Relational Algebra and SQL Query Visualisation

Giorgos Constantinou
gc106@doc.ic.ac.uk

Supervisor: Dr. Peter McBrien
Second Marker: Dr. Natasa Przulj

June 14, 2010

Abstract

Relational algebra and the industry standard SQL are core topic covered in undergraduate database courses. Database management systems translate SQL statements into a procedural query plan composed with operations similar to those that arise in relational algebra. Learning relational algebra makes students familiar with this process. Moreover they understand the difference between procedural and declarative query languages. Students that understand the relation between the two languages have the required knowledge to query databases. Additionally, it is equally important to appreciate the difference in expressiveness of the two languages and that SQL is a superset of relational algebra.

Currently there is no tool that can execute both types of queries and show the relation of the two in terms of the execution process. As a result students exercise on the two languages independently and do not acquire crucial knowledge which is useful in constructing optimised queries. Moreover, the available tools that support relational algebra fail to show the difference between basic and derived operations and students do not appreciate the simplicity of the evaluation process.

In this report we introduce Query Visualiser (QV), a tool that facilitates the learning process of both languages. Using extended relational algebra, SQL queries are translated into a procedural query plan. This tool provides transparency to the evaluation process by illustrating the results using a tree hosting intermediate results of the evaluation. The tree is constructed with operations that arise in relational algebra or in the extended one in an attempt to show their relations. In addition, the system can perform simple optimisations and see how they affect the performance using the analytical statistics of the evaluation. Also, the user can dynamically define new derived operators and see how they are converted and executed using basic operations.

Acknowledgments

I would first like to thank my supervisor, Dr. Peter McBrien for his valuable feedback and support during the course of the project. His critical reviews and guidelines always motivated me to carry on.

Finally I want to thank my friends and family for their endless patience throughout my time at Imperial College.

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Objectives . . . 10
- 1.3 Report Structure . . . 11

2 Background **13**
- 2.1 Relational Data Model . . . 13
- 2.2 Relational Algebra . . . 13
- 2.3 SQL . . . 16
- 2.4 Query Execution . . . 16
  - 2.4.1 Extended Relational Algebra . . . 17
  - 2.4.2 Translations . . . 18
  - 2.4.3 Intermediate results . . . 19
  - 2.4.4 Data lineage . . . 20
- 2.5 Relevant Technologies . . . 20
- 2.6 Related Work . . . 21
  - 2.6.1 WinRDBI . . . 21
  - 2.6.2 iDFQL . . . 22
  - 2.6.3 RELATIONAL . . . 23
  - 2.6.4 RALT . . . 24
  - 2.6.5 Aqua Data Studio . . . 25
  - 2.6.6 Features Comparison . . . 25

3 System Architecture **27**
- 3.1 View Layer . . . 27
- 3.2 Controlling Layer . . . 27
- 3.3 Model Layer . . . 29

4 Query Representation and Execution **30**
- 4.1 Query Syntax . . . 30
  - 4.1.1 RA Syntax . . . 30

4.1.2 ERA Syntax . . . 31
4.1.3 SQL Syntax . . . 32
4.1.4 Syntax Errors . . . 32
4.2 Query Tree . . . 32
4.3 Query Evaluation . . . 33
4.3.1 General Evaluation Module . . . 33
4.3.2 Specialist Evaluation Modules . . . 33
4.3.3 Evaluation Errors . . . 35
4.3.4 Views . . . 36
4.3.5 Execution Statistics . . . 37
4.3.6 Data Lineage . . . 39
4.4 Algebra Optimisation . . . 40
4.4.1 Unary Input Case . . . 40
4.4.2 Binary Input Case . . . 41

5 Workspace and Visualisers 42
5.1 Relation Explorer . . . 43
5.2 Import . . . 43
5.3 Tab Containers . . . 44
5.4 Query Panel . . . 45
5.5 Query Editor . . . 45
5.6 Result Visualisers . . . 47
5.6.1 Data Grid . . . 47
5.6.2 Intermediate Results . . . 48
5.6.3 Chart . . . 50
5.6.4 Console . . . 51
5.6.5 Execution Cost . . . 51
5.7 Visual Optimiser . . . 52
5.7.1 Tree Execution Time . . . 54
5.7.2 Tree Maximum Memory Usage . . . 54
5.7.3 Operation Nodes Execution Time . . . 54
5.7.4 Operation Memory Usage . . . 55

6 Implementation Details 56
6.1 Overview . . . 56
6.2 Graphical Interface . . . 57
6.3 Database Representation . . . 57
6.4 Workspace Persistence . . . 58
6.5 Query Tree Representation . . . 58
6.6 Parsers . . . 58
6.7 The Evaluation Engine . . . 59

6.8 Dynamic Operator Construction . . . 60
6.9 Mono and multi-platform compatibility . . . 61
6.10 Third-party Modules . . . 61
6.10.1 Text Editor . . . 61
6.10.2 Drawing Charts . . . 63
6.11 Creating ERA Trees from SQL . . . 63
6.12 Extending QV . . . 63
6.12.1 Extending the existing languages . . . 64
6.12.2 Introducing a new query language . . . 65

7 Correctness and Stability **67**
7.1 Experiments . . . 67
7.1.1 Unit testing . . . 67
7.1.2 Stress Testing . . . 68
7.2 Results . . . 68
7.3 Analysis . . . 69

8 Usability Experiments **70**
8.1 End-User Test . . . 70
8.1.1 End-user profiles . . . 70
8.1.2 Exercise Sheet . . . 71
8.1.3 Questionnaire . . . 73
8.1.4 Results . . . 73
8.1.5 Analysis . . . 74
8.2 Nielsen's Usability Heuristics . . . 75

9 Conclusion **79**
9.1 Achievements and contributions . . . 79
9.2 Future Work . . . 80

Bibliography **80**

A QV Details **83**

B Tutorial Solutions **87**

List of Figures **90**

Chapter 1

Introduction

1.1 Motivation

Relational databases are the core of the introductory and advanced database courses. Most of these courses cover formal query languages such as the relational algebra (RA) and Structured Query Language (SQL). It is difficult for students to decide whether queries written on paper are correct or not. Alternatively it is much easier to do so using trial and error on a system capable of evaluating such queries.

Almost all of today's database management systems support SQL. They translate declarative queries into lower-level operations similar to the RA operations. It is important for the students to familiarise with this concept in order to get a better understanding on how the two querying languages are related. One of the most important properties of relational algebra and algebra in general is that each operation performed provides a result that can become the input to another operation. Typical queries that students are asked to construct are made up of multiple operations that can be visualised as a tree where operations are evaluated starting from the bottom. Complex queries are usually hard to form and understand and visualising the intermediate result of nodes is helpful. Currently most of the teaching tools that support relational algebra lack the ability to give feedback to the user in order to help in constructing queries. Others provide some feedback using intermediate results but they fail in relating SQL to RA which is crucial in helping the user write optimised SQL queries and often results in unoptimised queries due to lack of appreciating the relation between declarative query evaluation with the relational algebra operations.

1.2 Objectives

We present QV - Query Visualiser, a tool that facilitates the learning of RA and SQL using detailed explanation of the query execution process.. Here we outline the main objectives.

- **Building a database** - The user should can construct a database using the graphical interface which allows schema and data manipulation.
- **Database import** - As an alternative to manually constructing a database, the user can provide connection setting for an existing database from where QV imports the data into the workspace.
- **Query Syntax**- We design and implement a parser for a suitable syntax for relational algebra (RA) and extended relational algebra (ERA). In addition we will implement a parser for the select and set operation statements of SQL.
- **Query Evaluation** - Evaluate queries written in RA, ERA and SQL. The RA language evaluates all the basic operations. The ERA evaluation inherits the operations of RA and also includes other operations which provide semantic completeness for SQL. The SQL evaluation translates the SQL into a tree of ERA operations which can then be evaluated with the ERA evaluator.
- **Error Reporting** - To help the user debug queries useful feedback is provided in case of errors. It should be clear whether errors are due to syntax or evaluation reasons. Line and position of the error in the query are also provided where appropriate.
- **Intermediate Results** - When evaluating the query we temporarily store the intermediate results of all the operators applied and present them in a tree that represents the evaluation process. The user can navigate the tree and analyse step by step how the query was evaluated.
- **Dynamic Operators** - The user can define new relational algebra operators, ranging from simple operators like *set intersection* to complex operators like *division*. After evaluation the user can browse the intermediate results and see how the new derived operator was dynamically converted into basic relational algebra operations.
- **Data Lineage** - From the result the user can select rows and see at all nodes in the intermediate results tree what data contributed to the result. This will be indicated by highlighting the respective rows of the nodes.

- **Execution Statistics** - Timing and memory usage statistics will be provided in order for the user to understand the complexity level of queries.
- **Query Optimisation** - The system will be able to modify evaluation trees in order to optimise the performance. The user will see step by step this modifications and statistics on how each one affects the performance.

1.3 Report Structure

The report is structured into the following chapters:

- **Background** In this chapter we discuss the fundamentals of relational database models and formal query languages and investigate relevant technologies. We then look at different designs and implementations of tools relevant to our goal and we conclude with a comparison of their features.

The next three chapters cover the specification and the design of a tool that meets the objectives in Section 1.2.

- **System Architecture** We outline the modules of the system and we describe their respective tasks along with a description on how different modules of the system are related and their dependencies.
- **Query Representation and Execution** We specify a suitable syntax for the query languages and their semantics. We provide a description of an intermediate representation of queries using syntactical trees. Also in this chapter there is a description on how to perform data lineage, simple query optimisations and calculate statistics on execution performance.
- **Workspace and Visualisers** After we specified the evaluation process of the system in the previous chapter we describe the user interface and how the user can manipulate the database model and execute queries. We show the different methods of showing the evaluation results and described how data lineage is displayed.

- **Implementation Details** Based on the design we documented in the previous chapters we describe the important implementation details of QV

The next two chapters cover the evaluation of the implemented system

- **Correctness and Stability** This chapter is concerned with quantitative tests on the implemented system.

- **Usability Experiment** We evaluate the performance of the system by performing an experiment with a set of end-users and we discuss the outcome.

- **Conclusion** This chapter concludes the report by recognising the areas of success and failure and suggesting future extensions.

Chapter 2

Background

2.1 Relational Data Model

Introduced by E.F Codd[5] along with the relational algebra and it is the most common data model used for databases.

In a relational model a **relation** consist of a ***heading***(set of attributes) and a ***body*** (set of tuples) that represents the relation value. The *degree* of a relation refers to the number of attribute in the relation header and the *cardinality* of a relation refers to the number of tuples in the relation body (value). An **Attribute** is formed by the *attribute name* and the *domain* (type) and a **tuple** is a set of zero or more attribute values that conform to the relation header.

There are two **Relation Classes** which are classified as base relations and derived relations (views):

- **Base Relations** are defines by the data tuples that represent the relation value. In relational model implementations base relations are defined using a Data Definition Language(DDL) such an example is SQL which uses tables to represent base relations [1].

- **Derived Relations(Views)** are the result of applying relational operations on one or more relations.

2.2 Relational Algebra

Relational algebra is a procedural language based on algebraic concepts. Codd suggested a modification of the mathematical approach and defined the term 'relational

[1]SQL uses the CREATE TABLE syntax to define the base relations.

algebra'. It essentially consists of a collection of operators that take as input relations and produce relations as result. Since both the operands and the results are relations, expressions can be nested in RA as in any algebraic expression. Also because relations are sets all the usual set operators such as union can be used. In addition to the set operators other operators such as select[2] and join are used.

Basic Operators

RA relies only on five primitive operators proposed by Codd (Selection, Projection, Product, Union, Difference), which make RA a relationally complete query language. In addition to these operators, the rename operator (ρ) is also treated as primitive with the introduction of ISBL [3].

Operator	**Description**
Project $\pi_{A_1,...,A_n}(R)$	Returns tuples from R that can be defined as the set that is obtained when all the tuples in R are restricted to $\{A_1, ..., A_n\}$
Select $\sigma_C(R)$	*Generalised Selection* returns tuples from relation R that satisfy C which is a propositional formula of atoms allowed in *Normal Selection* and the logical operators *and* ($\wedge$), *or* ($\vee$) and *negation* ($\neg$). *Normal selection* $\alpha\theta\beta$ or $\alpha\theta V$ where: α, β are attributes of relation R. θ is a binary set operator $\{=, \neq, <, \leq, >, \geq\}$. V is a value constant.
Cartesian Product $R \times S$	Cartesian Product returns tuples from all the possible combinations of R and S tuples. Given R and S have n and m attributes the result will hold tuples of $n + m$ attributes
Union $R \cup S$	Returns all the tuples from R and S . The two relations R and S must be **union compatible**
Difference $R - S$	Returns all the tuples from R that do not appear in S . The two relations R and S must be **union compatible**.
Rename $\rho_{A/B}(R)$	Returns an identical result to R except that in all tuples attribute A is renamed to B

Table 2.1: Relational Algebra Primitive Operators

[2]Also called restriction. Not to be confused with SQL SELECT.

[3]Information Systems Base Language: One of the earliest database management systems to implement E.F. Codd's relational data model.

Derived Operators

A range of operators are available which derive from the primitive ones.

Operator	Description
Intersection $\mathbf{R} \cap \mathbf{S} \equiv R-(R-S)$	Returns the common tuples from R and S . The two relations R and S must be **union compatible**.
Theta Join $\mathbf{R} \bowtie_\theta \mathbf{S} \equiv \sigma_\theta(R \times S)$	Returns tuples from the Cartesian product which satisfy θ. θ is a predicate $a\phi b$ where: a and b are attributes of R and S respectively ϕ is a comparison operator $\{=, \neq, <, \leq, >, \geq\}$ **Equijoin** is a special type of theta join where ϕ is always $=$.
Natural Join $\mathbf{R} \bowtie \mathbf{S} \equiv$ $\pi_{R \cup S}($ $\sigma_{R.A_1=S.A_1 \ldots}(R \times S))$	Returns tuples from the Cartesian product where common attributes between R and S have the same values and removes duplicated columns. Usually it is required that there is at least one common attribute, if not then the operator acts only as a Cartesian product.
Semi Join $\mathbf{R} \ltimes \mathbf{S} \equiv R \bowtie \pi_{R \cap S}(S)$	Returns the same result as the natural join but the result only contains tuples from relation R.

Table 2.2: Relational Algebra Derived Operators Examples

Useful Equivalences[3]

$$\pi_{x,y}\left(\sigma_{z>0}\left(R\right)\right) \equiv \sigma_{z>0}\left(\pi_{x,y}\left(R\right)\right) \tag{2.1}$$

$$\sigma_{A \wedge B}\left(R\right) \equiv \sigma_A\left(\sigma_B\left(R\right)\right) \equiv \sigma_A\left(R\right) \cap \sigma_B\left(R\right) \equiv \sigma_A\left(R\right) \bowtie \sigma_B\left(R\right) \tag{2.2}$$

$$\sigma_{A \vee B}\left(R\right) \equiv \sigma_A\left(R\right) \cup \sigma_B\left(R\right) \tag{2.3}$$

$$\sigma_C\left(R \times S\right) \equiv R \bowtie_C S \tag{2.4}$$

$$\sigma_C\left(R \cup S\right) \equiv \sigma_C\left(R\right) \cup \sigma_C\left(S\right) \tag{2.5}$$

$$\sigma_C\left(R - S\right) \equiv \sigma_C\left(R\right) - \sigma_C\left(S\right) \tag{2.6}$$

$$\pi_x (R \cup S) \equiv \pi_x (R) \cup \pi_x (S) \tag{2.7}$$

$$\sigma_{A \wedge B} (R) (R \cup S) \equiv \pi_x (R) \cup \pi_x (S) \tag{2.8}$$

$$\sigma_{A \wedge \neg(B)} (R) \equiv \sigma_A (R) - \sigma_B (R) \tag{2.9}$$

$$R \bowtie (S \cup T) \equiv (R \bowtie S) \cup (R \bowtie T) \tag{2.10}$$

2.3 SQL

Structured Query Language (SQL) is a declarative language used in many implementations of the relational data model and is a super-set of RA. Since it is as expressive as RA it is also relationally complete. SQL uses **tables** to represent relations and **columns** to represent attributes. SQL is a Data Definition Language (DDL) and uses commands like CREATE, ALTER, DROP and RENAME to define the database schema. At the same time it acts as a Data Manipulation Language (DML) used to manage data within a schema based on commands like INSERT, DELETE, UPDATE SELECT and UNION. We will not be concerned with data entry with SQL statements therefore we will concentrate on the basic forms of the SELECT and set operation[4] statements. One of the main differences that we need to have in mind is that SQL relations (tables) are bags or *multi-sets* and not *sets* as in the RA.

2.4 Query Execution

There are many implementations and discussions on how RA can and is used as the background of SQL from the early stages[4]. When executing an SQL query it is parsed and translated into a sequence tree. Then the references to tables and columns will be checked against the database to validate their existence. At that point the query is translated into a Query Processor Tree [5] which expresses the initial declarative query in a procedural manner. Since at the core of an RDBMS there is an engine that evaluates these algebraic operations, a modification of the standard RA can be used which can also be used as an alternative query language for multi-sets directly.

[4]UNION, MINUS, INTERSECT

[5]Optimisation is performed at this point using various optimisers that will make adjustments to the tree

2.4.1 Extended Relational Algebra

In order to evaluate RA expressions in relational data models with multi-set semantics the Extended Relational Algebra (ERA) proposed by Paul Grefen and Rolf By[8] can be used. The first thing to note is that in ERA, sets are treated as multi-sets. All the standard RA operators have the same definitions except that the duplicates are not removed. The interest is only in the available operators and therefore their formal definitions are omitted from this report.

Generalised Projection

$$\pi_{F1,\ldots,FN}(R) \tag{2.11}$$

It extends the projection operator and permits the use of arithmetic functions.

Example

$\pi_{employee\_name,(current\_salary-previous\_salary)}(Employees)$

The operation will return the difference between the new and previous salary of each employee. Note that duplicates will not be removed.

Aggregate Functions

$$CNT_a(R) \tag{2.12}$$

Counts the number of instances of attribute a

$$SUM_a(R) \tag{2.13}$$

Calculates the sum of all values of attribute a from all rows.

$$MIN_a(R) \tag{2.14}$$

Returns the minimum value of attribute a from all rows.

$$MAX_a(R) \tag{2.15}$$

Returns the maximum value of attribute a from all rows.

$$AVG_a(R) \equiv \frac{SUM_a(R)}{CNT_a(R)} \tag{2.16}$$

Returns the average value of all value of attribute a from all rows.

Unique

$$\delta(R) \tag{2.17}$$

Removes the duplicated rows from the result of R (converts a bag to a set).

Group By

$$_{G_1,...,G_n}\Gamma_{F1_{A_1},...,Fk_{Ak}}(R) \tag{2.18}$$

Aggregates the result by partitioning the tuples based on common attributes G. The other attributes returned are the results of aggregate functions F on attributes A.

Example

$$_{department}\Gamma_{CNT_{employee\_name}}(Employees)$$

The Operation will return the number of employees in each department.

Outer Join

$$(R)\,[(LEFT|RIGHT) + OUTER\,JOIN]_{\,C}(S) \tag{2.19}$$

Behave like normal joins in RA but the OUTER token indicates that unmatched columns from the left and/or right accordingly should be matched with NULL values and be part of the result.

2.4.2 Translations

Instead of using basic RA or SQL we will be using ERA for evaluating queries. Note that we can convert RA expressions to ERA using the unique expression δ or by telling the evaluator to use set based semantics. Here are a few SQL statements and there respective ERA equivalents:

Select Statements

SELECT A_1,...,A_3 FROM R_1,...R_k WHERE C_1 AND...AND C_m

$\longrightarrow \pi_{A_1,...,A_n}(\sigma_{C_1,...,C_m} R_1 \times ..., \times R_k)$

If we use the DISTINCT token in the SELECT clause the ERA expression will need to contain the δ operator either before or immediately after the projection.

When the FROM expression contains a JOIN [6] we can translate it using the RA Cartesian product and join [7] and vice-versa

SELECT A_1,...,A_n
FROM R_1JOINR_k
ON C_1AND...ANDC_N

$$\longrightarrow \pi_{A_1,...,A_n}(R_1 \bowtie_{C_1 \wedge ... \wedge C_N} R2)$$

In an SQL statement it is often the case that relation are joined using the WHERE clause by equality check on columns from the two tables to be joined. When translating a query we can identify these cases and treat them as the basic join case or vice-versa. Even if additional conditions are specified, this is still practical since the the result of one operation can become the input of another.

SELECT A_1,...,A_n
FROM R,S
WHERE
$R.A = S.A$
AND $R.N = S.N$
AND $R.X > V$

$$\longrightarrow \begin{array}{l} \pi_{A_1,...,A_n}(\sigma_{R.X>V \wedge R.A=S.A \wedge R.N=S.N}(R1 \times R2)) \\ \equiv \sigma_{R.X>V}(\pi_{A_1,...,A_n}(R_1 \bowtie_{R.A=S.A \wedge R.N=S.N} R2)) \end{array}$$

Set Operations

(SELECT_STATEMENT1)

$UNION|EXCEPT|INTERSECT$

(SELECT_STATEMENT2)

$$\longrightarrow \begin{array}{l} \text{UNION case: } \delta(R_1 \cup R_2) \\ \text{EXCEPT case: } \delta(R_1 - R) \\ \text{INTERSECTION case: } \delta(R_1 \cap R) \\ \equiv \delta(R_1 - (R_1 - R_2)) \end{array}$$

The set operators provided by SQL require their two parameter relations(tables) to be union compatible like RA. Also the set operators result has no duplicates and is the same as with RA. You can specify using ALL that duplicates should be returned and the respective ERA expression will not be encapsulated by δ.

2.4.3 Intermediate results

When executing the query a syntax tree with ERA operations can be created. This tree can be evaluated starting from the leaf nodes until the head node is reached

[6] NATURAL JOIN, JOIN, LEFT JOIN, RIGHT JOIN, LEFT OUTER JOIN or RIGHT OUTER JOIN

[7] $\bowtie, \ltimes, \bowtie_\theta$

which is the result of the query. At each step of the evaluation the result of each of these operations is an intermediate result.

2.4.4 Data lineage

Data lineage is an effective method for understanding from where a particular value in a query result was derived and at the same time it can show the source of error when the query result is wrong. Such a feature is valuable in understanding key concepts around both the RA and SQL. Many different approaches exist that successfully implement data lineage and each one has a different cost. One way of achieving this is through meta data attached to the result tuples in order to retrieve the source of the data. This is a complex method and is used for lineage at the schema level and not the data. An alternative method is to store all the intermediate results of the execution and by applying the inverse of the operations used to produce the result, traverse the execution tree backwards in order to find the sources.

2.5 Relevant Technologies

For the development of an interactive learning tool there is a clear need of an object oriented language that is cross platform compatible, has useful data structures for the application and there is a good GUI library that can be used. The two main alternatives for these purposes are Java and .NET C#.

Java is highly portable and can run on almost all platforms but it's GUI library *Swing* has been around for more that a decade and the available controls need a lot of customisation in order to provide the user with a decent interactive environment.

Microsoft C# on .NET Framework offers a lot of data structures such as *DataTables* and *Dictionaries* that can be proved useful and cutoff a lot of developing hours. The biggest benefit of .NET is that it offers an easy to use GUI toolkit that can produce professional results that maintain the standards of commercial applications. Although until recently .NET was considered only for applications targeting Microsoft Windows systems, an open source implementations of .NET, **Mono** has been developed and it can perfectly simulate the .NET framework on Microsoft Windows, Linux and Mac OS X. The main restriction is that the developer must not make system calls that will not work on different platforms(E.g. Do not call win32 functions since they are only available in Windows).

2.6 Related Work

In this section we investigate relevant proposed and implemented tools. We discuss their strengths and weaknesses and compare their features in an attempt to outline what our system will inherit and how it compares to them.

2.6.1 WinRDBI

WinRDBI[7] is a graphical interface for RDBI[1] which is written in Prolog and it evaluates queries in relational algebra, domain relational calculus, tuple relational calculus and SQL. It is a Windows tool developed in Visual Basic and it uses the Relation/Tuple notation for the database schema.The interface of the system is split into 3 areas:

1. The**Query Definition Panel** where user can input an SQL or RA query in a simple text editor.

2. The **Query Result Panel** where the result of the executed query is presented on a grid.

3. The **Database Browser** which includes a list of the available relations and a grid that displays the body (data tuples) of the currently selected relation.

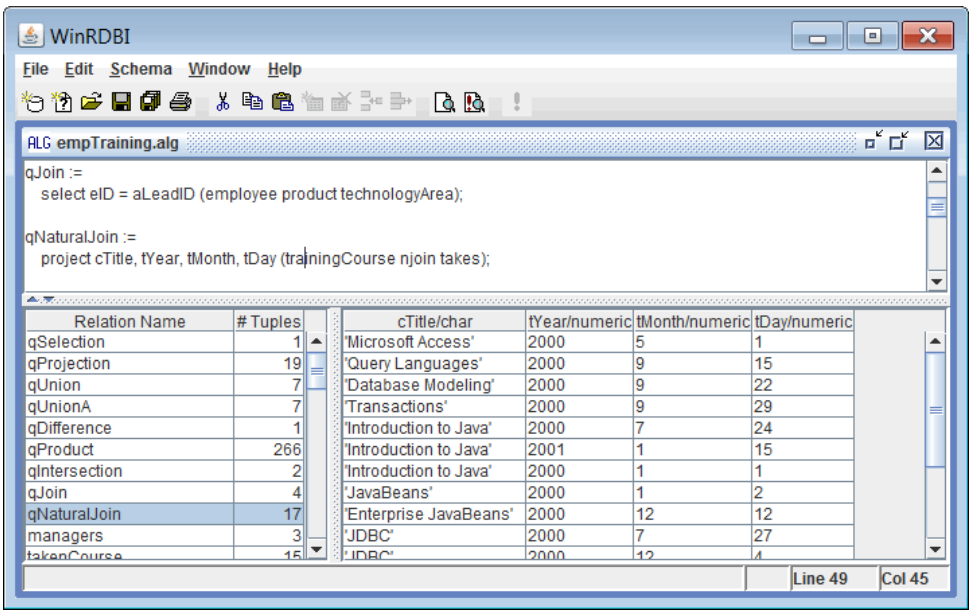

Figure 2.1: WinRDBI Interface

To alter the definition of a relations there is a property window that will let you configure the relation attributes and their types. Same properties window is used to create new relations. The first version supported adding and removing tuples to a relation from the main menus and the latest release allows direct editing of the grid. One of the nice features is that you can define multiple relations using RA and use each other as operands in other queries. Although this is one of the options it does not support storing these queries as views and you need to include the definitions of all the referenced relation in the same query file. Note that this also limits the relations from being part of the Database Browser and they can not be reused in an SQL query. Another useful feature is the ability to import relations from existing databases using ODBC.

2.6.2 iDFQL

Designed and developed by Appel et al[2]. iDFQL is an interactive environment where you can build RA queries by constructing a diagram that represents relations and operators using a data flow approach. It doesn't store a data model but instead it connects to a database on which it executes SQL queries to evaluate the RA expression defined in the diagram. The only way of getting data out of the database is by means of a query and there is no support for editing the model. iDFQL facilitate the learning of the relational algebra by showing intermediate results. This is achieved by requesting the result at any node on the diagram.

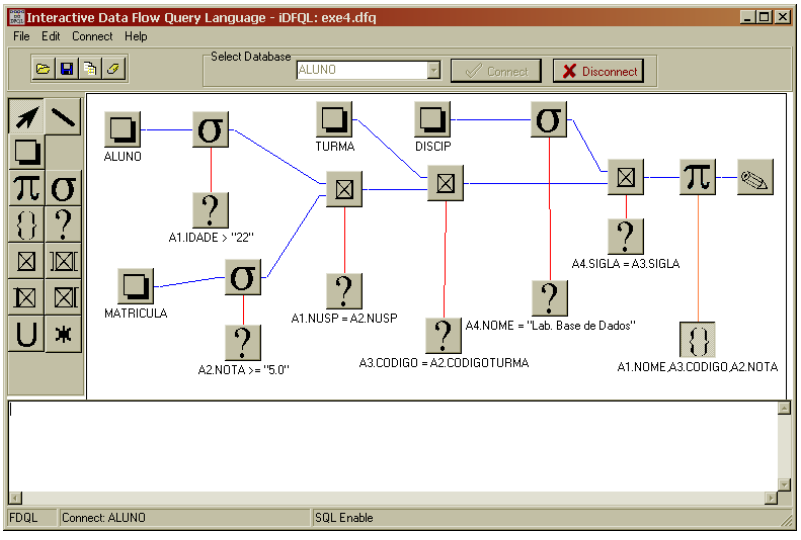

Figure 2.2: iDFQL Interface

2.6.3 RELATIONAL

RELATIONAL[15] is a Python application which serves as an educational tool for relational algebra which is available in distributions for Linux, Microsoft Windows and Mac OS X. It provides a simple and easy to navigate graphical interface which allows the user to load/store relations, execute queries on them and show the result as shown in Figure 2.3. Relations are loaded in memory from flat files prior to query execution and can not be edited within the application. The queries are specified in a one line text field using algebraic notation. The operator symbols can not directly be written by the user, instead there is a toolbox with the available operators that will add the corresponding Unicode symbol to the text field. The set of available operators in RELATIONAL extends to advanced operators such as *left outer join*. Also it provides an option that will apply a simple optimisation by moving the select operations as close to the basic relations as possible. What we found to be a major drawback is the fact that when a query execution fails there is no information for the possible errors and always the same error message "Check your query" is displayed which is not helpful in identifying the source of error.

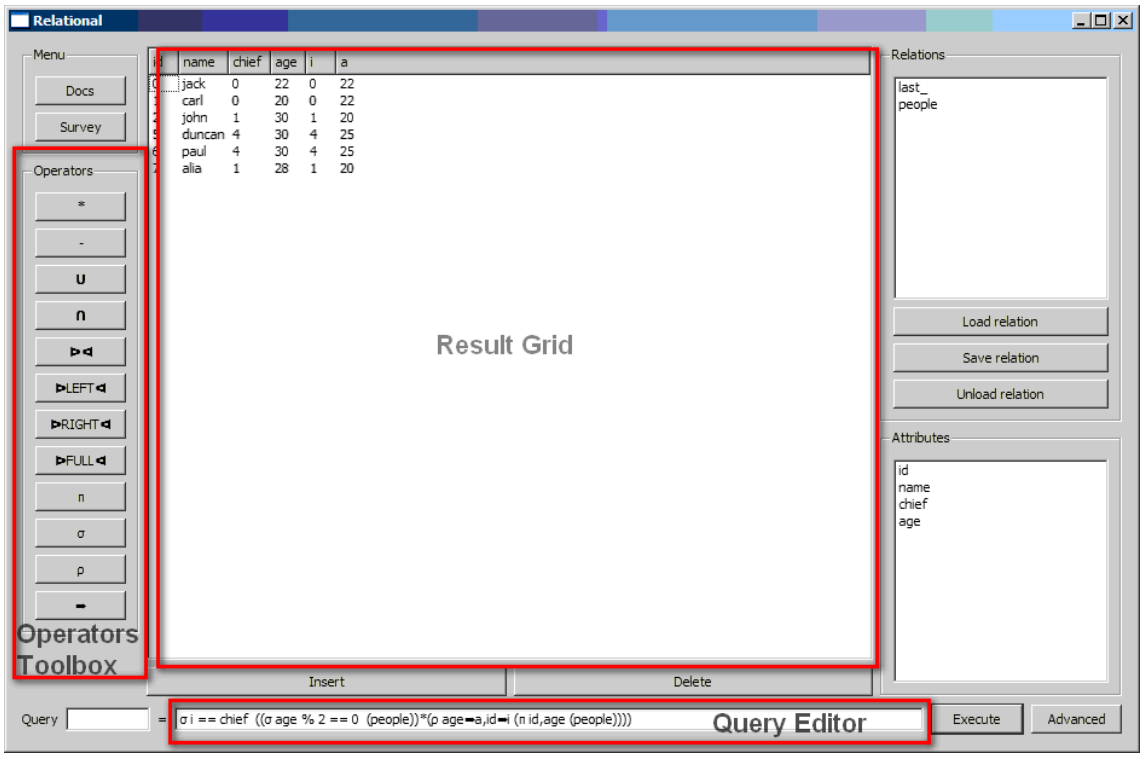

Figure 2.3: RELATIONAL Interface (Annotated)

2.6.4 RALT

RALT - Relational Algebra Learning Tool [11] is a Java application developed by Pritam Mitra which uses an interactive graphical editor for query construction. RALT has an extended set of available operators like RELATIONAL with the addition of *division*. The purpose of RALT is to serve as a query visualiser and does not support database editing. There is a panel on the left side of the workspace containing relational algebra operators which can be dragged on to the editor and assist the user in specifying the parameters and the inputs to the operator. This is a defensive approach which eliminates the possible syntax errors that arise when writing queries. When the required parameters have been correctly specified the result of the operation is drawn in the editor using a table. This way the user will have immediate feedback for every added operator before building th complete query. The query construction is simple and effective but becomes complex for large queries. Also making changes to the query diagram often requires rebuilding large portions of the query again which is time consuming. The unique feature of RALT compared to the other tools is that it shows data lineage. By selecting a row in the result the rows in all the previous nodes that contributed are highlighted to indicate why the selection is part of the result.

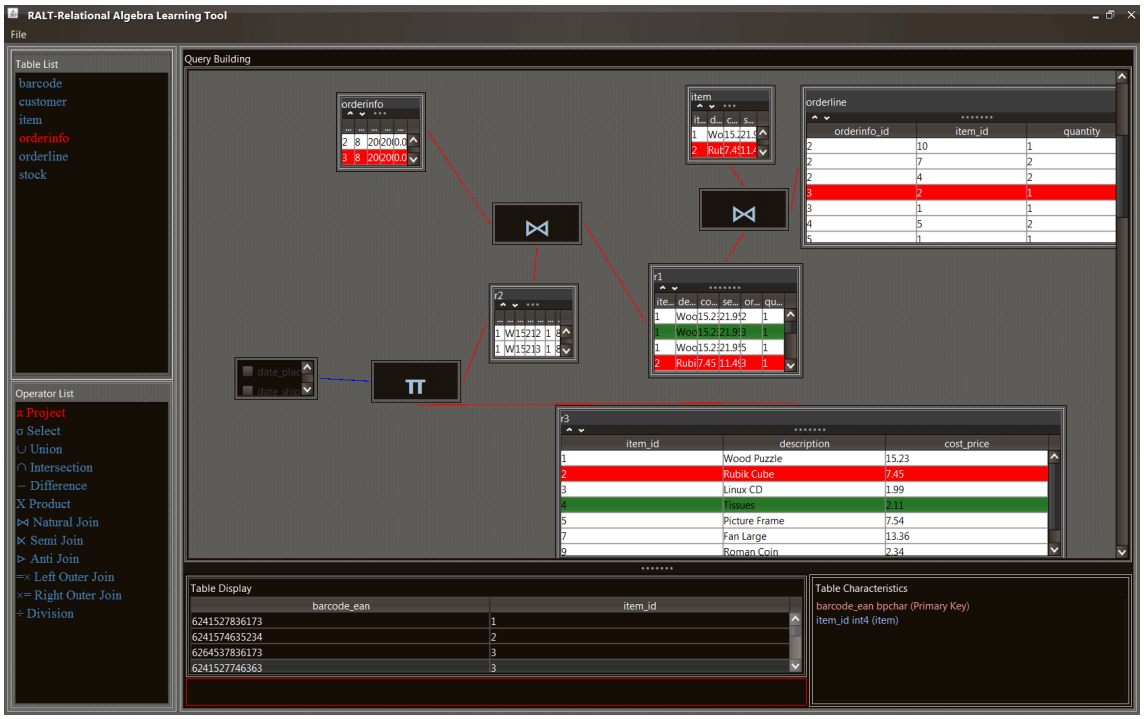

Figure 2.4: RALT Interface

2.6.5 Aqua Data Studio

A commercial Query Analyser [9] which has many useful features like most commercial tools. Unlike the previous two tools it doesn't support relational algebra but it has features that relational algebra tools lack. These are a few of the useful features:

- Grid Editing of Tables
- Views
- Execution Plan Tree
- Syntax Highlighting
- Auto-completion
- Query Export
- Query Compare Tool
- Chart Results

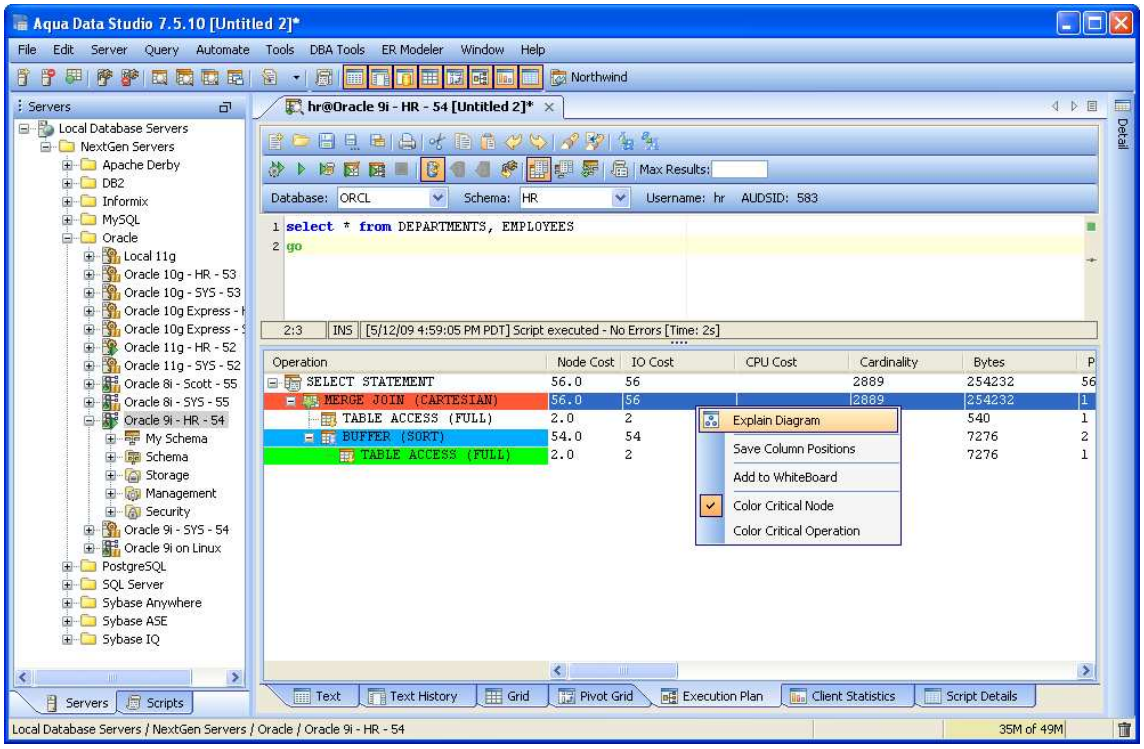

Figure 2.5: Aqua Data Studio Interface

2.6.6 Features Comparison

Based on the characteristics of the described query analysers their features are outlined in Table 2.3 along with what our implementation inherits from them. Graphical query editors, such as the one that iDFQL and RALT, clearly indicate the interme-

diate results of the evaluation. This is an important feature for a learning tool which the other non graphical editors do not provide. From another perspective we notice that the graphical editors are much harder to navigate and constructing complex queries is not practical mainly because they do not feet in one screen. Also there are some consistency issues when the user is trying to make changes. For example when deleting an operator, most of the times the whole query has to be reconstructed from the beginning.

Our attempt is to fill the gap between the two types of analysers by maintaining the same transparency to the evaluation process as the graphical editors but instead of graphical queries we will use textural queries which will help in constructing complex expressions. The user has the option to visualise the evaluation process in a tree structure composed with the operation nodes where each one provides a view of the result of the operation and cost information in terms of memory and time usage. Although the systems that support relational algebra make use of derived operators none of them shows to the user how the derived operator such as *natural join* is constructed using the basic operators. As a result there is no clear indication of the difference between the two. In our implementation all the derived operators can be specified by the user using an equivalent relational algebra expression with only basic operators. This will allow the user to introduce new operators and at the same time it will provide the option to convert a derived operator into a tree of basic operators when navigating the evaluation result.

Feature	WinRDBI	iDFQL	RELATIONAL	RALT	AquaDataStudio	**QV**
SQL Support	✓	✗	✗	✗	✓	✓
RA Support	✓	✓	✓	✓	✗	✓
Schema Editing	✓	✗	✗	✗	✓	✓
Data Editing	✓	✗	✗	✗	✓	✓
Graphical Query Editor	✗	✓	✗	✓	✗	✗
Views	✗	✗	✓	✗	✓	✓
Intermediate Results	✗	✓	✗	✓	✗	✓
Syntax Highlighting	✗	✗	✗	✓	✓	✓
Side-By-Side Query Comparison	✗	✗	✗	✗	✓	✓
Data Lineage	✗	✗	✗	✓	✗	✓
Dynamic Operators	✗	✗	✗	✗	✗	✓
Derived Operators break down	✗	✗	✗	✗	✗	✓

Table 2.3: Query Analysers Features Comparison

Chapter 3

System Architecture

Before going into specific modules of the system we provide an overall system architecture which describes what are the main tasks of the system and how they interact. The dependencies of each module are show in Figure 3.1. We follow the Model View Control (MVC) pattern with the aim to make future changes easier. The syntactical characteristics of a language should be independent from the semantics therefore we split the process into parsing and evaluation which have no dependency with each other.

3.1 View Layer

This layer is the graphical interface that the user interacts with. It consists of a query editor, a database browser and a results panel. For extensibility purposes no other module should rely on this. This will make future interface improvements much easier since the back-end of the system will not require any changes if we only need to fix graphical bugs or improve interface aesthetics.

3.2 Controlling Layer

All requests from the user will be handled by the respective controlling modules in this layer.

Query Controller

The query controller is responsible for handling query evaluation requests. It should call the respective modules in the lower layers in order to produce results. First it will call the *Parsing module* that will generate a *query tree* and then it will request an evaluation of the tree against the data model from the *Evaluation module*.

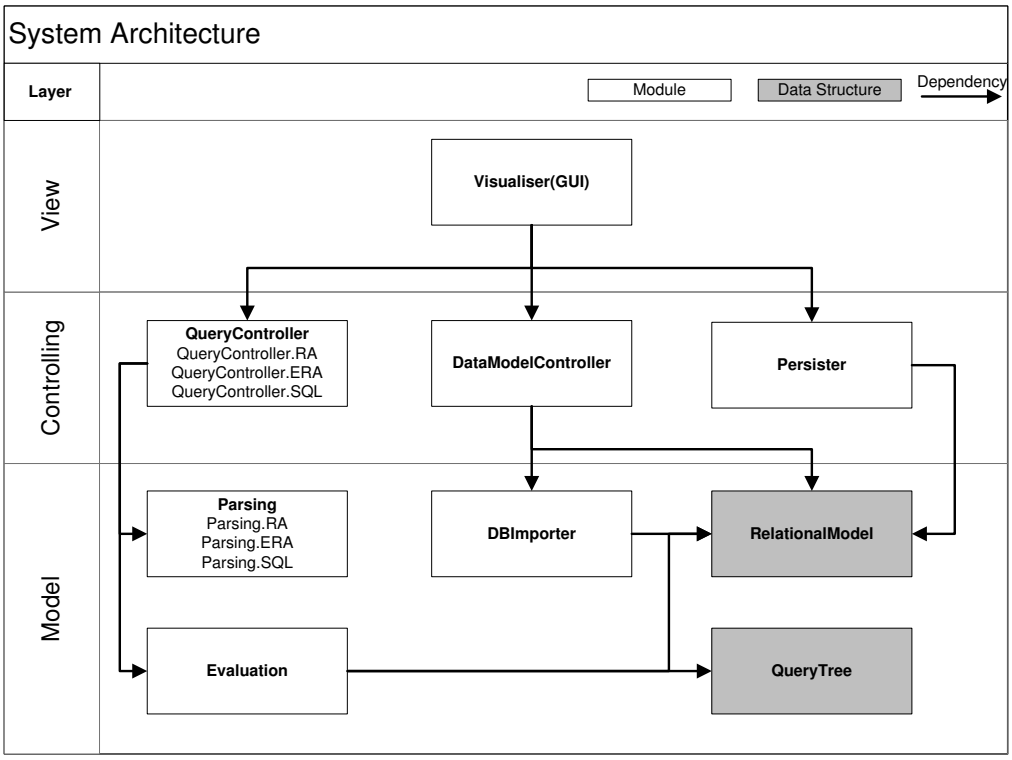

Figure 3.1: QV Architecture

Data Model Controller

The Data Model Controller is responsible for making changes to the data model as requested by the user in the *View* layer. Requests include database schema, data and stored queries (views) changes.

Persister

This module is responsible for storing the state of the workspace on stable storage. This should include the active data model and queries. It should support reloading of the workspace in the exact state that it was before the last save request.

3.3 Model Layer

Parsing

The module is responsible for parsing queries composed in RA, ERA or SQL. It should produce a query tree as a result.

Evaluation

The Evaluation module is responsible for evaluating a query tree on a data model. It assumes that the parsing phase has been successfully completed.

Relational Data Model

The Relational Data Model described here holds the data that the system will use to evaluate queries. The Data Model consist of a set of Relations and each relation holds a set of Attributes(Header) and a set of Rows(body) that satisfies the attribute conditions of the header. See 2.1 for more details.

Query Tree

Represents the structure used to host queries after parsing. It should have appropriate structure to host the intermediate result and data lineage information of each node that the *Evaluation Module* will create when traversing the tree.

DB Importer

This module makes a connection to a database server and using queries on the server reconstructs a copy of the database using thes *Relational Model* structure.

Chapter 4

Query Representation and Execution

At the core of the system is the query execution process which take as input a textural query (composed in RA, ERA or SQL) and populate a result by evaluating the query against the active data model. The process is split into parsing and evaluation. In this chapter we provide a complete design for both phases along with specification for a simple optimisation technique on query trees.

4.1 Query Syntax

In this section we cover the query syntax for RA, ERA and SQL which the parser implements. We will specify a suitable syntax for the RA and a minimalistic syntax for the SQL Select clause using *Extended Backus-Naur Form* **(EBNF)** [1].

4.1.1 RA Syntax

The basic syntax elements are described here plus an additional entry for the dynamic operators which can only be binary. The dynamic operators are derived operators that the user can define.

⟨*inner operation*⟩ ::= <relation> | ⟨*unary operation*⟩ | ⟨*binary operation*⟩;
⟨*operation*⟩ ::= '(' <inner operation>')';
⟨*unary operation*⟩ ::= ⟨*project*⟩ | ⟨*select*⟩ | ⟨*rename*⟩;
⟨*binary operation*⟩ ::= ⟨*product*⟩ | ⟨*union*⟩ | ⟨*difference*⟩ |⟨*dynamic operation*⟩;
⟨*relation*⟩ ::= (letter)+;

[1] See [16] for EBNF notation details.

⟨*project*⟩ ::= 'PROJECT' '[' ⟨*attribute list*⟩ ']' ⟨*operation*⟩;
⟨*select*⟩ ::= 'SELECT' '[' ⟨*boolean condition*⟩ ']' ⟨*operation*⟩;
⟨*rename*⟩ ::= 'RENAME' '[' (letter)+ '/' ⟨*attribute*⟩ ']' ⟨*operation*⟩;
⟨*product*⟩ ::= ⟨*operation*⟩ 'PRODUCT' ⟨*operation*⟩;
⟨*union*⟩ ::= ⟨*operation*⟩ 'UNION' ⟨*operation*⟩;
⟨*difference*⟩ ::= ⟨*operation*⟩ 'DIFFERENCE' ⟨*operation*⟩;
⟨*dynamic operation*⟩ ::= ⟨*operation*⟩ (letter)+ (⟨*boolean condition*⟩)? ⟨*operation*⟩;
⟨*attribute list*⟩ ::= ⟨*attribute*⟩ (',' ⟨*attribute*⟩)*;
⟨*attribute*⟩ ::= (letter)+ ('.' (letter)+)?;
⟨*boolean condition*⟩ ::= (⟨*boolean condition*⟩ 'OR')? ⟨*and condition*⟩;
⟨*and condition*⟩ ::= (⟨*boolean condition*⟩ 'AND')? ('NOT')? ⟨*single condition*⟩;
⟨*single condition*⟩ ::= ⟨*boolean condition*⟩ | ⟨*comparison*⟩ | ⟨*attribute*⟩;
⟨*comparison*⟩ ::= ⟨*attribute*⟩ | ⟨*string*⟩ | ⟨*signed integer*⟩ | ⟨*float*⟩;
⟨*string*⟩ ::= "' (alphanumeric)+ '";
⟨*signed integer*⟩ ::= ['+' | '-'] ⟨*integer*⟩;
⟨*integer*⟩ ::= (digit)+;
⟨*float*⟩ ::= ⟨*signed integer*⟩ '.' ⟨*integer*⟩;

4.1.2 ERA Syntax

The ERA syntax contains all the elements of the RA syntax with the following modifications/additions.

⟨*unary operation*⟩ ::= ⟨*project*⟩ | ⟨*select*⟩ | ⟨*rename*⟩ | ⟨*group*⟩ | ⟨*distinct*⟩ | ⟨*aggregate*⟩;
⟨*project*⟩ ::= 'PROJECT' '[' ⟨*attribute expression list*⟩ ']' ⟨*operation*⟩;
⟨*rename*⟩ ::= 'DISTINCT' ⟨*operation*⟩;
⟨*group*⟩ ::= 'GROUPBY' '[⟨*grouping attributes*⟩]' ⟨*operation*⟩;
⟨*attribute expression list*⟩ ::= ⟨*attribute expression*⟩ (',' ⟨*attribute expression*⟩)*;
⟨*attribute expression*⟩ ::= ⟨*attribute*⟩ | ⟨*attribute arithmetic expression*⟩
⟨*attribute arithmetic expression*⟩ ::= ((⟨*float*⟩ | ⟨*signed integer*⟩) ('+'|'-'|'*'|'/'))* ⟨*attribute*⟩ ((⟨*float*⟩ | ⟨*signed integer*⟩) ('+'|'-'|'*'|'/'))*
⟨*aggregate*⟩ ::= ⟨*aggregate function*⟩ ⟨*operation*⟩;
⟨*grouping attributes*⟩ ::= (⟨*attribute*⟩ | ⟨*aggregate function*⟩) (',' (⟨*attribute*⟩ | ⟨*aggregate function*⟩))*;
⟨*aggregate function*⟩ ::= ('MAX' | 'MIN' | 'COUNT' | 'SUM') '[' ⟨*attribute*⟩ ']'

4.1.3 SQL Syntax

⟨*select expression*⟩ ::= 'SELECT' ['DISTINCT' | 'ALL'] ⟨*column expr*⟩* 'FROM' ⟨*from expr*⟩ ['WHERE' ⟨*where expr*⟩] ['GROUP BY' ⟨*attribute list*⟩*] ['HAVING' ⟨*condition*⟩] ['ORDER BY' ⟨*orderby expr*⟩]

⟨*column expr*⟩ ::= ⟨*ident*⟩ ['AS'] [⟨*ident*⟩]

⟨*from expr*⟩ ::= ⟨*table*⟩ [',' ⟨*from expr*⟩]

⟨*from expr*⟩ ::= ⟨*table*⟩ [INNER | [[LEFT|RIGHT|FULL] [OUTER]] | NATURAL] JOIN ⟨*table*⟩ [ON <condition>]

⟨*orderby expr*⟩ ::= ⟨*ident*⟩ ['ASC' | 'DESC'] [',' ⟨*ident*⟩ ['ASC' | 'DESC']]*

4.1.4 Syntax Errors

The parser will identify syntactical errors and report them. Where applicable it will provide position information for the error in the query.

4.2 Query Tree

When the input is validated and there are no syntax errors the parser will construct a Query Tree that will be used in the evaluation process. The result tree contain only three types of nodes: *unary operators* , *binary operators* and *relations* (leaf) nodes.
For example the the following RA query translates to the tree in Figure 4.2:
$PROJECT[R.A, S.A, S.B](R\ PRODUCT(SELECT[S.B > S.C]S))$

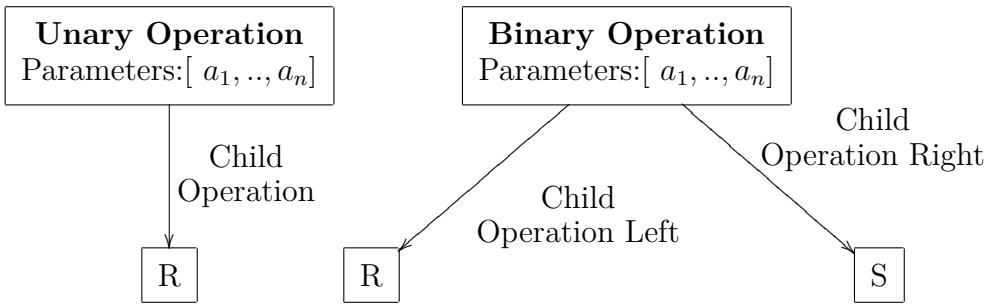

Figure 4.1: Unary and Binary Abstract Operation Trees

When translating a RA or ERA query to a tree we will use operations (nodes) that arise in the syntax of the respective language. In the case of SQL we will translate it to an ERA tree that is similar to the low-level operations that most RDBMS use. By doing so we can later display this ERA tree to the user in order to provide an insight on how SQL is translated and executed. At the parsing stage

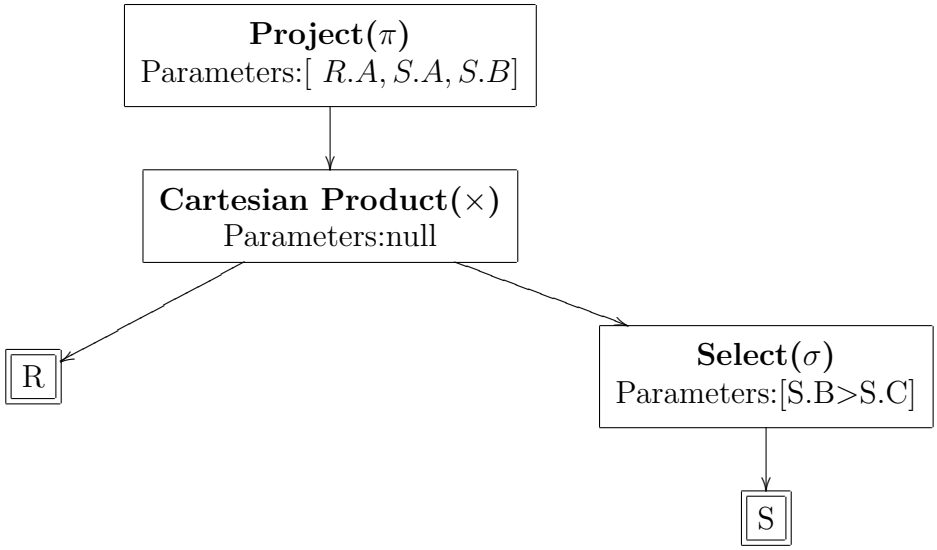

Figure 4.2: Operation Tree Example

we don't check if the query is referencing valid attributes or relations since the evaluation engine will be responsible for identifying this at the next stage.

4.3 Query Evaluation

The evaluation engine consist of specialist evaluators and a general evaluation module (entry point) which take as input a Query Tree and a Relational Data Model and evaluates the tree on the model in order to return a result (table).

4.3.1 General Evaluation Module

Determines the type of the tree *head node* (e.g. Project operator) and calls the relevant specialist module to evaluate it.

4.3.2 Specialist Evaluation Modules

Each of the following specialist modules will evaluate a query tree of the respective type.

Relation Evaluator: Using the data model it retrieves and returns the table with the specified name from the database.
Project Evaluator: Requests evaluation for child sub-tree by calling the general evaluator. From the result it removes attributes that are not included the list of

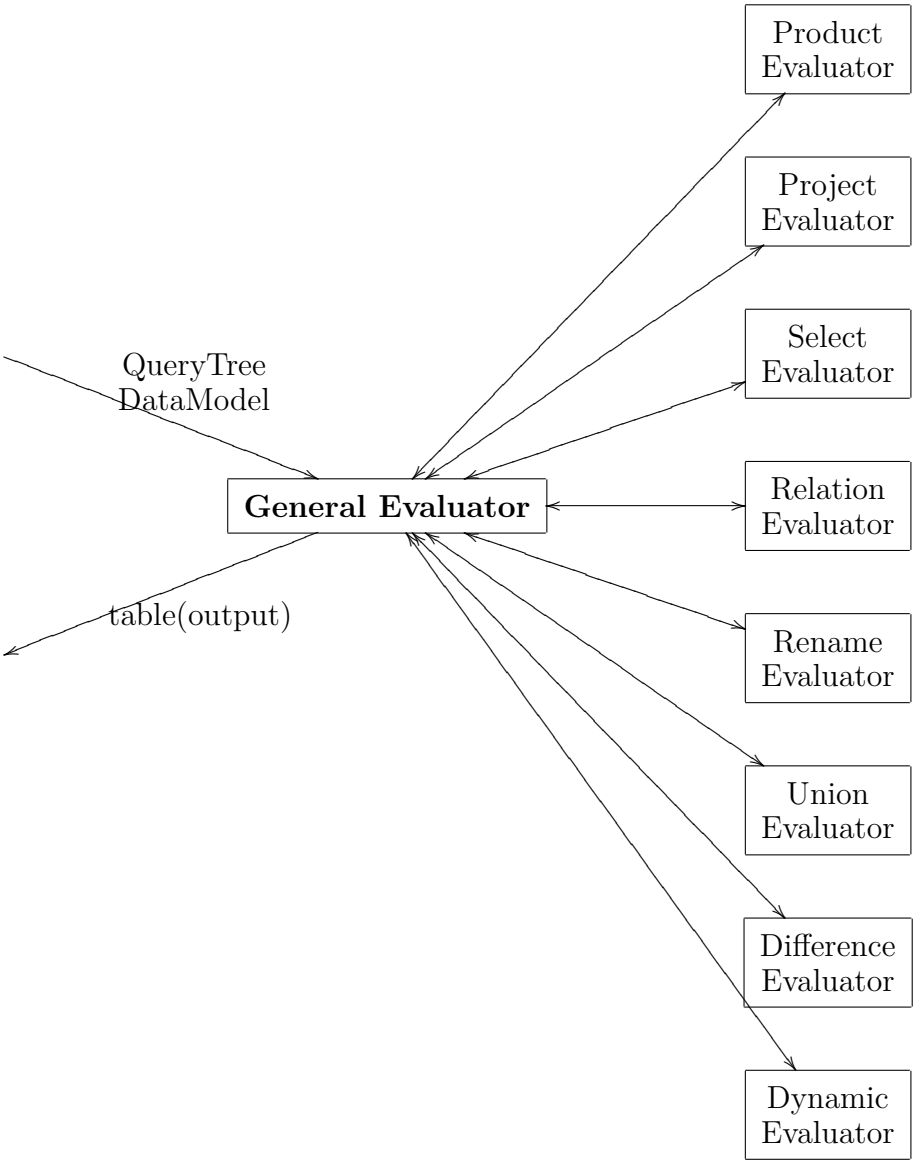

Figure 4.3: Evaluators Collaboration Diagram

attributes (parameter).
Select Evaluator: Requests evaluation for child sub-tree. From the result it removes rows that do not satisfy the selection condition(parameter).
Rename Evaluator: Requests evaluation for child sub-tree and renames the attribute specified in the parameter with the new name.
Cartesian Product Evaluator: Requests evaluation for child sub-trees and creates a new table with all attributes from both results. It then adds rows by combining all possible pairs from the two results(i.e. $R \times S$).
Union Evaluator: Requests evaluation for child sub-trees. Checks that their attributes are compatible and returns a new table by including all rows from both sub-trees.
Difference Evaluator: Request evaluation for child sub-trees. Checks that their attributes are compatible and returns a new table by including all rows from the left sub-tree and deleting any rows that occur in the second sub-tree too.
Dynamic Evaluator: It finds the definition of the operator in a repository. It parses the definition which results in a new tree which it then requests a new complete evaluation for. In other words it will convert a derived operation into a tree of operations as specified by the definition of the derived operator. We illustrate this with the case of Natural Join:

```
PROJECT[#REF1 UNION #REF2] (
        SELECT[#REF1.$=#REF2.$] (
                #REF1 PRODUCT #REF2
        )
);
```

This definition states that the result is the projection of all unique attributes of the two relations (#REF1 UNION #REF2) and the body of the result contains all rows that agree on all common attributes (#REF1.$=#REF2.$). Assume this definition is stored in the set of derived operators under the name 'NATURALJOIN' and the query "S NATURAL JOIN R" is being evaluated. The parser will initially parse this and create the tree in Figure 4.4. When the evaluation begins the Dynamic Evaluator will transform this into the tree in Figure 4.5 where the selection will match all common attributes and the projection includes all unique attributes of both R and S.

4.3.3 Evaluation Errors

Evaluation errors arise when the query tree cannot be evaluated against the data model because of missing relations or attributes. The Project, Select and Rename evaluators should check for missing attributes and report them. the missing relation

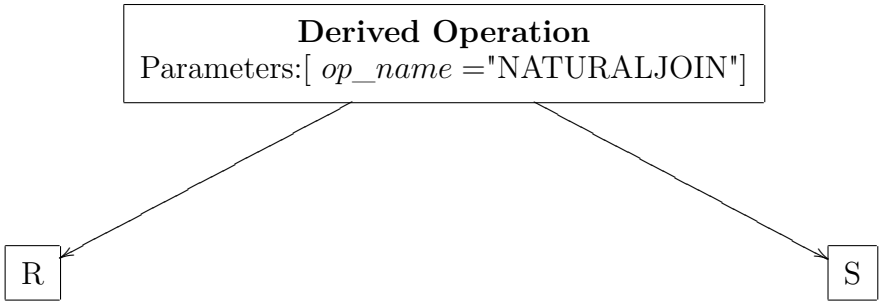

Figure 4.4: Natural join tree after initial parsing

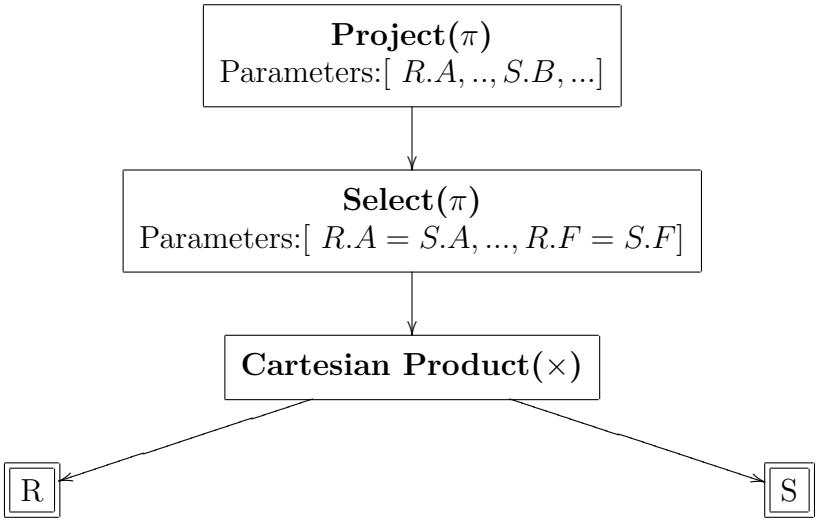

Figure 4.5: Natural join tree after expansion to original definition

case will only need to be tracked in the Relation Evaluator. In addition the Union and Difference evaluators should check attribute compatibility of the two sub-tree results (i.e. check that both sub-trees contain the same attributes). These errors are used by the user interface that gives visual feedback to the user.

4.3.4 Views

Queries composed by the user can be stored in the Relational Data Model as Views. Each view is assigned a name and it can be referenced in a different query as any other base relation of the Data Model. To make the system as generic as possible both SQL and RA views can be stored and queries can reference views of different type, for example we can reference an SQL view in a RA query. The evaluation engine will retrieve the query of the referenced view and execute it. After that it will treat it as a base relation.

4.3.5 Execution Statistics

We can derive execution time and memory usage statistics by capturing the time spent evaluating each node and the average row size along with the number of rows of the result. We demonstrate this with an example. Evaluation of the query tree in Figure 4.2 will update the tree so that it includes time and memory measures for each node as in Figure 4.6. The statistics and data lineage (Section 4.3.6) are not data that would occur in a normal execution therefore are not considered in the statistical calculations.

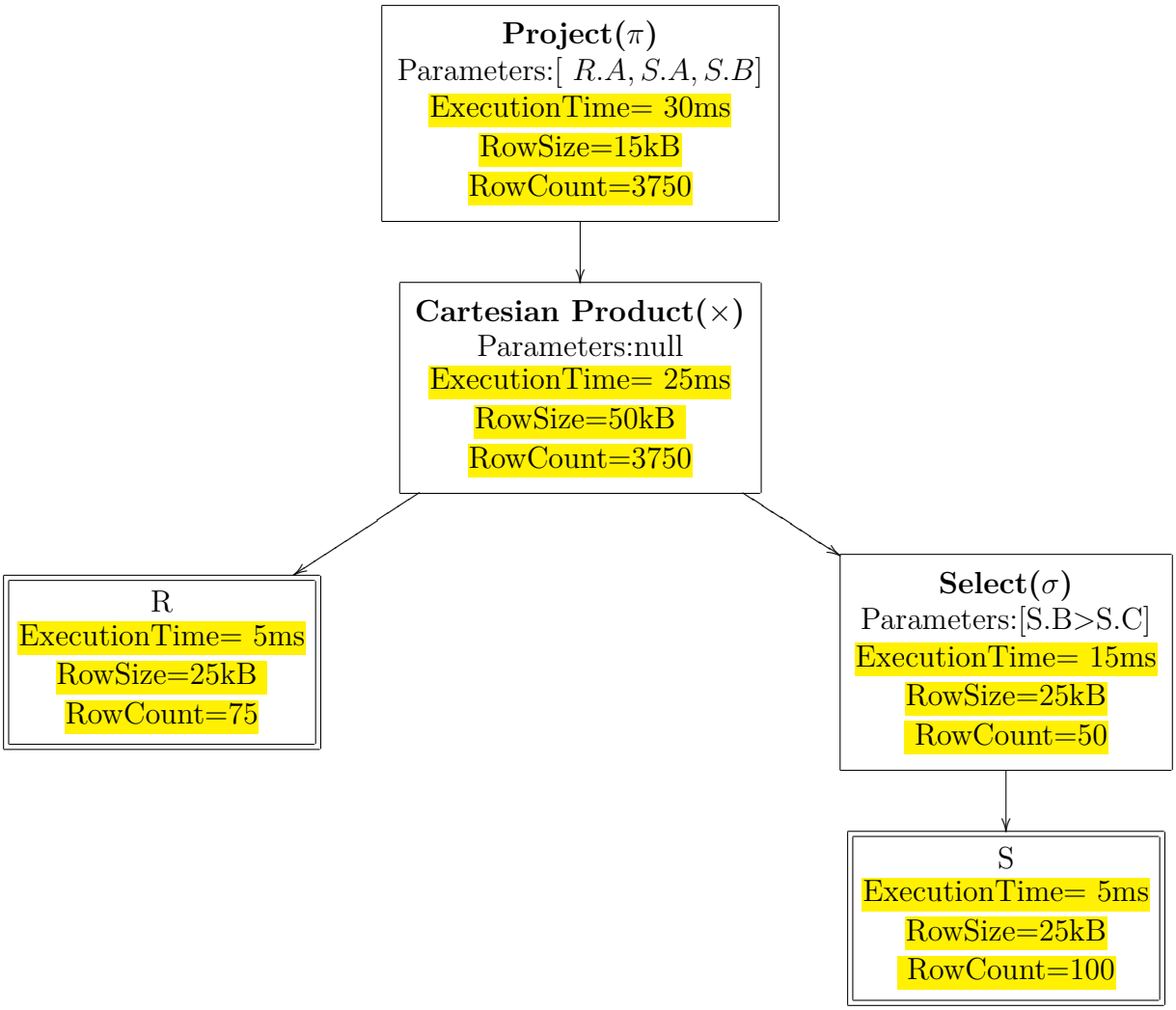

Figure 4.6: Operation Tree Example with statistics

Execution Time

From the time span logged at every node we can derive the following:

- *Overall Execution Time*: Execution Time of Head node.
- *Sub Tree Execution Time*: Execution Time of Head node of sub-tree.
- *Operation Contribution Time*: Execution Time of operation node minus Execution Time of children.

In the case of our example we can deduce the contribution of each operation performed:

- *Load relation S Contribution Time* $= 5ms = 5ms - (0ms)$
- *Load relation R Contribution Time* $= 5ms = 5ms - (0ms)$
- *Select Contribution Time* $= 10ms = 15ms - (5ms)$
- *Cartesian Product Contribution Time* $= 5ms = 25ms - (5ms + 15ms)$
- *Project Operation Contribution Time* $= 5ms = 30ms - (25ms)$
- *Overall Execution Time*$= 30ms$

Memory Usage

The size of each intermediate result calculated by an operation is estimated by multiplying the number of rows with the average row size. From this we can further derive maximum memory usage during evaluation of a tree. During an operation we evaluate the child operations and keep their results until the end of the evaluation. Assuming that sub-trees can be processed in parallel we can define the maximum memory usage using Equation 4.1. This is the worst case scenario because if operations are executed sequentially, then the memory usage is less.

$$maxmemory(x) = \max\left[\left(resultsize(x) + \sum_{i=0}^{n_x} resultsize(i_x)\right), \sum_{i=0}^{n_x} maxmemory(i_x)\right] \tag{4.1}$$

where n_x is the number of children of node x and i_x represents the i-th child node of x.

- MaxMemoryUsage(Load Relation S)$= 2500kB$
  $= \max\left[(25 * 100) + (0), 0\right]$

- MaxMemoryUsage(Load Relation R)= $1875kB$
  $= \max[(25 * 75) + (0), 0]$

- MaxMemoryUsage(Selection)= $3750kB$
  $= \max[3750, 1875] = \max[(50 * 25) + (2500), 1875]$

- MaxMemoryUsage(Cartesian Product)= $190625kB$
  $= \max[190625, 5625] = \max[(50 * 3750) + (1875 + 1250), 1875 + 3750]$

- MaxMemoryUsage(Projection)= $243750kB$
  $= \max[243750, 190625] = \max[(15 * 3750) + (187500), 190625]$

4.3.6 Data Lineage

To allow the user to drill down and see from where specific rows of the result comes from we need to add some overhead to our query tree during the evaluation process. In particular we create one map for every arc in the tree which is stored on the parent node. This map will contain entries linking rows of the parent node to rows in the child node. The maps should be created based on the following rules for each operator:

Unary Operations

Each row of the operation result should map to the row of the child that it derived from.

- **Project and Rename** *One to One relation.* The number of entries should equal the number of rows.

- **Select** *One to One relation.* The number of entries should equal the number of rows that satisfy the selection condition.

- **Group** *One to Many relation.* Each row of the result maps to all rows of the child that agree on the *group by* attributes.

Binary Operations

Each row of the operation result should map to the rows of the children that it derived from(i.e. two maps, one per child (arc))

- **Cartesian Product** *Many to Many relation.* Each row has an entry in both maps with the rows that it combines.

- **Union** *One to Many relation.* Each row of the result maps to one row of a child or one row of each child.

- **Difference** *One to One relation.* Each row of the result maps to one row of the first child. An additional map is created indicating rows that have not been included because they were part of both the left and right child(implicit contribution to result).

We don't need to specify rules for derived operators since they are always evaluated using the basic operators and the lineage rules are therefore always inherited.

4.4 Algebra Optimisation

We will specify a systematic way of improving execution performance by simple tree modifications which do not affect the end result of the query. We focus on moving selections to the lowest possible level of the query tree in order for them to be evaluated early and limit the size of the intermediate result that parent nodes need to work on. There are two general cases that arise. The first one is when the input operation (child node) of the selection is a unary operation and when it is a binary one. Implementing the following simple rules will yield an efficient selection strategy. Here we need to acknowledge the fact that some of these optimisations will not always improve the memory usage and a more sophisticated approach using database statistics can be employed to tackle this issue. Such an approach is out of the scope of the project although we design a platform that is capable of hosting such extensions.

4.4.1 Unary Input Case

In general if all the referenced attributes in the selection condition are part of the heading[2] of the child's input then it is safe to switch the select operation with its child. To avoid the computation of result headings and improve performance during optimization we can take a simpler approach. If the child operation is *project* then we move the selection as the child. If the operation is *rename* and the renamed attribute is not referenced in the select condition we move the selection as the child. If the child operation is another selection we combine the two operations into one using the boolean *and* connective. This will eliminate the redundant iterations.

[2]Set of attributes

4.4.2 Binary Input Case

We split the select condition based on the boolean *and* operators and we derive a set of sub-expression. We then decide for each such expression whether all its referenced attributes are part of the heading of only one of the children of the immediate binary operator. If so, we add a selection operation above that child. The sub-expressions that could not be matched to only one child are joined again with the boolean *and* connective and placed back in the original selection.

Chapter 5

Workspace and Visualisers

In this chapter we define the graphical interface of the system. We go through the various features of the QV workspace shown in Figure 5.1. The implementation details of the system are described in Chapter 6.

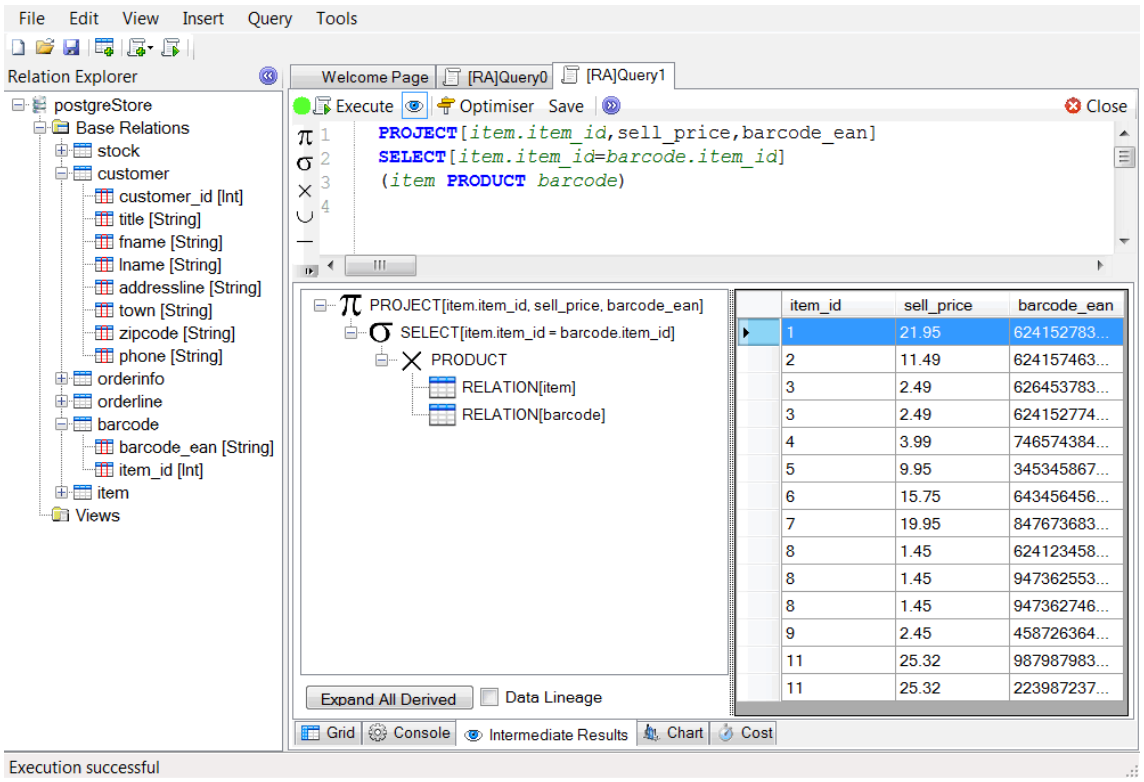

Figure 5.1: QV Interface

5.1 Relation Explorer

In order for the user to execute queries a database should be constructed first. There is no support for a Data Definition Language but schema and data editing can be achieved from the graphical interface. For this purpose like most query analysers the schema of the active database in the application is shown in a tree structure in the relation explorer on the left of the workspace. It is split into two sub-trees, *Base Relations* and *Views*. In *Base Relations* all the relations are listed with their attributes and their respective data types. The user can make changes to the database schema by adding deleting and editing relations and attributes. These options are available in the right click context menu and on the main tool bar. In order to modify the rows of a relation the user can either use the context menu "Open" option or double click on the relation name and a new window with a tabular grid will be created on which the user can make the relevant changes. When adding new relations or editing the name of an attribute we use a defensive approach to the name clashes problem by disabling the relevant options. For example if the user defines a new relation with the name "item" which already belongs to the database the modal "New Relation" dialog box will warn the user(Figure 5.3). Same approach has been followed for many cases such as creating a new view and editing the definition of an attribute (name and data type). The relation explorer panel can be collapsed using the button on the top right corner or from the *View* menu on the menu bar.

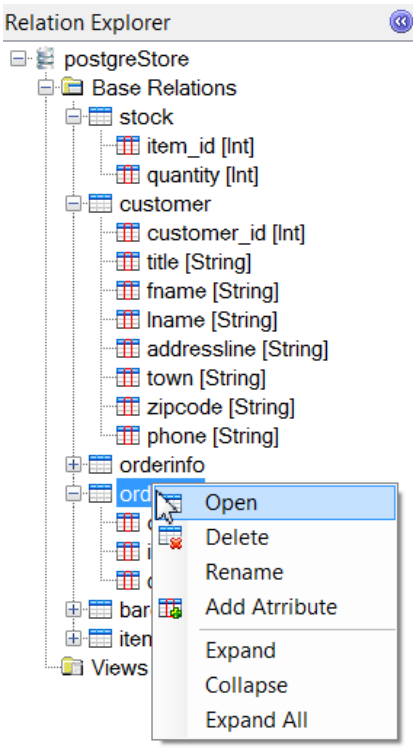

Figure 5.2: Relation Explorer

5.2 Import

The system allows data import from an existing database. Given a connection string it can replicate the schema and the data (excluding views and procedures). This should be stored into an instance of a *Data Model* that the system will be able to use in query evaluation. After the import is completed the user can use schema and data editing to make further database changes. These changes will not reflect the actual database used but they will only update the local data model.

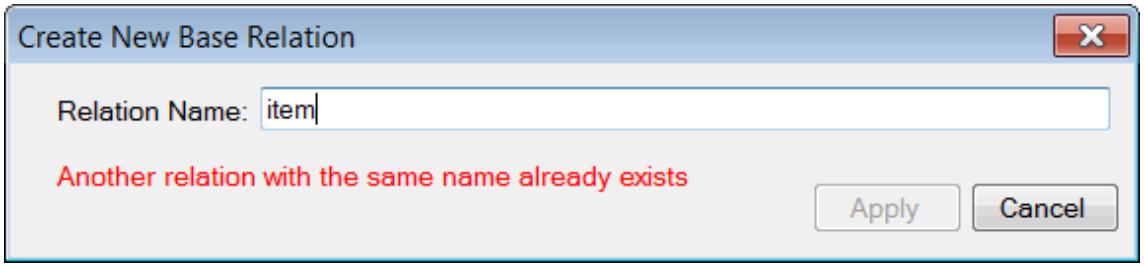

Figure 5.3: New Relation Dialog

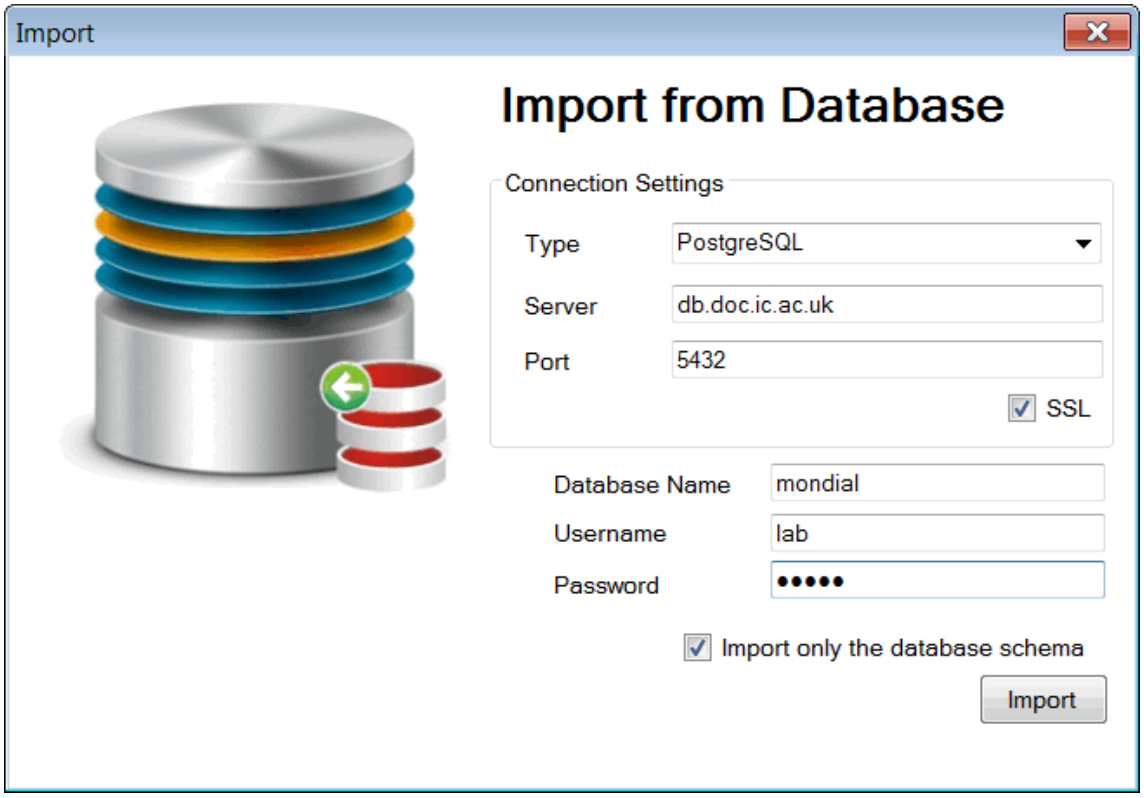

Figure 5.4: Import from database dialog

5.3 Tab Containers

To make the environment generic and customisable based on the user preferences the main area of the workspace can host multiple tab windows which can be *query editors* or *relation grid editors*. Also the user has the option to open a second tab

container to gain visual access to two windows at the same time. This is useful in cases that we want to see the difference between two queries illustrated in Figure 5.5.

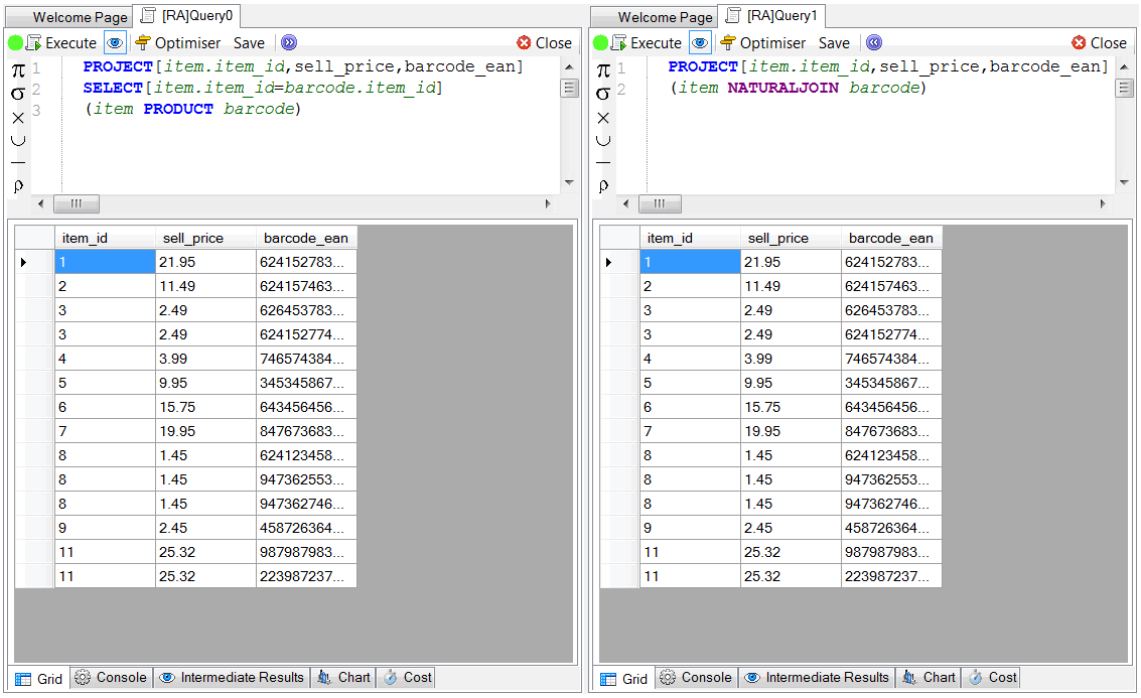

Figure 5.5: Two tab containers

5.4 Query Panel

The main component of the interface is the *query panel* which is split into two areas. The top one consists of an editor where the user specifies a query and the bottom one which displays the query results (Figure 5.6). We describe each one in detail in Sections 5.5 and 5.6 respectively. When the user creates a new *query panel* from the menu there are three options which open the panel in relational algebra, extended relational algebra or SQL mode.

5.5 Query Editor

The query editor provides appropriate highlighting to help the user spot mistakes easily. Highlighting is split into two types.

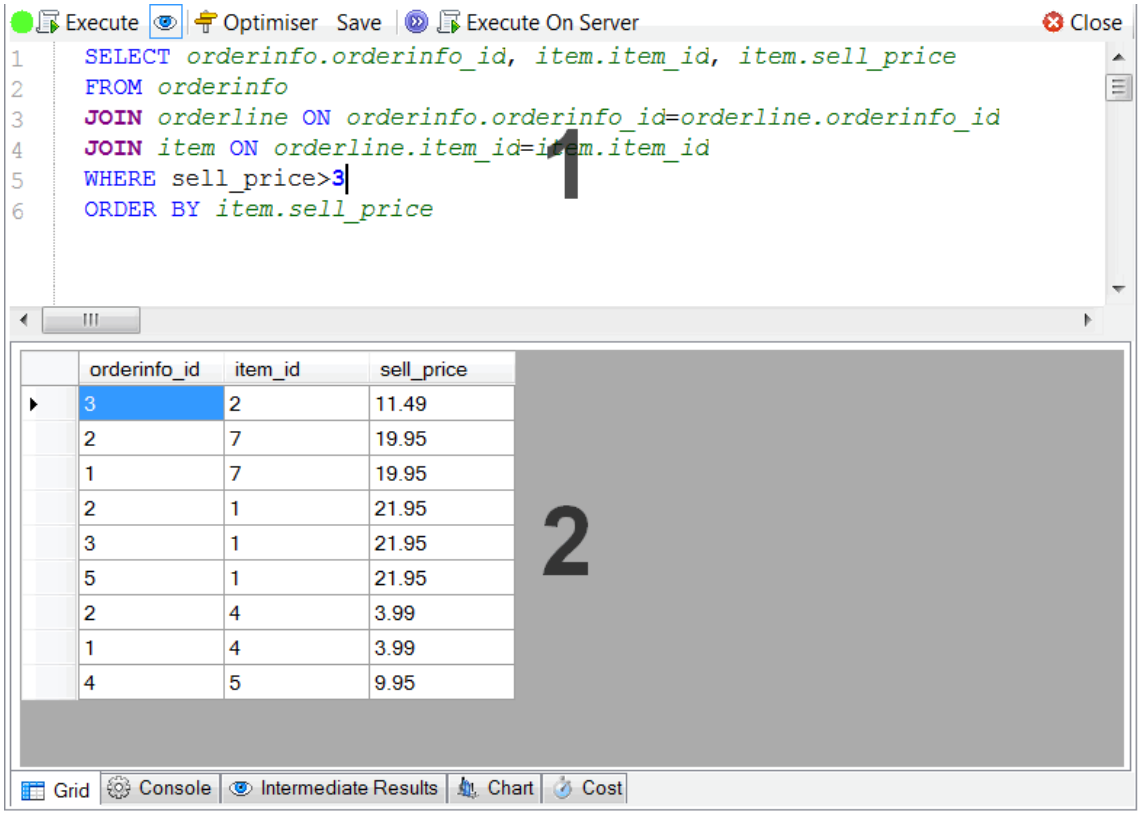

Figure 5.6: Query Panel

- **Matching Brackets Indicator**

  When the editor caret is at bracket and there is a matching opening or closing one they are both highlighted. This is a standardised way that many integrated development environments (IDEs) use and we can expect the user to be familiar with this method.

  ```
  PROJECT[item_id] (stock PRODUCT barcode)
  ```

- **Syntax Recognition**

  In a similar way we inherit the syntax highlighting principles. The syntax keywords are recognised and highlighted accordingly. When editing a query, basic operators are highlighted with blue and the derived operators in purple.

  ```
  PROJECT[item_id,sell_price]item NATURALJOIN stock
  ```

  In addition relation and attribute names of the active database are also dynamically recognised and highlighted in green. Constant values are also indicated,

strings and numeric values are highlighted in red and blue respectively.

```
SELECT[description=5 AND item_id='CD']item
```

Derived Operator Editor

Using the syntax we specified for defining derived operators in Section 4.1 we also incorporated another editor with the same highlighting mechanism as the query editor that allows the user to define new and edit existing operators. When adding such an operator the user specifies the name of the operator, it's definition and an icon to represent the operator in the results tree.

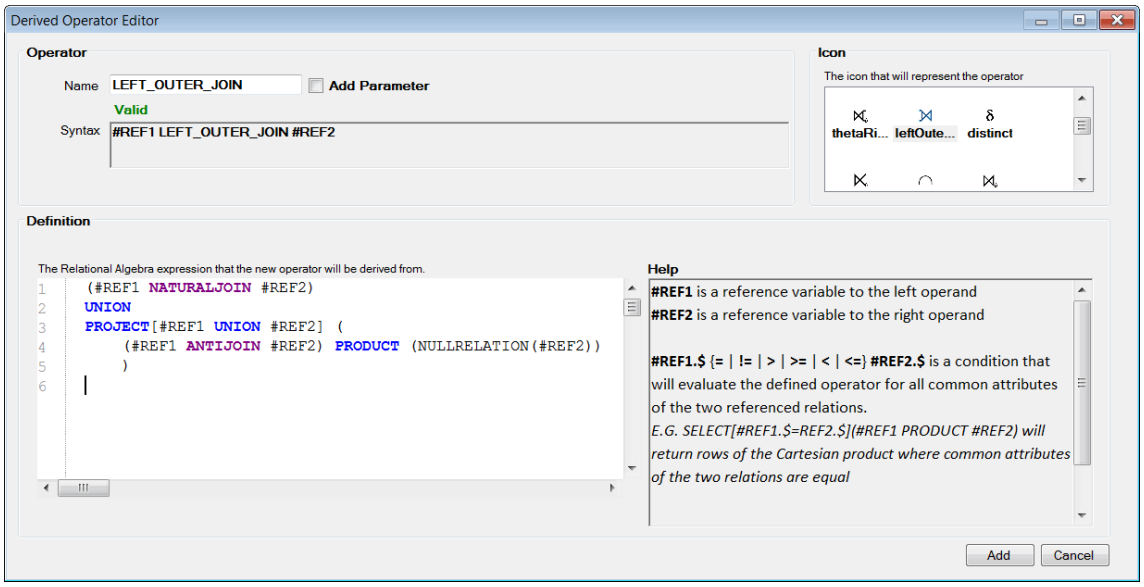

Figure 5.7: Derived Operator Editor

5.6 Result Visualisers

The result of a query evaluation are presented in the results panel below the *query editor*. There are five available options and each one serves a different purpose:

5.6.1 Data Grid

This is the simplest view which is an integral part of every query analyser. It displays the result of the query execution in a read only tabular grid. As an additional feature the rows of the grid can be sorted by clicking on the column headers.

5.6.2 Intermediate Results

As we explained earlier we need to emphasise on how we display query results to the user. This view provides an analytical description of the evaluation process using a tree structure that represents the query tree as described in Section 4.2. This tree is used as a navigation mechanism. When the user clicks on a node the intermediate result calculated at that node during evaluation is displayed on a grid on the right.

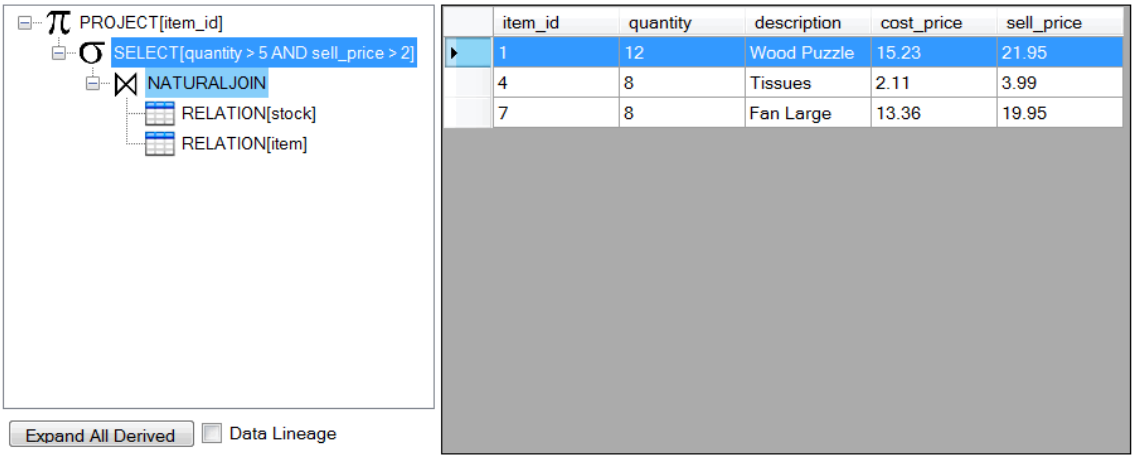

Figure 5.8: Intermediate Results View

Derived Operators Break Down

The derived operator nodes are highlighted with light blue. In the case that such an operator occurs in the tree the user can select to expand it which will transform the operator into a tree of operations based on it's definition as illustrated in Figure 5.9.

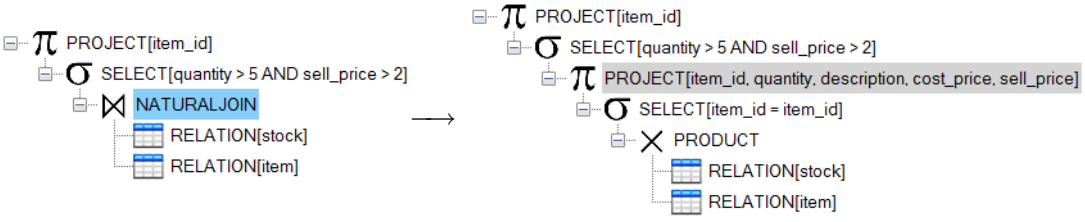

Figure 5.9: Expanding Derived Operators

Data Lineage

To provide a more detailed explanation on how the result was obtained the user can select rows in the grid of the head node and automatically all the rows directly associated in all child nodes will be highlighted. Information on how to find these rows can be derived from the mappings created during query execution in Section 4.3.6. In addition when a grouping operator is used the user has the option to open a new window that will display on a grid the different groups of rows that formed the result.

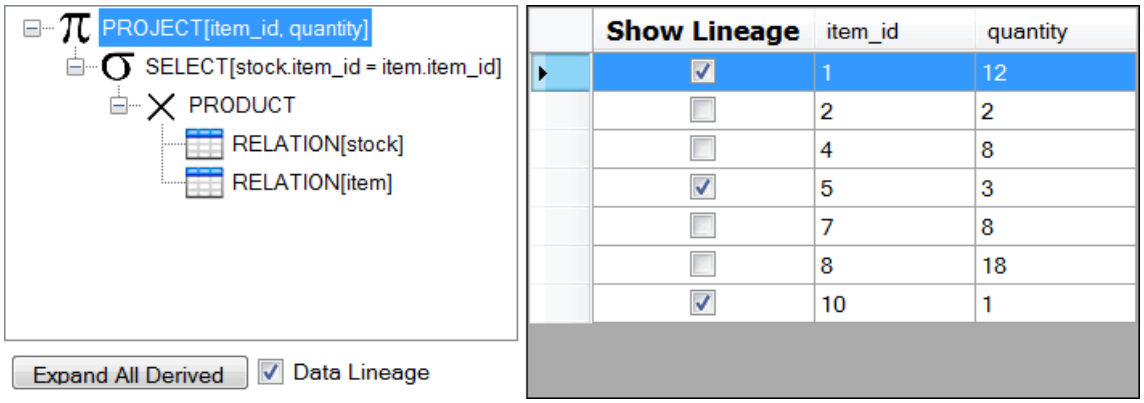

Figure 5.10: Selecting Rows for Lineage

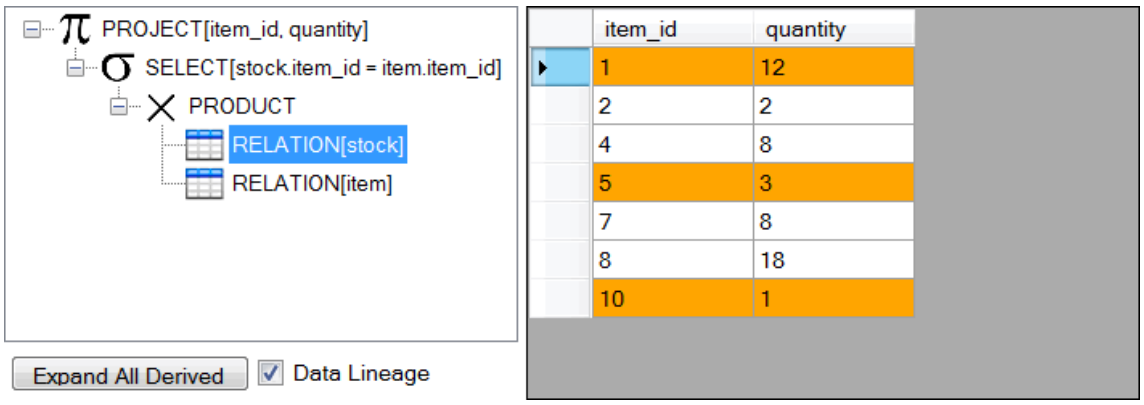

Figure 5.11: Showing Lineage at a leaf node

5.6.3 Chart

Whenever execution contains numeric values the chart view can draw interactive charts for a selection of numeric attributes grouped by a specified attribute. The chart has many options such as zooming and stretching along both axes and it can also be exported into an image in various formats. An example of a query displayed using a chart is shown in Figure 5.12.

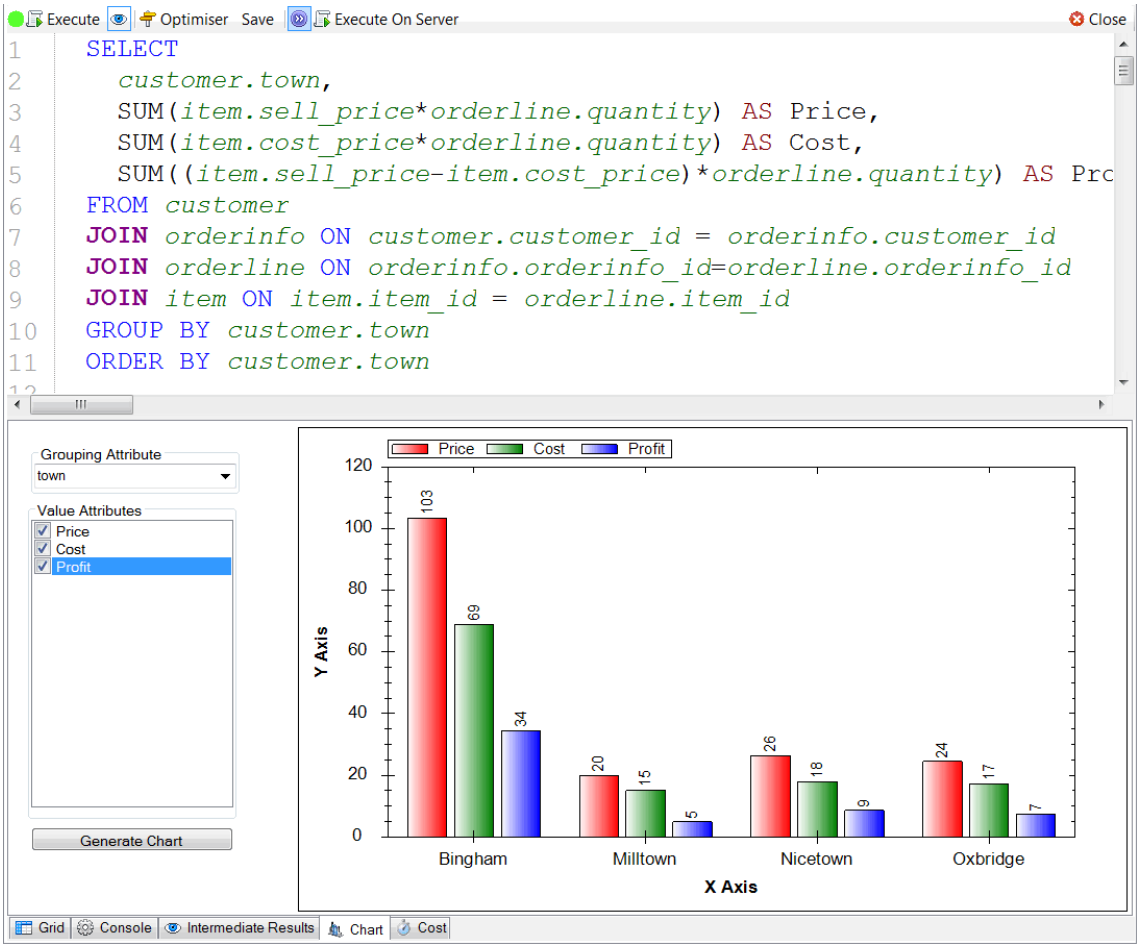

Figure 5.12: Chart View

5.6.4 Console

A simple but essential view that will indicate whether the execution was successful or if any errors occurred. If it was successful then the execution time and the number of rows are printed. In the case that there was an error it indicates if the error was a syntax or evaluation error along with information on the source of the error when applicable. Both type of messages are timestamped and represent the log of the evaluation engine actions.

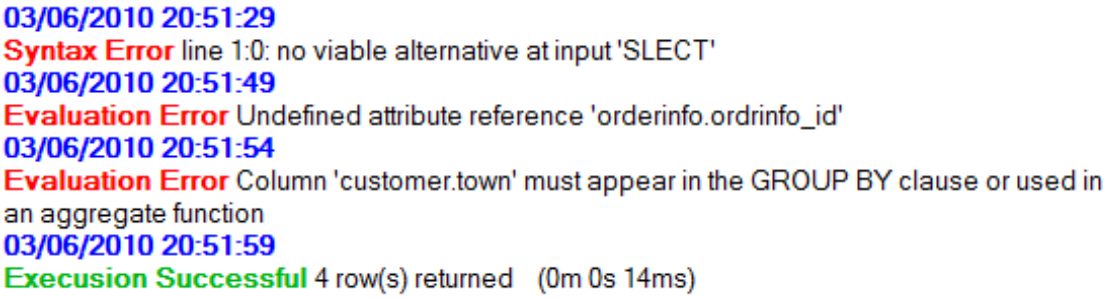

Figure 5.13: Console

5.6.5 Execution Cost

In Section 4.3.5 we explained how we can calculate various statistical measures of the execution. The user has the option to browse through these statistics. The Cost view will also be accompanied with a query tree like the intermediate results which will serve as a navigation mechanism in order to find statistics specific to each operation performed. The percentages of the evaluation time needed to evaluate the selected operator and the whole (sub) tree are shown using progress bars. The *Initial Cost* group indicates the time spent evaluating the child nodes therefore the execution time is less than the total time. Also there is a summary of the size of the output table at the node which shows the number of rows, the average row size and the overall size in bytes.

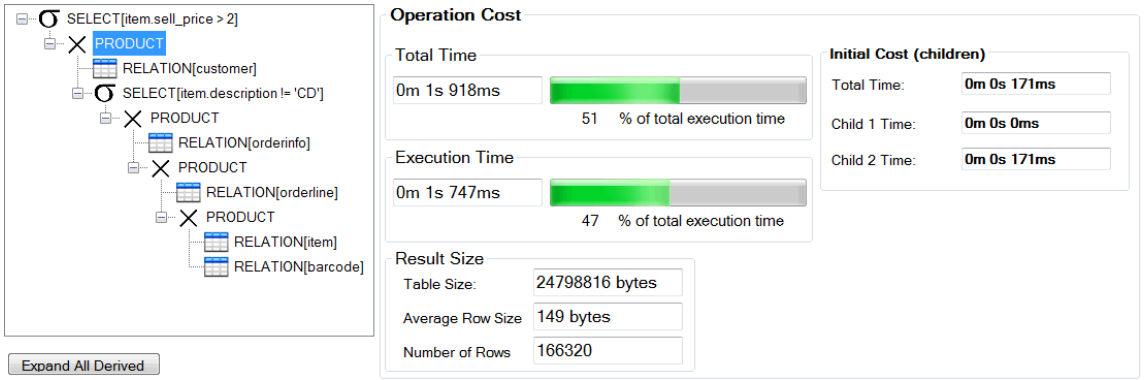

Figure 5.14: Execution Cost View

5.7 Visual Optimiser

The user can request from QV to optimise a query using the "Optimise" button on the tool bar above the query editor. This will create a new tab window with the tree structure of the query. In this window the system will apply the optimisation rules described in Section 4.4. The user can browse a series of trees that transform step by step the original tree to the most optimised one. In the tree browser the user can see the current state and the previous state as shown in Figure 5.15. Each time the browser proceeds to the next step the nodes that have been added or changed are highlighted with green. Also the previous state tree highlights in red the nodes that have changed.

There is an option to execute all the trees generated or only the most optimised. If all of the trees are executed a detailed statistics view is displayed which shows similar information as the *Execution Cost View* (5.6.5) with additional charts and graphs that compare the performance of each tree at the different stages of the optimisation.

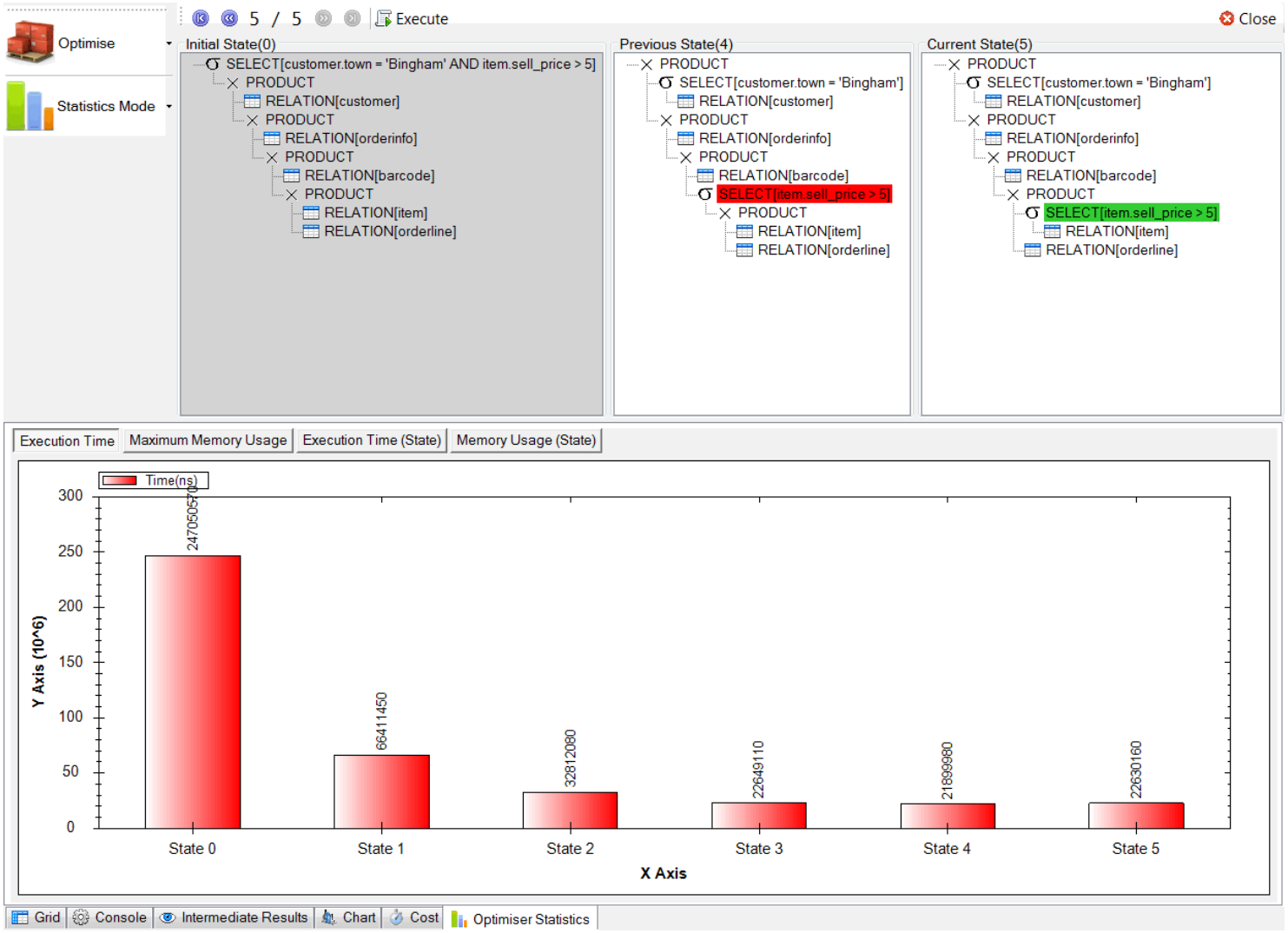

Figure 5.15: Optimiser Interface with execution statistics

5.7.1 Tree Execution Time

A bar chart that compares the execution time of each optimisation state with nanosecond accuracy. The optimiser in Figure 5.15 shows this view on the bottom section.

5.7.2 Tree Maximum Memory Usage

A bar chart that compares the maximum memory usage based on Equation 4.1. The memory size is calculated in bytes and an example is shown in Figure 5.16.

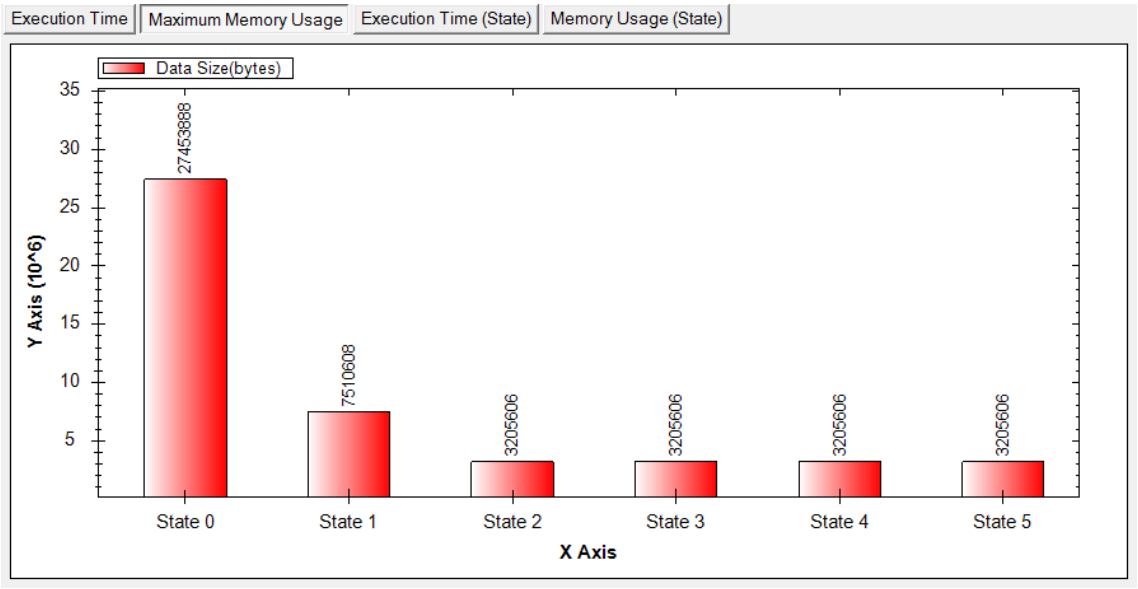

Figure 5.16: Maximum memory usage per tree

5.7.3 Operation Nodes Execution Time

For every state of the optimisation process the user has the option to view a pie chart that shows what percentage of the execution time each operation consumed. The pie chart included a legend that helps identifying the operations using the pie chart colours. In addition when the mouse cursor is on a specific piece of the chart the tool tip will indicate the operation as well. This is useful in cases that there are many operators and colours can not be distinguished. An example is shown in Figure 5.17.

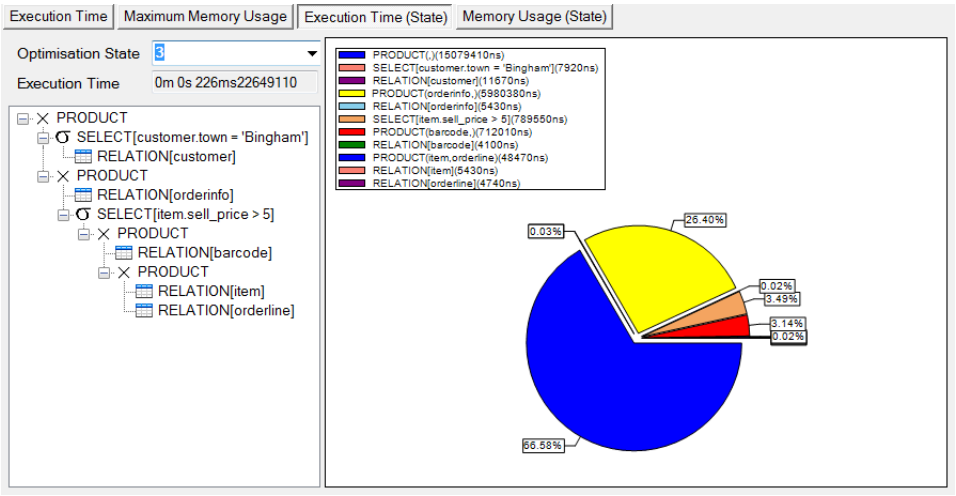

Figure 5.17: Execution time per operation

5.7.4 Operation Memory Usage

This is the last statistical view which shows for every state of the optimisation the memory usage of each operation performed. Also in a different colour (blue) the output size of the operation is shown. The purpose for displaying the two together is to see how close the two lines are to each other compared to other evaluations. This provides information on how quickly the evaluation drops redundant data.

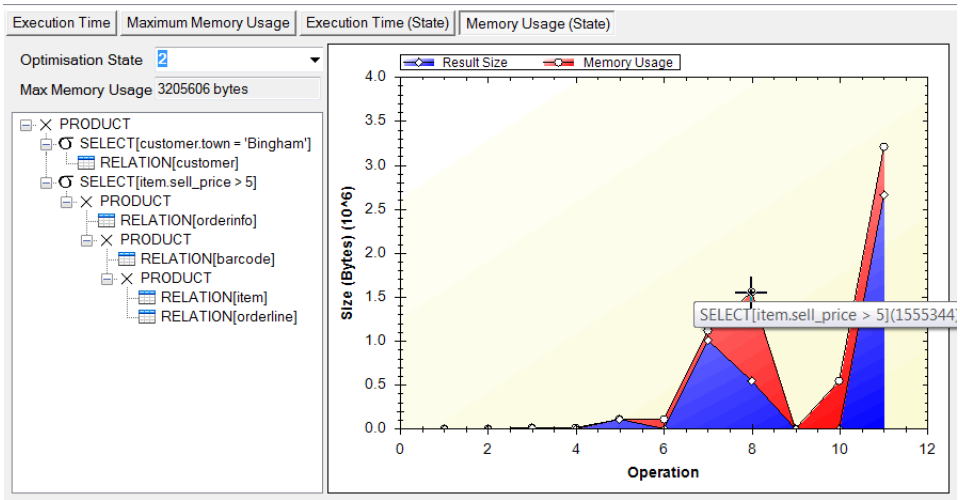

Figure 5.18: Memory usage per operation

Chapter 6

Implementation Details

*We developed **QV** based on the designed described in the previous chapters. Here we describe the most important details of the implementation. The main focus is on the parsing and evaluation process and how they produce the results as presented in the graphical interface described in Chapter 5. We emphasis on the modularity of the system and show how future changes can be incorporated.*

6.1 Overview

The implementation is split into several packages that can be compiled and be distributed independently as dynamic-link libraries (DLLs). Each such package serves a different purpose and are all based on the architecture described in Chapter 3. The packages that form QV are shown in Figure 6.1 with arrows showing the calls that each package makes to another. In addition to the dependencies implied by the arrows, packages are also dependent with lower layers (i.e. lower in the diagram) but there are no upward or cyclic dependencies.

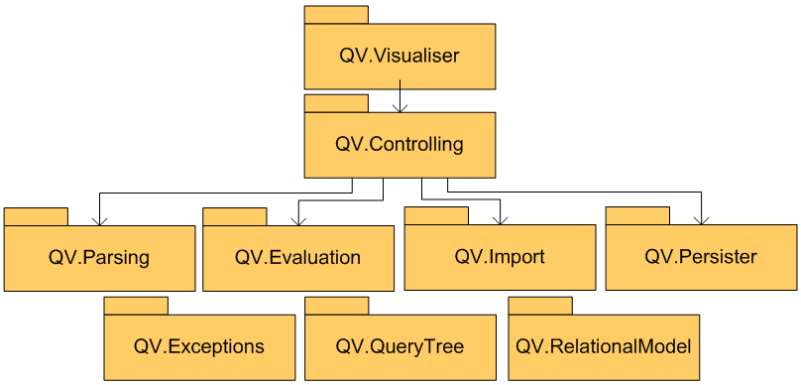

Figure 6.1: QV Packages

6.2 Graphical Interface

All the graphical interface components, including classes and images are contained in the *QV.Visualiser* package. The main entry point is the *Visualiser* class which contains the various panels that form the application workspace. The rest of the classes are distributed in the following sub-packages:

- **QV.Visualiser.RelationExplorer**: Contains classes that form the database browser on the left side of the workspace.
- **QV.Visualiser.Viewer**: The main tab container along with the relevant tab controls such as query, grid or optimiser tabs that can be drawn in the tab container. The important components of this package are the various classes that display the result of the query execution. For example, the *StatisticsResultsControl* class is responsible for the type of charts used in representing the statistics of the evaluation. Using the same logic the presentation details are split into other classes like the *IntermediateResultsControl* in which the tree structure and data lineage graphical settings are specified.
- **QV.Visualiser.QueryEditor**: The text editor components in which syntax highlighting and other editor features are defined. In this package using inheritance we reuse the main text editor for all supported query languages, including a special editor for defining derived operators.
- **QV.Visualiser.ToolStrips**: Contains the different toolstrips that are available throughout the system with the right mouse click option. Each one is responsible for firing events based on user selection. These events are captured and handled by the control that contains the toolstrip.
- **QV.Visualiser.Dialogs**: Contains dialog windows that arise in various cases such as the *new relation* and the *database import* dialogs.

6.3 Database Representation

The database that the user constructs is stored in memory while the application is running under objects that represent the relations in the *QV.RelationalModel* packages . Each such relation contains a *DataTable* which is an object in the ADO Library of .NET [6]. The *DataTable* is a wrapper of the *array* object which supports column text addressing and data types for each column. In addition all the *DataTables* that represent the relation are included in a *DataSet* which is another important object of .NET that can emulate a database by forcing restrictions such as unique relation names.

```
public class Project:UnaryOperation {
    public Project(List<Attribute> attributes,Operation child)
    :base(child){
        Attributes = attributes;
    }
    public List<Attribute> Attributes;
}
```

Figure 6.2: Node class of project

6.4 Workspace Persistence

In order to store the state of the workspace which includes the data model and currently open query windows the Persister class serialises the respective objects. First it serialises the content of the datatables that represent the relations and then it extracts the text from each query window. The serialised data are then stored in a single XML file with the extension ".qv" which can then be used to restore the workspace using the inverse of the serialisation process. All the relevant classes and the database connection libraries are included in the *QV.Persister* package.

An alternative persistence method is to store binary data of objects but it is not used since it will result in larger files and there is no backwards compatibility for the stored files if we make changes to the classes of these objects in future releases.

6.5 Query Tree Representation

For the purpose of representing the algebraic queries we use a tree structure. With abstract classes we split the unary and binary operators which inherit from the *UnaryOperator* and *BinaryOperator* classes respectively. Using inheritance we eliminate the code duplication and at the same time we can identify the characteristics of each node using type checking which is used extensively during the evaluation process. As an example the class that represents the *project operator* is shown in Figure 6.2. All operator classes are included in the *QV.QueryTree* package.

6.6 Parsers

In order to populate the query tree from the linear query input we will use a parser that will translate the input text into a tree. For this purpose ANTRL version 3 was used which is a parser generator. The grammar of each language is specified using similar notation to EBNF. This definitions besides checking if a query is valid they also incorporate code that creates the respective tree objects. When the grammar

file is compiled using ANTLR a C# parser class is generated. The grammar files and the generated parser classes are included in the *QV.Parsing* package. Detailed documentation of syntax that the grammar files use is available from the official ANTLR website [1].

The definition of the *project operation* is demonstrated in Figure 6.3 which returns a tree representing the project query. The code in curly brackets "{" "}" is C# code that is executed when a statement matching the syntax is found. This definition will match statements of the form "PROJECT[columnA,..,columnN] relation". Other operators are defined in a similar way.

```
1 PROJECT = 'PROJECT'
2 LSBRACKET       : '[';
3 RSBRACKET       : ']';
4 project_st returns [Project result]
5 @init{ Project result = null;}
6 @after{ retval.result = result;}
7 : PROJECT LSBRACKET colsSet=columns RSBRACKET
8         child=statement
9 {result = new Project( $colsSet.result, $child.result);};
```

Figure 6.3: ANTLR syntax definition of project

6.7 The Evaluation Engine

After the parsing phase has been complete the output is a query tree which the evaluation engine takes as input together with a database on which it executes the query.

The implementation is contained in the *QV.Evaluation* package and it follows the design described in Section 4.3. The main entry point is a static class that takes as input an operation and using type matching it determines which class (*specialist evaluator*) can evaluate the operation and creates an instance of it in order to perform the evaluation. This static class is responsible for identifying the type of the tree (RA or ERA) and tell the specialist class (e.g. *ProjectEvaluator* or *SelectEvaluator*) whether to use *set* or *bag* semantics. There is a total of 14 evaluator classes which is equal to the number of different operator classes and are listed in Appendix A. Each evaluator during evaluation will store the intermediate results it generates on the tree node using a *DataTable* together with a hash map that associates the rows of the table with the rows of the child table. The maps at every node are used by the graphical interface in order to show data lineage. We continue

[1] http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation

our demonstration with the project operator and we show the evaluation class code in Figure 6.4 that evaluates the *Project Node*. Other operators have been implemented in a similar way. The derived operators evaluator is more complex than others since it dynamically generates the parameters and we discuss the implementation in the next section.

```
1
2  public class ProjectEvaluator:IOperationEvaluator {
3
4   EvaluationResult evaluate(RelationSet set,Operation node,bool set){
5    Project projectNode = (Project)node;
6    if(null!=projectNode.Attributes){
7     childResult=base.evaluateNode(set,projectNode.Child,set);
8     EvaluationResult result=projectAttributes(projectNode.Attributes);
9     if (set){
10     SetFunctions.RemoveDuplicates(result.Table);
11    }
12   }
13   else{
14    throw new EvaluationException("No project attributes specified");
15   }
16   return result;
17  }
18
19  private EvaluationResult projectAttributes(List<Attribute> selected){
20   EvaluationResult result = new EvaluationResult();
21   result.Table = new DataTable(childResult.Table.TableName);
22   foreach (Attribute attribute in selected){
23    DataColumn newColumn =cloneColumn(childColumn);
24    result.Table.Columns.Add(newColumn);
25   }
26   copyRows(result.Table, childResult.Table);
27   return result;
28  }
29
30 }
```

Figure 6.4: Project Evaluator code

6.8 Dynamic Operator Construction

Using the relational algebra syntax the user can define new operators based on existing ones. The definition syntax also supports variable usage of projection attributes

and selection conditions which are converted to concrete references based on the database schema during evaluation.

- **PROJECT[#REF1 UNION #REF2]**

  The result will contain all unique attributes. *DIFFERENCE* and *INTERSECTION* are defined in a similar way.

- SELECT(#REF1.$=#REF2.$)

  The result will contain rows in which all attributes prefixed with the name of REF1 have equal value with attributes with the same name prefixed by REF2. Also except "=", any other comparison operator can be used.

In order to transform these definitions into concrete attribute references the derived operator evaluator will dynamically find them during the evaluation process. We describe the code of the evaluator using a flow chart in Figure 6.5.

6.9 Mono and multi-platform compatibility

All the source code of QV uses only managed code. Managed code is executed by the CLR - *Common Language Runtime* virtual machine and does not make any system calls associated with specific platforms. The reason for this is that the application can run under Mono[12] which is an implementation of a CLR and can compile and run C# code on different platforms. During development a tool that spots unmanaged code MoMA[2] - *Mono Migration Analyzer* was used and unmanaged code was replaced with managed code.

6.10 Third-party Modules

For the implementation of QV we used two third-party open source libraries which we extended in order to achieve certain functionalities specific to our system. the two cases that we used such libraries are for the query text editors and the graph drawing. For each one we explain what the library offers and what changes we made.

6.10.1 Text Editor

SharpDevelop[3] is an open source IDE for C#. One of its most important feature is the text editor which provides highlighting for C# code. Using this text editor we can define the syntax highlighting rules in an XML file. For our purposes we defined

[2]http://www.mono-project.com/MoMA
[3]http://www.icsharpcode.net/

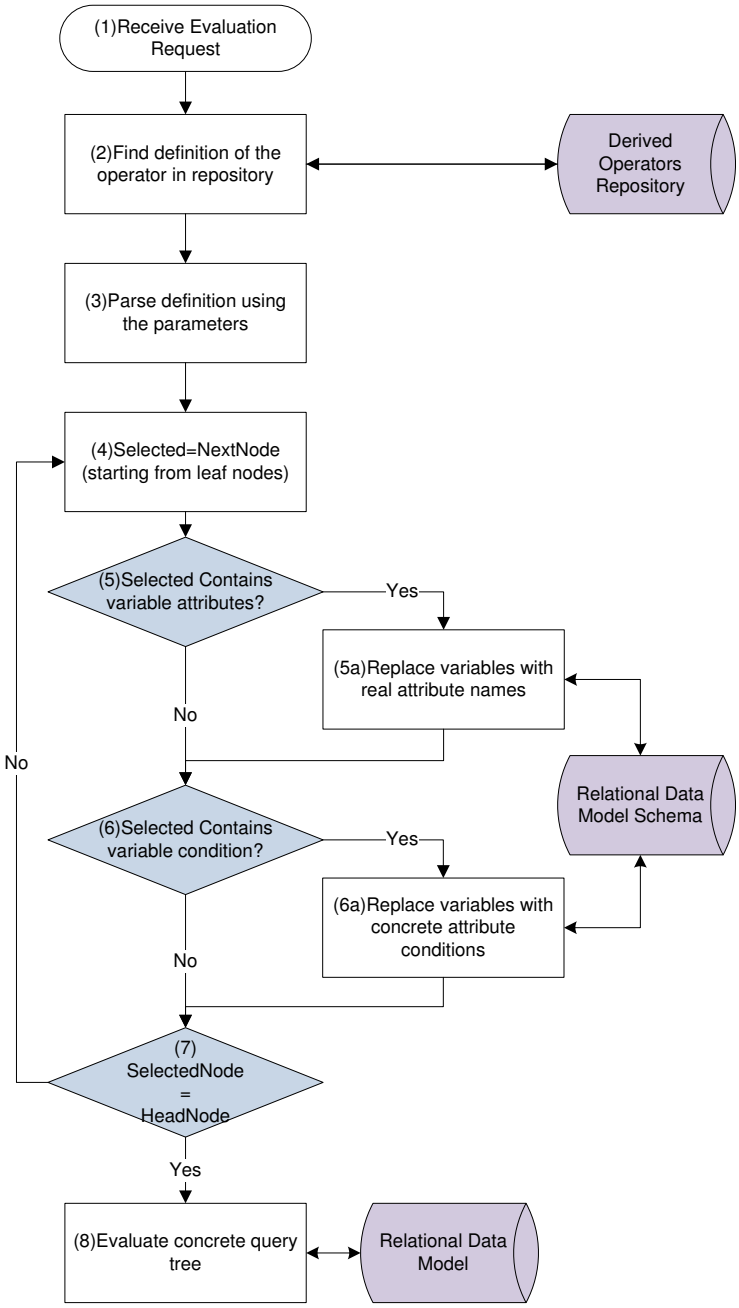

Figure 6.5: Dynamic Operator construction and evaluation

three such files that define for every keyword of each language that QV supports how they should be highlighted.

We extended this text editor into the *QVQueryEditor* which supports adding new syntax keywords at runtime. The purpose for this is to add the relation and attribute names in the highlighting rules which change at runtime . Also the derived operator names are added using the same method at the time that they are defined.

6.10.2 Drawing Charts

The users have the option to view the query result and the execution statistic using charts. We created a wrapper class around the open source 2D graph drawing library ZedGraph[4]. This wrapped class called *ChartView* can draw the content of a *DataTable* on a bar, line or pie chart. We encapsulated the work of setting up the charts in this class therefore the module is reusable in the various modules of QV. Future extensions to the system that will create charts only need to populate a DataTable with the values for the chart and specify the axes columns, the rest of the work is done by the ChartView that communicates directly with *ZedGraph*.

6.11 Creating ERA Trees from SQL

When the parsing of an SQL statement has successfully completed the result is a single object which has the form of either a boolean set operator (union,except and intersect) or it is a select statement. This object passes through a converter which will create a new ERA tree that the evaluation engine we implemented can support. For the purpose of this conversion we predefined a set of derived operator definitions such as the *theta join* that the user can only browse but can not modify. We use this operators mostly when we are converting an SQL join since using directly the basic operators creates a large tree that most users will find hard to understand. Alternatively the trees are presented with the derived operators which the user can still expand and analyse. A full list of the predefined operators can be found in in Appendix A.

6.12 Extending QV

This section outlines how to make changes to the existing languages and how support for new query languages can be added to QV with only a few modifications in the code base.

[4]http://zedgraph.org/

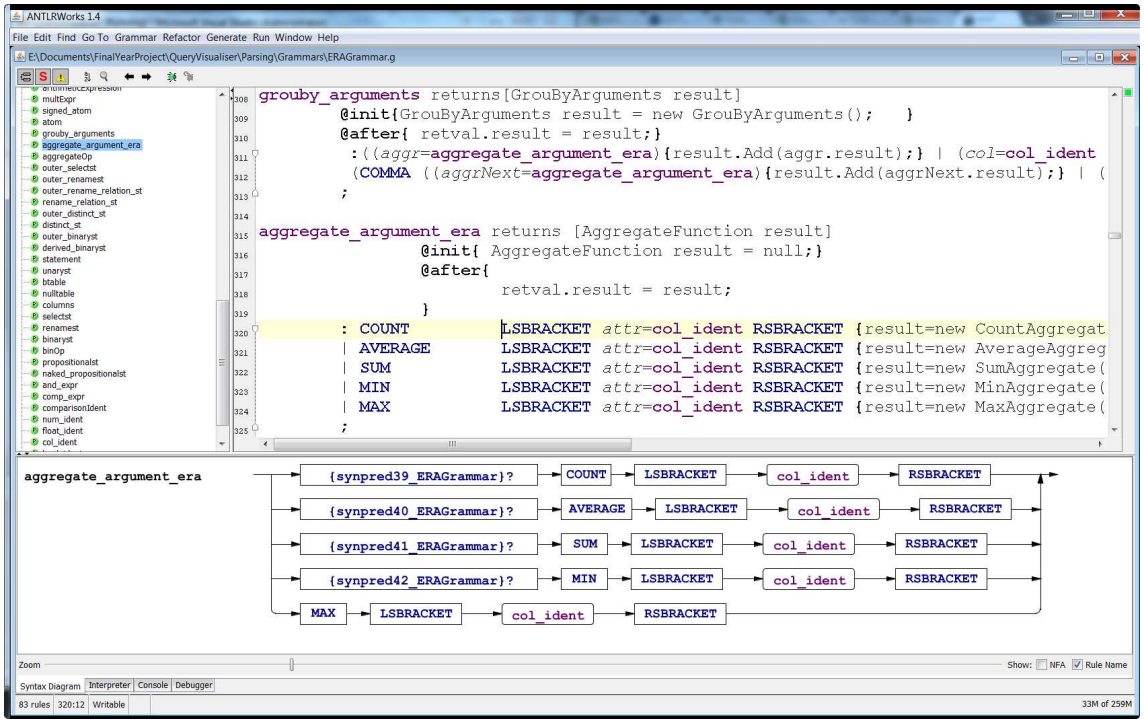

Figure 6.6: Extended relational algebra grammar in ANTLRWorks

6.12.1 Extending the existing languages

We can make changes to the languages such as the names of operators and the type of brackets without changing the semantics. The complete syntax of RA, ERA and SQL can be found in the grammar files *RAGrammar.g*, *ERAGrammar.g* and *SQLGrammar.g* under the *QV.Parsing* package. These files are editable in ANTLRWorks[5] which is an IDE for ANTLR. When writing a definition you can see the respective syntax diagram and test what statements it supports. Each file has been compiled in order to generate the respective parser and changes require recompiling of the modified file.

In order to introduce a new operator or change the parameters of an existing one, the appropriate changes need to be made in existing tree classes like the one in Figure 6.2 or a new class to represent the new operator has to be created in the *QV.QueryTree* package. In addition the appropriate modifications need to be made in the evaluation package, *QV.Evaluation*, by adding new evaluation classes or changing the existing ones.

[5]The latest version of ANTLRWorks can be downloaded from http://www.antlr.org/works

6.12.2 Introducing a new query language

QV supports three language RA, ERA and SQL but there is only one evaluation engine that can apply a set of operations. These operations provide semantic completeness for all three languages. The benefit from this common evaluation engine is that we can easily introduce a new query language such as *Tuple Relational Calculus*. In fact any query language that is equal or less expressive than RA or ERA can be integrated to our system without affecting the evaluation engine. The following is a summary of the required steps to introduce the new query language in QV.

1. **QV.Parsing**: Create an ANTLR grammar file for the language. In the grammar file include code that will generate the respective operation node for the possible statements of the new language. When the syntax with the incorporated transformation code has been defined compile the grammar file to create the C# parser. See Section 6.6 for more details.

2. **QV.Controlling**: Create a new Controller class that implements the IQueryController interface. This class will call this parser and then using the result call the existing evaluation engine. Then add the new language as one of the available languages of QV and associate it with the new controller in the controllers dictionary. This will automatically direct the execution calls to the correct controller for each language. See Figure 6.7 to see how this works.

3. **QV.Visualiser.QueryEditor**: Create a new query editor class that extends the *QVQueryEditor*. Then create an XML file that contains the highlighting rules and set it as the highlighting source. For the implementation details consult the existing editors.

4. **QV.Visualiser**: Extend the general workspace class (Visualiser.cs) main menu, so that it included an option to open a query window with the new query editor.

```
namespace QV.Controlling
{
 public class SetController
 {
  ...
  public enum QueryType { RA = 1, SQL = 2,ERA=3};

  private Dictionary<QueryType,IQueryController> controllers;
  controllers = new Dictionary<QueryType,IQueryController>(){
    {QueryType.RA,new RAQueryController()},
    {QueryType.SQL,new SQLQueryController()},
    {QueryType.ERA,new ERAQueryController()}
   };

   public EvaluationResult executeQuery(string query,QueryType queryT,
   params ResultType[] resultTypes){
   return controllers[queryT].evaluate(activeSet,query,resultTypes);
  }
 }
 ...
}
```

Figure 6.7: Calling the correct execution controller (QV.Controlling.SetController.cs)

Chapter 7

Correctness and Stability

It is important to test the system in terms of correctness of the query evaluation engine and the stability of the overall system. In this chapter we describe the test we performed and present the results.

7.1 Experiments

To asses the system we used a set of automated unit tests and performed stress testing to asses the stability of the system. The results and their analysis are documented in Sections 7.2 and 7.3 respectively.

7.1.1 Unit testing

The evaluation engine can be evaluated using automated tests that can verify that the results of the queries are correct. For this purpose we rely on a PostgreSQL server that our testing platform connects with and we compare the query results of QV against the results of the server. We split this testing phase into three sections and assess the result for correctness of the RA, ERA and SQL evaluation modules.

We define a set of queries for each language and their equivalent query in SQL[1]. The test evaluates the query on QV and the equivalent SQL query on the server. When the results have been computed it checks if the heading (attribute names and data types) and the body (rows) of the two results are the same.

To correctly asses the system we specify queries that are correct and queries that are not and we expect the system in both cases to agree with the server. We define 93 different queries which aim to check different capabilities of each language. All of the queries are applied on a relatively large database of world statistics on countries

[1]for the case of SQL the equivalent is the same

and cities which is publicly available from *Statistics Finland*[2]. A sample of the unit tests is shown in Appendix A. For the case of SQL the test cases cover the features of SQL that have been implemented and do not attempt to check all supported statement of SQL in general. The complete implementation of the testing platform including all the test cases can be found in the archive under QV.Tests.

7.1.2 Stress Testing

The purpose of this test is to find the load that QV can handle while loading a database and executing queries. The only parameter for the load capacity is the memory usage since QV uses an in memory database and does not make use of stable storage. The aim is to to determine if the system becomes unresponsive when it runs out of computational resources or if it can stay active and recover. For the purpose of this test we used a Dell Vostro 410 PC with an Intel Core Q9300 processor and 4GB of DDR2 RAM. Using the same database as for the unit tests we incrementally apply *cartesian product* and *select* operations until the system runs out of computational resources at which point we see how the system responds and determine the load it can handle in terms of database size and query complexity.

7.2 Results

Unit testing

All 93 query results of QV matched the results of the server. The only case where there was a small difference was on the column names in SQL for arithmetic expressions and aggregates when a name was not specified in the query. Besides the column names in this case the tables content were still correct.

Stress Testing

While incrementally extending a query by adding different types of operators, mostly *Cartesian product*, *select* and *union* which are the most expensive in terms of computation and memory usage we reached the limit of the system which is a query with 15 operations and a total of 4,302,150 rows hosted in the nodes of the tree. The total memory occupied by QV when it ran out of memory was 946 megabytes. The memory usage graph is shown in Figure 7.1 and the following are the actions performed at each checkpoint on the graph.

0. Launched QV.

[2]The database is available at http://pgfoundry.org/projects/dbsamples/

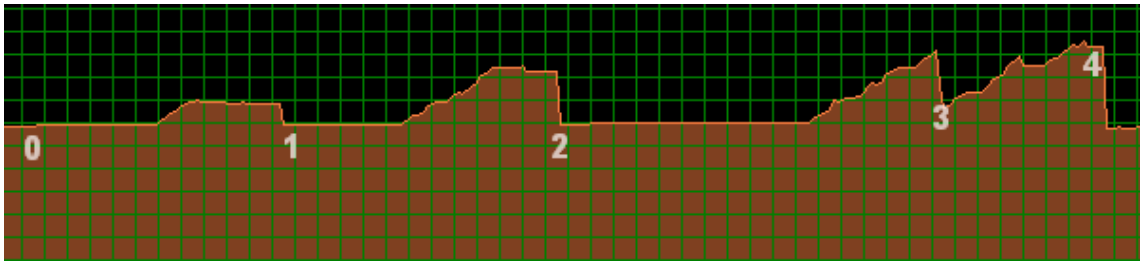

Figure 7.1: Memory usage graph

1. First evaluation of query completed.
2. Second evaluation of query completed.
3. Garbage Collection performed while executing third evaluation.
4. CLR virtual machine ran out of memory.

In order to find the maximum capacity that the system can handle, we changed one of the selection conditions in the query so that less rows would agree. By changing this condition to include one more row each time it was successful, we determined an approximate maximum in terms of memory size which was *907 megabytes* with *4,270,167 total rows* in the tree.

7.3 Analysis

The unit tests verify that the system correctly executes the queries that it allows the user to define. Some SQL feature such as summarising data with *CUBE* and *ROLLUP* can not be evaluated by the system since they are not part of our implementation. The aim of QV is to show how the evaluation engine works and serve as a teaching tool rather than a complete database management system.

The number of operations performed during the stress test does not reflect the capabilities of the system since the only limitation is the memory size. Also we need to note that the CLR virtual machine like most runtimes, reserves memory so it can perform other operations, such as garbage collection . Therefore, the system run out of memory although there were more than 500 free megabytes left.

The important part is that the system always clears the unnecessary memory when moving on to the next execution. Also when it runs out of memory it informs the user in the console, clears the memory and continues operating in a stable state where the user can continue working without crashing.

Chapter 8

Usability Experiments

Since the purpose of the system is to assist students in learning formal query languages we will asses the usability of the system using two methods. The first one is an end-user test where users are asked to solve an exercise sheet using QV and then answer a questionnaire. The second evaluation test is the assessment of QV against the Nielsen's Ten Usability Heuristics[13] which we describe in Section 8.2.

8.1 End-User Test

Five end users were asked to solve an exercise sheet using QV. While the users solve the exercises we monitored how they used the system and at the end we asked them to complete a questionnaire. Before the tutorial each user was given an introduction to the system for five minutes and then with no further guidance they solved the exercise sheet. We aim to gain useful insights on common user mistakes and determine whether such mistakes are related to our user interface. The exercise sheet and the questionnaire are shown in Sections 8.1.2 and 8.1.3 respectively. The solutions of the exercises can be found in Appendix B. The results of the experiment and their analysis are documented in Sections 8.1.4 and 8.1.5.

8.1.1 End-user profiles

The five users that took part in the experiment where all undergraduate computing students. Due to resource limitations all the students have already been introduced to relational databases which makes them familiar with relational algebra and SQL . This can influence the results of the test in different ways because they already know how to construct queries. Also we should note that they could also be certain for their answers and feel comfortable with the algebraic notation and not be interested in analysing their queries using QV. In order to limit the effect of these parameters

the exercise sheet that the users solved contains questions that required them to construct queries of different complexity levels.

User	Relevant Course	When
User 1 and User 2	Advanced Databases	One year before the test
User 3 and User 4	Advanced Databases	Six months before the test
User 5	Introductory Databases	One month before the test

8.1.2 Exercise Sheet

Find queries in the respective query language for all questions. Most of the questions provide the columns that the result must contain. Were indicated with an asterisk (*) answer the question on paper before using QV. The questions are related to the database of a small store[1]. The database has information on item prices, stock, customers and orders. It consists of the 6 tables shown in Figure 8.1.

item			
item_id	description	cost_price	sell_price
...			

stock	
item_id	quantity
...	

barcode	
item_id	barcode_ean
...	

customer							
customer_id	title	fname	lname	addressline	town	zipcode	phone
...							

orderinfo				
orderinfo_id	customer_id	date_placed	date_shipped	shipping
...				

orderline		
orderinfo_id	item_id	quantity
...		

Figure 8.1: Store Database Schema

[1] Available at http://www.postgresqlguide.com/postgresql-sample-database.aspx

Relational Algebra

Question 1* Create a list that contains only the item IDs and descriptions.
Header:item_ id,description

Question 2* Create a list of all items with their cost price and barcode EAN.
Header:item_id,cost_price, barcode_ean

Question 3* List all phone numbers in town *Bingham*.
Header: phone

Question 4* List the first and last name of customers that purchased the item with description *Linux CD*.
Header: fname,lname

Question 5 Semijoin $\mathbf{R} \ltimes \mathbf{S}$ is the derived operator that returns all tuples in R for which there is a tuple in S that is equal on their common attribute names. Using the derived operator definition language in QV create a definition for the right *semijoin* $\mathbf{R} \ltimes \mathbf{S}$ that performs the same operation with the only difference that the result only contains attributes from S instead of R.
Hint: Find the definition "SEMIJOIN" to see how $\mathbf{R} \ltimes \mathbf{S}$ *is defined.*

SQL

Hint: To solve the next two questions try to use the SUM aggregate and GROUP BY where applicable

Question 1 Find the total number of items in stock.

Question 2 Calculate the total sell price of all items for every order made.
Header:orderinfo_id,Total

Extended Relational Algebra

Question 1 List the profit (cost_price - sell_price) for all items.
Header:item_id,difference

Question 2 Find the total quantity of Wood Puzzles sold.

Question 3 List the number of barcodes assigned to each item. If no barcode is available for an item then the result will show an entry with count 0.
Header:item_id,barcode_count

8.1.3 Questionnaire

The users are asked to answer the following questions with rating from 0(=not at all) to 5(=definitely yes).

1. How easy was to use the user interface?
2. How easy was the relational algebra syntax?
3. How much confidence did you gain in your relational algebra answers compared to the paper answers?
4. How easy to understand was the extended relational language?
5. How helpful were the error messages?
6. To what extend is the intermediate results tree more informative than the simple results table?
7. How easy was to understand how the derived operators are executed using the break-down feature?
8. How easy do you think is the process of defining derived operators?

8.1.4 Results

During the test we recorded the time spent answering the questions for each user and at the end of the test we checked which answers where correct. The results are presented in Table 8.1. In brackets we indicate the solutions of the questions answered on paper. Also the result of the questionnaire are shown in Table 8.2

User	**Time**	**RA**					**SQL**		**ERA**		
		1*	**2***	**3***	**4***	**5**	**1**	**2**	**1**	**2**	**3**
User 1	25 minutes	✓(✓)	✓(✓)	✓(✓)	✓(✓)	✗	✓	✗	✓	✓	✓
User 2	34 minutes	✓(✓)	✓(✓)	✓(✓)	✓(✓)	✓	✓	✓	✓	✗	✓
User 3	19 minutes	✓(✓)	✓(✓)	✓(✓)	✓(✗)	✓	✓	✓	✓	✗	✗
User 4	26 minutes	✓(✓)	✓(✓)	✓(✓)	✓(✓)	✓	✓	✗	✓	✓	✓
User 5	35 minutes	✓(✓)	✓(✗)	✓(✓)	✗(✗)	✗	✓	✗	✓	✗	✓

Table 8.1: Exercise Sheet Results

User	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8
User 1	5	5	4	5	3	4	5	4
User 2	4	4	3	4	4	4	4	4
User 3	4	4	4	4	3	5	5	5
User 4	5	5	3	4	4	4	5	4
User 5	4	4	5	3	3	5	5	3
Average	**4.4**	**4.4**	**3.8**	**4**	**3.4**	**4.4**	**4.8**	**4**

Table 8.2: Questionnaire Results

8.1.5 Analysis

The majority of the users mostly spent their time trying to find the correct queries and not navigating the tool. Users 1 and 3 did not use the intermediate results view to verify their queries but they only used it to check the structure of the queries and whether the operators where executed in the correct order. The rest of the users spent time analysing step by step their result. User 2 was not sure how the different join operators of SQL work but he was able to figure them out using the query trees that were constructed from the SQL statement and found the correct answer to the SQL question 2. The number of correct answers of the experienced users were better than the student that did not participate in an advanced databases course although we get a positive feedback for his performance from the fact that on paper he was unable to find the correct answer for the fourth relational algebra question but by joining relations one by one and checking the result he was able to incrementally find the correct answer in QV. Some users could not understand why some of their answers were wrong and they realised their mistakes when they were encouraged to use the data lineage feature through which they realised why unnecessary rows were in the result and updated their queries to fix them.

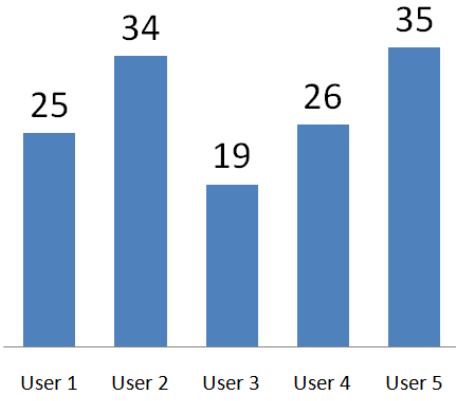

Figure 8.2: Exercise Sheet Time

From the answers to the questionnaire we get positive feedback about the usability and the value of the system. Although the users rated 3.8 out of 5 the value QV adds when writing relational algebra queries, the users had already been exposed to a great extend with relational algebra and similar queries to those that they were asked to construct have been covered in their courses. This tool will be more useful

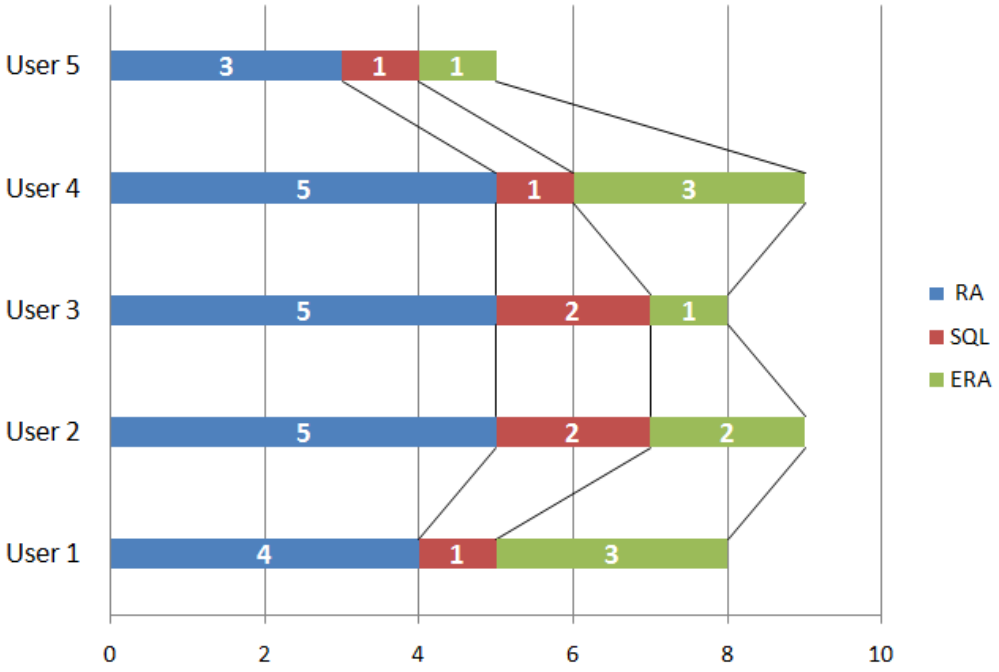

Figure 8.3: Exercise Sheet Correct Answers

to students during their introductory and advanced databases courses. In addition the users did not seem completely satisfied with the error messages they received and from their comments we deduce that more detailed feedback should be provided.

At the end of the exercise we also asked the users what could be improved in QV and most of them mentioned that the derived operators editor would be much better if it could execute their definitions so that they could debug them without saving them first. In addition some of them mentioned that when there is a syntax error the editor should underline the tokens that were wrong in addition to the console message.

Besides the above indications of the questionnaire and the recommendations of the users the general feedback we received is that QV is a useful tool that it can be really helpful while learning about formal query languages, especially for the purpose of solving exercises.

8.2 Nielsen's Usability Heuristics

We evaluate QV with the Nielsen's Ten Usability Heuristics[13] by showing how it meets the terms of each one by stating relevant facts about QV's graphical interface.

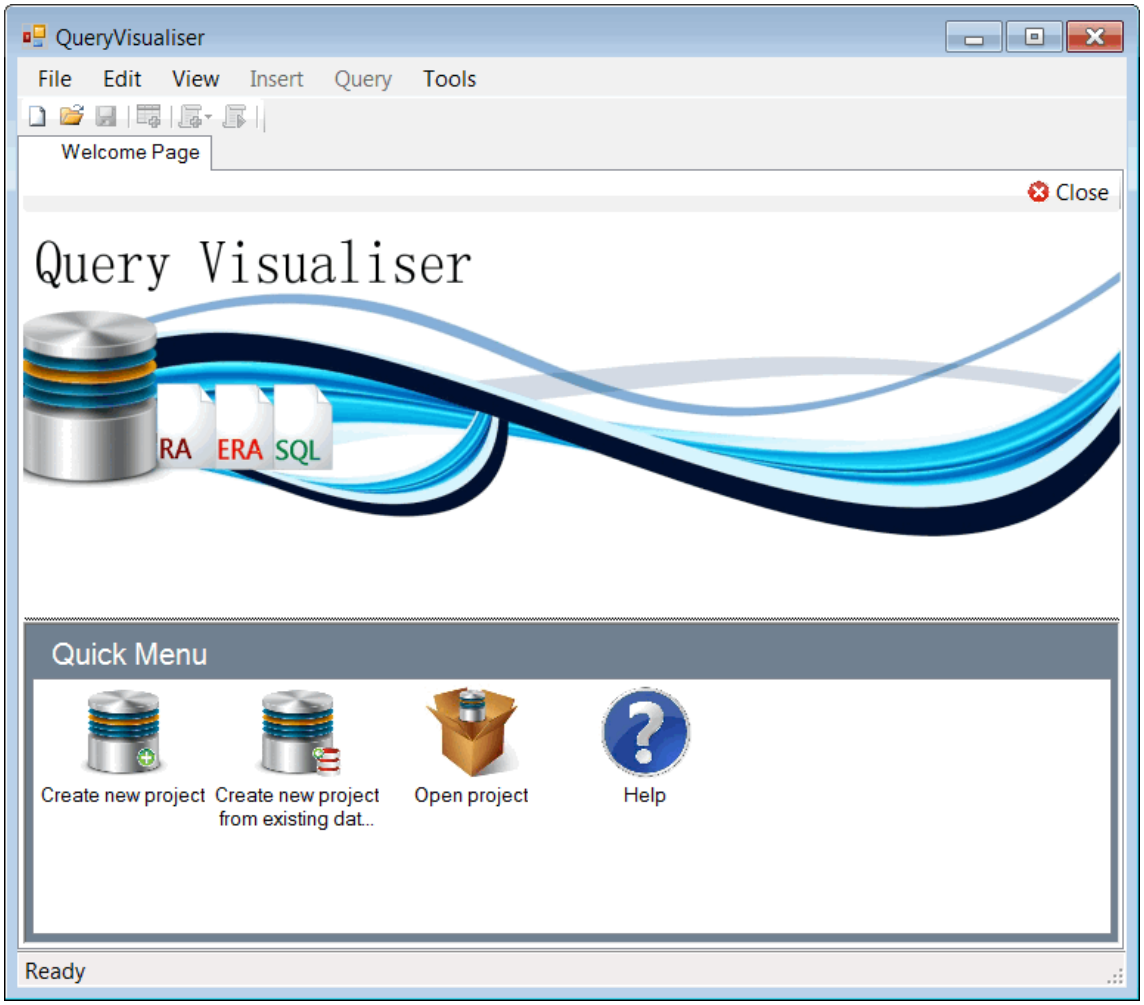

Figure 8.4: QV Welcome Screen

1. ***Visibility of system status***

   The status of the system is indicated in the status bar at the bottom of the screen at all times.

2. ***Match between system and real world***

   The terminology used throughout QV it either uses real world terminology or simple terms that arise in the context of databases. System specific terms such as *Query Evaluator Engine* are not displayed to the user but rather simple statements that the user can understand without prior experience with the system are used. The icons used in the system such as on buttons try to explain the action that they perform. For example the *Minimize*, *New Project*,

Execute and other button icons are not misleading but rather they act as extra information for their usage. All options that are visually available to the user such as toolbar buttons and menus either they can be clicked like real buttons with visual feedback and they perform the expected action, such as "Save As" and "Execute" or they are disabled in which case this is visually indicated with flat gray buttons which are not responsive when clicked.

3. ***User control and freedom***

   The *query editor* supports undo and redo operations. All the modification actions that can be performed on the database using the *relation explorer* are reversible.

4. ***Consistency and standards***

   Throughout the system the terminology is the same. For example in all screens we always refer to *attributes* and not both *attrubutes* and *columns*

5. ***Error prevention***

   The user is prevented from making mistakes and render the system in an unstable state. A few of these cases are the following

   - The user can not change the attribute data type because this could lead to data inconsistency.
   - When adding new relations, attributes views and operators their names are always checked and the user can not perform the change if there is a name clash.
   - The definition of the derived operators are always validated before saving and syntactically invalid operators are not saved.
   - Overloading the execution engine is not possible since the execution button is disabled while the system is evaluation a query.

6. ***Recognition rather than recall***

   The user does not need to remember the content of previous dialogs with the system since all information for every action are captured using a single screen and not using a wizard with multiple dialogs. In addition while constructing queries the user has visual access to the relations and their attributes and does not need to remember their names and types.

7. ***Flexibility and efficiency of use***

   QV like most popular IDEs offers a tabular workspace in which the user can work on diffenet tasks interchangeably using multiple tabs that can host query

editor, relation editor and visual optimiser instances. To further improve the the flexibility of the system the user has the option to add a second tab container to gain visual access to two *tasks* at the same time. The state of the workspace is stored in the project and it can be restored in the same state later. In addition for efficiency the user can use the major keyboard shortcut of most applications such as *CTRL+N* to create a new project, and *CTRL+O* to open an existing one. In addition QV specific shortcuts have also been added such as *F5* to execute the active query and *CTRL+SHIFT+N* to create a new project from an existing database source.

8. ***Aesthetic and minimalist design***

   When working in QV no extra information is displayed and the options available to the user at any time are only the relevant ones. For example the toolbar for the relational algebra language is not shown in the SQL editor. The tab headers of the workspace indicate the type of the tab and where applicable the relevant name.

9. ***Help users recognise, diagnose and recover from errors***

   When executing an invalid query the output screen is always redirected to the *Console View* where a new timestamped message with the type of the error and further information for the source of the error is shown. Also when dialog validations fail the user receives warning messages that explain what caused the validation to fail.

10. ***Help and documentation***

    Part of this report and especially the Workspace description throughout Chapter 5 describes in detail all the features of QV which is sufficient documentation for a first time user. In addition the distribution of QV includes a document with a list of keyboard shortcuts and a *cheat sheet* of the important operators with their syntax and a description of the supported query languages. To help the users more we could integrate an interactive guide with tutorials. Due to time constrains we were unable to produce this feature although because of the modular architecture used we can easily integrate this feature in the future.

Chapter 9

Conclusion

In this final chapter we discuss our achievements and discuss how we can extend QV with future work

9.1 Achievements and contributions

One of the major motives for the project was the lack of transparency of the query evaluation process to the user. Using multiple methods for presenting the work of the evaluation engine, QV provides the user with enough visual information to understand the fundamentals of the execution process of relational algebra, extended relational algebra and SQL.

An important property of QV is that it uses only basic operations for evaluation which gives the user the ability to see how the the derived operators are constructed from them. Additionally users get more involved with the relational algebra by defining new operators that are dynamically constructed.

To assist in the learning of SQL the system provides an insight on the query evaluation process by showing an easy to comprehend tree, constructed by the actual operations that are performed. These operators are part of the extended relational algebra which fills the expressiveness gap of the basic relational algebra and SQL. Optionally queries can be defined directly in the extended relational algebra which helps appreciating the difference between declarative query statements and procedural algebra expressions.

The algebra optimisations that QV performs is a simplified version of the techniques used in database management systems. By presenting step by step the tree transformations we can see how the result remains unaffected and at the same time see the performance improvements in the analytical statistic displays.

The various information on the evaluation results are not shown on a unified display but rather in multiple sections throughout QV. This makes it a suitable tool

for users that have no prior experience with databases and formal query languages and need a tool with visual information which contains no redundancies. Optionally more advanced users can construct more complex queries and analyse them by accessing the advanced displays.

9.2 Future Work

The system can be extended in different ways depending on what is considered more important. This can be determined by considering the experience level of the users and what would be more helpful for them. Bellow we describe some interesting extensions that can follow:

- The implementation of SQL is limited in functionality and can be extended to support more OLAP operations such as CUBE and ROLL UP. Additionally other features of SQL such as conditional projection using the CASE clause can also be added to the current implementation. The extended relational algebra is equally expressive as SQL therefore adding support for this type of SQL features is a simple process of correctly translating the statement to the correct operations.

- Currently the database and the intermediate results are stored in memory which limits the performance of QV. We can modify the implementation to make use of stable storage to unload and reload information when the system runs out of memory. This would increase the capacity that QV can handle. Such an approach was not considered in our original design since the aim is to use QV mostly for learning purposes. Making this change along with the previous extension on SQL can widen the usage of the system.

- We implemented some optimisation methods which are purely based on the semantics of the operations. The rules of the optimiser could be improved by making use of statistics of the database. Additionally the user could use a scripting language to introduce customised rules for the optimisation. These rules can be based on criteria such as maximum memory usage and loop complexity. Using the rules the optimiser can correctly decide whether it is better to limit early the number of rows or columns depending on the available resources.

Bibliography

[1] RDBI. http://rdbi.sourceforge.net/.

[2] Ana Paula Appel, Elaine Quintino Silva, Caetano Traina Jr., and Agma Juci Machado Traina. idfql: A query-based tool to help the teaching process of the relational algebra. In *World Conference on Enginnering and Technology Education - WCETE'2004*, Guarujá, SP, 2004.

[3] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill Book Company, 1999.

[4] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. (4):324–345, April 1985.

[5] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6)(1):377–387, 1970.

[6] Microsoft Corporation. MSDN library, ADO.NET 3.5 documentation. http://msdn.microsoft.com/en-us/library/aa286484(v=MSDN.10).aspx.

[7] Suzanne W. Dietrich, Eric Eckert, and Kevin Piscator. Winrdbi: a windows-based relational database educational tool. In *SIGCSE*, pages 126–130, 1997.

[8] P. W. P. J. Grefen and R. A. de By. A multi-set extended relational algebra: a formal approach to a practical issue. In *Proc. 10th International Conference Data Engineering*, pages 80–88, February 14–18, 1994.

[9] AquaFold Inc. Aqua data studio. http://www.aquafold.com/.

[10] Peter J McBrien. *Advanced Databases*. Imperial College London, 2008.

[11] Pritam Mitra. *Relational Algebra Learning Tool*. 2009.

[12] Mono. Cross-platform .NET framework. http://mono-project.com/.

[13] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 152–158, New York, NY, USA, 1994. ACM.

[14] Josep Soler, Imma Boada, Ferran Prados, Jordi Poch, and Ramón Fabregat. An automatic correction tool for relational algebra queries. In *ICCSA (2)*, pages 861–872, 2007.

[15] Salvo Tomaselli. RELATIONAL - educational tool for relational algebra. `http://galileo.dmi.unict.it/wiki/relational/`.

[16] W3C. Extensible markup language EBNF notation (third edition), February 2004. `http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation`.

Appendix A

QV Details

Evaluation Engines

List of the implemented evaluation engines found under the namespace QV.Evaluation

- RA.DerivedBinaryEvaluator
- RA.DifferenceEvaluator
- RA.ProductEvaluator
- RA.ProjectEvaluator
- RA.RelationEvaluator
- RA.RenameEvaluator
- RA.SelectEvaluator
- RA.UnionEvaluator
- RA.GroupByEvaluator
- RA.DistinctEvaluator
- RA.ProjectExtendedEvaluator
- RA.RelationRenameEvaluator
- RA.SelectExtendedEvaluator
- RA.SortEvaluator

Predefined Derived Oeprators

INTERSECTION

```
#REF1 DIFFERENCE ( #REF1 DIFFERENCE #REF2 )
```

NATURAL_JOIN

```
PROJECT[#REF1 UNION #REF2]
```

```
SELECT[ #REF1.$=#REF2.$ ] (#REF1 PRODUCT #REF2)
```

SEMI_JOIN

```
#REF1 NATURAL_JOIN PROJECT[#REF2 INTERSECT #REF1] #REF2
```

ANTI_JOIN

```
#REF1 DIFFERENCE (#REF1 SEMI_JOIN #REF2)
```

LEFT_OUTER_JOIN

```
(#REF1 NATURAL_JOIN #REF2)
UNION PROJECT[#REF1 UNION #REF2]
((#REF1 ANTI_JOIN #REF2) PRODUCT NULLRELATION(#REF2))
```

DIVISION

```
PROJECT[#REF1 DIFFERENCE #REF2]
#REF1 DIFFERENCE PROJECT[#REF1 DIFFERENCE #REF2]
((PROJECT[#REF1 DIFFERENCE #REF2]
(#REF1 PRODUCT #REF2) PRODUCT #REF2)
DIFFERENCE #REF1)
```

THETA_JOIN

```
SELECT[#PARAM](#REF1 PRODUCT #REF2)
```

LEFT_THETA_SEMI_JOIN

```
PROJECT[#REF1] (#REF1 THETA_JOIN[#PARAM] #REF2)
```

RIGHT_THETA_SEMI_JOIN

```
PROJECT[#REF2] (#REF1 THETA_JOIN[#PARAM] #REF2)
```

LEFT_THETA_JOIN

```
(#REF1 THETA_JOIN[#PARAM] #REF2) UNION
((#REF1 DIFFERENCE (#REF1 LEFT_THETA_SEMI_JOIN[#PARAM] #REF2))
PRODUCT (NULLRELATION(#REF2)))
```

RIGHT_THETA_JOIN

```
(#REF1 THETA_JOIN[#PARAM] #REF2) UNION
((#REF2 DIFFERENCE (#REF2 LEFT_THETA_SEMI_JOIN[#PARAM] #REF1))
PRODUCT (NULLRELATION(#REF1)))
```

FULL_THETA_JOIN

```
#REF1 THETA_JOIN[#PARAM] #REF2) UNION
(((#REF1 DIFFERENCE (#REF1 LEFT_THETA_SEMI_JOIN[#PARAM] #REF2))
PRODUCT (NULLRELATION(#REF2))) UNION
((#REF2 DIFFERENCE (#REF2 LEFT_THETA_SEMI_JOIN[#PARAM] #REF1))
PRODUCT (NULLRELATION(#REF1))))
```

Unit Test Samples

Language	Query	SQL Equivalent
RA	SELECT[(percentage<10 OR percentage>99.12) AND (countrycode='ANT' OR countrycode='BFA') AND isofficial=TRUE]countrylanguage	SELECT DISTINCT * FROM countrylanguage WHERE (percentage<10 OR percentage>99.12) AND (countrycode='ANT' OR countrycode='BFA') AND isOfficial=true
RA	PROJECT[country.name]((SELECT[continent='Africa']country) NATURALJOIN (SELECT[countrylanguage.language='Greek']countrylanguage))	SELECT DISTINCT name FROM country NATURAL JOIN countrylanguage WHERE continent='Africa' AND language='Greek'
ERA	PROJECT[country.name,(surfacearea * 10) - 1298.4343 AS sa ,lifeexpectancy/lifeexpectancy * 10 AS le] ((SELECT[surfacearea<600000 AND surfacearea>150]country) NATURALJOIN (SELECT[countrylanguage.language='Greek']countrylanguage))	SELECT name,(surfacearea*10)-1298.4343 as sa, lifeexpectancy/lifeexpectancy*10 as le FROM country NATURAL JOIN countrylanguage WHERE surfacearea<600000 AND surfacearea>150 AND language='Greek'
SQL	SELECT country.name ,countrylanguage.language, countrylanguage.isofficial,country.continent FROM country LEFT JOIN countrylanguage ON country.code=countrylanguage.countrycode AND percentage>0.5 WHERE population>20000000	

Appendix B

Tutorial Solutions

Relational Algebra

Question 1* Create a list that contains only the item IDs and descriptions.
Header:item_ id,description
Solution
PROJECT[item_id,description]item

Question 2* Create a list of all items with their cost price and barcode EAN.
Header:item_id,cost_price, barcode_ean
Solution
PROJECT[item_id,cost_price,barcode_ean](item NATURALJOIN barcode)
or
PROJECT[item.item_id,item.cost_price,barcode.barcode_ean] (SELECT [item.item_id =barcode.item_id](item PRODUCT barcode))

Question 3* List all phone numbers in town *Bingham*.
Header: phone
Solution
PROJECT[phone] (SELECT[town='Bingham'](customer))

Question 4* List the first and last name of customers that purchased the item with description *Linux CD*.
Header: fname,lname
Solution
PROJECT[fname,lname](customer NATURALJOIN (orderinfo NATURALJOIN (orderline NATURALJOIN (SELECT[description='Linux CD']item))))

Question 5 Semijoin $\mathbf{R} \ltimes \mathbf{S}$ is the derived operator that returns all tuples in R for which there is a tuple in S that is equal on their common attribute names. Using the derived operator definition language in QV create a definition for the right *semijoin* $\mathbf{R} \ltimes \mathbf{S}$ that performs the same operation with the only difference that the result only contains attributes from S instead of R.

Hint: Find the definition "SEMIJOIN" to see how $\mathbf{R} \ltimes \mathbf{S}$ *is defined.*

Solution

#REF2 NATURALJOIN (PROJECT[#REF1 INTERSECT #REF2] #REF1)

SQL

Hint: To solve the next two questions try to use the SUM aggregate and GROUP BY where applicable

Question 1 Find the total number of items in stock.

Solution

SELECT SUM(quantity) FROM stock

Question 2 Calculate the total sell price of all items for every order made.

Header:orderinfo_id,Total

Solution

SELECT orderinfo.orderinfo_id, SUM(item.sell_price*orderline.quantity) AS Total FROM orderinfo JOIN orderline ON orderinfo.orderinfo_id=orderline.orderinfo_id JOIN item ON item.item_id = orderline.item_id GROUP BY orderinfo.orderinfo_id ORDER BY orderinfo.orderinfo_id

Extended Relational Algebra

Question 1 List the profit (cost_price - sell_price) for all items.

Header:item_id,difference

Solution

PROJECT[item_id, sell_price-cost_price]item

Question 2 Find the total quantity of Wood Puzzles sold.

Solution

SUM[quantity](SELECT[description='Wood Puzzle'](orderline NATURALJOIN item))

Question 3 List the number of barcodes assigned to each item. If no barcode is available for an item then the result will show an entry with count 0.

Header:item_id,barcode_count

Solution

GROUPBY[item.item_id,COUNT[barcode_ean]] (item LEFTOUTERJOIN barcode)

List of Figures

2.1 WinRDBI Interface 21
2.2 iDFQL Interface 22
2.3 RELATIONAL Interface (Annotated) 23
2.4 RALT Interface 24
2.5 Aqua Data Studio Interface 25

3.1 QV Architecture 28

4.1 Unary and Binary Abstract Operation Trees 32
4.2 Operation Tree Example 33
4.3 Evaluators Collaboration Diagram 34
4.4 Natural join tree after initial parsing 36
4.5 Natural join tree after expansion to original definition 36
4.6 Operation Tree Example with statistics 37

5.1 QV Interface 42
5.2 Relation Explorer 43
5.3 New Relation Dialog 44
5.4 Import from database dialog 44
5.5 Two tab containers 45
5.6 Query Panel 46
5.7 Derived Operator Editor 47
5.8 Intermediate Results View 48
5.9 Expanding Derived Operators 48
5.10 Selecting Rows for Lineage 49
5.11 Showing Lineage at a leaf node 49
5.12 Chart View 50
5.13 Console 51
5.14 Execution Cost View 52
5.15 Optimiser Interface with execution statistics 53
5.16 Maximum memory usage per tree 54
5.17 Execution time per operation 55

5.18 Memory usage per operation 55

6.1 QV Packages 56
6.2 Node class of project 58
6.3 ANTLR syntax definition of project 59
6.4 Project Evaluator code 60
6.5 Dynamic Operator construction and evaluation 62
6.6 Extended relational algebra grammar in ANTLRWorks 64
6.7 Calling the correct execution controller (QV.Controlling.SetController.cs) 66

7.1 Memory usage graph 69

8.1 Store Database Schema 71
8.2 Exercise Sheet Time 74
8.3 Exercise Sheet Correct Answers 75
8.4 QV Welcome Screen 76