Translating between SQL and Tutorial D

Ahmed Yusuf Almulla

Dissertation Supervisor: Dr Nigel Stanger

A dissertation submitted as a partial fulfilment for the degree of Bachelor of Applied Science with Honours in Software Engineering

UNIVERSITY of OTAGO

SAPERE AUDE

Te Whare Wānanga o Otāgo

UNIVERSITY OF OTAGO

Department of Applied Science

Acknowledgments

I would like to take this opportunity to thank my supervisor Dr Nigel Stanger for the great support and guidance he provided me throughout the course of my dissertation. I would also like to appreciate the contribution of Dr Colin Aldridge in running the research methodology seminars that provided me with a great deal of useful knowledge. I thank Professor Martin Purvis for being the course coordinator of this valuable course and showing his support. I also thank my friend, Abdulrahman Shams, for the assistance he provided me in the implementation of my experiment which led to the successful completion of my dissertation.

This work is dedicated to my family, friends and Software Engineering colleagues who stood beside me throughout the course of my dissertation and showed their support.

Finally, my greatest gratitude goes to my dear parents who put their faith in me and supported me in my studies.

Abstract

The relational model of data has a strong influence on the database field and cannot be discarded for any future direction regarding the development of databases (Date and Darwen, 2007). Two relational database languages that are associated with this model are the Structured Query Language (SQL) and Tutorial D. The purpose of this dissertation is to translate between these two languages so that SQL clients can interoperate with a Tutorial D database via SQL. Through studying the literature, it is shown that such translation is important. It will allow the most popular relational database language (i.e. SQL) to interoperate with a true relational database language (i.e. Tutorial D) utilising its features to solve some of the deficiencies that exist in SQL.

The main method conducted for this research is through implementing a translation layer between SQL and Tutorial D using direct string manipulation. The translation logic utilises some of the available methods for string manipulation to manipulate the executed SQL statements and translate them to the corresponding Tutorial D statements. The translation layer then takes care of executing the resultant Tutorial D statements on a chosen Tutorial D database called Rel.

Testing the translation layer consisted of three steps. The first one is to test the translation layer against valid SQL statements to study the feasibility aspect of such translation. It is shown that all the chosen SQL statements for this step pass the test. The second step is to test the translation layer as a solution to the SQL deficiencies defined. It is shown that the test cases that are defined in this step pass the test, and thus, the translation layer solves the deficiencies of SQL. Finally, the translation layer is tested against robustness. All the invalid SQL statements defined in this step are not translated and suitable error messages are shown.

Finally, the dissertation concluded with proving that the translation between SQL and Tutorial D is feasible and can be a suitable approach to solve SQL deficiencies while still using SQL.

Table of Contents

List of Figures

List of Tables

1. Introduction

Since the introduction of the relational model of data (Codd, 1970), relational database management systems came into existence and proved to be dominant in the database field. These are software products that can be used in creating and managing relational databases. These software products are all based on relational database languages that give clients the ability to interact with data that are stored in relations (Silberschatz, Korth and Sudarshan, 2002). There are many relational database languages in the market but the most popular and standardised one is the *Structured Query Language* (SQL). The purpose of this dissertation is to investigate the feasibility and needs for implementing a translation layer between SQL and one of the new relational database languages called *Tutorial D*.

1.1 Research Problem

There are many deficiencies and flaws surrounding the use of SQL as a relational database language. In an attempt to avoid using SQL, Tutorial D was introduced as a new relational database language that addresses SQL deficiencies and introduces new features which realise the full potential of the relational model of data (Date and Darwen, 2007). However, due to the popularity of SQL, adopting Tutorial D can be difficult. Hence, the need to find a suitable mechanism to address these issues is crucial. One suggested solution is to form a translation layer between SQL and Tutorial D in order to allow legacy SQL clients to interoperate with Tutorial D databases. However, this translation layer has not been implemented yet, and hence, the need for more investigation surrounding the feasibility and needs of such translation.

1.2 Research Objectives

Given the research problem, this dissertation aims to implement a translation layer between SQL and Tutorial D allowing legacy SQL clients to interoperate with Tutorial D databases. The research will first study the relevant literature to show the importance of such a translation and the key factors that drive the need for implementing it. After completing the implementation of the translation, it will be investigated in terms of showing how SQL deficiencies can be solved through interoperating with Tutorial D via SQL.

1.3 Research Hypotheses and Scope

There are few questions to be addressed in this dissertation: Is the translation layer between SQL and Tutorial D a feasible approach to preserve the interoperability of the two languages? How can such translation be implemented? How important is this translation layer and how can it solve the problems of SQL deficiencies?

The aim of the dissertation is to answer the previous questions. However, for the purpose of this dissertation, the translation layer will only deal with a small subset of the relational language to see if the translation layer is achievable in its basic form. The subset that is investigated for translation is provided in Section 8.1 when describing the implementation of translation.

1.4 Research Importance

If the translation layer is achievable, it will give the most popular and widely used relational language, which is SQL, the ability to interoperate with next generation database systems utilising their advantages of being true relational and solving SQL deficiencies. Therefore, this research may lead to a better approach of manipulating relational databases.

1.5 Research Methodology

The methodology undertaken for this research consists of a few stages in order to address the research objectives described earlier. These stages are:

- reviewing the literature to study the importance of the research;
- reviewing the literature to study previous attempts made at translation;
- implementing the translation layer; and
- testing the translation layer.

1.5.1 Literature Review: The First Part

The first stage of the methodology is a literature review relevant to this research. The purpose of the literature review is to provide sufficient background information that can aid in understanding the foundations that this research was based upon. In addition, the research importance will be illustrated through reviewing the literature. This spans across multiple sections and it is expected to result in the following findings:

- SQL is the most widely used language for interacting with relational databases;
- SQL has some deficiencies in complying with the relational model of data; and
- Tutorial D is the next generation relational database language to address SQL deficiencies.

After showing the above findings, a suggestion is provided to build a translation layer between SQL and Tutorial D due to its importance.

1.5.2 Literature Review: The Second Part

The second part of the literature review is a shorter one. The purpose of it is to study the feasibility of translation through studying two previous attempts that were conducted between SQL and other database languages. One of them is between SQL and *AmosQL* and the other one is between SQL and *XQuery*. The outcome of this review is to show:

- the similarities of the attempt made at translation to the one that is conducted for this research;
- the differences of the attempt made at translation from the one that is conducted for this research; and
- the success of the attempt made at translation.

1.5.3 Implementing the Translation Layer

The next stage of the methodology is to implement the translation layer between SQL and Tutorial D after showing how important it is. The implementation goal is achieved through building an interface that can accept SQL statements and execute them on a Tutorial D database after conducting the translation. For this purpose, a third generation programming language called *Java* is used to perform the translation logic.

1.5.4 Testing the Translation Layer

Finally, the translation layer that was implemented is tested. The purpose of testing is to gather results that can show:

- valid SQL statements for the subset chosen can be executed on a Tutorial D database;
- invalid SQL statements for the subset chosen are not allowed to be executed on a Tutorial D database; and
- the translation between SQL and Tutorial D solves the deficiencies of SQL that are investigated in this dissertation.

1.5.5 Structure of the Report

This dissertation has 11 main sections. Section 1 provides a general introduction to the research topic, its objectives, scope and importance. It also outlines a suitable methodology that is followed for the course of this dissertation. In Section 2, sufficient background material is provided to the reader. Section 3, 4 and 5 forms the first part of the literature review. The issue of SQL legacy and standardisation is covered in Section 3 while the deficiencies of SQL are explained in Section 4. Tutorial D and its true relational features that solve those SQL deficiencies are explained in Section 5. In Section 6, an analysis of the findings in the first part of the literature is conducted. Section 7 comes next and it forms the second part of the literature review by studying the previous attempts made at translation. Section 8 onwards form the original work made in this dissertation. In Section 8, the implementation of the translation layer between SQL and Tutorial D is explained.

Section 9 comes next and its purpose is to test the translation layer implemented and show the results of testing. An analysis and discussion of these results is conducted in Section 10. Finally, Section 11 of the dissertation outlines some conclusions and implications regarding future work on the translation layer built for this dissertation.

2. Background

This section provides detailed information of the relevant background related to the study of translation between SQL and Tutorial D. It will first provide a background on the relational model of data and its importance in the database field. Getting an overview on the relational model of data is important because both of SQL and Tutorial D are based upon this model. Finally, some background information will be provided for each of SQL and Tutorial D. For those sections, only the relevant material regarding the history and importance of those languages will be provided. Detailed analysis for some of the issues regarding the two languages is provided in later sections.

2.1 The Relational Model of Data

This dissertation is based on the widely accepted model of data called *The Relational Model* which was published by Codd (1970). It came out as a motive to separate the internal representation of data from the users of large data banks. Large data banks are known nowadays as databases. For the purpose of this dissertation, the term *databases* will be used when referring to large data banks.

As Codd suggests, the relational model can protect users from any knowledge related to the physical data storage for databases. The following subsections describe the relational model structure, its advantages and importance in the database field.

2.1.1 The Relational Model Structure

The relational model of data, as described by Codd, gives an abstracted view of the data without any notion of its physical representation. This means that users are not required to know the internal structure of the data and how they are stored. This abstraction is achieved through the use of relations consisting of tuples that form the necessary data an application program needs to interact with (Codd, 1970).

To explain, in the relational model of data, a relation can be defined as unordered set of elements. These elements are known as tuples. Each tuple is defined as a set of attributes. Each attribute in a tuple contains a value that is drawn from a domain. Finally, a domain is a pool of all the possible values that an attribute can have (Codd, 1990). However, an alternative description of the relational model structure exists. To illustrate, it is noted that users perceive the structure of the relational model of data in a different manner. Date (1988) describes users' perception of the relational model as a collection of tables which contain rows that have some column values. Those column values represent data in the relational model.

The former description of the relational model uses the correct intended terms that were originally defined by Codd (1970). These terms are used by Tutorial D when implementing the relational model of data. Nevertheless, SQL uses the latter description of the relational model. To avoid confusion, this dissertation will use the terms *relation, tuple and attribute* when describing the relational model of data and the Tutorial D implementation of it. On the other hand, the equivalent terms of *table, row and column* will be used when describing the SQL implementation of the relational model of data.

In addition, the relational model of data must abide by some integrity rules in order to ensure a consistent state of the model which complies with the relational model specification that was published by Codd. For the purpose of this dissertation, one such integrity rule is crucial. This is the *Entity Integrity* which is about ensuring that every tuple in the same relation is unique. In other words, there must be no two tuples in the same relation having the same values across all attributes (Burns, 1990).

2.1.2 Advantages of the Relational Model

The relational model provides several advantages to the users of databases as Codd discussed. Through separating the internal representation of data, application programs can remain unchanged when any change should happen to the internal representation of data. Furthermore, as Codd discussed, the relational model of data acts as a suitable solution that replaces other models of data which are inadequate because of their dependency on the physical internal structure of data. Such models include the network models and tree-structured files. Thus, separating the internal physical representation of data from the logical one is very crucial because of the different kinds of physical dependencies that an application program may end up having. For example, the ordering of elements that are stored in the database may affect the functionality of the application program when a change happens to the ordering of those elements. This is referred to as ordering dependence. Similarly, different physical dependencies besides the ordering of elements may affect the application program and these may include indexing and access path dependencies. All of those types of dependencies which are provided by Codd show how an application program can largely depend on the physical representation of data which is inefficient and can cause the application program to fail. Hence, the need for a separation between the physical representation of data and the logical one is needed so that application programs can interact with the logical representation of data without the need to know about its physical structure and the changes made at that level.

2.1.3 Relational Model Importance

Since Codd defined the relational model of data and throughout the years, it became evident, through the wide collection of the many database management systems that are based on the relational model of data, that the relational model has a strong impact on the database field. This is supported by a later argument made by Codd (1981) who made a clear statement on how the relational model aids in conducting a sensible logical database design before implementing the database. Thus, as Codd believes, the relational model is the foundation for relational database systems which took off as being the dominant database products in the market nowadays (Codd, 1990).

In addition, Researchers in the database field acknowledge the relational model of data to be highly relevant to database theory nowadays. According to Date and Darwen (2007), the relational model of data has a strong foundation that will remain in use for the upcoming future. This is because of the numerous advantages of the relational model of data in the database field. Apart from the advantages mentioned earlier, Date (1988) points out many other advantages that emphasise on the importance of the relational model of data. One of the advantages is the simplicity that the relational model of data provides to the users in terms of usability. Usability refers to the ability of unskilled users to use relational model systems for the purpose of managing data and obtaining results without the need of external help from an administrator (Date, 1988).

Therefore, it can be seen that the relational model of data is important to the database field. However, its importance cannot be utilised unless there is some form of a commercial exploitation of the model. In order for the relational model to be implemented and used commercially, some database query languages came out which conformed to the relational model and gave a specification that database designers and users can use for interacting with the databases. The database management systems that utilise the relational model of data are subsequently known as relational database management systems and the database query languages used for those types of systems are known as relational database languages. This dissertation is related to two of those relational database languages which are the Structured Query Language (SQL) and Tutorial D. The following subsections will provide a background on these two relational database languages.

2.2 The Structured Query Language – SQL

SQL is a language used for interacting with relational databases that are based on the relational model of data. Interaction in this context means the ability to create, access and manage data (Date and Darwen, 1997).

It all began when a company called *IBM* conducted a research in the field of introducing a suitable language to interact with relational databases. As a result, SQL was introduced and increasingly became popular as commercial database products used it in the early and mid 1970s. These include *non-IBM* commercial products, such as *ORACLE* which was developed by *Oracle Corporation* (Date and Darwen, 1997). As Date and Darwen (1997) state, SQL has become the de facto standard for interacting with databases. This is evident from the wide variety of relational database management system products that are based on SQL, nowadays. In addition to ORACLE, these products include *IBM's DB2* and *Microsoft SQL Server* (Silberschatz et al., 2002).

For the purpose of this dissertation, some issues that are related to SQL will be provided in later sections.

2.3 Tutorial D

Tutorial D is a relational language proposed by Date and Darwen (2007) for the purpose of forming a solid foundation for future database management systems.

It all started when Date and Darwen observed some of the trends which attempted integrating object and relational technologies but in an ill-defined manner. The first of these attempts was the *Object-Oriented Database System Manifesto* which was proposed by Atkinson et al. (1989). It considered object-oriented database systems to be theoretically and experimentally useful, and thus, need a common data model to formalise those systems. According to Atkinson et al. (1989), the reason behind this is that object-oriented database systems lacked a common data model before the introduction of the Object-Oriented Database System Manifesto. However, Date and Darwen (2000) criticised this approach to be ignoring the relational model of data, and thus, not suitable for a future direction of database management systems to be moved along. This is due to the importance of the relational model of data, as described earlier.

The second attempt came along after the Object-Oriented Database System Manifesto and was called the *Third-Generation Database System Manifesto*. This manifesto realised the need for evolving the original relational model of data to treat objects as any other data type of the model (Stonebraker et al., 1990). As this assertion suggests, dealing with objects is important to the Third-Generation Database System Manifesto because of their essential need in supporting business data processing applications. As the authors claim, objects are necessary in two scenarios. One of those scenarios is to store certain types of data elements, such as images, which are difficult to be stored in the original relational model of data. The second scenario is that objects are necessary to aggregate all of the related data elements into a single object in which it can be manipulated easily by application programs (Stonebraker et al., 1990).

Although this manifesto acknowledged the relational model of data and evolved it by including the ability to store objects, Date and Darwen (2000) criticised it to be accepting SQL as a relational language that is suitable for the evolved relational model of data. This is because Date and Darwen (2000) believe that SQL is considered to be a perversion of the relational model of data because of the many deficiencies SQL has, and thus, it must be rejected when moving forward. Section 4 provides a detail explanation of those SQL deficiencies.

In short, Date and Darwen (2000) made some assertions based on the above observations. The first assertion made is that integrating objects with the relational model of data is necessary in the light of the recent events which include the erroneous attempts that came from the previous two manifestos. Secondly, for any attempt to move forward and evolve the relational model of data, SQL must be rejected because of its deficiencies.

As a result, Date and Darwen proposed a third attempt of directing the next generation of database management systems. A manifesto was formed and was called *The Third Manifesto* with the purpose of forming a theoretically correct mechanism of building object/relational database management systems which evolve from the original relational model of data but not replace it because of its importance and relevance to the database field (Date and Darwen, 2000). Therefore, the Third Manifesto builds on the original ideas of the relational model of data and extends them for the purpose of correcting the current trends of integrating object and relational technologies. As Date and Darwen (2000) claim, the relational model of data needs no extension per se but needs another relational language to support the object/relational features without using the flawed SQL.

Subsequently, a new database programming language called Tutorial D was defined to replace SQL. The purpose of Tutorial D is to form a solid foundation in which the Third Manifesto can be built upon. This foundation must be originating from the roots of the relational model of data that was proposed by Codd (1970) without any existence of SQL (Date and Darwen, 2000). This is due to the many deficiencies that SQL has which are described in Section 4.

In terms of Tutorial D as a database language, it can be described as "a computationally complete programming language with fully integrated database functionality" (Date and Darwen, 2007, p.93). It is intended for education in order to correct the common misconceptions provided earlier. In addition, it supports the relational and object features of the manifesto through a list of features. For the purpose of this dissertation, only the relevant relational features that address SQL deficiencies and are considered to be proscriptions to the Third Manifesto are provided (refer to Section 5).

3. SQL Legacy and Standardisation

The first key factor that is related to this dissertation and its importance is SQL standardization and how it influenced the popularity of SQL as a relational database language. This section outlines how standardisation contributed towards SQL legacy of being the most popular relational database language.

3.1 Standardisation in General

In general, a standard can be defined as “an approved model to be used for comparison or judgement, and of established authority” (Soderstrom, 2002, p.1048). As the definition implies, a standard can develop a formal agreement between several parties to use a common model for the purpose of comparison and judgement.

Furthermore, other researchers in the standards field define a standard as “a specification (or perhaps a product) that is widely accepted at least by some identifiable market segment” (Eisenberg and Melton, 1998, p.54). As Soderstrom (2002) believes, this can help businesses in synchronising their processes for co-operation purposes and connecting their systems in a unified manner by the use of a standard.

However, there could be many standards that an industry can choose from for any given problem. The elements that characterise the most useful standard to be chosen are outlined by Eisenberg and Melton (1998) who claim that an industry tend to choose the most relevant standard that solves the given problem in a good but not necessarily perfect manner and is available when needed. This is evidently true in the information technology industry considering the latest standards that have been adopted by businesses which include those standards that are developed by formal standards organisations. *The American National Standards Institute* (ANSI) and *The International Organisation for Standardisation* (ISO) are two of the most recognised formal standards organisations (Eisenberg and Melton, 1998).

In addition, Eisenberg and Melton (1998) point out that ANSI and ISO standards organisations can make the adoption of the developed standards faster. This is evident through looking at how ANSI and ISO standardised SQL which made it popular and widely used. The following subsection details the standardisation of SQL.

3.2 The SQL Standard

Standardising SQL can be described as the process of making SQL the standard relational database language that can be used in interacting with databases that are based on the relational model of data. Date and Darwen (1997) describe this process in two steps. The first step was making SQL the standard language for interacting with relational databases. This was achieved through the ANSI standard in 1986. The second step was making the SQL standard of ANSI accepted internationally and this was done by ISO in 1987.

As a result, SQL became widely popular and supported by many relational databases as the chosen relational language used for interaction. This wide acceptance of SQL can be attributed to ANSI and ISO, in which it was mentioned in the previous section that those two standards organisations can make the adoption of the developed standards faster (Eisenberg and Melton, 1998). In addition, it was also mentioned in the previous section that standardisation is desirable for many cases. In short, these include using a common model, product or specification to co-operate in a unified manner and solving a common problem in a good, relevant and timely manner.

Applying those principles to the SQL standard, it can be seen that the ANSI and ISO SQL standards can give vendors, who are developing a relational database management system but are hesitant of which relational language to choose from, the choice of selecting SQL. This is due to what the SQL standard will provide from specifying a common relational database language which can be supported by multiple vendors, and thus, can lead to the increase of demand for the relational database management systems that support SQL. It can also be seen that, despite SQL deficiencies which are going to be outlined in later sections, SQL solves the common problem of interacting with relational databases in a good, relevant and timely manner.

Those points are also supported by Date and Darwen (1997) who claim that because of the common relational database language supported by multiple vendors through the use of the SQL standard, many advantages are guaranteed. These include:

- Reduced training costs because of the ability to switch between a relational database management system and another without the need for retraining. This is due to the use of a common relational database language which is SQL.
- Intersystem communication between different systems. Because of the SQL standard, different relational database management systems can easily communicate with each other.
- It is widely perceived that customers would go for the relational database language that has been standardised to remove the complexity of choosing the language and concentrating on choosing the best relational database management system that meets their needs.

Hence, it is evident that SQL became popular as a relational database language because of standardisation.

A final point regarding the popularity of SQL is that one might think that standardising SQL by ANSI in 1986 and ISO in 1987 may have led other subsequent relational languages to be standardised in a similar manner to SQL. However, throughout the years, the original ANSI and ISO SQL standards were expanded to include the numerous features that SQL introduced. Thus, the ANSI and ISO standards were modified as needed (Date and Darwen, 1997). According to Eisenberg and Melton (2000), three major editions of the SQL standard were published during the period of 1986 and 1999 which are *SQL-86*, *SQL-92* and *SQL: 1999* standards. A subsequent edition of the SQL standard was also added in 2003 and was called *SQL: 2003* (ISO, 2003). This contributed towards keeping the ANSI and ISO SQL standards in business and avoiding their obsoleteness. Therefore, as Date and Darwen (1997) point out, this guaranteed the long lifetime of SQL and ensured its legacy.

3.3 Summary

The popularity of SQL and its legacy of being the most widely used relational database language can be attributed to the standardisation of the language. It is important to take the legacy factor of SQL into consideration when attempting to introduce new relational database languages. This is because of the difficulty of adopting any new relational database language that comes after SQL because of its legacy.

4. SQL Deficiencies

The purpose of this section is to provide a detailed explanation of some of the deficiencies and flaws that exist with SQL. The deficiencies provided are by no means a complete list of all the SQL deficiencies as only those ones that are relevant to the purpose of this dissertation were chosen. For this purpose, the SQL deficiencies that are described were chosen on the basis of, firstly, if there are any violations in the relational model of data due to the behaviour of SQL, and secondly, if the Third Manifesto addressed those deficiencies in the list of proscriptions that Tutorial D conforms to.

4.1 Basic Scenario

For the purpose of illustrating the deficiencies of SQL, a basic database with one table was created. The purpose of this table is to store the records of employees in a company. The data requirements for this table are illustrated in Figure 1.

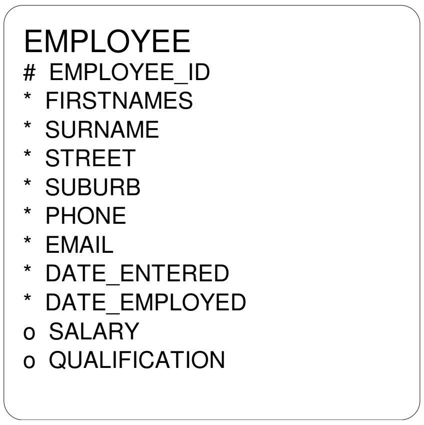


Figure 1: Employee Entity Relationship Diagram

The above figure is an *Entity Relationship Diagram* (ERD) created using Barker notation (Barker, 1990). It shows an entity called Employee corresponding to a table with the same name when implemented with all the columns defined for that table. All of the data stored in those columns are mandatory as they must be entered when the employee rows get created. However, for the purpose of illustrating SQL deficiencies, the Salary and Qualification columns are optional meaning that it is not mandatory to enter their values when the employee rows get created. The SQL code used to create the Employee table is provided as an Appendix (refer to Appendix 1). In addition, using an insertion script written in SQL, the Employee table was populated with sufficient data (refer to Appendix 2).

4.2 Overview of the Deficiencies

Many researchers in the field of the relational model of data claim that SQL has serious flaws that violate the relational model. For example, Codd states that "SQL departs significantly from the relational model, and where it does, it is clearly SQL that falls short" (Codd, 1990, p. 372). In addition, Date and Darwen claim that SQL "fails to realize the full potential of the relational model" (Date, 1984, p.270).

To prove those claims and as mentioned earlier, a selected number of SQL deficiencies were chosen for the purpose of this dissertation. These are:

- the appearance of duplicate rows in relations;
- the use of NULL for specifying unknown values; and
- the reliance on column ordering.

4.2.1 First SQL Deficiency: Duplicate Rows

In the relational model of data, a duplicate row can be defined as any identical row that has the same values for all the columns when compared to another existing row in the same relation.

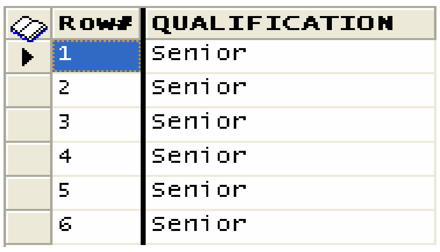
As Codd (1990) states, duplicate rows are a violation of the relational model of data and must be avoided. The basic problem of duplicate rows, as Codd describes, is that duplication may lead to the confusion of users when interpreting the meaning of duplicate rows. This is because duplicate rows are considered to be unwanted redundancy of the data as it can be difficult to form a common meaning of those redundant data. Date (1990) supports this claim by stating that duplicate rows may confuse users on which rows to choose from when retrieving a set of duplicate rows as they will all appear the same to the users. Above all, duplicate rows are considered to be a violation of the Entity Integrity rule of the relational model which was mentioned earlier (refer to Section 2.1.1).

Hence, duplicate rows are considered to be a serious violation to the relational model of data. In this regard, there is a flaw with SQL as it can allow duplicate rows to appear to the users. This is illustrated through the following example using the basic scenario described earlier:

A possible case that SQL users may need to do with the Employee table is to retrieve the "Qualification" column to see all the possible qualifications that the employees have. To achieve this purpose, a SELECT query on the Qualification column will give the required result. This will return a relation containing the different values of qualifications that are found in the Employee table. The query to be run for this is:

SELECT Qualification FROM Employee;

However, when running the above query on the database created for the scenario, the result is a table which contains duplicate rows because of all employees having the same qualification. This is shown in Figure 2.



ROW#	QUALIFICATION
1	Senior
2	Senior
3	Senior
4	Senior
5	Senior
6	Senior

Figure 2: SQL Deficiency - Duplicate Rows

Therefore, this is a violation of the relational model as mentioned earlier. Ideally, the result of the above query should return a table with duplicate rows being eliminated so that users would not get confused with the meaning of different duplicate rows when they all mean the same. Thus, duplicate rows can appear when using SQL and this is considered to be a deficiency. This argument is also supported by Date and Darwen (2007) as they state that duplicate rows can appear with SQL, and when they do, SQL tables are not considered to be true relations because of the violation of the relational model as Codd (1990) points out.

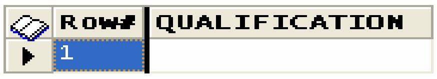
4.2.2 Second SQL Deficiency: Specifying Unknown Values as NULL

In relational databases that use SQL, NULL can be referred to as any value that is recorded as unknown in the database due to the incompleteness of information (Date, 1983). For example, it can be possible that a record contains one or more values that are unknown at the time of inserting the record into the database, and thus, those unknown values can be assigned NULL to solve the problem.

This can be illustrated through using the Employee table from the basic scenario described earlier. For instance, when inserting a new record in the Employee table, it is possible for the "Qualification" value to be unavailable for the new inserted employee record. This is supported in the database design, as mentioned in the scenario description, as the Qualification column is defined as optional. Thus, through the use of NULL when specifying the Qualification value, it is possible to insert a new employee record using the following INSERT statement:

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (7, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, NULL);

Consequently, when retrieving the Qualification of the new record that was inserted, the result will show no value for the retrieved qualification, which is achieved through the use of NULL; as shown in Figure 3.



```
Row# QUALIFICATION
1
```

Figure 3: NULL Value for Qualification

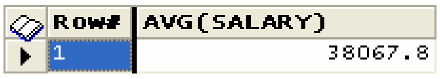
However, specifying unknown values as NULL is considered to be a deficiency in SQL. This is because of the many anomalies that can appear when using NULL as stated by Date (1983). Those anomalies are outside the scope of this dissertation but it is worthwhile to illustrate one of them for the purpose of showing that specifying unknown values as NULL is considered to be a deficiency with SQL.

For this purpose, the chosen anomaly is pointed out by Date (1983) by claiming that values that are specified as NULL are always eliminated from the argument to a built-in function. Examples of built-in functions include SUM and AVG. The former calculates the sum of the values of the column passed as an argument to the function whereas the latter calculates the average of those values.

For example, one query that may arise after inserting the employee records is to calculate and return the average value of employees' salaries. Using the AVG built-in function, this can be achieved using the following SQL statement:

SELECT AVG(Salary) FROM Employee;

The result of the above SQL statement will return a single value showing the average of employees' salaries; as shown in Figure 4.



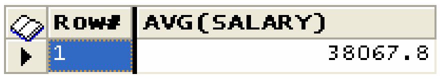
```
Row# AVG(SALARY)
1        38067.8
```

Figure 4: Average of Salaries

However, when a new employee is recorded, it can be possible that the salary value is unknown, and when using SQL, this can be achieved through the use of NULL when specifying the Salary value, as follows:

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (8, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', NULL, 'Senior');

As a result, the new employee must be counted in for calculating a new average value using the same SQL statement used earlier. It is expected that the new average value will be different from the previous one because of the addition of a new employee. However, the average value is found to be the same as the old one; as shown in Figure 5.



```
ROW#  AVG(SALARY)
1         38067.8
```

Figure 5: The New Average of Salaries

This can only mean that due to the NULL value of Salary for the new employee, it was eliminated from the argument to AVG function. This behaviour of SQL is considered to be inconsistent, as Date (1983) describes, as there is no guarantee that all of the records will be considered for the AVG function because of the existence of values specified as NULL. Therefore, this can lead to undesired incorrect results.

4.2.3 Third SQL Deficiency: Reliance on Column Ordering

In the relational model of data, specifying the column order is not important when having to retrieve their values or insert new values in them. Date (2003) supports this argument by claiming that a proper relational language pays no attention to column ordering. The assertion made by Date is valid because of the fact that a "proper" relational language means that the language itself is a concrete realisation of the relational model of data and conforms to it. Having said that, as the relational model of data does not require columns in a relation to be specified in a particular order (Codd, 1990), a proper relational language must do the same.

However, SQL falls short in this regard by violating the principle of column ordering. This is because in some cases, users must remember the order of columns from left to right when using SQL (Date, 2003).

To illustrate, when inserting a new Employee record using SQL, it is possible to only specify the column values without having to specify the column headings, as follows:

INSERT INTO Employee VALUES (9, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');

However, when doing this, the values that are entered rely on the original order of the columns. The original order of columns is the "left-to-right" order they were specified in when creating the Employee table.

Consequently, this can have some negative outcomes on users as Codd (1990) points out. This includes putting the burden on users to remember the correct order of columns which can get difficult when having a large number of columns in a relation. In addition, Date (2003) adds to the negative outcomes of this SQL flaw by pointing out that relying on column ordering when doing an insert can lead to incorrect results if the original ordering of columns changed but the order of columns in the SQL INSERT statement did not.

Fortunately, SQL defines a solution for the above problem by allowing users to specify the column names instead of relying on column ordering. For example, the previous SQL statement can be written as:

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (9, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');

Thus, column names can be specified in the SQL INSERT statement and this can be done in any order. For example, the previous SQL statement can also be written as follows with a different column ordering:

INSERT INTO Employee (Employee_ID, Surname, Firstnames, Suburb, Street, Email, Phone, Date_Entered, Date_Employed, Salary, Qualification) VALUES (9, 'Smith', 'John', 'Dunedin', '11 Ethel Benjamin Place', '-', '(03) 497-0752', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');

However, the choice of not specifying the column names as originally described is still available for use and is considered to be a flaw in the language. In addition to the negative outcomes that this flaw has, it is considered to be a violation of the relational model of data as described earlier and must be avoided.

4.3 Summary

It is evident that SQL is deficient in conforming to the relational model of data. Some of these deficiencies include the appearance of duplicate rows in relations, the use of NULL for specifying unknown values, and the reliance on column ordering. Hence, the need for a new relational database language for any step to move forward is crucial.

5. Tutorial D: A True Relational Database Language

Tutorial D is a relational database language that was proposed as an outcome of the Third Manifesto to form a solid foundation for future database systems in which objects and relational features can be integrated and utilised in a correct manner (refer to Section 2.3). For this reason, it is called a 'true' relational database language (Date and Darwen, 2007). For the purpose of this dissertation, only the relevant relational features that SQL has deficiencies in and addressed by Tutorial D are described. Therefore, the provided features of Tutorial D will not include any object or additional relational features because they are outside the scope of this research.

This section will give an overview of the relational features of Tutorial D that address the SQL deficiencies that were mentioned.

5.1 The True Relational Features

For the purpose of illustrating the true relational features of Tutorial D, the same scenario that was used for illustrating the SQL deficiencies is used (refer to Section 4.1). Using Tutorial D, a relation called Employee is created and then populated with some tuples (refer to Appendix 3 & 4).

The features of Tutorial D that are provided in this section show how Tutorial D addresses the deficiencies of SQL mentioned in Section 4. According to Date and Darwen (2007), these include:

- no duplicate tuples;
- no NULLs; and
- no attribute ordering.

5.1.1 First True Relational Feature: No Duplicate Tuples

Date and Darwen (2007) point out that Tutorial D prevents the appearance of duplicate tuples in a relation. This can be illustrated through applying the same scenario used earlier when showing the SQL deficiency of duplicate rows (refer to Section 4.2.1).

To illustrate, when retrieving the Qualification value, the following Tutorial D statement can be used:

Employee {Qualification}

This returns a relation with one tuple as follows:

RELATION {Qualification CHAR} {
  TUPLE {Qualification "Senior"}}

Although there are many other tuples with the same Qualification value, returning them in the result will lead to duplicate tuples. Therefore, Tutorial D eliminated those duplicates before showing the final result of the retrieval. Thus, Tutorial D addresses the SQL deficiency of duplicate rows and conforms to the relational model in this aspect.

5.1.2 Second True Relational Feature: No NULLs

All attributes in Tutorial D have values and those cannot be NULL (Date and Darwen, 2007). This can be illustrated through applying the same scenario used earlier when showing the SQL deficiency of specifying unknown values as NULL (refer to Section 4.2.2).

To illustrate, when inserting a new employee record with a NULL salary, as follows:

INSERT Employee RELATION
{TUPLE {Employee_ID 39, Firstnames "John", Surname "Smith", Street "11 Ethel Benjamin Place", Suburb "Dunedin", Phone "(03) 497-0752", Email "-", Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary NULL, Qualification "Senior"}};

Tutorial D will return an error saying:
"Semantic error: Unable to find catalog entry for NULL"

This means that NULL is not defined in Tutorial D, and thus, values are not allowed to be NULL when inserted into Tutorial D relations. Hence, this addresses the SQL deficiency of specifying unknown values as NULL.

5.1.3 Third True Relational Feature: No Attribute Ordering

Date and Darwen (2007) point out that Tutorial D has no concept of a relation that pays attention to ordinal position of attributes. This is because attributes in Tutorial D are not ordered which is considered to be a true relational feature because the relational model of data pays no attention to attribute ordering. Consequently, instead of distinguishing attributes by their ordinal position, they are distinguishable by name (Date and Darwen, 2007).

This is illustrated through applying the same scenario used earlier when showing the SQL deficiency of reliance on column ordering (refer to Section 4.2.3).

To illustrate, when inserting a new employee record with Tutorial D, the only mechanism to do this is to distinguish the attributes by name instead of their ordinal positions. For example, inserting the following tuple using the attributes' names is valid in Tutorial D:

```
INSERT Employee RELATION
{TUPLE {Employee_ID 39, Firstnames "John", Surname "Smith", Street "11 Ethel
Benjamin Place", Suburb "Dunedin", Phone "(03) 497-0752", Email "-",
Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary
43610.0, Qualification "Senior"}};
```

Running the above Tutorial D statement will insert one tuple in the specified relation and will return the following message:
"Inserted 1 tuple"

However, relying on ordinal position of attributes is invalid in Tutorial D. For instance, when executing the previous Tutorial D statement without identifying the attributes' names and relying on their ordinal position will return an error, as follows:

INSERT Employee RELATION
{TUPLE {40, "John", "Smith", "11 Ethel Benjamin Place", "Dunedin", "(03) 497-0752", "-", "2007-09-29 09:13:00", "2007-09-29", 43610.0, "Senior"}};

The Tutorial D error returned says:
"Parse error: Encountered "40" at line 9, column 9.
Was expecting one of:
<IDENTIFIER>"

Therefore, Tutorial D avoids referring to attributes by their ordinal position and the only approach for referring to attributes is by specifying their names. Thus, Tutorial D addresses the SQL deficiency of reliance on column ordering and conforms to the relational model in this aspect.

5.2 Summary

To conclude this section, it is found that Tutorial D addresses the SQL deficiencies that were described in Section 4. This is achieved through providing true relational features that include rejecting duplicate tuples, NULLs and attribute ordering. Hence, utilising Tutorial D can be useful in order to conform to the relational model of data and avoid SQL deficiencies.

6. Analysis and Findings

After analysing the literature, a number of findings emerged. The purpose of this section is to summarise these findings and report the fundamental problem that needs to be addressed during the investigation of this research.

6.1 First Finding: SQL is the most widely used relational database language

Due to the issue of standardisation, SQL has a legacy of being the most widely used relational database language. This can make the adoption of any new relational database languages that are introduced after SQL difficult.

6.2 Second Finding: SQL is deficient

SQL has some deficiencies in conforming to the relational model of data. It is evident that solving those deficiencies is important because of their numerous problems.

6.3 Third Finding: Tutorial D is the solution

Tutorial D is a new relational database language that was introduced for the purpose of integrating objects and relational features, and also to address SQL deficiencies. It provides true relational features that realise the full potential of the relational model of data. The literature review concentrated on the main features that address the SQL deficiencies that were reviewed.

Based on the findings, the fundamental problem that this dissertation is based upon emerges.

6.4 The Fundamental Problem

SQL must be rejected as a language for next generation database systems due to its deficiencies. Date and Darwen support this claim as they state "We reject SQL as a long-term foundation for the future" (Date and Darwen, 2007, p.227). Hence, the introduction of Tutorial D as the next generation relational database language.

Although Tutorial D is found to be a suitable alternative to SQL because it addresses SQL deficiencies in conforming to the relational model of data, adopting it as the relational language for next generation database systems can be difficult due to the issue of SQL standardisation which made SQL popular and dominant as a relational database language.

6.5 Suggested Solution

In an attempt to utilise Tutorial D features that address SQL deficiencies but at the same time use SQL to interact with relational databases because of its legacy, a translation layer between the two languages can prove to be a worthwhile approach.

To explain, the suggested translation layer can be formed between SQL and a Tutorial D database. In other words, the translation layer can accept SQL statements and execute them on a Tutorial D database via a translation process.

This dissertation will first investigate the feasibility of implementing the translation layer through concentrating on a subset of the relational language. If the translation layer is proved to be feasible, this will allow legacy SQL clients to interoperate with the next generation database systems utilising their features. In addition, the dissertation will investigate the usefulness of the translation layer through studying the SQL deficiencies described earlier and how they are solved when SQL statements get executed on a Tutorial D database.

7. Previous Attempts at Translation

Translating between SQL and other database languages has been attempted beforehand and was proven to be feasible. This section will outline two such attempts at translation and how they were conducted. The purpose of this is to critically analyse these attempts before conducting the translation between SQL and Tutorial D which is the core study of this dissertation project.

7.1 Translating between SQL and AmosQL

Translating between SQL and AmosQL was conducted by Jagerskogh (2005) for the purpose of creating an SQL front-end that can enable using SQL to run queries on a mediator system.

To begin with, Jagerskogh (2005) describes a mediator as a system which allows the execution of queries over a collection of heterogeneous data sources. The study concentrated on a particular mediator system called Amos II which uses a query language called AmosQL. In addition, the purpose of translating SQL to AmosQL is to be able to use SQL as the query language that sits on top of the Amos II mediator system. The desirability of choosing SQL was because of its popularity as a database query language as it has been standardised (Jagerskogh, 2005). Therefore, it can be seen that the issue of SQL standardisation is very similar to the assertion made in this dissertation regarding SQL legacy.

However, the method of conducting the translation between SQL and AmosQL is different from the one followed in this dissertation. To explain, translating SQL to AmosQL happens by translating SQL statements into internal representations that are interpreted at the AmosQL side producing the required results (Jagerskogh, 2005). This is different from the approach undertaken in this dissertation when translating between SQL and Tutorial D. The approach followed here is a direct string manipulation of SQL statements to produce Tutorial D statements without transforming the SQL statements into internal representations first. This will be described in more detail in Section 8.2. However, the former approach that is used for translating between SQL and AmosQL is useful when there is a need to translate a large subset of the query language rather than a small subset of it. This is because it may get difficult to translate every possible SQL statement directly because of the complexity of the language. Thus, the approach undertaken by Jagerskogh (2005) is considered to be suitable for translating a large subset of the SQL language to AmosQL and can also considered to be suitable for translating a large subset of the SQL language to Tutorial D. However, due to the limited scope of this dissertation and the concentration on translating a small subset of the SQL language, the approach undertaken by Jagerskogh (2005) will not be considered when translating between SQL and Tutorial D.

The method of translation conducted by Jagerskogh (2005) led to successful results. When the translation was evaluated, it was found that 72% of SQL statements executed correctly. The error rate found is due to not implementing some of the functions that are used in the SQL statements used for testing (Jagerskogh, 2005). This is acceptable as the scope of the translation was not set to include the entire SQL language but a large subset of it.

7.2 Translating between SQL and XQuery

Translating between SQL and XQuery is another attempt made at translation. Its goal, as described by Jigyasu et al. (2006), is to enable the interaction with multiple XML-based views of heterogeneous information sources using SQL. XML stands for the *Extensible Markup Language* which is becoming the standard format for data interchange. It is noted that the current mechanism of handling such interaction is through XQuery which is becoming the standard language for querying XML data (Jigyasu et al., 2006).

Therefore, the approach undertaken by Jigyasu et al. is to enable this interaction by using SQL due to its legacy and popularity. Hence, translation is needed between SQL and XQuery in order to use SQL for the purpose of querying XML-based data using XQuery. Thus, it can be seen that the objective of this translation is very similar to the one undertaken in the previous section when translating SQL to AmosQL and also to one of the objectives of the translation undertaken for this dissertation. This objective is to use the standard legacy language SQL to interact with underlying data sources that are under the control of another query language, which is XQuery in this case.

In terms of the approach undertaken for this translation, it is found that it is different from the one followed in this dissertation. Translating SQL to XQuery was conducted by capturing the pieces of semantic information that SQL has and then storing them and processing them as needed. To explain, a progressive step-wise translation was conducted in order to achieve this. The first step is to perform SQL recognition and build an abstract representation of the SQL semantics. This can be an abstract syntax tree and after building it, the appropriate SQL semantics of information represented in the tree can be moved to specific locations in the same tree in preparation for the next step. The final step comes next and it performs XQuery expression generation on the moved SQL semantics of information (Jigyasu et al., 2006).

Therefore, it can be seen that the actual translation is effectively conducted in the third step when generating the XQuery expressions as the first two steps are merely a preparation for the translation process in order to carefully capture the semantics of information before translation. This approach has some similarities to the one undertaken for this dissertation for the purpose of translating between SQL and Tutorial D. In this regard, the preparation step is very important because it checks the input SQL statements for validity when performing the SQL recognition. Thus, the translation between SQL and Tutorial D took this step into consideration. However, building an abstract syntax tree to handle the semantics of information is not necessary for a small scope of the relational language, and thus, was discarded when implementing the translation layer between SQL and Tutorial D. As an alternative, a direct string manipulation is a suitable approach for such translation and was followed in this dissertation. This will be explained in more detail in Section 8.2. Nevertheless, there is still a need for some auxiliary data structures to be used in the translation logic conducted for the purpose of this dissertation and this is also described in Section 8.2.

Finally, the approach conducted by Jigyasu et al. (2006) led to successful results. It was shown that such translation is feasible and can be done with a variety of SQL statements, including multi-table queries.

7.3 Summary

In summary, translating between SQL and other database languages is not a new area in the database field. Two of the previous attempts made at translation were studied for the purpose of pointing out the similarities to and differences of those attempts with the one made in this dissertation. In addition, it was shown that those attempts were feasible and led to successful results. After showing this, the next step of the methodology followed in this dissertation is to implement the translation layer between SQL and Tutorial D. This is outlined in the following section.

8. Implementation of the Translation Layer

After studying the relevant issues from the literature, the next step is to implement the translation layer between SQL and Tutorial D. This section outlines this process.

8.1 The SQL Subset

Before implementing the translation between SQL and Tutorial D, it is important to understand the subset of the SQL language that is considered for translation for the purpose of this research. It was mentioned in the scope of the research that the translation layer will only deal with a small subset of the relational language. In SQL, this includes simple statements of the following:

- SELECT statements;
- INSERT statements;
- UPDATE statements; and
- DELETE statements

The following subsections outline the basic structure for these statements that is used for translation.

8.1.1 The SQL SELECT Statement

The SQL SELECT statement is used for the purpose of retrieving data. Data can be retrieved from multiple tables at once. However, for the purpose of this research, only single table queries were considered.

The basic structure of the SELECT statement that is used for translation in this research is simple as shown in Figure 6. It begins with a SELECT keyword and a select list which contains the column names or a "*" if wanting to select all the columns. The FROM keyword comes next which specifies the table reference which the columns will be selected from. If there is a need to limit the number of rows by a restrict operation, a WHERE clause is specified next with the condition that the restriction is based upon.

It is also noted that the full SELECT statement specified by the SQL standard includes several other clauses, such as GROUP BY and ORDER BY (ISO, 2003). However, for the sake of simplicity and to reduce complexity, these clauses have not been included in the translation. Selecting from multiple tables in the same SELECT statement also has not been included in the translation for the sake of simplicity. Thus, the translation layer only handles the basic structure of the SELECT statement which is described earlier and shown in Figure 6.

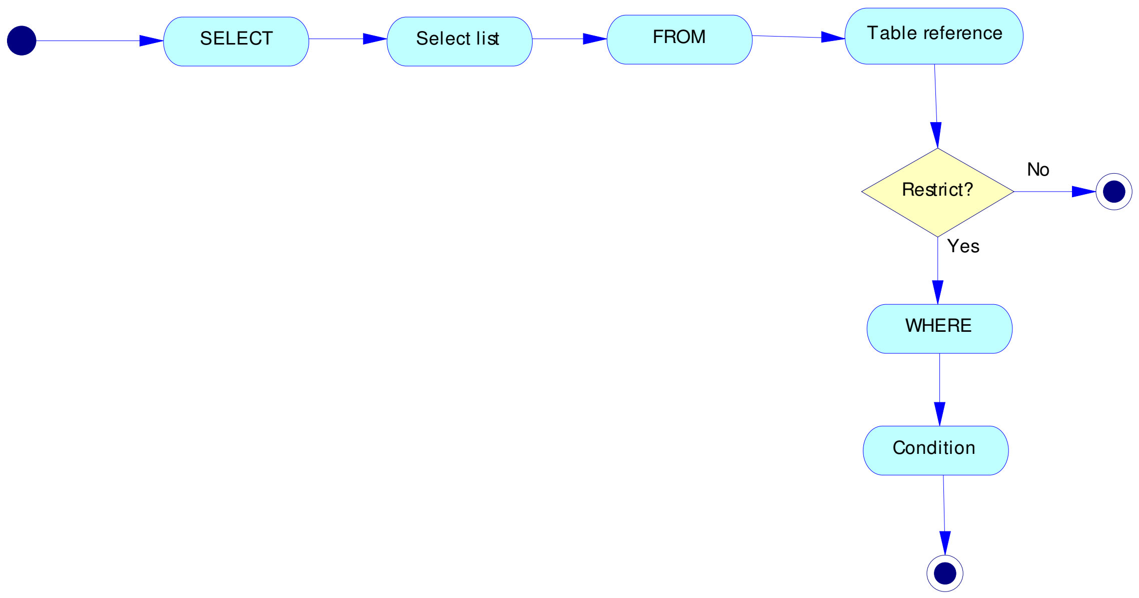


Figure 6: The SQL SELECT Statement

8.1.2 The SQL INSERT Statement

The purpose of this statement is to insert new rows into a table. New rows can be inserted one at a time by relying on the ordinal position of columns or by specifying column names. This is shown in Figure 7.

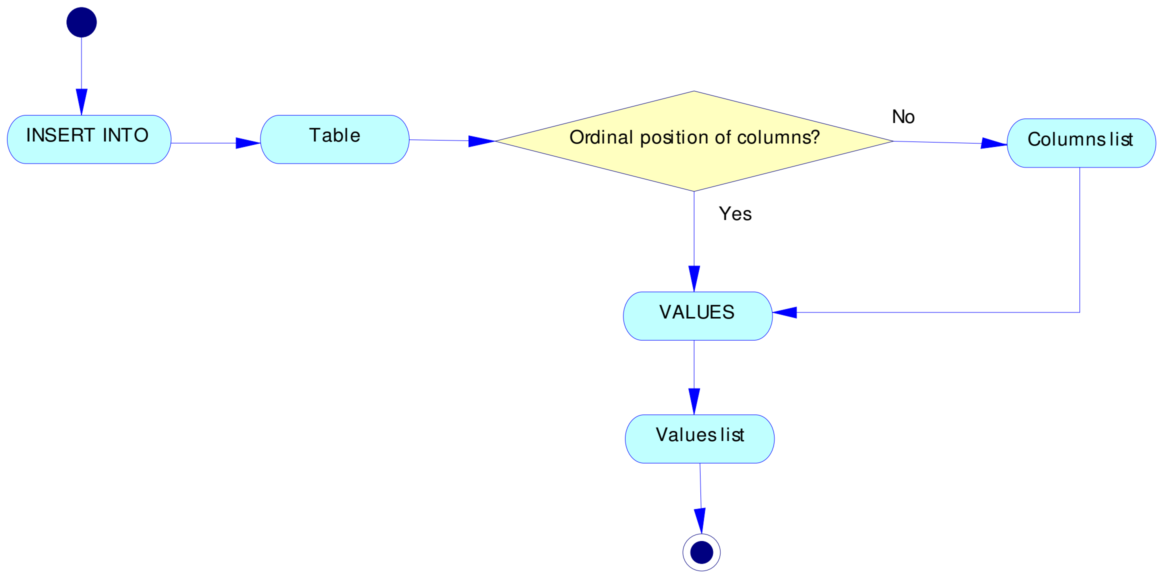


Figure 7: The SQL INSERT Statement

8.1.3 The SQL UPDATE Statement

The purpose of this statement is for changing the existing values in a table. This is specified in the update's SET clause. In addition, updating can happen for all the rows in a table or restricted to a number of rows based on a WHERE clause. This is shown in Figure 8.

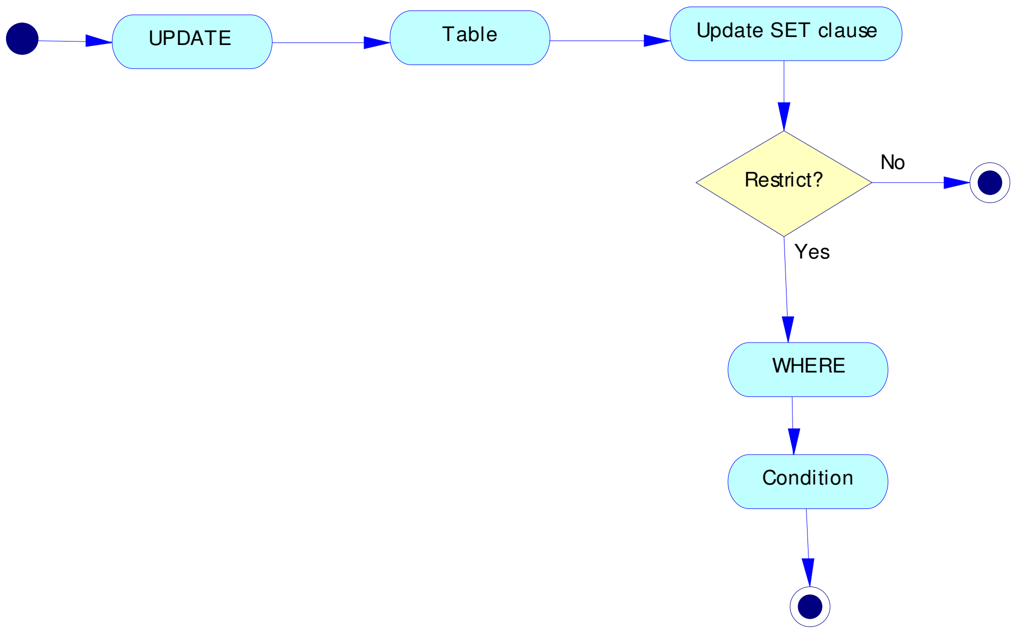


Figure 8: The SQL UPDATE Statement

8.1.4 The SQL DELETE Statement

This statement is used to delete rows from a table. Like SELECT and UPDATE, the DELETE statement can also restrict the deletion to certain rows based on a condition. If the restrict option is not specified, then all the rows in the specified table will be deleted. The structure of the DELETE statement is shown in Figure 9.

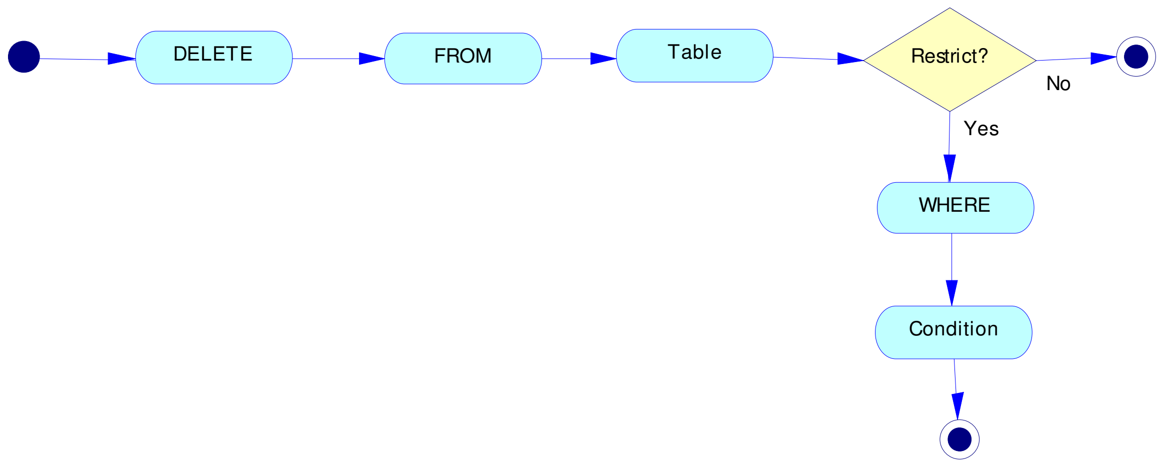


Figure 9: The SQL DELETE Statement

8.2 The Translation Process

Java was chosen as the programming language to write the application that forms the translation layer between the two languages. The process of building the translation layer consists of several steps. These are:

- understanding string manipulation;
- writing the translation logic in Java;
- designing the interface for translation;
- selecting an open source Tutorial D database product; and
- executing SQL statements on the Tutorial D database.

The following subsections detail the translation process.

8.2.1 String Manipulation

The translation between SQL and Tutorial D was conducted through string manipulation. This means that each SQL statement is treated as a string object in Java. To explain, a string in Java is a class that can represent text and manipulate it through many methods that can be called on the string object. Thus, by manipulating the SQL string object, a different string object can be formed and returned. Through some defined logic, it is easy to set the returned string object to be the resultant Tutorial D statement. Thus, translation can happen through string manipulation. These are the main string manipulation methods used for the purpose of such translation:

- **indexOf(String str):** returns the position of the first occurrence of the string object that is passed as an argument. This is useful for locating the position of keywords in the SQL statement.
- **substring (int beginIndex, int endIndex):** returns a new string which is a substring starting at the begin index and ending at the end index which are specified as arguments. This can be used to manipulate a portion of the SQL statement.
- **charAt(int index):** returns the character value found at the index passed as an argument.
- **contains(CharSequence s):** returns a true or false value depending on whether a string object contains the specified character sequence passed as an argument. This is useful for confirming if the SQL statement contains a specified keyword or not.

8.2.2 The Translation Logic

The second step of translation after understanding string manipulation is writing the translation logic in Java. For this purpose, a new class called *Translate.java* was created to perform the translation. The complete Java code written for this class is provided in Appendix 5.

The purpose of the translation logic class is to accept an SQL statement as a string object and return the equivalent Tutorial D statement as a new string object. This is done through the string manipulation described earlier. The translation logic also handles invalid SQL statements by throwing suitable exceptions that terminate the logic before conducting the translation. This can ensure that the SQL statement is valid before translating it to a Tutorial D statement. The following subsections outline the translation logic for each of the types of SQL statements found in the subset described earlier.

8.2.2.1 Translating the SELECT Statement

The logic handles the translation of the SELECT statement by starting with the WHERE clause. If the WHERE clause does not exist in the statement, the table reference is found between the FROM keyword and the end of statement. Otherwise, it is found between the FROM keyword and the WHERE clause. In both cases, the logic checks if there are any columns that are selected in the select list. If any, these are returned as the last portion of the Tutorial D statement.

In general, the translation logic translates SELECT statements to Tutorial D statements as follows:

SQL:

SELECT <select list> FROM <Table reference> WHERE <Condition>;

Tutorial D:

(<Table reference> [<Condition>]) {<select list>}

For the purpose of illustration, the Employee scenario can be used to show the translation of the SELECT Statement as follows:

SQL:

SELECT Firstnames, Surname, Qualification FROM Employee WHERE Employee_ID = 1;

Resultant Tutorial D:

(Employee[Employee_ID = 1]) {Firstnames, Surname, Qualification}

8.2.2.2 Translating the INSERT Statement

Translating the INSERT Statement is handled by first storing the columns list and values list into auxiliary data structures. This is because Tutorial D always specifies each value by the column name, unlike SQL where this is optional. For this purpose, the chosen auxiliary data structure is an array list. Thus, the translation of the INSERT statement begins by storing the columns into an array list and the corresponding values into another array list. After completing this process, each column name and its corresponding value is fetched from the array lists and appended to the resultant Tutorial D statement.

In general, the translation logic translates INSERT statements to Tutorial D statements as follows (for a columns list size of N):

SQL:

INSERT INTO <Table> (<Columns list>) VALUES (<Values list>);

Tutorial D:

<Table> += TUPLE {<Column1> <Value1>, <Column2> <Value2>... <ColumnN> <ValueN>};

For the purpose of illustration, the Employee scenario can be used to show the translation of the INSERT Statement as follows:

SQL:

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (40, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');

Resultant Tutorial D:

Employee += TUPLE {Employee_ID 40, Firstnames "John", Surname "Smith", Street "11 Ethel Benjamin Place", Suburb "Dunedin", Phone "(03) 497-0752", Email "-", Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary 43610.0, Qualification "Senior"};

8.2.2.3 Translating the UPDATE Statement

In a similar manner to translating the INSERT statement, the logic handles the translation of the UPDATE statement by storing the column names and values found in the SET clause into array lists. If the WHERE clause exists, the condition is specified at the beginning of the Tutorial D statement. Then the column names and values are fetched from the array lists and appended to the resultant Tutorial D statement.

In general, the translation logic translates UPDATE statements to Tutorial D statements as follows:

SQL:

UPDATE <Table> SET <columns and new values> WHERE (<Condition>);

Tutorial D:

<Table> @= [<Condition>] (columns and new values);

For the purpose of illustration, the Employee scenario can be used to show the translation of the UPDATE Statement as follows:

SQL:

UPDATE Employee SET Qualification = 'Senior' WHERE Employee_ID = 1;

Resultant Tutorial D:

Employee @= [Employee_ID = 1] (Qualification := "Senior");

8.2.2.4 Translating the DELETE Statement

In a similar manner to translating the SELECT statement, the logic handles the translation of the DELETE statement by checking if the WHERE clause exists. If it does, then the condition is specified in the Tutorial D statement. If not, then the keyword *ALL* is appended to the statement to delete all the tuples from the specified relation in Tutorial D.

In general, the translation logic translates DELETE statements to Tutorial D statements as follows:

SQL:
DELETE FROM <Table> WHERE <Condition>;

Tutorial D:
<Table> -= [<Condition>];

If the WHERE clause does not exist, then the translation is handled as follows:

SQL:
DELETE FROM <Table>;

Tutorial D:
<Table> -= ALL;

For the purpose of illustration, the Employee scenario can be used to show the translation of the DELETE Statement as follows:

SQL:
DELETE FROM Employee WHERE Employee_ID = 1;

Resultant Tutorial D:
Employee -= [Employee_ID = 1];

8.2.3 The SQL-to-Tutorial D Interface

The next step of translation is to provide the user interface which allows users to execute SQL statements on a Tutorial D database. For the purpose of this dissertation, the user interface that was built is simple and easy to use as shown in Figure 10.

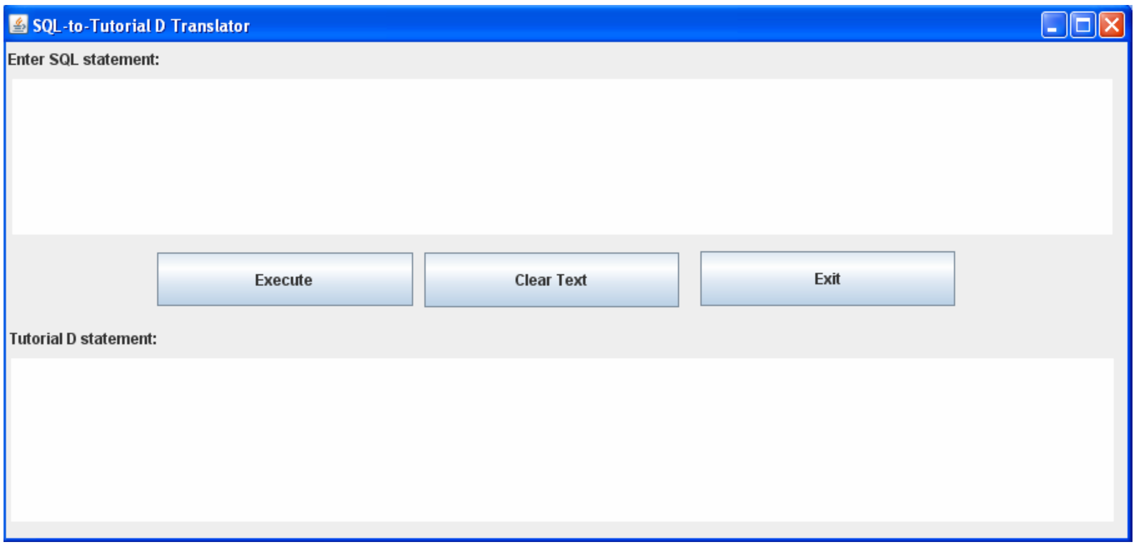


Figure 10: The SQL-to-Tutorial D Interface

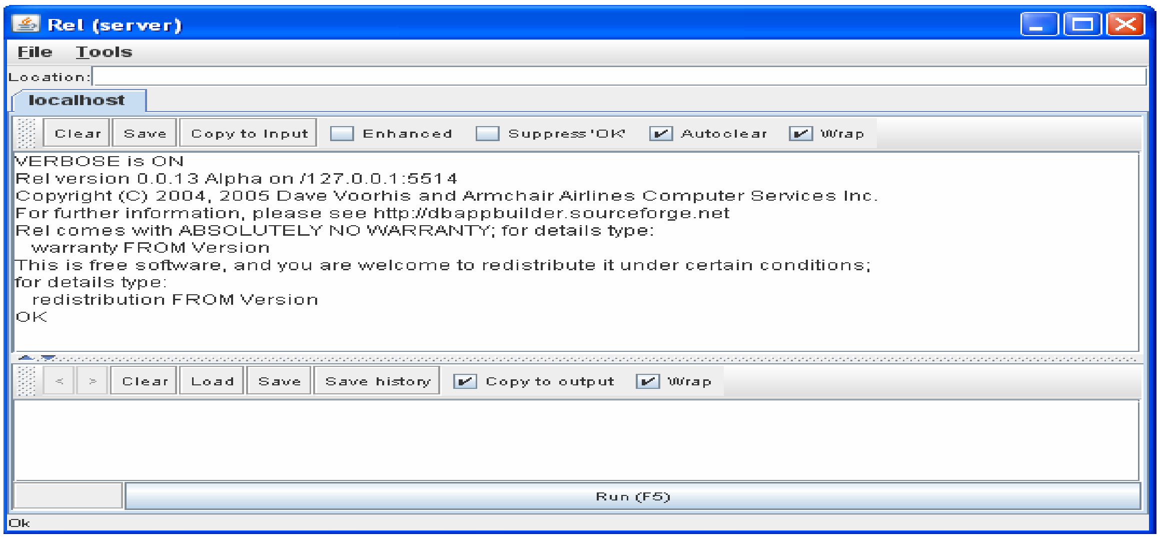
As shown in the Figure, the SQL-to-Tutorial D interface has three main options. Firstly, users can enter an SQL statement in the top text area and by clicking on 'Execute', the statement is translated using the logic provided earlier. The result of this is a Tutorial D statement that will appear in the bottom text area which is not editable. This Tutorial D statement will get executed automatically on a Tutorial D database. Secondly, users can clear the text for both text areas by clicking on the 'Clear Text' button. Finally, the 'Exit' button will exit the application if clicked. The following sections will outline the process of executing the resultant Tutorial D statement on a Tutorial D database.

8.2.4 Rel: An Open Source Tutorial D Database[1]

For the purpose of executing the resultant Tutorial D statements, *Rel* was chosen as the Tutorial D database. Rel is an open source relational database server that is based on Tutorial D. It can be seen as an implementation of Tutorial D that can allow creating and managing true relational databases.

Rel is implemented in Java, and because of this, it was easily integrated with the SQL-to-Tutorial D interface that was built for the purpose of this research. The Rel components that were integrated consist of two parts. These are a database server and a browser. The purpose of the Rel database server is for executing the Tutorial D statements whereas the Rel browser shows the results on screen.

For the purpose of this research, the Rel browser component is invoked whenever the SQL-to-Tutorial D interface is executed. Thus, in addition to the SQL-to-Tutorial D window that will appear when starting up the application, another window for the Rel browser appears on screen. The Rel browser window is shown in Figure 11.



```
VERBOSE is ON
Rel version 0.0.13 Alpha on /127.0.0.1:5514
Copyright (C) 2004, 2005 Dave Voorhis and Armchair Airlines Computer Services Inc.
For further information, please see http://dbappbuilder.sourceforge.net
Rel comes with ABSOLUTELY NO WARRANTY; for details type:
  warranty FROM Version
This is free software, and you are welcome to redistribute it under certain conditions;
for details type:
  redistribution FROM Version
OK
```

Figure 11: Rel Browser

[1] Rel Official Website: http://dbappbuilder.sourceforge.net/Rel.html

8.2.5 Executing SQL Statements on Rel

The code snippet that handles the execution of SQL statements on Rel is provided in Appendix 6.

To explain this process, after integrating the Rel components with the SQL-to-Tutorial D interface, the final step is to execute the SQL statements on Rel. Executing SQL statements means that the resultant Tutorial D statements that are produced after translation and displayed in the bottom text area in the SQL-to-Tutorial D interface get executed on Rel. In order to do this, a few modifications on the open source Java library for the integrated Rel browser were necessary. This is completely legitimate as it is mentioned in the official website of Rel[1] that the open source Java library of Rel is free and can be modified under the terms of the GNU General Public License[2].

In this regard, only two additions were done for the purpose of getting a reference to the necessary objects of the running browser that can enable the execution of Tutorial D statements through the SQL-to-Tutorial D interface. Thus, two getter methods were added to the integrated Rel browser library for the purpose of getting the references needed. After the references to the necessary objects were retrieved, the same method that is used for executing the Tutorial D statements on the database server component of Rel was invoked on the obtained references from the SQL-to-Tutorial D interface. The method passes the resultant Tutorial D statement after translation as an argument which will then be executed on the Rel database server and the result will be displayed on the Rel browser.

8.3 Summary

To summarise, this section demonstrated how such a translation between SQL and Tutorial D can be implemented when having a small subset of the relational language. The process is broken up into several steps which lead to the overall task of executing SQL statements on a Tutorial D database. However, testing the translation solution is a necessary task which is the purpose of the following section.

[1] Rel Official Website: http://dbappbuilder.sourceforge.net/Rel.html

[2] GNU General Public License Website: http://www.gnu.org/copyleft/gpl.html

9. Testing the Translation Layer

Following the implementation of the translation layer between SQL and Tutorial D is the testing stage of the methodology (refer to Section 1.5.4). This is an important activity in order to ensure the successful operation of such translation and to achieve the objectives of this research (refer to Section 1.2).

Therefore, the purpose of this section is to outline the testing procedure and results. The aim of this is to answer the remaining questions that were raised earlier in Section 1.3 and have not been answered yet by either the Literature Review or the Implementation stages. In particular, testing aims to provide findings that can lead to the following conclusions:

- the translation layer between SQL and Tutorial D is a feasible approach to preserve the interoperability of the two languages; and
- SQL deficiencies can be solved through interoperating with Tutorial D via SQL.

9.1 Testing Procedure

Before staring the testing procedure, a preliminary step is necessary to set up the Tutorial D database by creating the necessary relations and loading them with data. For the purpose of testing, it is assumed that the same Employee relation used earlier is used again in testing the translation layer. After ensuring that the Employee relation exists in the Tutorial D database running on the Rel database server component of the translation layer, the testing procedure can be started.

The testing procedure was divided into several steps. The first step is to test the translation layer in terms of feasibility. This means that the translation layer must be tested against a combination of valid SQL statements that should be executed on Rel after translating them to the corresponding Tutorial D statements. The combination of valid SQL statements was chosen on the basis of covering all the cases handled by the translation layer and which were defined in the SQL subset chosen for this research (refer to Section 8.1). The aim of this step is to show that all the chosen SQL statements in the combination chosen pass the test and are translated to Tutorial D statements that are automatically executed on Rel giving the correct desired results. Section 9.2.1 details the results of this step of testing.

The second step is to test whether the SQL deficiencies defined in Section 4 are solved when interoperating with Tutorial D via SQL. In theory, they should be solved because Tutorial D addresses those deficiencies as described in Section 5. However, this still needs to be tested and shown through the use of the translation layer. Therefore, a combination of suitable SQL statements was also chosen for this purpose. Section 9.2.2 details the results of this step of testing.

Finally, the third step of testing the translation layer is testing the robustness of the SQL-to-Tutorial D interface. This means that the interface should catch all the invalid SQL statements before applying the translation logic and return a suitable error message. By doing this, invalid SQL statements are prohibited from translation and this is desirable because, for example, users will get notified if the submitted SQL statement is invalid. Otherwise, it might be difficult to know where the error comes from if it was not caught by the interface and the invalid SQL statement was translated and executed on Rel which can lead to undesired behaviour. Therefore, a combination of invalid SQL statements was chosen for the purpose of testing this aspect of the translation layer. Section 9.2.3 details the results of this step of testing.

9.2 Testing Results

This subsection details the results gathered from testing the translation layer. It follows the same structure defined above in the testing procedure. Therefore, this subsection is divided as follows:

- testing the feasibility aspect of the translation layer;
- testing the solution to SQL deficiencies; and
- testing the robustness aspect of the translation layer.

9.2.1 Testing the Feasibility Aspect of the Translation Layer

For each type of the SQL operations that are defined in the SQL subset chosen for this research, a combination of SQL statements was chosen to test the feasibility of the translation layer between SQL and Tutorial D. Those statements are valid and they can be run on any database that uses SQL. In addition, they were carefully chosen to cover all the cases that should be handled by the translation layer. Tables 1 to 4 detail the testing process and results of those valid SQL statements for each of the combination chosen for valid SELECT, INSERT, UPDATE and DELETE statements.

To begin with, valid SQL SELECT statements were tested first. Table 1 shows the results of this. It is important to note that the combination of SQL statements chosen for this purpose covers all the possible cases that are handled by the translation layer. For the combination of SELECT statements, this includes the basic case of selecting all the tuples from the relation with all its attributes (i.e. case number VS1), projecting certain attributes of the relation (i.e. case numbers VS5 & VS6) and restricting the number of tuples in the relation based on a condition (i.e. case numbers VS2, VS3 & VS4). A combination of both projection and restriction operations is also included (i.e. case numbers VS7, VS8 & VS9). Finally, a case that returns an empty relation with no tuples was also tested (i.e. case number VS10).

Valid SQL INSERT statements were tested next. Table 2 shows the results of this. There were only two cases that needed to be tested. The first one is the normal INSERT statement case with the entire attributes list defined (i.e. case number VI1). The second one is the case of having the attributes defined in a different order (i.e. case number VI2).

Valid SQL UPDATE statements were also tested. Table 3 shows the results of this. Few cases were needed for testing valid UPDATE statements. This includes the basic case of updating one attribute value of a relation for all the tuples (i.e. case number VU1), the case of restricting the number of tuples to be updated by specifying a condition (i.e. case numbers VU2, VU3 & VU4), the case of updating multiple attribute values of a relation for all the tuples (i.e.VU5) and the case of updating multiple attribute values of a relation for a restricted number of tuples based on a condition (i.e. case number VU6).

Finally, valid SQL DELETE statements were tested. Table 4 shows the results of this. Two cases were needed in order to test valid DELETE statements. The first one is the case of deleting a restricted number of tuples in the relation based on a condition (i.e. case numbers VD1, VD2, VD3 & VD4). The second one is the case of deleting all the tuples in the relation (i.e. case number VD5).

No.	SQL Statement	Resultant Tutorial D Statement	Rel Result (on the Rel Browser)	Outcome (Pass or Fail)
VS1	SELECT * FROM Employee;	Employee	A relation of all tuples with all attribute values for the Relation Employee is displayed	Pass
VS2	SELECT * FROM Employee WHERE Employee_ID = 1;	(Employee[Employee_ID = 1])	A relation of one tuple with all attribute values for the Relation Employee is displayed	Pass
VS3	SELECT * FROM Employee WHERE Firstnames = ‘Sewal’ AND Surname = ‘Kirby’;	(Employee[Firstnames = "Sewal" and Surname = "Kirby"])	A relation of one tuple with all attribute values for the Relation Employee is displayed	Pass
VS4	SELECT * FROM Employee WHERE Employee_ID = 1 OR Employee_ID = 2;	(Employee[Employee_ID = 1 OR Employee_ID = 2])	A relation of two tuples with all attribute values for the Relation Employee is displayed	Pass
VS5	SELECT Employee_ID FROM Employee;	Employee{Employee_ID}	A relation of all tuples with one attribute value for the Relation Employee is displayed	Pass
VS6	SELECT Employee_ID, Firstnames, Surname, Salary FROM Employee;	Employee{Employee_ID, Firstnames, Surname, Salary}	A relation of all tuples with four attribute values for the Relation Employee is displayed	Pass
VS7	SELECT Employee_ID, Firstnames, Surname, Salary FROM Employee WHERE Employee_ID = 1;	(Employee[Employee_ID = 1]) {Employee_ID, Firstnames, Surname, Salary}	A relation of one tuple with four attribute values for the Relation Employee is displayed	Pass

VS8	SELECT Employee_ID, Firstnames, Surname, Salary FROM Employee WHERE Firstnames = ‘Sewal’ AND Surname = ‘Kirby’;	(Employee[Firstnames = "Sewal" AND Surname = "Kirby"]) {Employee_ID, Firstnames, Surname, Salary}	A relation of one tuple with four attribute values for the Relation Employee is displayed	Pass
VS9	SELECT Employee_ID, Firstnames, Surname, Salary FROM Employee WHERE Employee_ID = 1 OR Employee_ID = 2;	(Employee[Employee_ID = 1 OR Employee_ID = 2]) {Employee_ID, Firstnames, Surname, Salary}	A relation of two tuples with four attribute values for the Relation Employee is displayed	Pass
VS10	SELECT * FROM Employee WHERE Employee_ID = 100;	(Employee[Employee_ID = 100])	An empty relation (a relation with no tuples) because there is no Employee tuple with an Emplyee_ID attribute value of 100	Pass

Table 1: Testing Valid SQL SELECT Statements Translation

No.	SQL Statement	Resultant Tutorial D Statement	Rel Result (on the Rel Browser)	Outcome (Pass or Fail)
VI1	INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (40, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');	Employee += TUPLE {Employee_ID 40, Firstnames "John", Surname "Smith", Street "11 Ethel Benjamin Place", Suburb "Dunedin", Phone "(03) 497-0752", Email "-", Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary 43610.0, Qualification "Senior"};	Inserted 1 tuple.	Pass
VI2	INSERT INTO Employee (Employee_ID, Surname, Firstnames, Suburb, Street, Email, Phone, Date_Entered, Date_Employed, Salary, Qualification) VALUES (41, 'Smith', 'John', 'Dunedin', '11 Ethel Benjamin Place', '-', '(03) 497-0752', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');	Employee += TUPLE {Employee_ID 41, Surname "Smith", Firstnames "John", Suburb "Dunedin", Street "11 Ethel Benjamin Place", Email "-", Phone "(03) 497-0752", Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary 43610.0, Qualification "Senior"};	Inserted 1 tuple.	Pass

Table 2: Testing Valid SQL INSERT Statements Translation

No.	SQL Statement	Resultant Tutorial D Statement	Rel Result (on the Rel Browser)	Outcome (Pass or Fail)
VU1	UPDATE Employee SET Qualification = ‘Junior’;	Employee @= (Qualification := "Junior");	Updated 41 tuples.	Pass
VU2	UPDATE Employee SET Qualification = ‘Senior’ WHERE Employee_ID = 1;	Employee @= [Employee_ID = 1] (Qualification := "Senior");	Updated 1 tuple.	Pass
VU3	UPDATE Employee SET Qualification = ‘Senior’ WHERE Firstnames = ‘Sewal’ AND Surname = ‘Kirby’;	Employee @= [Firstnames = "Sewal" AND Surname = "Kirby"] (Qualification := "Senior");	Updated 1 tuple.	Pass
VU4	UPDATE Employee SET Qualification = ‘Senior’ WHERE Firstnames = ‘Sewal’ AND Surname = ‘Kirby’ OR Employee_ID = 4;	Employee @= [Firstnames = "Sewal" AND Surname = "Kirby" OR Employee_ID = 4] (Qualification := "Senior");	Updated 2 tuples.	Pass
VU5	UPDATE Employee SET Qualification = ‘Senior’, Salary = 43610.0, Email = ‘noemail@nowhere.com’;	Employee @= (Qualification := "Senior", Salary := 43610.0, Email := "noemail@nowhere.com");	Updated 41 tuples.	Pass
VU6	UPDATE Employee SET Qualification = ‘Senior’, Salary = 43610.0, Email = ‘noemail@nowhere.com’ WHERE Firstnames = ‘Sewal’ AND Surname = ‘Kirby’ OR Employee_ID = 4;	Employee @= [Firstnames = "Sewal" AND Surname = "Kirby" OR Employee_ID = 4] (Qualification := "Senior", Salary := 43610.0, Email := "noemail@nowhere.com");	Updated 1 tuple.	Pass

Table 3: Testing Valid SQL UPDATE Statements Translation

No.	SQL Statement	Resultant Tutorial D Statement	Rel Result (on the Rel Browser)	Outcome (Pass or Fail)
VD1	DELETE FROM Employee WHERE Employee_ID = 1;	Employee -= [Employee_ID = 1];	Deleted 1 tuple.	Pass
VD2	DELETE FROM Employee WHERE Firstnames = ‘John’ and Surname = ‘Smith’;	Employee -= [Firstnames = "John" and Surname = "Smith"];	Deleted 7 tuples.	Pass
VD3	DELETE FROM Employee WHERE Employee_ID = 2 OR Employee_ID = 3;	Employee -= [Employee_ID = 2 OR Employee_ID = 3];	Deleted 2 tuples.	Pass
VD4	DELETE FROM Employee WHERE Qualification <> ‘Senior’;	Employee -= [Qualification <> "Senior"];	Deleted 0 tuples.	Pass
VD5	DELETE FROM Employee;	Employee-= ALL;	Deleted 38 tuples.	Pass

Table 4: Testing Valid SQL DELETE Statements Translation

9.2.2 Testing the Solution to SQL Deficiencies

One of the research objectives of this dissertation is to investigate the translation layer in terms of showing how SQL deficiencies can be solved. After studying the literature, it was shown that Tutorial D addresses those deficiencies through introducing true relational features that are not achieved by SQL (refer to Section 5). However, SQL cannot be simply discarded because of the legacy factor, and thus, a suggested solution to this problem was to utilise the translation layer between the two languages so that legacy SQL clients can still use SQL but interact with a Tutorial D database to solve SQL deficiencies (refer to Section 6.5). Thus, after testing the feasibility aspect of the translation layer, the translation layer is tested in terms of solving the SQL deficiencies. Few cases were needed to test this aspect; as shown in Table 5.

No.	SQL Statement	Resultant Tutorial D Statement	Rel Result (on the Rel Browser)

TR1	SELECT Qualification FROM Employee;	Employee{Qualification}	A relation of only one tuple with one attribute value for the Relation Employee is displayed
			i.e.
			RELATION {Qualification CHAR} {
			TUPLE {Qualification "Senior"}}

TR2	INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (43, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, NULL);	Employee += TUPLE {Employee_ID 40, Firstnames "John", Surname "Smith", Street "11 Ethel Benjamin Place", Suburb "Dunedin", Phone "(03) 497-0752", Email "-", Date_Entered "2007-09-29 09:13:00", Date_Employed "2007-09-29", Salary 43610.0, Qualification NULL};	ERROR "Semantic error: Unable to find catalog entry for NULL"
TR3	INSERT INTO Employee VALUES (44, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');	Employee += TUPLE {43, "John", "Smith", "11 Ethel Benjamin Place", "Dunedin", "(03) 497-0752", "-", "2007-09-29 09:13:00", "2007-09-29", 43610.0, "Senior"};	ERROR "Parse error: Encountered "43" at line 4, column 11. Was expecting one of: <IDENTIFIER> ... "}" ... "

Table 5: Testing the Solution to SQL Deficiencies

9.2.3 Testing the Robustness Aspect of the Translation Layer

For each type of the SQL operations that are defined in the SQL subset chosen for this research, a combination of SQL statements was chosen to test the robustness of the translation layer between SQL and Tutorial D. Those statements are invalid and they should be caught by the SQL-to-Tutorial D interface, and hence, they should not be translated. They cover most of the common cases that any SQL interface should deal with to ensure the robustness of the interface. Tables 6 to 9 detail the testing process and results of those invalid SQL statements for each of the combination chosen for invalid SELECT, INSERT, UPDATE and DELETE statements.

No.	SQL Statement	Resultant Tutorial D Statement	Error Message	Exception caught (yes or no)
IS1	SELECT * WHERE Employee_ID = 1;	None	FROM keyword not found where expected	Yes
IS2	SELECT * WHERE Employee_ID = 1 FROM Employee;	None	FROM keyword not found where expected	Yes
IS3	SELECT FROM Employee;	None	missing expression	Yes
IS4	SELECT Employee_ID Firstnames Surname FROM Employee;	None	missing expression	Yes
IS5	SELECT Employee_ID, Firstnames, Surname FROM *Employee;	None	invalid SQL statement	Yes
IS6	SELECT Firstnames, Surname FROM *Employee WHERE Employee_ID = 1;	None	invalid SQL statement	Yes

Table 6: Testing Invalid SQL SELECT Statements Exception Handling

No.	SQL Statement	Resultant Tutorial D Statement	Error Message	Exception caught (yes or no)
II1	INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification, IRD_No) (45, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');	None	missing VALUES keyword	Yes
II2	INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification, IRD_No) VALUES (46, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior');	None	not enough values	Yes
II3	INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb, Phone, Email, Date_Entered, Date_Employed, Salary, Qualification) VALUES (47, 'John', 'Smith', '11 Ethel Benjamin Place', 'Dunedin', '(03) 497-0752', '-', '2007-09-29 09:13:00', '2007-09-29', 43610.0, 'Senior', ’45-125-546’);	None	too many values	Yes

Table 7: Testing Invalid SQL INSERT Statements Exception Handling

No.	SQL Statement	Resultant Tutorial D Statement	Error Message	Exception caught (yes or no)
IU1	UPDATE Employee Qualification = ‘Senior’, Salary = 43610.0, Email = ‘noemail@nowhere.com’;	None	missing SET keyword	Yes
IU2	UPDATE SET Qualification = ‘Senior’, Salary = 43610.0, Email = ‘noemail@nowhere.com’;	None	invalid table name	Yes

Table 8: Testing Invalid SQL UPDATE Statements Exception Handling

No.	SQL Statement	Resultant Tutorial D Statement	Error Message	Exception caught (yes or no)
ID1	DELETE FROM;	None	invalid table name	Yes
ID2	DELETE FROM WHERE Employee_ID = 1;	None	invalid table name	Yes

Table 9: Testing Invalid SQL DELETE Statements Exception Handling

10. Analysis and Discussion of the Testing Results

The purpose of this section is to analyse and discuss the results gathered from the previous section after testing the translation layer between SQL and Tutorial D.

10.1 The Feasibility Aspect

From the gathered testing results shown in Tables 1 to 4, it can be seen that the translation layer is feasible. From the tables, it was shown that all valid SQL statements are translated to valid Tutorial D statements. Those valid Tutorial D statements are then executed automatically on the Rel database server, showing the result on the Rel browser. In particular, the result of executing the valid SQL SELECT statements is displaying the resultant relation in the Rel browser window. For the other types of SQL statements, the result will be the desired behaviour sought from the executed statements. If it was an INSERT statement, the result will be a newly inserted tuple into the specified relation in Rel. Moreover, if it was an UPDATE statement, the result will be updating the necessary tuples of the specified relation in Rel. Finally, if it was a DELETE statement, the result will be deleting the necessary tuples from the specified relation in Rel.

As can be seen in Tables 1 to 4, all those cases defined earlier pass and the Employee relation created in Rel can be manipulated using SQL. Hence, the translation layer is a feasible approach to ensure the interoperability between SQL and Tutorial D.

10.2 The Solution to SQL Deficiencies Aspect

From the gathered testing results shown in Table 5, it can be seen that the translation layer works as a solution to the deficiencies of SQL. To begin with, the translation layer is tested to see whether the duplicate rows deficiency is solved or not (i.e. case number TR1). Because SQL statements are executed on Rel which is a Tutorial D database, the relevant relational model terminologies that are associated with Tutorial D will be used. Therefore, rows are tuples in Tutorial D and the deficiency at hand can be referred to as the duplicate tuples deficiency.

When executing the SQL statement defined in TR1 using the SQL-to-Tutorial D interface, it will be translated to the corresponding Tutorial D statement because the original SQL statement is considered to be valid. Subsequently, the resultant Tutorial D statement will be executed automatically on Rel returning a relation with one tuple as specified in TR1. Although there are many other tuples with the same Qualification value, returning them in the result will lead to duplicate tuples. This true relational feature of eliminating duplicate tuples is associated with Tutorial D and was explained in Section 5.1.1. Therefore, it can be seen that this deficiency is solved through the translation layer while still using SQL.

Secondly, the translation layer is tested to see whether values specified as NULL are avoided or not. This is tested by specifying an unknown value of Qualification as NULL (i.e. case number TR2). When executing this SQL statement, it is considered to be valid and will be translated to the corresponding Tutorial D statement without discarding NULL. However, all attributes in Tutorial D cannot have values specified as NULL which is a true relational feature as discussed earlier (refer to Section 5.1.2). In this case, the resultant Tutorial D statement will be executed on Rel but will return an error saying that the catalog entry for NULL cannot be found. Thus, legacy SQL clients are forced not to use NULL when specifying values and this is achieved through the translation layer.

Finally, the translation layer is tested to see whether the SQL deficiency of the reliance on column ordering is solved or not. Columns are attributes in Tutorial D so this deficiency effectively means the reliance on attribute ordering because SQL statements are executed on Rel which is a Tutorial D database. This is tested through executing an SQL INSERT statement that only specifies the values without their corresponding attributes list (i.e. case number TR3). This means that distinguishing the values rely on the ordinal position of attributes. However, the only mechanism to distinguish values in Tutorial D is by their attribute names as there is no reliance on attribute ordering (refer to Section 5.1.3). Nevertheless, the INSERT SQL statement in TR3 is considered to be valid and translated to the corresponding Tutorial D statement. The resultant Tutorial D statement, however, is considered to be invalid and when it is executed automatically, a parse error will be returned; as shown in TR3. Therefore, legacy SQL clients are forced to use the only mechanism of insertion which is by relying on the attribute names instead of their ordinal position and this is achieved through the translation layer.

10.3 The Robustness Aspect

From the gathered testing results shown in Table 6 to 9, it can be seen that the translation layer is robust in terms of not translating invalid SQL statements and returning suitable error messages. To begin with, invalid SQL SELECT statements were tested first. Table 6 shows the results of this. It can be seen that the interface handles all the exceptions specified and returns suitable error messages. For the exceptions of not specifying the FROM keyword or if it was specified in the wrong place, an error message saying “FROM keyword not found where expected” will be displayed (i.e. case numbers IS1 & IS2). In addition, if the attributes list was missing from the SELECT statement or if the attributes are not separated from each other by commas, an error message saying “missing expression” will be displayed (i.e. case numbers IS3 & IS4). Finally, for the case of specifying a star sign aside the table name, an error message saying “invalid SQL statement” will be displayed (i.e. case numbers IS5 & IS6). All of those exceptions are caught by the interface and not translated to Tutorial D statements making the debugging process easier.

Secondly, invalid SQL INSERT statements were tested next. Table 7 shows the results of this. There were only three exceptions that needed to be tested. The first one is when the VALUES keyword is missing from the INSERT statement (i.e. case number II1). This is caught by the interface showing the error message “missing VALUES keyword”. Furthermore, another exception that needs to be caught in invalid INSERT statements is the case of having more attributes in the attributes list than values (i.e. case number II2). In this case, the interface catches the exception and returns an error saying “not enough values. Finally, having more values than attributes is considered to be invalid and needs to be caught (i.e. case number II3). This is done by the interface and an error message saying “too many values” is shown.

Thirdly, invalid SQL UPDATE statements were also tested. Table 8 shows the results of this. There were only two exceptions that needed to be tested. This includes missing the SET keyword from the UPDATE statement (i.e. case number IU1). In this case the interface returns an error saying "missing SET keyword", and thus, this exception is caught by the interface. The other exception is when the table name is not specified in the UPDATE statement (i.e. case number IU2). This will display an error saying "invalid table name", and hence, this exception is caught by the interface.

Finally, invalid SQL DELETE statements were tested. Table 9 shows the results of this. There were only two exceptions that needed to be tested. Both of them are similar and test the case of not specifying the table name in the DELETE statement (i.e. case numbers ID1 & ID2). Both exceptions are caught by the interface and an error message saying "invalid table name" is shown.

Therefore, the translation layer that was implemented is made robust in terms of not allowing invalid SQL statements to be translated to Tutorial D statements, and thus, they are not executed on Rel. Suitable error messages will be shown to the user which can make the debugging process of finding the source of errors an easier process.

10.4 Summary

Through analysing the testing results gathered from the previous section, it was shown that the translation layer between SQL and Tutorial D is a feasible approach to preserve the interoperability between the two languages. This was shown through testing the execution of valid SQL statements using the SQL-to-Tutorial D interface built and showing that they are translated to valid Tutorial D statements, using the translation logic defined. These resultant Tutorial D statements are then executed automatically on the integrated database server component of Rel showing the results in the browser window component of Rel. In addition, it was shown that SQL deficiencies are solved through interoperating with Tutorial D utilising the true relational features of it. Finally, the translation layer is tested against robustness. It was shown that it is made robust by not allowing invalid SQL statements to be translated and returning a suitable error message to the SQL users.

11. Conclusions and Future Work

Translating between SQL and Tutorial D can provide many advantages in the database field. The basic advantage is embracing the relational model of data and moving towards the right direction in the database field. It was shown that the relational model of data is a powerful and efficient model that will remain in use for the upcoming future, and therefore, cannot be ignored. Although SQL is the most widely used relational database language in this regard because of its standardisation, it falls short in many aspects and is deficient in complying with the relational model of data. Although, Tutorial D came into existence and one of its objectives was to address those deficiencies of SQL, its adoption can be difficult because of the legacy factor of SQL. Thus, translating between SQL and Tutorial D extends the basic advantage mentioned earlier and can be a suitable solution for this problem.

This dissertation aimed at implementing a translation layer between SQL and Tutorial D and studying its importance and needs in solving SQL deficiencies. It first provided a review on the relevant literature regarding SQL legacy, SQL deficiencies and how Tutorial D addresses them. It then studied other attempts made at translation between SQL and other database languages, such as AmosQL and XQuery. The core of the dissertation came next as a translation layer between SQL and Tutorial D was implemented through direct string manipulation. After testing the implemented translation layer, it was shown that such translation is feasible.

Hence, legacy SQL clients can now use the translation layer built to execute SQL statements on a Tutorial D database utilising the true relational features of Tutorial D. What is more, SQL deficiencies that are addressed by Tutorial D are solved through using the translation layer, while at the same time, SQL is still used because of its legacy. Thus, the translation layer between SQL and Tutorial D is a feasible approach to preserve the interoperability of the two languages and can lead to a better way of manipulating relational databases.

However, due to the limited scope of this dissertation, the translation layer built has some limitations that can be considered for future work. As mentioned in the scope of the dissertation, the translation layer aims at translating a small subset of the relational language. Thus, there are some complex SQL statements that are not handled by the translation layer built fir this research. These include joining multiple tables in the SELECT query, ordering and grouping the result of the SQL SELECT statement. Although these complex SQL statements are not handled by the translation layer, the SQL-to-Tutorial D interface of the translation layer was built to catch those complex SQL statements and return a message to the users saying that the statements they have executed are not currently handled by the translation layer. Appendix 7 provides some examples of these complex SQL statements and the messages returned to the users. In addition to these complex SQL statements, the translation layer was built with the assumption that the relations to be manipulated already exist at the Tutorial D end as the translation layer does not handle the creation of relations through SQL.

All of the above limitations can imply a suitable extension on the translation layer for future work. However, it can be a better approach to attempt translating those by following one of the approaches mentioned earlier when translating SQL to AmosQL and SQL to XQuery. This is because both approaches considered a large subset of the relational language and are proven to be feasible. Further work and study on this is still need to be conducted in future.

To sum up, this dissertation implemented a translation layer between SQL and Tutorial D based on a small subset of the relational language. Other objectives of the dissertation were also achieved and questions were answered as mentioned above. Thus, this dissertation showed that translating between SQL and Tutorial D can be a feasible and a suitable approach to solve some of the problems that exist in the current trends of manipulating relational databases.

12. References

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., & Zdonik, S. (1989). *The object-oriented database system manifesto*, in Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89). Kyoto, Japan, Dec 4-6, 223-240.

Barker, R. (1990). *Case*Method: Entity relationship modelling.* USA: Addison-Wesley Inc.

Burns, R. K. (1990). Referential secrecy. *IEEE, Symposium on Security and Privacy*, 133-142.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377-387.

Codd, E. F. (1981). Data models in database management. *ACM SIGMOD Record*, *11*(2), 112-114.

Codd, E. F. (1990). *The relational model for database management: version 2.* Massachusetts, USA: Addison Wesley Longman Inc.

Date, C. J. (1983). Null values in database management. In C. J. Date (Ed.), *Relational database: Selected writings.* Massachusetts, USA: Addison-Wesley.

Date, C. J. (1984). A critique of the SQL database language. In C. J. Date (Ed.), *Relational database: Selected writings.* Massachusetts, USA: Addison-Wesley.

Date, C. J. (1988). Why relational. In C. J. Date & A. Warden (Ed.), *Relational database: Writings.* Massachusetts, USA: Addison-Wesley.

Date, C. J. (1990). Why duplicate rows are prohibited. In C. J. Date & A. Warden, *Relational database: Writings.* Massachusetts, USA: Addison-Wesley.

Date, C. J. (2003). A sweet disorder. In C. J. Date (Ed.), *Date on database: Writings 2000 – 2006.* California, USA: Apress.

Date, C. J., & Darwen, H. (1997). *A guide to the SQL standard* (4th ed.). USA: Addison-Wesley Inc.

Date, C. J., & Darwen, H. (2000). *Foundation for future database systems: The third Manifesto* (2nd ed.). Massachusetts, USA: Addison Wesley Longman Inc.

Date, C. J., & Darwen, H. (2007). *Databases, types, and the relational model: The third manifesto* (3rd ed.). USA: Pearson Education, Inc.

Eisenberg, A., & Melton, J. (1998). Standards in practice. *ACM SIGMOND Record. 27*(3), 53-58.

Eisenberg, A., & Melton, J. (2000). SQL standardization: The next steps. *ACM SIGMOND Record*, *29*(1): 63-67.

GNU General Public License. (2007). Retrieved September 13, 2007, from http://www.gnu.org/copyleft/gpl.html

ISO (2003) *Database Language SQL, International standard ISO/IEC 9075:2003*. International Organisation for Standardisation (ISO). Geneva, Switzerland.

Jagerskogh, M. (2005). *Translating SQL expressions to functional queries in a mediator database system.* Unpublished master's thesis, Uppsala University, Uppsala, Sweden.

Jigyasu, S., Banerjee, S., Borkar, V., Carey, M., Dixit, K., Malkani, A. & Thatte, S. (2006). *SQL to XQuery translation in the AquaLogic data services platform*, in Proceedings of the 22nd International Conference on Data Engineering (ICDE'06). p. 97.

Rel: An implementation of Date and Darwen's Tutorial D. (2007). Retrieved August 29, 2007, from http://dbappbuilder.sourceforge.net/Rel.html

Silberschatz, A., Korth, H., & Sudarshan, S. (2002). *Database System Concepts.* (4th Ed.). Singapore: McGraw-Hill

Soderstrom, E. (2002). Standardising the business vocabulary of standards, *ACM SIGMOND Record*, 1048-1052.

Stonebraker, M., Rowe, L. A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., & Beech, D. (1990) Third-generation database system manifesto. *ACM SIGMOD Record*, *19*(3), 31-44.

Appendix

Appendix 1: Employee Table - SQL “Create Table” Code

```
/*
* Employee table SQL "Create Table" Code
*/

CREATE TABLE Employee (
	Employee_ID INTEGER NOT NULL,
	Firstnames VARCHAR(32) NOT NULL,
	Surname VARCHAR(32) NOT NULL,
	Street VARCHAR(32) NOT NULL,
	Suburb VARCHAR(32) NOT NULL,
	Phone VARCHAR(32) NOT NULL,
	Email VARCHAR(32) NOT NULL,
	Date_Entered VARCHAR(32) NOT NULL,
	Date_Employed VARCHAR(32) NOT NULL,
	Salary NUMBER(7,2),
	Qualification VARCHAR(32),
	-----------------------------------
	CONSTRAINT Employee_PK		PRIMARY KEY (Employee_ID)
);
```

Appendix 2: Employee Table - SQL “Insertion Script” Code

```
/*
* Employee table SQL “Insertion Script” Code
*/

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (1, 'Sewal', 'Kirby', '40 Ashmore Road', 'Mornington', '(03) 427-6515', '-',
'1996-10-15 08:11:00', '1996-11-27', 41688.0, 'Senior');

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (2, 'Philadelphia', 'Allsop', '5/36 Nicholson Road', 'Northfield', '(03) 483-
2144', '-', '2001-06-01 08:43:00', '2001-07-11', 41100.0, 'Senior');

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (3, 'Raoul', 'Brocket', '137 Maybank Road', 'Mornington', '(03) 486-8179', '-
', '1990-11-11 09:01:00', '1990-11-15', 34538.0, 'Senior');

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (4, 'Gwladys', 'Corkran', '47B Brooklyn Street', 'Roseneath', '(03) 487-4680',
'-', '1995-05-18 09:01:00', '1995-07-06', 41964.0, 'Senior');

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (5, 'Boris', 'Belt', '76 Green Terrace', 'Kensington', '(03) 460-9801', '-',
'1990-08-12 08:40:00', '1990-09-29', 31049.0, 'Senior');

INSERT INTO Employee (Employee_ID, Firstnames, Surname, Street, Suburb,
Phone, Email, Date_Entered, Date_Employed, Salary, Qualification)
VALUES (6, 'Jane', 'Dassen', '12 Chapman Heights', 'Brentwood', '(03) 497-0752', '-',
'1990-08-16 09:13:00', '1990-09-10', 43610.0, 'Senior');
```

Appendix 3: Employee Relation - Tutorial D Code

```
/*
        File: $Id: employee-schema.d
        Base relation definition.
*/

VAR Employee REAL RELATION
{
        Employee_ID INTEGER,
        Firstnames CHAR,
        Surname CHAR,
        Street CHAR,
        Suburb CHAR,
        Phone CHAR,
        Email CHAR,
        Date_Entered CHAR,
        Date_Employed CHAR,
        Salary RATIONAL,
        Qualification CHAR
} Key {Employee_ID};
```

Appendix 4: Employee Relation -Tutorial D “Insertion Script” Code

```
/*
	File: $Id: employee-data.d

*/

DELETE Employee;

INSERT Employee RELATION
{
	TUPLE { Employee_ID 1, Firstnames "Sewal", Surname "Kirby", Street "40
Ashmore Road", Suburb "Mornington", Phone "(03) 427-6515", Email "-",
Date_Entered "1996-10-15 08:11:00", Date_Employed "1996-11-27", Salary 41688.0,
Qualification "Senior" },
	TUPLE { Employee_ID 2, Firstnames "Philadelphia", Surname "Allsop",
Street "5/36 Nicholson Road", Suburb "Northfield", Phone "(03) 483-2144", Email "-
", Date_Entered "2001-06-01 08:43:00", Date_Employed "2001-07-11", Salary
41100.0, Qualification "Senior" },
	TUPLE { Employee_ID 3, Firstnames "Raoul", Surname "Brocket", Street
"137 Maybank Road", Suburb "Mornington", Phone "(03) 486-8179", Email "-",
Date_Entered "1990-11-11 09:01:00", Date_Employed "1990-11-15", Salary 34538.0,
Qualification "Senior" },
	TUPLE { Employee_ID 4, Firstnames "Gwladys", Surname "Corkran", Street
"47B Brooklyn Street", Suburb "Roseneath", Phone "(03) 487-4680", Email "-",
Date_Entered "1995-05-18 09:01:00", Date_Employed "1995-07-06", Salary 41964.0,
Qualification "Senior" },
	TUPLE { Employee_ID 5, Firstnames "Boris", Surname "Belt", Street "76
Green Terrace", Suburb "Kensington", Phone "(03) 460-9801", Email "-",
Date_Entered "1990-08-12 08:40:00", Date_Employed "1990-09-29", Salary 31049.0,
Qualification "Senior" },
	TUPLE { Employee_ID 6, Firstnames "Jane", Surname "Dassen", Street "12
Chapman Heights", Suburb "Brentwood", Phone "(03) 497-0752", Email "-",
Date_Entered "1990-08-16 09:13:00", Date_Employed "1990-09-10", Salary 43610.0,
Qualification "Senior" },
	TUPLE { Employee_ID 7, Firstnames "Sextus", Surname "Derbie", Street
"65A Columbia Lane", Suburb "Roseneath", Phone "(03) 460-9377", Email "-",
Date_Entered "2000-04-07 08:08:00", Date_Employed "2000-04-27", Salary 27470.0,
Qualification "Senior" },
	TUPLE { Employee_ID 8, Firstnames "Evelyn", Surname "Fairbirn", Street
"144 Buford Road", Suburb "Mornington", Phone "(03) 424-3714", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
	TUPLE { Employee_ID 9, Firstnames "Ewen", Surname "Larroche", Street
"57 Oxford Crescent", Suburb "Northfield", Phone "(03) 438-7290", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
```

TUPLE { Employee_ID 10, Firstnames "Glynis", Surname "Henare", Street "14 Brighton Street", Suburb "Newington", Phone "(03) 431-4824", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 11, Firstnames "Frank", Surname "Cecchi", Street "111 Passmore Road", Suburb "Richmond", Phone "(03) 461-0765", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 12, Firstnames "Tess", Surname "Lapslie", Street "52 Somerville Street", Suburb "Lovemore Heights", Phone "(03) 410-4089", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 13, Firstnames "Rufus", Surname "Scown", Street "26 Seaforth Road", Suburb "Kew", Phone "(03) 485-1727", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 14, Firstnames "Loretta", Surname "Nader", Street "1/85 Ranfurly Street", Suburb "Armadale", Phone "(03) 484-3624", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 15, Firstnames "Stanley", Surname "Malcolm", Street "70 Fenton Street", Suburb "Hampstead", Phone "(03) 423-7709", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 16, Firstnames "Berenice", Surname "Niksic", Street "116 Mulford Avenue", Suburb "Epping", Phone "(03) 478-1067", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 17, Firstnames "Sharon", Surname "Purwo", Street "65 Goodall Street", Suburb "Glencoe", Phone "(03) 448-8020", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 18, Firstnames "Quenild", Surname "Hopwood", Street "11C Churchill Avenue", Suburb "Mornington", Phone "(03) 425-8442", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 19, Firstnames "Darby", Surname "Macassey", Street "6/15 Barton Crescent", Suburb "Walmer", Phone "(03) 415-2526", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 20, Firstnames "Gloria", Surname "Gribben", Street "148 Vine Street", Suburb "Ipswitch", Phone "(03) 449-2910", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 21, Firstnames "Ned", Surname "Hitchings", Street "33 Oates Street", Suburb "Forbury", Phone "(03) 428-0389", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },

TUPLE { Employee_ID 22, Firstnames "Ada", Surname "Julian", Street "75 Marlowe Road", Suburb "St Kilda", Phone "(03) 470-8323", Email "-", Date_Entered

```
"2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification
"Senior" },
        TUPLE { Employee_ID 23, Firstnames "Leopold", Surname "Horder", Street
"134 Sycamore Road", Suburb "Woodstock", Phone "(03) 482-9772", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 24, Firstnames "Eleanora", Surname "Napper", Street
"28 Bentley Road", Suburb "Grange Hill", Phone "(03) 482-8003", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 25, Firstnames "Randal", Surname "Hyndman",
Street "7/121 Wharfdale Place", Suburb "Albany", Phone "(03) 438-6641", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 26, Firstnames "Marina", Surname "Doust", Street
"28 Jubilee Road", Suburb "Kensington", Phone "(03) 418-3421", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 27, Firstnames "Barney", Surname "Gilbert", Street
"126B Russell Road", Suburb "Newington", Phone "(03) 497-2641", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 28, Firstnames "Astrid", Surname "Gurnsey", Street
"89 Evans Street", Suburb "Armadale", Phone "(03) 415-3640", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 29, Firstnames "Eugene", Surname "Kumar", Street
"43 Factory Lane", Suburb "St Bathans", Phone "(03) 489-1922", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 30, Firstnames "Ivy", Surname "Couch", Street "28
Church Terrace", Suburb "Northfield", Phone "(03) 450-6125", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 31, Firstnames "Achilles", Surname "Abdul", Street
"44 Seaview Road", Suburb "Roseneath", Phone "(03) 474-6025", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 32, Firstnames "Arabella", Surname "Busch", Street
"6/62 Ashmore Street", Suburb "Rosebank", Phone "(03) 403-6576", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 33, Firstnames "Melchior", Surname "Ball", Street
"68 Beach Heights", Suburb "Lovemore Heights", Phone "(03) 454-2762", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
        TUPLE { Employee_ID 34, Firstnames "Consuelo", Surname "Bacher", Street
"134A Tweed Crescent", Suburb "Brentwood", Phone "(03) 408-8430", Email "-",
Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0,
Qualification "Senior" },
```

TUPLE { Employee_ID 35, Firstnames "Aubrey", Surname "Ballock", Street "48 Banks Road", Suburb "Caversham", Phone "(03) 440-5507", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },
TUPLE { Employee_ID 36, Firstnames "Minerva", Surname "Aitken", Street "80 Wesley Street", Suburb "Richmond", Phone "(03) 478-4075", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },
TUPLE { Employee_ID 37, Firstnames "Jehu", Surname "Ayoub", Street "121 Brooklyn Street", Suburb "Glenview", Phone "(03) 466-4738", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" },
TUPLE { Employee_ID 38, Firstnames "Flavia", Surname "Abnernethy", Street "47 Nicholson Street", Suburb "Kew", Phone "(03) 407-7879", Email "-", Date_Entered "2001-06-26 08:08:00", Date_Employed "2001-08-06", Salary 34844.0, Qualification "Senior" }
};

Appendix 5: The Translation Logic - Translate.java

```
/*
 * Created on Aug 29, 2007
 *
 * The purpose of this class is to handle the translation logic
 *
 * @author: Ahmed Almulla
 */
package translator;

import java.util.ArrayList;

public class Translate {

      private String TutDString = "";

      private String TutDStringResult = "";

      public String EvaluateExpr(String SQLString)
      throws IllegalArgumentException {

            /*
             * Trim the SQLString from the whitespaces in the beginning
             * and the end and convert to UpperCase so that case will be
             * ignored
             * This satisfies the case insensetivity of SQL
             */
            String trimmedSQLString = SQLString.trim();
            String preparedSQLString = prepareSQLString(trimmedSQLString);
            String UpperCaseSQLString = preparedSQLString.toUpperCase();

            if (UpperCaseSQLString.indexOf("SELECT") == 0) {
                  TutDStringResult = TranslateSelect(preparedSQLString);
            } else if (UpperCaseSQLString.indexOf("INSERT INTO") == 0) {
                  TutDStringResult = TranslateInsert(preparedSQLString);
            } else if (UpperCaseSQLString.indexOf("UPDATE") == 0) {
                  TutDStringResult = TranslateUpdate(preparedSQLString);
            } else if (UpperCaseSQLString.indexOf("DELETE FROM") == 0) {
                  TutDStringResult = TranslateDelete(preparedSQLString);
            } else {
                  throw new IllegalArgumentException(
                              "This translation layer was "
                              + "designed to only handle simple SELECT,
                              INSERT, UPDATE and DELETE statements");
            }
            return TutDStringResult;
      }

      public String TranslateSelect(String SQLString)
      throws IllegalArgumentException {

            TutDString = "";

            int SELECTIndex = SQLString.toUpperCase().indexOf("SELECT");
            int FROMIndex = SQLString.toUpperCase().indexOf("FROM");
            int WHEREIndex = SQLString.toUpperCase().indexOf("WHERE");
            int STARIndex = SQLString.indexOf("*");
            int terminationIndex = SQLString.lastIndexOf(";");

            /*
             * The following are the exception cases that are checked
             * before translation The cases are for Invalid SQL SELECT
             * statements
             * Some of the cases are invalid for the purpose of this
             * translation layer (e.g no complicated SELECT statements)
             * If any of them occur, the translation never happens and an
             * apprpriate error message is thrown
             */
            if (FROMIndex == -1) {
                  throw new IllegalArgumentException(
                  "FROM keyword not found where expected");
            }

            while ((SQLString.charAt(FROMIndex - 1) != ' ' || SQLString
                        .charAt(FROMIndex + 4) != ' ')) {
            // this is not the FROMIndex wanted
                  FROMIndex = SQLString.toUpperCase().indexOf("FROM",
                  FROMIndex+1);
                  if (FROMIndex == -1) {
                        throw new IllegalArgumentException(
                        "FROM keyword not found where expected");
                  }
            }

            if (FROMIndex - 1 == SELECTIndex + 6) {
                  throw new IllegalArgumentException("missing
                  expression");
            }

            if (WHEREIndex != -1) {
                  while (SQLString.charAt(WHEREIndex - 1) != ' '
                        || SQLString.charAt(WHEREIndex + 5) != ' ') {
                        // this is
                        // not the
                        // WHEREIndex
                        // wanted
                                    WHEREIndex =
            SQLString.toUpperCase().indexOf("WHERE",
                                    WHEREIndex + 1);
                        if (WHEREIndex == -1) {
                              break;
                        }
                  }
            }

            if (FROMIndex > WHEREIndex && WHEREIndex > 0) {
                  throw new IllegalArgumentException(
                  "FROM keyword not found where expected");
            }

            /*
             * Checking the columns
             */
            String columnsCheck = SQLString.substring(SELECTIndex + 7,
                        FROMIndex - 1);

            if (columnsCheck.charAt(columnsCheck.length() - 1) == ',') {
                  throw new IllegalArgumentException("missing
                  expression");
            }
```

```
for (int i = 0; i < columnsCheck.length(); i++) {
      if (columnsCheck.charAt(i) == ' '
            && columnsCheck.charAt(i - 1) != ','
                  && columnsCheck.charAt(i + 1) != ',') {
            throw new IllegalArgumentException("missing
            expression");
      }
}

if (WHEREIndex != -1) {
      if (SQLString.substring(FROMIndex,
      WHEREIndex).contains("*")) {
      throw new IllegalArgumentException("invalid
      SQLStatement");
      }
} else {
      if
      (SQLString.substring(FROMIndex,terminationIndex).contain
      s("*")) {
      throw new IllegalArgumentException("invalid SQL
      Statement");
      }
}

/*
 * The translation layer was designed to handle simple
 * SELECTstatements
 * The following exception cases prevents the user from using
 * some of the complicated SELECT statements
 *
 * No use of: ORDER BY clause
 */
int ORDERIndex = SQLString.toUpperCase().indexOf("ORDER");
if (ORDERIndex != -1) {
      while (SQLString.charAt(ORDERIndex - 1) != ' '
            || SQLString.charAt(ORDERIndex + 5) != ' ') {
            // this is
            // not the
            // ORDERIndex
            // wanted
                                    ORDERIndex =
            SQLString.toUpperCase().indexOf("ORDER",
                        ORDERIndex + 1);
            if (ORDERIndex == -1) {
                  break;
            }
      }
}
if (ORDERIndex != -1) {
throw new IllegalArgumentException(
            "This translation layer was not designed to handle
ORDER BY clause");
}

/*
 * No use of: GROUP BY clause
 */
int GROUPIndex = SQLString.toUpperCase().indexOf("GROUP");
if (GROUPIndex != -1) {
      while (SQLString.charAt(GROUPIndex - 1) != ' '
            || SQLString.charAt(GROUPIndex + 5) != ' ') {
```

```
            // this is
            // not the
            // GROUPIndex
            // wanted
                        GROUPIndex =
SQLString.toUpperCase().indexOf("GROUP",
                        GROUPIndex + 1);
            if (GROUPIndex == -1) {
                  break;
            }
      }
}
if (GROUPIndex != -1) {
throw new IllegalArgumentException(
"This translation layer was not designed to handle GROUP BY
clause");
}

/*
 * No use of: JOIN, Cartesian product
 */
if (WHEREIndex == -1) {
      if
      (SQLString.substring(FROMIndex + 5,
terminationIndex).contains(
      ",")
      || SQLString.substring(FROMIndex + 5, terminationIndex)
      .contains(" ")) {
throw new IllegalArgumentException(
"This translation layer was designed to only handle single
table queries");
      }
} else {
      if (SQLString.substring(FROMIndex + 5, WHEREIndex - 1)
                  .contains(",")
                                    ||
      SQLString.substring(FROMIndex + 5, WHEREIndex - 1)
                  .contains(" ")) {
throw new IllegalArgumentException(
"This translation layer was designed to only handle single
table queries");
      }
}

/*
 * The following code handles the SQL SELECT statement
 * translation
 */
if (WHEREIndex == -1) { // no restriction on the selection
TutDString += SQLString.substring(FROMIndex + 5,
terminationIndex);
if (STARIndex == -1 || STARIndex > FROMIndex) { // projection
// on the
            // selection
            TutDString += "{"
                  + SQLString.substring(SELECTIndex + 7,
                  FROMIndex - 1)
                  + "}";
      }
} else { // restriction on the selection
```

```
        /*
         * Replacing single quotation signs to double quotation signs
         * that are used in Tutorial D
        */
            String SQLStringRestriction =
            SQLString.substring(WHEREIndex + 6,
                        terminationIndex);
            for (int i = 0; i < SQLStringRestriction.length(); i++)
{
                String s1 = "`";
                String s2 = "’";
                String s3 = "'";
                char quotationSign = '"';
                if (SQLStringRestriction.charAt(i) == s1.charAt(0)
                            || SQLStringRestriction.charAt(i) ==
                s2.charAt(0)
                            || SQLStringRestriction.charAt(i) ==
                s3.charAt(0)) {
                    SQLStringRestriction =
                SQLStringRestriction.substring(0, i)
                    + quotationSign
                    + SQLStringRestriction.substring(i + 1);
                }
            }

            TutDString += "("
                + SQLString.substring(FROMIndex + 5, WHEREIndex -
                1) + "["
                + SQLStringRestriction + "])";
        if (SQLString.charAt(SELECTIndex + 7) != '*') {
                // projection on the
                // selection
                TutDString += " {"
                    + SQLString.substring(SELECTIndex + 7,
                FROMIndex - 1)
                    + "}";
            }
        }
        return TutDString;
    }

    public String TranslateInsert(String SQLString) {

        TutDString = "";
        ArrayList<String> columnFields = new ArrayList<String>();
        ArrayList<String> columnValues = new ArrayList<String>();
        String columnField = "";
        String columnValue = "";

        int INSERTIndex = SQLString.toUpperCase().indexOf("INSERT
        INTO");
        int VALUESIndex = SQLString.toUpperCase().indexOf("VALUES");
        int columnsStartIndex = SQLString.toUpperCase().indexOf("(");
        int terminationIndex = SQLString.lastIndexOf(";");

        /*
         * Checking the VALUES keyword
         */
        if (VALUESIndex == -1) {
            throw new IllegalArgumentException("missing VALUES
            keyword");
```

```
}

if (SQLString.charAt(VALUESIndex - 1) != ')'
      && SQLString.charAt(VALUESIndex + 6) != '(') {
                  while ((SQLString.charAt(VALUESIndex - 1) !=
' ' || SQLString
                  .charAt(VALUESIndex + 6) != ' ')
                  && SQLString.charAt(VALUESIndex - 3) != ')'
                        && SQLString.charAt(VALUESIndex - 4)
            != ')') { // this is
            // not the
            // VALUESIndex
            // wanted
                              VALUESIndex =
      SQLString.toUpperCase().indexOf("VALUES",
                        VALUESIndex + 1);
            if (VALUESIndex == -1) {
                  throw new IllegalArgumentException("missing
                  VALUES keyword");
            }
      }
}

TutDString += SQLString.substring(INSERTIndex + 12,
columnsStartIndex)
+ "+= TUPLE {";

/*
 * Storing the column fields in an ArrayList
 */
for (int i = columnsStartIndex + 1; i < VALUESIndex; i++) {
      if (SQLString.charAt(i) == ')') {
            columnFields.add(columnField);
} else if (SQLString.charAt(i) != ' ' && SQLString.charAt(i)
!= ',') {
            columnField += SQLString.charAt(i);
} else if (SQLString.charAt(i) != ' ') {
            columnFields.add(columnField);
            columnField = "";
}
}

/*
 * Storing the column values in an ArrayList
 */
for (int i = VALUESIndex + 6; i < terminationIndex; i++) {
      if (SQLString.substring(i, i + 1).contains("'")
                  || SQLString.substring(i, i +
      1).contains("`")
                  || SQLString.substring(i, i +
      1).contains("’")) {
            char quoteSign = '"';
            columnValue += quoteSign;
      } else if (columnValue.compareTo("") != 0) {
            if (columnValue.charAt(0) == '"') {
                        if
(columnValue.charAt(columnValue.length() - 1) == '"'
                        && columnValue.length() > 1) {
                        columnValues.add(columnValue);
                        columnValue = "";
                  } else {
                        columnValue += SQLString.charAt(i);
                    }
                }
            }

        if ((columnValue.compareTo("") != 0 && columnValue.charAt(0)
        != '"')
                    || columnValue.compareTo("") == 0) {
                if (SQLString.charAt(i) == ')'
                    && columnValue.compareTo("") != 0) {
                    columnValues.add(columnValue);
                } else if (SQLString.charAt(i) != ' '
                    && SQLString.charAt(i) != ','
                        && SQLString.charAt(i) != '(') {
                    columnValue += SQLString.charAt(i);
                } else if (SQLString.charAt(i) != ' '
                    && SQLString.charAt(i) != '('
                        && columnValue.compareTo("") != 0) {
                    columnValues.add(columnValue);
                    columnValue = "";
                }
        }
        }

        if (columnFields.size() == 0) {
            TutDString = SQLString.substring(INSERTIndex + 12,
            VALUESIndex)
            + "+= TUPLE {";
            for (int i = 0; i < columnValues.size(); i++) {
                TutDString += columnValues.get(i);
                if (i == columnValues.size() - 1) {
                    TutDString += "};";
                } else {
                    TutDString += ", ";
                }
            }
        } else if (columnFields.size() == columnValues.size()) {
            for (int i = 0; i < columnFields.size(); i++) {
                TutDString += columnFields.get(i) + " " +
                columnValues.get(i);
                if (i == columnFields.size() - 1) {
                    TutDString += "};";
                } else {
                    TutDString += ", ";
                }
            }
        } else if (columnFields.size() > columnValues.size()) {
            // an exception case
            throw new IllegalArgumentException("not enough values");
        } else if (columnFields.size() < columnValues.size()) {
            // an exception case
            throw new IllegalArgumentException("too many values");
        } else {
            TutDString = "";
        }
        return TutDString;
    }

    public String TranslateUpdate(String SQLString) {

        TutDString = "";
```

```
int UPDATEIndex = SQLString.toUpperCase().indexOf("UPDATE");
int SETIndex = SQLString.toUpperCase().indexOf("SET");
int WHEREIndex = SQLString.toUpperCase().indexOf("WHERE");
int terminationIndex = SQLString.lastIndexOf(";");

if (SETIndex == -1) {
      throw new IllegalArgumentException("missing SET
      keyword");
}

while (SQLString.charAt(SETIndex - 1) != ' '
      || SQLString.charAt(SETIndex + 3) != ' ') {
      // this is not the SETIndex wanted
      SETIndex = SQLString.toUpperCase().indexOf("SET",
      SETIndex + 1);
      if (SETIndex == -1) {
            throw new IllegalArgumentException("missing SET
            keyword");
      }
}

if (SETIndex == UPDATEIndex + 7) {
      throw new IllegalArgumentException("invalid table
      name");
}

if (WHEREIndex != -1) {
      while (SQLString.charAt(WHEREIndex - 1) != ' '
            || SQLString.charAt(WHEREIndex + 5) != ' ') {
            // this is
            // not the
            // WHEREIndex
            // wanted
                                    WHEREIndex =
            SQLString.toUpperCase().indexOf("WHERE",
                                          WHEREIndex + 1);
            if (WHEREIndex == -1) {
                  break;
            }
      }
}

/*
 * Replacing single quotation signs to double quotation
 * signs that are
 * used in Tutorial D
 */
for (int i = 0; i < SQLString.length(); i++) {
      String s1 = "`";
      String s2 = "′";
      String s3 = "'";
      char quotationSign = '"';
      if (SQLString.charAt(i) == s1.charAt(0)
                  || SQLString.charAt(i) == s2.charAt(0)
                  || SQLString.charAt(i) == s3.charAt(0)) {
            SQLString = SQLString.substring(0, i) +
            quotationSign
            + SQLString.substring(i + 1);
      }
}
```

```
            TutDString += SQLString.substring(UPDATEIndex + 7,
            SETIndex - 1)
      + " @= ";

      String prepareSetString = "";

      if (WHEREIndex != -1) {
            TutDString += "["
                  + SQLString.substring(WHEREIndex + 6,
                  terminationIndex)
                  + "] ";
            prepareSetString = SQLString
            .substring(SETIndex + 4, WHEREIndex - 1);
      } else {
            prepareSetString = SQLString.substring(SETIndex + 4,
                        terminationIndex);
      }

      ArrayList<String> columnFields = new ArrayList<String>();
      ArrayList<String> columnValues = new ArrayList<String>();

      while (prepareSetString.indexOf("=") != -1) {
            int equalSignIndex = prepareSetString.indexOf("=");
            int commaIndex =
            prepareSetString.toUpperCase().indexOf(",");
            columnFields.add(prepareSetString.substring(0,
            equalSignIndex));
            if (commaIndex != -1) {

columnValues.add(prepareSetString.substring(equalSignIndex+1,
                              commaIndex));
                  prepareSetString =
                  prepareSetString.substring(commaIndex+1);
            } else {

columnValues.add(prepareSetString.substring(equalSignIndex+1,
                              prepareSetString.length()));
                  break;
            }
      }

      String finalSetString = "";
      if (columnFields.size() == columnValues.size()) {
            for (int i = 0; i < columnFields.size(); i++) {
                  finalSetString += columnFields.get(i) + ":="
                  + columnValues.get(i);
                  if (i != columnFields.size() - 1) {
                        finalSetString += ", ";
                  }
            }
      }

      TutDString += "(" + finalSetString + ");";

      return TutDString;
}

public String TranslateDelete(String SQLString) {

      TutDString = "";
```

```
int DELETEIndex = SQLString.toUpperCase().indexOf("DELETE
FROM");
int WHEREIndex = SQLString.toUpperCase().indexOf("WHERE");
int terminationIndex = SQLString.lastIndexOf(";");

/*
 * Replacing single quotation signs to double quotation signs
 * that are
 * used in Tutorial D
 */
for (int i = 0; i < SQLString.length(); i++) {
      String s1 = "`";
      String s2 = "’";
      String s3 = "'";
      char quotationSign = '"';
      if (SQLString.charAt(i) == s1.charAt(0)
                  || SQLString.charAt(i) == s2.charAt(0)
                  || SQLString.charAt(i) == s3.charAt(0)) {
            SQLString = SQLString.substring(0, i) +
            quotationSign
            + SQLString.substring(i + 1);
      }
}

if (WHEREIndex != -1) {
      while (SQLString.charAt(WHEREIndex - 1) != ' '
            || SQLString.charAt(WHEREIndex + 5) != ' ') {
            // this is
            // not the
            // WHEREIndex
            // wanted
                                    WHEREIndex =
            SQLString.toUpperCase().indexOf("WHERE",
                                    WHEREIndex + 1);
            if (WHEREIndex == -1) {
                  break;
            }
      }
}

if (WHEREIndex != -1) {
      if (DELETEIndex + 12 == WHEREIndex) {
            throw new IllegalArgumentException("invalid table
      name");
      }
                        TutDString +=
      SQLString.substring(DELETEIndex + 12, WHEREIndex)
      + "-= ["
      + SQLString.substring(WHEREIndex + 6, terminationIndex)
      + "];";
} else { // Delete all records
      if (DELETEIndex + 11 == terminationIndex
                  || DELETEIndex + 12 == terminationIndex) {
            throw new IllegalArgumentException("invalid table
            name");
      }
      TutDString += SQLString.substring(DELETEIndex + 12,
                  terminationIndex)
                  + "-= ALL;";
}
```

```
        return TutDString;
    }

    /*
     * This method gets rid of the subsequent whitespaces that may
     * appear in a SQL statement before translation
     */
    public String prepareSQLString(String SQLString) {

        String newSQLString = "";
        ArrayList<String> characterArray = new ArrayList<String>();

        for (int i = 0; i < SQLString.length(); i++) {
            String s = "" + SQLString.charAt(i);
            characterArray.add(s);
        }

        for (int i = 0; i < characterArray.size(); i++) {
            while ((characterArray.get(i)).compareTo(" ") == 0
                        && (characterArray.get(i +
                        1)).compareTo(" ") == 0) {
                characterArray.remove(i + 1);
            }
        }

        for (int i = 0; i < characterArray.size(); i++) {
            newSQLString += characterArray.get(i);
        }

        /*
         * inserting ";" in the end if the user did not enter it
         */
        int lastTerminationIndex = newSQLString.lastIndexOf(";");
        if (lastTerminationIndex != newSQLString.length() - 1) {
            newSQLString += ";";
        }
        return newSQLString;
    }
}
```

Appendix 6: Executing SQL Statements on Rel

```
/**
       * This method initializes btnExecute
       *
       * @return javax.swing.JButton
       */
      private JButton getBtnExecute() {
            if (btnExecute == null) {
                  btnExecute = new JButton();
                  btnExecute.setBounds(new Rectangle(121, 165, 205, 42));
                  btnExecute.setText("Execute");
                              btnExecute.setText("Execute");
                              btnExecute.addActionListener(new
            java.awt.event.ActionListener() {
                        public void
            actionPerformed(java.awt.event.ActionEvent e) {
                              System.out.println("actionPerformed()");
                        // TODO Auto-generated Event stub
                  actionPerformed()
                              txtTutD.setText("");
                              SQLString = txtSQL.getText();
                              Translate t = new Translate();
                              try {
                                    TutDString =
                                    t.EvaluateExpr(SQLString);
                                    txtTutD.setText(TutDString);

                     /* Running the Translated SQL statement into Rel */
                          String runMe = txtTutD.getText().trim();
                          PanelCommandline pcl =
                          browser.getPanelCommandline();
                          remote = pcl.getRemote();

                          if (runMe.indexOf(';') >= 0) {
                              remote.run(runMe);
                          } else {
                              remote.evaluate(runMe);
                          }
                        } catch (IllegalArgumentException e1) {
                                    JOptionPane.showMessageDialog(new
                                    JFrame(), e1.getMessage(),
                                     "error", JOptionPane.ERROR_MESSAGE);
                              } catch (IOException ioe) {
                              System.out.println(ioe.toString());
                          }
                        }
                  });
            }
            return btnExecute;
      }
```

Appendix 7: Handling Complex SQL statements

SQL Statement	**Resulted Tutorial D Statement**	**Error Message**
SELECT * FROM Employee ORDER BY Firstnames;	None	This translation layer was not designed to handle ORDER BY clause
SELECT count(*), Qualification FROM Employee GROUP BY Qualification;	None	This translation layer was not designed to handle GROUP BY clause
SELECT * FROM Employee, Company;	None	This translation layer was designed to only handle single table queries
SELECT * FROM Employee NATURAL JOIN Company;	None	This translation layer was designed to only handle single table queries

Table 10: Handling Complex SQL Statements