Relational Algebra Interpreter

Tamim Alkhalifah
School Of Computer Science, Engineering & Mathematics Flinders University
Adelaide, Australia
Alkh0065@flinders.edu.au

Denise de Vries
School Of Computer Science, Engineering & Mathematics Flinders University
Adelaide, Australia
Denise.deVries@flinders.edu.au

***Abstract*— Relational Algebra is a procedural language that defines database in terms of algebraic expressions. It is used to explain query execution and optimization in a relational DBMS. However, the tools available to teach the concepts of Relational Algebra are limited. Most of the tools that teach database concepts are concentrated around SQL. In this paper, we introduce an application that implements five different relational operators (select, project, union, intersect and difference) by using Irony technology. This .NET web is a tool that transforms a Relational Algebraic expression into SQL queries. The resultant SQL then is used to query a database.**

Keywords-component; Relational Algebra; SQL; e-Learning, educational software

I. INTRODUCTION

The relational model uses the concept of a mathematical relation as its basic building block. Introduced by Codd [3], it is considered to be the most common data model used for database management. The database is represented as a relational model that consists of a collection of relations where each relation resembles a table of values. There are two types of query languages, procedural and non-procedural [2], we can distinguish between them by how they obtain their results. In the procedural approach the user informs the system to perform a specific sequence of tasks on the database, while in the non-procedural approach the user indicates the desired information but not how it should be done. Most current relational databases combine both procedural and non-procedural approaches. Relational Algebra (RA), a procedural language, is used to manipulate relations and specify queries. They can work on one or more relations to define another relation. The output of one expression can become the input to another. A study of database curriculum by Robbert et al. [10] indicates that RA is considered to be a main topic covered in undergraduate computer science database courses. Many textbooks that cover the relational database model have a section or perhaps a whole chapter dedicated to RA. Therefore we strongly believe that RA is an integral topic for a database course. Learning RA makes the students aware of conceptual and practical differences between procedural and non-procedural query languages. Although the RA is important in database courses, there are few applications that support the language implementation. Moreover, most of the teaching tools available do not provide the equivalent SQL expression, which we believe would make it easier for students to learn the RA Language. This paper presents a software tool that executes the RA operators. The user is able to apply a RA expression in order to execute a specific operator in one or more tables, and, the equivalent SQL expression is displayed.

II. INTERPRETER DEFINTION (BACKGROUND)

An interpreter is a language translator that performs the operations implied by the source program. Interpreters are also often used in educational and software development situation, where programs are likely to be translated and retranslated many times. [12]

An interpreter may be a program that either:

1- Executes the source code directly.
2- Translates source code into some efficient intermediate representation (code) and immediately executes this.
3- Explicitly executes stored precompiled code made by a compiler which is part of the interpreter system. [9]

The phases of an interpreter are:

- Lexical analysis
- Syntax analysis
- Code generation
- Execution
- Error Handling

A. Lexical Analysis

Lexical analysis is also known as "the scanning process", it is a phase of the interpreter that is responsible of reading the source program as a file of characters and dividing it up into tokens (logical units). Each token consists of a sequence of characters that represents a unit of information in the source program.

Programming languages tokens tend to fall into several categories, in most languages the scanner needs to generate one token at a time (known as single symbol lookahead). Therefore, a single global variable can be used to hold the token information. In different cases an array of tokens might be needed. Table 1 contains typical examples of the tokens. [5].

TABLE I. EXAMPLES OF TOKENS

Example	Description
Keywords	fixed string of letters, (if, else, while, etc.)
Identifiers	Letters and numbers (beginning with a letter) (x, y, average, etc.)
Special symbols	+, *, >= and <>
Integer constants	(42, 0xFF, 0177 etc.)
Floating point constants	(5.6, 3.6e8, etc.)
String constants	("hello there\n", etc.)
Comments	(To be ignored.)

Each token as a logical entity must be distinguished from each string of characters they represent, for example the reserved word IF must be distinguished from the string of two characters "if" that it represents. "String value" is the definition of representing the string of two characters by a token (also known as lexeme), some tokens only have one string value, and some can represent potentially many string values. However, identifiers, despite the fact that they are represented by a single token ID, also have many different string values representing their individual names. In addition, the lexical analyser phase must handle invalid text, for example:

- A number may be too large or incomplete, (e.g. 5., 5e, etc).
- A string or an identifier may be too long.
- Invalid special symbol. Such as "=="(assuming that we didn't define this symbol before). Missing the end of the comment, and the final quote on a string.

B. Syntactic Analysis (Parsing)

This phase of the interpreter defines the syntax, structure of the program and explicitly constructs a parse tree (usually defined as dynamic data structure) or syntax tree that represents the structure. The construction of the parse tree relies completely on the syntactic structure of the language.

Usually the syntax of any programing language is given by the grammar rules of a context-free grammar, which uses similar naming conventions and operations to regular expressions. More importantly, rules in the context-free grammar are recursive, a typical example is the use of **if – else** statement: it should allow other statements to be nested inside it. By using context-free grammar, recognizing the class of structures is increased dramatically over those in regular expressions. Moreover, the algorithms used in the scanning process are different from those in the parsing. Also in this process, the representation of the data structure is recursive rather than linear (for tokens and lexemes). However, the parsing process is more complicated than the scanning process especially in terms of fixing and handling the errors. In the previous stage (lexical analysis), if a character is discovered that is illegal within the rules, then it is easy to generate an error token and consume the offending character. On the contrary, in the parsing process, in addition to reporting the errors, it must recover from errors and continue parsing. There are two different parsing techniques, top-down and bottom-up parsing, both of which use their own techniques and methods.

C. Specification of a Context-Free Grammar Rule

Context-free grammar rules determine the set of syntactically legal strings of tokens. A context-free grammar rule (from an alphabet), contains a string of symbols. The first symbol is the name of the structure, while the second symbol is "::=" (double-colon-equals), or the metasymbol "→", or "=", or ":". This is followed by a string of symbols which might be a name of the structure, or the metasymbol "|", or any symbol from the alphabet.

For example, the rule Expr ::= ProjExpr | SelectExpr | UnionExpr | DifferenceExpr | IntersectExpr | JoinExpr | ProductExpr | DivideExpr; will be interpreted as follows. The name of the rule, before the "::=", which defines the structure, and on the right-hand side, the structure contains multiple options separated with the "|" symbol (either project, select, union, difference, intersect, join, product and divide). Each one of the options contains another rule. For example, ProjExpr ::= project AttrList over (Expr). The sequence of symbols and structure names within each option defines the layout of the structure. For instance:

- UnionExpr ::= (Expr UNION Expr);
- MinusExpr ::= (Expr MINUS Expr);
- IntersectExpr ::= (Expr INTERSECT Expr);
- ProductExpr ::= (Expr PRODUCT Expr);

D. Parse Trees

A parse tree is an (ordered, rooted) tree in which the interior nodes are labelled by nonterminals, and the children of each node represent the replacement of the associated nonterminal in one step of the derivation, while the leaf nodes are labelled by terminals. [1]

To illustrate a simple example [8], the following structure:
select (S) where status > 20 giving R
project (R) over SNAME giving RESULT

corresponds to the following parse tree as shown in Figure 1:

E. Code generation

This phase of the interpreter takes the intermediate code and generates code for the target machine. In this application, the generated code is SQL code.

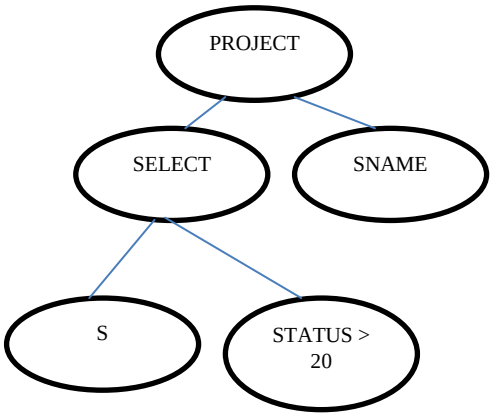

Figure 1: Tree representation of RA expression (select (s) where status >20 giving R) Project R over SNAME giving Result

III. OVERVIEW OF IRONY (.NET LANGUAGE IMPLEMENTATION KIT)

Irony, developed by Roman Ivanstov [4], is a language implementation kit for implementation on a .NET platform. It differs from tools such Lex and Yacc [7] as it does not employ any scanner or parser code generation from grammar specifications written in a specialized meta-language. It is coded directly in C# by using operator overloading to express grammar constructs. Irony's modules (scanner and parser) use the grammar encoded as C# class to control the parsing process.

IV. SYSTEM DESIGN

Figure 2 illustrates an overview of the system. A system user should go through the following process:

- Input is received from the user.
- Scanning and parsing the input and dividing it into meaningful "tokens" and checking if the input matches the defined grammar rules in the (RACGrammar) class.
- Error reporting, if there is a problem with the syntax entered by the user.
- Assuming that there is no error found, the input is converted into Structured Query Language.
- After converting the Relational Algebra query into SQL, the query is executed on a RDBMS.

V. SYSTEM IMPLEMENTATION

In this section, the implementation of the project is described and reviewed.

A. Overview of the Implementation

In the implementation process, different classes are created. Most of these classes deal with the non-terminals. Further, each class extends the base class in Irony "ASTNODE" and it is called at the appropriate non-terminal in the tree, so when a non-terminal is found, it calls the class object that returns the piece of code for that non-terminal.

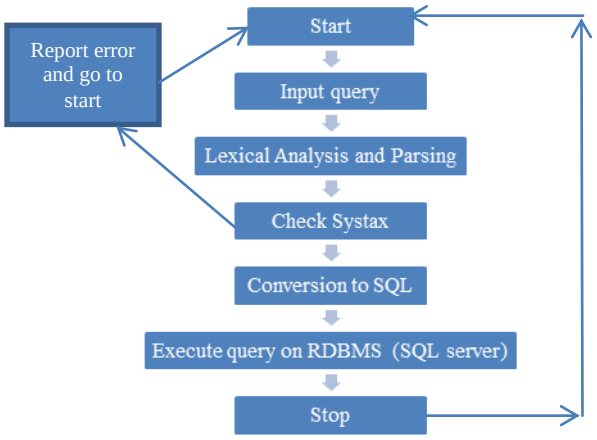

Figure 2: System Overview

The class "CaMainNode" is the base class from which all the other classes are derived. This class contains one virtual method named "GenerateSQL" that all classes extend to produce the required SQL code. This method calls a static method of a helper class named "GenerateNodes" for each non-terminal. The class "IcaMainNode" is an interface that all child classes implement. Subsequently, there are classes according to the nodes hierarchy. The hierarchy is as follows:

- A program will consist of one or more lines,
- A line will consist of a statement list,
- A statement list will contain statements,
- A statement will consist of a conditional or column expression,
- A conditional expression will consist of either a single expression or list of expressions,
- An expression will be either a variable or binary expression,
- A binary expression will consist of an operand, operator and operand style,
- A column expression will consist of a list of arguments,
- And arguments will be variable names of columns.

Thus, a class is created for each non-terminal in the hierarchy. Each class when called by the parse tree returns the desired code.

B. Defining the Terminals, Non-Terminals and the Grammar

Before defining the grammar for the language, we have defined two important concepts (terminals and non-terminals). Terminals are elements with which the sentence of the language may be constructed. A terminal symbol is one which is atomic. In the implementation process of the terminals, a "region" is created to hold and define the terminals. For example:

```
#region Define Terminals
Terminal Selection = Symbol("select");
Terminal projection = Symbol("project");
```

However, the non-terminals are elements which are only used in the derivation of a sentence. In other words, the non-terminals have to be replaced by even more specific components, either terminals or other non-terminals [6]. Similarly, another "region" is created which holds and defines the non-terminals. For example:

```
#region Define NonTerminals
NonTerminal Program = new
   NonTerminal("Program",
   typeof(ProgramNode));
NonTerminal Line = new
   NonTerminal("Line", typeof(LineNode));
```

C. Defining the grammer of the language:

After defining the terminals and non-terminals it is it possible to start defining the grammar rules. The grammar of the language that is used in this project is declared inside the class "RACGrammar.cs" and the namespace "Irony.Compiler" has been included. Each statement the user applies to the textbox is considered as a program (which is the top non-terminal).

```
   this.Root = Program;
```

The program can consist of one or more lines, rules define such things, for example:

```
Program.Rule= MakePlusRule(Program, null,
   Line);
```

This defines that a program that might contain one or more lines.

```
STATEMENT.Rule = Stmt + "where" +
   ConditionExp | Stmt + "over" +
   ColumnExp;

ConditionExp.Rule= EXPR | EXPR_LIST;

EXPR.Rule = Number | VarExpr |
   BINARY_EXPR;

BINARY_EXPR.Rule = EXPR + BINARY_OP +
   EXPR;

BINARY_OP.Rule = Symbol("=") | "<=" |
   ">=" | "<" | ">" | "<>" | "or" | "and";

RegisterOperators(30, "=", "<=", ">=",
   "<", ">", "<>");

RegisterOperators(20, "and", "or");
```

Statements can consist of two types of expressions (ConditionExp and ColumnExp); one for the SELECT statement and other for the PROJECT statement. If the expression is conditional then it will consist of one or more expressions and an expression can be a variable, a number or a binary expression. The binary expression can have two operators and one operand. After that, the binary operators and symbols are registered. The same concept is applied to ColmnExp expression:

```
ARG_LIST.Rule= MakePlusRule(ARG_LIST,
   comma, VarExpr);

Stmt.Rule = LP + TName + RP;

TName.Rule = VarExpr;

RegisterPunctuation("(", ")", ",",
   "select", "project", "'","over",
   "where");
```

The ARG_LIST rule contains a comma to separate a list of column names (e.g. PROJECT SPJ over sname, snumber). Finally, the table name and punctuation are defined. Stmt represents the table name, LP is left parenthesis, and RP is the right parenthesis.

D. The SQL algorithm structure:

1) Selection operator: The following algorithm is used to construct the SQL statement for the selection operator: (sample code)

```
string[] tokens = new string[10];
tokens = s.Split(',');
string[] SQL = input.Split(')');
string tablename = SQL[0].Replace("(", "
   ");
String SQLSTMY = "SELECT";
if  (txtinput.Text.Contains("select"))
    {
      SQLSTMT += " " + "*" + " " +
   "FROM"+ tablename + " " + "WHERE";
for (int j = 3; j <= tokens.Length - 3;
   j++)
    if (j == 3 && tokens.Length <= 6)
       {
SQLSTMT+=" " + tokens[j].Replace("'", "
   ") + "" +   tokens[j + 1];
      }
```

The split method takes an array of chars that indicate which characters are to be used as delimiters, in our case the character is a ",".The FOR loop is used to construct the SQL statement for the selection operator from the number of tokens returned by the SQL generator method. This method builds the SQL based on the number of tokens. As shown in the previous code segment, the elements are placed in such a way to generate an appropriate SQL expression by looping through the tokens and assign it to "SQLSTMT".

2) Projection Operator:

```
String SQLSTMY = "SELECT";
if (txtinput.Text.Contains("project"))
    {
     for (int i = 3; i < tokens.Length -
         1; i++)
     {
        if (i > 3)
        {
          SQLSTMT += "," +
  tokens[i].Replace("'", " ");
        }
         else
        {
           SQLSTMT += " " +
  tokens[i].Replace("'", " ");
        }
     }
     SQLSTMT += " " + "FROM" + tablename;
}
  txtSQL.Text = SQLSTMT;
```

The same concept that applies to the selection operator applies to the projection operator. As Ranjan and Litoriya indicated in [9], the projection operator should be processed as follows:

- Replacing the "project" token with "select".
- Add the word "from" after naming the attributes.

In the above sample code, the code written in bold has explicitly converted the project operator to the select operator (hence SQLSTMT = SELECT). After that, the variable SQLSTMT is passed to the textbox of the equivalent SQL which has the id name (txtSQL). The variable SQLSTMT is then passed to the SQL connection.

3) Union, Intersect And Difference Operators:

The three operators share the same concept in terms of converting to SQL equivalence. For example the union operator has been implemented as follows:

```
if (input.Contains("union"))
    {
       string [] token = input.Split('
  ');
       string sql = "(Select * FROM " +
  token[0]+ ")" + " union" + "(Select *
  FROM " + token[2] + ")";
     txtSQL.Text = sql;
     }
```

Similarly, for the intersection operator:

```
else if (input.Contains("intersect"))
                    {
              string[] token1 =
  input.Split(' ');
              string sql = "(Select *
  FROM " + token1[0] + ")" + " intersect
  " + "( Select * FROM " + token1[2] +
  ")";
              txtSQL.Text = sql;
          }
```

and finally the difference operator:

```
else if(input.Contains("difference"))
           {
              string[] token1 =
  input.Split(' ');
              string sql = "(Select *
  FROM " + token1[0] + ")" + " Except "
  + "( Select * FROM " + token1[2] + ")";
              txtSQL.Text = sql;
           }
```

Hence, the three operators share the same concept. An array of string is defined. The user's input is split and assigned to the array. The user applies the three operators by typing the following: <table name> <operator name> <table name>. The first table name corresponds to token[0] while the <operator name> corresponds to token[1] and the second table name corresponds to token[2]. This explains the order of tokens written in the SQL statement which is assigned to the variable sql.

VI. DATABASE CONNECTION

In the implementation phase, The SPJ database was created. It has four relational tables (Stable, Ptable, Jtable and SPJ). The namespace `System.Data.SqlClient` is the .NET provider for dealing with the SQL server. The user can add new databases, columns, rows as well as insert, deleted and modify the relational tables. All of which can be done in the server explorer section.

In order to connect to the database the first step involves creating a connection string which acts as 'proxy' for database operations, in this example to Microsoft SQL server. The variable strSQLconnection holds the following connection string value:

```
"DataSource=TAMIMPC\\SQLSERVERR2;Init
iaCatalog=SPJ;Integrated Security=True"
```

After that SqlConnection is used in conjunction with Sqlcommand to increase the performance when connecting to a SQL server. A new instance of Sqlconnection (named "sqlconnection") is given the connection string value. Similarly a new instance of Sqlcommand (named "sqlcommand") is created which holds the variable sql (containing the SQL query) and the sqlconnection instance. Then using the ExecuteReader(); method which returns an instance of a SqlDataReader. Any object from SqlDataReader can be read in a forward-only sequential manner. Below is a sample code used to connect to the SPJ database:

```
string strSQLconnection = "Data Source
  =TAMIM-     PC\\SQLSERVERR2;     Initial
  Catalog=SPJ;Integrated Security=True";
```

```
SqlConnection sqlConnection = new
  SqlConnection( strSQLconnection );
SqlCommand sqlCommand = new
  SqlCommand(sql , sqlConnection);
sqlConnection.Open();
sqlDataReader reader =
  sqlCommand.ExcuteReader();
GridView1.DataSource = reader;
GridView1.DataBind();
GridView1.Visible= true;
sqlConnection.Close();
```

VII. Recommendations For Future Enhancements

Despite the fact that this application has many good features, it is missing a number of features which could be added to boost the learning process of the language. The following features would enhance the system:

A. Missing operators :the application has implemented five different relational operators (i.e. select, project, union, intersect and difference), but can be improved by implementing more operators such as (product, join, division). Furthermore, implementing the optional operators would boost the learning process.

B. Error handling: The error handling mechanism that has been used in this application is basic. Only a general error message if the syntax does not match the grammar rules. This can be improved by generating more specific error messages.

C. The user interface: The GUI used for application can be improved. It is possible to add a .CSS file (cascading style sheet) to style web pages written in HTML. Also, by adding a drop down list that includes a history of written queries to which users can refer.

D. Nesting support: This application only supports queries with one level of nesting. This can be improved by establishing new classes to maintain nesting support.

VIII. Conclusion

We consider the primary achievements of the project to be:

- Establishing a web application that can execute five Relational Algebra operators.
- The use of Irony technology to identify the terminals, non-terminals and the grammar.
- Displaying the equivalent SQL query in a separate textbox.
- Using the .NET framework for creating a powerful and user friendly interface.
- Displaying the results from a database (SQL

This web application is written in Visual C#.NET by using Visual Studio 2010. It is intended for those who want to learn Relational Algebra. We have implemented a system which allows users to compute Relational Algebra queries by transforming them to SQL,. then applying the SQL on a Microsoft SQL server.

References

Chiswell, I., Hodges W., (2007). *Mathematical logic*. USA: Oxford University Press.

Date C., (2004*). An Introduction to Database Systems*. 8th edition, USA: Addison-Wesley.

Codd, E. F., (1972). *Relational completeness of data base sublanguages*. In R. Rustin (Ed.), *Data base systems*. Courant Computer Science Series 6. Prentice Hall.

Ivantsov, R., (2011). *Irony - .NET Language Implementation Kit* [online] available at: http://irony.codeplex.com [Accessed 1/12/2013]

Klein, K., (2009). *JFLEX: The Fast Lexical Analyser Generator* [online] available at: http://www.jflex.de/manual.pdf

Levelt, W., (2008). *An Introduction to the Theory of Formal Languages and Automata*. USA: John Benjamins Publishing Company.

Levine, J., Manson, T., Brown, D., (1995) *Lex & Yacc*. 3rd edition, USA: O'Reilly & Associates, Inc.

Louden, K., (1997). *Compiler Construction: principles and practice*. USA: PWS Publishing Company.

Ranjan, A., Litoriya R., (2011). *Relational Algebra interpreter in context of query languages*. [online] available at http://www.ijcte.org/papers/276-D014.pdf [Accessed 25/010/2013]

Robbert, M., Ricardo, C.M. (2003). *Trends in the evolution of the Database Curriculum. In: Innovation and Technology in Computer Science Education* ITiCSE, pp. 139–143

Silberschatz, A., Korth, H. and Sudarshan, S., (1997). *Database System Concepts*, 3rd edition, USA: McGraw Hill.

Sommerville, I., (2007) software engineering. 8th edition, USA: Pearson Education Limited.

Walther, S., (2004). *ASP.NET Unleashed*, 2nd edition, USA: Sams publishing.

Williams, M., (2002). *Microsoft Visual C#.NET*, USA: Microsoft Corporation.