SIQL: Simple Interactive Query Language

Dept. of CIS - Senior Design 2012-2013

Daniel Trujillo
dtru@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Travis McKenzie
travism@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Gary Menezes
gmenezes@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Edward Funger
funger@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Susan Davidson
susan@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT

The purpose of this document is two-fold. First, it describes a new, conceptually simplified, interface to facilitate database management. Then, it explores that application’s architecture, implementation, criteria for evaluation, and future items for consideration.

This simplified interface provides a method for non-database-savvy users to manage simple, yet practical databases that can serve various purposes. Use cases range from a graduate biology student maintaining a database of the sources for their thesis, to a store owner who needs to maintain a database to manage an archive of retail orders.

1. INTRODUCTION

Database creation and management are commonly tasks fit only for database administrators in large scale systems. However, in small scale systems, this does not necessary need to be the case. In such smaller scale systems, a user may need to use only 10% of the features of an industry standard database management system (DBMS) to design their database. Does that user, who we will assume is not a database administrator, want to see the additional 90% of unnecessary options? This pattern is common, whereby a non-database-savvy user has a real need for a simple database. For example, a doctor may need to access a medical database to gain basic information about a patient. For users not experienced with database management principles and common techniques in practice, the process of creating and managing a database can be a nightmare.

The most common database type in practice is the relational database, which means that relational database management systems (RDBMS) require a special syntax to effectively communicate with a database. This syntax is a Structured Query Language, or SQL. SQL is highly structured as the name implies, and queries must be carefully crafted by the user to ensure that no syntax errors are committed. SIQL on the other hand, provides two crucial differences from direct SQL. Primarily, simplicity of interface and options allows for a short learning curve. Additionally, the interactive component of SIQL means that the application guides the user through a process of database creation, configuration, and later query generation. The expectations that are normally placed on the user - data types, relations, integrity constraints - are now placed on the software.

SIQL is, in more general terms, a web application that facilitates the creation and management of database systems for users that fit the non-database-savvy user group. SIQL simplifies relational database management by enabling database design and creation through a drag and drop graphical user interface (GUI), rather than via a query language such as SQL. SIQL further simplifies interaction with database after it has already been created by converting English text into proper SQL queries for a limited set of queries that can be run on the database.

SIQL provides a starting point to create a more robust DBMS hosted in the cloud for non-database-savvy users. Considerable success has been achieved for creating simple database schemas quickly that additionally generate valid SQL syntax. Single table conjunctive normal form queries have a high rate of success in translation to direct SQL without asking the user for clarification. With these techniques this system can be readily expanded to address more complex database queries.

2. BACKGROUND

A relational database is a database that conforms to the relational data model proposed by Codd in 1970 [6]. Relational databases are operated upon using Codd’s relational algebra and tuple relational calculus, founded in first order logic. The mathematical nature of manipulating the relational model allows some of the syntactical complexity to be abstracted away in database management systems. Codd’s visualization of relations as row-column tables has continued to be the most common implementation in database management systems. For non-trivial data sets, this causes difficulty in designing effective relations, writing efficient queries, and understanding pre-existing databases enough to query them for complex relations (closures).

Several applications have been researched and designed over the years to simplify the process of creating appropriate data relations and constructing efficient, powerful queries. In many ways these applications have helped users to visualize existing data, construct complex queries without using strict syntax, and create a layer of abstraction that provides added safety to prevent data loss or corruption. However, the focus of these applications has been largely in database management rather than database development. This means that these applications generally require that someone familiar with query language syntax has already painstakingly configured the database properly to constrain data fields and set up data relations between those fields.

The common industry standard applications often lack simplicity and elegance. This is not an issue if a user is familiar enough with
an application. However, one might question how much effort they can save by learning a complex application to avoid writing query language syntax. While the second is much more involved, for the majority of queries only simple syntax is required to get the job done.

Additionally, industry applications can be overwhelming for users who are not database savvy for two more reasons. First, they often require non-trivial set up from the user, such as configuring a local webserver or MySQL database instance. Second, these applications are not always free to use. Generally the applications that are free to use are the ones that lack advanced debugging tools and require substantial database experience to be used correctly.

SIQL aims to address many of the issues that prevent non-database-savvy users from successfully managing their own databases. In so doing, compromises in functionality are necessary to preserve simplicity and ease of use. Audience dictates the entire workflow of SIQL, and form follows function. Thus simplicity and efficiency are the two main evaluation criteria.

3. RELATED WORK

There are numerous graphical user interfaces (GUIs) designed to manage databases, each with its own benefits and limitations. Before attempting to contribute to the field of database management via an interactive GUI, it has been critical to first consider the progression of the applications currently available. Different design goals drive different development processes, but there are still commonalities and trends apparent in industry. While current industry standards lack a niche for less technical users, the techniques and principles employed in those applications have been highly influential in the development of SIQL.

3.1 Early Implementations

The most popular database administration tool today is almost certainly phpMyAdmin [17]. This tool provides a web interface as its GUI, allowing a user to manually enter SQL commands or generate basic commands on existing database tables. A user must be aware of all of the settings that they wish to use in their database, but they can choose these settings from lists of options rather than strictly writing SQL. SQL queries are generated as a result of any action taken in phpMyAdmin, so a user can get an intuition for SQL syntax without actually writing SQL. Additionally, imports are allowed from SQL or comma-separated values (CSV) files, and various export methods are also supported. In this way, phpMyAdmin provides a good basis of the rudimentary functionality that is necessary to properly manage a database.

QBE, or Query By Example, was the first graphical query language developed for relational databases. Initially this language was designed solely for accessing existing databases, but this has changed over time. Today, generalized QBE (GQBE) projects have drawn ideas from QBE and incorporated user-friendly GUIs [13]. However, these implementations work on top of existing databases and lack the ability to generate arbitrary new databases and data relations. Even so, QBE provides important insights as to how relatively simple natural language processing can eliminate the need for SQL syntax.

3.2 Increasing Graphical Focus

Various graphically-based applications have expanded upon the ideas found in the applications mentioned above. One implementation, WWW SQL Designer, provides a simple, low-level abstraction that relies entirely on foreign keys [24]. A user can create data relations by dragging and dropping visual representations of database tables which are all connected by foreign keys. While this can effectively model many simple data relations, the scope of this application is limited by its simplicity. For example, with hundreds of connections between data tables, the visual representation of the database includes all of these connections together on the screen. Thus it is easy to see that this is not a scalable solution as the number of connections between data fields increases. However, joining tables is a necessary function for database management. To that end, SIQL adopts a simplified join that simply links two tables rather than worrying about foreign keys, a concept of which non-database-savvy users are normally unfamiliar. Additionally, the SIQL GUI only displays the aspects of the database schema that the user cares about at the time, rather than constantly displaying the entire schema diagram.

3.3 Full-Featured IDEs

Currently the state-of-the-art interfaces for SQL management act as interactive development environments (IDEs) which allow users to generate SQL and refine that SQL manually. This requires users to attain only basic competency and allows them to accomplish advanced tasks relatively efficiently. The most notable open-source example is MySQL Workbench [8]. One initial drawback is that this application is only designed for MySQL, though migration from other major SQL database types, such as Microsoft SQL Server [16], is supported. However, this still means that a user must work exclusively with MySQL during database development, and that they can only export databases that use proprietary MySQL syntax. It would be better if there were a way to use an arbitrary SQL implementation, rather than exclusively supporting MySQL syntax and offering no substitute such as PostgreSQL [12]. MySQL Workbench functions as both an IDE complete with syntax highlighting and SQL execution history, as well as a visual tool for demonstrating data relations. A database agnostic counterpart to MySQL Workbench is FlySpeed SQL Query [22]. FlySpeed SQL Query allows drag and drop capabilities to generate queries, as well as other advanced features, but its focus is data processing rather than database design and creation. While this is valuable for users who inherit databases that they must access, there exists an implicit assumption that a well-versed database administrator configured the database for that end user.

The IDEs mentioned above address a few important concepts used in SIQL. Both syntax highlighting and execution history can be seamlessly integrated with the natural language processing module. MySQL is the current supported database type as it has the most IDE support, and translations to different query languages can be added in the future through SIQL’s abstracted interface. Additionally, visualization has become much more interactive in the
newest versions of these IDEs. The IDE approach works very well for technical users and the applications mentioned above have a solid following, but for non-database-savvy users it is more worthwhile to abstract away some of the lower level details and impose logical design decisions on the user without bombarding them with advanced options for simple operations.

3.4 Recent Improvements

One new product that goes beyond the current IDE approach is Microgen DBClarity Developer [15]. This product is database agnostic, allowing the user to develop database logic for an arbitrary SQL implementation. Key features include a proprietary method of representing database logic through an interactive GUI, full-feature database schema diagrams that give the user control over more than just foreign keys, a debugging system, and modularization of database logic for use in multiple projects. While all of these features provide the user convenience and efficiency, there still exist basic assumptions about the user’s general familiarity with programming. The native syntax of the application is relatively simplistic in nature, but it is arguable that a less technical, more human-readable language could be developed to describe database components. Furthermore, it is also possible that a method more intuitive than a 50-button GUI could readily be developed to describe database logic. Therefore, SQL attempts to translate from natural English text to eliminate the need for a complex GUI for data access. This, along with a concise GUI for database creation, significantly reduces the overall application complexity.

So far we have seen that the current state-of-the-art applications for database management via a GUI have converged upon a single trend [Figure 1]. This trend results in complex interfaces that hide native SQL syntax from the user, which is the ultimate goal. However, the compromise of abstracting SQL syntax comes in the form of a new syntax that is not much simpler, a complex arrangement of buttons providing specific functionality, or some combination of the two. The SIQL interface is designed to be both more intuitive to the end user and at least as effective for that end user to perform database operations of complexity without prior knowledge of native SQL syntax.

4. SYSTEM MODEL

SIQL’s goal is to develop a graphical query language designed to help non-database-savvy users that is implemented and hosted on a third-party server to avoid set-up time. This web application helps non-database-savvy users create and interact with a database, without having to understand the complexities of SQL. With the success of graphical programming languages like Scratch [2], it is evident that there is a need for simple tools that do not involve writing code. With SIQL, it is anticipated that valuable programmer time will not be wasted writing SQL queries for non-database-savvy users in a collaborative environment. The application consists of four essential pieces [Figure 2]. All of the user interaction will take place in the GUI. Once the user creates an account, the Rails application begins instantiating a new RDS server specifically for the user. Once the instance is ready, the user can then edit the schema and add entries using the Database Creation GUI. To retrieve data from SIQL, the user enters a natural language sentence, and the NLP module will convert it to proper SQL syntax.

4.1 Database Creation GUI

As stated previously one of the main goals is to simplify relational database management, making the Graphical User Interface an imperative portion to provide an intuitive interface for users to utilize in the Database Creation process. The user interface model is much like an Entity Relationship (ER) Model [Figure 3]. An ER Model is a general abstraction of a database that displays very little granular detail, but establishes the data included in the database. To accomplish this ER Model-esque graphical interface tables are represented by blue square, attributes are represented by orange squares and relations are represented by green squares. These figures are connected by a linking line, attributes are connected to their tables by a thin black line and relations are connected to tables by a thicker red line to differentiate relation connections to attribute connections within the interface.

The interface provides the user the ability to have n-ary connections to other tables through a single relation with the assumption that the relation will follow a many-to-many or one-to-one relationship. This assumption is made due to the simplicity of the interface, where adding more complex relationships of one-to-many or many-to-one might confuse the user and thus defeat the intuitive nature of the interface. This interface would behave much like any other drag and drop graphical application where tables and entities are dragged and moved around the graphics box.
Another implementation includes the ability to zoom in and out and pan the diagram up, down, left and right. Using the mouse scroll wheel the user is able to scale their diagram to a desired size and using the arrow keys the user can move the diagram in any directionality to fit the current graph view. This grants the user the ability to properly scale for larger diagram with the ability macro-manage by viewing the entire diagram as a whole or micromanage by focusing on a specific part of the diagram.

Predefined schemas also allow the user the ability to choose from a list of schemas for their applications. This allows the users to have a starting point for their own use cases to modify and expand as they see fit.

In addition to the graphical representation of the tables and attributes, the interface represents each element’s data in textual form below the graphics. This data is shown when selecting a particular element of the graphical interface. Tables display the table name and its attributes, attributes display the attribute name and its datatype (text, date, number), and relations will display the relation name and the tables connected to it. Certain database restrictions are also implemented into the UI, mainly no two tables should have the same name and no two attributes in the same table should have the same name and datatype. This provides a more intuitive interface for users to utilize when creating a database.

Lastly the interface provides a Finalize function before creation on Amazon RDS [5] and redirecting to the Database Manipulation stage. This function is essential to further define the diagram’s schema without burdening the user with too much database knowledge. First, primary keys are assigned to every table. If the user is unsure of the primary key for their table, SIQL generates a system key for the user. Second, the user is asked which attributes are required (not null) and which must only contain unique values. Third, the user finalizes the relationships between tables. These relationships are n-ary and can be many-to-many or one-to-one. This step in particular requires more effort from the user than the previous finalization steps, so examples are given to help illustrate the important database design concepts for which the user must make decisions. These examples are in the form of common database schemas that exhibit the possible relationship types. For example, a book and author table represent a many-to-many relationship since a book can have many authors, and an author can write many books.

4.2 Natural Language Processing Module

Once a user has a database built and populated, they will need to query that database. In keeping with SIQL’s goal of simplicity and ease of use, the NLP (Natural Language Processing) module enables syntax-free queries. The user enters a query in natural language English and, after a simple validation process, they are given the results of their query.

The NLP module takes as input a JSON representation of the database schema and an English language query, and returns a SQL sentence. The conversion from English to SQL is a three step process. The following explanation will use the query “Students with gpa over 3.0” as an example input.

In the first step, the input query string is parsed into a tree structure which implicitly describes the structure of the input sentence and the logical dependencies between words and phrases [Figure 4].

The second step is a custom heuristic that parses the dependency tree into a list of ‘segments’ [Figure 5]. These segments are phrases or partial phrases that correspond to segments of the resulting SQL sentence.

The third step in this algorithm is to take the list of segments and, in a generally 1-to-1 mapping, convert it to a proper SQL sentence.

Once this complete SQL syntax has been computed, it is presented to the user in an editable form. In this final validation step, the user fixes any errors and submits the query. The resulting table of data is then retrieved and presented.

4.3 Amazon RDS

Amazon’s AWS provides SIQL with all of its data services. The Amazon Relational Database Service (RDS) [21] allows an AWS user to spin up and manage a Mysql database. It stores all of the users’ databases as separate database server instances. Because each instance lives inside of its own virtual machine, it is protected from other users’ instances. The risk from cross contamination is eliminated in this model and if a user’s account information or instance do become compromised, it will not affect the other users. Amazon RDS also provides a gem that wraps its web API in Ruby. Amazon AWS also provides the hosting solution for SIQL with the EC2 service. EC2 allows the application to scale indefinitely as well as allows the system to rely on Amazon’s content distribution network (CDN). The CDN creates more reliable load times by serving the web data from a geographically close location to the end-user.

4.4 Ruby on Rails Server

The Ruby on Rails [1] application is the exchange for all transactions that are happening. It facilitates communication between the front-end GUI and Amazon services. Rails was chosen for its extendability. Because the framework is so popular, there are many libraries, called gems, that add functionality to the basic features. With the use of Rails to connect the user with a database, there is no need to install software on the user’s local computer, thus simplifying the process greatly.

5. SYSTEM IMPLEMENTATION

The following section describes the implementation of the model described in section 3. The libraries and paradigms used to implement the functionality of the GUI, NLP, and back-end, will be discussed.
5.1 Back-end Implementation

The back-end of SIQL is implemented in the Ruby programming language and different web APIs, using the Ruby on Rails MVC framework. Rails was chosen because of its speed in prototyping web applications as well as its robustness in a production environment. The Ruby on Rails application, is hosted on Amazon’s EC2 [4]. Ruby on Rails is an open source, full stack web development framework that implements the Model, View, Controller architecture pattern. The application keeps track of the user accounts, as well as the instances that are currently spun up in RDS. This is done by creating a model that reflects the state of the RDS account, and uses lifecycle callback functions to make sure that they are correctly synced. For this to be able to work, two gems were added. To the Server model, the aws-sdk [21] gem allows the Rails to control the initialization of a MySQL server on RDS. When the resources are created, a call through the gem to the Amazon API is made. Because of ActiveRecord’s transaction model, and because the API call is made in a lifecycle callback, if the API call fails and is not able to create the database for any reason, ActiveRecord will rollback the entire transaction. This mechanism maintains synchronization between the back-end and Amazon services. A similar technique is used in the Instance model, but the Mysql2 [14] gem is used to communicate with the live instance through the endpoint that is created once the database is spun up.

Methods were also added to the Server and Instance model that allows the application to know their status, such as if they are spinning up, being destroyed, or ready. These also use gem methods, but instead of being implemented as lifecycle functions, they were created as class methods so that they can be accessed outside the scope of the object.

To allow the application to communicate with the front end, a RESTful web API is exposed. Using this API, the front end is able to create and edit database schemas, as well as execute any query that is in the SQL domain. To access the API, a POST request is sent to

```
/user/:u_id/server/:s_id/instance/:i_id/query
```

Because the GUI is written in Javascript, the API is designed to return the data in the form of JSON. The GUI can also send multiple SQL statements at a time, and the back-end will process them and execute them individually. This allows data to be inserted into the database using a comma separated value sheet or manually filling out a web form.

For application-side security, the Devise [18] gem was added. Currently, the salt-hash method of storing passwords is being used to prevent any leakage that could compromise an account, but could be extended to use OAuth, which uses tokens to authenticate the user. On Amazon’s side, there are many security features in place. To communicate with the API, the amazon-aws gem implements message signing which hashes the request with the SHA256 encryption method. A public and private key for encrypting and decrypting is provided by Amazon. Also, all of the individual user Servers are secured with a randomly generated password. Even if an account were to be compromised, the sandboxing of Servers prevents one account from effecting other users’ data.

The entire round trip from client browser and back is represented by the figure below. The upper right loop shows all the work done by the aws-sdk [21] gem. The bottom right loop is represented by the mysql2 [14] and the left most loop is the communication done over the application API.

5.2 Database Creation GUI Implementation

In respect to web technologies for SIQL, with the provided constraints, HTML5 and Javascript provided the best technological solution. Specifically the implementation utilizes HTML5 Canvas feature in conjunction with a Javascript graphics library called KineticJS [20] that provides some base graphical elements that help create the desired User Interface.

The GUI initially creates a Kinetic.Stage object that allows it to hold a container of the figures the users create. Additionally a Viewport object is created to help calculate the proper scaling and translation of the diagram based on the user input. This viewport acts like a camera view and creates relative positioning for graphical objects. The viewport is initialized with zoom scale 1.0 and an x and y offset of 0 pixels. When the user moves up, down, left, or right the x and y offset positions for the viewport change and the entire schema diagram translates across the screen. When the user zooms in or out the scale changes from 1.0 in the range of 0.5 to 2.5 times the original zoom scale. Each time an object is dragged and moved the relative position of the object is recalculated based on the delta for the x and y positioning and the positioning and scale of the viewport.

As per the data from the model the interface stores Javascript Arrays for the Tables, Relations and Attributes of the schema. This data is consistently updated based on the modifications and additions of graphical elements to the user’s diagram. The interface also stores some base schemas as Javascript Object Notation (JSON) objects, a text based object that allows simpler parsing, so that users can load these preset diagrams for their own use case.

Lastly in the Finalize step of the diagram the interface converts the data that is gathered from the Arrays and the Viewport into a JSON object and submits a proper SQL query to the Amazon RDS [5] database. The JSON conversion calculates the absolute position of each data object represented in the graphical stage based on the relative position of the object to the viewport. Similarly to how the interface converts the data to JSON format, it later converts that data to an SQL schema by converting simple datatypes to their proper SQL types and systematically creating tables for each Table and Relation found in the Arrays. This newly created SQL schema is then sent to the Amazon RDS servers and the JSON object is then stored in the Rails server to later restore the database diagram in the Database Manipulation stage.

5.3 NLP Implementation

Since the queries input to the NLP module are to be executed over a specified database, the schema (table names, column names and datatypes) is available and significantly simplifies the problem space. Given this ability to directly match input to the database structure, this module is designed primarily as a heuristic based expert system. A test bank of approximately 100 natural language English queries was built and the following algorithm was developed by iteratively improving accuracy on the test bank. This explanation intermittently refers to the previous example query: ‘Students with gpa over 3.0’.
This module is implemented as a Java program using several external libraries. The input database schema JSON is parsed using the JSON.simple library. The database labels (table and column names) are stored in WordComparison objects in a mapping that maintains their relations. The WordComparison class stores a word’s text, stem, and lemma in order to ease future similarity calculations between words. In the first parsing step, the input query is converted to a dependency parse tree using the Stanford CoreNLP library. The next two parsing steps involve custom heuristics.

The second step is to discover ‘segments’ that correspond to SQL sentence components. In particular, these cases include standalone noun phrases that correspond to the columns and tables to SELECT FROM and more complex phrases that correspond to WHERE conditions.

The general idiom developed for identifying subtrees that correlate to WHERE conditions is to identify minimal noun and verb phrases that contain the part of speech elements corresponding to the average condition phrase. From the example query, the segment ‘with gpa’ is identified by matching a preposition-noun pair. This process is heuristic based but has been reduced to four cases that generalize well over the test bank.

In the third step in this algorithm, the list of segments is run through an in-order parse that generates an intermediate structure that closely resembles actual SQL syntax. This intermediate SqlQuery class contains the table being queried, a set of the columns being selected, and a set of the conditions over the query. Each segment generally maps to a single section of the SqlQuery object. Segment words are correlated with database schema elements by iterating through each word and matching with database labels and values. This matching process uses a weighted average of direct matching, stemming, lemmatization, and synonym word-similarity. Words stems are generated using the Porter Stemmer algorithm, and synonym distance is computed using the MIT Java WordNet library. If a lone noun matches a database label, it is added to the SELECT or FROM component of the SqlQuery object. To also find WHERE conditions, relational adjectives are matched by part of speech and values are matched by determining and comparing datatypes. In the example query, ‘over 3.0’ is parsed as a WHERE condition since it has a relational adjective and value.

After this initial process in the third step, the intermediate structure is usually redundant and only partially completed. For example, the ‘over 3.0’ segment maps to a WHERE condition that is missing a column name. To address this, the structure is run through a normalization process. This process determines and inserts missing database labels (in WHERE conditions) and adds LIMIT and ORDER BY commands to the resultant SQL sentence. Missing database labels are copied from the closest prior segment - the ‘over 3.0’ condition would have ‘gpa’ added as its column. LIMIT and ORDER BY commands are identified when normalized WHERE conditions have no value (e.g. ‘highest gpa’). The specific operators and ORDER directions for all conditions are translated using a custom dictionary of relational adjective roots (e.g. ‘high’, ‘low’, ‘great’, ‘less’).

The final SQL sentence is thus generated and returned.

6. SYSTEM PERFORMANCE

The following section will discuss how SQL performs in terms of the goals set forth in the introduction such as simplicity and efficiency, as well as objective performance like page load time, and correctness.

6.1 Application Performance

Many techniques were utilized to increase back-end application performance. In the production environment, precompiling all css and javascript assets reduced page load time from ~3 seconds down to ~500 ms. By utilizing the transaction model of ActiveRecord, modifications to the application database were grouped into single transactions. This helps read and write time because most of time spent is overhead. By wrapping multiple transactions in one transaction block, it becomes an n-time operation, whereas it was n² -time before. Instead of writing entire HTML pages as a response to the API, JSON is sent to the client, thus reducing the data transferred and speeding up the request.

6.2 Amazon Web Services Performance

A large bottleneck in the system will be the communication between the Rails application and Amazon RDS. Because the database is remote, the results of all queries must be transported using the HTTP protocol. Transfer over the internet is much slower than a read from disk on a local machine. Amazon has systems in place that combat this problem, such as using their content distribution network to deliver information from the closest geographical server, good database caching, and proportioned In/Out Operations per Second for consistent performance. Also, the largest operation in the entire user lifecycle takes place the moment they sign up when the database is initialized. It was found that this takes ~6 mins and is orders of magnitude longer than any other operation.

6.3 Database Creation GUI Performance

To test the performance of the GUI 40 participants were used to test out the interface and compare it to conventional methods of creating database. Participants were split into seven groups based on their knowledge of databases and their overall experience with computers and programming. Participants create a database with a Student that has a name, an id, and a gpa, a Teacher with an id, a name, and a relation to the student, and a Test with a score and a title with a relation to both the Student who takes it and the Teacher who grades it. A form asks the users to evaluate how long it took them to create the database using this interface and how long it would take to perform the same actions using conventional SQL tools including mysql or phpMyAdmin. Lastly the form asks the user to rate the ease of use of the interface.

The results found that users are able to create databases five times faster using this interface and have an average 8.1 Ease of Use rating for the interface. These results prove that the graphical approach to database creation is not only more efficient than alternative resources used in industry, but provides users with limited database experience a simpler interface to utilize.

6.4 NLP Performance

A large bottleneck in the system will be the communication between the Rails application and Amazon RDS. Because the database is remote, the results of all queries must be transported using the HTTP protocol. Transfer over the internet is much slower than a read from disk on a local machine. Amazon has systems in place that combat this problem, such as using their content distribution network to deliver information from the closest geographical server, good database caching, and proportioned In/Out Operations per Second for consistent performance. Also, the largest operation in the entire user lifecycle takes place the moment they sign up when the database is initialized. It was found that this takes ~6 mins and is orders of magnitude longer than any other operation.
As previously mentioned, the NLP module was tested and built against a test bank of approximately 100 queries. The module’s final accuracy was 71% over this set. This percentage represents the queries that were handled accurately - returning proper SQL syntax with the intended meaning. The tested correctness of this module, therefore, is slightly higher. A query that is ‘incorrect’ will still return some part of the intended SQL and thus allow the user to make corrections rather than write the query from scratch. The drawback of this testing and development method is that the algorithm is currently overfitting against the test bank - its actual accuracy is some unknown quantity lower. The current test bank, and therefore algorithm, leans towards conjunctive normal form sentences with database schema labels explicitly stated.

7. ETHICS

Because user data is being stored, there is an obligation to ensure security and reliability of the backend and front end. All data is sandboxed on Amazon RDS, so there is no possibility of contamination of the user data. In terms of application security, Devise is open source and rigorously tested, as well as using a very basic method to secure passwords. It was chosen because writing a similar authentication scheme would have been less safe. In Ruby, there is a feature called mass-assignment which allows the creation of an object by passing a hash to its constructor. To mitigate this concern, mass-assignment was set up to work only with a white-list of attributes that are defined in each model. Finally, there are the standard web attacks that may cause problems. There is a possibility that an account could be compromised with a social engineering attack such as phishing, or the entire application could be affected with a SQL injection or DDoS. These types of attacks are not particularly concerning because the user data is stored on Amazon, and is separated from the application data.

8. FUTURE WORK

The GUI implementation currently only supports Safari, Firefox, and Chromium. In a future iteration it should expand the browser support for Chrome on Mac and Windows and provide support for versions earlier than Internet Explorer 9 that do not support HTML5 canvas. Another feature for future iterations is expansion of database language support. The interface currently supports MySQL [7], future iterations should expand this support to other database languages like Oracle SQL [9] and NoSQL languages like MongoDB [3].

The accuracy results for the NLP module are colored by overfitting to the small test bank. The groundwork can be laid for significant improvements by developing a collection process for natural language English queries in order to build a larger and more representative test set. For the algorithm itself, the greatest room for improvement is in matching varying sentence formats. The NLP algorithm does not currently handle complex sentence structures with any level of success. Improvements can also be made in the process for finding database schema table / column labels in the query. For example, the current algorithm cannot match the word ‘1/13/2013’ with a ‘hired’ column of type DATE. This would be helped by adding tags such as ‘proper noun’ and ‘currency’ to the input database schema.

The back-end could be improved by utilizing more of the features available in the gems that are used. Currently, only Mysql databases are available to spun-up, but RDS has a selection of different database types. This will allow more data freedom. Another feature that could be added in this same vein would be the ability to export data from RDS.

9. CONCLUSION

The results demonstrate that SIQL achieves the goals previously set in section 1. SIQL is able to achieve greater efficiency than current database management solutions in the industry. By utilizing a web application approach to database management, the steps to instantiate a database are reduced and provide a database that is accessible from any device with no setup required. The Database Creation GUI provides an intuitive solution for non-database-savvy users to create and manipulate a sandboxed database through a drag and drop interface. Database manipulation is simplified for the user by querying data in natural English language and utilizing the Natural Language Processing module to convert the query into correct SQL syntax.

10. REFERENCES

dev/index.html