates from the second, then the user enters the textual Query and the SQuilL application marks (highlights) the data being requested. The hope is that SQuilL will eventually be able to visualize SQL queries of arbitrary complexity and across and arbitrary amount of tables, both limited only by the resources of the user's computer.

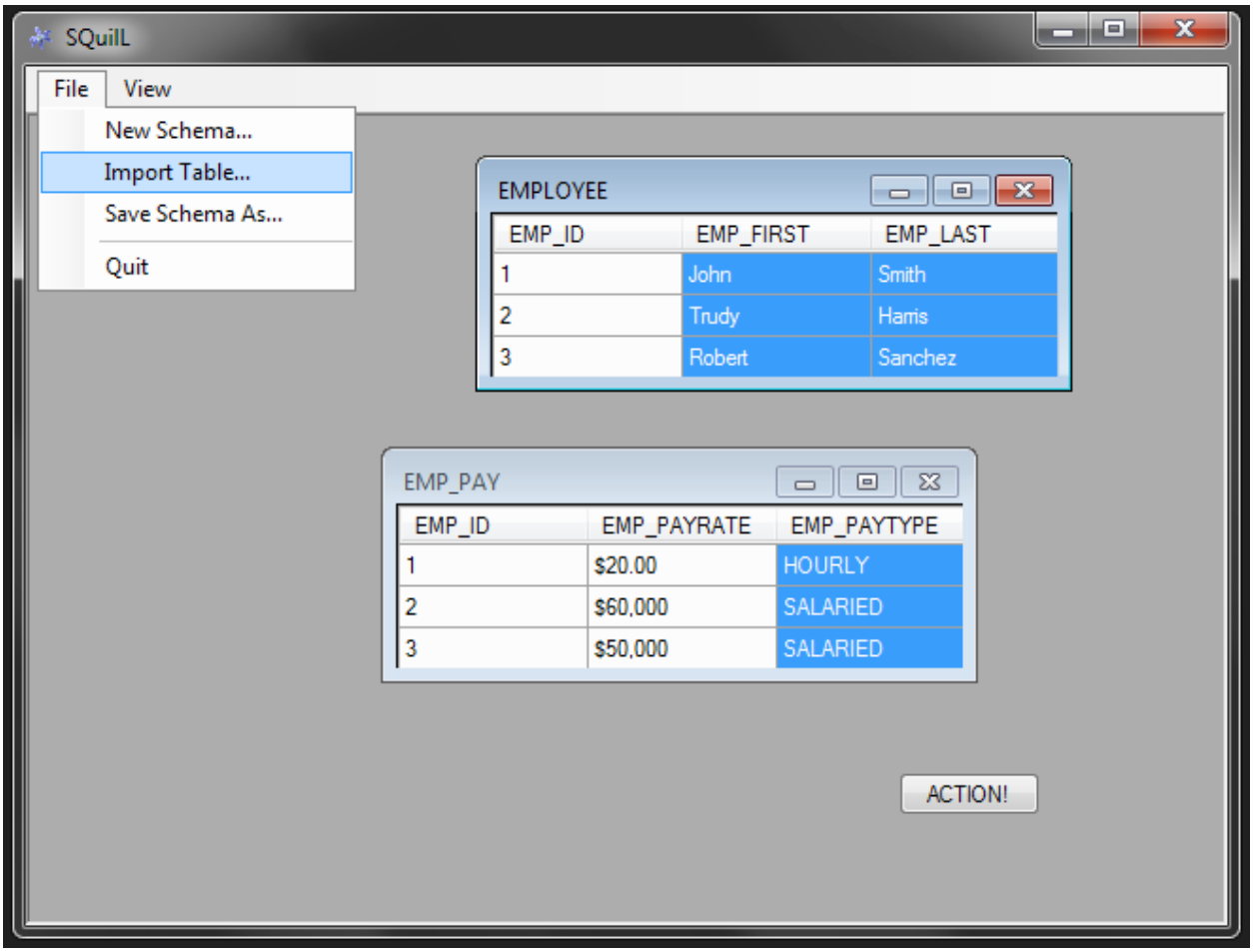

Figure 1. In the preliminary screenshot above one can see a very basic and minimal example of the visualization. The database schema contains 2 tables, one representing employees and another representing employee pay information.

Project Scope

With the exception of the goal of connecting to and querying live SQL databases, SQuilL will be self-contained and monolithic and will pose challenges of logic rather than advanced technical programming. Given those facts and taking into account the project goals, the project budget will be only that of time since no additional resources will need to be purchased for

SQuilL's development. The author is in possession of all the resources needed for the project: A suitable desktop computer, a suitable laptop computer (as needed), a suitable Integrated Development Environment (Microsoft Visual Studio 2012), a reliable internet connection to facilitate research within the author's home (where the majority of development will take place), and various hard-copy and electronic textbooks such as Coronel, Morris, and Robb's *Database Systems*.

Approach and Methodology

The programmatic approach used to develop SQuilL will be a window-based graphical user interface and will utilize object-oriented programming and design patterns programmatically to develop. Frequent testing and debugging will be accomplished during the course of development and most of the challenges are expected to be abstract challenges of logic and challenges in fine-tuning the user interface – ensuring that windows "fit," are updated upon changes, etc. As of the time of this writing a basic graphical framework (as seen in the mock-up shown earlier) has been developed as a skeleton to support core and optional features. The first and primary goal will be the implementation of an SQL statement parser along with a mechanism to highlight the data requested from the tables – both of which are core functionality and hard requirements. Steps taken to attain the first principle goal will be achieved by adding a text area to the application GUI in which the user can enter SQL statements. Fortunately, Java and C# both support programmatic SQL statement validation such that connection to a database is not required to validate SQL statements. The next step will be to link the variables (tables, columns, etc.) in the parsed SQL statements to their corresponding GUI elements. The following step will be to add logic to the GUI elements to cause them to respond graphically (highlighting of the proper row, columns, cells, etc.) to user operations. The final step, which is non-trivial, will be to

implement the rules of SQL and the addition of primary and foreign keys (fields which aid in describing how tables are related to other tables) as well as validating complex statements such as compound statements, JOIN statements (which are used to combine data from one or more tables), etc. Once this core functionality is achieved, it will be tested thoroughly before additional features will be added. Additional features which aren't part of the core functionality mentioned here will be implemented one-by-one as time allows.

Ethical Concerns

SQuilL is a primarily educational tool and is meant to be a supplementary visual aid to instruction rather than instruction itself. Still, the most obvious ethical concern is that it could be used improperly for cheating during interviews, exams, or assignments which require unassisted ability to work effectively with SQL. SQuilL itself would be a benign software application which would be released as Free and Open Source and without the ability to collect data of its users or its users' hardware. A repressive governmental regime could, say, not have a decent database architect but still wish to implement a database which could be used for repressive or otherwise malevolent purposes – and so the regime's members responsible for constructing the database may refer to SQuilL as an aid to construct the database to aid in unscrupulous behavior. However, the scenario is so farfetched that SQuilL could be considered a tool of repression just as any benign object could. The positive ethical aspects of SQuilL vastly outweigh the negative.

Social Justice

SQuilL serves to advance social justice by providing an interactive visual aid to learning SQL statements. Since SQuilL will be released as Free and Open Source software anybody is welcome to download, use, and modify it. Unlike other educational tools there is no cost barrier to download and use it, which will help poor and under served students so long as they have an

access to a computer. Since SQuilL is visual in nature, it will provide an advantage to students who learn visually rather than in terms of code and statements, and so will be an attractive tool for not only visual learners but people with learning disabilities who express interest in computer science. Since it is free software it could be trivially modified to support multiple languages and locales, extending its benefits to those who understand any written language. In short, SQuilL advances Social Justice by providing a tool freely available and for anybody to use to provide a learning experience for those who would not otherwise have educational tools to learn the ubiquitous SQL, ensuring that the less privileged and wealthy can stay competitive in an increasingly global market.

Legal Considerations

No legal challenges are expected to arise during the development and deployment of the software. The application will be released as Free and Open Source Software and implemented using only standard libraries of the language in which it is being developed. The author does not consider it his place to enforce violations of the software license in which users are bound in this case.

Risks and Dependencies

The primary risk to project completion is the academic and professional workload of the author other than that of the class in which the project is being developed, which is why the final deliverable of the project is being proposed as a simple and standalone desktop application which can be developed with the aid of literature and tools already possessed by the author. The author also claimed that extended functionality will be implemented "as time allows" due to foresight of academic and professional workload outside of the context of the project. Risk to core functionality is believed by the author to be minimal if not non-existent.

Usability Evaluation

One of the core beliefs of the author regarding usability is to keep the user-interface minimal and instead use external documentation and tooltips (tooltips are sections of text which appear when hovering over a component with the mouse cursor, they are typically used to explain the functionality of a component) as a real-time reference. Fonts will be of a small or medium size (14 or 16 pt, for example), buttons will be large enough to click comfortably and will be few in number, and there will be verbal descriptions rather than potentially cryptic icons to indicate functionality. There will be no pictorial decorations other than perhaps an introductory "splash-screen" and/or a background with a neutral color-gradient. Rather than cluttering the user interface with numerous buttons and fields, extended functionality will be implemented unobtrusively using menu bars and right-click menus. Usability-testing will not, in the author's opinion, be required due to the program simplicity.

The Author and Development

The author, Nicholas Fullerton, will assume all duties in the design, development, testing, and documentation of the application. At first glance it may appear that the proposed tasks are too numerous and unwieldy to be completed by only one person, however, the author believes that assuming total ownership of the project is the best option due to personal and professional circumstances which make teamwork needlessly difficult and stressful. The author's justification of working alone is that, due to personal and professional circumstances outside the project, the overhead group work introduces (unproductive delays due to lack of coordination as a result of differences in personal and professional lifestyles, communication overhead, differences in style, unexpected availability-limiting life changes of potential team members over the course of development, various other uncertainties and risks introduced with teammates, etc.) is believed

to negate any possible potential benefits of group work and in fact introduce unnecessary personal stress as well as the introduction of overhead and lack of continuity.

Conclusions Summary: Project Progress at Course Completion

The purpose of this section and of the format and content of following conclusion sections is to indicate, *in summa*, the actual progress and lessons learned from the SQuilL design, development, and documentation upon completion of the project and to facilitate the comparison and contrast of the lessons and processes learned from those described in the preliminary proposal. Not all of the previous sections will have their own dedicated follow-up sections due to lack of significant changes regarding the suppositions and/or ideologies between the time they were proposed and the time of completion of the SQuilL project. The few conclusion sections which follow could be considered the primary “lessons learned” during the process of SQuilL's development.

Conclusions: Ideological and Technical Goals, Revisited

Several of the project's ideological goals were compromised for pragmatic reasons, especially the limited amount of time the author had to develop the project. SQuilL's source code and other necessary files are stored in a public GitHub repository (located at the URL https://github.com/nfullertonCST/SQuilL) that anybody with an internet connection may access. The project is released under the Open-Source MIT license, which allows people to use the project as they wish provided that they include all applicable copyrights and licenses in derivative works and do not hold the author liable. SQuilL was also successfully made a relatively small and self-contained application, however, users with older hardware may notice a graphical slowdown when manipulating tables graphically. Locale/Culture functionality was also not implemented, though it could be done trivially with enough person-hours dedicated. Perhaps

one of the largest obstacles to SQuilL being completely "free" is that, although the project is open-source, it must be compiled and run on an actual or virtualized instance of Microsoft Windows due to it being written in C# (a language developed and mostly controlled by Microsoft). Although there is at least one framework to allow cross-platform development of C# (specifically, the Mono Project), the feasibility of cross-platform development not tested by the author due to the reputation of cross-platform C# frameworks of being incomplete or unworkable given the time allotted to develop the project. Although there were pragmatic reasons that several of SQuilL's ideological goals were compromised, there are no reasons why they cannot be fulfilled eventually given enough resources – specifically the amount of person-hours dedicated to porting, refactoring, and translating SQuilL.

Within the preliminary discussion of the project technical goals was a disclaimer that not all of the features suggested would actually be implemented due to time constraints. C# was the language eventually chosen to develop the project as well as the primary reason for the compromise of some of the project's ideological goals — the choice of the proprietary C# was one of rapid and stable development in favor of compromising freedom in using proprietary technologies. The functionality of saving and loading files as tables and vise-versa, of clean representation of tables as graphical objects, and the traditional representation of SQL query results as table-like objects themselves was all implemented. The most glaring omissions of suggested features were the bidirectionality of SQL queries and the validation of SQL queries before their processing — in the case of the former, the implementation of the generation of SQL statements from highlighting portions of the graphical Table objects was deemed too complex to implement in a timely manner; and in the case of the latter, it was discovered that the automatic validation of SQL statements in C# was available in only a specific version of Visual Studio

unavailable to the author – which necessitated a directed, button-controlled approach (rather than the user directly typing text) to minimize undesired behavior and serving as an unexpected "feature" of training a user to the specific sequence of keywords entered. Another notable feature missing (but relatively trivial to implement given enough time) is the ability to use SQuiL's functionality to manipulate an actual SQL database. As with the project ideological goals, the author is confident that the unrealized technical goals may also be implemented eventually given moderate levels of programming experience and person-hours dedicated to the project.

Conclusions: Approach and Methodology, Revisited

The general approach to SQuilL's development, a window-based approach representing SQL tables as child windows being manipulated within a large parent window, remained unchanged throughout the course of development. The author had anticipated early in development academic and professional obligations which would detract from the development of SQuilL and did indeed compromise the approach and methodology. From the perspective of approach and methodology, it was not temporally feasible to use generally-accepted design and software-engineering principles. Although SQuilL uses an object-oriented approach, there are still multiple "monolithic" code files which are hundreds of lines long. Convention dictates that, to be readable and usable, larger "Classes" (each code file typically represents a "Class" although there are exceptions such as interfaces) be broken up into as many smaller Classes(code files) as possible. Similarly, operations in each Class should be broken up into as many subtasks (functions/methods) as possible within each code file. The subdivision of classes and functions/methods into smaller and simpler units is to facilitate code clarity and re-use — The smaller a Class is, the more easier it is to read; the more Classes one has available, the more each class can be reused elsewhere; and shorter and simpler functions/methods are more easily

understood than longer and more complicated ones. Design Patterns, standardized ways to write code to satisfy more established roles, were neglected due to temporal constraints (though it is possible that some design patterns were implemented unconsciously by the author or implicitly due to the way Visual Studio represents objects and operations while inheritance, though not explicitly used, is used implicitly with a maximum inheritance of seven generations according to code analysis tools). Testing and debugging were implemented according to the preliminary methodology, that is, to test and debug each feature as it is developed — not the most professional method, as it neglects proper testing using edge-cases, but it is relatively efficient building features piecewise and under time constraints — In the author's opinion, it is better to have few features which are functional rather than many features which aren't. Unfortunately, one of the two features considered "hard requirements" for the project, the highlighting of existing Table objects to represent queries, was not implemented in the final (to date, at least) release although the functionality is in the code and couple be implemented in the final release a week or two later than the date of this paper's writing.

What follows is a description, in greater detail, of how exactly the SQL statements are entered by the user. The functionality to allow the user to enter their own statements using their keyboard, while functional in early iterations of the project but eventually disabled in later versions, was eschewed in favor of a guided approach allowing the user to enter SQL keywords and fields/tables by clicking buttons and selecting entries in comboboxes (known informally as "dropdowns"); respectively. As an example, if a user wants to view a column called "COL_1" in a table named "TABLE_A", they would enter the SQL statement "SELECT COL_1 FROM TABLE_A;" by first clicking the "SELECT" button, then selecting "COL_1" from the "fields" dropdown, then press the "FROM" button, then selecting the table "TABLE_A" from the "tables"

dropdown. The statement is shown in a text box as it is formulated by the user. After the statement is formulated, the user clicks the "Execute" button which adds the final semicolon marking the end of the statement. The buttons are enabled and disabled selectively according to the statement so as to minimize errors and allow the user to practice the proper placement of SQL keywords.

After the user enters a statement and signals the application to execute the statement, what happens next is that the statement text is tokenized—that is, it uses a Tokenizer class to strip each token (linguistic unit, or informally, "word") of non-alphanumeric characters such as commas and whitespace and put the alphanumeric "words" into a list. The list of processed tokens is passed to another class named SQLWorker, which could be considered the actual parser of the SQL statement. The instance of the SQLWorker then scans the list of incoming tokens and further divides the tokens into smaller lists. In the example statement above, all tokens between "SELECT" and "FROM" are processed into a list which holds the fields to be selected. All tokens between "FROM" and the ending semicolon are processed into a list which holds the tables involved in the query. If the statement contains a JOIN, then the fields are loaded into the sublist of fields, the tables are loaded into the sub-list of tables, and then the third major set of data — called the "keys" — are loaded into a third sublist of keys. In the case of simple SELECT statements the SQLWorker simply iterates across the lists of fields and tables and marks (highlights) the listed fields in the tables by using a member function of the Table class used to mark columns. For JOIN statements the operations are more complex, especially so because 4 different types of JOIN statements are supported: natural ("old-style") join, right join, left join, and full (outer) join. For clarity only the natural join implementation will be described. When a natural join statement is recognized, the SQLWorker has 3 sets of data it uses to process the

JOIN: a list of fields, a list of tables, and a list of keys. A "key" in this case is a field that relates one table to another in the sense that a row in one table has a value in its "key" column which matches the value in the "key" column of another row in the other table. When a row in a table has a key which matches the key of another row in the other table, the two "matched" rows in each table are marked. The JOIN operation can further restrict what is selected based on the type of JOIN and the columns named in the list.

The use of lists is as extensive as possible to facilitate ease of implementation. To carry out an operation, major items are placed into lists, and the lists are iterated across. If an item in the list (word, Table, Columns, etc.) meets the criteria to be checked (table name, cell value, etc.) the operation on the checked object is performed. Lists in this case are generic data structures which support a wide variety of existing operations. Lists, along with other Object and operations in the C# standard library, minimize the need to "reinvent the wheel" with respect to software development.

Conclusions: Usability Evaluation, Revisited

The usability evaluation was neglected initially because the author (very incorrectly) believed that a usability evaluation was unnecessary due to the amount of graphical control being not only simple and uncluttered but out of the user's control. Upon development, however, it became readily apparent how usability factors into even the most deceptively simple operation. Earlier the author noted that the ability for users to directly type their own SQL statements was removed, and one of the reasons for that functionality removal was that it would have created profound problems with usability given the amount of undesired behavior which would result from malformed statements or grammatically correct statements attempting to use unimplemented functionality. The solution to the problem was to change the approach to a

"guided" statement entry allowing the user to enter only supported statements and loaded tables and fields by clicking buttons and selecting from dropdowns — which proved to be a usability problem in itself: What would be the placement of buttons and dropdowns to allow the best compromise? The compromise was to place all the buttons in an "L" shape on the left side of the area, and to place the dropdowns in the gap left by the "L" shape. Although the clean layout was maintained, there was a bug in which loaded tables would appear "covered" by the data entry area, which was very detrimental to the usability of the application. Some slowdown was noticeable when Tables were being dragged throughout the usable area of the application when tested on the author's slightly dated (circa-2005) but still reasonably responsive desktop computer. The author enlisted the assistance of his relative, who has some experience with statistical programming but is mostly non-technical, to perform usability testing. The primary criticism of the tester was that of project incompleteness as an educational tool. Since SQuilL lacks a built-in help page or other tutorial information (other than button tooltips), it is in its current state useful as a supplementary learning tool but not as a standalone learning tool. The author, in response, plans to implement a full-set of graphical help pages as well as a flowchart to address the problem. Additionally, when a new query was processed, the query entry area of the application hides itself so that the new query (Table) can be shown. The behavior was a response to an unfixed bug, but the tester remarked that the behavior is confusing since it is not mentioned in a tutorial. The tester did like that the application supported file operations and was small and standalone in nature.

Conclusions: Parting Words

At the time of this writing, the Author is in agreement with his usability tester in that the project is still in an incomplete state, but functional, and with tutorial functionality implemented

could serve as a lightweight educational substitute for a SQL database able to be used without the computationally-expensive overhead of a proper SQL database or a network connection. The project's primary goals should be updated to reflect its purpose as a lightweight database simulator rather than merely a novel way to highlight tables and visualize queries – although since a workable highlighting feature has been coded, it should be integrated into the project along with extensive self-contained tutorial functionality. Finally, the importance of the development of SQuilL to the author was a valuable learning experience in itself. The lessons learned will serve the author throughout his career.

References

Oracle Corp. (2016). MySQL Workbench: Visual Database Design. Retrieved April 17, 2016, from https://www.mysql.com/products/workbench/design/

RJMetrics. (n.d.). SQL Joins Explained. Retrieved April 17, 2016, from http://www.sql-join.com/

Vasilev, A. (2015, January 05). SQL Joins Visualizer. Retrieved April 17, 2016, from http://sqljoins.leopard.in.ua/