Relational Algebra Learning Tool

Pritam Mitra
pm105@doc.ic.ac.uk

Supervisor: Dr. Peter McBrien
Second Marker: Dr. Fariba Sadri

June 22, 2009

Abstract

Currently there is a lack of good quality learning tools which makes the process of learning Relational Algebra an interesting and exciting activity. Majority of the learning tools available for teaching database concepts concentrate solely around SQL, which is based on the concepts of Relational Algebra. As a result, often lecturers introduce Relational Algebra as conceptual operations which can be performed by actual query languages. This often leads students to believe that Relational Algebra play no important role in the actual implementation of database applications. They fail to realise that SQL queries act as declarative language telling the Database Management Systems (DBMS) what it wants. Relational Algebra on the other hand is a procedural language since it is concerned with the execution of SQL queries. Hence it is very important that students get a good understanding of Relational Algebra as it allows them to understand database operations in more detail and motivate them to write optimized queries.

In this report, we introduce a new tool called RALT - Relational Algebra Learning Tool. This tool allows the creation of Relational Algebra queries by using a graphical interface following a dataflow approach without the need to manually enter the queries into the system.

Contents

Chapter 1

Acknowledgements

I would first like to give special thanks to my supervisor, Dr Peter McBrien, for his continual support and encouragement throughout the course of the project. Secondly, I would also like to thank my second marker, Dr Fariba Sadri, for providing key insights into this project.

I would like to thank my family and friends who have always given their full support throughout my time spent at university.

Chapter 2

Introduction

2.1 Motivation

The huge success of relational databases has inspired the development of Structured Query Language (SQL) - a query language designed for interaction with relational databases.

Over the past two decades, SQL has gradually evolved from its first commercial use into a computer product. SQL has been accepted as the industry standard for database programming language. It is used in systems of various sizes - from mainframes to personal computers and even handheld devices.

SQL is built on the concept of Relational Algebra. Relational Algebra can be seen as the mathematics which underpins the SQL operations and acts as a formal description on the behaviour of relational databases [1]. It bears resemblance to normal algebra ($as\,in\,x \times 3 + y \times 2$) but uses relations as values instead of numbers [2]. The inner, lower-level operations of relational databases are, or are similar to, Relational Algebra operations. Gaining expertise in Relational Algebra is the foundation needed for the student to effectively craft queries in any commercially available languages.

Along with these important applications, the simple, compactness and platform independent nature of Relational Algebra makes it an integral part of Database courses. Students equipped with a good understanding of Relational Algebra concepts can easily break down a large problem into smaller and easily solvable problems. Practicing SQL without having a good understanding of Relational Algebra often encourages students to write complex SQL queries from the very beginning. Students fail to realise how SQL queries are broken down during execution and often compute un-optimised queries (e.g. computing JOIN operation and then performing SELECT on the results produced by the JOIN operation). Querying with Relational Algebra forces students into a disciplined reasoning process that involves a partitioned, step-by-step and sequential scheduling of tasks. As a result student gain a knowledge of writing optimised query.

Despite the fact that Relational Algebra is such important aspect of the computing world particularly in the database subject area, lecturers when conducting database courses, often do not spend much time explaining the fundamentals of Relational Algebra. Relational Algebra is introduced as conceptual operations which can be performed by actual query languages. This leads computing students to believe that Relational Algebra play no important part in the actual implementation of

database applications. The advantage of learning SQL and writing SQL queries without understanding Relational Algebra in depth strengthens this belief. Students fail to realise that SQL queries act as a declarative language- telling the Database Management Systems (DBMS) what it wants [2]. Relational Algebra on the other hand is a procedural language as it is concerned with how the SQL queries should be executed.

Majority of the learning tools available for teaching database concepts concentrate solely around the SQL language. The lack of high-quality learning tools for Relational Algebra acts as a factor for not being able to draw student's interest in this subject. Students often shy away from executing Relational Algebra queries using the learning tools at hand, as they are required to construct the queries by manually typing them into the system. When building queries by entering them manually into the system, users often make syntatical errors and they spend a large proportion of their time getting their query syntatically correct rather than understanding the concept of Relational Algebra.

In this report we introduce a new tool called RALT - Relational Algebra Learning Tool - for assisting in teaching Relational Algebra. RALT allows users to build Relational Algebra queries using an interactive graphical interface following the data flow approach and hence exempts them from the hassle of entering query manually into the system. RALT also provides visualisation for different stages of the query execution process. These features of RALT make the learning of Relational Algebra an easy and enjoyable activity.

Through an example we will explain how Relational Algebra can be learnt in an easy and effortless manner by using our system RALT.

Assume we have two relational tables *branch* and *account* as shown in Table 2.1 and Table 2.2.

sortcode	bname	cash
56	Wimbledon	94340.45
34	Goodge St	8900.67
67	Strand	34005.00

Table 2.1: branch

no	type	cname	rate	sortcode
100	current	McBrien, P.	NULL	67
101	deposit	McBrien, P.	5.25	67
103	current	Boyd, M.	NULL	34
107	current	Poulovassilis, A.	NULL	56
119	deposit	Poulovassilis, A.	5.50	56
125	current	Bailey, J.	NULL	5 6

Table 2.2: account

The result displayed in table Table 2.3, is achieved by executing the query $branch\, X\, \sigma_{rate>0}\, account$ i.e. execute a product operation between the tuples of *branch* and those tuples of table *account* which have the value of *rate* attribute value greater than 0. However, the person executing the query fails to observe the intermediate steps and results produced when executing this query.

sortcode	bname	cash	no	type	cname	rate	sortcode
56	Wimbledon	94340.45	101	deposit	McBrien, P.	5.25	67
56	Wimbledon	94340.45	119	deposit	Poulovassilis, A.	5.50	56
34	Goodge St	8900.67	101	deposit	McBrien, P.	5.25	67
67	Strand	34005.00	101	deposit	McBrien, P.	5.25	67
67	Strand	34005.00	119	deposit	Poulovassilis, A.	5.50	56

Table 2.3: Table produced as a result of executing the query

$$branch\, X\, \sigma_{rate>0}\, account$$

In Figure 2.1 we show how the same query can be displayed in a graphical interface. To build the query, a step-by-step approach is taken. The user breaks the above query into two parts - first executing the operation $\sigma_{rate>0}\, account$ and then performing the operation $branch\, X\, result$, where *result* is the output generated by the first operation. A similar approach is taken when building queries in RALT. Using RALT's user friendly graphical interface, queries can be built effortlessly just by providing the inputs for operators and then using the result table produced as input for other operators.Executing queries this way together with the graphical representation of the query building process, enhances user's understanding of how queries are built and improves their knowledge about different Relational Algebra operators.

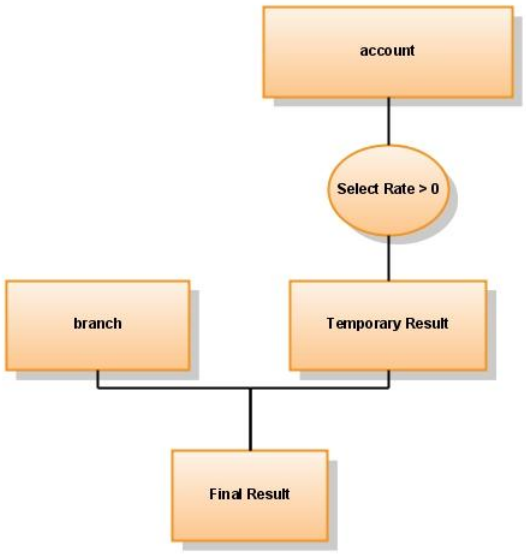


Figure 2.1: Buidling Relational ALgebra queries using data flow approach

2.2 Project Aims

The primary aims of RALT are listed below.

1. **Connecting to Database**
   Our system will allow the users to connect to a desired database. As a result a front end must be present which will allow the users to enter necessary information to connect to a database. Once all the data is provided our system will connect to the database and will allow the user to perform actions on the contents of the database.

2. **Storage of Data**
   Our system should handle user request in an intuitive way such that if the data is altered the changes are not reflected on the mother database (i.e. the database to which user is connected). Hence we must intuitively store all the necessary data from connected database on the client machine and allow the user to perform operations on this duplicated data.

3. **Displaying Data**
   A fundamental element of our system is that it will allow visualization of the contents of the data present in the tables of a database. Also the system should display the relevant metadata information associated with each table of the database necessary for query building.

4. **Implementation of the basic Relational Algebra Operators**
   Our system takes the unique approach of performing all the operations in memory and should provide the implementation of the seven basic Relational Algebra operators - *Select, Project, Natural Join, Union, Intersection, Difference,Product.*

5. **Building Queries**
   The aim of our system is to allow the users to build queries. The system should allow the user to build queries by selecting the operators from a list and providing as inputs to these operators tables fetched from the database the system is connected to. The system will provide a visualisation of the query in a data flow approach. Our system will also allow users to use results produced from a particular query to act like the input for another operator.

6. **Deletion of Tree Node**
   The system should be able to delete a particular section of a query tree on request.

7. **Implementing advanced Relational Algebra Operators**
   To ensure our users gain a further advanced knowledge on the concept of Relational Algebra, we will aim to add advanced Relational Algebra Operators such as *Division, Left Outer Join, Right Outer Join, Semi Join, Anti-Join* to our system.

8. **Introduction of Data Lineage**
   This feature will allow our users to track quickly how a particular data is derived and hence will help in enhancing their knowledge on Relational Algebra operations. Currently no learning tool for Relational Algebra is equipped with the feature of Data Lineage.

Chapter 3

Background & Research

We begin this chapter by reviewing what Relational Algebra is (Section 3.1) and understanding the functionality of the different Relational Algebra operators which have been implemented in our system. In Section 3.2 we investigate the different ways visualising query building process. We then evaluate some existing work done for assisting the learning of Relational Algebra (Section 3.3) before looking into different ways of finding the lineage of an information (Section 3.4). Lastly in Section 3.5 we review the technology and tool used in our project.

3.1 Relational Algebra

Relational Algebra is an off shoot of first-order logic. In 1970 E.F.Codd while working for IBM proposed how Relational Algebra can be used as a basis for database query language. Since then Relational Algebra has find extensive use in the development of query languages, the most popular of them being SQL.

The beauty of Relational Algebra is that each of its operators takes as input one or more relational tables and outputs a relational table as result, which can again act as an input to another operator. This unique characteristic makes it simple to construct complex queries using the relational operators. SQL being based on relational algebra follows a similar approach when building query. For e.g. let us again take the table *account* (Table 2.2) into consideration. If now the query $\sigma_{rate>0}\, account$ is executed the result displayed in Table 3.1 is produced.

no	type	cname	rate	sortcode
101	deposit	McBrien, P.	5.25	67
119	deposit	Poulovassilis, A.	5.50	56

Table 3.1: temp_result

The result produced in Table 3.1 can then act as the input for the query Π_{type} to produce the result displayed in Table 3.2. This feature of Relational Algebra allows us to go on building complex queries by providing output of one operation as the input of another. In our application RALT we use this feature of Relational Algebra to build query trees.

type
deposit

Table 3.2: Result produced by

$$\Pi_{type}\, temp\_result$$

Relational Algebra operators are usually divided into two groups - *basic* and *advanced* operators. Below we will see a small description of the five basic operators of Relational Algebra:

- **Project** : A unary operation is used to select particular columns of a table. Written as $\Pi_{a_1,a_2,...a_n}(R)$, where $a_1, ...a_n$ corresponds to a set of attribute names, the Project operator produces tuples in R which are restricted to the columns $a_1, ...a_n$
- **Select**: Another unary operation used to select particular rows of a table. Written as $\sigma_\varphi R$, this operator selects the set of rows from table R which satisfy the condition φ.
- **Union**: A binary operation takes two tables with identical attributes as inputs and produces a single table containing a set of elements containing elements from both the tables.
- **Intersection**: Another binary operation which takes two tables with identical attributes as inputs and produces a single table containing a set of elements which are shared by both the tables.
- **Difference**: Is a binary operation that also takes two tables with identical attributes as inputs and produces a single table containing a set of elements which are present in the first table removing the ones common to both the input tables.
- **Product**: The Product operator again takes two tables as input and produces an output with all the possible combination of tuples from both the tables. For an example consider the tables A and B and their Product

id	name
1	John
2	Jack

Table 3.3: Employee

department
Sales
Marketing

Table 3.4: Department

id	name	department
1	John	Sales
2	Jack	Sales
1	John	Marketing
2	Jack	Marketing

Table 3.5: Result procued by executing

$$Employee \, \chi \, Department$$

Apart from the basic operators, Relational Algebra consists of some advanced operations such as:

- **Natural Join**:
  A binary operation written $R \bowtie S$ produces a set of all combinations of tuples in R and S which are equal on their common attribute names. For example consider the tables *Employee* and *Dept* and their *natural join*

name	empId	deptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Table 3.6: Employee

deptName	manager
Finance	George
Sales	Harriet
Production	Charles

Table 3.7: Department

name	empId	deptName	manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

Table 3.8: Result after executing

$$Employee \bowtie Department$$

- **Semi Join**:
  Similar to Natural Join but the result of a Semi Join is only the set of all tuples in the first input R for which there is a tuple in the second input S such that they are equal on their common attribute names.

- **Anti Join**:
  Anti Join, written as $R \triangleright S$ is another binary operator which only displays the tuple of R for which there is no common attribute in S.

- **Left Outer Join**:
  Written as $R = \chi S$, the Left Outer Join produces the set of all possible combination of R & S

that are equal in their common attribute names. Additionally it also displays the tuples in R for which there is no tuple in S common on the shared attribute names.The output tuple for such rows in R show Null values in the columns of S which are not common between R and S. For example given two table R and S, the Left Outer Join of the two

name	empId	deptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Table 3.9: Employee

deptName	manager
Sales	Harriet
Production	Charles

Table 3.10: Department

name	empId	deptName	manager
Harry	3415	Finance	null
Sally	2241	Sales	Harriet
George	3401	Finance	null
Harriet	2202	Sales	Harriet
Tim	1123	Executive	null

Table 3.11: Output for executing the Left Outer Join

$$Employee = \chi\, Department$$

- **Right Outer Join**:
  Similar to the Left Outer Join, but this operator displays the tuples of the tuple S which have not featured in the Natural Join between R and S.

- **Division**:
  Division takes two inputs say R & S and produces the attributes unique to R for which it holds that all their combinations with tuples in S are present in R. It is written as $R \div S$.

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sara	Database1
Sara	Database2

Table 3.12: Student

Task
Database1
Database2

Table 3.13: Department

Student
Fred
Sara

Table 3.14: Output for

$$R \div S$$

3.2 Visual Query Systems

3.2.1 Introduction

Query language is composed of a set of formal operators, using which users can express requests to a database [3]. Execution of these queries produces results from the database which are consistent with the requests made. In order to teach the fundamentals of query languages and how they can be used, same text materials are provided both for the naive and the expert users. The high technical jargon used in these manuals often is too hard to understand especially for the beginners. This motivated in the development of a new type of query language through systems called Visual Query System (VQS). VQSs could be seen as an evolution from the traditional query languages, focussing on providing a friendly man-machine interaction which would simplify how non-technical users interact with the database.

As our system is mainly targeted for users who are learning database operations for the first time, it is important that we evaluate the different ways effective man-machine interaction can be obtained using the idea of VQS. VQS ensure that the effort needed from non-technical users is reduced and no intermediate learning is required by the user. This is achieved by exploiting drawings/images which when used as metaphor; cater information at the correct level of abstraction within a concise form [4]. VQSs are gaining popularity because it reduces significant amount of the mental load users previously were forced to carry and provide sufficient visual information of the computing processes and their relationships, which are much easier to perceive. VQS takes advantage of the high bandwidth of human-vision channel, allowing quick gathering of large amount of information while exploiting the visual feedback techniques [5].

One of the primary aims of adopting a visual representation in a query based system is that it opens up a clear communication channel between the user and the contents of the database. A visual representation presents in front of the users essential features concerning the data, making it easily absorbable while omitting any unnecessary details such as the internal structure of the data. Existing VQS make use of common objects such as tables, diagrams, icon etc as a means of representing information. An experienced user does not value the representation friendliness much as his/her technical skills assist in adapting to abstract concepts with reasonable effort. However, for a novice user representation friendliness is of prime importance as they prefer to interact with items similar to the reality they are in and not be acquainted with the existence of the underlying abstract model.

3.2.2 Visualisation Representation Approaches

Models are used in VQS for indicating the data and queries. Their corresponding visual representations together with the strategies provided by the system assist in formulating the query. The usability of

the VQS is determined in terms of these models [3]. As query representation is generally dependent on data representation, both data and query are treated through a unique classification scheme which includes different concepts, according to the choice and organisation of the chosen symbol. A brief explanation of the concepts is given below:

- **Form Based**
  It is seen as the first attempt to migrate from the traditional linear string representation of query and beginning of the exploitation of the bi-dimensional space to provide an easy-to-use-interface for data manipulation [4]. Forms are used to display objects which have similar structure. The form concept used in program visualisation maps to an abstraction of the conventional paper forms. Here forms are used as abstractions of tables. The main characteristic of this approach is the visualisation of the table prototype when a query is formed, by inserting text into a form field. Query-By-Example (QBE) applications have adapted this method of visualisation.

id	name	country
3	John	...

Table 3.15: Form for table *user*

  For example as shown in Table 3.15, by completing the form the SQL query $SELECT * from\, user\, WHERE\, id = 3\, AND\, name = `John'$ is generated.

- **Diagram Based**
  In a diagram based representation, the visual components share a one-to-one correspondence with specific concepts. However, diagram has a broader meaning and it is also used to describe charts, network mode and graphs. The layout of a diagram can be modified following certain protocols to produce new relationships. In Entity Relationship Model, this approach has been adopted. In such models, rectangles are used to denote entities; diamonds are for relationships while circles are used to denote attributes and relationship between these components are established by drawing lines among them. If some of the lines are redrawn then a new relationship model is produced.

- **Icon Based**
  Here a set of icons are used to represent the entities of the database together with the operations available to be performed on them. A query can be composed by combining these icons. Systems using this approach mainly target users who are not familiar with the concepts of the data models and who may find it difficult to adapt/interpret the Entity Relationship diagram.

- **Hybrid**
  Such visual representation uses all three concepts discussed above, offering the user various alternative representations of database and queries by combining different visual formalism into a single representation.

Although VQS are proving to be a threat to the traditional query systems, they have their drawbacks too. No standard has yet been established enabling different icons being used in different applications to represent the same concept. This increases in potential ambiguity while interpreting a set of icons and the discrimination power decreases. Moreover, given the space available for displaying the icons, overcrowding of icons often occur. The limitations of screen size cluttered drawing and images are

often seen while displaying information, thus reducing their readability. Although one can take the advantage of scrolling technology to space out the components, a comprehensive representation of the whole reality of interest may be lost.

We feel the *Hybrid* representation is the best choice for our system. Such a representation allows us to use *icons* in order to represent Relational Algebra operators while *dataflow* diagrams are used for building for queries. *Dataflow* diagrams can be seen as a collection of boxes, connected by lines which represent the information flow. Some of the boxes in the *dataflow* diagrams will represent data tables whose contents can be shown in a *form* representation. Also user input for the operators such as Select or Project could be received using the Form representation.

3.2.3 Interaction Strategies

In order to formulate query, VQS should be equipped with simple, easy to adapt interaction strategies which will allow users to fetch the desired results from the database they are interacting with. Being able to identify the information one is interested, especially when the database schema is made up of a large number of concepts is a complex and difficult task. To solve this problem a mechanism is required which will present the schema with varying amounts of detail and allowing the user to control it. This is achieved by two approaches Top-Down and Browsing.

In a Top-Down schema navigation approach the general aspects of the reality are perceived first followed by the introduction of specific interest [6]. Some of the different means by which this navigation is performed are:

- **Selective Zoom**
  Here the concepts are layered according to their importance. The concepts can be graphically examined at different levels of abstraction allowing objects above a specified importance level being displayed at a time. For e.g. in a E-R diagram, *customer* table has a relation *has_account* with the *account* entity. Again *customers* can be generalised into *regular_customers*, *premium_customers*. When showing a visual representation in such an approach, the E-R diagram will show only the relationship between the *customer* entity and the *account* entity removing information on the type of customers. On drilling down further on the *customer* entity the two generalisations can be seen.

- **Hierarchical Zoom**
  This approach allows objects to be examined at different levels of detail. Such an approach is presented in ESCHER [7]. In ESCHER the data model reflects the structure of the objects by an extended relational model in which four types of attribute values (list, tuples, atomic values and multi-sets) are defined at arbitrary depth of nesting. ESHCER allows the aggregation of objects based on their background relation and a visual interface shows a nested table to the user in several levels of detail.

The Browsing schema navigation scheme is designed keeping in mind that the users possess little knowledge about the database and its interaction techniques and more than often do not have a predefined goal when accessing the database. A brief description of the different means of navigation under the Browsing navigation approach is given below:

- **Intentional Browsing**
  This browsing is performed in the conceptual level of the database schema. It identifies all the

existing paths between two concepts and specifying conditions on the length of the path and/or the presence of particular concepts.

- **Extensional Browsing**
  Here the user lives inside a single Entity-Relation tuple and sees the database from the perspective of that tuple. The systems play its role in showing all the relationships concerning the tuple in which the user is residing in. A tuple from one of the linked entity could be selected by the user and that tuple becomes the centre of focus.

As we will see, in our application we have used a variety methods of the methods discussed above for displaying various types of information. Selective zoom is used to show the contents of a table present in the query dataflow diagram. Initially the query will be made up of operators and table names and user can use selective zooming to view the contents of a table or the conditions attached to particular relational algebra or SQL operator. Hierarchical zoom will be used to see the contents of a table selected from a list of all the tables present in the database. The tool will also exploit intentional browsing for extracting information such as which two tables can be combined when the JOIN operator is performed or the table names with which the interested table shares a relation. Extensional browsing is used when performing data lineage to show which database tuples contributed in deriving a selected tuple.

Along with being able to extract the details of database easily, VQS also need an easy way for query formulation. Different methods for formulating a database query are given below:

- **Arbitrary Connected Path**
  In this method the user selects the intentional pattern of interest i.e. the entities and their associations. Once an entity is selected, the data the user is looking for could be fetched by specifying restriction conditions which involve attributes of a single entity while inter-entity clauses could be performed by attributes of different entities.

- **Connected Hierarchial Path**
  In this approach the user selects an entity of the database following which the system builds a hierarchical tree view of the database with the selected entity at the root. For example in GORDAS [6], user first selects a root concept to determine the direction of reference for the involved relationships. Relationship attributes are assigned to the entity at the lower level in the hierarchy. Then, the user provides the selection conditions. First, conditions on the attributes of the root entity are specified, and, later on, those involving related entities. It is worth noting that different root entities may be specified for the same query giving rise to different views.

- **Unconnected Path** Users are often interested in creating new concepts by combining different concepts already existing in a database. The unconnected path approach for creating query can be used to meet such user requirements. HIQUEL [6] is a system where this approach is used. Following a step by step approach, a query can be built first by deleting the non-relevant attributes of the selected entities and then proposing conditions on the applicable attributes. Entities not explicitly related in the database schema can be linked with the help of relational algebra's join operator.

In RALT the concept of Arbitrary Connected Path and Unconnected Path when creating a query dataflow diagram has been implemented. Arbitrary Connected Path is used when specifying conditions on the select operator when applied to a database table. Our system also allows queries to be created only between tables which share a relationship but also between other tables as long as they meet the

relationship criteria (for example if a union operator can only be applied on two tables if they are compatible).

3.3 Related Work

In this section we will discuss some of the previously published work which has acted as an inspiration/benchmark when designing the learning tool for the teaching of relational algebra. In the context of database learning, a majority of the learning tools developed has focussed solely on the SQL language [4]. One such tool is *SQLator* - an online SQL learning workbench [8]. It is a web-based interactive tool for learning SQL.

It allows the user to evaluate his/her query formulation with the help of an evaluation engine based on complex heuristic algorithm. However, queries are created not by graphical interaction but through manually writing the query. Moreover, such tools lack the advanced Relational Algebra operators like Division, Anti-Join etc. Tools such as *DBTool* have also been designed to support the process of designing databases. Such tools provide a graphical interface for drawing the entity-relationship data model. Fortunately, some tools have concentrated in highlighting the importance of the Relational Algebra in databases applications. Below we provide a brief introduction to some of these tools.

- **RAIN**
  Relational Algebra Interface Using Java is a tool aimed at university students to enhance their learning in the creation and execution of the Relational Algebra query [5]. The tool operates by connecting to a database and allowing users to perform database operations on that database. The tool provides an interactive visual display of the physical nature of the database contents and presents useful information such as Primary Key of a table. The application allows users to create Relational Algebra queries in the algebraic notation. The queries created are then transformed into their respective SQL representations and the results are maintained in a persistence state in the database the system is connected to. This allows the user to re-use these results any time later on.

  However, *RAIN* expects users to be familiar with the Relational Algebra query notation. Hence for users interested in using this tool, must first possess some basic concepts on how to formulate Relational Algebra queries. Moreover users cannot view the results until the syntax of their query is correct. We have tried to address this issue in our system by eliminating the need for the user to construct queries using the algebraic notation. Instead all the user needs to do is just select the operators and tables required for building the query together with any constraints needed and the system takes care of the rest. This way user spends more time understanding the concept of relational algebra operators and less time on fixing syntatical mistakes in their query.

- **RELATIONAL**
  RELATIONAL is another application developed in Python which helps in the learning and creation of Relational Algebra queries. It allows users to build Relational Algebra queries by selecting Relational Algebra operations represented as icons in the application. Once the desired query has been constructed by the user, the query can be executed by clicking a button which in turn displays the results in the result panel of the tool.

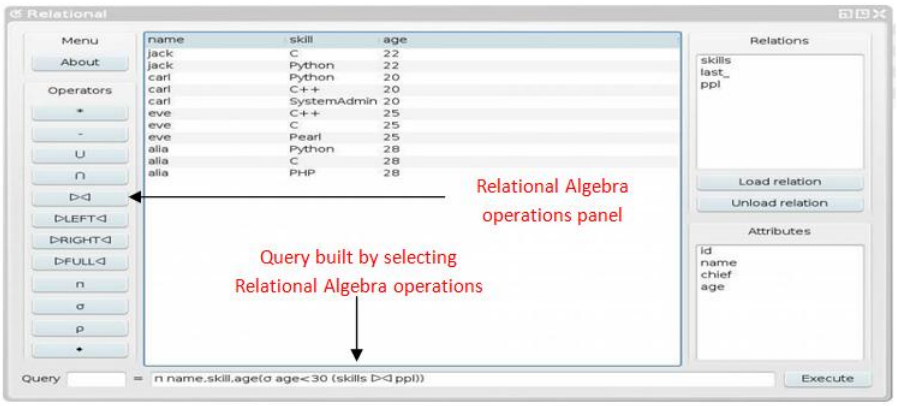


Figure 3.1: GUI for RELATIONAL

This tool gives more flexibility to the user compared to RAIN, as it allows the user to create the Relational Algebra query by just selecting some Relational Algebra operators, tables and attributes. However, the interface provided by the tool is not enough interactive as well as informative and does not give the user a better understanding of how Relational Algebra query works. For example as shown in Figure 3.1, a long sequence of query has been formulated to compute the result shown in the result panel located at the centre of the application window. However, by no means can a user find out what intermediate results were produced which had contributed to the final result and hence a compromise is made on their understanding of the functionality of relational operators.RALT addresses this problem by allowing the user to build a query step-by-step and displaying any intermediate results produced. The user can then carry on building a query by using any intermediate result as inputs to operators in the query.

- **Relax**
  RELAX - Relational Algebra Explorer is a project aimed at students to help them reason queries in an algebraic style while at the same time improving their skills in SQL programming. In this project a Rlational Algebra parser in a graphical interface has been implemented. The interface allows the creation of query in a text format by choosing Relational Algebra operators. The queries could be of any degree of complexity and can be nested, one inside the other of the parenthesis. Once created, the query is it can be analysed and resolved.

  RELAX allows the user to create complex nested queries which is not the best way for a new user to learn Relational Algebra. Moreover, although *RELAX* provides a graphical interface, the user needs to create query in an expression which *RELAX* understands. This technique requires the user to be accustomed with the different types of expressions *RELAX* supports. Along with that the user while creating a query must provide the correct spelling for the table and field names. This makes the user to spend time ensuring that their query is syntactically correct, which could have been utilised for learning Relational Algebra.

- **idFQL**
  iDFQL - Interactive Data Flow Query Language, is a tool which follows a teaching methodology

different from the tools mentioned above. *iDFQL* provides an interactive graphical environment which allows users to create Relational Algebra queries by selecting graphical elements. It uses an icon-based paradigm together with a dataflow approach which represents a query.

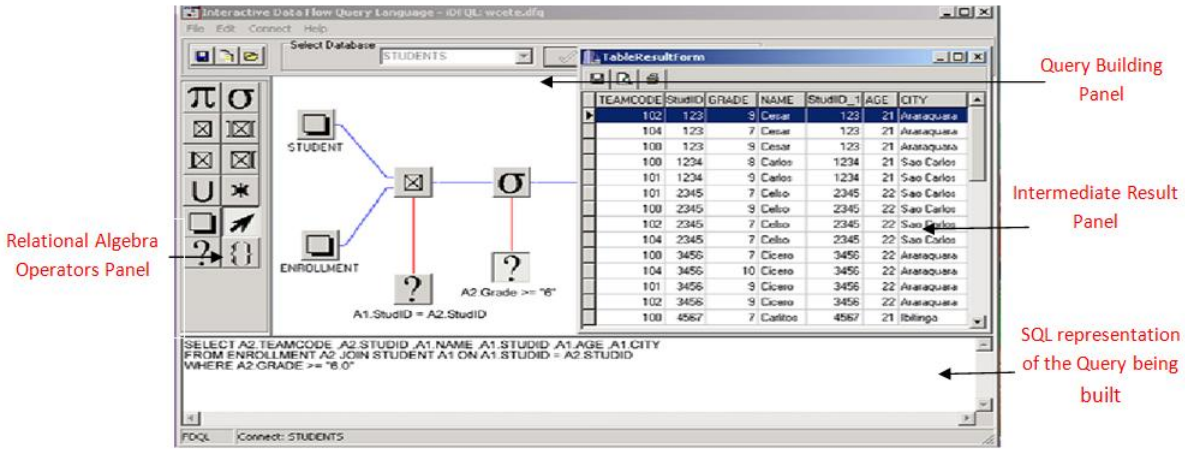


Figure 3.2: GUI for idFQL

iDFQL takes advantage of colours to depict the relationships between different components when building the query. Lines are used to represent the relationship different components and each type of line is given a unique colour. Line connections are of three types - Data Connection connects a table to an operator, Condition Connections sends the condition to the JOIN and SELECT operator, Attribute Connections for sending the list of attributes to the PROJECT operator. It gives the user to flexibility of selecting any portion of the query and executing it. Although *iDFQL* provides a great cognitive perception of the query representation, the contents of the table which act input to the Relational Algebra operators cannot be seen easily. A query is only executed if and only if every operator is connected and the parameter properly set, thus increasing the user's technical role in query creation. Moreover, it is not possible to determine how a particular row in one of the result tables of the query was derived.

Despite some of the issues with the tool as mentioned above, we can see that *iDFQL* presents a unique way of learning Relational Algebra by using dataflow diagrams for query building process. We decided to use the dataflow approach *iDFQL* follows when building queries in our system, RALT. We have tried to address the shortcomings of *iDFQL*. RALT allows its users to view the contents of a table at anytime like in *RAIN* and *RELAX*, along with its schematic representation. Unlike RELAX, no specific syntax is required for query creation in this project. In *iDFQL* the whole query must be built before its intermediate tables can be viewed. In this project the result of an operation is displayed as soon as it is completed and this will provide the user with a better understanding of query execution mechanism.

iDFQL builds the query from left to right which in order to view a long query, the user may have to perform horizontal scrolling. This may not be user friendly for most users as they are

more accustomed to vertical scrolling and hence in our application queries are built in a top-bottom approach, reverse of how data is inserted in a stack data structure. *iDFQL* also sacrifices the space occupied by the panel for viewing the entire query that has been built by using the same panel for displaying intermediate table information. In our application the actual visual representation of the query built will never be compromised.

3.4 Data Lineage

The increase in the number of applications using distributed databases has resulted in the rise of data warehousing systems. Data warehousing systems integrates data from several heterogeneous databases in order to present a "single version of the truth' to its users [9]. In such an environment, data analysts often like to identify the source of a particular piece of data. In industry terminology they want to be able to trace the data lineage. In a materialised view provided, identifying the source of the data that produced the view together with the process by which it was produced is termed as data lineage [10]. Some applications of data lineage are:

- **OLAP and OLAM**: Effective data analysis and mining requires facilities which will allow the exploration of data at different levels. Being able to select a portion of relevant view data and then drilling down to its origin can be very useful. Additionally, analysts may also want to verify the origin of suspect view data, validate the reliability of the sources or even repair the data.
- **Materialised View Schema Evolution**: In a data warehouse environment, users are able to change view definitions (e.g. adding an attribute to a view). View data lineage can help in retrofit existing view contents to the new definition without computation of the entire view.

Data lineage is usually applied in a data warehouse environment (a materialised view for storing the tuples of the view over a number of data sources). However, it can be easily applied to a single database environment. We will see through an example how the concept of data lineage can be easily applied in different sectors of the academic world, especially when teaching the fundamentals of relational databases. Unfortunately, none of the learning tools available for teaching Relational Algebra make use of data lineage. Using the data lineage feature, such tools can assist users in easily identifying how a particular data was derived and hence enhancing their understanding of how Relational Algebra operators work.

Motivating Example
With a simple example we provided a precise definition of data lineage and show how lineage tracing can be useful when learning Relational Algebra and SQL. Consider a database with data of a retail store spread over three source tables. The schema and sample contents of these tables are shown in Tables 3.16,3.18,3.17 [10].

The result as displayed in Table 3.19 was computed by carrying out the following Relational Algebra operators in sequence:

1. Perform a NATURAL JOIN operation on table *store* and *sales* table.
2. Executing another NATURAL JOIN operation on the result produced in step 1 with the *item* table.

s_id	s_name	City	State
001	Target	Palo Alto	CA
002	Target	Albany	NY
003	Macys	San Francisco	CA
004	Macys	New York	NY

Table 3.16: store

i_id	i_name	Category
0001	Binder	Stationery
0002	Pencil	Stationery
0003	Shirt	Clothing
0004	Pants	Clothing
0005	Pot	Kitchenware

Table 3.17: items

s_id	i_id	price num	
001	0001	4	1000
001	0002	1	3000
001	0004	30	600
002	0001	5	800
002	002	2	2000
002	004	35	800
003	0003	45	1500
003	0004	60	600
004	0003	50	2100
004	0004	70	1200
004	0005	30	200

Table 3.18: sales

3. Performing a SELECT operation on the result produced in step 2 by filtering out rows whose state value is not 'CA'.

4. Carrying out a PROJECT operation on the result produced in the previous step and only displaying the s_name, i_name and num fields.

A new view created to show the selling pattern of the stores in the state of California (marked as CA in the store table 3.16) is Table 3.19.

s_name	i_name	num
Target	Binder	1000
Target	Pencil	3000
Target	Pants	600
Macys	Shirt	1500
Macys	Pants	600

Table 3.19: Result after carrying out the sequence of operations in setp 1-4
title of Table

After viewing the result in Table 3.19 we may be interested in the rows of the parent table which has produced a particular row (say the tuple $\langle Target, pencil, 3000\rangle$ marked in grey above) in our result table. Using data lineage process we can not only identify the base table rows (rows marked grey Table 3.16, 3.18, 3.17) but will also highlight the rows of any intermediate table produced, that will contribute in producing the tuple $\langle Target, pencil, 3000\rangle$ .This small example explains how the feature of data lineage can offer incredible benefits to students who are beginners in Relational Algebra. Students can use this feature not only to trace how a particular result was derived and also to understand where they have made a mistake in their query when a result is produced different from their expectation.

Usually a view definition acts as a mapping from the base data to the view data. It is easy to compute the corresponding view when the base data and view definition is provided. However, trying to perform the inverse mapping i.e. from the view back to the base data is a difficult task. In order to determine the inverse mapping accurately some additional information is needed along with the base data.

Different Approaches

From the above example we can identify that the main problem with data lineage is based on two arguments how to store information for recording data lineage and how to use this information stored for tracing data lineage. The data lineage problem in a data warehouse environment has been increasingly become a focus of database engineering. Many different papers have been published which look into these issues of data lineage proposing different solutions for them.

Traditionally data lineage was performed using metadata. Metadata is seen as a relationship data [11] as data transformations imply different dependency on different types of data. For example a change made to a particular data must also be made to other sets of data derived from this data. However the use of metadata is more suited when applied to schema-level lineage tracing and not for very effective for finer fine-grained instance level [12][10]. For example, in order to trace a specific floating point value in a processed data to a particular satellite image pixel belonging to a source data set, it is not feasible and practical to store all the necessary information in terms of metadata.

A correct but brute force method for performing data lineage is to store all intermediate results in addition to the initial input. The lineage can then be traced backwards through one transformation at a time until the base tables are reached. However, such an approach turns out to be highly inefficient due to the high storage space needed to record all intermediate results. Also the longer the sequence is, the less efficient the approach becomes as a large number if tracing procedures needs to be carried out when iterating through the transformations.

[12] suggests a framework for computing instance-level data lineage using the views weak inverse mapping. Taking into account that many database transformations or functions are irreversible, [12] introduces the notion of weak inversion. Let us represent a function as *f*. *f*, if weakly invertible has a corresponding function f^{-w}. The aim of a weakly invertible function for *f*, is to map to the input of f from its output, but it cannot be guaranteed to be perfect due to the fact that many functions are irreversible. So a verification function is provided to make this result accurate. The verification function denoted as f^{-v} takes as inputs the input table to f as well as result generated by the function f^{-w} and outputs data which is a subset of the results generated by the function f^{-w}. Hence it provides a more accurate answer. The verification function also has access to the original data used as input for the function . [12] aims at providing the weak inverse and verification functions at attribute level instead of tuple level as they it allows the view definer to provide inverse functions only for the attributes they are interested in. Once the weak inversion and the verification functions are carried out for each attribute of the tuple of interest, their results are then combined to provide the lineage for the tuple. An inversion planner is present which carries out the sequence of operations that should be carried and the order in which they will occur. It determines which weak inversions and verification functions will be applied to which tables in which order. However, such an approach requires that the view definer to also provide the views weak inverse. This may not be practical all the time.

Trio [13] is a recently developed database management systems built to address the shortcomings of conventional DBMS by providing features like data lineage. The basic data in Trio follows the standard relational model allowing it to be used by existing query languages and focuses data lineage at the tuple level. In Trio if a particular data is either updated/deleted, the existing original data is not changed and is just made to expire but not erased from the system. When an update has happened, a new data tuple gets inserted into the system. The benefit of such an approach is that if a data item A was derived from B and B was updated later on, As original lineage can still be obtained from the expired portion of the database. Trio implements two new features a new relational data model called ULDB along with a new query language based on the existing SQL called TriQL [14]. These features allow Trio to perform data lineage procedures effectively. The main new features added in ULDB are alternatives which determine the uncertainty about the contents of a tuple, maybe for representing the uncertainty about the presence of a tuple, lineage for connecting a tuple alternative to other alternatives from which it was derived. ULDB relations consist of special type of tuples called x-tuple which is made up of one or more alternatives mentioned above. Each of these alternatives is simple regular tuples over the schema of the relation. For e.g. we have a table ULDB relation Food made up of three alternatives shown in Figure 3.3

(food Type, food)		
(fruit, apple)	(fruit, orange)	(fruit, banana)

Figure 3.3: Relational Model ULDB used in Trio

The relation has three possible instances, one for each alternative. With this unique structure of ULDBs relational model, lineage can be recorded at the granularity of tuple alternatives. The lineage feature of ULDB connects a derived x-tuple alternative to the x-tuple alternatives from which it was derived [14], automatically whenever a TriQL query is executed. In the system specified query language TriQL, for querying lineage a built-in predicate is designed to be used as a join condition. This predicate can be used in a recursive way to find the lineage of a tuple. Trio keeps a track of the lineage structure in the database and uses this recursive approach to produce a fixed set of lineage based join.

In [15] another method of lineage tracing is discussed. [15] presents an annotation management system for relational databases. It has decided to sacrifice storage space used for the time taken to compute lineage of a piece of data by attaching annotations to a piece of data which can be transparently carried along as the data is being transformed. How a particular data is produced can be easily identified by examining the annotations attached to the data. The annotations represent the address of the source of the data. Every data is given a unique address. Considering the annotations can be used for different purposes, the base system provides two default types of annotation propagation schemes. The default scheme uses provenance as the basis for propagation schemes. The default-all scheme maintains annotation according to where the data is copied from all equivalent formulations of the query. [15] introduces a new query language pSQL which extends the existing popular query language SQL. pSQL allows users to specify how the annotations should be propagated. The system

stores the data for annotations in a naive way by allocating an extra column which will store the annotation information, for every attribute of a relation. Using this annotation information for a data, one can easily trace back to the all the source data from which it was derived by performing repeated recursive calls.

[10] describes a new approach for performing data lineage using canonical form. It assumes that the contents of a view are computed by evaluating the view definition query from the bottom to the top. If V is a view derived from the base tables $R_1...R_n$, then v is a sequence of queries used to map from the base tables to the view V and it is the view definition query. Each operator in the tree generates result table based on the results its child nodes produces and this result can be passed upwards in the tree. The data lineage for a tuple can be computed by calculating the tuple derivations for the operators which comprises the view definition tree. A view tuple ts derivation is the set of all base and intermediate tuples which contributed to t after being executed from the bottom to the top in the view definition tree. However all the queries defined here are based on set semantics and can easily be applied to Views with duplicate elements.

Deriving the source for a tuple in a Select-Project-Join (SPJ) query can be computed using single relational query over the base data. All SPJ queries need to be transformed into their canonical form first. Using a sequence of algebraic transformations [16] any SPJ query can be transformed to the form - $\pi_A(\sigma_C(R_1 \bowtie R_2 \bowtie ...R_n))$ where $\bowtie, \sigma, \pi$ represent the Join, Select Project relational operators and $R_k$denotes the relations involved. Once the tuple is transformed to its canonical form, a single query can be used on the canonical form to derive the tracing query for the tuple. A new operator called a Split Operator (Split)[10] is derived. Given a table T as an input, the Split Operator chops the input table into several tables, each being a projection of T based on certain set of attributes ie Split with parameters the attributes of table A & B when applied to a table T, breaks T into two halves one matches the schema of A and the other that of B. With the Split Operator and the canonical form in hand, the query which when applied to the base table, all the derivations for the tuple one is looking for is derived. Such a query is called a tracing query.

If D is a database with base tables $R_1 \dots R_n$ and V= v(D)= $\pi_A(\sigma_C(R_1 \bowtie R_2 \bowtie ...R_n))$ a view and v is the view definition query. For a tuple t $\in$ V, data linega for t in D can be computed by applying the query D can be computed by applying the query $Split_{R_1...R_n}(\sigma_C \wedge A = t(R_1 \bowtie R_2 \bowtie ...R_n))$

However when we include aggregation operators in the view definition query, deriving data lineage for a tuple from the view the query produces is not simple without storing or computing some intermediate results. Although using a views derivation, it is possible to trace lineage for a data one operation at a time based on the original view definition, it requires the storage of every intermediate results. Once a simple one-level Aggregation-Select-Project-Join (ASPJ) query - $\alpha_{G,aggr(B)}(\pi_A(\sigma_C(R_1 \bowtie R_2 \bowtie ...R_n)))$ is transformed into its canonical form, the lineage for a tuple t produced by the ASPJ query can be found by carrying out the following query on the base tables:

$$Split_{R_1...R_n}(\sigma_C \wedge G = t.G(R_1 \bowtie R_2 \bowtie ...R_n))$$

For finding the lineage of a tuple formed by a multilevel ASPJ query, the query must be transformed into its corresponding canonical form, divide it intoa set of ASPJ segments and define and intermediate view for each segment. This intermediate view acts as the base table on which the lineage tracing query is applied.

For union operator, given $t \in T_1 \cup T_2 \cup T_3$, each tuple from any input table contributes to t. However, in a difference operator, given $t \in T_1 - T_2$, the tuple t from T_1 and all tuples from T_2 contribute to the lineage of t.

All the different approaches mentioned in this section provide a source of inspiration to some new ways of performing lineage which suits the environment of RALT. From these systems we get an idea of the pitfalls we should be avoiding when implementing the data lineage feature in the system. The scale of our learning tool implemented in this project is not in par with commercial data warehouse systems, making the lineage problem simpler.

Unlike in [13], showing any data in a lineage derivation which the user has deleted previously will only raise confusion in the users mind and hence if a tuple is deleted by the user in the GUI it also deleted from our actually system. In our system we also keep a track of the sequence of the operations carried out while building a kept. In our application we will take the approach of implementing our own function similar to the tracing queries method mentioned in [10] .

In Section 5.3 we explain how Data Lineage is implemented in our system in detail.

3.5 Development Environment

3.5.1 Technologies

Java

Our aim was to develop a portable, extensible, highly interactive system which eases the process of learning Relation Algebra. We realised very early that the project would take a considerable amount of time to design, implement and test, thus to get a swift start to the project, we chose to give priority to technologies that we were already familiar with that would sufficiently satisfy our goals. We would however develop into new technologies wherever we found our knowledge or skill set to be inadequate to achieve an objective. The final decision was in favour of Java for a number of reasons such as:

- It is a highly portable language and Java programs can be installed in systems supporting different operating systems (Windows, Linux etc.)
- Garbage collector is an important of feature of Java which provides automatic memory management.
- Numerous open source libraries are available for java together with countless online tutorials on different aspects of Java language.
- Documented Network,IO library available via Java.
- Swing a popular GUI toolkit is based on Java assists in the development of sophisticated user interface.
- My previous experience with JAVA made me feel more comfortable with it than any other popular object oriented language.

Java2D

Java2D [17] is a popular Java API for drawing two-dimensional graphics. It is a core element of Java technology. Every Java2D drawing operation can ultimately be treated as filling a shape using paint and the composition the result onto the screen. RALT relies heavily on Java swing for simple graphical output.

SVN

No project of this scope can be realised without a suitable version control system. Although such tools are particularly useful in group projects as it prevents team members from overwriting each others code, they are invaluable for individual projects also. They allow the developer to track the changes made in with previous versions of the system and if necessary it also allows the user to revert them.

Testing Framework: JUnit

JUnit is a regression testing framework written particularly for the Java programming language. Important features of JUnit include:

- Assertions for testing expected results.
- Test suites which allow organising and running tests easily.
- It uses a graphical interface to alert the user once the code shows any anomaly.

Eclipse

To aid in coding development we used the Eclipse IDE. This provided us with a comprehensive set of tools with which writing code was made more effective and efficient. The main tools used include a class browser, built in debugger, syntax checking and highlighting, code completion and suggestions. It also integrated well with our version control system. Eclipse also supports some state-of-art software engineering features such as:

- Code Refactoring. Many complex operations such as renaming a variable/method or moving a class to a different package can be done instantly.
- The source code can be easily formatted which makes reading the code easier. the programmers expected behaviour.

Chapter 4

User Iterface

A powerful user interface is one of the key advantages of our system. This chapter introduces the main features of our Graphical User Interface (GUI) and explains their purpose. We start with a brief introduction to Java Swing (Section 4.1 4.2)and then see how various components of Java Swing have been used for building different components of the GUI together with their respective functionality.

4.1 GUI Toolkit

A good user interface is very important for a learning tool as it plays a key role in transforming an average learning tool into a superior one. Hence it was important for us to design a powerful user friendly Graphical User Interface (GUI). As we had decided to choose Java as our core development platform, we leaned towards the idea of designing the GUI of the application using components of Java developed for designing user interface. Java offers more than one alternative toolkits for developing GUI - Abstract Windowing Too(AWT), Swing, Standard Widget Toolkit(SWT).

AWT offers only a basic set of user interface components, which does not allow the creation of a user interface rich in features. This left us to with only two choices - Java Swing or SWT. *Swing and SWT: A Tale of Two Java GUI Libraries*, an article by *M. Marinilli* [18] helps us in making a decision about the toolkit to choose for our application. According to the article, the main advantages of Swing over SWT are its larger set of features, more elegant appearance and higher abstraction level (which is very helpful especially when complex GUIs are designed). However the ease with which SWT can be used made it a potential alternative. But the flexibility Swing offers and the complexity SWT generates when building complex GUI made the decision swing in favour of Java Swing.

4.2 Benefits of Java Swing

Java Swing was developed for providing a sophisticated set of GUI components It is completely written in Java and this makes the development of Swing application faster as we were already familiar with Java and werr not required to learn any new language before using Swing's features. An important aspect of Swing that it allows the alternation of every component (for example changing the colour of a component) in an application without the need for making substantial changes to the application code. Also it is easy to change the look and feel of any application written in Swing which makes it look very different from default settings of the components.

Some of the components Swing supports is shown in Table 4.1:

Component	Functionality
JButton	Used for submitting user requests.
JLabel	Displays text information.
JList	Shows items in a list form.
JProgressBar	Used for displaying the progress made before an action is completed.
Scale	Helps in zooming in and out of a component.
JTextArea	Allows users to enter text spanning more than one line.
JTree	Displays items in tree-like structure.
JMenu	A menu provides a space-saving way to let the user choose one of several options.
Toolbar	Used for displaying common used actions.
JTable	Displays information in a table format.
JPanel	Used as a container to which other components can be added.
JScrollpane	Makes the component added to it scrollable.
JDialog	Used for displaying a dialog box. Acts like a JPanel and can contain Java Swing components such as JTable, JLabel, JList etc.

Table 4.1: Features supported by Swing

It is important to mention that Java Swing follows a component-based framework. The distinction between components and objects is fairly subtle - components are seen as well-behaved objects which are aware of their specified characteristic pattern of behaviour. Also Swing provided components which acts as containers to which items can be added easily. For e.g. *JPanel* is container to which items of different types such as JLabel, JTable can be added easily.

Another important concept of Java Swing is the *Layout Manager*. *Layout Managers* are often used for determining the size and positioning of elements within a container. Swing provides a number of *Layout Managers* such as *FlowLayout, GridLayout* etc. which lays out components in a container in various ways. Different layouts are used for various purposes. In a *FlowLayout* when components are added to the container, they are placed one after another in a horizontal way. When placing a new component in a container will exceed the width of the latter, the component is placed in the next line. *GridLayout* is used for displaying components of same size in rows and columns.

However the *Layout Managers* provided by Java were not enough for designing our GUI and we opted to use a more customised *Layout Manager* known as the *Spring Layout*. Spring layouts do their job by defining directional relationships, or constraints, between the edges of components. For example, we can define that the left edge of one component is a fixed distance (5 pixels, say) from the right edge of another component. *SpringLayout* can be visualized as a set of objects that are connected by a set of springs on their edges.

Unfortunately Swing has its drawbacks too. It is extremely difficult to debug Swing applications due to the toolkit's visual nature. We cannot take advantage of the step-by-step debuggers like we do when debugging objects.

4.3 User Interface in RALT

Having given a brief introduction to Swing, we can move forward and understand the functionality of the different components which make up the user interface of RALT. The main functionality of RALT is that it allows users to build Relational Algebra queries without requiring them to write any queries themselves. Here we are faced with a dilemma as we would like to enable users to conveniently carry out the task of building queries, yet at the same time keeping the interface simple and avoid overloading it with any unnecessary detail which raises confusion in the minds of the user.

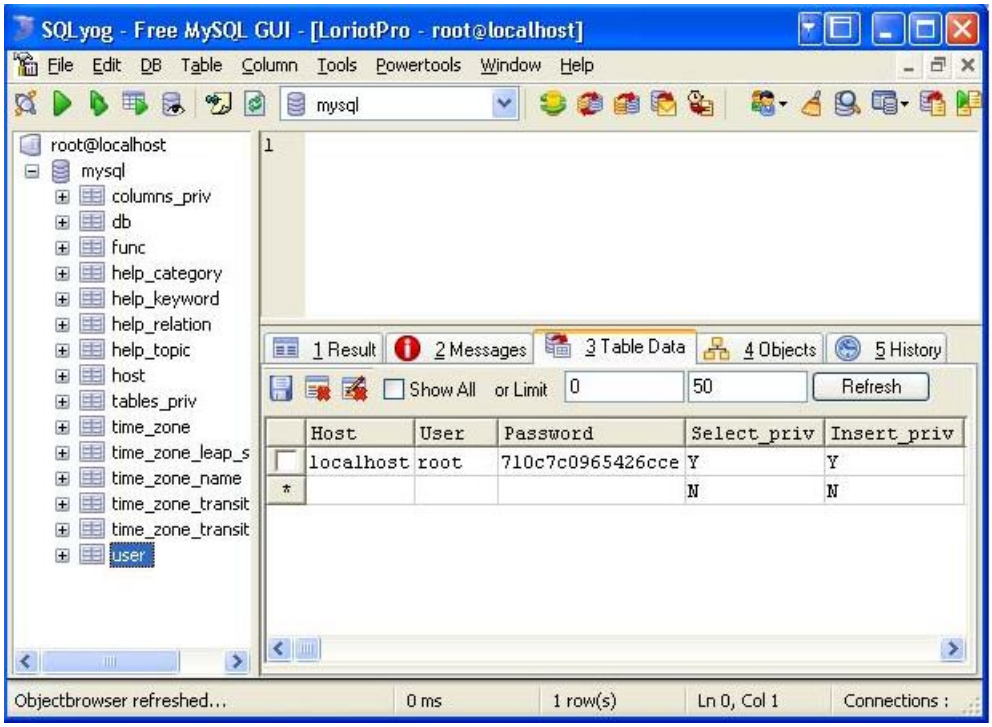


Figure 4.1: SQLYog

We consider the software SQLYog-a GUI tool used for popular Relational Database Management Systems. As we can see from Figure 4.1 this application allows users to visualise different tables of the database the system is connected too. Notice the left panel displays the tables in the databases the user is connected to. For each database the system is connected to, the software displays the table names present in that particular database together with the attributes of the table (attribute names etc.). The upper part of the right panel allows users to write the query they want to execute while the bottom part displays contents of the query (i.e. the result in the form of a relational table) generated by the system. SQLYog inspired us to design a simple interface as shown in the Figure 4.2

Figure 4.2 displays the different components which make up the GUI. Each of these individual components are labelled from **A** to **F**. Each of these components are designed independently and then added into container component thanks to the adding component feature of Java Swing Components (e.g. JPanel) . We will now give a brief description of the major components of our GUI.

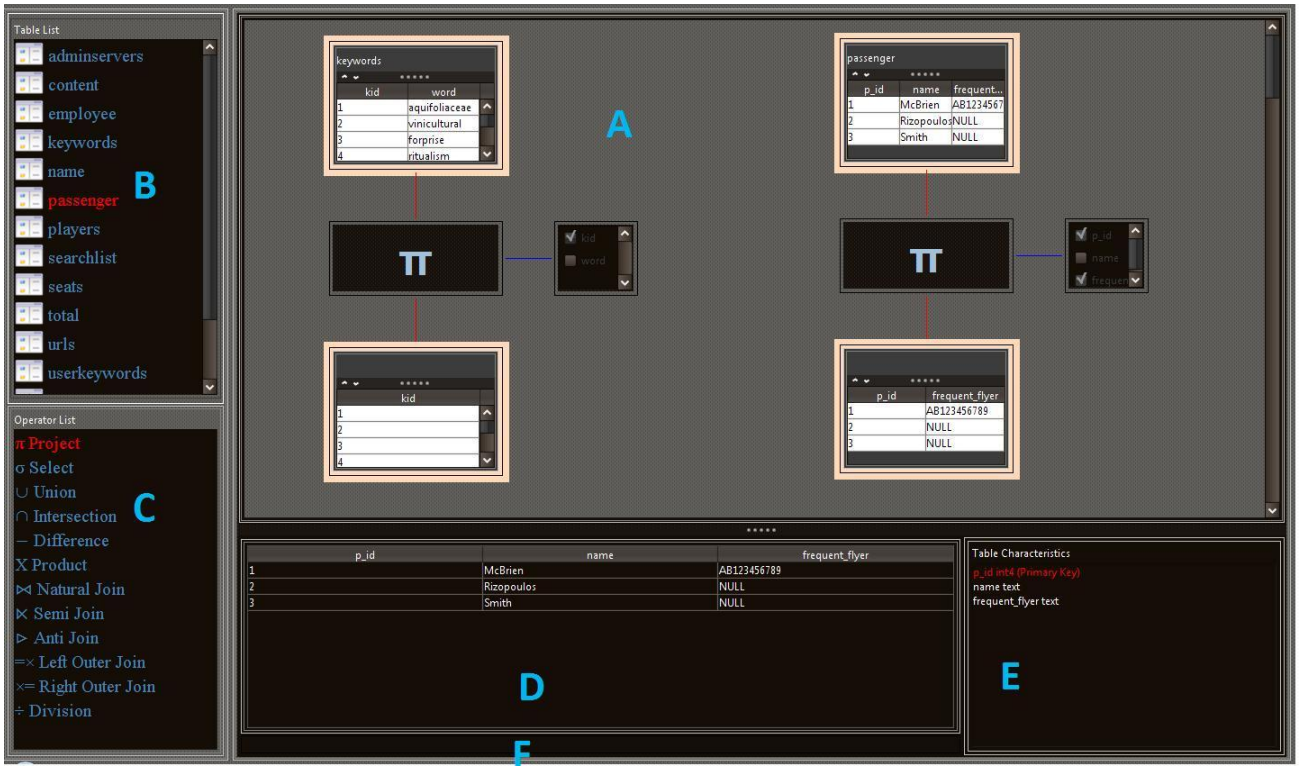


Figure 4.2: User Interface for RALT

Table Name Panel

Labelled as **B** in the Figure 4.2, this panel is used to display the names of the different tables present in the database to which the system is connected to. We used JList component of Java Swing for displaying the items in a list format and adding the JList into a JScrollPane component to make the list scrollable.

Operator Panel

Designed in a similar way as the Table Panel, the Operator Panel (marked **C**) displays the different operators in the system. Each operator name is assigned with a symbol which corresponds to the symbols used when constructing Relational Algebra queries using the algebric notation. This allows users who are used to developing queries with the algebraic notations, to quickly get used to the system. Moreover, since icons are used to identify operations when building queries in the Query Visualisation Panel (**A**), users can easily cross reference the symbol with the operator name in the Operator Panel,if need be.

Table Data Display Panel

Marked as **D**, this panel is used to display the components of a table. Java Swings drag and drop feature is used for dragging table name from the Table Name Panel and dropping them on this panel when the contents the database under this table name is displayed in the panel. We use the JTable feature of Java to display the contents. We embed the JTable into a JScrollpane component to make the table scrollable.

Table Characteristic Panel

This panel is used for displaying the properties of the table i.e. the types of the attributes of a table, whether the attribute is Primary key to the table or it is a Foreign Key. If the attribute is a Foreign Key we also display the table name whose Primary Key acts as a Foreign Key for this table. This panel is labelled as **E**. When a table name is dropped on the Table Display Panel, this panel automatically displays the properties of the table that has been dropped. This panel is designed in the same way as panel **B** & **C**. However in this panel we use different colours for indicating different properties (e.g. Red for Primary Keys). Bright colours are chosen for displaying important properties as it gets users attraction instantly.

Error Console Panel

Marked as **F** this error displays any error messages the system thinks the user should know about. The message is shown by using the JLabel feature of Java Swing.

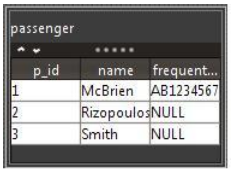
Query Visualisation Panel

Marked as **A**, this is most important component in our GUI. Users build their queries in this panel. This panel is of type JPanel whose Layout Manager is set to **null**. This is a special kind of Layout Manager and is required in this case since it gives us the liberty to position elements on the JPanel to the coordinates we set. This panel accepts requests from the Table Name Panel and Operator Panel using Java Swings drag and drop feature and acts accordingly. When building a query in this panel, each element of a query (table names, operators, constraints) are created as independent JPanels and

then position them at desired co-ordinates on this panel. These individual JPanels are then connected by straight lines which can be drawn using Java2D.

(a) QueryTreeOperatorPanelGUI

(b) QueryTreeConstraintsPanelGUI



(c) QueryTreeTablePanelGUI

These JPanels can be classified into two three types:

- QueryTreeTablePanelGUI – Used for displaying contents of a table.
- QueryTreeOperatorPanelGUI - Displays the symbol for the operator.
- QueryTreeConditionPanel – Shows any condition attached to operators like Project, Select.

We sometimes require users to input additional information when building a query. For example users should specify the columns to be selected when carrying out a Project query or the constraints based on which the Select operation should be carried out. This additional information is captured by using Swing feature Jdialog, components used for display dialog boxes.

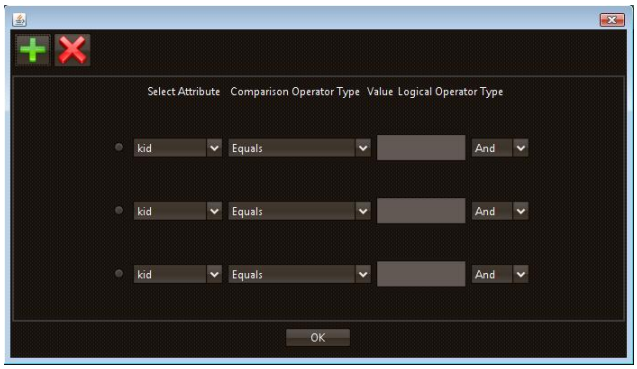


Figure 4.3: Dialog Box used for entering constraints for SELECT operator

Chapter 5

Architecture

The chapter opens with a broad overview of the system's architecture, then focusses on specificc subsystems. Architectural considerations are illustrated by UML diagrams. In this chapter we also look at some specifc classes and packages. Lastly, we consider how a query tree is stored in our customised data structure and then get an idea how the lineage of a row is traced in our system.

Figure 5.1 shows a high level overview of the software's architecture. RALT consists of a number of subsystems, each concerned with a particular aspect of its functionality. In accordance to the good software design processes, the subsystems were designed to be logically coherent and to minimise the number of connections (coupling) in the system. This practice was applied in the lower level of the of the design (packages, classes). There is a high degree of correspondence between the subsystems and the Java packages (particularly in the section responsible for the designing View of the system).

5.1 System Architecture

As it can be easily seen from Figure 5.1 , we have followed the popular Model-View-Controller (MVC) architecture paradigm. The beauty of this genre of architecture is that it allows the separation of business logic from the user interface design. This makes this application modular as it allows modification of the visual appearance of the application without causing disturbances to its business logic. In MVC architecture, the *model* represents the information i.e. the data of the application. The *view* corresponds to the interface which enables the user to interact with the system. The *controller* is responsible for communicating requests made by the user at the *view* layer to the *model* level and then taking back the data to the *view* once it has been manipulated according to the business rules of the system. Adopting such architecture helped in speeding up the development process, particularly in the beginning. We could simultaneously develop the GUI of the application as well as implement the rules governing the application.

We now explain the role of each layer together with the components it is made up of in detail:

5.1.1 Model Layer

The model layer also called the Data Access Object Data (DAO) of RALT implements the business logic behind the application. Distributed over the packages *com.imperial.dao.opt*, *com.imperial.dao.lineage* and *com.imperial.dao.utility*, the model layer acts as an interface between the actual data and the user

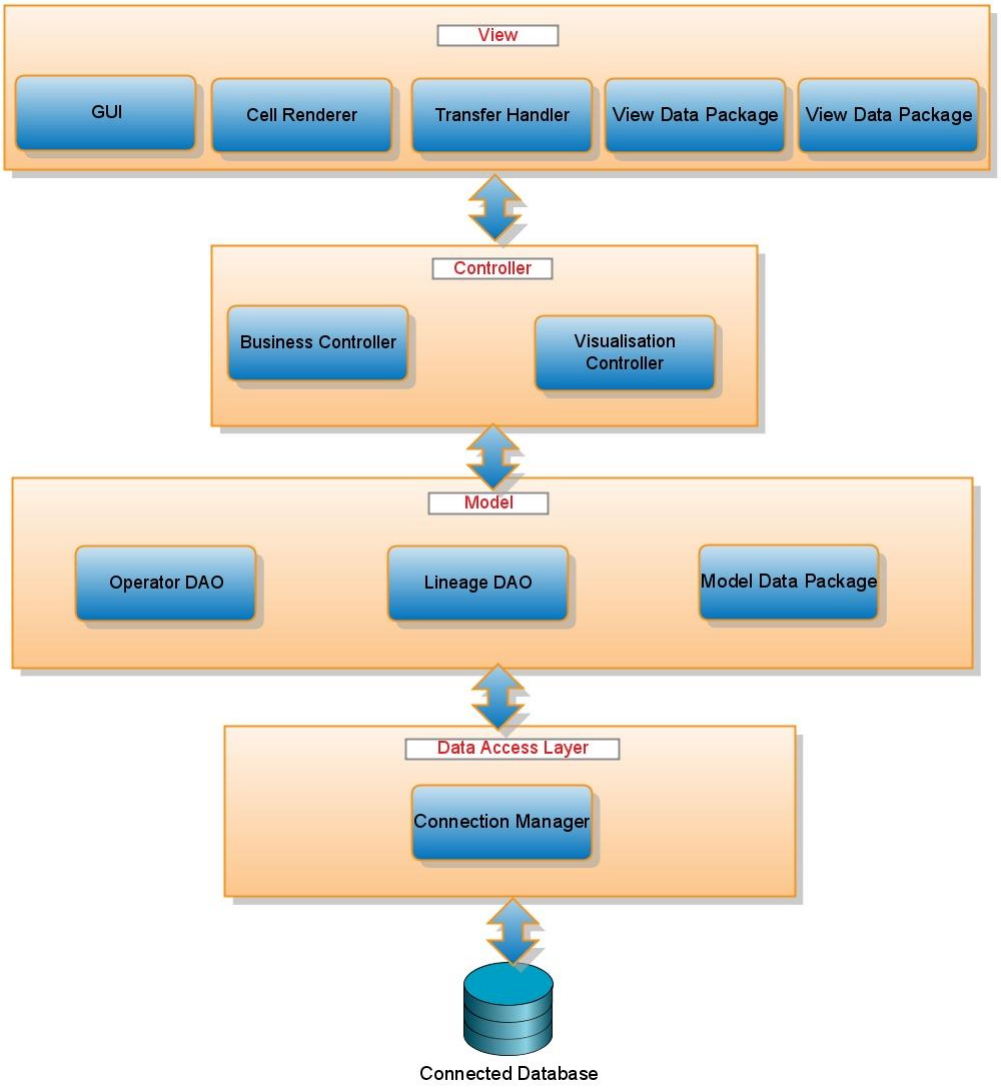


Figure 5.1: System Architecture

interface. This layer receives requests from the Controller layer, depending on which acts on the data and then returns the back to the Visualisation Controller.

Data Storage

An important component of the Model Layer is the way data is stored in the system. We have modeled our system such that our implementation of the Relational Algebra operators take data input which are similar in structure to the tables found in relational databases. We have followed the same approach when implementing the data lineage operators. Moreover, we have decided to store all the data of the database in memory so it was important for us to come up with an effective way of storing data in our memory with all the relation needed for our system.

We noticed some similarities between table cells when observing how data is represented in database tables which were helpful when we designed our own data structure for storing table information in our memory.

sortcode	bname	cash	no	type	cname	rate	sortcode
56	Wimbledon	94340.45	101	deposit	McBrien, P.	5.25	67
56	Wimbledon	94340.45	119	deposit	Poulovassilis, A.	5.50	56
34	Goodge St	8900.67	101	deposit	McBrien, P.	5.25	67
67	Strand	34005.00	101	deposit	McBrien, P.	5.25	67
67	Strand	34005.00	119	deposit	Poulovassilis, A.	5.50	56

Table 5.1: A general relational table

As we seen in Table 5.1 top most row displays the header columns of the database table. The rest of the rows display the data contained in the table. We can see that there are some similarities between the cells representing the header information and the cells representing the data of the table. Both these two types of cells store a value. The cells in the header of the table are equipped with additional information such as the type of attributes in that column (i.e. integer or varchar), whether the column acts as the primary key of the table etc.

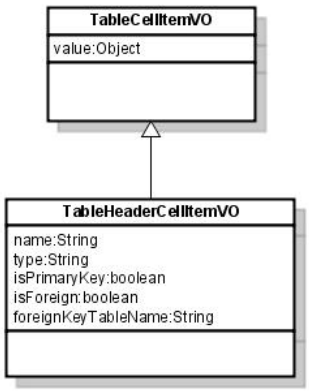


Figure 5.2: Class TableCellItemVO used for storing cell data and TableHeaderCellItemVO used for storing header information

In Figure 5.2 we can see that we have created the class called TableCellItemVO which can be used for holding value in a cell of a database table. This class contains one attribute of type Object (root of the class hierarchy in Java). This class represents the data held in each cell (each cell of the header row as well as each cell of the rows) of a database. However, extra information is required when storing the headers of a table and for that we extend the TableCellItemVO class to create the Java class TableHeaderCellItemVO.

Each row is seen as an array of TableCellItemVO (e.g. TableCellItemVO [])which is represented in the class RowItemVO.

We can combine two classes mentioned above to create a representation of a database table as shown in the class diagram in Figure 5.3. The *name* attribute of class TableVO is used for storing the name of a table. The *header* attribute is used for storing column information.The last attribute *rows* is used for storing each row of a database table.

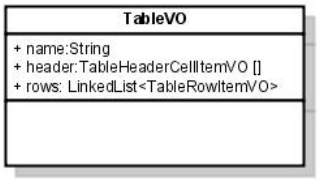


Figure 5.3: TableVO

Operator DAO

Having designed the data structure for storing table information, we could now implement the Relational Algebra operators which took objects containing table information as parameters (i.e. objects of type TableVO). This part of the model layer implements the different Relational Algebra operator

present in our system. In order to give our users the option of using some of the advanced operators such as Division, Anti Join etc. we went in the direction of implementing all the operators in our system ourselves.

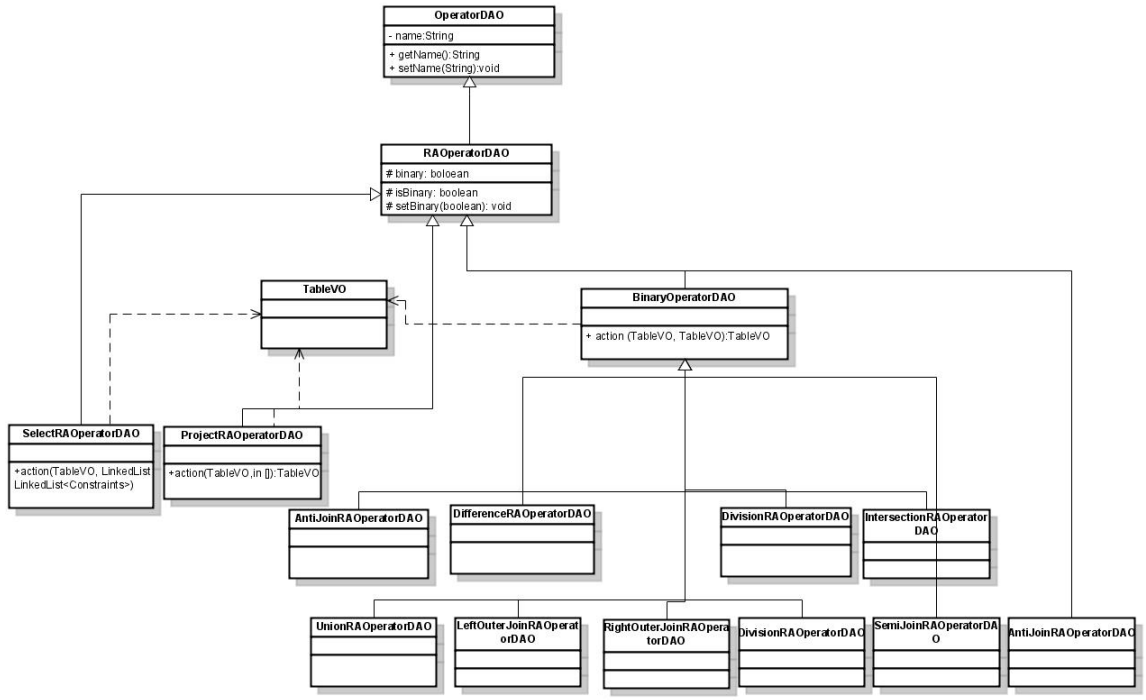


Figure 5.4: Class Diagram for Operators

We realized that the Relational Algebra operators to be implemented in our system fall under two different types:

- Unary Operators: Operators which take one table as input. e.g. Select, Project
- Binary Operators: Operators which take two tables as inputs e.g. Union, Difference etc.

We also realized that all the operators share some common properties such as name. Moreover, to make our system extendible easily, it is important that the framework should be such that users may add operators which are not of the type Relational Algebra (e.g. SQL operators) or add operators of different kinds such as aggregate operators (e.g. Count etc.) . This motivated us to come with a an OperatorDAO class which could be extended by an RAOperatorDAO (Relational Algebra Operator) class. BinaryOperatorDAO class extends the RAOperator which could be further extended depending on the type of operator we are implementing. Each operator has a method called *action* which implements the functionality specific to that operator. The fact that all the binary operators take inputs of same kind, this type of architecture became more appropriate for the model component of our system. This makes our model layer very modular as a programmer can easily add their own operators

to their system. The classes implementing the operators are located under the *com.imperial.dao.opt* package. The class diagram for the the operators are shown in Figure 5.4.

Lineage DAO

Data Lineage is a key feature of our system. Unfortunately the existing database systems, for e.g. Postgres SQL do not offer this feature. As a result we decided to implement this feature ourselves for our system. In our system, queries are built in a tree like structure. Hence, we are required to keep a track of the inputs to an operator together with the output it produces and any constraints associated with it. We used these available data for implementing functions for finding the lineage of each Relational Algebra operators which we discuss in detail in section 5.3. Similar to the way we implemented each of the Relational Algebra operators, we approach the same method for implementing the linegae for each Relational Algebra operator. As an example, for the Union operator we implemented a class UnionLineageDAO.java in the package *com.imperial.dao.lineage*. This approach again makes our system very modular as it allowed a programmer to easily add the lineage functionality of a new operator he/she has added to the system without disturbing the contents of the other lineage operators already present in the system.

5.1.2 View Layer

With our model layer being able to modify information according to user requests, we needed a way to display this information to the user. We could do this with the help of the View Layer. As the name suggests this layer forms the interface between the user and the rest of the system. This layer is responsible for displaying all the different types of information to the user and also recognises the requests made by the user. Hence this layer is equipped with all the functionalities required to accomplish these objectives. The main components of the View layer are:

GUI

The GUI component is responsible for displaying information to the user. It also accepts requests made by the user at the user interface. The GUI is itself again divided into several sub-components. Figure 4.2 shows the the main GUI screen of RALT. The labels **A,B,C, D, E** and **F** indicate the individual components which make up the GUI for the application. The beauty of Java Swing is that it allows us to build components separately and then add them to another component. For e.g. we can build the components labelled as **A,B,C, D, E** and **F** as shown in Figure 4.2 individually and then integrate them into single component. This way we are able delegate specific functionalities to the sub-components. Each of the sub-components is responsible for display of the data it is supplied with.

In Figure 5.5 we see a breakdown of the different subcomponents which make up the GUI of our application. Each GUI component is mapped to a Java class (the java class for the GUI component Table Name Panel is TableNamePanelGUI.java). These classes are located under the package name *com.imperial.gui.panel*. The subcomponents (Query Visualisation Panel, Table Data Display Panel, Table Characteristic Panel, Error Console Panel) once designed independently are added to the swing components (JPanel) Right Panel GUI and Left Panel GUI. In the end, Left Panel GUI and Right Panel GUI are combined together to form the Application GUI.

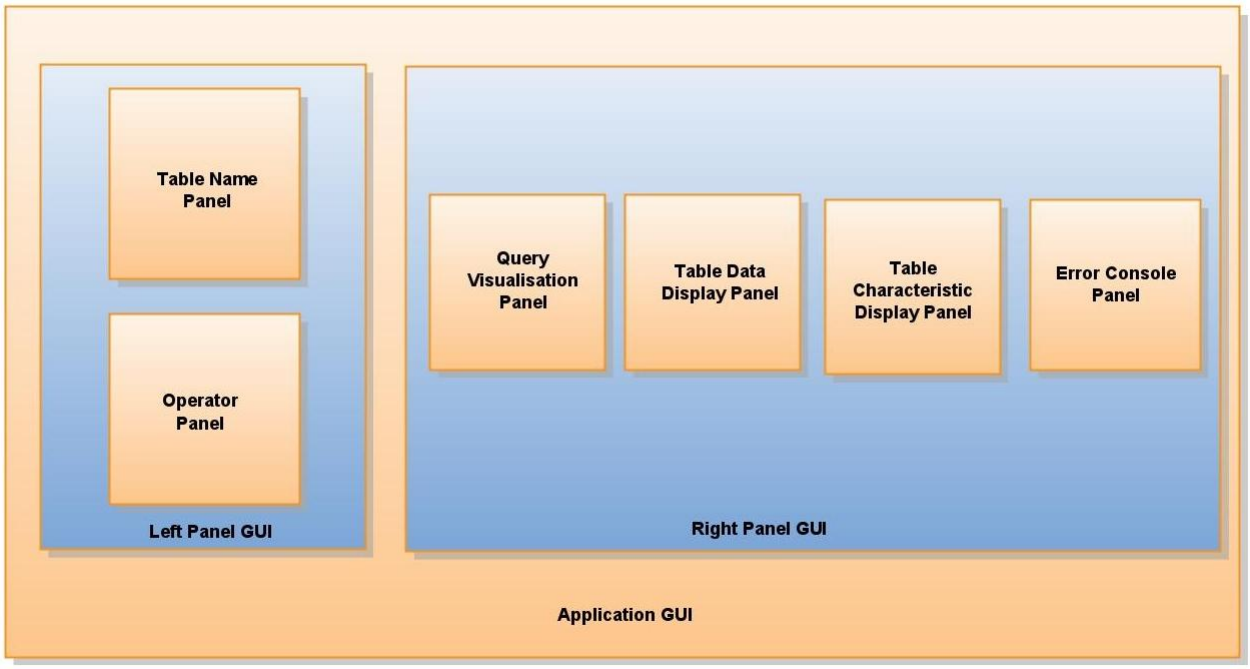


Figure 5.5: Sub-components of Application GUI

Transfer Handler

An important feature of RALT is that it allows users to build Relational Algebra queries just by dragging table and operators name(s) from panels labelled **B** and **C** in Figure 4.2 and dropping them on the Query Visualisation Panel (labelled **A** in Figure 4.2). This is achieved by using the drag and drop feature supported by Java Swing.

Before progressing any further we provide a small step by step example of how the drag and drop feature works in our system. [19].

Let us assume the user is interested in dragging a table name from the Table Name Panel (labelled B in Figure 4.2) and drop it onto the Query Visualisation Panel (marked as A in the same diagram). The Table Display Panel displays the names of all the tables present in the database RALT is connected to. The table names are displayed using the JList feature of Java Swing and each component of this Jlist is a List Component.

In a nut shell the drag and drop feature works in the following way:

- As the user starts to drag the selected List Component from the Table Name Panel, a drag gesture is initiated.
- As the drag begins the JList packages up the data(which is of type String) into a customised form.
- As the user continues to drag the list component across different components of the screen, Java Swing continuously calculates the location and provides any rendering required.

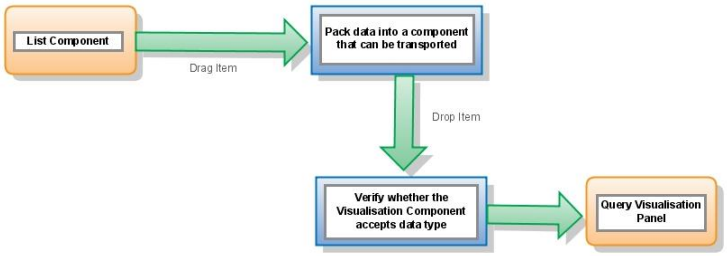


Figure 5.6: Workflow for Drag and Drop feature

- Once the user releases the component over a particular component (say the Query Visualisation Panel in our case) we inspect whether the component on which the list component is dropped, should accept the data (of type String since this is what is present inside the package)which is being packaged.
- If yes, the data is imported and the necessary actions are taken.

This example shows us the need to equip each of the components of the GUI that supports the drag and drop feature with custom transfer handler functionalities. This is possible as all JComponents of Java Swing (such as JPanel, JTable, JScrollPanne, JList etc.) support the feature of drag and drop. In package *com.imperial.transferhandler* we have declared the different customised TransferHandler classes which extend the default class (TransferHandler) provided by Java Swing for implementing the drag and drop functionality.

Cell Renderer

Often we decided, to modify the default appearance of Swing components such as JList, JTable etc. This could be achieved by writing our own customised CellRenderer classes. Located under the package name *com.imeprial.cellrender*, these classes allowed us to design each component of the GUI to our specific needs.

View Data Package

Often in RALT we are required to display data in particular formats within a particular component. Query Visualisation Panel is a perfect example for explaining this idea. In Figure 5.7 the Query Visualisation Panel is made up of components displaying different data such as JTable, image of an operator or constraints for an operator.

We have designed separate classes for displaying each different kinds of information especially for the GUI and they are stored under the package name *com.imperial.vo.gui*.

5.1.3 Controller

With the View and the Model layer at hand, we needed a mechanism that would allow these two components to talk to one another. This is where Controller comes in. This layer forms the interface between the GUI and the Model of the application. It accepts the request made by the user at the GUI and communicates them to the Model. Once the Model layer perform the necessary operations,

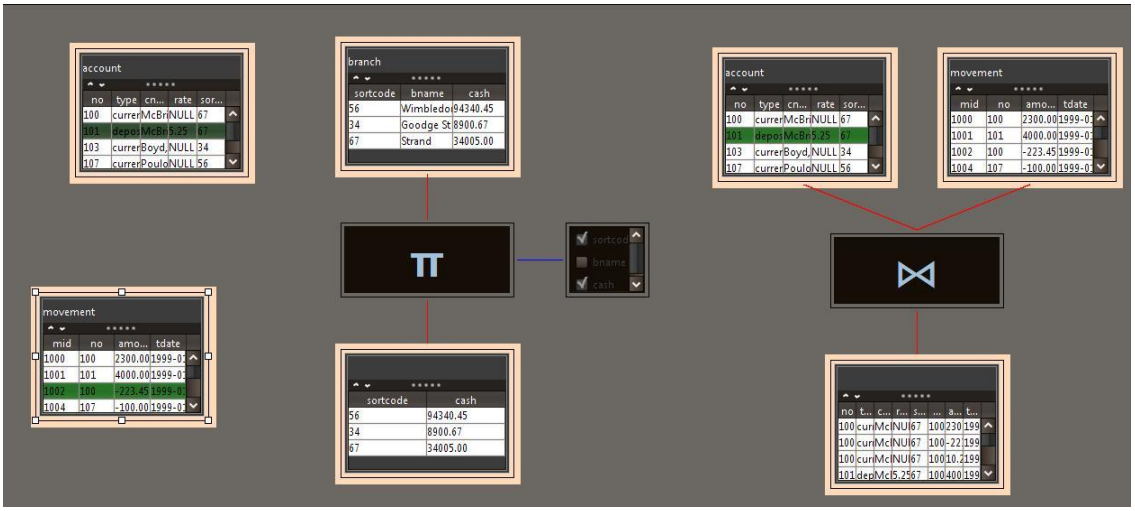


Figure 5.7: Query Visualisation Panel

it sends data back to the Controller layer. The Controller passes the data to the View layer either unchanged or after performing some necessary modifications. Due to the wide variety of functions it performs we have decided to split the Controller layer into two components Visualisation Controller and Business Controller.

Visualisation Controller

This part of the Controller speaks to the View Layer. It receives request from View Layer and depending on the nature of the request it either passes on to the Business Controller or handles it itself. The Visualisation Controller keeps a track of the different panels in the application and is hence able to pass messages to them. This layer plays an important part of the query building process. Although we discuss the query tree structure in detail in Section 5.2, it is important to mention here that Visualisation Controller plays an important role in building the queries which are displayed in the Query Visualisation Controller. It determines how new nodes should be added to the tree and also determines their position when displayed on the visualization panel. It is the sole duty of the Visualisation Controller to maintain the correct tree structure as otherwise users may get wrong perception from the query tree being displayed on the table. Moreover, as we will see later on having the right query structure is crucial since our data lineage algorithm is highly dependent on it.

Business Controller

This is the second part of the Controller layer which mainly takes orders from the Visualisation Controller and acts accordingly. When the application begins this layer reads the information for all the tables present in the database to which the system is connected to and stores them in the memory. Later on whenever the user wishes to view the contents of a particular table or wants to operate on them, the Business layer passes a representation of the table.

5.1.4 Data Access Layer

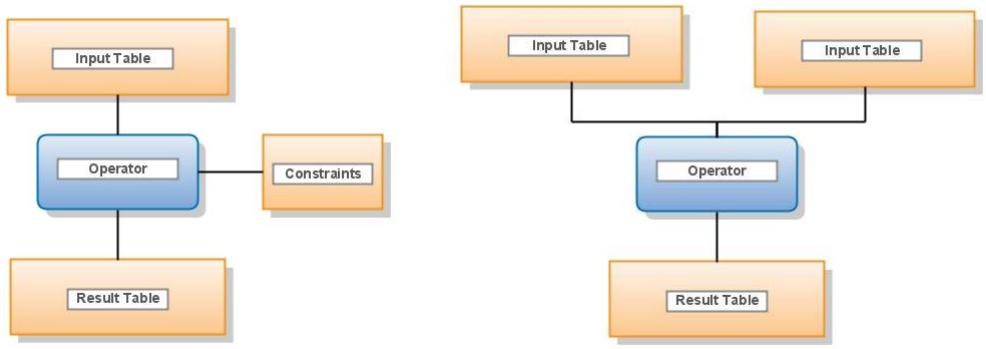
Data Access Layer: This layer handles the functionality of connecting to a database and fetching data from the database.

5.2 Query Tree Architecture

Having discussed the different layers of our system, we now discuss how we approached the process of building queries in our system.

One of the key features of our application is that it allows users to build queries in a tree like structure. This approach enhances the understanding of the Relational Algebra concepts as the user can easily see the step by step approach undertaken in building the query together with any intermediate results being produced. When building a query the main task is to find the operation sequence that will produce the correct result. Since the operations of the Relational Algebra are quite simple, many intermediate results might have to be produced before the final result is reached. The intermediate results are used as operands in the operations that produce new intermediate results. As a result we are required to store the sequence of operations carried out in the query building process.

Our system supports two types of operators unary and binary. In Figure 5.8(a) and 5.8(b) we see a diagrammatic representation of presenting these operators in a tree form.



(a) Unary Operators (b) Binary Operators

Figure 5.8: Tree like representation for different Relational Algebra queries

As we can see in Figure 5.8 each node (represented by rectangular box) has at most two child nodes. Moreover each node can have at most two parent nodes. We were required to come up with a data structure that will allow us to implement the functionality for assisting users build queries in a tree like structure. The tree like structure when formulating unary and binary operators for Relational Algebra motivated us to devise our own data structure for this purpose.

We propose a data structure, in which we treat each item (input table, operator, result table or constraints) in our query as a node. We call this Query Node as shown in Figure 5.9. A query is made

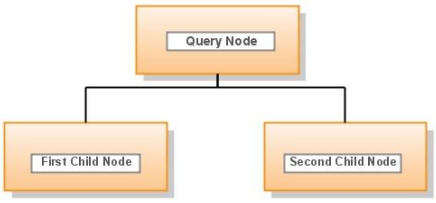


Figure 5.9: Query Node

up of a finite set of nodes which is either empty or consists of a data item (called the Query Node) and two disjoint Nodes (called the First Child Node and Second Child Node). Each node can have at most one parent. The trees are hierarchical in nature indicating that there exists a parent-child relationship between the nodes of a tree. Having such a relationship between the nodes of a tree was important as it allowed us to show the step by step approach taken when building a query. The first node in a query building tree (the node without any parent) is called the Root Node.

With such a data structure, we were quicklt able to build more complicated queries than the ones shown in Figure 5.8, by using result from one Relational Algebra operation as an input to another relational algebra operation. Let us take an example of query formulated by carrying out the following steps:

- Performing a Natural Join on input tables Input Table1 and Input Table2. The result of this operation is ResultTable1.
- Perform a Select operation on the ResultTable1, with the constraint $attribute1 > 0$(assuming ResultTable1 has column name attribute1 and of type integer). This operation produces the resultant table ResultTable2.
- Perform a Semi-Join operation on the ResultTable2 together with InputTable3 to produce the table ResultTable3.

The above query when executed in our system is stored using the data structure displayed in Figure 5.10 in the following way:

In Figure 5.10 the tables are represented as orange rectangles, the operators as blue Rounded Rectangles, the Constraints as green rectangles. Each of these three items in our query structure is treated as a node (Query Node).The red line between two nodes in Figure 5.10 is the relationship between a parent node and its first child node. For e.g. the red line between Input Table1 and Input Table2 means that Input Table2 is the first child of Input Table1. The black line between two nodes indicates the parents relationship with its Second Child Node.

Hence as shown in Figure 5.10 our new data structure allows us to store all the necessary information required to support the methodology of creating queries in a tree like format. In RALT users are able to build more than one queries at the same time which are completely independent of one another. We maintain a list of all the individual queries built in the application as shown in Figure 5.11. Each element of the list refers to the Root Node of a query which is unique and displays a table.

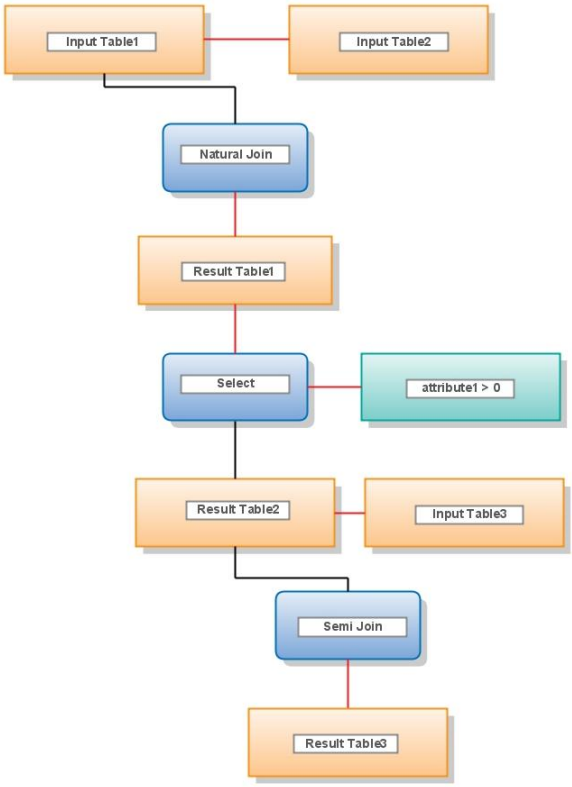


Figure 5.10: Diagram shows how a query tree is stored in RALT

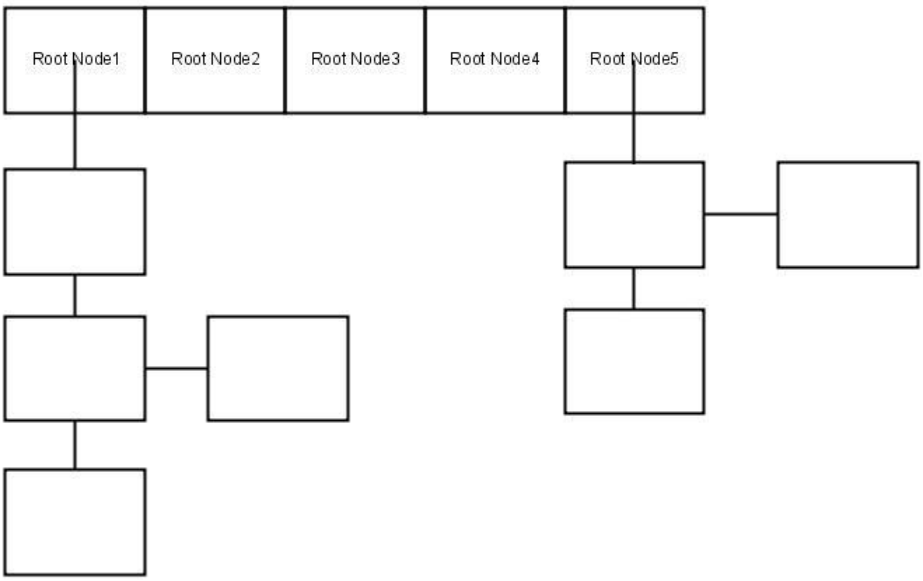


Figure 5.11: Storing multiple query Trees

5.3 Data Lineage

Data Lineage is an important feature of RALT. Having such a feature allows users to quickly track how a particular piece of information in a query is derived. It is important to mention here that we have decided to undertake data lineage at a tuple level than at the row level i.e. when performing data lineage, users can see how a particular row of a database table is derived and not how the cell of a particular row is derived. In Section 3.4 we explained different ways of performing data lineage.

As the feature of data lineage was a late addition to our system we had to make the best use of the existing architecture of RALT. Fortunately the modularity of the system made the task of integrating this feature without creating many development issue. In RALT, for visualisation purposes, we keep a representation of the different components i.e. base tables, operators, constraints and intermediate result of a query data flow. As the developer is in control of the data structure used for storing data in RALT, it can be easily modified to accommodate address of the source of a data. One way of performing data lineage can be of assigning unique ids to each tuple in a query. When a new tuple is created in the intermediate tables as a result of some operation, it is assigned a unique id. For each query being created, an index table is kept which keeps a track of all the $< child, parent >$ association. When we want to find the lineage of a particular row we can look up the index table and find the parent(s) and then perform data lineage on the parent row(s).

However, this method requires us to store additional information such as the unique id given to rows of a table. We implemented the data lineage functionality in RALT using a recursive approach which does not require storage of any additional information.We will explain this approach with an example. In Figure 5.12 shows the visual representation of a query which has been executed. First a Select Operation is executed on the relational table *Table1* and then a Natural Join is performed with the output from the previous operation(*Table2*) and *Table3* to produce the final result Final Table.

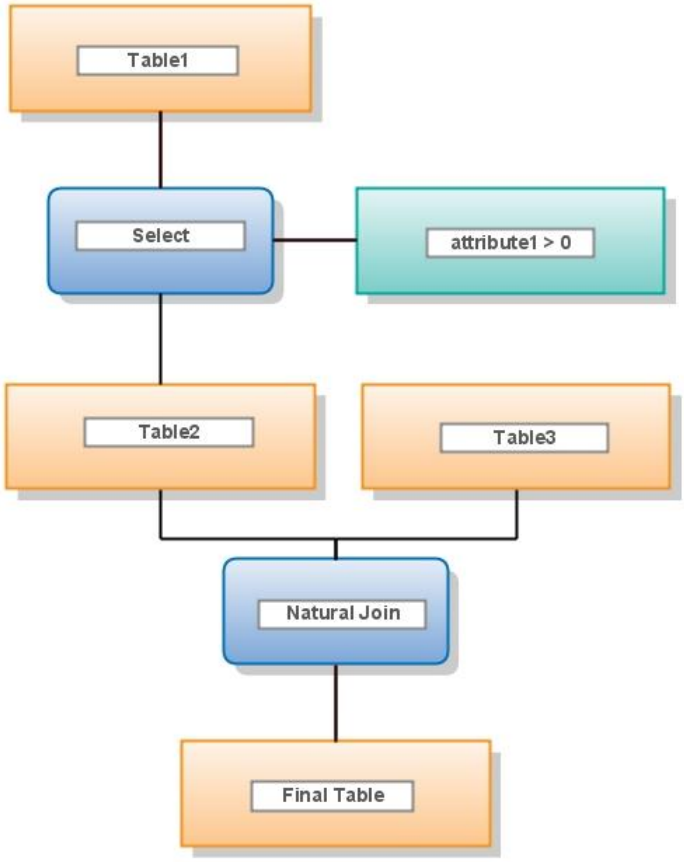


Figure 5.12: Explaining Data Lineage in RALT

We are now interested in finding how a particular row in table *Final Table* is derived. Due to data structure used to store information when building a query (Figure 5.9), we can access the node designated as Natural Join in Figure 5.12 since it is the parent of the node labeled *Final Table*. From this node we can retrieve the two input nodes for the Natural Join operation. As we had mentioned earlier, in section 5.1.1 we use our implementation of the lineage functionality for Natural Join operator to discover the rows of *Table2* and *Table3* that had had contributed in the derivation of our row of interest in table *Final Table*. Once the correct lineage row(s) for *Table2* is established, we can carry on similarly find the out the rows in *Table1* which had contributed in producing the lineage rows in *Table2*. *Table3* on the other hand do not have any parent and hence is not produced from any other table higher up in the query tree. Hence we have already determined the rows of *Table3* which have participated in producing our row of interest in *Final Table*. Hence we continue our propogation just with *Table2*. This propogation stops when the rows of *Table1* participating in the lineage are found and we are unable to move up the query tree anymore due to the absence of a parent for *Table1*.

Chapter 6

Implementation

This chapter describes some selected aspects of the program's implementation. It begins with describing the initial set of actions carried out when RALT loads (Section 6.1). We then go on explaining in Section 6.2 how tables from the database is fetched and stored in our memory and then illustrate our implementation of different Relational Algebra operators in Section 6.3. Section 6.4 explains how a query tree is built in our system which leads on to Section 6.5 where we demonstrate how Data Lineage is performed. Finally we end with Section 6.6 giving an workflow of how a programmer can add new operators to our system.

6.1 The Beginning

When RALT loads up it calls the ApplicationGUI class located under the *com.imperial.gui.panel* package. This class extends the JFrame class of Java Swing. JFrame is a top-level container which must be present in order to display a swing application. The ApplicationGUI when initialised also initialises instances of LeftPanelGUI and RightPanelGUI which in turn initializes their respective sub-components.The ApplicationGUI also keeps an instance of an object of type BusinessController and VisualisationController. These two instances are used for supporting the query building process as they communicate messages between the View Layer and Model Layer. The ApplicationGUI on its instantiation, requests the BusinessController to load the relational tables present in the database the system is connected to. Once this task is completed the VisualisationController is instructed to display the table names fetched by the BusinessController on the application screen. As each of the sub-components of GUI are initialised, they register with the Visualisation Controller. This way the Visualisation Controller can keep a record of all the different sub-components and send them information when need be.

6.2 Loading Tables from Database

Once the BusinessController receives the request for fetching the relational tables, it calls a method in the Model Layer. BusinessController invokes the *loadTables()* method of the TableDAO class located under the package *com.imperial.dao.utility*. Fortunately it is quite simple to fetch data from a database, thanks to the inbuilt methods provided by Java. However in addition to fetching just the column names and row data of a table, we were also interested in fetching additional information such as the Primary Key, column types etc.

```
String query = "Select * from tableName";
ResultSetMetaData rsmd = fetchData.getMetaData();
                        \\get the Metadata for the table tableName

ResultSet fetchData = fetchStmt.executeQuery(query);
                         \\ get the rows of the table tableName

ResultSet primaryKeysSet = meta.getPrimaryKeys(null, null,tableName);
                        \\ get the Primary Key of tableName

ResultSet foreignKeys = meta.getImportedKeys(null, null,tableName);
                        \\get the foreign keys of the table

                                          ...................
while (fetchData.next()){ // while there are more rows

                add rows to a list
}
```

This was possible by using the metadata associated with each table of the database. Metadata is nothing but data about data. An interesting fact to notice is that although in Java it is possible to fetch the column names and types while traversing through the list containing all the attributes of the table, we cannot identify which attributes represent the Primary Key or Foreign Key of the table. Fortunately, Java developers have provided us with the option of fetching this information separately by exercising the methods *meta.getPrimaryKeys()* and *meta.getImportedKeys()*. meta is the Metadata of the entire database and we fetch the Primary Keys and Foreign Keys from this database Medatadata by providing the name of the table we are interested in.

We keep a record of this information and when traversing through the list of all column attributes of a table and verify whether they represent the Primary Key or Foreign Key of a table. If yes, we set the appropriate value of object type in TableHeaderCellItemVO, used for storing header information of a database table in our system. *loadTables()* returns an object of type *HashMap*, which is stored at the BusinessController and is used for future references when requests are made by the VisualisationController. Each element of the *HashMap* is of type $\langle String, TableVO \rangle$, where the key represents the name of the table [1] and the value refers to the TableVO object which contains both the header information and row values of the table.

6.3 Implementing Relational Algebra Operators

We had mentioned in section 5.1.1 that a unique feature of RALT is that it implements its own Relational Algebra operators. Implementing these operators increases the flexibility of our system. Moreover, the modular approach taken when implementing these queries makes it very easy to add new operators into the system or even modify the existing ones without disturbing the functionality

[1] Since each table in a database must have a unique name, we have chosen the table name to be the key when storing information in a HashMap

of the operators already present in the system. However, it is important to mention that when implementing some operators we take advantage of using the functionality offered by operators already implemented in the system. This is done in order to minimise code duplication. For example, when implementing the functionality of the Left Outer Join, we make the use of the functionality of Natural Join, Semi Join and the Difference operator. We will explain this in more detail when illustrating the implementation process for each of these operators.We have tried to pay utmost attention to the problem of code duplication as having duplicated code increases the potential chance of bug in the system. Often when implementing the functionalities of Relational Algebra operators we use the same functionality again and again. This results in code duplication and we have opted to keep these functions under one class called UtilityDAO (shown in Figure 6.1)which is located under the package name *com.imperial.dao.utility*. Whenever we need to perform these functionalities, we initialise an instance of the UtilityDAO and invoke the required function.

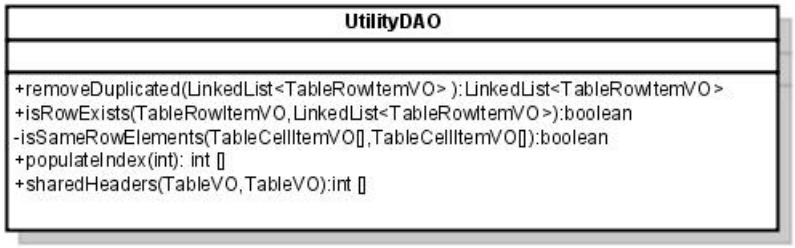


Figure 6.1: Class Diagram for the class UtilityDAO

Operator Name	Operator Type	Java Class
Project	unary	ProjectRAOptDAO
Select	unary	SelectRAOptDAO
Product	binary	ProductRAOptDAO
Union	binary	UnionRAOptDAO
Intersection	binary	IntersectionRAOptDAO
Difference	binary	DifferenceRAOptDAO
Natural Join	binary	NaturalJoinRAOptDAO
Semi Join	binary	SemiJoinRAOptDAO
Anti Join	binary	AntiJoinRAOptDAO
Natural Join	binary	NaturalJoinRAOptDAO
Left Outer Join	binary	LeftOuterJoinRAOptDAO
Right Outer Join	binary	RightOuterJoinRAOptDAO

Table 6.1: Operators in RALT together with their respective Java classes
title of Table

In Table 6.1 we give a list of all the operators that have been implemented in RALT together with the name of their corresponding Java classes in our system. We now explain how we implemented these operators in detail.

Project

- **Class** - ProjectRAOptDAO
- **Input** TableVO table, int [ ] index, boolean removeDuplicates
- **Output** ResultVO
- **Description** – The first parameter to this method refers to the input table on which the project operator must be performed. The int array called the *index* refers to the column indices of the input table which are to be selected. For example if the input table has four attributes and the *index* array has values [1, 3], our Project operator selects only the second and the fourth[2] columns for each row data of the input table. The user is responsible for selecting the columns by interacting with the view layer and the column indices selected are passed by the Controller to our operator. The third parameter informs the operator whether to remove duplicate rows before returning the result to the Business Controller. When the Project operation of Relational Algebra is called, this value is always set to true since we follow the set algebra and hence all duplicates must be removed. The reason for having this parameter is because when implementing some operators, we perform project like operation but we wish to keep the duplicates as well. Hence this one parameter helps in preventing duplication of code.

  Our Project operator traverses through each row of the input table, selecting cell values from each row whose column index match the values present in the *index* parameter.

Select

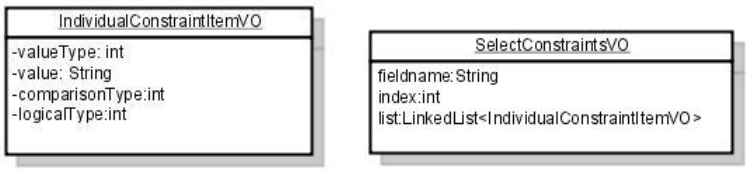
- **Class** - SelectRAOptDAO
- **Input** TableVO table, LinkedList ⟨ SelectConstraintsVO ⟩ constraints
- **Output** ResultVO
- **Description** – The implementation of this operator was a complex task due to the wide variety of items (constraints) that had to be considered during its implementation. Here is a typical Relational Algebra query on Selcet operator.

$$\sigma_{rate>=5.25 \wedge type=`current'} account$$

As we can see that the Select operator is made up of both logical (e.g. $\wedge$) operators as well as arithmetic (e.g. $=, \geq$) operators. Moreover, each attribute may have more than one constraint (e.g. $rate > 0 \wedge rate < 5.5$). This made things more complicated. The second parameter sent when invoking the Select method is a list of constraints. We decided to group each of the constraints according to the attributes. For example each element of the *constraints* parameter represents the constraints assigned to a particular attribute of the input table to the operator. Each individual constraint for a particular attribute can be captured by the class IndividualConstrainItemVO. The *comparisonType* field can take six different integer values each a unique Comparison operator (e.g. $=, >, \geq, <, \leq$). The *logicalType* represents whether constraint is AND or an OR type.

These individual constraints for an attribute are then wrapped under a single object of type SelectConstraintVO. This instance also keeps a track of the column index in the first parameter for

[2]Since in computer programs the indexing of arrays start from 0 and not 1



(a) IndividualConstrainItemVO (b) SelectConstraintVO

Figure 6.2: Classes used for capturing constraints for the Select operator

the Select method which it represents.When carrying out the Select operator, we first traverse through all the rows of the table and then for each row we traverse through the constraints list. We verify if each row satisfies all the constraints and if yes then we add it to the result table.Before returning the resultant table, we remove all the duplicate rows by calling the *removeDuplicated* method of the UtilityDAO.

Product

- **Class** - ProductRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – It produces all the possible combination of each row of the first table with rows in the second table. It does not remove duplicates.

Union

- **Class** - UnionRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – The operator combines the rows present in both the input tables and then removes the duplicated rows.

Intersection

- **Class** - IntersectionRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – It compares all the rows in the first table with the rows in the second table and returns on the common rows between the two tables. Any duplicated rows from the final table are removed.

Difference

- **Class** - DifferenceRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – It returns only the rows of the first table which do not appear in the second table.

An interesting thing to mention here is that for Union, Intersection and Difference to be successfully computed, we must first find out whether the two input tables are compatible or not. We have prepared classes which does this validation which are present under the *com.imperial.validator* package. We determine if two tables are compatible or not just by comparing the information present in the header of the two tables.

Natural Join

- **Class** - NaturalJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – Returns all the combination of rows in table1 with rows in table2 which are equal on their common attribute names. Unlike in the Product operator, in case of Natural Join the common attribute between the two tables appear only once in the result table. We first find the attribute(s) which are common between the two tables and then compare which rows of the two tables are equal on these common attributes. If they are, we add them to the result table.

Natural Join

- **Class** - NaturalJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – Returns all the combination of rows in table1 with rows in table2 which are equal on their common attribute names. Unlike in the Product operator, in case of Natural Join the common attribute between the two tables appear only once in the result table. We first find the attribute(s) which are common between the two tables and then compare which rows of the two tables are equal on these common attributes. If they are, we add them to the result table.

Semi Join

- **Class** - SemiJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO

- **Description** – Computed similarly to the Natural Join operator, but instead of displaying rows from the table2, we display only the rows of table1 which are equal to rows in table2 on common attributes.

Anti Join

- **Class** - AntiJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – The definition of Anti Join says that it is those rows of table1 which have not featured in the Natural Join with table2. To implement this operator we could have adhered to the standard way of going through all the rows of the table1 and table2 and select only the rows which are not equal on the shared attributes. However we opted for a way which would allow us to utilise the already implemented method in our application. We first performed a Semi between table1 and table2. From this operation we get only those rows of table1 which have appeared in a Natural Join with table2. Since we are interested in finding those rows of table1 which which do not appear in the Natural Join with table2, we simply use the Difference operator and subtract the result by the Semi Join operator from table1.

```
SemiJoinRAOptDAO semi = new SemiJoinRAOptDAO ();
ResultVO tempResult = semi.action(table1, table2);
                        // find the Semi Join between table1 and table2

DifferenceRAOptDAO diff = new DifferenceRAOptDAO ();
ResultVO finalResult = diff.action(table1, tempResult.getTable());
                        // subtract the result produced by the semi join from table1 to get the
                        // Anti Join rows
```

Left Outer Join

- **Class** - LeftOuterJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – According to the definition of Left Outer Join, the output table of this operation consists of the output of a Natural Join between table1 and table2 together with those of table1 which have not featured in the Natural Join. The rows which do not feature in the Natural Join in the output table have null values under the columns of table2 which are not common with table1. We implement this operator by first finding a Semi Join between table1 and table2 and perform a Difference between table1 and the result produced by the Semi Join. We add these rows to the rows produced by the Natural Join between table1 and table2, setting the cell values under columns of table2 (which are not shared with table1) to NULL values.

Right Outer Join

- **Class** - RightOuterJoinRAOptDAO
- **Input** TableVO table1, TableVO table2
- **Output** ResultVO
- **Description** – As we know, Right Outer Join follows the similar concept as Left Outer Join, but instead of displaying the rows of table1 which have not featured in the Natural Join between table1 and table2; it displays the rows of table2. Like in Left Outer Join, this operator can also be implemented by using the already implemented operators in our system.

  $$LeftOuterJoin = ((table1 \bowtie table2)) \cup ((table2 - \pi_{table2_1, table2_2 \ldots table2_n}(table1 \bowtie table2)) \chi (NULL \ldots NULL))$$

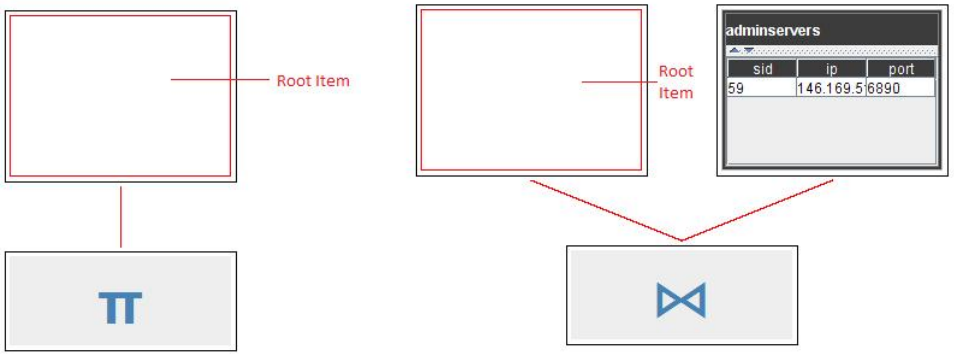
  From the above equation we can say that the Right Outer Join can be computed by a combining the operators Natural Join, Project and Product. In our implementation we first compute a Natural Join between the two input tables (table1 and table2) and then Project all the columns of table2 in the order corresponding to that of table2. We then subtract the rows produced by the Project operator from table2 to get the rows in table2 which have not performed in the Natural Join. These rows are extended so that they match the column header of the rows produced by the Natural Join between table1 and table2, placing NULL values in the columns which only belong to table1. Hence we get our result.

This section gives us an idea how we implemented each of the Relational Algebra in our system. Having this knowlege we can move onto the next Section which explains how queries are built.

6.4 Graphical Query Building

As mentioned earlier, graphical representation of the query building process is an important feature of our system. What makes this feature exclusive is the fact queries can be built entirely by graphical interaction without the need for writing anything down. As shown in Figure 4.2, we use the Query Visualisation Panel (marked **A**) for displaying the query building process. The Query Visualisation Panel accepts inputs such as table and operator name from the Table Name Display Panel and Operator Panel, marked as **B** and **C** respectively in Figure 4.2. Drag and drop feature is supported by all the three Panels. When the user selects a table name or an operator name (which are displayed as JList items in the Panels), the Transfer Handler class responsible for implementing the drag and drop feature, wraps it into an object which is transferable and sends it off to the Query Visualisation Panel. The Query Visualisation Panel is designed to accept the transferable type of object and can can immediately decode its contents (e.g. operator name or table name which is of type String). Moreover it can also determine the point where the drop was made. Once these two information is obtained, the Query Visualisation Panel calls the *addToQueryTree()* method of the Visualisation Controller, passing a String and a Point object, representing the item dropped and the location respectively.As we have mentioned earlier, the Visualisation Controller plays an important part in the query building process. From the information passed in the *addToQueryTree()* method, the Visualisation Controller can build a query.

The Visualisation controller keeps a track of all queries being built with the help of two LinkedLists -*querySequence*, *expectingSequence*. There is a reason for keeping queries in two separated lists. Our system allows users to just drop an operator on the Query Visualisation Panel without specifying any input table to the operator. Since the Root Item of every query displays an input table and this is not the case when only an operator is dropped, we keep this empty query structure (empty since the Root Item does display table contents)in the *expectingSequence* as we expect the user to soon add input tables to this empty query.



(a) Empty Query Structure for Unary Operator

(b) Empty Query Structure for Binary Operator

Figure 6.3: Queries stored in the *expectingSequence* with Root Item empty

This way we save searching time, as we are not required to navigate through all queries in the *querySequence* when the user provides a table as Root Item for an incomplete operator. Once a table is provided for the rootItem, we remove it from the *expectingSequence* and add it to the *querySequence*.

In Figure 6.4 we present a flow chart of the *addToQueryTree()* method. On receiving the parameters *name* (String) and a *location*(Point), the Visualisation Controller verifies whether the *name* matches any of the implemented operators present in the system or equivalent to one of the tables names fetched from the database. Once it is known what the *name* represents, we use the *location* attribute to determine where the drop occurred. If the user dropped a table name just on the panel, then we display the contents of the table in a QueryTreeTablePanelGUI object and add it to the *querysequence*. If the user just drops an operator on the Query Visualisation Panel itself, we create an empty tree structure which contains a QueryTreeOperatorPanelGUI Object displaying symbol of the operator being dropped together with empty JPanel(s), where the user can drop tables for input to the operator. The empty query structure can contain one or two empty JPanels depending whether the operator dropped is unary or binary. Alternatively users can drop operators on a table name. If a binary operator is dropped, the Visualisation Controller adds an empty JPanel where another table can be dropped to complete this query.

Users can drop tables on these empty JPanels. The JPanels are able to accept both a String name belonging to a table fetched from the database or of type TableVO. In our system users can provide inputs to operators either by dragging table names (fetched from the database)onto the JPanels in which case the Visualisation Controller fetches the contents of the respective table from the Business Controller and displays the information on the Query Visualisation Panel. Otherwise, the user can

provide as input to operators results produced by operators which may be present in a different query tree. Like we use the drag and drop feature when dropping table name we use a similar approach. However, we cannot wrap up the name of the table of the intermediate result and send it to the Visualisation query like we did in the first case since the Business Controller does not keep a record of the intermediate results produced. Fortunately Java drag and drop allows us to transfer customised data structure. Each node in a query tree representing a table has an attribute of type TableVO, containing all the header and row information related to the table. When making such a table as an input to an operator by dragging the table contents onto the empty JPanel, our customised data transfer operation, sends a copy of the TableVO object whose contents the JPanel displays on receiving.

When users drop tables on these empty JPanels, which if satisfies the input condition for an operator, instigates the Visualisation Controller to give the command to execution of the operator with these valid inputs and returning the result for the operation. An interesting thing to note here is that some operators (Project and Select) require additional information in which case dialog box are displayed to capture input. When the user input the necessary data, the operator is executed and result displayed. This result is then displayed in a QueryTreeTablePanelGUI and set as a child for the query node displaying the operator symbol. The Visualisation Controller is equipped with auxiliary functions which when given correct inputs create different types of JPanels for displaying various types of information -e.g. contents of the result produced by an operator, displaying an operator symbol or the constraints related to an operator. The Visualisation Controller is responsible for positioning these newly created panels so that they can show the parent-child relationship correctly in a query building tree. As we had mentioned earlier in Section 5.2, each item node present in a query tree belongs to the type Query Node. A Query Node represents JPanels (QueryTreeTablePanelGUI, QueryTreeOperatorPanelGUI, QueryTreeConditionPanelGUI) to display different types of information. Each Query Node contains the x & y coordinate for each JPanel and also the height and width of the panel. These values are set by the Visualisation Controller when the nodes are created. However, when the user manually moves the Query Node on the Query Visualisation Panel or changes its size, these values are updated so that the changes are reflected in the Query Visualisation Panel. To show the parent-child relationship every Query Node keeps a record of all the other Query Nodes it is connected to.

At the end of the execution of the *addToQueryTree()* method, the Visualisation Controller requests the Query Visualisation Panel to reflect the changes made to the query trees. The latter iterates over the items(Query Node) present in the *expectingSequence* and *querySequence* and displays them on the panel positioning them according to their x and y coordinate values set by the Visualisation. Each Query Node contains a list of nodes it is connected to. The Query Visualisation Panel when iterating over the two sets of LinkedLists representing all the complete and incomplete queries present in the system, also draws any line between two nodes to represent the parent-child relationship.

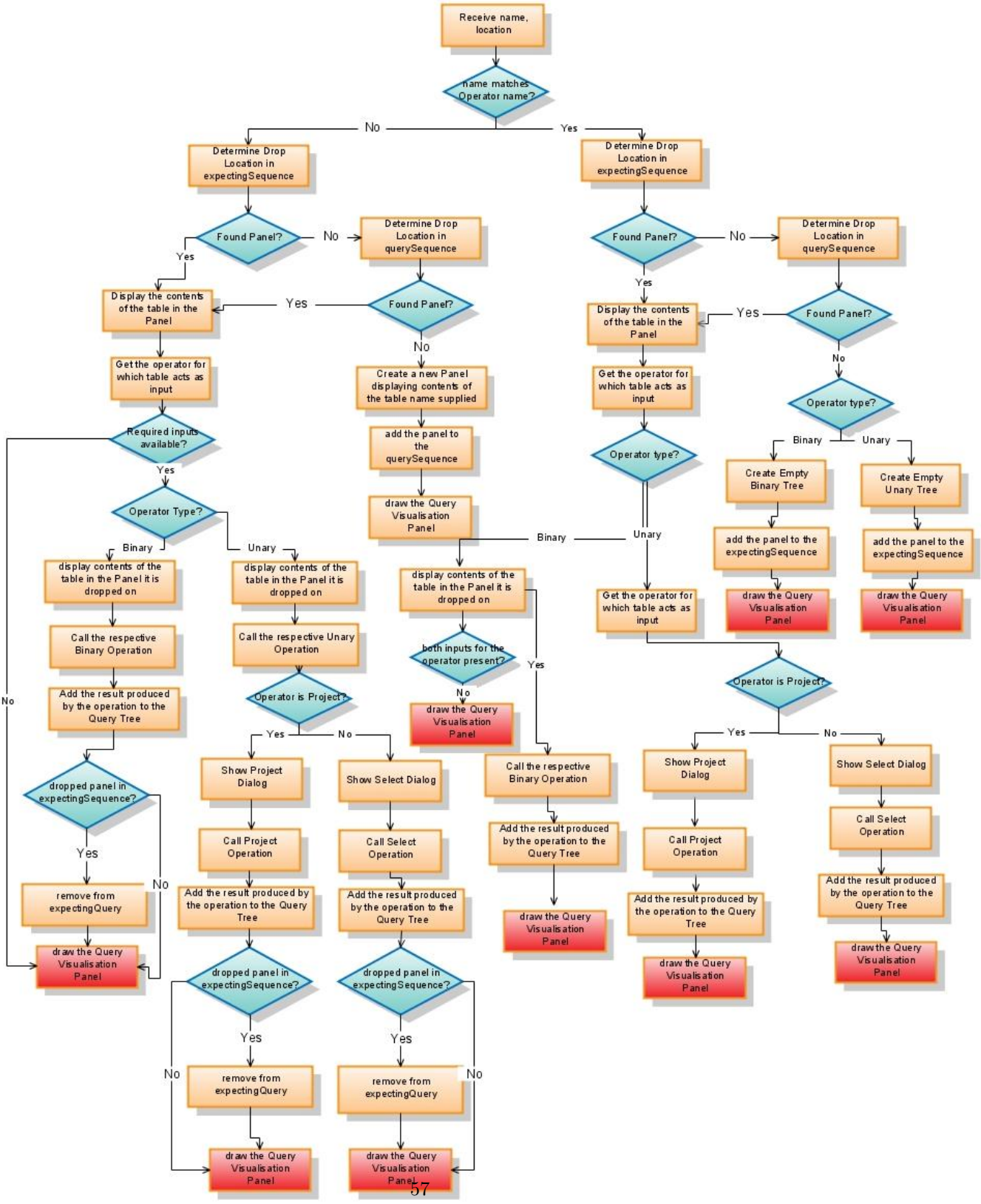


Figure 6.4: Flow Chart explaining the method *addToQueryTree()*

6.5 Performing Data Lineage

In this section we will explain the how data lineage is performed in our system. We understand the process of tracking data lineage through an example. Let us assume we have built the query as shown in Figure 6.5

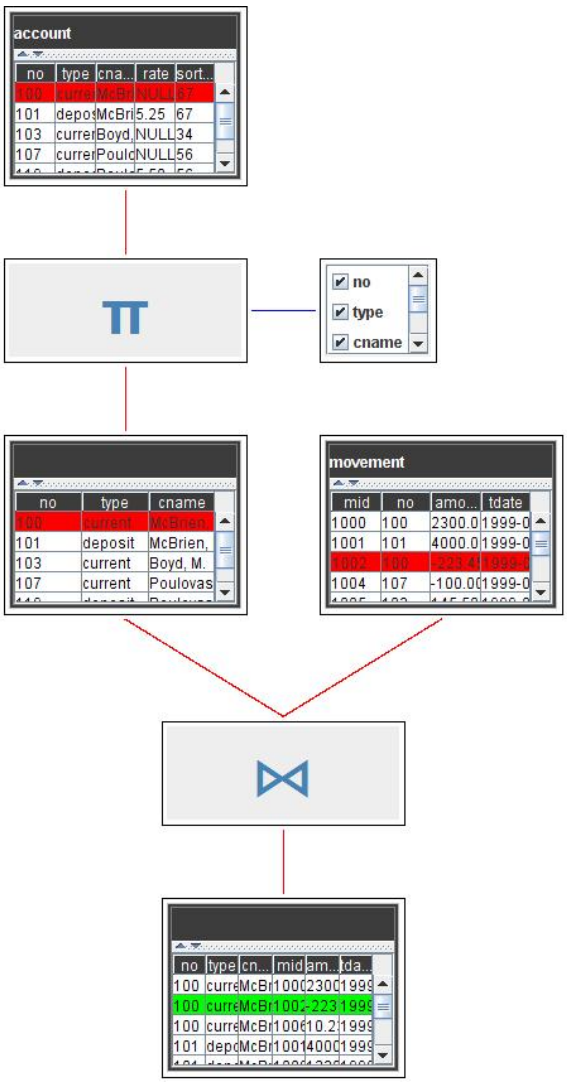


Figure 6.5: Example query for illustrating Data Lineage in RALT

The query is built by first carrying out the Project operator by projecting the *account* table with the columns *no*,*type*,*cname*. We then execute a Natural Join between the result produced by this Project operator and the *movement table*. We are interested in finding the lineage for the second row of the result produced by the Natural Join (the row marked in green). It is important here to mention that we have implemented functionality for tracking the lineage for each of the Relational Algebra operators. We have implemented these functionalities in separate classes, which are located under the

package *com.imperial.dao.lineage*. For example the class implementing the lineage functionality for the Difference operator is DifferenceLineageDAO while that for Product operator is ProductLineageDAO. The functions for performing data lineage take as input the following parameters:

- *target* - it is of type TableVO. It is made up of the rows of a table whose lineage we are trying to find.
- *first input table* - also of type TableVO and represents the first input for the operator which produced the row(s) whose lineage we are interested in.
- *second input table* - as the name suggests it refers to the second input table and is only applicable for the binary operators. It is of type TableVO also.

We perform data lineage by moving up the query tree, calling the appropriate data lineage functionality when we encounter an operator node in the tree. The parent-child relationship shared by the nodes in the query tree helps us in achieving our goal. Once we determine the rows in the parent(s) of the operator node which act as the parent of the rows of the target table whose lineage we are trying to find are determined, we perform a lineage on these rows of the parent tables.

In our example once the user selects a row in the result produced by the Natural Join operator and requests the system to find how that particular row was derived, we do the following steps:

1. We call the method *showLineage* in the VisualisationController class, passing as parameter the co-ordinates of the point on which the user has clicked in relation to the Query Visualisation Panel. We use the co-ordinates passed as parameters to verify which Query Node in our list of queries contains that point. Once the node is found, we can easily find which row in the JTable (target table) contained in that node has being clicked thanks to Java Swings in built method. We call the method *recursiveLineageDisplay* passing the Query Node and the rows selected in the JTable (represented as an array of integers).

2. In the *recursiveLineageDisplay* method, we get the parent of the Query Node. If no parent is found, we know that the table has not been derived from any table that have appeared before in the query and hence stop our propagation. If we find a node we detect whether the JPanel inside the node represents an operator or not (i.e. whether it is an operator node of type QueryTreeOperatorPanelGUI). If it is, we find the name of the operator represented in that node together with its type (i.e. whether unary or binary). In our example, we have found this node to contain a QueryTreeOperatorPanelGUI JPanel representing the Natural Join operator.

3. If the operator node is of type unary we get the parent of this node which is the input table to the operator. We call the respective lineage operation for that operator with the help of our Business Controller. For example for finding the lineage of the Select operator we call the functionality of the class SelectLineageDAO from the Business Controller.

   If the operator is of type binary, we get the two input tables to the operator. In our case we refer to the two input tables for the Natural Join operator. Once we get the input tables, we call the class designed for implementing the tracing lineage for this binary operator (in our case it is the NaturalJoinLineageDAO). We can determine the input tables easily as the three nodes one representing the operator node, while the other two represent the input table nodes, are connected directly/indirectly. For example the first input table node acts as a parent to the operator node as well as the second input table. Hence once we access the parent of the operator node (which is the first input table) we can also access the second input table since it

is the child of the first input table node. Having the required data we call the lineage operation for that operator.

4. The lineage operations returns two arrays of int - the first representing the indices of rows in the first input table which have participated in generating the row(s) whose lineage we are interested in. The second array contains the same information but for the second input table. If the operator is unary, we are just interested in the first array. We instruct the Query Nodes containing these input tables to display the rows whose indexes are contained in the array in Red colour when displayed in the Query Visualisation Panel.

5. We go back to step 2 passing the Query Node(s) containing the input table(s) together with the array that contains the indices of the row that has participated as source from which rows in our target table was derived. This recursive approach allows us trace back how a particular row is derived from all the tables present in that tree that have appeared before our target table. In our example we call the *recursiveLineageDisplay* method twice - first one containing the Query Noe containing the first input table to our Natural Join operator together with the first array passed by the NaturalJoinLineageDAO class. The second *recursiveLineageDisplay* is called which contains the Query Node containing the second input table to the operator and the second array returned by the lienage function for the Natural Join operator.

As we go to back to step 2, we notice that the node containing the second input table has no parent hence the propogation stops. The first input table however has a parent - a Query Node representing the Project Operator. Hece we carry on the data lineage process for this node until a node is reached which has no parent (the node displaying the table *account*).

This way we achive data lineage in our system.

6.6 Adding New Operators

When designing our system, we have kept in mind the possibility that the code maintainers may wish to add new operators to the system. We allow users to add new operators to our system by just making few modifications to the existing system, without disturbing the existing implemented operators. Let us see how a new operator can be added by the Programmer (person in charge of maintaining the code) through an example. We assume that our Programmer wants to add a Left Outer Join Operator into the system, which is not already present in the system. The Programmer needs to carry out the following steps to add a new operator into the system.

- The Programmer creates a Java class in the package *com.imperial.dao.opt* . This is the package where we create and keep individual classes for all Relational Algebra operators. We call this class LeftOuterJoinRAOptDAO. Since we know that Left Outer Join is a binary operator our class extends the abstract class BinaryOperatorRADAO and implements the method *action* of that class. All binary operators in our system extend this class and the *action* method takes as parameter two TableVO (object we used to represent database table) type objects.

- In Figure 6.6, we can see how our Programmer has implemented the Left Outer Join operator. We can see that when implementing this method the Programmer has taken advantage of the already existing operators in the system (for example Semi Join, Natural Join) etc. Also we can see that Programmer has decided to implement a method named *addExtraRows* specific to this class. Sometimes we may wish to verify whether the inputs to an operator are valid or not. In

```
package com.imperial.dao.opt;

import java.util.LinkedList;

public class LeftOuterJoinRAOptDAO extends BinaryOperatorRADAO  {

    public LeftOuterJoinRAOptDAO() {
        name = "Left Outer Join";
        binary = true;

    }

    public ResultVO action(TableVO table1, TableVO table2) {

        NaturalJoinRAOptDAO natural = new NaturalJoinRAOptDAO();
        ResultVO naturalResult = natural.action(table1, table2);

        SemiJoinRAOptDAO semi = new SemiJoinRAOptDAO();
        ResultVO semiResult = semi.action(table1, table2);

        DifferenceRAOptDAO diff = new DifferenceRAOptDAO();
        ResultVO diffResult = diff.action(table1, semiResult.getTable());

        LinkedList<TableRowItemVO> extraRows = addExtraRows(diffResult
                .getTable().getRows(),
                naturalResult.getTable().getHeader().length);

        LinkedList<TableRowItemVO> newRows = naturalResult.getTable()
                .getRows();

        for (int i = 0; i < extraRows.size(); i++) {
            newRows.add(extraRows.get(i));
        }

        return naturalResult;
    }

    private LinkedList<TableRowItemVO> addExtraRows(

}
```

Figure 6.6: LeftOuterJoinRAOptDAO.java

that case the user needs to implement a class for validation which extends the GeneralValidator located under the package *com.imperial.validator*. This package is used for containing all the classes used for validating functions. The validation class returns an error message if something is wrong. If the inputs are accepted by the validator class, we carry on implementing the functionality of the operator. However for simplicity purpose, our Programmer has decided not to validate the inputs. The Programmer must remember that the *action* method returns an object of type ResultVO, which is made up of object TableVO and a String. The TableVO object represents the result produced by the operator while the String signifies any error message generated due to incorrect execution of the operator. We use this String to alert the user in case something has gone wrong (for example the inputs are of wrong type for the operator). The message is displayed to the user in the Error Console Panel.

- Now that the Programmer has implemented the functionality of the Left Outer Join operator, it should be integrated with the rest of the system. First the Controller should be able to call the *action* method of LeftOuterJoinRAOptDAO class. All binary operators have the same inputs two objects of type TableVO. The method *callBinaryOperation* in the Business Controller method is responsible for calling binary operations (for example Natural Join, Semi Join etc.) while the callUnaryOperation calls the operators Project and Select. The callBinaryOperation accepts as parameters the following:

  - *operationName*: a String object representing the operator name.
  - *table1*: the first input table. It is of type TableVO.
  - *table2*: another TableVO object representing the second input table.

  The Programmer adds the code in this method which allows the calling of the *action* method of LeftOuterJoinRAOptDAO class when the *operationName* is equal to Left Outer Join. The Programmer adds the following line of code as one of the else statements in the method:

```
else if (operationName.equals("Left Outer Join")) {
            LeftOuterJoinRAOptDAO semi= new LeftOuterJoinRAOptDAO ();
            newTable = semi.action(table1, table2);
            }
```

- The Programmer must also provide information to the Visualisation Controller which will allow the latter to determine that Left Outer Join is the name for an operator and it is of type binary. Visualisation Controller must be also informed of the symbol to be displayed in the Query Visualisation Panel to demonstrate that a Left Outer Join operation has occurred. This can be done by simply adding one line of code. In the VisualisationController we have a HashMap named *operatorKeys* which stores information about operators. In constructor of the class VisualisationController, our Programmer adds the following code,

```
operatorKeys.put("Left Outer Join", new OperatorItemVO("binary","="+"\u00D7"));
```

  Since we use symbols the for representing Relational Algebra operators in Query Visualisation Panel, the icon for Left Outer Join can be easily fetched from the HashMap. Also when the user drops the Left Outer Join operator on the Query Visualisation Panel, the Visualisation Controller can immediately determine whether the operator is binary or unary and act accordingly.

- When a new operator is added to the system, it must be displayed in the panel displaying all the operators in the system. The OperatorPanelGUI class located under the *com.imperial.gui.panel* contains an attribute of type a two dimensional array of String, called *operators*. In this component of the GUI we display the operator names together with a symbol for the operator. We have opted for displaying the symbol for the operator using Unicode. Each item of the *operators* attribute is representing as *operator name*,*operator Unicode value*. The Programmer should add the following element to the two dimensional attribute operators:

  ```
  {"Left Outer Join","="+"\u00D7"}
  ```

  Now the Left Outer Join is displayed in the OperatorPanelGUI.

We can see from this example how easily one can add new operators to our system. This makes our system easily extendible.

Chapter 7

User Guide

This chapter presents a brief, overview of RALT from the user's perspective. The basic functionality of each aspect is illustrated by screenshots.

7.1 Introduction

RALT is started by double clicking on the Java application icon labeled *RALT* present in the CD-ROM provided for this application. The application can be installed anywhere on the system according to the system owners desire [1]. When the application is started the Login Screen as shown in Figure 7.1 appears.

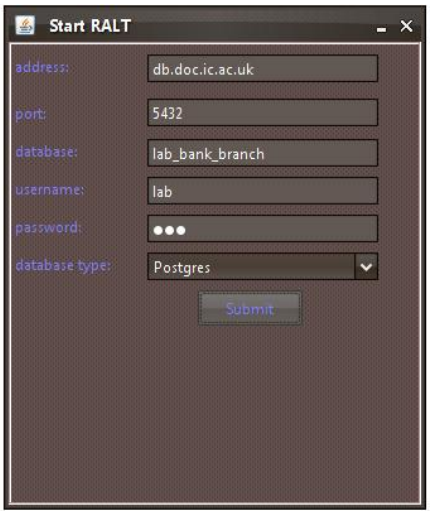


Figure 7.1: Login Screen for RALT

The user is required to provide the information needed to connect to a database. The following information are required:

[1] The user must have Java 1.6 installed in the system in order to start up this application.

- *address* – The url of where the database server is located. For example *db.doc.ic.ac.uk*. If the database the user wants to connect to is located in the same machine where RALT is currently running, the user is required to provide *localhost* .
- *port* – The user may provide the port it wishes to connect to otherwise the system takes number to be 5432, the standard port used for Postgres databases.
- *database* - This is a mandatory field and the user is required to provide the name if the database he/she wants to connect to at the database server. A possible name of a database can be lab_bank_branch.
- *username* - The username for logging into the server should be provided here.
- *password* - This is the password for required for connecting to the database.
- *database type* – The user must choose the type of Database. RALT allows users to connect to two kinds of database - Microsoft SQL Sever 2005 DBMS and Postgres 7.4 DBMS.

7.2 On Loading

Once the details have been entered, the user can submit the information by clicking on the *Login* button. The system tries to connect to the database. If it fails to connect to the database, an alert message in a popup box is triggered, asking the user to verify the details provided. Once the user confirms the details, it must re-submitted.

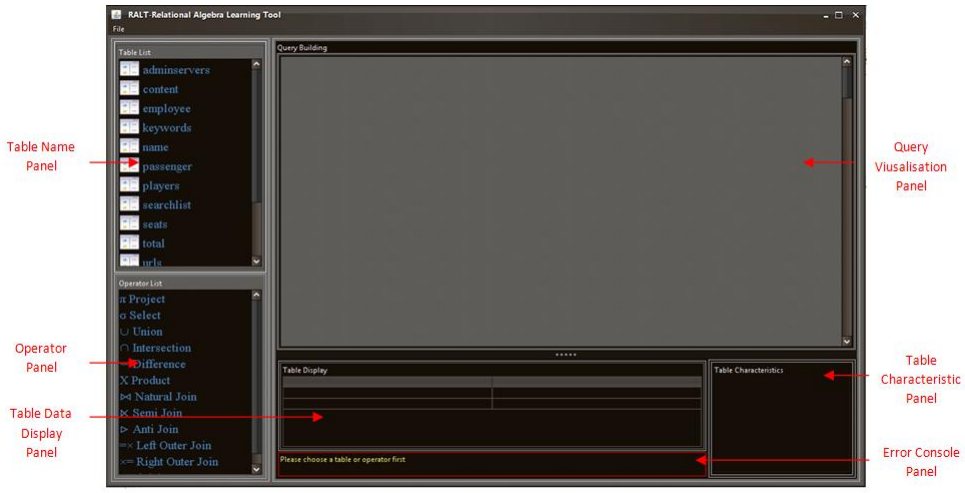


Figure 7.2: Screenshot: Application Screen of RALT

If the system manages to connect to the system, it displays to the user the main screen of the application (Figure 7.2). On loading the main application screen, the system displays the names of the tables fetched from the database in a list at the top left panel of the screen. This panel is marked as *Table Name Panel*. The bottom left panel displays all the operators in the system in a list and is called the *Operator Panel*. Each operator is assigned a symbol which is equivalent to the symbol used when building Relational Algebra queries using algebraic notations.

7.3 Drag and Drop

User can drag components from one part of the screen to another. This way user can also build queries by dragging and dropping operator names from the panels displaying table and operator names. Users can also use this feature to view the contents of a table. When the user hovers the mouse over a table or operator name the mouse cursor changes from *arrow* to a *hand*. This indicates that the item is eligible for drag and drop. While dragging an item, when the mouse icon becomes  , it indicates that no drop is possible on that component.Also at the bottom of the application screen, the area bordered by a red line, is used for displaying messages to the user.

We explain the effect of dragging and dropping components in the following section.

We have discussed the different the functionality of each component of the Application Screen in Chapter 4.

7.4 Viewing Table Contents

To view the contents of a table the user needs to drag a table name and drop it onto the panel labelled *Table Data Display Panel* (Figure 7.2). Dropping the table name there automatically displays the contents of the table. For example dropping the table *account* on *Table Data Display Panel*, displays the data of that table in that panel as shown in Figure 7.3. Users must realise no action is taken if they try to drop an operator on this panel. The tables displayed are scrollable allowing users to view all the rows of the table in case they exceed the space allocated to them.

As soon as a table name is dropped, the table properties attribute types, Primary key for the table etc. are displayed on the panel labelled as *Table Characteristics*.

7.5 Building Queries

We will explain the query building process through an example. We want a project operation on the table *account*. We first select the table name *account* from the panel displaying all the table names and drop it onto the *Query Visualisation Panel*. Figure 7.3 shows the effect of this action. The table *account* is displayed in the latter panel as a rectangular box. The user can scroll through the rows of the table.

We now select the Project operator from the list displaying all the operator names, drag it and drop it on the rectangular box displaying the table *account*. Since Project operator requires some additional information, it displays a dialog. The dialog displays the column names of the *account*

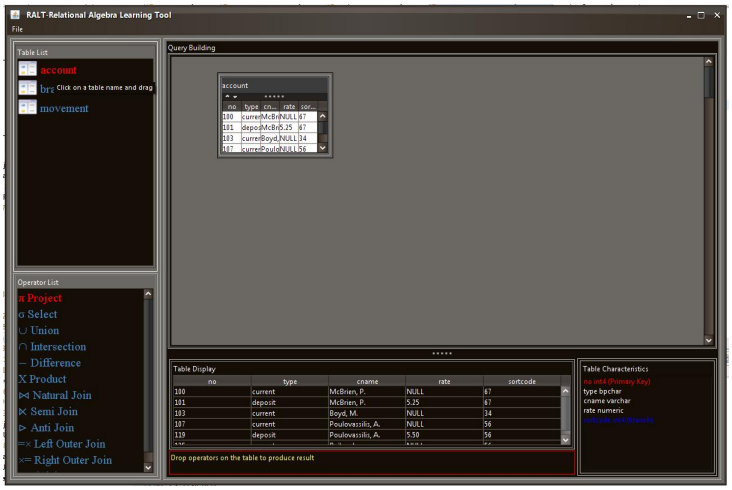


Figure 7.3: Screenshot: Effect of dropping the table *account* onto the *Table Data Display Panel* panel and *Query Visualisation Panel*

table. A checkbox is assigned to each column name which the user can select to instruct the Project operator which attributes to select.

We select the first and last field by checking the checkboxes. We then click the **Ok** button. The Project Operator is executed and the whole query is displayed in a tree-like structure as displayed in Figure 7.4.

As we can see in Figure 7.4, the *account* table we had seen earlier is now connected to a rectangular box displaying the Project symbol (Π). This box is connected to a rectangular box displaying some data in a table form. This is the result generated by the Project Operation. Moreover, the box displaying the Project symbol is also connected to another box by a blue line. The latter displays the constraints for the project operator, i.e. the columns selected. We use red lines for connections between boxes displaying operator symbols and those displaying table data. On the other hand, blue lines are used for connecting any operator with any constraints it may be related to, as clearly explained in the above figure.

Alternatively, we can build a query by first dropping an operator and then providing it with inputs. The operator will not execute until it has the required inputs. As shows in Figure 7.5, we have dropped the Natural Join operator on to the Query Building panel, an incomplete tree structure is created by the system. Since Natural Join is a binary operator, we see two empty red rectangular boxes are connected to the box displaying the operator symbol. We can provide the input operator by dropping table names on these boxes.

When we first provide one input to the Natural Join, the inputs for the operator are incomplete. Hence it is not executed as shown by Figure 7.6. However, on receiving a valid second input as shown in Figure 7.7, the Natural Join is executed and the result displayed.

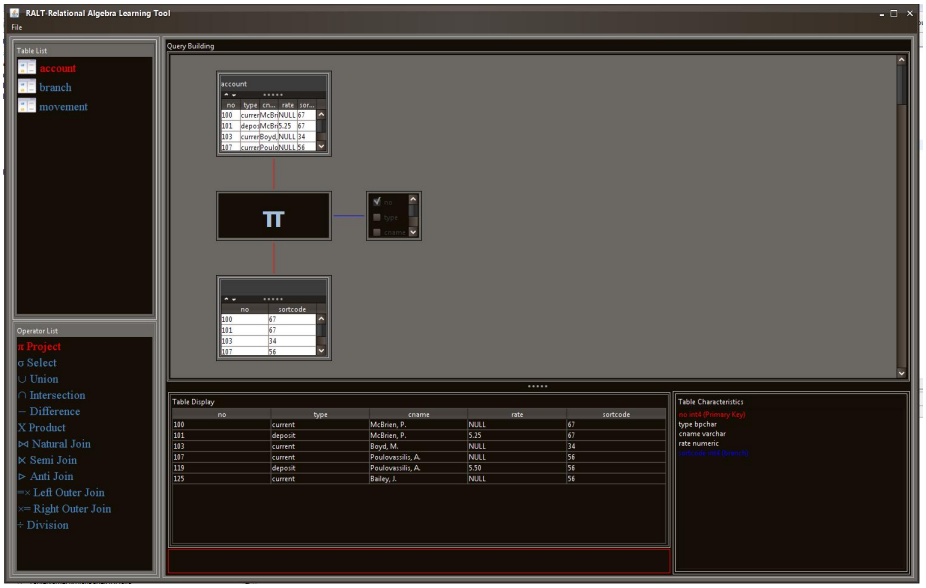


Figure 7.4: Screenshot: A tree-like representation for a Project query

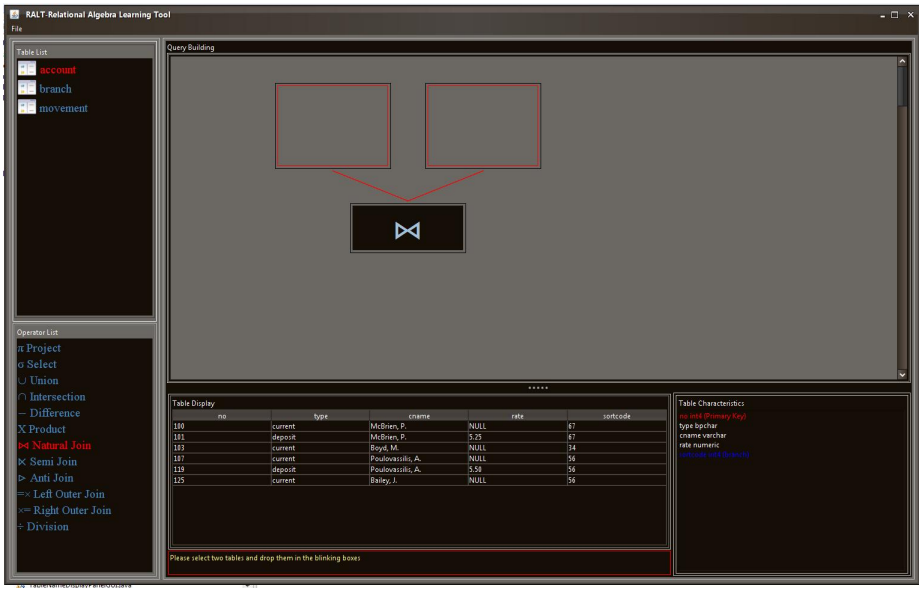


Figure 7.5: Screenshot: Dropping the Natural Join operator without any input table

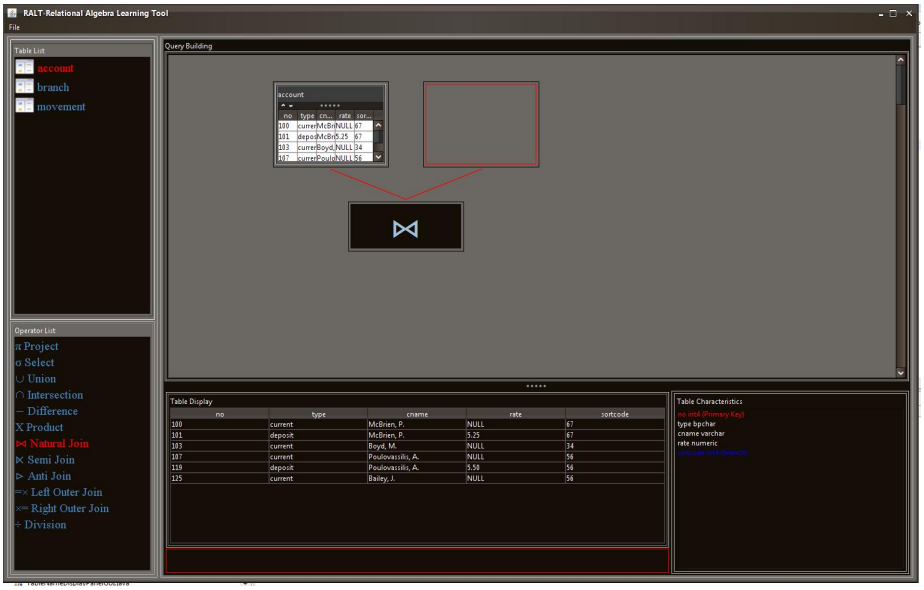


Figure 7.6: Screenshot: Adding one input table to the Natural Join Operator

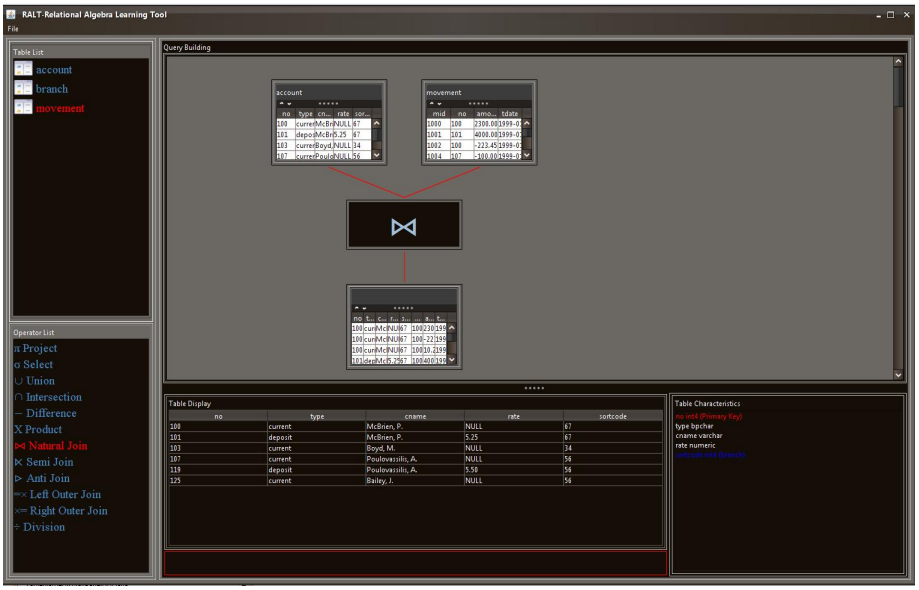


Figure 7.7: Screenshot: Executing of the Natural Join

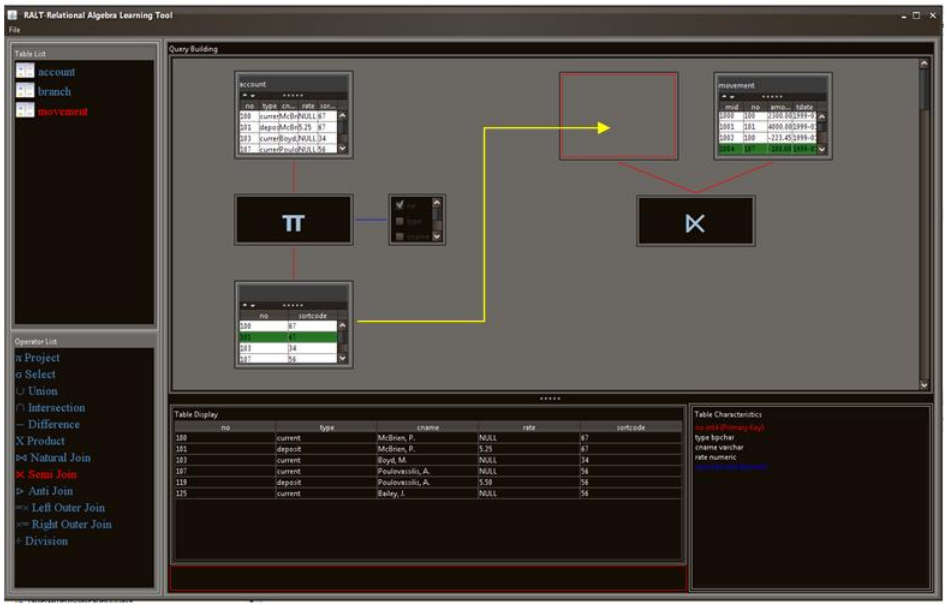


Figure 7.8: Screenshot: Providing output of an operation as input for another

Our system also allows us to provide as input to operators, results generated by another operation. Let us take the example given in Figure 7.8. As we can see, we have two separate queries. The first query is a Project operation on the *accounts* table which selects the columns *no* and *sortcode*. The second query is an incomplete Semi Join query whose second input is only available. Our application allows us to provide the result generated by the Project operation as the first input to the Semi Join operator. To do this we left-click on any rows of the table produced by the Project operator (which highlights the row) and then drag that row and drop it on to the red box. The yellow arrow in Figure 7.8 demonstrates the process of dragging the row of the result table and providing it as the first input of the Semi Join oprator.On receiving this input, the requirements for the Semi Join operator is complete and it gets executed as shown in Figure 7.9.

Our system can also alert the user when correct inputs are not provided for an operator. Let us taken the Union operator for example which must have as inputs two tables which are compatible i.e. both table have the same number of columns and the type of the column at a particular index in the first table must be equal to the type of the column at the same index in the second table. In Figure 7.10, we have provided two non-compatible tables *account* and *movement* as input to the Union operator. The system will not execute this operator and will show the user an error message, as pointed by the green arrow in Figure 7.10. The message given to the user in this case is "The two tables are not Union compatible".

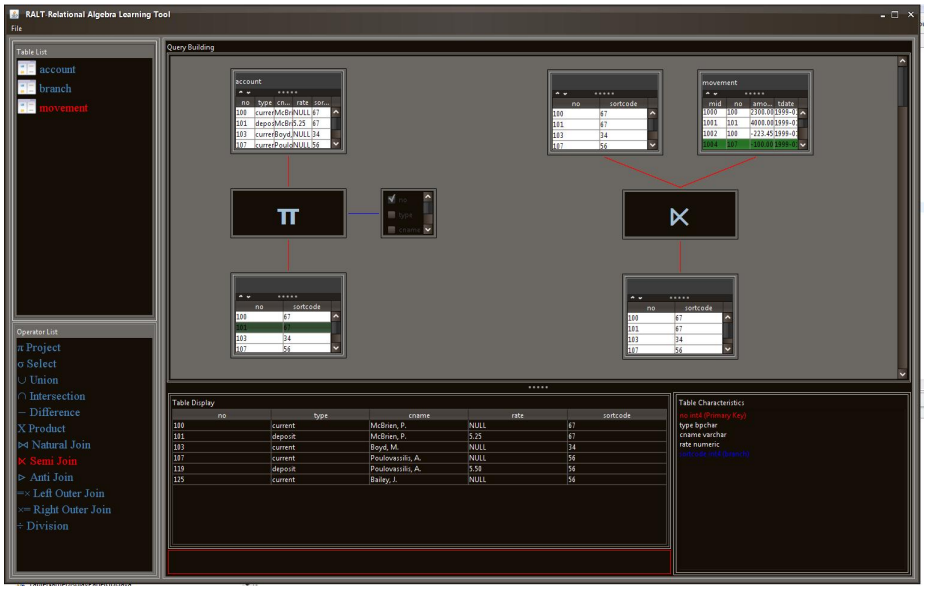


Figure 7.9: Screenshot: Execution of Semi Join after receiving input as output from the Project operator

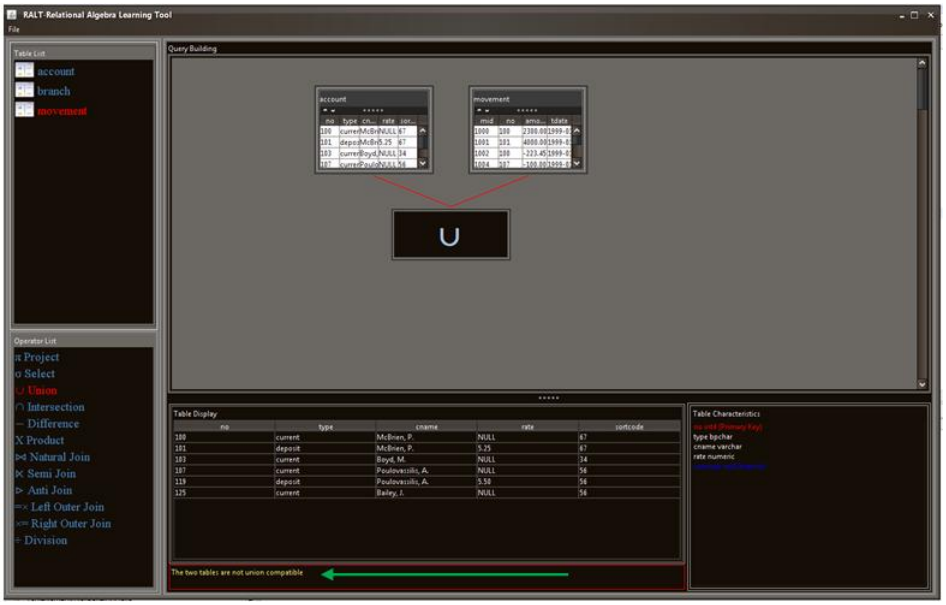


Figure 7.10: Screenshot: Showing error message displayed when Union operator is supplied with incompatible inputs

7.6 Data Lineage

In RALT user can easily see how a particular row has been derived. To view the lineage of a row, we first select the row by left-clicking on it. The selected the row gets highlighted. Once our desired row is selected, we right click on the box containing the panel which displays a pop-up as shown in Figure 7.11. We have selected the third row of the table.

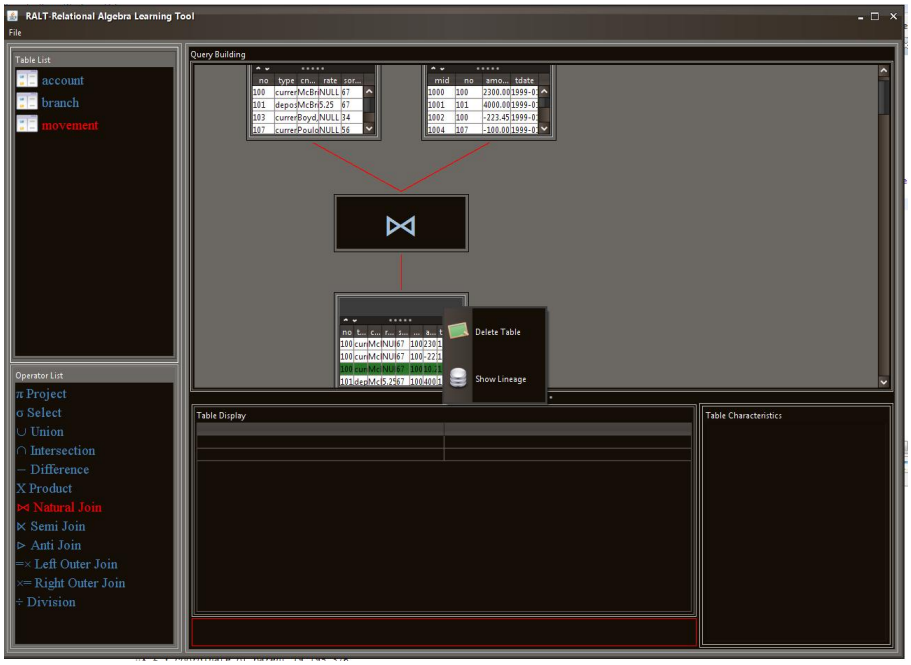


Figure 7.11: Screenshot: Showing Popup when the mouse is right clicked on the box

By clicking on the *Show Lineage* item of the popup menu we can view the rows in the other tables of the query, which have appeared higher up in the query tree. This can be seen by Figure 7.12. Notice that the row whose lineage we are finding has been highlighted in red together with the rows of tables which have contributed in producing this row.

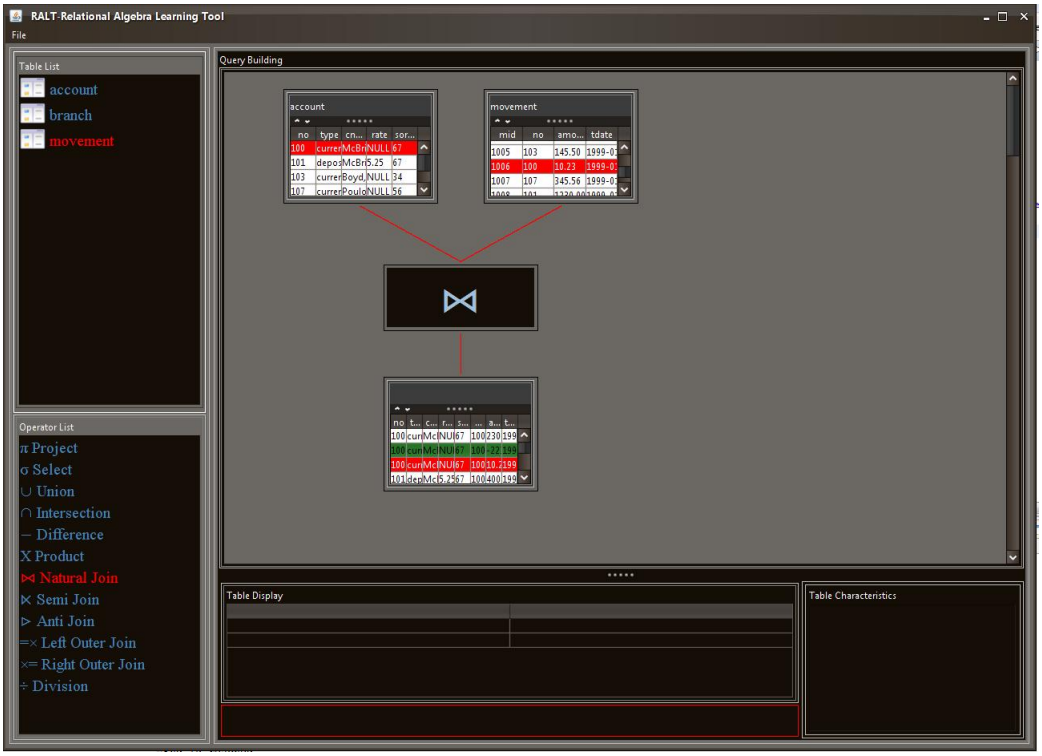


Figure 7.12: Screenshot: Data Lineage of the selected row

7.7 Delete Components

Our system is equipped with a feature that allows user to delete components. For example let us consider the query shown on Figure 7.13, we have executed.

After executing the query, the user may realise that they provided a wrong table as the second input and they wish to change it. This can done by right-clicking on the second input when the popup box. Selecting the first item in the popup list, labeled as *Delete Table*, the query takes a new form as shown in figure 7.14. We can see that the second input has been replaced by an empty panel and the result of the Natural Join is missing. The previously complete tree structure of the Natural Join is currently replaced by an incomplete structure with one input table (the first table).

The user can add a new table as the second input to the Natural Join, by dropping it on the empty panel (the bordered by red box). On receiving the second input the Natural Join operator is complete and gets executed showing the new result.

However, we have certain rules when a user wants delete an item from the system. They are the following:

1. If the user deletes a table which does not act as an input to an operator, it gets removed from the screen.

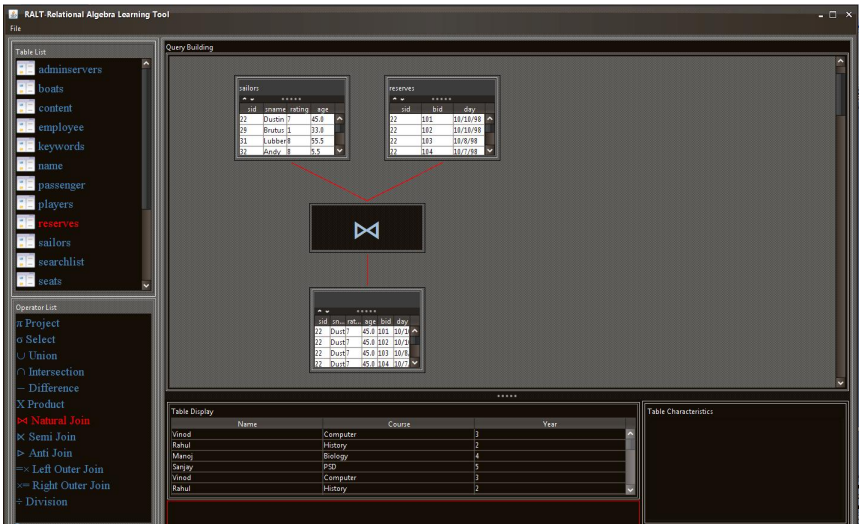


Figure 7.13: Screenshot: Showing a complete execution of a Natural Join query

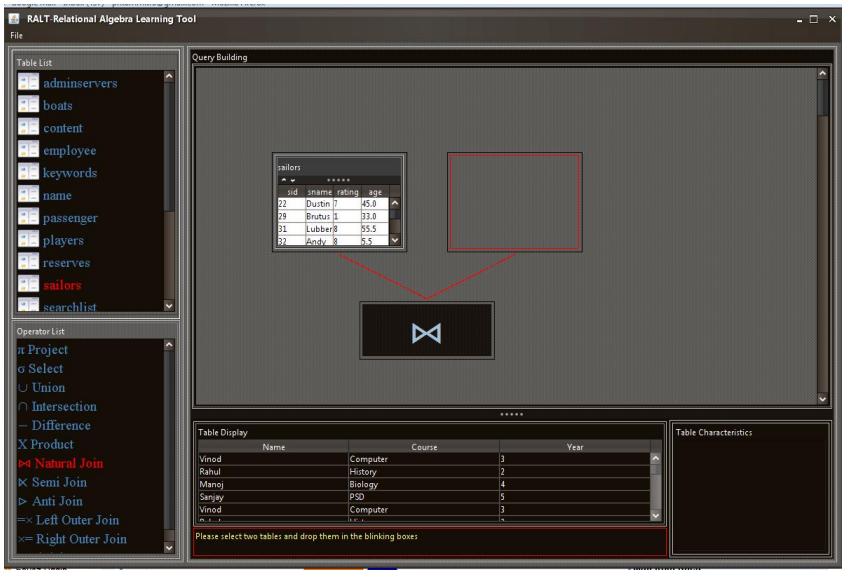


Figure 7.14: Screenshot: Transformation of the Natural Join query when the second input is deleted

2. If the item has no parents, then deleting it deletes the entire query.

3. If the item deleted is a result of an operation the following happens:

   - Everything below the box displaying the operator symbol(whose result we are deleting) is deleted.
   - If the operator is binary , it replaces its second input with an empty item where users can drop new tables to complete the necessary inputs to the operator so that the query can execute.
   - If the operator is a unary operator, it gets removed leaving its sole input (first input in case of binary)on the screen on which new operators could be dropped.

4. If the item deleted is an operator node, it just keeps its first input table and deletes anything else it is connected to by the red or blue lines.Any sub-queries built on the result produced by this operation also gets deleted.

5. If the item deleted is an input for an operator two things happen:

   - If the operator had all the required input(s) and had already executed, it will delete the result it has produced together with any sub-queries built on that result.
   - If the operator is binary and the table deleted is the second input, it will just transform into an incomplete query structure by replacing the second input with an empty panel where the user can add inputs later. If the item to be deleted is the first input, we delete all the components below it and either step2 or step3 will apply.

Following these rules, the user can delete items for queries.

Clear panel

The user can clear a panel of all elements if they decide to. To do this they must right-click anywhere on the screen which will display a popup and form that select the **Clear Panel** option. This clears the entire screen.

7.8 Playing with visualisation elements

In RALT users can move the boxes displayed on the scrren when building a query. These boxes represent different types of information. They sometimes display data in a table, sometimes the symbol for an operator or the constraints attached to an operator. Sometimes they display nothing but appear as an empty box to which user can add table. No matter what they represent, user has the privilege of adjusting the size of the boxes or even dragging them to a different location on the scree.

Figure 7.15 displays one such box. In order to change the size of the box or move it we first need to left-click on the outer *black* border of the box labelled with the red arrow. On clicking on the outer border of the rectangular box, eight small white square are displayed around the border. On hovering the mouse over these square boxes user, can notice that the mouse cursor changes to an two directional arrow. When the cursor takes this form the user by holding onto the left mouse button can move the mouse cursor to change the size of the box.

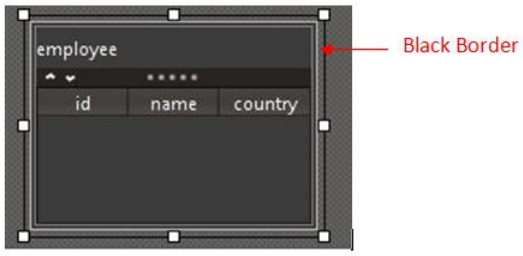


Figure 7.15: Screenshot: Rectangular Box for representing displaying information in Query Building Panel

Alternatively after clicking on the black outer border of the box, when the user hovers the mouse over the area on the border not containing the white boxes, the curson changes to a diamond shape. User can now hold on to their left mouse button and move it over the screen when the rectangular box also moves. The box remains where the user drops it. Once dropped all the connections the box previously had are re-drawn in relation to the boxes current position. This feature allows users to change the position of boxes if they overlap with boxes in other queries, does making query computation difficult.

7.9 Log Out

When the system wants to Log Out of the system, they can click on the File menu located at the top left corner of the application screen and select the Log Out option. The system gives asks the user whether they intend to disconnect from the system using a popup box. If the user clicks on the **Cancel** button, no action takes place and the popup box dissapears. If the user selects the **Ok** button, then we the user is directed to the Login Screen.

Chapter 8

Testing & Evaluation

In this chapter we attempt to evaluate the application built during the course of this project. We realised that it is difficult to come up with a fair, unbiased appraisal by just carrying out tests on the system without any user testing. This encouraged us to involve people who have been not been associated with the development process when testing this application and gather their views on the system.

8.1 Introduction

To evaluate the success of the RALT, we have chosen different ways of assessing the software. First, several test cases were created, and the program is applied to solve both simple and complex Relational Algebra queries. We also present the system in front of some Computing students allowing them to interact with the system and later asking for their feedback.

8.2 System Assesment

Majority of the components in RALT has been developed from scratch. This includes the GUI and also our own implementation of the Relational Algebra operators. Hence responsibility lay with the development team to make sure that all the components worked correctly. Fortunately, modular approach taken for the development of the different components of RALT made testing these components fairly straight forward. Table 8.1 summarizes the aims before we started the project together with information about whether they were achieved or not at the end.

We now justify how we came to the conclusion that these aims were achieved.

8.2.1 Connecting to Databases

This was a simple test. On our systems login screen, we entered the necessary information and monitored whether the system connected to the correct database or not. For example, we submitted the following information on the login screen of RALT:

- *address* – db.doc.ic.ac.uk
- *port* – 5432

Aims	Result
Connecting to Database	Completed
Storage of Data	Completed
Displaying Data	Completed
Implementation of the basic Relational Algebra Operators	Completed
Building Queries	Completed
Deletion of Tree Nodes	Completed
Implementing advanced Relational Algebra Operators	Completed
Introduction of Data Lineage	Completed

Table 8.1: Status of predefined aims at the end of the Project

- *database* - lab_bank_branch.
- *username* - lab
- *password* - lab
- *database type* - Postgres

On submitting this information the main application screen of the system loaded up and displayed table names in the Table Name Panel which matched with the tables present in the database whose details we had provided. Hence we could conclude that our system was connecting to a database from the information provided by the user.

8.2.2 Storage of Data

A unique feature of our system is that it stores all the data fetched from the database in memory. We ensured that our system was extracting all the necessary information by printing all the components of the data structure which we had designed for storing this information. We noticed that not only all the rows for each table in the database were fetched correctly but the metadata fetched (e.g. Primary key of a table, Foreign key reference etc) were also correct. This feature could be more easily tested once the user interface was ready as we could see the contents fetched from the database in a graphical manner.

8.2.3 Implementing basic and advance Relational Algebra operators

We chose to test our implementation of the Relational Algebra Operators through Unit testing. Unit testing is a testing methodology which gives the programmer the confidence that individual units of source code are not behaving abnormally. We chose JUnit test framework for this purpose.

Being able to test whether the code is behaving properly after any modifications is made to it allows us to be reassured that changing small amount of codes does not break the larger system[20]. Without automated testing tools like JUnit, retesting becomes a tedious and inaccurate process. Allowing testing process to occur frequently and automatically, we can keep software coding errors to a minimum. We create JUnit test classes for each operator. These are located under the package *com.imperial.testCases*.

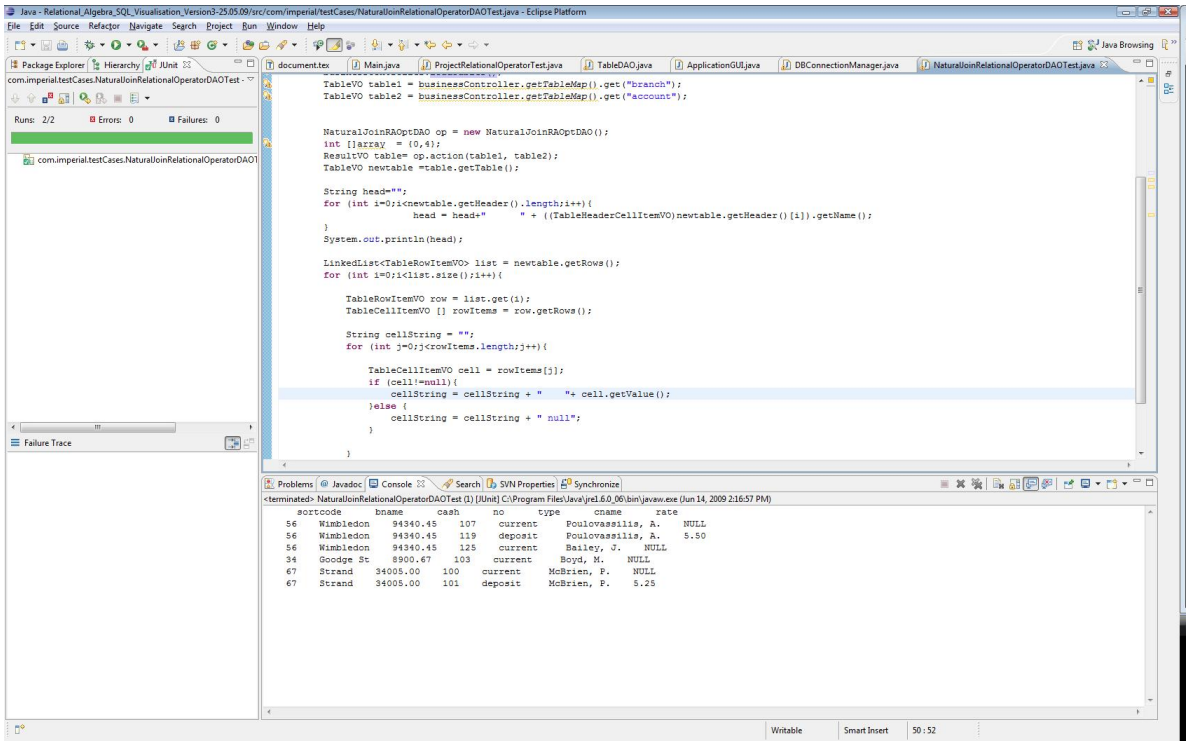


Figure 8.1: Screenshot: JUnit Test Case for Natural Join Operator

In Figure 8.1, we can see the result produced by the JUnit test case for the Natural Join operator. When running the JUnit test cases produce results for known inputs different from our expected result, we immediately realise that the method has not behaved properly. Moreover we can write separate test cases testing various functionalities of the function and run all of them together with the help of JUnit. JUnit proved to be extremely helpful since we were implementing several operators and often had to change the code for the operators. With JUnit at hand, we could quickly run the designed test cases after a modification is made to the code for an operator and observe the results. Hence, after these test results we could also confirm that our implementation of the Relational Algebra operators were behaving correctly.

8.2.4 Testing GUI

While implementing the Relational Algebra operators, we also simultaneously carried out the development of the GUI. As we had mentioned earlier in Section 4.3, our main application GUI is made up of several components. Using Java Swing, we could develop and test these components individually without the need of integrating them into the application GUI. This way we could determine the look and feel of the individual subcomponents without worrying about the other parts of the system GUI.

Once all the individual parts of the GUI are designed, we integrated them together into the system to create the main application screen. We could now test the interaction between different components of the GUI. For example we could now see whether dragging a table name from the Table Name Panel

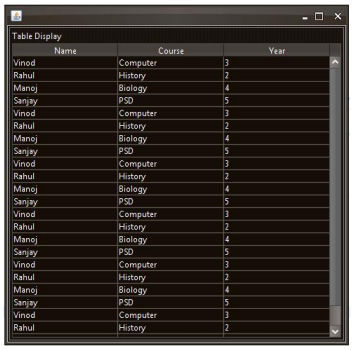


Figure 8.2: Screenshot: After the development of Table Data Display Panel

and dropping it onto the Table Data Display Panel, displays the data correctly or not. This was not possible earlier as the different components did not interact with one another.

8.2.5 Query Building

Query building is an essential feature of our system. We tested this functionality by creating a number of test cases which involved a sequence of actions which had to be carried out. We started with keeping things small, testing each of the individual operator, verifying whether correct input and output were displayed for each operator. Once satisfied we took a bigger step of using the results produced by one query as input to other operators. When observed whether queries being built, displayed the correct sequence of activities. Happy with the results we decided to build individual queries and then using the result, produced after a sequence of operators in one query as inputs to another. For example if we consider Figure 8.3, we have executed the following sequence of operations [1]:

- We first drop a Project Operator and a Semi Join operator onto the Query Visualisation Panel.
- We drop table name *account* as input for the Project operator and choose the fields *no* and *sortcode* as fields to be attributed. The Project operator does its computation and displays the results in a table (marked as "Project on Account).
- We drop the movement table as second input to the Semi Join operator.
- As indicated by the yellow right angled arrow, we drag the result produced by the Project operator and drop it on the empty panel acting as the first input to the Semi Join Operator.
- The Result is shown in Figure 8.4

[1] we carry out these operations on the database *lab_bank_branch* located at *db.doc.ic.ac.uk*

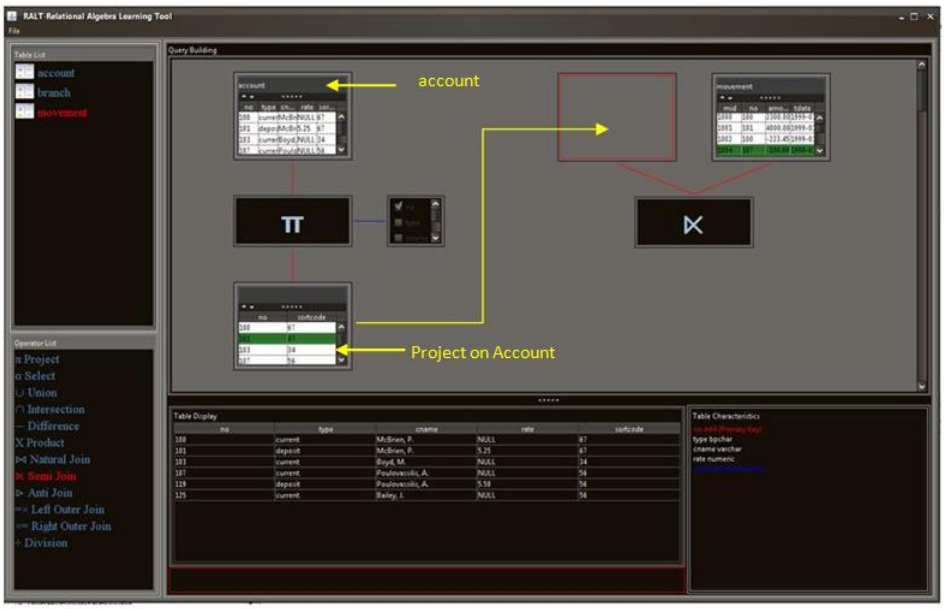


Figure 8.3: Screenshot: Testing the transfer of output of one operator as input of another in a separate query

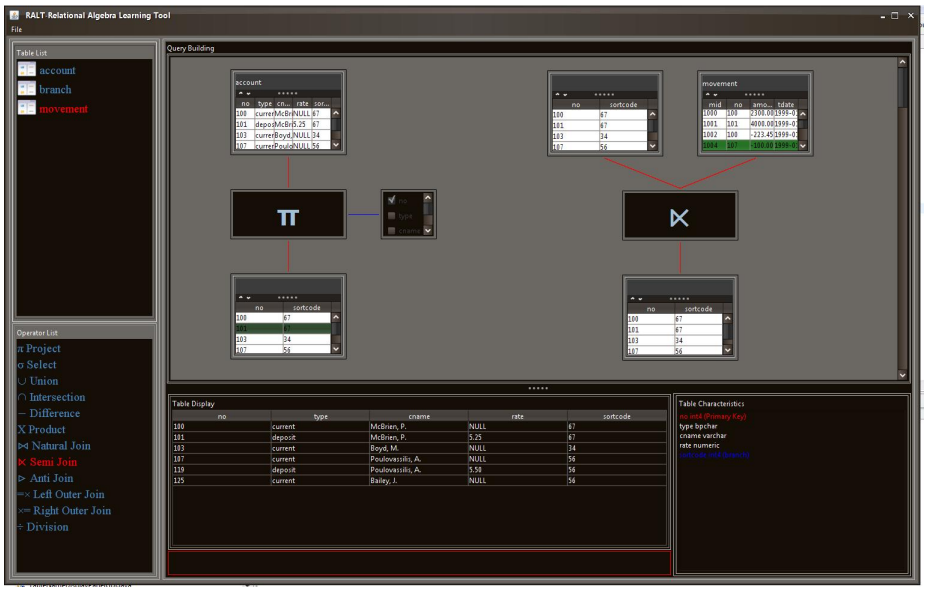


Figure 8.4: Screenshot: Testing the construction of two separate queires

8.2.6 Introduction of Data Lineage

Since Data Lineage was a late addition to our project, the user interface was already complete when we began testing this feature. Hence, we thought the best way to test data lineage will be to select a row of a table and call the data lineage operation on that and verify whether the correct in the query tree were highlighted as the lineage of our selected row. We first started with testing the Lineage on individual operators. For example we provided two inputs two a Natural Join Operator which gets executed and produces a result. We carry data lineage on say the first row of the result produced and verify whether rows in the input tables marked as the lineage for our row of concern is actually accurate or not.

Having successfully verified that lineage operation worked perfectly for each of the operators implemented in our system, we ventured to test it on a more complex query. Below we give an example of a test case query we created for testing purposes. The query can be built by carrying out the following sequence of operations and diagrammatic representation of the query is given in Figure 8.5:

- Execute a Project operator on the table *account* selecting column *no*, *type* and *cname* of the table.
- Execute a Natural Join operation on the result produced by Project with the *movement* table.
- Select the second row of the result produced and execute the data lineage operation on this row.
- Observe the rows in the query tree that have been highlighted red.
- Verify whether the rows are the correct rows or not.

By carrying out such tests like this we received an assurance that our data lineage functionality could work correctly for multi level query tree.

8.3 User Testing

Since RALT is designed to help students in learning Relational Algebra, we realised the best to evaluate this system is by allowing users to use it and monitor their activities and listening to their feedback, specially the negative ones. On using the system the following response was received from a Computing student at Imperial College.

Md. Salih Noor, MEng Computing: "RALT makes the process of query building very easy. And the best part is I dont need to write anything."

Similar responses were received from other students who got a chance to lay their hands on the system. However, to obtain a statistical meaningful measure of RALT's usability based on the opinions of its prospective users, we need to carry out a formal survey. Unfortunately some elements limit the scope of this study, including the following:

- To attain a high level of precision, a large number of students must be surveyed. However, due to the lack of time and the busy schedule of most of our colleagues prevented us from surveying many students.

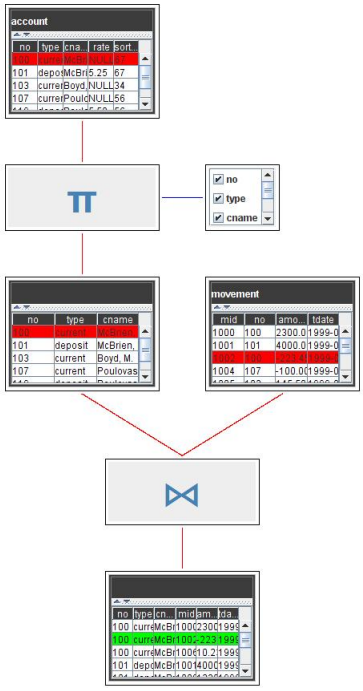


Figure 8.5: Screenshot: Testing Data Lineage

- We cannot rule out the fact that the survey is inherently biased since the people being surveyed are member of the same academic discipline. All of them are at present, final year MEng Computing student at Imperial College. These people are already accustomed to the Relational Algebra algebraic notation and may even feel more comfortable to write queries using algebraic notation instead of visual interaction.
- Since our tool is actually designed for students learning Relational Algebra for the first time, the surveyed students do not match the best profile for our target audience. To produce more meaningful results, the study must be extended to include users from various backgrounds, primarily the ones learning Relational Algebra for the first time.
- Given the time the students could provide, we kept the scope of this survey to a minimum.
- Given the small sampling population, the sampling error and non-sampling errors, such as that due to false response, become highly significant.

Although we need to consider these factors when evaluating the results of our survey, nevertheless the survey highlighted the strength and drawbacks of our system. Four students participated in our survey. We call them User1, User2, User3 and User4. Our goal was to solve some Relational Algebra queries on paper and time how long each user take to solve each query. Then we ask our users to solve the same set of queries using RALT. We again time how long it takes each user to execute the each query and compare the results with the paper based approach.

8.3.1 Carrying out the survey

Before starting, we handed out printed copies of Section 3.1 to each user. This allowed them to refresh their knowledge about Relational Algebra just by flicking through a few pages. We then presented each user a sheet of paper containing 3 relations and 9 questions. The sheet is attached in Appendix A. Model answers for the query are provided in Appendix B. User had to transform the verbal description of each question into a Relational Algebra queries using the twelve Relational Algebra operators mentioned in Section 3.1. They were allocated 27 minutes for completing all 9 questions. Since an important feature of RALT is that it allows complex queries to be broken down into small easily solvable queries, our questions often require the users to write complex queries. This way we can test how easy it is for users to build complex queries in RALT, when in the second half of the survey they solve these questions in RALT.

We monitored the workout of each user individually, timing how much time they took to solve each question. Once they finished or the time allocated was over, we collect their workout. After a short break we introduce them to RALT. We give them a brief introduction on how queries are built. We work out some demo queries in front of them, allowing them to get used to the user interface before we start working out. We also show them the feature of data lineage and how they can use it to track how a result was derived. Then we provide them with two use cases and ask them to carry out the activities. These tests gave our user a flavour of RALT and they felt more confident to solve the nine queries under a time constrained environment.

We then began the second phase of the survey. We connect the RALT applications to a database which displays the same relations user had worked with in the first part of the survey. We again ask them to carry out their activities. They are again given 27min for solving the 9 questions. We observe how long they take to solve each query. We also pay attention how the user is solving the questions. We wait until the user finishes all the queries, or 27 minute is over. After this we collect the users work.

8.3.2 Analysis of Survey

As we can see from Table 8.2, we can see that users were able to answer more questions correctly using RALT than the paper based approach. Majority of the students have been able to able to answer 7-9 queries correctly. Also from Figure 8.6 we can notice that the most of students have managed to finish building queries using RALT in significantly less time than paper based.

No. of correct question	Paper based	RALT
0-3	2	0
4-6	2	2
7-9	0	2

Table 8.2: Results from Paper based and RALT based Test

When analysing the workout of our users in the paper based test, we notice that they have spent a large proportion of their time trying to find the intermediate results when building a complex query. Making an error when producing one of the intermediate results correctly gets carried on to the later stages of the query building process, as a result of which incorrect result is derived. We also notice that the users struggled considerably in the last two questions of our set which involved the Division operators. All the users took the path of not answering the question using the Division operator.

When analysing the work done by the users in RALT, we can see a considerable number of improvements. Most users built small sub-queries and coordinated the output of these queries to get the final result. The fact that users did not have to compute the intermediate results themselves, gave them extra time investigate different methods of solving a query. We also noticed that in RALT for the last two questions, user used the Division operator. When asked about the idea of using operators like the Division, the users mentioned the fact the unfamiliarity of the operator was a reason for not using it in the paper based version of the test, as it had slipped out of their mind. However, as the Division operator symbol could be seen in the user interface, it prompted them to use it.

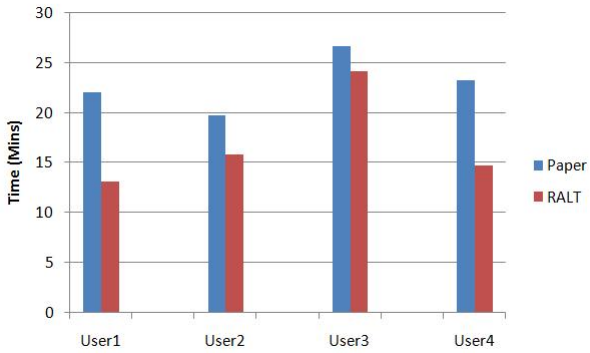


Figure 8.6: Comparison of time taken by user to complete the 9 questions using both paper-based approach and RALT

Although from Figure 8.6 we observe that majority of users managed to solve the nine queries in subsequently less time with RALT when compared to the paper based approach, there was an exception. Though User3 took less time when using RALT, the time difference was not significantly low compared to the average improvement of 40%. When investigating the reason for this, we found User3 found that when building complex queries, the Query Visualisation Panel soon got filled up with too much information which affected her navigation time. We realized that User3 did not use the feature of deleting unwanted queries with our delete option. Also in RALT we have tried to minimise any importance given to syntax when making a query. So when in a Select operator, the user is required to enter a conditional value the quotes attached to the value as we see in traditional system, is left out. For example, if we want to select the row from table *account* where type is current, instead of specifying the constraint like type=‘current as we do in traditionally, we write it simply as rate=current. However, User3 did not realise this feature and she received wrong result for entering a String constraint inside quotes.

When writing queries by hand, we noticed our users often wrote unoptimised queries. For example they would often execute a Select operation after performing a Natural Join. This way we compare many rows unnecessary. But when building the same queries in RALT, the visualization of the intermediate results made them aware of the unnecessary operations being performed and they became more motivated to write optimized query. We were happy to see this change in attitude of our users as one of the main aims of our learning tool was to make users realise how queries can be optimised.

8.3.3 Questionnaire

At the end of the survey we gave our users a questionnaire to give us a feedback of their user experience with RALT. The questionnaire was designed to capture their reaction when working through the system in a relaxed manner. This was carried out before the compputer based test started when we handed our users two use cases so that they can get them accustomed to the system. These use cases were for building query trees and performing data lineage. We used the sequence of activities illustrated in 8.2.5 for query building while that in 8.2.6 for data lineage. The following was the questionnaire:

1. (a) Carry out the procedure of query building in RALT as described in section 8.2.5.
    - Did u find the correct Result?
    - How difficult did u find the procedure of query building?
      (Very Easy, Easy, Medium, Difficult, Very Difficult)

   (b) Carry out the procedure for determining data lineage in RALT as described in section 8.2.6.
    - Did u find the correct Result?
    - How difficult did u find the procedure of finding the lineage for a row?
      (Very Easy, Easy, Medium, Difficult, Very Difficult)

2. To what extent do u rate the following statements:
   (Very Easy, Easy, Medium, Difficult, Very Difficult)
    - Can a new user get familiar with the system easily?
    - Do you think the user interface is easy to use?

3. Do you think RALT should be used for the purpose of teaching?
   a) Yes b) No

4. 4. Do you think you learned more using this tool?
   a) Yes b) No

We got the following response:

1. (a) Carry out the procedure of query building in RALT as described in section 8.2.5.
    - Did u find the correct Result?
      Yes: 4 No: 0
    - How difficult did u find the procedure of query building?
      (Very Easy, Easy, Medium, Difficult, Very Difficult)
      Very Easy = 1, Easy =2, Medium =1, Difficult=0, Very Difficult=0

(b) Carry out the procedure for determining data lineage in RALT as described in section 8.2.6.

- Did u find the correct Result?
  Yes: 4 No: 0
- How difficult did u find the procedure of finding the lineage for a row?
  (Very Easy, Easy, Medium, Difficult, Very Difficult)
  Very Easy = 3, Easy =1, Medium =0, Difficult=0, Very Difficult=0

2. To what extent do u rate the following statements:
   (Very Easy, Easy, Medium, Difficult, Very Difficult)

   - Can a new user get familiar with the system easily?
     Very Easy = 1, Easy =2, Medium =1, Difficult=0, Very Difficult=0
   - Do you think the user interface is easy to use?
     Very Easy = 1, Easy =1, Medium =1, Difficult=1, Very Difficult=0

3. Do you think RALT should be used for the purpose of teaching?
   a) Yes=4 b) No=0

4. 4. Do you think you learned more using this tool?
   a) Yes=3 b) No=1

The main negative point stated by the student was the fact, RALT at times had too many information on the query building screen which made it confusing for the user as they sometimes lost track of the work. Also, they stated that though initially they like the idea of being able to move the query tree elements on the visualisation panel, sometimes they would overlap one query tree with another and hence would like an automatic feature which will separate and align the query trees properly.

Besides the criticism we received, further discussions with the students together with the results of the study indicated that users have a generally favourable opinion of RALT. They enjoyed the method of building queries through graphical interactions. Everyone said the feature of data lineage assisted significantly in understanding how a particular row was derived. However, given the limitations of the study as we discussed earlier, one must interpret these results with care. For a more statistically meaningful appraisal of the system's usability, a larger and more rigorous study needs to be conducted.

Chapter 9

Conclusions & Future Work

We open this chapter by looking at how our current work can be extended and outline the potential direction for further research. We then try to make a conclusion about the outcome of this project. We attempt to answer questions like – What has been particularly successful? What went wrong?

9.1 Future Work

Although the user feedback we received for RALT was overwhelming, we feel there are still a large number of features which could be added to our system that will enhance the learning of Relational Algebra for students. Below we discuss some of these features:

- Our system is missing the basic aggregate operations which are included in all databases. Such operations include Sum, Count, Average, Maximum, and Minimum. Having these operations in our system will allow users to manipulate integer and they can execute queries such as *find the customer in the table with the maximum account balance.* Execution of such queries is currently not possible in our system.
- Although not part of Relational Algebra, OLAP (ROLL UP, PIVOT, GROUP BY etc.) operators are extremely useful when viewing data. For example they allow in grouping together particular rows of a table based on a certain column field. We feel the OLAP CUBE operator in particular will be a fine addition to our system. CUBE operator is extremely helpful in the analysis of data. Currently not all DBMS like Postgres support the CUBE operation; as a result users using such databases is forced to miss out of this important operator. However, it is a different case with our system. We store all the data fetched from a database in memory of the client machine. Hence, we can implement our own CUBE operator that will allow the user to perform this operation on the data held in memory. This way the user need to be connected to a particular type of in order to execute OLAP operators which are supported in all DBMS.
- Although our tool is meant for the teaching of Relational Algebra, we can extend it to make it an SQL learning tool as well. This way the user can execute both Relational Algebra and SQL queries from one single application and can compare the results produced by the two.
- An important feature to have in our system will be to allow the user to write SQL queries into the system in the traditional SQL way i.e. SELECT * from *users* WHERE uid=10, and then transform this query into a tree like format showing the stepwise step execution of the query using Relational Algebra.

- Although the current user interface of RALT is more powerful than the other learning tools for Relational Algebra, we feel having a 3-D interface will allow us to entice more users to use our tool. Having 3-D capabilities, the tool will no doubt make the learning of Relational Algebra a more exciting activity.
- Our system has no option where user can save their query and then load them at a later point in time. As a result everytime a user loads the system they have to start building all queries from scratch. This is a waste of time and we must provide them with an option of saving thier queries.

We have thus presented some of the more interesting possibilities for further work. Some of these ideas may be more promising than others.

9.2 Conclusion

Below we provide what we consider to be the primary achievements of the Project:

- Creating an application which can execute Relational Algebra operations without the need for the user to input any query by hand.
- Being able to display the query building process in a tree like structure.
- Using Java Swing for creating a user interface which is powerful as well as user friendly. It allows users to get accustomed to the system quickly and provides a easy man-machine interaction.
- Being able to implement all the Relational Algebra operators on our own and understanding how some of the advanced operators (i.e. Natural Join, Semi Join etc.) can be computed by y combining the basic Relational Algebra operators (Project, Union etc).
- Being able to implement the feature of data lineage in a learning tool, which to our knowledge has never been used before for academic purposes.

A major area of failure we think has been the way data is displayed on the Query Visualisation Panel. We noticed if the user did not go on building queries without deleting the ones they do not require anymore, the application screen will get filled with information very soon and it will be hard to navigate. Also the results produced by an operator when displayed on our Query Visualisation Panel is not given a name, which makes it hard for some users to identify the correct result. We should address these two issues quickly.

To conclude, we have implemented a system which allows users to compute Relational Algebra queries just by graphical interaction.

APPENDIX A

Given the three relations as shown below,

sid	sname	rating	age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	5.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5

Table 9.1: Sailors

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Table 9.2: Reserves

bid	bname	colour
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Table 9.3: Boats

Solve the following questions by computing them into equivalent Relational Algebra queries:

1. Find the names of sailors who have reserved boat 103.
2. Find the names of sailors who have reserved a red boat.
3. Find the colors of boats reserved by Lubber.
4. Find the names of Sailors who have reserved at least one boat.
5. Find the names of sailors who have reserved a red or a green boat.
6. Find the names of Sailors who have reserved a red and a green boat.
7. Find the sids of sailors with age over 20 who have not reserved a red boat.
8. Find the names of sailors who have reserved all boats.
9. Find the names of sailors who have reserved all boats called Interlake.

APPENDIX B

Some model Answers for the above queries are:

1. $\Pi_{sname}\left((\sigma_{bid=103}Reserves) \bowtie Sailors\right)$
2. $\Pi_{sname}\left((\sigma_{colour='red'}Boats) \bowtie Reserves \bowtie Sailors\right)$
3. $\Pi_{colour}\left((\sigma_{sname='Lubber'}Sailors) \bowtie Reserves \bowtie Boats\right)$
4. $\Pi_{sname}\left(Sailors \bowtie Reserves\right)$
5. $\Pi_{sname}\left(((\sigma_{colour='red'}Boats) \cup (\sigma_{colour='green'}Boats)) \bowtie Reserves \bowtie Sailors\right)$
6. $\Pi_{sname}\left(((\Pi_{sid}\left((\sigma_{colour='red'}Boats) \bowtie Reserves\right)) \cap (\Pi_{sid}\left((\sigma_{colour='green'}Boats) \bowtie Reserves\right))) \bowtie Sailors\right)$
7. $\Pi_{sid}\left(\sigma_{age\geq 20}Sailors\right) - \Pi_{sid}\left((\sigma_{colour='red'}Boats) \bowtie Reserves \bowtie Sailor\right)$
8. $(((\Pi_{sid,bid}Reserves) \div (\Pi_{bid}Boats)) \bowtie Sailors)$
9. $\Pi_{sname}\left(((\Pi_{sid,bid}Reserves) \div (\Pi_{bid}\left(\sigma_{bname='Interlake'}Boats\right))) \bowtie Sailors\right)$

Bibliography

[1] Dr Gordon Russell. *Relational Algebra*. http://db.grussell.org/section010.html.

[2] *Relational Algebra*. http://www.databasteknik.se/webbkursen/relalg-lecture/index.html.

[3] C. Batini, T. Catarci, M.F. Costabile, and S. Levialdi. Visual query systems: a taxonomy. In *Proceedings of the IFIP TC2/WG*, volume 2, pages 153–168. Citeseer.

[4] A.P. Appel, EQ Silva, C. Traina Jr, and A.J.M. Traina. iDFQL–A query-based tool to help the teaching process of the relational algebra. In *Proceedings of World Congress on Engineering and Technology Education (WCETE2004)*, pages 429–433, 2004.

[5] Steven Wallace Harris. *RAIN Relational Algebra Interface*. http://www.cs.stir.ac.uk/courses/it/projects/PastDissertations/Abstracts/2005-2006/Harris.RTF.

[6] S.Levialdi of University of Roma. C.Bantini C.Catarci of University of Roma, M.F.Costabile of University of Bari. *Visual Strategies for Querying Databases* .

[7] T.L.Kunji Eds. North Holland. *ESCHER Interactive Visual Handling of Complex Objects in the Extended NF2-Database Model. In Visual Database Systems.*

[8] S. Sadiq, M. Orlowska, W. Sadiq, and J. Lin. SQLator: an online SQL learning workbench. *ACM SIGCSE Bulletin*, 36(3):223–227, 2004.

[9] Ganesh Variar. *The Origin of Data*. http://www.intelligententerprise.com/020201/503feat3$_1$.*jhtml*.

[10] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[11] Samsung Electronics. Won Kim. *On Metadata Management Technology: Status and Issues*. http://www.jot.fm/issues/issue$_2$005$_0$3/*column*4/*column*4.*pdf*.

[12] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. *Computer*, 1997.

[13] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. CIDR, 2005.

[14] P. Agrawal, O. Benjelloun, A.D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1151–1154. VLDB Endowment, 2006.

[15] D. Bhagwat, L. Chiticariu, W.C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal The International Journal on Very Large Data Bases*, 14(4):373–396, 2005.

[16] A. Silberschatz, M. Stonebraker, and J.D. Ullman. Database systems: Achievements and opportunities. *ACM Sigmod Record*, 19(4):6–22, 1990.

[17] *Java2D.* http://java.sun.com/products/java-media/2D/index.jsp.

[18] *Swing and SWT: A Tale of Two Java GUI Libraries.* http://www.developer.com/java/other/article.php/$10936_2179061_2$.

[19] *Java Drag and Drop.* http://java.sun.com/docs/books/tutorial/uiswing/dnd/intro.html.

[20] Jamie Scheinblum. Creating junit test cases.