

RASQL

Relational Algebra and SQL

An application to work with Relational Algebra and SQL queries



Mangesh Bendre (bendre1)

Nilam Sharma (sharma30)

4/22/2010

University of Illinois

Abstract

Relational algebra is a query language that is being used to explain basic relational operations and their principles. Many books and articles are concerned with the theory of relational algebra; however, there are few available tools that allow the students to have hands on experience with the relational algebra queries. Most of the currently used relational database management systems work with SQL queries. This work therefore describes and implements a tool that transforms relational algebra expressions into SQL queries, allowing the expressions to be evaluated in standard databases. This also allows the user to integrate the queries to various DBMS engines and have direct interaction with the database using relational algebra queries.

Contents

1. Overview.....	4
1.1. Introduction:.....	4
1.2. Proposal	4
1.3. Relational Algebra Translator:.....	6
2. The Front-End Application:	8
2.1. The First Look:.....	8
2.2. Major Components:.....	9
2.2.1. Editor:	9
2.2.2. The Relation Tree:.....	9
2.2.3. Data Grid:	10
2.2.4. SQL Query:	10
2.2.5. Trace:	11
2.2.6. Menu Options:	11
2.2.7. The Operators:	12
2.3. The Associated File Types:.....	14
2.4. Database Connectivity:	15
2.5. SQL Parsing:	16
2.5.1. Parsing Flow:.....	16
2.5.2. Normal Operation Syntaxes:	18
2.5.3. Renaming Operation Syntaxes:	19
2.6. Executing Queries:	20
2.7. Features:	24
2.7.1. Single Line Comments	24
2.7.2. Highlighting:.....	24
2.8. A Sample Workflow:.....	25
3. The Back-End Parser.....	27
3.1. The RASQL Parser:.....	27
3.2. Flow of the Parser:.....	27
3.3. Symbol Processing Logic:.....	29
3.4. Schema aware processing:.....	29
3.5. Query Optimization:	30
4. The API / Command Line Interface:	31
4.1. Using the APIs:	31
4.2. Plain Text Query Format:	32
5. Architecture:	33
5.1. Parser Tool:	33
5.2. DB Engines:.....	34
6. Conclusion:	35

1. Overview

1.1. Introduction:

RASQL is an interactive tool to evaluate relational algebra queries. The application can be mainly used to:

1. Write relational algebra queries with the common Relational Algebra operators – basic (select, project, union, difference, Cartesian product, rename) and derived (intersection, natural join, theta join), forming complex RA queries,
2. Parse the RA queries into SQL queries.
3. Executing the Relation Algebra against a DB with results.

1.2. Proposal

Relational algebra is a query language that is being used to explain basic relational operations and their principles. Many books and articles are concerned with the theory of relational algebra; however, there are few available tools that allow the students to have hands on experience with the relational algebra queries. Most of the currently used relational database management systems work with SQL queries. This project therefore describes and implements a tool that transforms relational algebra expressions into SQL queries, allowing the expressions to be evaluated in standard databases. This also allows the user to integrate the queries to various DBMS engines and have direct interaction with the database using relational algebra queries.

We as part of our DBMS course strongly felt the need of a end-to-end application that can help students with learning relational algebra. After the various requirement analysis and brainstorming sessions, we have concluded the following as the “must have” requirements:

1. **User Friendly Editor:** It should have a front end where the user will be able to write the RA queries without much effort.
2. **RA to SQL parsing:** The basic need of the application is RA to SQL parsing. Provided single/multiple SQL queries, the application should be able to return an SQL query with same output.
3. **Database Connectivity:** The application should have connectivity to a backend DBMS where the user can define his relational schema and can directly integrate the relational queries he is working on. Besides, it should offer seamless execution of the queries against the DB and display the outputs.

Along with these, we also looked for requirements that will help the user to effortlessly work and understand the concepts of relational algebra. The work was targeted to an audience mostly comprising of students and instructors and the following “nice to have” features, we thought will make their life lot easier while dealing with relational algebra:

1. Support for Unicode Greek symbols:

Which one of the following makes more sense to a student – a fully formatted relational algebra query with Greek Symbols and conditions in suffix like $\pi_{name, address}(Employee)$ or a plain text query like PROJECTION{name, address} (Employee)? Definitely it is the one with actual relational operators. Instead of writing the queries in plain text (and yes, it involves lot of more typing too!), we wanted to provide the user an editor where he can type the queries exactly he learns relational algebra i.e. with full Unicode Greek symbols.

2. Multiple Database:

You work in SQL Lite? That is fine. You like MySQL? Oh, we can support that too!!! By allowing the user to choose his/her own DB implementation, we can make the tool more versatile and allow the user to concentrate on the RA queries only and not on the underlying database.

3. Integrated support for SQL:

Consider the following scenario-

The user has just finished an RA query and then to test it, he needs the corresponding relation in the database. He wants to see the output and for that he needs the data in the relations. He defined a column to support 20 characters and now he needs to enter a 30 character string.

Why should the user go to toad, SQL Lite or MySQL prompt to make all these DB changes? As long as the DB supports the commands entered by the user as valid SQL commands, the application should allow the user to write all the DDL and DML queries right in the application editor.

4. Save/Load:

Many a times, the user needs to save his queries for reuse. The application should have in build functionality for saving the queries entered by the user with all the details including the Unicode operator symbols so that they can be used again. Also, support for comments will be helpful.

5. Assignment:

Although assignment of relation is not part of standard relational algebra, it is something that is used extensively by the user community while working with complex RA queries. The application should provide an assignment mechanism which allows the user to save part of the RA queries in a temporary relation and use the temporary relation in the following queries.

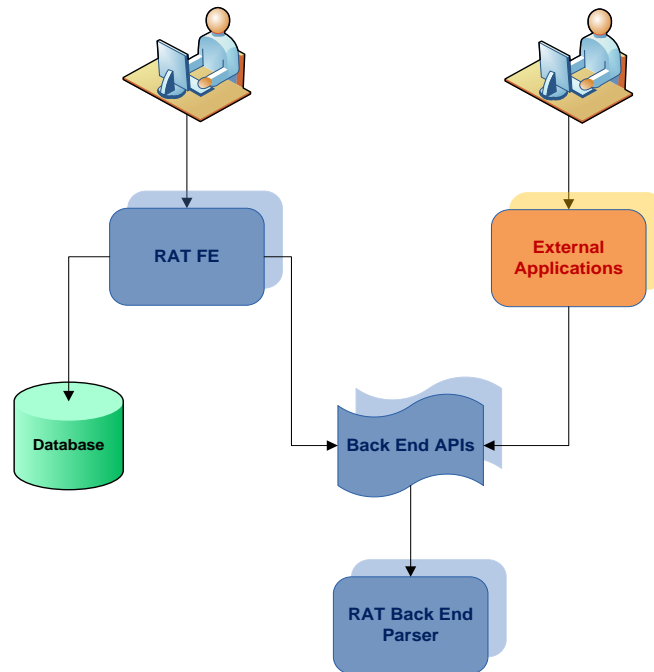
The output is the RASQL application. It includes all the above functionalities and more. Built on the robust QT framework and Linux platform, it is capable of handling SQL Lite and MySQL DBs as backend. We tried our best to incorporate an environment where the user can work on the queries effortlessly and efficiently and have fun with learning relational algebra.

1.3. Relational Algebra Translator:

The RASQL system consists of three major components:

- i. RASQL Front End Application
- ii. RASQL Back End Engine
- iii. RASQL APIs

The diagram below shows the main need of the components and how the various components interact with each other.



The RASQL Back End Parser is the back bone of the system. As the name suggests, its functionality is to parse plain relational algebra queries and return corresponding SQL queries. It is implemented in terms of static libraries in C++ which can be accessed using the APIs.

The RASQL FE application accesses the RASQL Back End Parser via the APIs and provides the user a full-fledged interface to create/load/save/parse/execute relational algebra queries. It has capability to connect to multiple DBMS drivers. The user can connect to a DBMS of his/her choice and execute the SQL queries returned by the Back-End Parser.

The APIs also work as an interface for the Back-End Parser libraries. These can be used to integrate the back-end parser to another external application.

2.2. Major Components:

The major components of the application are discussed in the following sections.

2.2.1. Editor:

The Editor is the section where the user can enter the relational algebra to be used by the application.

The Editor supports Unicode character set and thus can handle the relational algebra queries in the native format. It is very user friendly and supports most of the features in a modern text editor including cut, copy, paste, redo and undo.

The application allows the user to load/save the text present in the Editor. The application specific file extension is “.rasql”.

The user can manually type in the RA query. For the RA operators, the user can simply double-click on the operator buttons in the Operator toolbox at the left hand side of the application interface. For the relations and attributes, the relation Tree at the right hand side can be used.

The user can parse/run single or multiple RA queries using this application. In case of multiple queries, the queries should be separated by ‘;’.

The Editor also supports single line comments for the user convenience. The single line comments should start with ‘//’. No multi-line comment is supported by this application; however the user can enter multiple single line comments one after another.

The RA queries in the editor should be written strictly as per the rules used for relational algebra. Care should be taken not to mix the suffixes and brackets doing which may lead to parsing errors.

2.2.2. The Relation Tree:

On successful connection to a database, the Relation Tree section shows the relational schema of the database in a tree structure:

2.2.5. Trace:

Trace is the output tab that contains useful information about the parse/run of the current query being run by the user.

During parsing,

1. The trace shows the plain text equivalent of the RA query entered by the user in the editor.
2. If the parsing fails, it shows the error information.

During execution of the SQL query, if the query has any DB error (e.g. the relation is not present in DB), the trace shows the error information.

2.2.6. Menu Options:

1. File → New

It resets the Editor with a new “untitled.rasql” document. If the user has any unsaved changes in the current document, the application prompts the user to save his/her changes.

2. File → Open

This allows the user to load a pre-saved *.rasql file to the Editor.

3. File → Save

If the user is working on a pre-saved file, the current content of the Editor is saved back to file. Otherwise, the Save As dialog is opened.

4. File → Save As

The Save As prompt allows the user to save the Editor contents in the two formats: .rasql and .txt. They are discussed in the later sections.

5. File → Exit

Exits the application

6. Edit → Undo

Undoes the last change made by the user. The application supports unlimited undo steps.

7. Edit → Redo

If the user undoes changes made, the same changes can be reapplied by using this option.

8. Edit → Cut/Copy/Paste

These three options allow the user to cut/copy/paste the selected Editor contents to/from the clipboard.

9. Session→ Connect

Opens the Connect window that allows the user to connect to a pre-configured database or to create a new connection

10. Session→ Settings

This opens the Settings window that allows the user to create/update/delete a DB connection to RA operations.

11. Action→ Parse

Parses the RA query to an SQL query

12. Action→Run

Parses the RA query to an SQL query and on successful parsing, executes the query against the connected database.

13. Action→Stop

Not implemented

14. Action→Clean

Clears the SQL Query Tab and the Trace Tab

2.2.7. The Operators:

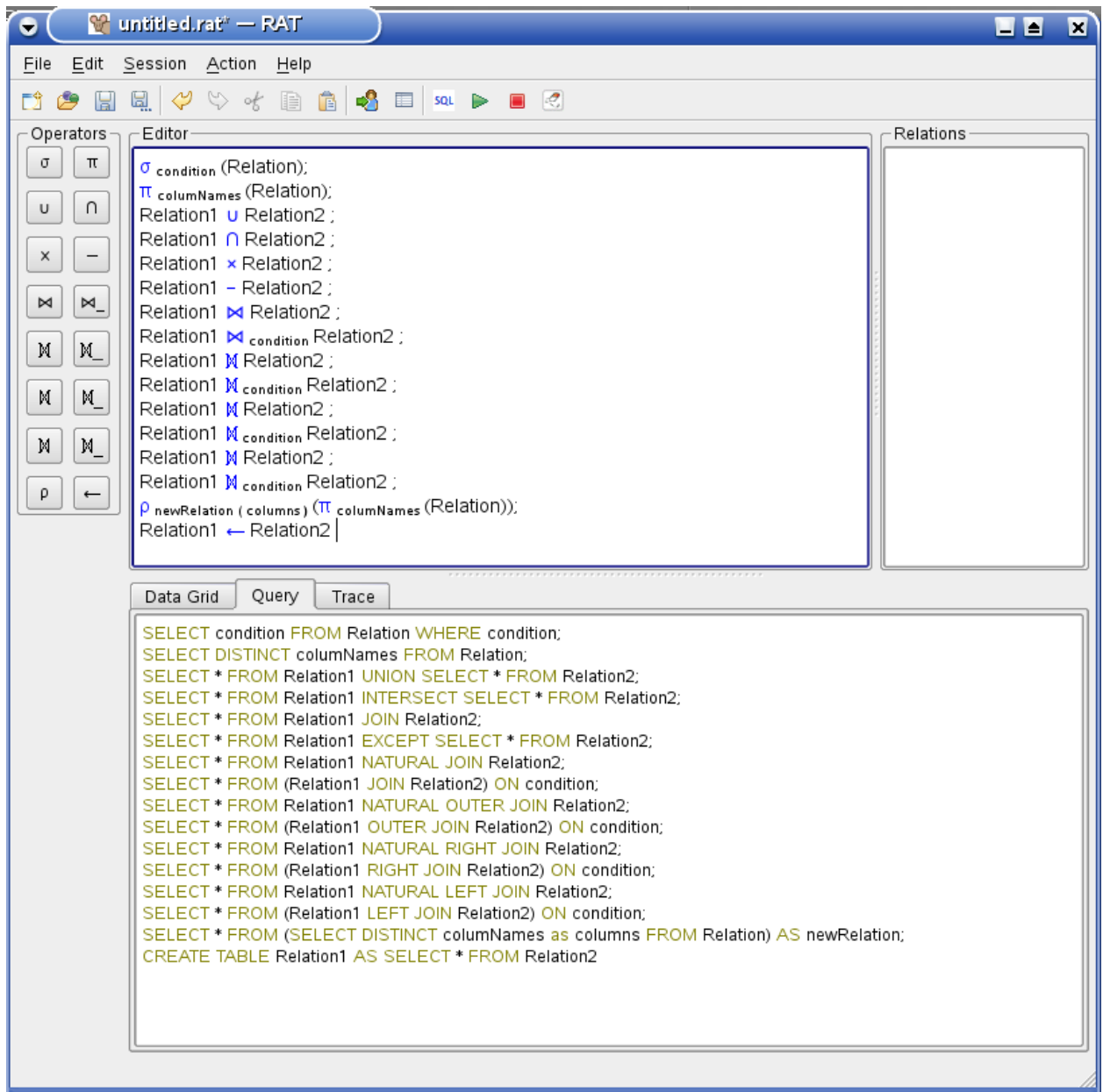
The GUI has a toolbar that allows the user to input the relational algebra operators.

The buttons will include:

- Select

- Project
- Union
- Intersection
- Cartesian
- Difference
- Natural Join
- Theta join
- Full Outer Join
- Theta Outer Join
- Natural Left Join
- Theta Left Join
- Natural Right Join
- Theta Right Join
- Rename
- Assignment

Clicking on any of this button will add the corresponding operator the editor at the current position.



If the operator has any condition or suffix, the suffix will be highlighted by default. Otherwise, the relation portion of the operation will be highlighted allowing the user to directly work on it.

2.3. The Associated File Types:

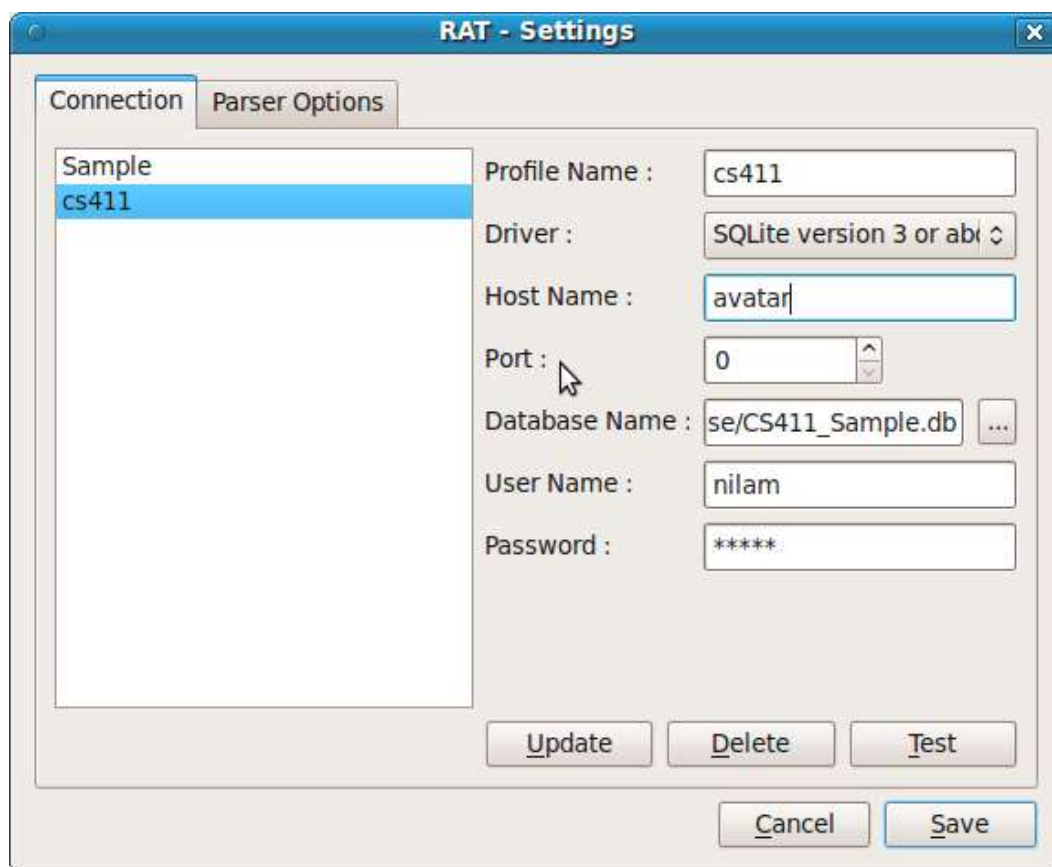
The application uses a native file type with extension .rasql to save the rich text RA queries entered by the user. Saving the queries in this format allows the user to store complete information present in the editor including the comments, which can be later loaded back to the application.

If the user wants to get a plain text representation of the RA queries present on screen, he can have it using the .txt format option. To save in .txt format, the queries must be syntactically correct. Saving in plain text mode does not save the comment information.

2.4. Database Connectivity:

RASQL allows the user to maintain multiple profiles for DB connections.

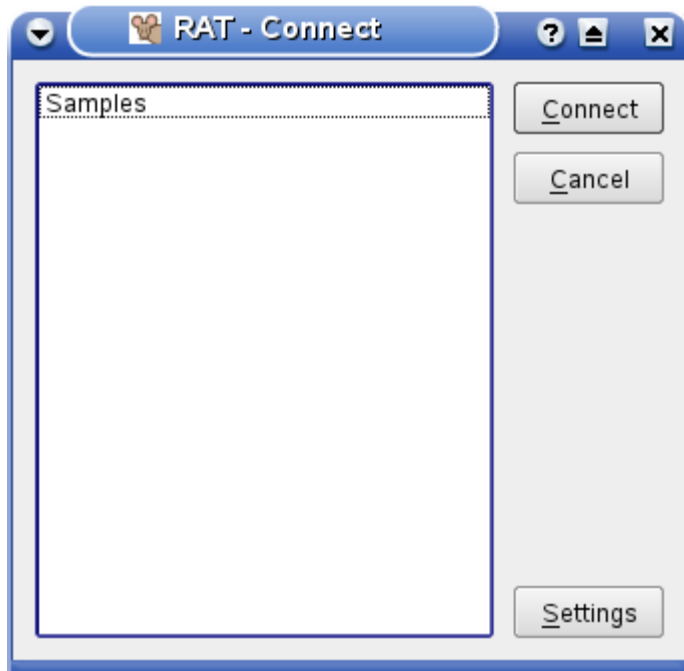
This can be done using the Settings screen that can be opened from menu option Session→Settings for the main screen.



Based on the drivers available on the host operating system, the user can select different DBMS systems for his operations.

DBMS systems like SQL Lite also allow the user to create a new DB directly from the Settings screen.

At run time, the user can choose the DBMS profile he wants to connect to using the Connect screen. The Connect screen shows a list of saved profiles available for connection. The user can choose one and connect using the Connect button.



Also, the user can open the Settings screen directly from this screen using the Settings button on this screen.

2.5. SQL Parsing:

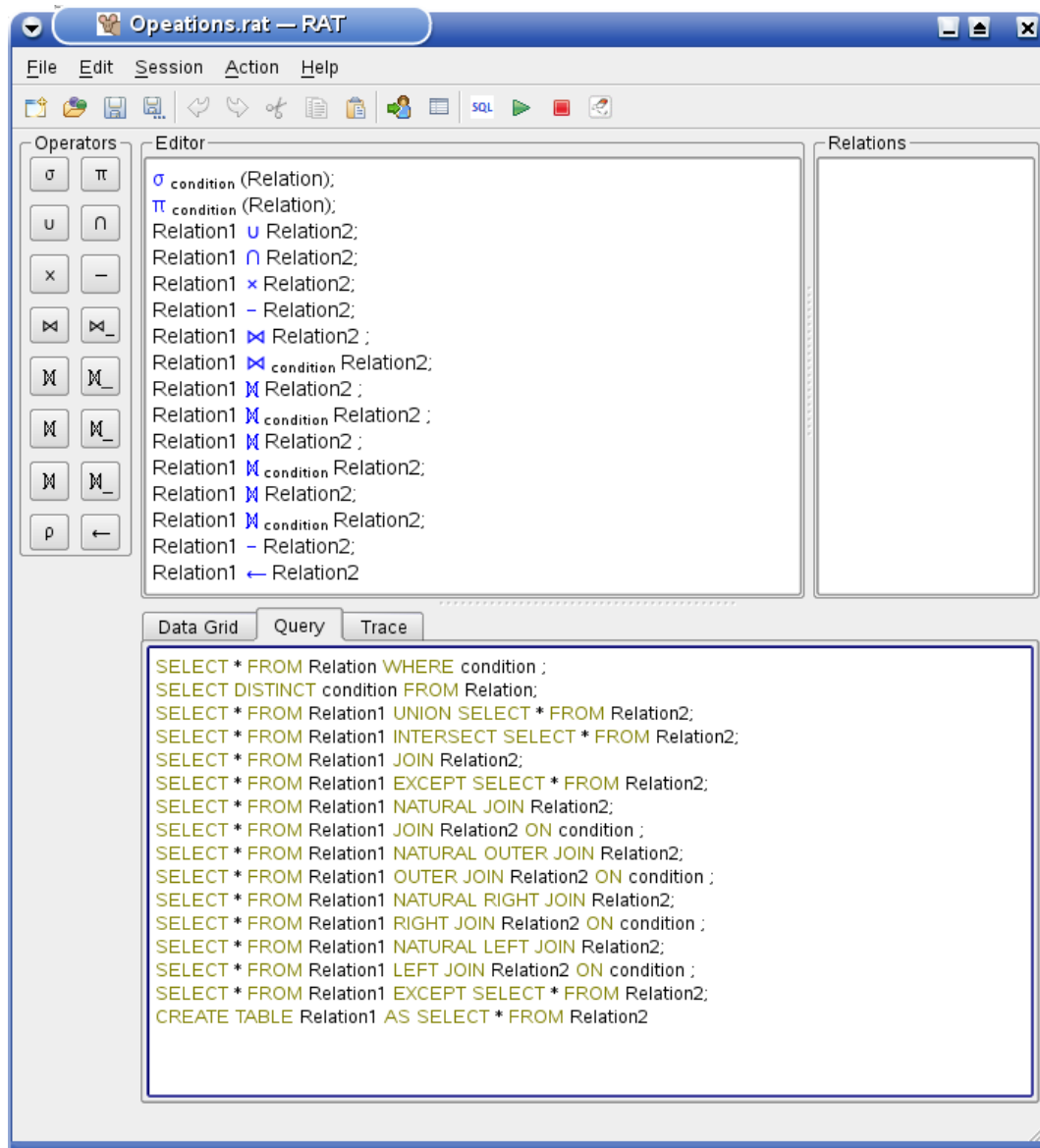
2.5.1. Parsing Flow:

Clicking on the SQL parser button on the main window parses the queries present in the Query Editor.

Depending on user selection, the parser parses the queries in two ways -

1. If no part of the text in the editor is selected, the parser parses all the text present in the query area.
2. If part of the text is selected, the parser parses only the selected text.

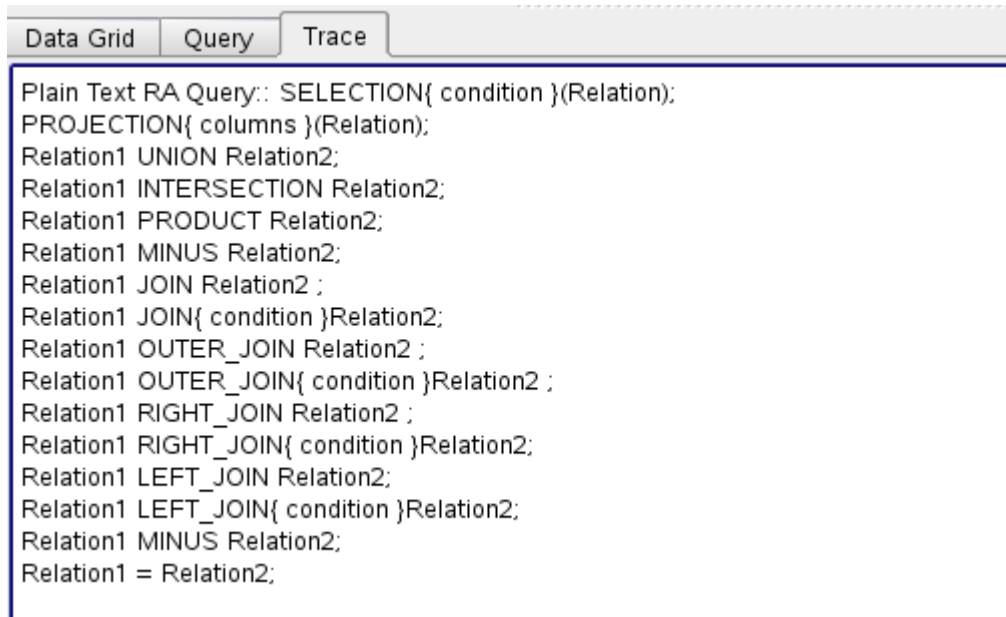
The output SQL query is displayed in the Query tab of the Output section



The SQL parsing is done in two phases:

1. **Rich Text to Plain Text:** The front end application first parses the rich text query with Unicode characters to convert it to a plain text query
2. **Plain Text to SQL:** This plain text queries are then sent to the backend parsing engine which does the actual parsing of the RA queries to corresponding SQL queries. Details about the parsing operations are explained in the later sections.

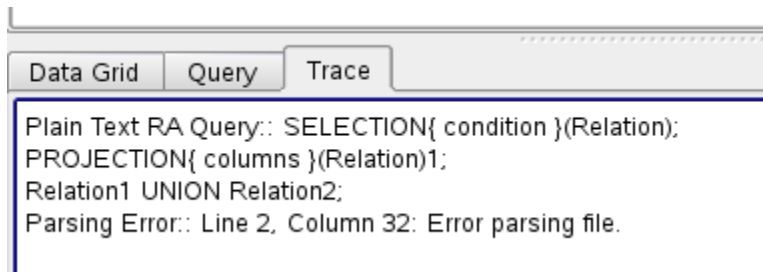
The Trace tab contains additional details about the parsing operations. E.g. It shows the plain text queries sent to the backend parser.



The screenshot shows the 'Trace' tab selected in a window with 'Data Grid', 'Query', and 'Trace' tabs. The text area displays a list of valid plain text RA query operations:

```
Plain Text RA Query:: SELECTION{ condition }(Relation);
PROJECTION{ columns }(Relation);
Relation1 UNION Relation2;
Relation1 INTERSECTION Relation2;
Relation1 PRODUCT Relation2;
Relation1 MINUS Relation2;
Relation1 JOIN Relation2 ;
Relation1 JOIN{ condition }Relation2;
Relation1 OUTER_JOIN Relation2 ;
Relation1 OUTER_JOIN{ condition }Relation2 ;
Relation1 RIGHT_JOIN Relation2 ;
Relation1 RIGHT_JOIN{ condition }Relation2;
Relation1 LEFT_JOIN Relation2;
Relation1 LEFT_JOIN{ condition }Relation2;
Relation1 MINUS Relation2;
Relation1 = Relation2;
```

In case of a parsing error, the trace tab also displays the line from the plain text input which contains the incorrect text.



The screenshot shows the 'Trace' tab with a parsing error message displayed at the end of the query list:

```
Plain Text RA Query:: SELECTION{ condition }(Relation);
PROJECTION{ columns }(Relation)1;
Relation1 UNION Relation2;
Parsing Error:: Line 2, Column 32: Error parsing file.
```

2.5.2. Normal Operation Syntaxes:

Depending on the operator type, the syntax of the added text will be as follows:

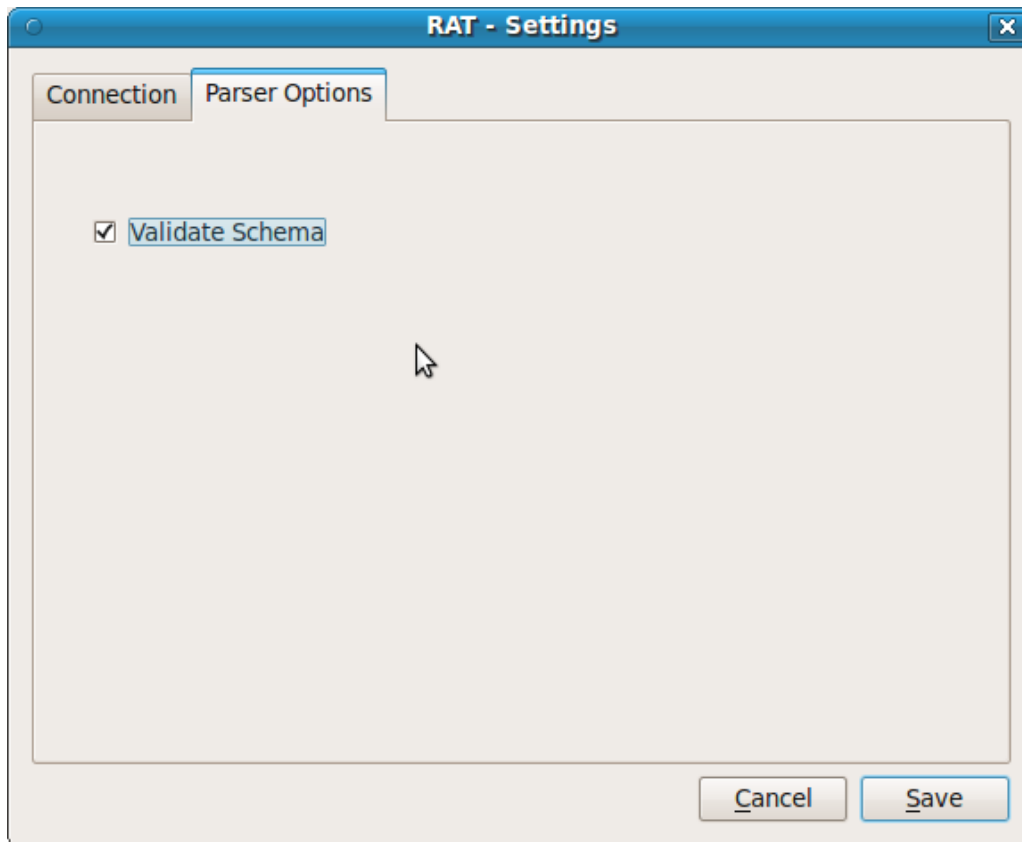
1. **Unary operator:**
The operator will be added with the pattern \rightarrow <operator><suffix><relation>
2. **Binary operator with condition**
The operator will be added with the pattern \rightarrow <relation> <operator><suffix><relation>
3. **Binary operator without condition**
The operator will be added with the pattern \rightarrow <relation> <operator><relation>

2.5.3. Renaming Operation Syntaxes:

Renaming is a special operation and needs special care by user while using in his/her queries. This is because of the following reasons:

1. When the user is parsing his queries, the relational schema information **may not be available to the parser**. This happens when the user is not connected to a backend database and so does not have a schema loaded. In such a scenario, if the user enters a rename operation without the information about the underlying relation, the parser cannot parse it. The parsing should generate an SQL query that renames the columns one by one and in this case, the parser does not know what columns to replace. So when the user is not connected to a database, it is a must to enter a projection inside the rename operation so that the parser gets the columns the user wants to rename.
2. Irrespective of whether the user is connected to the database or not, the user may want to rename the complete relation to a new one. In that case, he can simply enter the rename operation specifying the new relation name only.

RASQL allows the user to feed the relational schema information using the Validate Schema functionality.



Once the user connects to a schema, RASQL retrieves the table and column information. The user can go to the Settings->Validate Schema tab and check the Validate Schema checkbox to feed this information automatically to the parser. Doing this will allow the user to enter the rename operation without providing the column information as mentioned above.

So the syntax of rename can be of three types –

i. ρ **newRelation** (newCol1, newCol2) (π col1, col2 (**Relation**))

The above syntax can be used when the user wants to rename specific columns of a relation. As per the normal convention, the user is also needed to enter a new relation name for the selected columns.

The above syntax will be used when the user is not connected to a database or the user wants to rename specific columns from the relation only.

ii. ρ **newRelation** (newCol1, newCol2) (**Relation**)

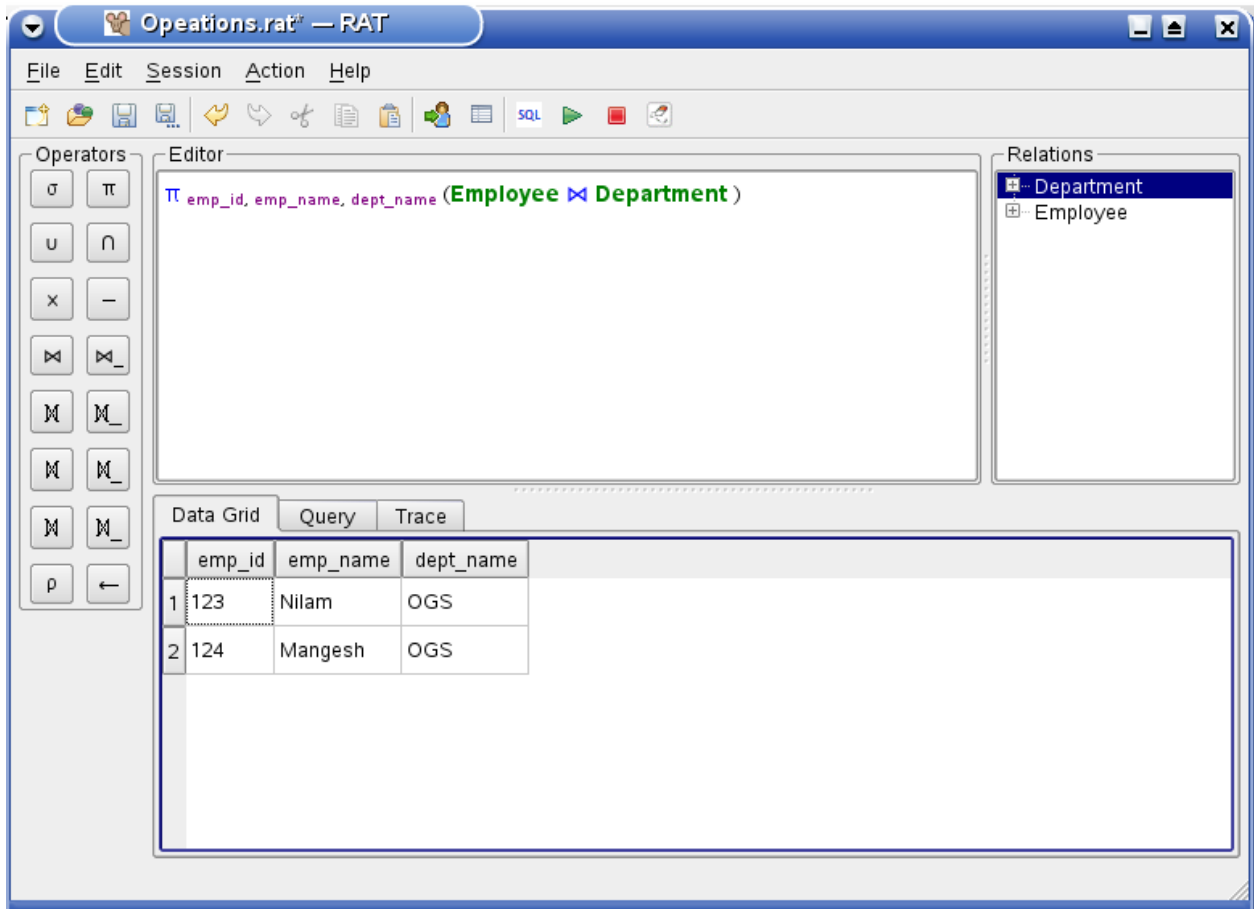
The above syntax will be used when the user is connected to a database and wants to rename all the columns from the relation. In the example above, Relation should have two columns which will be renamed as newCol1 and newCol2 respectively.

iii. ρ **newRelation** (**Relation**)

The above syntax is useful when the user wants to rename a relation to a new name. This is particularly useful when the user wants to do a self join on a relation in which case, having a different name for each of the relations is a must.

2.6. Executing Queries:

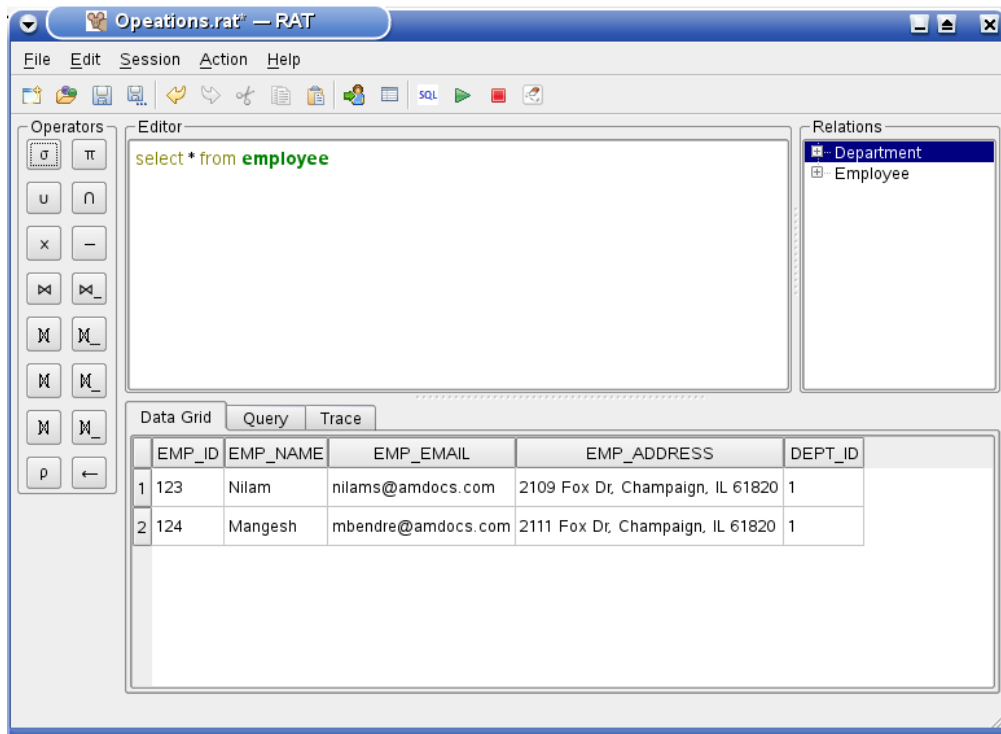
The relational queries entered by the user can be executed against a connected database using the Run button in the main screen toolbar. If the query entered by the user is a valid query, the results of the query are displayed in the Data Grid tab of the output section.



If the user is not connected to any database, clicking the run button will automatically open the Connect screen. The user will have to select and connect to a saved profile to continue.

SQL support:

The RASQL application supports simple SQL along with the RA queries. If the user does not enter a valid RA query, the system checks if the query can be executed against the connected DB as an SQL statement. In case of a valid SQL query, the output is displayed accordingly.



DDL statements like **create and delete table** are also supported which eliminates the need of a separate application to create and maintain the DB used for RA operations.

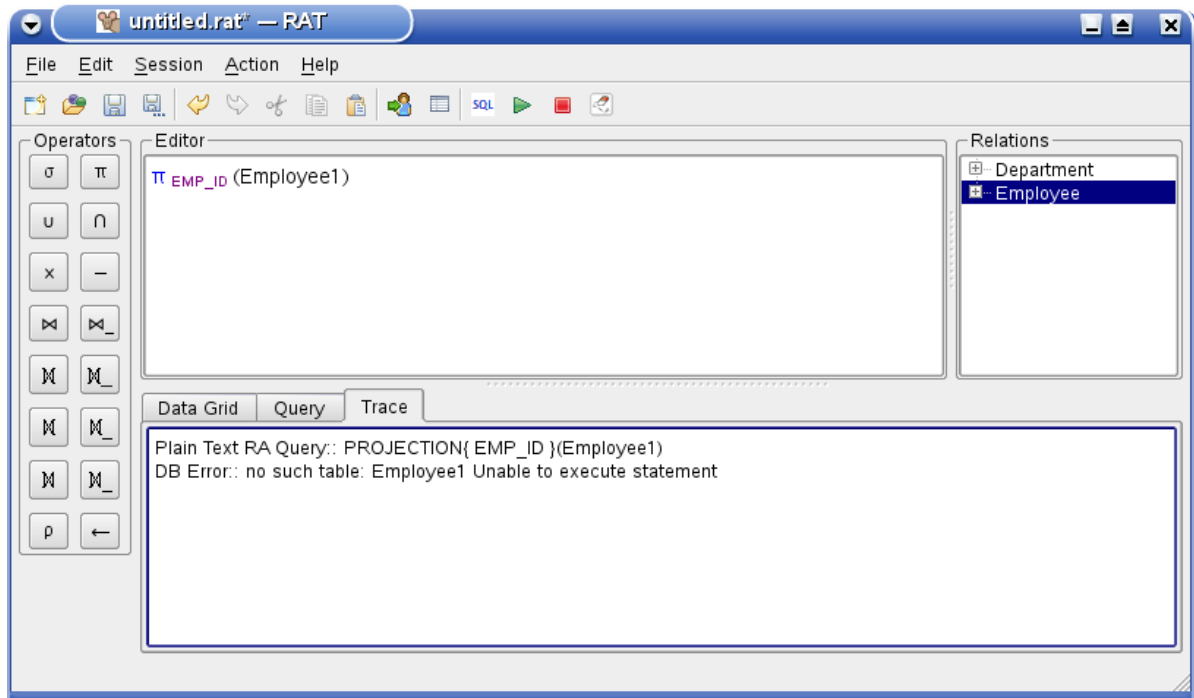
Multiple Queries:

In case of multiple queries, the queries are executed sequentially. The output of the final query is displayed in the Data Grid.

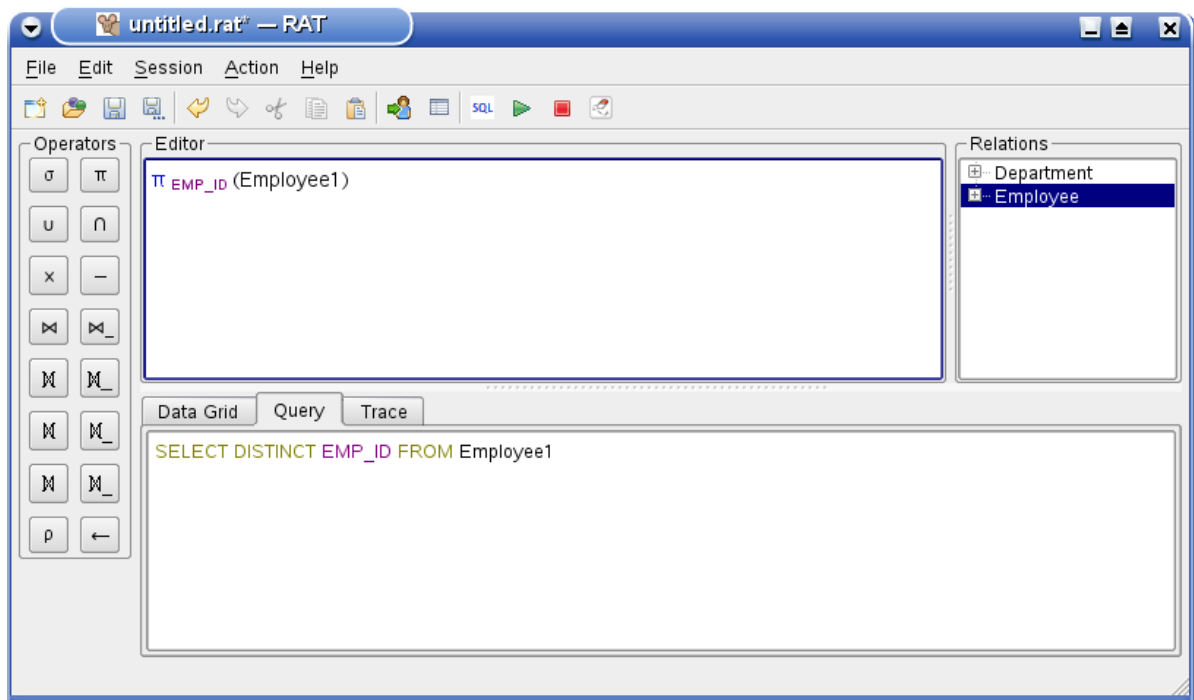
Error Scenario:

Various error scenarios may occur while the user tries to execute the statements present in the editor:

1. **Correct SQL parsing but incorrect DB statement:** Sometimes, the entered RA queries can be successfully parsed to SQL queries. However, the SQL queries may not be correct for the DB schema. In such a scenario, a DB error will be displayed in the Trace tab.



Since the SQL query was a correct one, it can be viewed in the SQL Query tab in such a scenario.



2. **Incorrect SQL parsing, correct DB statement:** In such a scenario, the application will ignore the SQL parsing of the queries and will execute them directly (Please refer screenshot in the SQL support section above) against the DB. No parsing information will be available in such a scenario.

3. **Incorrect SQL parsing, incorrect DB statement:** In such a scenario, the Trace tab will display the parsing error.

If the editor contains multiple statements, the application tries to execute them one after another against the DB even when the statements are not valid RA queries. If the execution of these statements against the DB causes change in the DB schema or data, the changes cannot be rolled back as they are auto-committed.

e.g. if the user has a statement to delete rows from a relation followed by an incorrect RA or SQL statement, the first statement will still be executed against the DB. So It is advised not to have DDL/DML statements causing updates along with other statements.

2.7. Features:

2.7.1. Single Line Comments

The editor supports single line comments so that the user can add additional information about the queries as part of the script. The comments should be started with '--' in a new line.

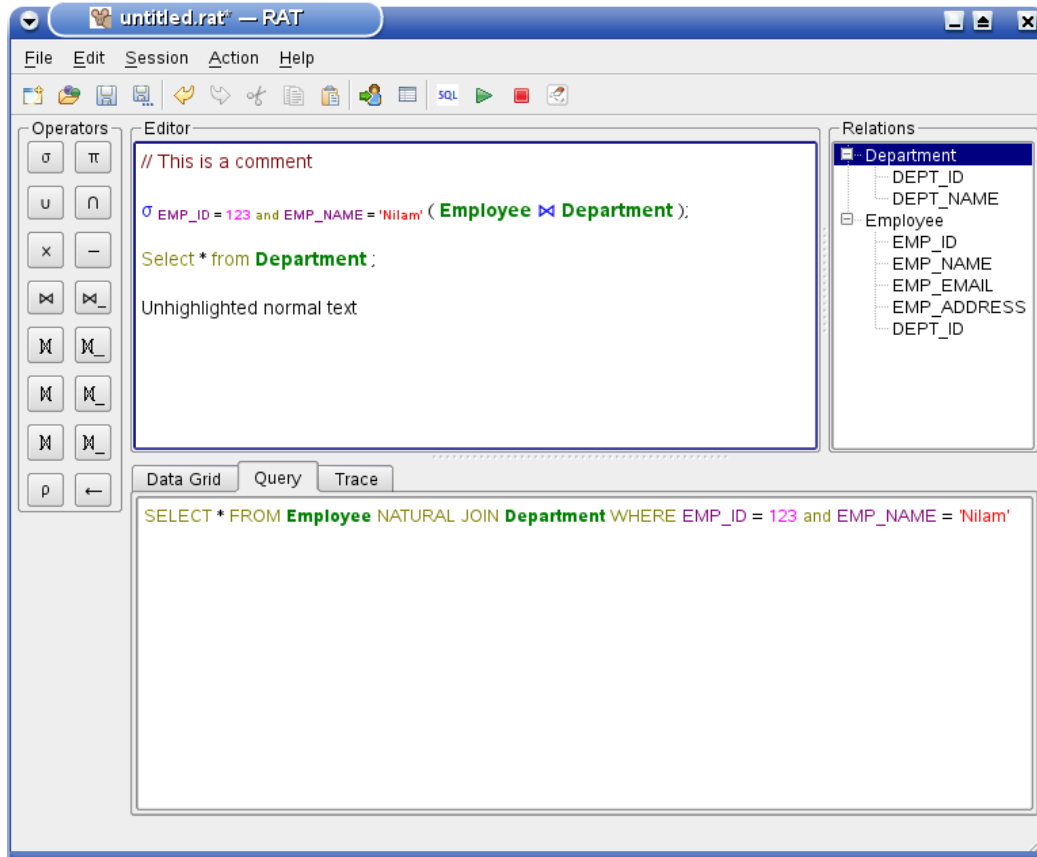
The comments can be saved and loaded back as .rasql files along with the RA queries. The parser ignores the comments.

2.7.2. Highlighting:

The RA Query Editor has a highlighting functionality that helps the user to distinguish keywords, relation, attributes etc.

At runtime, the highlighter checks the query being entered by the user and highlights the query in the following way:

1. Relational operators - Blue
2. Relations (provided the user has connected to a DB) - Dark Green
3. Attributes - Dark Magenta
4. SQL keywords – Dark Yellow
5. Numbers – Pink
6. String (within single quotes) –Red
7. Comments – Dark Red



As can be seen above, the highlighter works in the Query tab too.

2.8. A Sample Workflow:

1. On first starting the application, the user will land on the main application screen.
2. The user uses the buttons and the input box to create the relational algebra query. Clicking on a button will add the corresponding symbol to the input area along with proper brackets depending on the type of the operator. The user can then add the relation name/conditions to complete the query.
3. The user clicks on the Parse button to parse the RA query. If successful, the corresponding query will be presented in the Query tab in the output section. If it is an error, the corresponding information will be displayed in the Trace tab.
4. The user can then click on Execute button to run the query against the selected database. The output will be presented in the output section of the application.

5. If already not connected to a DB, the user will be asked to enter the DB information that he wants to connect to. The user can enter a new DB profile and save it. The DB information will be stored in the application settings and used to get the table information. The user can also alter already entered database details and also add more than one database.

3. The Back-End Parser

3.1. The RASQL Parser:

The back-end parser is based on the GOLD Parsing System. The Gold parsing system combines a tokenizer along with a LALR parser. The RA grammar is defined in the Backus-Naur Form, which is a standard notation used to describe the context free grammars. The notation breaks the grammar into a set of rules which are used to define different programming language tokens.

The following is a small snippet of the grammar used in the application

```
<Relation>      ::= <Projection>
                  | <Selection>
                  | <Union>
                  | <Difference>
<Projection>    ::= PROJECTION'{'<Column List>'}'('<Relation>')'
<Selection>     ::= SELECTION'{'<Expression>'}'('<Relation>')'
<Union>         ::= <Relation> UNION <Relation>
<Difference>    ::= <Relation> MINUS <Relation>
```

The grammar is compiled into LALR (Left-to-right Right-derivation parsing) and DFA (Deterministic Finite Automata) parse tables. The compiled version of these tables is used by the application at the runtime to parse the RA syntax. The DFA is responsible for tokenizing the input into token which are fed to the LALR parser which actually identifies the syntax represented by the tokens processes it.

The LALR parser creates a tree of the symbols which are either a Terminal symbol or a non-Terminal Symbol. A terminal symbol is at the leaf of the tree and does not have any children. On the other hand, the non-Terminal symbol has children which need to be handled by the class defined for the non-terminal symbol.

3.2. Flow of the Parser:

The following is a high-level Flow of the Parsing Engine.

1. The input RA query is tokenized using the DFA parsing table.
2. The LALR parser uses the tokens created in first step to do the parsing. For each state, the engine checks the next token on the input queue against all tokens that expected at that stage of the parse.

- a. If the token is expected, it is "shifted". This action represents moving the cursor past the current token. The token is removed from the input queue and pushed onto the parse stack.
 - b. A reduce is performed when a rule is complete and ready to be replaced by the single non-terminal it represents. Essentially, the tokens that are part of the rule's handle - the right-hand side of the definition - are popped from the parse stack and replaced by the rule's non-terminal plus additional information including the current state in the LR state machine
 - c. When a rule is reduced, the algorithm jumps to the appropriate state representing the reduced non-terminal. This simulates the shifting of a non-terminal in the LR state machine.
 - d. Finally, when the start symbol itself is reduced, the input is both complete and correct. At this point, parsing will terminate.
3. For each reduction of rule in the above step an object is created corresponding to the class of rule matched. This object determines the actions taken for that symbol. Each class had a execute method which is actually responsible for the processing.
4. The execute method of the root Symbol is called, which in turn calls the execute method of its children. Each node of the tree based on the logic for processing the symbol creates the output and returns to the parent. For this it calls the execute method of the children if needed. Using this approach, the final SQL gets build in the depth first mode.

3.3. Symbol Processing Logic:

There are different classes defined for handling the logic for different Symbols. The following is a snippet of code for a class which processes the “UNION” symbol.

```
class SymbolUnion : public JoinRAOperations
{
    virtual wstring getJoinOperator() {
        return L" UNION ";
    }

    bool Execute(ExecuteValue* pResult, ExecuteData* pData)
    {
        ExecuteValue Relation1, Relation2;
        if (!ExecuteChild(0, &Relation1, pData))
            return false;

        if ((MyExecuteData *)pData)->plainRelation)
            pResult->String=L"SELECT * FROM " + Relation1.String + getJoinOperator();
        else
            pResult->String=Relation1.String + getJoinOperator();

        if (!ExecuteChild(2, &Relation2, pData))
            return false;

        if ((MyExecuteData *)pData)->plainRelation)
            pResult->String+=L"SELECT * FROM " + Relation2.String;
        else
            pResult->String+=Relation2.String;
        ((MyExecuteData *)pData)->plainRelation = false;
        return true;
    }
};
```

The above code illustrates the parsing of any symbol is nothing but processing the children and then combining the results based on the requirements of the Symbol. Here the child relations are combined together with a join symbol, which is UNION in case of this symbol. To make the code more manageable and to promote the reuse the classes have a hierarchy based on the type, functionality and required parsing of symbol. For this specific case the *SymbolUnion* class is derived from *JoinRAOperations*. In case there is an error parsing, the class returns false, which is bubbled up the hierarchy.

3.4. Schema aware processing:

There are certain operations in Relational algebra which cannot be converted into the SQL equivalent without the knowledge of the schema. The Rename operator when converted to SQL needs to know the schema of the child relation. In case it is a Projection the columns selected in the projection is considered as the schema. For the cases when it is a plain relation the schema information needs to be populated with the engine.

3.5. Query Optimization:

The Parser engine tries to optimize the output query by moving the selection and/or join conditions to the inner most query. There is a generic scheme implemented for the nodes to communicate with each other and move the operation to be performed on one node to be moved to its child node. For example in case there is a selection having a child as projection, both the operations can be combined into a single select statement.

4. The API / Command Line Interface:

4.1. Using the APIs:

As per the project requirement, we have developed a set of APIs that can be directly used to parse relational algebra queries using the RASQL Backend Parser. These APIs can be used in two ways:

1. Library Files:

The parsing engine can be used as a standalone component and integrated to another FE application to parse relational algebra queries.

The engine is accessible via a class RAParser. The class can be used as shown in the sample code below.

```
#include <raparser.h>

RAParser *ra=new RAParser();

/* Update the schema details to the parsing engine */
list<wstring> attrlist;
attrlist.insert(L"t1col1");
attrlist.insert(L"t1col2");
attrlist.insert(L"t1col3");
attrlist.insert(L"t1col4");

ra->addRelationAttributes(L"tab1", attrlist);

ret = ra->parseRA(buf);
if (ret)
    printf("Error - [%s]\n", ra->ErrorMessage.c_str());
else
    printf("Success - [%s]\n", ra->getResult().c_str());
```

As can be seen, the following API functions are available:

1. **Constructor:** Constructs and returns the RAParser Class.
2. **addRelationAttributes:** This function can be used to add relational schema information to the parser. Please refer section 2.5.3 to know about the need of this function. Note: this step is optional.
3. **parseRA:** parses the string passed as input parameter. Returns non zero value if the string cannot be parsed correctly.
4. **getResult:** Returns a string containing the SQL query for the input RA query.
5. **ErrorMessage:** Contains the error message in case of an error.

4.2. Plain Text Query Format:

The various operations should be entered using the following syntax:

Selection	= SELECTION {condition} (Relation)
Projection	= PROJECTION {columnNames} (Relation)
Union	= Relation1 UNION Relation2
Intersection	= Relation INTERSECTION Relation
Cartesian Join	= Relation PRODUCT Relation
Difference	= Relation MINUS Relation
Natural Join	= Relation JOIN Relation
Theta Join	= Relation JOIN{Condition} Relation
Full Outer Join	= Relation OUTER_JOIN Relation
Theta Outer Join	= Relation OUTER_JOIN{ condition }Relation
Right Outer Join	= Relation RIGHT_JOIN Relation
Theta Right Outer Join	= Relation RIGHT_JOIN{ condition } Relation
Left Outer Join	= Relation LEFT_JOIN Relation
Theta Left Outer Join	= Relation LEFT_JOIN{ condition } Relation
Rename	= RENAME {NewRelationName (NewCol1, NewCol2 ...)} (Relation)

The operations should be fed to the backend engine in the same way as it is mentioned above irrespective of the front end.

5. Architecture:

We are choosing the following tools/framework for the implementation RASQL:

Framework: QT

Language: C++

Platform: Linux 2.6/(Windows/MacOS under development)

DB Engine: SQLite/MySQL (aimed to have multiple DB support)

5.1. Parser Tool:

For the backend parsing engine implementation, we are using a free parsing application called GOLD (Grammar Oriented Language Developer). It is a development tool that can be used with numerous programming languages and on multiple platforms.

Like most parsing systems, GOLD uses the LALR algorithm to analyze syntax and a Deterministic Finite Automaton to identify different lexical units. However, GOLD takes a different approach than common compiler-compilers.

The LALR and DFA algorithms are simple Automata - using lookup tables to determine both state transition and parsing actions. As a result, most of the computational work is performed before any text is parsed. It is the computation of these tables that is both time-consuming and complex. Once created, these parsing tables are essentially independent of any programming language.

GOLD takes a unique approach of logically separating the development of the LALR and DFA parse tables from the algorithms that use them. As a result, GOLD does not require developers to embed the source code directly into the grammar. Grammars are completely independent of any implementation language or platform. Instead, the application analyzes the grammar and then saves the parse tables to a separate file. This file can be subsequently loaded by the actual parser engine and used.

5.2. DB Engines:

The application is not directly coupled with any specific engine. Depending on the DB engines installed in the OS, it can be used to connect to any of them. It has been tested with SQL Lite and works fine.

6. Conclusion:

Given the short timeline, this is all that we could implement. A number of features are in the pipeline. They include – windows and MAC versions of the software, in built print functionalities and further optimization of the generated SQL queries.

Please let us know in case of any defects in the application or any queries you have. Feel free to contact us at Mangesh Bendre <bendre1@illinois.edu>, Nilam Sharma <sharma30@illinois.edu>