UNIVERSITY INSTITUTE OF ENGINEERING & TECHNOLOGY

C.S.J.M. UNIVERSITY, KANPUR

B.TECH PROJECT REPORT

ON

SQL QUERY GENERATOR FOR NATURAL LANGUAGE

Prepared By

Amit Kumar Jaiswal (001)

Vidya Sagar Singh (036)

Vivek Yadav (040)

April - 2017

Developed At

Under the guidance of

Er. Deepak Kumar Verma

'For the Partial fulfilment of degree of Bachelor of Technology at Computer Science & Engineering Department'

At

UIET, Kanpur University – 208024

UNIVERSITY INSTITUTE OF ENGINEERING & TECHNOLOGY
C.S.J.M. UNIVERSITY, KANPUR

Certificate

This is to certify that project entitled **SQL Query Generator for Natural Language** has been developed under my supervision and, to the best of my knowledge. It has not been submitted elsewhere as part of the process of obtaining a degree.

Signature

Dated: April 29th, 2017

Er. Deepak Kumar Verma
Project Mentor

Acknowledgements

First and foremost, we would like to thank our guides, Er. Deepak Kumar Verma and Er. Alok Kumar, for having suggested the topic for our project and for their constant support and guidance, without which we would not have been able to attempt this project.

It is my pleasure to be indebted to various people, who directly or indirectly contributed in the development of this work and who influenced my thinking, behavior, and acts during the course of study.

Thanks to our mentors for being a constant help throughout the project. I would specially thank Er. Deepak Kumar Verma for constantly motivating me to work harder and Er. Alok Kumar for getting me the NLP techniques.

Lastly, I would like to thank the almighty and my parents for their moral support and my friends with whom I shared my day-to-day experience and received lots of suggestions that improved my quality of work.

Abstract

Our application SQGNL, a Perl based application which processes queries submitted in natural language and translates them into SQL queries.

SQGNL is designed to be database and platform independent. It can be used by users with no knowledge of SQL to translate natural language queries to SQL. It uses linguistic dependencies and metadata to build sets of possible SELECT and WHERE clauses. Also the SQGNL has the ability to learn new grammar. Only the translation of Natural Language to SQL queries is implemented in the SQGNL.

Our program is written in Perl with a simple user interface implemented using Tk. It uses Parse::RecDescent module to build the underlying parser and DBI module for various database functionality.

CONTENTS

Introduction

Databases are very powerful means of storing and retrieving large amounts of data quickly and efficiently. There are many different commercially available database management systems used around the world. However getting data out of these databases is not an easy task. A special database interaction language called SQL (Structured Query Language) is used to communicate with these databases. Even though there is an ANSI standard for SQL, there are still minor differences between various database management systems, making it more difficult for even an experienced user to access data. A large goal of modern computational studies is to develop intelligent agents that are able to reason. Intelligent systems must be equipped with tools that allow them to learn about their environments, process the information, and make decisions. From a human perspective, the root of such tools lies in language. Thus if we are to be able to enable computational reasoning, we must develop useful computational language models.

SQGNL is aimed at reducing this complexity of database querying. First it is necessary to use a language that is understood by anybody, whether an expert database programmer or person with no computer knowledge. The best-suited language for this purpose is the English language. This means SQGNL has to translate English language queries into SQL statements before retrieving data from database.

In order to translate natural language, SQGNL needs to know the database architecture This process is automated as much as possible to minimize the complexity to the user.

Database access privileges are different for each user. Therefore multiple user access has to be supported by SQGNL. Only database querying is implemented in SQGNL and database features such as adding, deleting or updating the data in the database is not considered at this point.

SQGNL is implemented as a computer software. It translates natural language queries into SQL statements and also add to executes these to retrieve data from the database and displays them to the user.

SQGNL is implemented as database and platform independent program thus adjusting to the minor differences between various databases. SQGNL translates English sentences, phrases and keywords into SQL statements. It is also equipped with learning capabilities for the grammar that it does not understand.

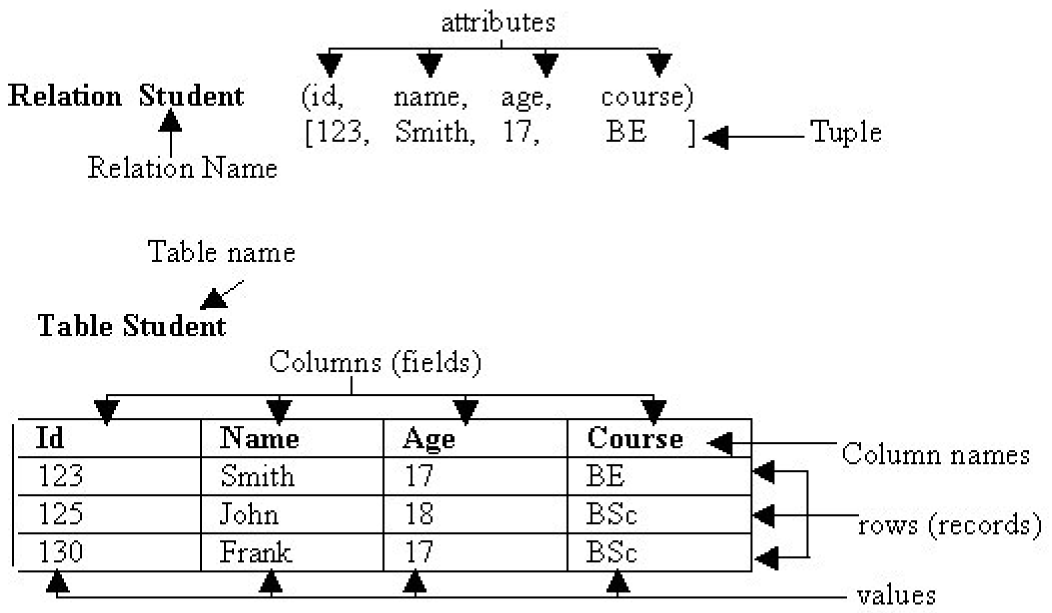
Theory

Relational Databases :

A database is a collection of related data stored in a computer. These data can be arranged and stored in many different ways. Various models explain the way in which data is arranged in the database. One of the most common and widely model is the relational database model, which is used in relational database architectures.

Relational Database Model :

A relation is defined mathematically based on set theory. The basic component of a set is an object. An object has series of attributes, where attribute is a property of the object. A series of these attributes describe an object which are organised in a tuple (record) of values. A set of objects of same type forms set of tuples, which then form a relation. These logical relations are represented universally by two-dimensional tables. Attributes related to the columns of the table and tuples forms the table rows. These concepts can be represented in a diagram as below.



Id	Name	Age	Course
123	Smith	17	BE
125	John	18	BSc
130	Frank	17	BSc

Relational databases consist of number of these relations or tables. The relations have a primary key and optional foreign key(s). The primary key is an attribute (or combinations of attributes), which uniquely identifies each tuple in a relation. Foreign key is an attribute in a relation, which serves as an primary key in another relation in the same database. This foreign key and the matching primary key of the other relation are used to create logical table joins are relationships.

This concept can be explained using the following example.

Relation Supplier (<u>supplier_id</u>, name, address, telephone)

Relation Product (<u>product_id</u>, description, suppier_id, cost, quantity)

First attribute in both relations is their primary key. The supplier_id in the relation 'Product' is a foreign key as it serves as the primary key of the 'Supplier' table. Therefore supplier_id attribute in the two relations (tables) are used to create the relationship between the two tables. Advantage of such breakdown is that the same supplier details do not need to be repeated over an over in the product table. Also the relational model does not allow such repeated values in relations. Some of these properties of the relations are,

1. each relation in a database has a unique name

2. there are no multi-valued attributes in a relation

3. no two rows can be identical

4. each attribute (column name) within a relation (table) must be unique

5. order in which table columns and rows arranged is insignificant as interchanging these columns or rows does not change the meaning of the relation

The relational model consists of three parts:

1. data structure - as explained , data is organised in tables with rows and columns

2. data integrity - various rules that defines how the data need to be managed. There are three different rules that govern data integrity.

- domain constraints - type, size (or length) or range of values that is acceptable
- entity integrity - every relation has a primary key with all primary key data been valid values (not null).
- referential integrity - either the foreign key value matches a primary key value in the other relation or foreign key value is null.
  3. data manipulation - various operations use to manipulate the data stored in the database

Relational database management systems (RDBMS) need to handle all the above operations. These DBMS not only have to manipulate data correctly, but also need to provide a means of creating, altering, deleting tables and rows in the database. The language used to do these operations is the SQL (Structured Query Language).

Structured Query Language(SQL) :

There are three types of SQL commands:

1. Data definition language (DDL) - these commands are issued to create, alter or delete tables. Usually these commands can be executed by database administrator only to prevent any accidental or deliberate damage to the database.

2. Data manipulation language (DML) - these commands are used to insert, update, delete or query data from tables.

3. Data control language (DCL) - these commands are used to grant various access privileges of the database structure to users. Only the database administrator can execute these commands.

Database Querying using SQL :

SQL commands (or commonly referred to as SQL statements) are issued to DBMS, which then execute these commands and return the results (if any). In order to query or retrieve some specific data from the database, `SELECT` statement has to be issued. The syntax for `SELECT` statement is,

```
SELECT column_list
FROM table_list
[WHERE conditional_expression]
[GROUP BY group_by_column_lis]
[HAVING conditional_expression]
[ORDER BY order_by_column_list]
```

The expressions in square brackets are optional and the words in capital letters are the SQL keywords. All the lists in the statement are comma separated. Column names and table names are usually case sensitive. The most common SQL statements involve only the SELECT, FROM and WHERE clauses.

Columns_list in SELECT clause

This list contains what columns to be displayed in the results. The asterisk (*) represents all the columns. If the tables list contain more than one table, then each column needs to be referred to as <table_name>.<column_name> format.

Following examples use the sample database shown below.

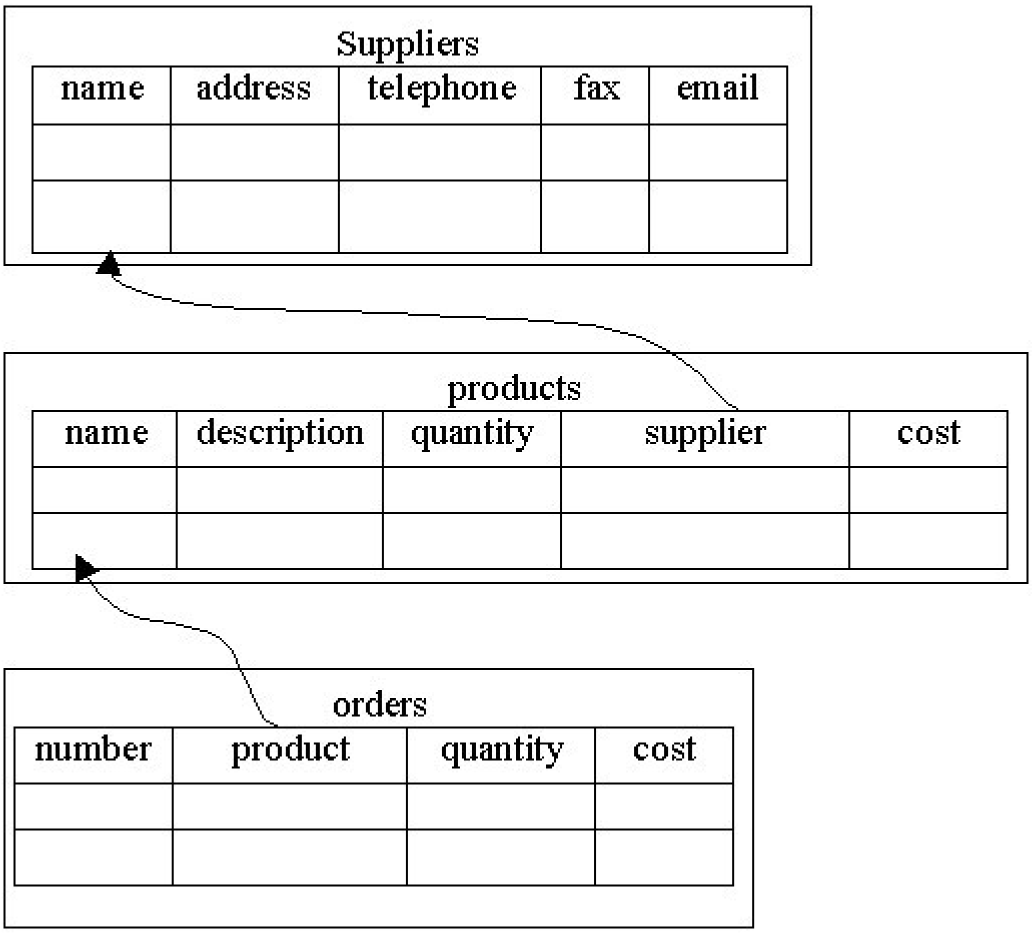


Fig. 1

Examples -

show all columns in suppliers table
SELECT * FROM suppliers

show the ‘name’ and ‘address’ columns from the suppliers table
SELECT name, address FROM suppliers

show the ‘name’ column from suppliers table and ‘description’ column from products table.
SELECT suppliers.name, products.description FROM suppliers, products

Columns list can also contain expressions or functions. Expressions have the format of:
<column_name> <operator> <column_name> AS <name_for_the_returned_column>

Example -

```
SELECT description, quantity * cost AS total_value
FROM products
```

Examples -

show total of the column quantity in products table
SELECT SUM(quantity) FROM products

show number of rows in the suppliers table
SELECT COUNT(*) FROM suppliers

show the minimum and maximum cost from the products table
SELECT MIN(cost) AS min_cost, MAX(cost) AS max_cost
FROM products

The word DISTINCT is used to filter out any repeated rows in the returned results and therefore all there returned rows would be unique.

Conditional Expression (in WHERE clause)

The conditional expression restricts what rows are considered in the final results. Expressions have the format,

```
<column_name> <operator> <value>
```

List of valid operators allowed in these conditional expressions are shown below.

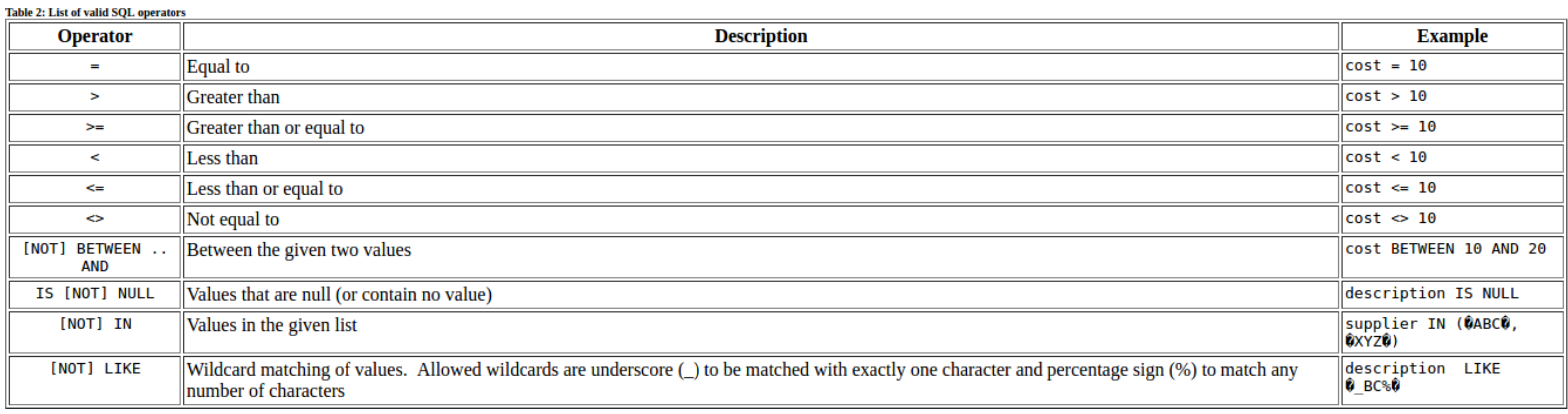


Table 2: List of valid SQL operators

Operator	Description	Example
`=`	Equal to	`cost = 10`
`>`	Greater than	`cost > 10`
`>=`	Greater than or equal to	`cost >= 10`
`<`	Less than	`cost < 10`
`<=`	Less than or equal to	`cost <= 10`
`<>`	Not equal to	`cost <> 10`
`[NOT] BETWEEN .. AND`	Between the given two values	`cost BETWEEN 10 AND 20`
`IS [NOT] NULL`	Values that are null (or contain no value)	`description IS NULL`
`[NOT] IN`	Values in the given list	`supplier IN (�ABC�, �XYZ�)`
`[NOT] LIKE`	Wildcard matching of values. Allowed wildcards are underscore (_) to be matched with exactly one character and percentage sign (%) to match any number of characters	`description LIKE �_BC%�`

Fig. 2

Parsing:

Translating from one language to another always requires some kind of paring. Whether we translate from English to German or English to SQL, parsing plays one of the most important roles. Parsing is the process of identifying structure in data. It determines whether the input data has some pre-determined structure and respond accordingly.

Parsing requires a set of grammars to be defined. These grammars are set of rules, which define how the language is structured. Rules are specific pattern of data, which appears in the input. These rules can further reference to other rules, which are called subrules. Rules can also be recursive if they somehow refer back to themselves. The rules are consists of products, which are the different ways in which the rule can be satisfied. The actual data is referred to as terminals.

Terminology used in parser grammar

```
grammar    :  rule(s)         # grammar is one or more rule

rule       :  production      # there are number of ways
           |  production_2    # the rule can be satisfied
           |  production_3

production :  terminal_x  subrule  terminal_y

subrule    :  terminal_x {action}

terminal_x :  'x'             # terminals are the actual data

terminal_y :  'y'

action     :  "print 123"     # actions can execute piece of code
```

Given a set of grammar, there are two ways of implementing a parser: bottom-up parser or top-down parser.

SQGNL Design

The design of SQGNL can be broken down into three sections: database functionalities, natural language translation and user interface.

Required Database Functionalities :

Number of database functions need to be implemented for the operation of SQGNL. First it is required to establish a connection to the chosen data source. Since SQGNL is intended to be database independent, the implementation must support establishing connections to different types of databases. The data source may require user login. Therefore the program needs to know the current SQGNL user and his/her password to access the database. Except for password, all the other information (database type, data source and user name) needs to be saved for future use. Password however is required to be entered by the user every time user runs the program.

Before translating any English statements, it is essential to know the database structure. This structure can then be used to match tables and columns to the natural language queries. In order to know the database structure, SQGNL needs to retrieve table names, column names and column data types. Column data type is required to format the values in SQL statements as string values need to be quoted and date values need hashes (#) around the value. Since each user have different access privileges to the same database, the database structure visible to the user may differ from one another. SQGNL only needs to consider what is visible to the user, not the complete database structure.

The third database functionality required is the execution of the translated SQL statements. Only `SELECT` queries are executed and other types of SQL statements are not implemented in SQGNL. One of the main reasons for this is that the `SELECT` queries do not alter the database and therefore it is more secure against accidental changes. Once the SQL statement is executed, the results have to be displayed to the user. The results could either be an SQL error or a number of database records. The maximum number of database records to be displayed must be limited, as it is possible that the resulting record set may contain thousands of records. Displaying such huge information is not required and it will consume lots of computer resources.

Design of Natural Language Translator :

In order to translate English statements we need to use grammar to define various rules. To explain these concepts, I will use a sample database with the following structure from Fig. 1

Now if we want to see all the data in the 'suppliers' table or 'product' table, we can write natural language queries as,

show me all our suppliers
show me all our products

and these English statements are translated into SQL as,

```
SELECT * FROM suppliers
```
and
```
SELECT * FROM products
```

respectively. When we analyse the above two English sentences, we can see that these are very much similar in there structure. The only difference between them is the table name. This can the be generalise to form the grammar,

show me all our <table_name>

and the translated SQL statement is,

```
SELECT * FROM <table_name>
```

The words within angle brackets can be any table name, and for the above examples they are suppliers and products.

The same statements can also be expressed in slightly different ways. Following are eight different variations of the first English query.

show me all our suppliers	show me our suppliers
show me all suppliers	show me suppliers
show all our suppliers	show our suppliers
show all suppliers	show suppliers

These eight queries can be generalised to one single grammar as below.

show (me) (all) (our) <table_names>

The words in brackets are optional words, which can be omitted in the queries.
Furthermore the first word of the query may also be changed to create more queries.

show me all our suppliers
list me all our suppliers
display me all our suppliers

For the above three queries, the first word is substituted with three similar words. This can also be included in our grammar to extend it even further, which leads to the new grammar:

  [show|list|display] (me) (all) (our) <table_names>

The one word from the above list (with the square brackets) must be appeared in the query. The above grammar can produce 24 different English statements for one table name, but will translate to only one SQL statement.

  SELECT * FROM <table_name>

To increase more translating possibilities, we can define more grammar. However there is a slight problem with this kind of grammar definitions. For example if we define grammar similar to

  [what|who] are our <table_name>

This will not only accept queries like,

  who are our suppliers
  what are our products

the grammar will also accept queries like,

  what are our suppliers
  who are our products

which are grammatically incorrect. However correcting this problem is quite hard as we need to compare the table name against the first word. Since this grammar accepts the correct grammatical queries, we can ignore this problem as we can assume that the users will always write the correct English.

More grammar rules can be defined to retrieve information from only one column of a table. For example;

  show the names of our suppliers
  show the descriptions of our products

statements can be generalise to create the grammar,

  show (the) <column_name> of (our) <table_name>

and the translated SQL will look like,

  SELECT DISTINCT <column_name> FROM <table_name>

Note that the words names and descriptions are not the exact column names, but are the plurals of columns name and description. It may also be possible to have similar words to column names or table names in the queries instead of the exact words. Therefore we not only need to keep track of table and column names, but also need to keep track of similar words. Except for plurals, users need to manually enter other synonyms. If I was to overcome this problem, I need to include a thesaurus with SQGNL. But due to time limitations, this is not possible. The word DISTINCT is used in the SQL statement to get rid of any repeated values as we are looking at only one column.

To retrieve two columns from the database, queries can be written as,

show me names and addresses of our suppliers
show me descriptions and cost of our products

and the grammar for this is,

show (me) <column_name1_> and <column_name_2> of (our)<table_name>

and the resulting SQL statement will be,

```
SELECT <column_name_1>,<column_name_2>
FROM <table_name>
```

More grammar rules can be defined to create SQL statements that contains COUNT and SUM functions. SQGNL will only implement these two functions, as they are the most commonly used SQL functions.

The queries with conditions can also be recognised and generalised to form the grammar. For example,

show me the names of our products supplied by ABC
show me the address of our supplier whose name is XYZ

queries will have the grammar,

show (me) (the) <column_name_1> of (our) <table_name> (whose) <column_name_2> [by|is] <value>

which can translate to the SQL statement,

```
SELECT <column_name_1> FROM <table_name>
WHERE <column_name_2> = <value>
```

Many of these conditions can be generalised by separate grammar. Some of these grammar rules and corresponding SQL statements are summarised in the following table.

Sample query	Grammar	SQL
� supplied by ABC	� <column_name> [by|with|is|are|�] <value>	`... WHERE <column_name> = <value>`
� cost is between 10 and 20	� <column_name> ([is|are]) between <value_1> and <value_2>	`... WHERE <column_name> >= <value_1>` `AND <column_name> <= <value_2>`
� cost is higher than 10	� <column_name> ([is|are]) [greater|larger|higher|�] than <value>	`... WHERE <column_name> > <value>`
� cost is less than 10	� <column_name> ([is|are]) [less|smaller|lower|�] than <value>	`... WHERE <column_name> < <value>`
� name starts with abc	� <column_name> [starts|start] with <value>	`... WHERE <column_name> LIKE �<value>%�`
� name ends in abc	� <column_name> [ends|end|ended] in <value>	`... WHERE <column_name> LIKE �<value>%�`
� name contains abc	� <column_name> [contains|contains] <value>	`... WHERE <column_name> LIKE �<value>%�`

If the value consists of more than single word, it must be quoted to recognise as a single value.

Two table queries are another complicated queries that the SQGNL should be able to translate. If the query looks like,

show me our suppliers and their products

The grammar for this is,

show (me) (our) <table_name_1> and (their) <table_name_2>

and the translated SQL statement is,

SELECT * FROM <table_name_1>, <table_name_2>
WHERE <relationship_between_tables_1_and_2>

To create this SQL statement, SQGNL needs to know the relationships between tables. It is quite hard for the program to determine these relationships automatically. Therefore user assistance is required to determine these. The user has to manually enter these relationships, which can then be stored for future use. However if the user writes a natural language query similar to,

show me our suppliers and orders

but there is no direct relationship between supplier and orders table. Even though this can be translated into a valid SQL as,

SELECT * FROM suppliers, order

It will not produce the expected results and the results may contain thousands of unrelated records. Therefore if the relationship between the two tables is not defined, SQGNL should not translate the English query to SQL.

To translate natural language queries, SQGNL needs to have a parser, which contains all these grammar rules. The parser also has to know the table names, column names, any relationships between tables and related words for the table and column names.

It is not possible to have all possible combinations of grammar included in SQGNL parser. Only the most common grammar is included in the parser. For all the other possibilities, a learning facility can be design and implemented to the parser. For example, if the SQGNL does not understand the query,

show me our suppliers names

but when rephrased it can understand the query,

show me names of our suppliers

Then the program can analyse the first statement to find table names, column names or conditions and then create a new grammar. By analysing the first English statement, SQGNL can create the grammar as,

show me our <table_name> <column_name>

The corresponding SQL statement is the same for the second English query. That is,

SELECT <column_name> FROM <table_name>

Now this grammar and the corresponding SQL statement can be append to the parser so that it can learn the new grammar for other similar statements. Once this grammar is added to the parser, SQGNL can translate queries such as,

show me our suppliers address
show me our product descriptions
show me our order dates

Next challenge is storing this new grammar and database structure for future use. There are two options available. These information could either be stored in a file and load every time the user run the program, or save the complete parser every time a change occurs. The first option could be a very slow process if there are lots of new grammars to be learnt. Therefore second option of saving the parser to a file is implemented in SQGNL.

Design of User Interface :

The basic user interface consists of at least four windows. First we need a user login window to get the user password to access the database. This window should contain two areas for the user to enter the user name and password.
Secondly a configuration window is required to get the information about the database from the user. This information consists of database type, data source, user name and the password. Additional information such as maximum number of records to display and choice of enabling the learning process can also be entered in the same window.

Another window is required to get extra information about the database structure. This information includes related words for tables and columns and relationships between tables.

Finally the SQGNL main window, which contains areas for English statement entry, SQL output of the translation and area to display play the resulting records from the database.

SQGNL Implementation

SQGNL is programmed in Perl. Perl is chosen as the preferred programming language because of its simple and powerful string manipulation capabilities. Since SQGNL relies much on string manipulation tasks, this is the major factor of considering the implementing language. Furthermore Perl already has many useful modules written by other people, which can be used in SQGNL. In particular the use of a parser module will simplify the process of coding, as I do not have to spend time on writing my own code. Also there are many different modules for database communications and implementing user interfaces in Perl. Perl can also write platform independent program and this is suited for the purpose of SQGNL. Considering all the above factors, Perl is the best candidate for writing SQGNL.

Implementation of SQGNL is discussed in more details in the following sections.

Implementing Database Functionalities

DBI module is used to implement the required database functions. DBI is chosen as it can be used to write database independent code without repeating similar code to different types of databases. In order for DBI module to work, the corresponding DBD module is required to be installed in the system. These DBD modules have the database specific code, which is encapsulated by the DBI module. If the appropriate database driver (or DBD module) is not installed in the user system, the program cannot progress any further. If the user does not know the type of database, SQGNL can then try all the available database types until a valid connection is established.

Table names can be accessed using the DBI:table_info() method. However for DBD::ADO module, table_info() method is not yet implemented. Therefore SQGNL cannot access table names from ADO type (e.g. Microsoft Access) databases and does not currently support them.

There is no direct method of retrieving column names. The only way is to execute SQL statement for each table to receive information from all columns and look up the returned structure. This structure contains column names, their type and other information, which are not relevant for the purpose of SQGNL.

Natural Language Translation :

Rather than writing my own parser for SQGNL, I have decided to use Parse::RecDescent module to implement the natural language parser. Parse::RecDescent is a top-down parser, which gives all the advantages of top-down parsing. There are other bottom-up parsers such as Parse::Yapp and perl-byacc , which I could have used. But the Parse::RecDescent has some special features that are more suited for this application. Most importantly, the RecDescent module can,

1. generate run-time parsers - so we can embed the database structure during the run-time. This is important as we do not know the database structure until we connect to the database during run-time.

2. modify or extend the parser during run-time - useful for SQGNLs learning functionality as we need to extend the parser to learn new grammar.

3. save the parser object to a file and reload it in future - this is another important feature as this enables the parser object to be created only once and new extensions to the grammar can be saved in the same file as the parser rather than a separate file for the new grammar learned. Also loading the parser from a file is significantly faster than creating the parser.

There are many other useful features of this module , but is not much of an importance for SQGNL purposes. The only significant drawback of the RecDescent module is the slow speed in creating the parser object. This is expected from a top-down parser and for the RecDescent module slow speed is more evident due to run-time creation of the parser. Given the advantages of the Parse::RecDescent module, we can afford to trade of the slow speed to other useful features.

The use of Parse::RecDescent module in SQGNL can be explained using examples. Lets start with some simple grammar as explained in the design section.

Grammar : [show|list|display} (me) (all) (our) <table_name>

SQL : SELECT * FROM <table_name>

The above grammar can be translated to perl code that the parser can understand.
my $grammar = q{

```
translate : select

select    : ask /(me)?/ /(all)?/ /(our)?/ table_name
            { "SELECT * FROM $item[5]" }

ask       : /show|list|display/
```

```
    table_name:    TABLES
};
```

Since we do not know database structure at compile time, we cannot include the table names. Therefore during run-time the word `TABLES` needs to be replaced with actual table names. This can easily be done with the help of regular expressions.

my $table_names = get_table_names();

```
$grammar =~ s/TABLES/$table_names/;
```

The words within the curly brackets ("{}") are the actions to execute if the rule is successfully matched. For this case, words within the brackets are returned on a successful match. If the rule does not contain any action, then the matched token is returned. For example, the subrule '`ask` may return the word '`show'` or '`list'` or '`display'` if it matches it. Otherwise the subrule will return undef and this will cause that branch of parsing to be invalid.

The variable $item[5] is the 5th token matched for that rule. For the 'select' rule it is the table name.

Once we defined the grammar, parser object needs to be created using this grammar. Following is the perl code used to create the parser.

```
my $parser = new Parse::RecDescent($grammar);
```

Once the parser object is created, we can call any of the rules defined in the grammar as a function call. Since this is a top-down parser, only the most general rule is called. Therefore we can translate some natural language query to SQL as below.

```
my $input = "show me our suppliers";

my $sql = $parser->translate("\L$input");

print $sql;
```

This block of code will translate the input to SQL and output the results. Since the parsing is case sensitive, the input has be converted to lower case ("\L") before calling the translate function. Now we can add more grammar rules to the $grammar variable, but do not need to do changes to any other code.

Grammar : show (the) <column_name> of (our) <table_name>

SQL : SELECT <column_name> FROM <table_name>

The parser grammar is,

```
my $grammar = q{
    translate : select

    select :  ask /(me)?/ /(all)?/ /(our)?/ table_name
                   { "SELECT * FROM $item[5]" }

                | ask /(he)?/ column_name /of/ /(our)?/ table_name check[$item[3],$item[5]]

                  { "SELECT $item[3[ FROM $item[6]" }

    ask : /show|list|display/

 table_name : TABLES

 column_name : COLUMNS

    check : <reject !check_column($arg[0], $arg[1]) >
};
```

The check rule checks whether the given column (argument 0) belongs to a particular table (argument 1) and if no reject the current production.

Example -

show the names of our suppliers – accepted

show the costs of our supplier – rejected

Since the grammar itself does not know which column belongs to which table, the check_column function has to know this information. Therefore SQGNL has to store the database structure in a variable so that it does have to read the database structure every time. To add new grammar we only need to modify the $grammar variable. So even at run-time, we can modify the grammar. This form the basis of the learning functionality in SQGNL. Example - lets say SQGNL encountered following new grammar during run-time.

Grammar : how may <table_name> are there

SQL : `SELECT COUNT(*) FROM <table_name>`

Following code is use to extend the `select` rule in the grammar.

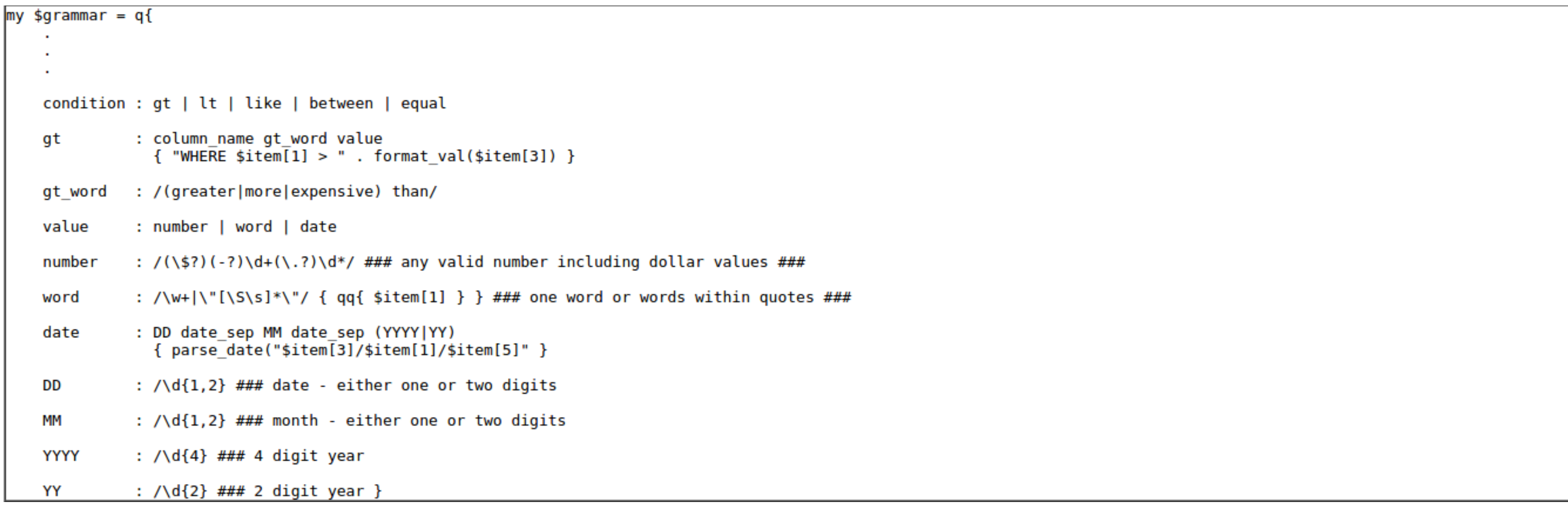
```
$rule = "select";
$production = "/how many/ table_name /are there/";
$action = "{ qq(SELCT COUNT(*) FROM $item[2]) }";

### define the new grammar to be learn ###
my $new_grammar = qq{ $rule : $production $action };

### extend the parser with new grammar ##
$parser->Extend($new_grammar);
```

This code will append the new grammar to the end of the select rule as a new production. (i.e., joined by "|").

Grammar rules for the conditions are slightly complicated than other grammar. Complication rises where we need to determine the type the value. These conditions can be explained used the following sample code for the 'greater than' condition.



```
my $grammar = q{
    .
    .
    .

    condition : gt | lt | like | between | equal

    gt        : column_name gt_word value
                { "WHERE $item[1] > " . format_val($item[3]) }

    gt_word   : /(greater|more|expensive) than/

    value     : number | word | date

    number    : /(\$?)(-?)\d+(\.?)\d*/ ### any valid number including dollar values ###

    word      : /\w+|\"[\S\s]*\"/ { qq{ $item[1] } } ### one word or words within quotes ###

    date      : DD date_sep MM date_sep (YYYY|YY)
                { parse_date("$item[3]/$item[1]/$item[5]" }

    DD        : /\d{1,2} ### date - either one or two digits

    MM        : /\d{1,2} ### month - either one or two digits

    YYYY      : /\d{4} ### 4 digit year

    YY        : /\d{2} ### 2 digit year }
```

- format_val function will format the value into appropriate SQL format. That is, it will put quotation marks around sting values and hashes (#) around date values.
    - Example
      `format_val("20.92")->20.92`
      `format_val("ABC")->"ABC"`
      `format_val("22-Oct-1976")->#22-Oct-1976#`

- The date parsing shown above grammar is only one way of representing the date. There are may other ways of expressing date including dates strings such as "today", "yesterday", etc. SQL does not interpret these date stings and therefore need to convert it into US-standard date string (MM/DD/YYYY). This is done within the `parse_date` function with the help of Date::Manip module. Even though 20-20-2000 will parse as a date string, the `parse_date` function will reject it and therefore the parser will not accept it as a valid date. (it even rejects 2/29/1900 but accepts 2/29/2000)

User Interface :

The user interface is implemented using the Tk module. Tk is chosen instead of Win32::GUI module because it can create windows that are platform independent.

The User Manual

Outline :

- Software Requirements
- Hardware Requirements
- Installing SQGNL
- Running SQGNL
- Login in to the Database
- SQGNL Main Window
- Database Structure

1. Software Requirements

SQGNL was successfully tested on Linux operating system. In order to run this application, you need to have Perl installed in your computer. There are number of Perl modules required by this application. The modules required are :

List of Perl modules required by SQGNL :

Module	Function
Parse::RecDescent	Parser module for SQGNL

DBI	Handles various database functionalities
Tk	Implements the User Interface
Date::Manip	Handles various database related activities
Lingua::EN::Inflect	English Grammar module

To download the latest perl from the Internet site `www.perl.com` and all the above modules from `www.perl.com`. Installation instructions for perl modules can be accessed from the following sites.

- For Windows/non-Windows operating systems : `http://www.perl.com`

2. Hardware Requirements

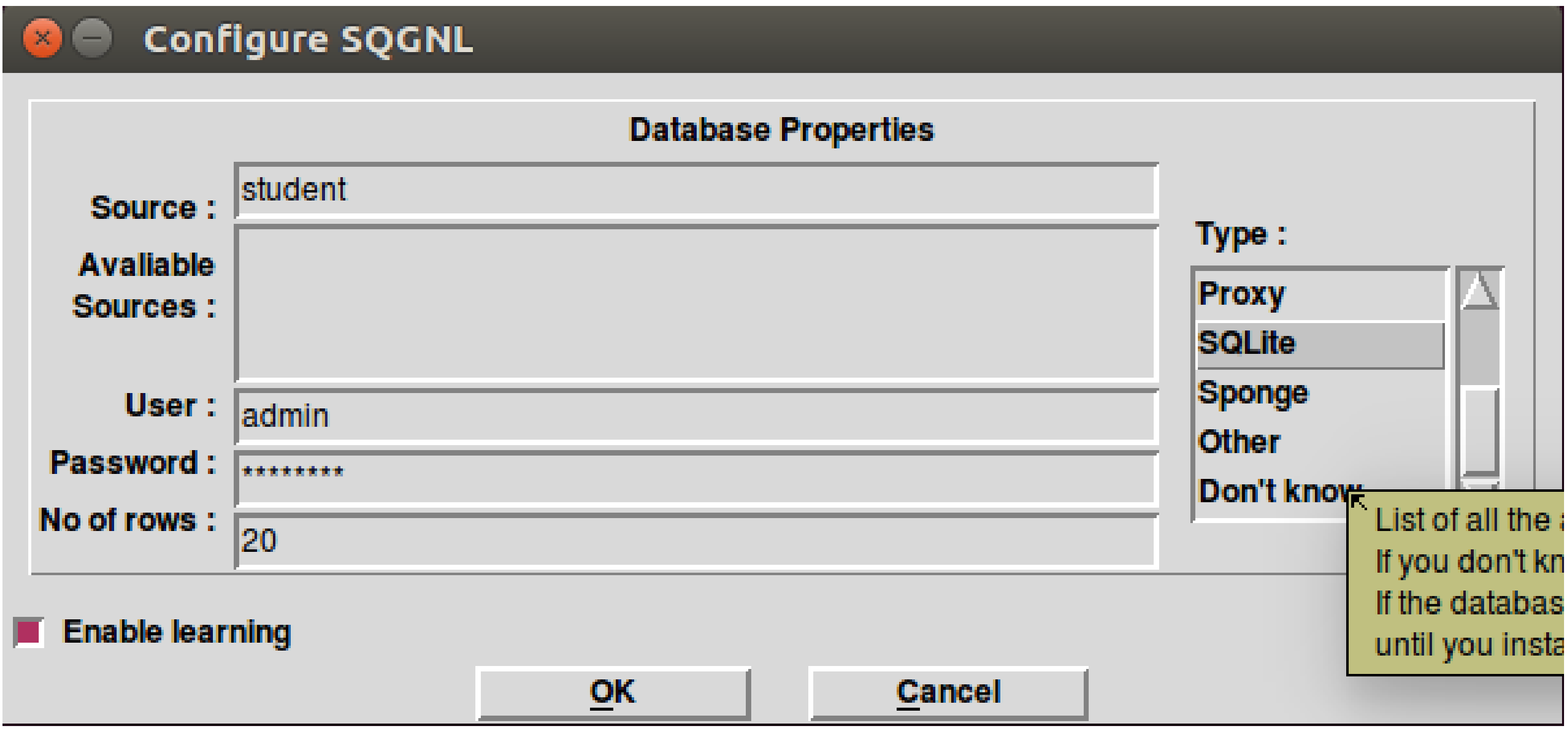
You need about 200 KB space for the SQGNL codebase and at least 2 MB of disk space for the SQGNL parser. Also the screen resolution of 800x600 or higher is recommended.

3. Installing SQGNL

Copy all the perl source code (files with the extension ".pl") to a new directory where the user has the write permission. The write permission is required as the application needs to save the parser for future use. Do not delete, rename files or modify any of the source code unless you know what you are doing. You can run SQGNL by typing "`perl sq-hal.pl`" in the command prompt.

4. Running SQGNL

When you run SQGNL for the first time, following configuration window will come up. You need to enter all these information before continuing to the SQGNL.



Configure SQGNL

Database Properties

Source : student

Avaliable Sources :

User : admin

Password : ********

No of rows : 20

Type :

Proxy

SQLite

Sponge

Other

Don't know

List of all the

If you don't kn

If the databas

until you insta

Enable learning

Configuration Window

Source - The location and/or the name of the data source

Type - List of all the available database drivers installed in you computer. Select the type of the database if you already know or otherwise chose the option "Don't Know". SQGNL will attempt to find the database type if you do not know the database type.

If type of database is not in he list (i.e. "Other"), then the SQGNL cannot proceed any further until you install the appropriate database driver. For more information about database drivers, check software requirements[1].

Available sources - These are the available data sources for the selected database type. You can either select the data source from this list or type it in the source entry field.

User - user name to access the database.

Password - password to access the database.

No of rows - Maximum number of rows to display when the SQL statement is executed. It is strongly recommended to keep this value below 100 as this may consume lots of computer resources. (default – 20)

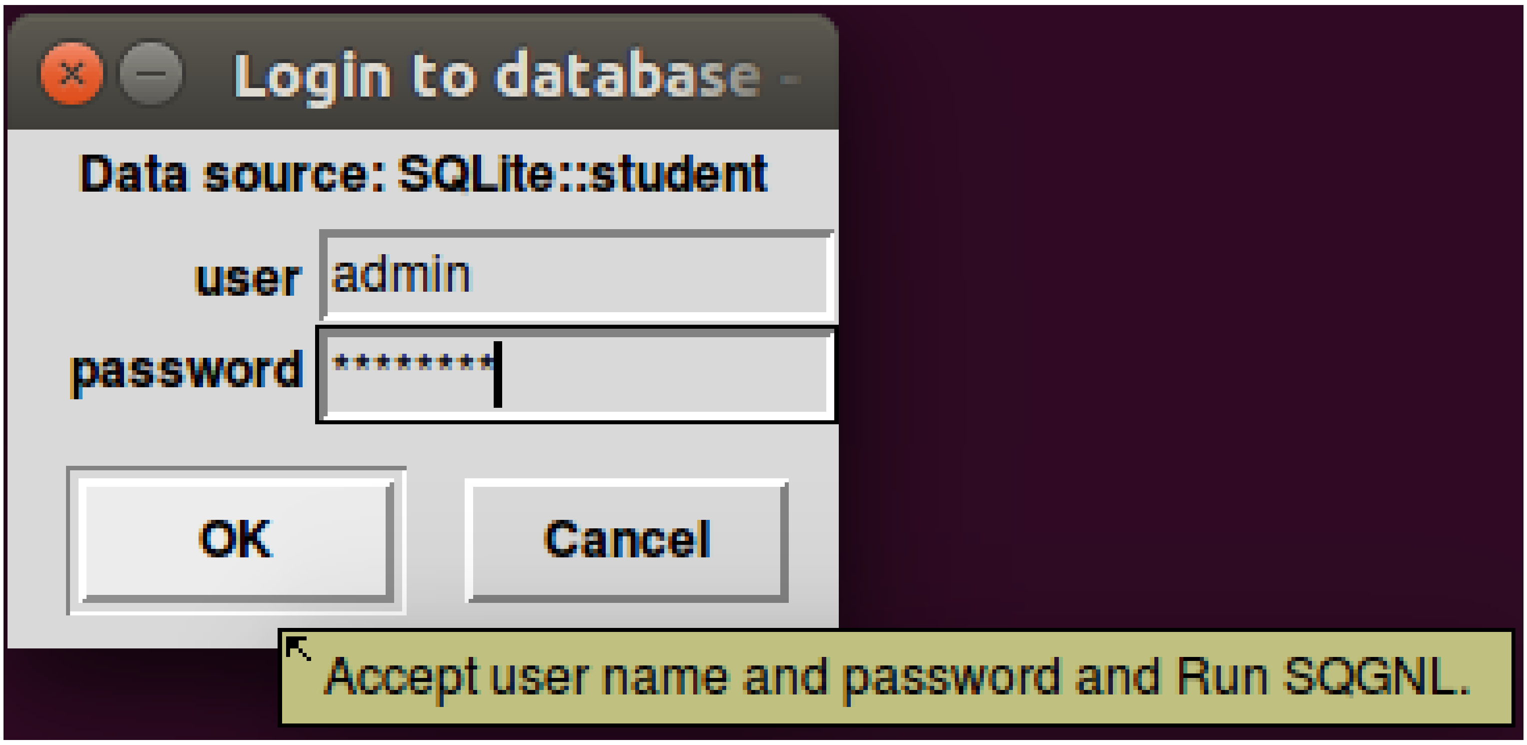
Enable learning - Enables or disables the learning functionality of SQGNL. (Default – Enabled)

OK - Accept all the configuration values entered and close this window. These information (except the password) will be saved to the configuration file “sq-hal.ini” for future use. If this configuration file is not found (e.g. got deleted), then SQGNL will bring up this configuration window to recreate it.

`Cancel` - Do no accept changes and close this window. As it is very important for SQGNL to know the database information, pressing this button will exit the application.

5. Login in to the Database

Since SQGNL does not store your password to access the database, you need to enter the database password every time you run the program. Therefore SQGNL will always bring up the following login window at the start-up.


Login to database

Data source: SQLite::student

user admin

password ********

Accept user name and password and Run SQGNL.

Database Login Window

`user` - User name to access the database.

`password` - password to access the database.

OK - Accept the user name and password and continue running SQGNL.

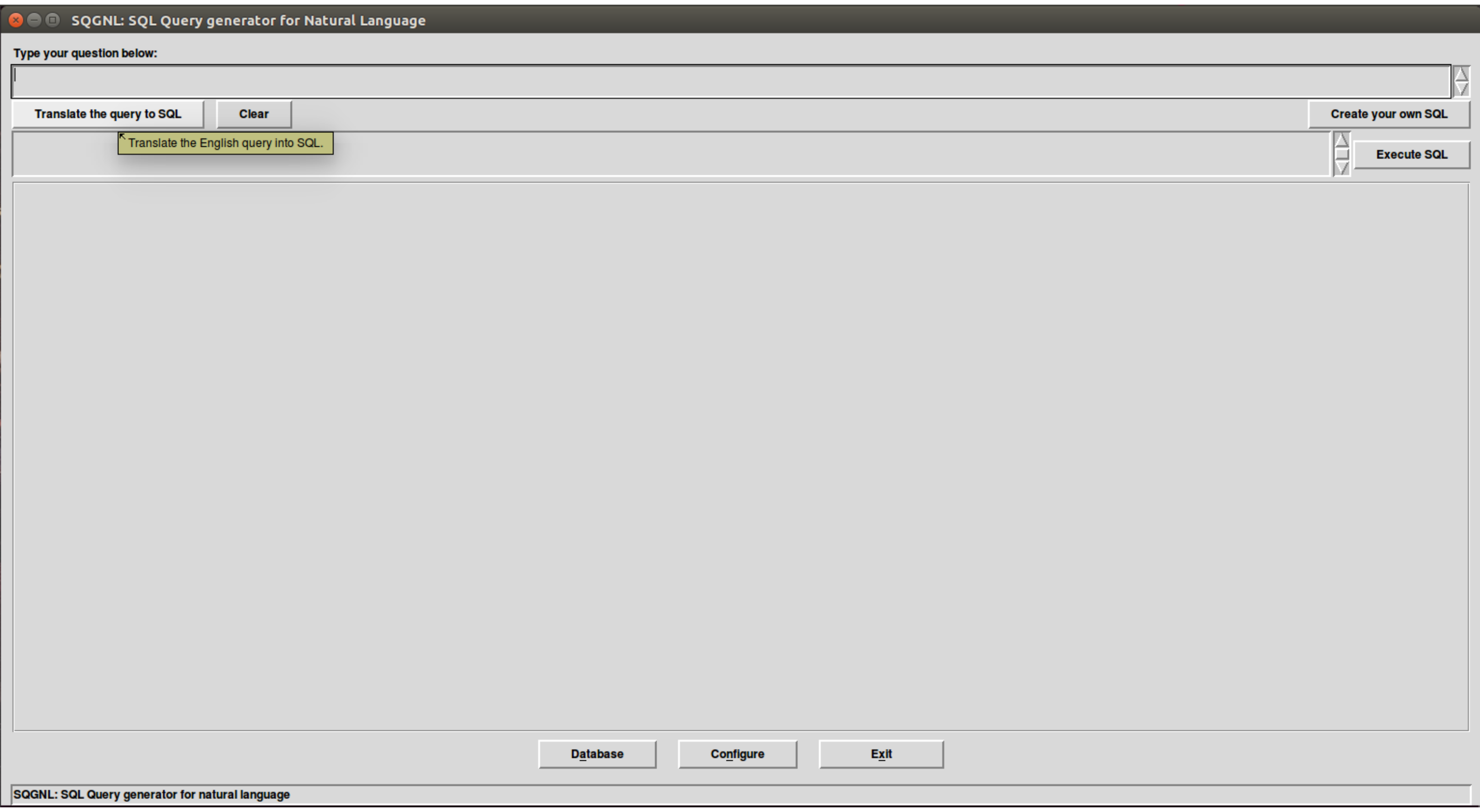
Cancel - do not accept the user name and the password and close this window. This will exit the program.

Once you press OK, then a splash screen will be displayed. During this time, the program will establish a connection to the database and will create or load the SQGNL parser. Any errors in database connection or loading the parser will be displayed and the program will be aborted.

Once the SQGNL establish successful connection to the database and loaded the parser, it will bring up the SQGNL main window.

6. SQGNL Main Window

This is an easy to use, simple window. Following sections describe the features of the main window.


SQGNL: SQL Query generator for Natural Language
Type your question below:
Translate the query to SQL
Clear
Create your own SQL
Translate the English query into SQL.
Execute SQL
Database
Configure
Exit
SQGNL: SQL Query generator for natural language

SQGNL Main Window

<u>`Translate the query to SQL`</u> - As the name suggests, this button calls the SQGNL parser to translate your natural language query to SQL. The results of the translation is displayed the text area immediately below this button.

<u>`Clear`</u> - clear the content in the natural language query area.

<u>Create your own SQL</u> - This brings up the window where you can easily create SQL statements by selecting tables and columns.

SQL Statement - Translation of natural language queries is displayed here. If the translation is successful, the SQL statement is displayed in blue colour or otherwise error messages is displayed in red colour.

As soon as valid statement is displayed in here, SQGNL automatically executes this SQL statement and displays the data in the results table. Also if your previous English queries were unsuccessful, then a learn grammar window will also pop up.

You can also change the SQL statement or type your own SQL statements here and press "Execute SQL" button to execute and display the results.

Execute SQL - Execute the SQL statement and display the results in the results table. If here is an error in the SQL statement, an error message is displayed.

Note that you can only execute SELECT queries. Trying to execute any other type of SQL statement will generate and error message.

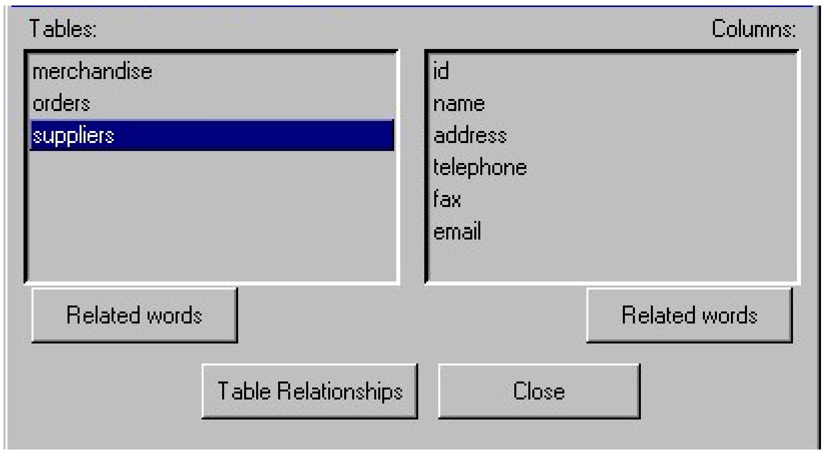
Database - brings up he window that displays the database structure.

`Configure` - brings up the configuration window where you can change database and other information.

`Exit` - Exit from SQGNL to the system. On exit, SQGNL may save the parser if it has been newly created or has been changed during the current session. The parser is save to a file call "sq_hal_<database_type>_<data_source>_<user_name>.pm". Saving of the parser may take some time. So please be patient if this is the case.

7. Database Structure

The database structure of the current database can be viewed in the database structure window. Various database related information can also be entered in here.

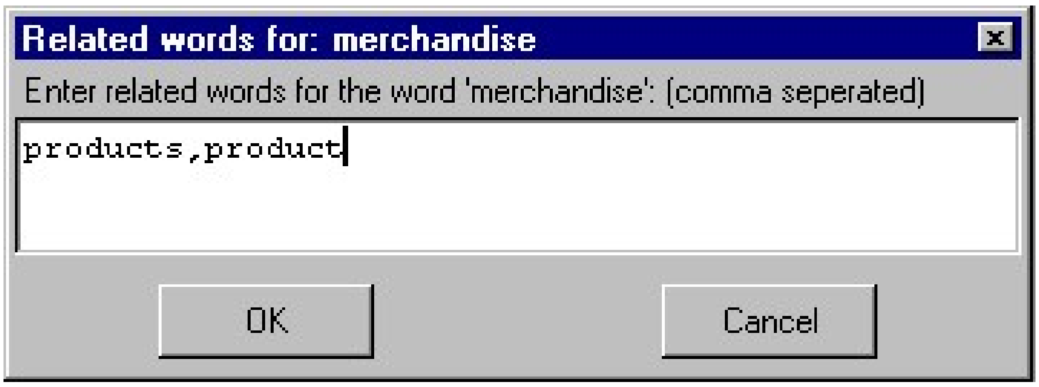


Database Structure Window

Tables - This left most list contains all the available table names in the database. When you select a table name from the list, corresponding column names in the selected table will appear in the column names list.

Columns - List of column names for the selected table.

Related word - select either a table name or column name before pressing this button. This will bring up the following dialog box.


Related words for: merchandise
Enter related words for the word 'merchandise': (comma seperated)
OK
Cancel

products,product

Related words entry dialog box

You need to enter any related words for the selected table name or column name and press OK to accept these words.These words need to be comma separated without any spaces in between them. These new words will then be appended to the SQGNL parser.
Close - Close this window and bring back the SQGNL main window to the front.

Table Relationships - This brings up the following window where you define relationships between various tables in the database.

Table names list - This left most list contains the list of all the table names in the database. Select the two table names, which involves in the relationship by clicking on them. When you chose the two table names, the two column names list to the right will be updated with corresponding column names.

Column names list 1 - Column names list for the first selected table. Select the column name involves in the table relationship.

Column names list 2 - Column names list for the second selected table. Select the column name involves in the table relationship.

Current relationship - This is the table relationship you have just crated by selecting two table names and corresponding column names.

Add - Adds he current relationship to the list of all the relationships. If the relationship already exists, confirmation is asked to overwrite the existing relationship.

All the relationships - List of all the relationships for the database, which were created during the current session or previous sessions. You can change any of these relationships by recreating that relationship.

Accept all - Accept any changes you have made to the relationships list (including new relationships and changes to existing relationships) and close this window. These relationships are saved to a file for future use.

Cancel - Cancel all the changes made to the relationships list and close this window.

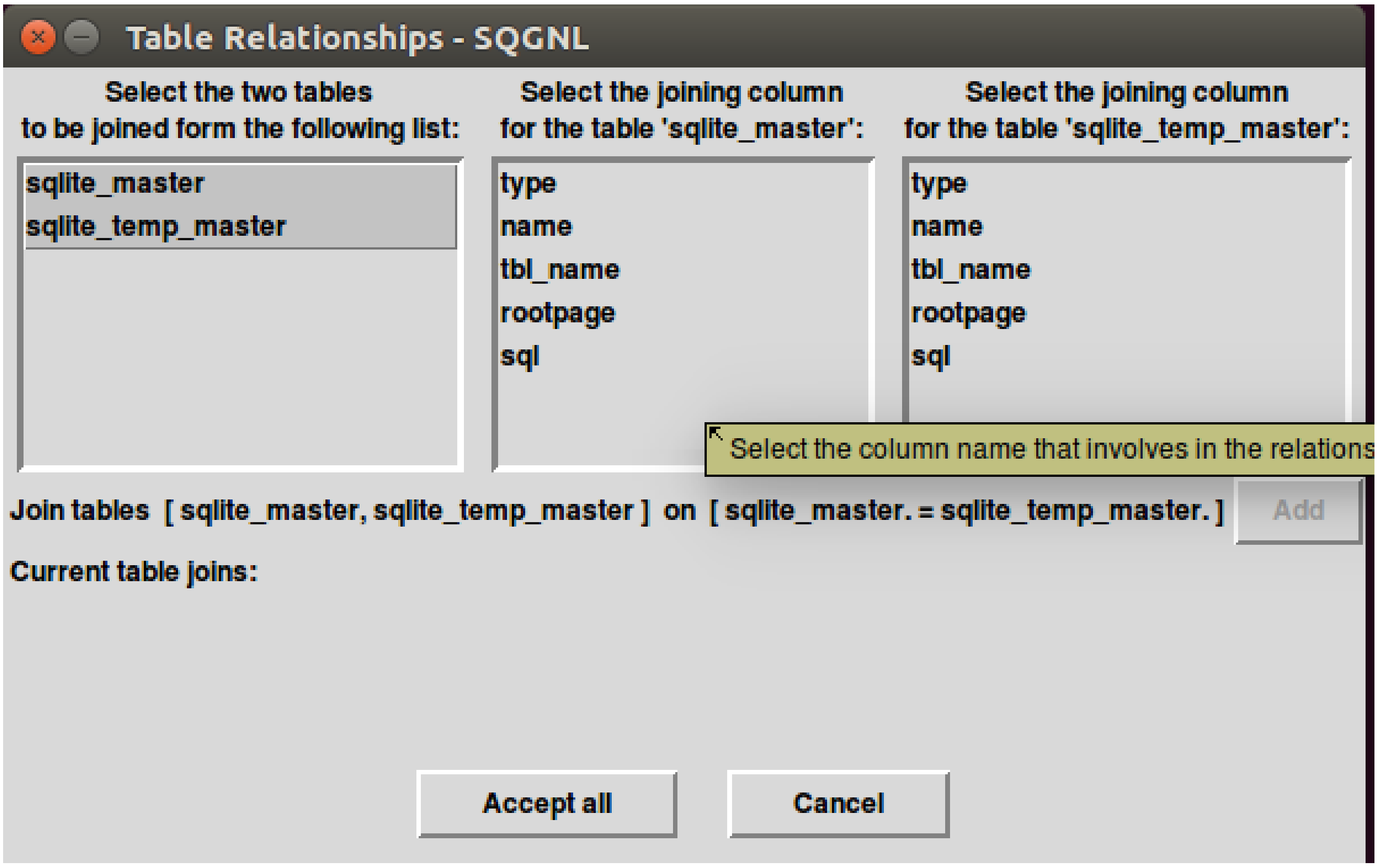


Table Relationships - SQGNL

Select the two tables
to be joined form the following list:

sqlite_master
sqlite_temp_master

Select the joining column
for the table 'sqlite_master':

type
name
tbl_name
rootpage
sql

Select the joining column
for the table 'sqlite_temp_master':

type
name
tbl_name
rootpage
sql

Select the column name that involves in the relations

Join tables [sqlite_master, sqlite_temp_master] on [sqlite_master. = sqlite_temp_master.]

Current table joins:

Table Relationships Window

CONCLUSION :

Most of the aims of SQGNL were implemented successfully. The program can translate simple natural language queries in to SQL. It can translate to different types of SELECT queries, which include retrieving data from single or two tables with or without a single condition. Learning capability of SQGNL is also been implemented with some success. It is not as efficient as expected because it can only detect table names, field names and conditions in the queries but cannot generalise other words such as determining which words can be optional and may omitted in queries.
SQGNL is almost database independent with the exception of ADO databases. The program successfully runs on Linux (Unix) environments. Multiple user access is also supported as a new parser is created for each user and saves in different files.

LIMITATIONS :

SQGNL is known to have a few limitations and problems. First of all database table names and column names have to be valid English words. Multiple words of these names (eg. telephoneNo) will be treated as single word and may not produce expected results. SQGNL cannot determine synonyms for table names and column names and therefore user has to manually enter these words. Similarly program is not capable of determining relationships between tables and user assistance is expected to solve this problem. Even for a small grammar, parser is noticeably slow when loading and saving to a file (specially in Win32 platform). This problem is minimised by loading the parser only when the program starts and saving it when the program exists. Currently SQGNL does not support ADO (e.g. Microsoft Access) databases because the program cannot retrieve table names from an ADO database and no support for DML statements.

FUTURE WORK :

SQGNL parser can be improved significantly. More new grammar can be added to the parser to increase effectiveness. Adding a thesaurus is another suggestion, which could help automating the related words for table and column names. With the help of a thesaurus, the user input can be pre-processed to substitute related words with table or column names and also remove unwanted words. This pre-process could lead to smaller but more efficient parser. More work has to be done on the learning grammar functionality to make it more effective. Finally SQGNL can be given the ability to translate queries other than just selecting data from the database. SQGNL parser can be extend so that it can also add and delete data from tables, modify or create new tables or even create new databases.

REFERENCES :

1. Akeel I Din, "Structured query language (SQL) A practical Introduction", NCC Blackwell Ltd, 1994

2. Fred R. McFadden, Jeffery A. Hoffer, Mary B. Prescoh, "Modern Database Management", 5th Edition, Wesley Educational Publishers Inc, 1999

3. Computer Technology Research Corporation, "Standard SQL Relational Database Language guide and Reference Manual", Borrisis Musteate and Robert Lesser TLM Inc, 1988

4. Alligator Descartes and Tim Bunce, "Programming the Perl DBI" Cambridge, MA : O'Reilly, 2000

5. Nancy Walsh, "Learning Perl/TK", Beijing ; Cambridge : O'Reilly, c1999

6. Microsoft Corporation, 2000, "Microsoft Servers - English Query", http://www.microsoft.com/sql/productinfo/english.htm

7. Microsoft Corporation, April 1998, "Add Natural Language Search Capabilities to Your Site with English Query, MIND April 1998", http://msdn.microsoft.com/library/periodic/period98/equery.htm

8. EasyAsk Inc, "EasyAsk Product features", http://www.easyask.com/pages/pdf/ea_prod_featurelist.pdf , April 2000

9. Damian Conway, "The man(1) of descent", The Perl Journal, Issue 12, Vol. 3, No. 4, Winter 1998, pg 46-58

10. Damian Conway, "Practical Parsing with Parse::RecDescent" Perl Conference 3.0, 1999

* SQGNL : SQL Query Generator for Natural Language