Linking CORAL to MySQL and PostgreSQL

Guang Wang

A Thesis
In
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 2004

©Guang Wang , 2004

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-91132-2
Our file Notre référence
ISBN: 0-612-91132-2

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

Abstract

Linking CORAL to MySQL and PostgreSQL

Guang Wang

A Graph Database System is developed at Concordia University. In this system, CORAL is used as the deductive engine. Since MySQL and PostgreSQL are selected to manage the persistent data, connections between CORAL and MySQL as well as PostgreSQL need to be established.

This thesis mainly proposes a solution for these connections. Firstly, CORAL's architecture is analyzed. Its relational database interface is described. Class diagrams, object diagrams, and interaction diagrams are drawn to illustrate CORAL's rules and to identify the requirements. Then, classes are designed as an extension of CORAL's structure to communicate with MySQL and PostgreSQL. Data type conversion rules are defined based on the characteristics of each database system. Classes and parent classes are described in detail and patterns applied in these classes are discussed for the reusability of the design. Additionally, databases for MySQL and PostgreSQL are created and the extended CORAL system is tested. Finally, a CORAL client process is designed and implemented so as to integrate the TGL translator, which translates from the GraphLog graph database language to CORAL, with client-server mode CORAL and communicate with the underlying relational databases.

The implementation of this solution uses C++ in a UNIX environment.

Acknowledgement

My thanks go out to all those who helped me during my thesis period! Especially:

Grateful thanks to Dr. Gregory Butler for his patience and invaluable guidance.

Thanks to the thesis examiners for their precious advice to improve this work.

Thanks to Mr. Stan Swiercz, who helped me a lot while I was in the jungle of compilation errors.

Thanks to my colleagues, including Yue Wang, Christopher Baker, Ju Wang, Fang Lin, and Liqian Zhou, for their kindly help during different steps of my work.

Finally, special thanks to my wife, Zhaoxia Sang, for her selfless support and endless love!

Table of Contents

List of Figures viii
List of Tables ix
Chapter 1 Introduction 1
1.1 Motivation 1
1.2 Contribution of the Thesis 2
1.3 Organization of the Thesis 3
Chapter 2 Background 4
2.1 CORAL 4
2.1.1 System architecture 4
2.1.2 Runtime data management 6
2.1.3 Extensional database interface 7
2.1.4 Client-server model 8
2.1.5 Optimization strategies 8
2.2 MySQL 10
2.2.1 History 10
2.2.2 C API 11
2.2.3 Supported data type 11
2.3 PostgreSQL 12
2.3.1 History 12
2.3.2 C API 13
2.3.3 Supported data type 14
2.4 Design Pattern 14
2.4.1 Abstract factory 15
2.4.2 Façade 16
2.4.3 Command 16
2.4.4 Iterator 17
2.4.5 Template method 18
Chapter 3 Requirement 19
3.1 Requirement Definition 19
3.2 Functional Requirement and Specification 21
3.2.1 Connection establishment 22
3.2.1.1 Open database 22
3.2.1.2 Open MySQL database 24
3.2.1.3 Open PostgreSQL database 25
3.2.1.4 Map table 26
3.2.1.5 Map MySQL table 28
3.2.1.6 Map PostgreSQL table 29
3.2.1.7 Join mapped table 30
3.2.2 Relation process 32
3.2.2.1 Query table 32
3.2.2.2 Retrieve tuple set 34

3.2.2.3 Insert tuple 35
3.2.2.4 Delete tuple 36
3.2.3 Execute command 38
3.2.3.1 Execute command 38
3.2.3.2 Execute MySQL command 40
3.2.3.3 Execute PostgreSQL command 41
3.2.4 Transactional support 42
3.2.4.1 Commit database 42
3.2.4.2 Rollback database 44
3.3 Non-functional Requirement and Specification 46
3.3.1 Computer hardware and software requirement 46
3.3.2 System performance and reusability 46
Chapter 4 Design 47
4.1 Architecture Design 47
4.1.1 CORAL system structure 47
4.1.1.1 Classes for creating CORAL workspaces, relations, and tuples 47
4.1.1.2 Classes for managing CORAL workspaces and relations 49
4.1.1.3 Runtime workspaces and relations structure 49
4.1.1.4 Relational database interface 51
4.1.2 Classes for MySQL and PostgreSQL 54
4.1.2.1 Classes for MySQL 55
4.1.2.2 Classes for PostgreSQL 56
4.1.3 Behavioral modeling 56
4.1.3.1 Establish database connections 57
4.1.3.2 Establish table connections 58
4.1.3.3 Create relations based on RDB relations 59
4.1.3.4 Relation process 60
4.1.3.5 Execute command 62
4.1.3.6 Transactional support 63
4.1.4 Design patterns used 63
4.1.4.1 Abstract factory pattern 64
4.1.4.2 Façade pattern 64
4.1.4.3 Command pattern 65
4.1.4.4 Iterator pattern 66
4.1.4.5 Template method pattern 67
4.2 Major Classes Specifications 67
4.2.1 RDB system classes 68
4.2.1.1 The RdbSystem class 68
4.2.1.2 The RdbMysqlSystem class 69
4.2.1.3 The RdbPostgresqlSystem class 69
4.2.2 RDB database classes 70
4.2.2.1 The RdbDatabase class 71
4.2.2.2 The RdbMysqlDatabase class 71

4.2.2.3 The RdbPostgresqlDatabase class 72
4.2.3 RDB query classes 73
4.2.3.1 The RdbQuery class 74
4.2.3.2 The RdbMysqlQuery class 74
4.2.3.3 The RdbMysqlQueryResult class 75
4.2.3.4 The RdbPostgresqlQuery class 76
4.2.3.5 The RdbPostgreSQLQueryResult class 76
4.2.4 RDB insert classes 77
4.2.4.1 The RdbInsert class 77
4.2.4.2 The RdbMysqlInsert class 78
4.2.4.3 The RdbPostgresqlInsert class 78
4.2.5 RDB delete classes 78
4.2.5.1 The RdbDelete class 79
4.2.5.2 The RdbMysqlDelete class 79
4.2.5.3 The RdbPostgresqlDelete class 79
4.3 Database Design 80
Chapter 5 Test 81
5.1 Test Requirement 81
5.2 Test Methods 82
5.3 Test Cases 82
5.3.1 Connect CORAL to MySQL 82
5.3.2 Connect CORAL to PostgreSQL 83
5.3.3 Create CORAL mapped tables based on MySQL tables 84
5.3.4 Create CORAL mapped tables based on PostgreSQL tables 85
5.3.5 Create CORAL joined table 86
5.3.6 Query mapped tables 87
5.3.7 Insert tuple 88
5.3.8 Delete tuple 89
5.3.9 Execute command 89
5.3.10 Transactional support 90
Chapter 6 Application 92
6.1 System Architecture 93
6.2 Integrating CORAL with the TGL Translator 95
6.3 Optimization Experiment 96
Chapter 7 Conclusion 98
References 99
Appendix 101
A. University Data Model Schema (script files) for MySQL 101
B. University Data Model Schema (script files) for PostgreSQL 105

List of Figures

Figure 2-1 CORAL System Architecture [1] 6
Figure 2-2 Abstract Factory Pattern [6] 15
Figure 2-3 Façade Pattern [6] 16
Figure 2-4 Command Pattern [6] 17
Figure 2-5 Iterator Pattern [6] 17
Figure 2-6 Template Method Pattern [6] 18
Figure 3-1 Use Case Diagram 20
Figure 4-1 CORAL Workspace Classes 48
Figure 4-2 CORAL Relation Classes 48
Figure 4-3 CORAL Tuple Classes 49
Figure 4-4 CORAL Workspace and Relation Management Classes 49
Figure 4-5 CORAL System Runtime Snapshot 50
Figure 4-6 CORAL Main RDB Classes Inheritance Hierarchy 51
Figure 4-7 CORAL RDB Classes Dependency Relationship 53
Figure 4-8 CORAL RDB Classes for Join Actions 53
Figure 4-9 CORAL Relational Database Interface 54
Figure 4-10 Extended CORAL RDB Class Diagram 55
Figure 4-11 Open a MySQL Database 57
Figure 4-12 Map a MySQL Table 58
Figure 4-13 Join Mapped Tables 59
Figure 4-14 Query a Mapped Table 60
Figure 4-15 Insert a Tuple to the Backend Table 61
Figure 4-16 Execute Command at the Backend MySQL 62
Figure 4-17 Commit the Backend MySQL Database 63
Figure 4-18 Abstract Factory Pattern in RDB Classes 64
Figure 4-19 Façade Pattern in RDB Classes 65
Figure 4-20 Command Pattern in RDB Classes 66
Figure 4-21 Iterator Pattern in RDB Classes 66
Figure 4-22 Template Method Pattern in RDB Classes 67
Figure 4-23 RDB System Classes 68
Figure 4-24 RDB Database Classes 70
Figure 4-25 RDB Query Classes 73
Figure 4-26 RDB Insert Classes 77
Figure 4-27 RDB Delete Classes 78
Figure 6-1 Graph Database System Architecture 93

List of Tables

Table 2-1 CORAL RDB Command Set 7
Table 2-2 CORAL System Level Annotations 9
Table 2-3 CORAL User Level Annotations 9
Table 2-4 MySQL C API Commands Used in Implementation 11
Table 2-5 MySQL Data type to CORAL Data Type Conversion 12
Table 2-6 PostgreSQL C API Commands Used in the Implementation 13
Table 2-7 PostgreSQL Data Type to CORAL Data type Conversion 14
Table 6-1. Relative Query Execution Time 97

Chapter 1 Introduction

Due to the intensive data that have been generated by genomics projects, data management, fast access and data mining are at the heart of bioinformatics. While relational databases are widely applied within the industry, there has been considerable research into deductive and graph databases to extend the capabilities of relational databases. Deductive databases allow a view to be defined using logical rules, and allow logical queries against the view. Since the rules allow recursive definitions, the resulting expressive power of the query language is greater than the relational query languages. Graph query languages are even more expressive, while having the very important property of a visual representation. Diagrams are an intuitive way for scientists to pose queries to relational, object- relational, and object databases. They allow the full range of queries, from the very simple to the very complex, to be much more easily expressed and understood than SQL-like languages or form-based queries, which are less intuitive to scientists.

1.1 Motivation

A graph database system is established at Concordia University (described in Chapter 6). It is to apply the benefits of diagrammatic queries, deductive query language, and visualization of results more broadly in genomics. In this system, CORAL[1] works as the deductive engine to evaluate queries and deduce results. Persistent relations are required to supply and store facts for CORAL. From version 1.2, CORAL tried to access data stored in relational databases. Two types of relational databases, Sybase and

Commercial Ingres, were connected with CORAL and the beta code was released. Since both of these databases are commercial products, the potential performance improvement on the database side solely depends on the software suppliers. Most genomics projects use the open-source MySQL and PostgreSQL DBMS, so we require CORAL to interact with them. MySQL and PostgreSQL are two advanced DBMS under open-source agreement and their C application program interfaces provide convenient ways to access the data in the databases from CORAL.

1.2 Contribution of the Thesis

This thesis aims at extending CORAL's structure so as to connect it with MySQL and PostgreSQL. The main achievements this thesis contributes are as follows:

- Design and implementation of CORAL's MySQL and PostgreSQL interface using design patterns and C/C++. It gives CORAL a solution to deal with the persistent data.
- Design and implementation of a CORAL client process based on the requirement of the Graph Database System.
- Study and comparison of some of CORAL user-level optimization strategies.
- Integration of the CORAL client with the TGL translator in the Graph Database System.
- Design and creation of MySQL and PostgreSQL databases according to the University Model database structure [2].
- Redesign and implementation of CORAL's rdb_join database interface.
- Update of CORAL's documentation with respect to the RDB-commands.

1.3 Organization of the Thesis

This thesis consists of seven chapters and two appendices. Chapter 1 introduces the motivation and contribution of the thesis. Chapter 2 describes the related disciplines for the thesis. CORAL, MySQL, PostgreSQL, and design patterns are introduced in this chapter. Chapter 3 presents the requirement of the CORAL extensional database interface. Chapter 4 illustrates the solution according to the MySQL and PostgreSQL client interface. Chapter 5 discusses the test result on these connections. Chapter 6 demonstrates CORAL's application in the Graph Database System. Chapter 7 concludes the thesis. Appendix A provides the University Data Model Schema in MySQL format. Appendix B demonstrates the University Data Model Schema in PostgreSQL format.

Chapter 2 Background

2.1 CORAL

CORAL is a powerful deductive engine. It supports recursive queries, queries with aggregate functions, and queries with negation. It was developed at the University of Wisconsin-Madison. This work began in 1988 and the latest version is 1.5.2, which was released on November 26, 1997. Information about CORAL can be accessed at http://www.cs.wisc.edu/coral/. The full version of source code, from version0.1 to version 1.5.2, is stored at ftp://ftp.cs.wisc.edu/coral/.

There are two highlights of CORAL's implementation: extensibility and flexibility. First, CORAL was developed under C++ class structure. Patterns were applied in its design and implementation. It allows programmers to enhance and reengineer the C++ class structure so as to enrich its functions, which include new data types, operations, relations, and indexes. Second, CORAL supports quite a few evaluation strategies. Users can influence the optimization techniques so as to exploit the full power of CORAL.

2.1.1 System architecture

As Figure 2.1 shows, CORAL consists of three subsystems: user interface, query processor, and data manager.

The user interface is responsible for accepting a user's request, passing it to the query processor, and receiving the query result.

The query processor is composed of two main parts: a query optimizer and a query evaluation system. Such simple queries as selecting facts from a base relation will be sent to the query evaluation system directly. Only complex queries, which are usually declared as program modules, will be transformed to an internal representation by the query optimizer. Several control annotations will be added to it during this procedure. Then, the optimized program will be transferred to the query evaluation system. In the next step, the query evaluation system takes this annotated program and database relations as input. It interprets and executes the program under the direction of annotations. The query result will be retrieved from the relations and sent to the user interface.

The query evaluation system has a well defined 'get-next-tuple' interface with the data manager to access the relations [3]. This high level interface works in the same way no matter what kinds of relations it deals with.

The data manager is in charge of maintaining and manipulating the data in relations. One advantage of CORAL's data manager is that it allows facts to contain variables, which is different from most other deductive database system. CORAL supports in-memory hash-relations as well as persistent relations. Multiple indices can be created and added to the existing relations. The persistent relations can be stored in text files, using the EXODUS storage manager [4], or in extensional databases. The data manager performs these functions in different ways. Relations in the text files can be "consulted" to the main memory and converted to in-memory relations. EXODUS is a client-server model storage manager that is independent of CORAL. Each CORAL process is a client that can access persistent data from the EXODUS server. Relations stored by EXODUS storage manager

can be paged into the client side buffers as required. The extensional databases act in the similar way as EXODUS but provide flexibility for CORAL users to select databases based on their relations characteristics. It is a new function of the data manager and supports Sybase4.6 and Ingres6.2 in CORAL's latest version. By talking with the client interface of a RDBMS, the data manager loads and stores relations on demand.

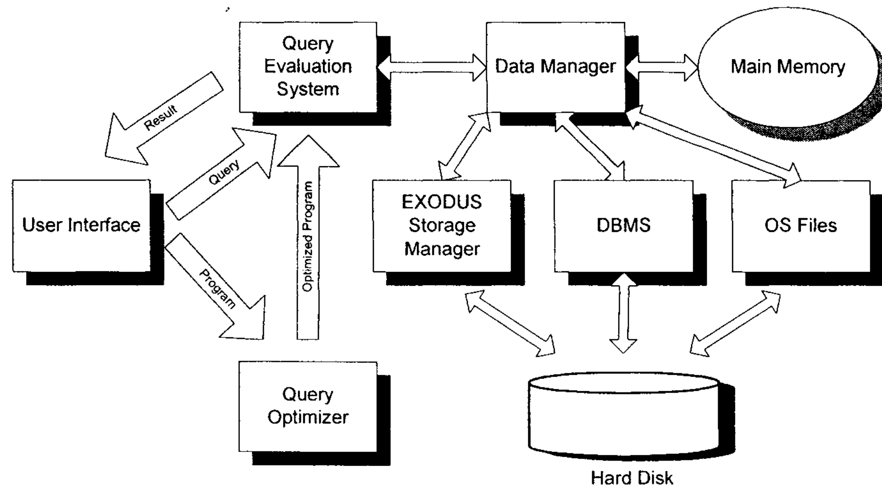

Figure 2-1 CORAL System Architecture [1]

2.1.2 Runtime data management

CORAL creates a logical memory structure to manage its runtime data. The key concepts involved are the workspace (or database), relation, and tuple. A workspace is the highest level logical structure that contains one or more relations. A relation is the container of tuples. A tuple is the smallest unit to compose a relation. The name of a workspace is unique in CORAL and the name of a relation is unique in a workspace. In addition, a relation belongs to a unique workspace only. However, tuples may be duplicated in a relation. They can be added to a specific relation from the persistent storage or just from CORAL's command line.

When CORAL starts, two workspaces, builtin_ws and default_ws, are created during the CORAL initialization period. The builtin_ws is used to store all built-in predicates and default execution parameters. The default_ws acts as the workspace to process user requests. CORAL names it as the current workspace. New workspaces can be created, deleted, and switched as the current workspace by CORAL commands. Two rules in CORAL must be aware that the builtin_ws can not be the current workspace and there is only one current workspace in CORAL. When CORAL terminates, all existing workspaces will be destroyed.

2.1.3 Extensional database interface

CORAL has designed an extensional database access interface, which is under coralroot/src/class/rdb. A command set is provided to communicate with relational databases and manage the extensional database interface. These commands can connect CORAL with a relational database, map relational tables as CORAL relations, and process data interactively between CORAL and the relational database system. All of them produce side-effects and can be used either in a rule or in a CORAL command line. The specification of these commands follows:

Command	Description
rdb_open_db	Create a CORAL database link based on a backend relational database.
rdb_map	Give a CORAL relation name to a backend table, which works as a base relation once mapped.
rdb_join	Give a CORAL relation name to a join of backend relations
rdb_execute	Issue an arbitrary database command. Most database commands which cannot be directly expressed in CORAL can be processed by the pass-through mechanism.
rdb_commit	Commit all changes to the database.
rdb_rollback	Undo all changes to the database.

Table 2-1 CORAL RDB Command Set

The two major requirements for using these features are that CORAL is compiled with RDB support and the database system is running and accessible by CORAL. CORAL's traditional commands, such as rel_copy, delete, and insert, are supported by this interface. Integer, real number, and string are the only supported types for this interface. Nulls are not well supported because they must be converted to the string "null" when loading from relational database table. Unless specified in rules, the relational database will be connected and managed under the current workspace of CORAL.

2.1.4 Client-server model

CORAL can be used either in standalone mode or in server mode. To start the CORAL server, run coral with the '-s' option. Client processes use sockets to connect with CORAL server, which is under TCP/IP protocol. The default location is the local machine and the default port is 6006. Changing the environment variables CORALSERVER and CORALPORT can change these default settings as required.

2.1.5 Optimization strategies

In CORAL, annotations are added at the level of each module to guide query optimization and control query evaluation. CORAL has a default execution environment setting for the module that does not specify an annotation. Such CORAL commands as display_defaults(), set(), clear(), and assign() are used to show and manipulate these default settings. During the compilation of a module, these default annotations will be added to the target file, the .M file. If annotations are declared in a module, they will override the default annotations while the module is consulted.

CORAL's annotations are divided into system-level annotations and user-level annotations. System-level annotations include @convert_functions, @single_scc, and @no_preprocessing. User-level annotations consist of Rewriting Annotations, Execution Annotations, and Per-Predicate Annotations. They are listed in Table 2-2 and Table 2-3.

Name	Function
System-Level:	
@convert_functions	convert arithmetic expressions to evaluable predicates
@single_scc	turns scc-by-scc seminaive evaluation on/off scc: Strongly Connected Component
@nested_scc_eval	Not fully implemented
@no_preprocessing	no compile time preprocessing

Table 2-2 CORAL System Level Annotations

Name	Function
Rewriting annotations	
@no_rewriting	perform no transformations
@sup_magic	perform supplementary magic rewriting (Default)
@magic	perform magic rewriting
@factoring	perform context rule rewriting
@naive_backtracking	use naive backtracking
Execution Annotations	
@pipelining	execute using pipelining
@ordered_search	execute using ordered search
@monotonic	modifies treatment of negation and grouping
@lazy_eval	computes answers in a lazy fashion(returns answers at end of each iteration)
@multiset	execute with multiset semantics
@check_subsumption	do subsumption checks on all predicates
@index_deltas	Create indexes on delta relations
@return_unify	perform return-unify optimizations for non-ground terms
@interactive_mode	return answers one at a time and prompt the user for more answers
Per-Predicate Annotations	
@make_index	index on specified adorned predicate
@allowed_adornments	adornment algorithm tries to use only these adornments
@check_subsumption	subsumption checking on specified adorned predicate
@multiset	specified adorned predicate treated as a multiset predicate
@aggregate_selection	modifies duplicate elimination by specifying tuples to retain.
@aggsel_per_iteration	the selection is only done once per iteration
@aggsel_multiple_answers	the selection allows multiple answers in each grouping
@prioritize	prioritize use of facts in a derived relation

Table 2-3 CORAL User Level Annotations

Although CORAL developed a number of query evaluation strategies, it still uses heuristic programming rather than a cost estimation package to choose evaluation methods [1]. System-level annotations are used only internally by CORAL and for debugging purposes. However, user-level annotations can be added directly to the source code and they give the programmer freedom to control query optimization as well as evaluation. They can be declared at the level of each module and override the default environment setting for the module.

2.2 MySQL

2.2.1 History

As the representative open-source RDBMS (Relational Database Management System) in the research and business fields, MySQL is specialized at fast speed, multi-user, multi-thread, and robust SQL (Structured Query Language) support. It is founded and developed by MySQL AB [5].

MySQL performs as a client-server system that consists of a multi-threaded SQL server and different types of backend clients. It supplies transactional and non-transactional storage engines. MySQL provides reference constraints by utilizing InnoDB type tables to guard data integrity from version 3.23.43b. However, MySQL does not support triggers, to be implemented in MySQL version 5.1. The recommended release version of MySQL is 4.0. Version 4.1 and 5.0 are also available at MySQL's website http://www.mysql.org.

2.2.2 C API

MySQL is written in C and C++. It provides APIs (Application Program Interface) for such languages as C, C++, JAVA, Perl, and PHP. The C API is developed by MySQL AB and is the basis for most of the other APIs. All the functions are included in the libmysqlclient library and fall into three distinct categories. Client programs that use libmysqlclient must include the header file mysql.h and must link with the libmysqlclient library. The main functions used in the implementation are shown in Table 2-4.

Name	Function
mysql_close	Closes a previously opened server connection.
mysql_error	Returns an error message for the most recent API failure
mysql_fetch_fields	Returns an array of all columns structure
mysql_fetch_lengths	Shows the length of all fields in current row of the result set.
mysql_fetch_row	Retrieve the next row from a result set.
mysql_free_result	Releases the memory occupied by a result set.
mysql_init	Gets or initializes a MYSQL object structure.
mysql_list_tables	Returns a table name set in the current database.
mysql_num_fields	Returns the number of columns in a result set.
mysql_num_rows	Returns the number of rows in a result set.
mysql_query	Executes a SQL query
mysql_real_connect	Try to connect to a MySQL database engine.
mysql_store_result	Returns a complete result set to the client process.

Table 2-4 MySQL C API Commands Used in Implementation

2.2.3 Supported data type

While retrieving data by the C API from MySQL, the data types in the MySQL field structure are retrieved at the same time. Because CORAL only supports three kinds of persistent data types currently, these data must be converted into corresponding CORAL data type as stipulated in Table 2-5. The null is a special CORAL string for an empty attribute.

Type Value	**Type Description**	**CORAL Type**
FIELD_TYPE_ENUM	ENUM field	Integer
FIELD_TYPE_INT24	MEDIUMINT field	Integer
FIELD_TYPE_LONG	INTEGER field	Integer
FIELD_TYPE_LONGLONG	BIGINT field	Integer
FIELD_TYPE_SHORT	SMALLINT field	Integer
FIELD_TYPE_TINY	TINYINT field	Integer
FIELD_TYPE_DECIMAL	DECIMAL or NUMERIC	Real
FIELD_TYPE_DOUBLE	DOUBLE or REAL field	Real
FIELD_TYPE_FLOAT	FLOAT field	Real
FIELD_TYPE_BLOB	BLOB or TEXT field	String
FIELD_TYPE_DATE	DATE field	String
FIELD_TYPE_DATETIME	DATETIME field	String
FIELD_TYPE_STRING	CHAR field	String
FIELD_TYPE_TIME	TIME field	String
FIELD_TYPE_TIMESTAMP	TIMESTAMP field	String
FIELD_TYPE_VAR_STRING	VARCHAR field	String
FIELD_TYPE_YEAR	YEAR field	String
FIELD_TYPE_NULL	NULL-type field	Null

Table 2-5 MySQL Data type to CORAL Data Type Conversion

2.3 PostgreSQL

2.3.1 History

PostgreSQL is an object-relational database management system (ORDBMS) developed by the computer science department at the University of California, Berkeley. Its ancestor is the POSTGRES project, which was led by Professor Michael Stoneraker in the same university between 1986 and 1993. Information about the design and data model of POSTGRES can be found at http://www.postgresql.org .

The latest version of PostgreSQL is Version 7.4. Besides supporting SQL92 and SQL99, PostgreSQL accepts complex queries, reference constraints, triggers, views, transactional integrity, and multi-version concurrency control. As an open-source extensible system, PostgreSQL gives users power to create their own data types, functions, operators, aggregate functions, index methods, and procedure languages. The released code can be

downloaded at many websites, which is listed at the PostgreSQL website. PostgreSQL runs in a client-server mode. The default connection port is 5432.

2.3.2 C API

Name	Function
PQclear	Frees the storage associated with a PGresult. Every result object should be freed via PQclear when it is no longer needed.
PQconnectdb	Makes a new connection to the database server. All parameters are passed in as a combined string. A PGconn memory structure is created to hold the connection information.
PQdb	Returns the database name of the current connection.
PQerrorMessage	Returns the latest error message generated by an operation on the connection.
PQexec	Submits a request to the server and waits for the result. The return object type is PGresult.
PQfinish	Closes the connection to the server. Also frees memory used by the PGconn object.
PQfname	Returns the column name associated with the given column number, which starts at 0.
Pqgetlength	Returns the actual length of a field value in bytes. Row and field numbers start at 0.
PQgetisnull	Tests a field for a null value. Row and field numbers start at 0.
PQgetvalue	Returns a single field value of one row of a PGresult. Row and field numbers start at 0.
PQnfields	Returns the number of fields in the query result.
PQntuples	Returns the number of rows in the query result.
PQresultStatus	Returns the result status of current request.
PQstatus	Returns the connection status (CONNECTION_OK or BAD)
PQsetdbLogin	The predecessor of PQconnectdb. Makes a new connection to the database server. Parameters are passed in as a fixed set.

Table 2-6 PostgreSQL C API Commands Used in the Implementation

Besides its standard command-line interface for data access, PostgreSQL provides a C API, a TCL interface, and a JDBC interface for accessing the database data in user applications. The C API, libpq, is composed of a set of functions that allow client programs to send queries to the PostgreSQL backend server and to receive the result of these queries. Similar to libmysqlclient library in MySQL, libpq is also the underlying engine for several other PostgreSQL application interfaces, such as libpq++, libpgtcl, and

Perl. Client programs that use libpq must include the header file libpq-fe.h and must link with the libpq library. The main functions used in the implementation are mentioned in Table 2-6.

2.3.3 Supported data type

PostgreSQL defines all its data types as objects and assigns a unique object ID (OID) to each of them. These OIDs are defined in the source file postgresql-7.4/src/include/catalog/pg_type.h. Main data types involved in our implementation are:

OID name	OID value	Data type	CORAL Type
BOOLOID	16	Bool field	Integer
INT2OID	21	Smallint field	Integer
INT4OID	23	Integer field	Integer
INT8OID	20	Bigint OR Bigserial field	Integer
FLOAT4OID	700	Deoubleprecision field	Real
FLOAT8OID	701	Real field	Real
NUMERICOID	1700	Numeric field	Real
BPCHAROID	1042	Character field	String
DATEOID	1082	Date field	String
TEXTOID	25	Text field	String
TIMEOID	1083	Time field	String
TIMESTAMPOID	1114	Timestamp field	String
VARCHAROID	1043	Varchar field	String

Table 2-7 PostgreSQL Data Type to CORAL Data type Conversion

2.4 Design Pattern

The design pattern describes collaborating objects and classes that are customized to solve a general design problem in a particular context [6]. It provides an easier way to reuse successful designs and architectures. Design patterns can be classified by two terms: purpose and scope. Design patterns can have creational, structural, or behavioral purposes. Creational patterns deal with the object creation process. Structural patterns are

concerned with the composition of classes or objects. Behavioral patterns characterize the interaction in classes or objects. On the other hand, design patterns can be divided into class patterns and object patterns based on their application on classes or objects. Class patterns are static and fixed at compile-time. They use inheritance to represent class relationships. Object patterns consist of the majority of design patterns. They are used in a more dynamic way because object relationships they deal with can only be decided at run-time.

In CORAL's design and implementation, many design patterns were utilized so as to make the program extensible. The main patterns I followed in the design are: abstract factory, façade, command, iterator, and template method.

2.4.1 Abstract factory

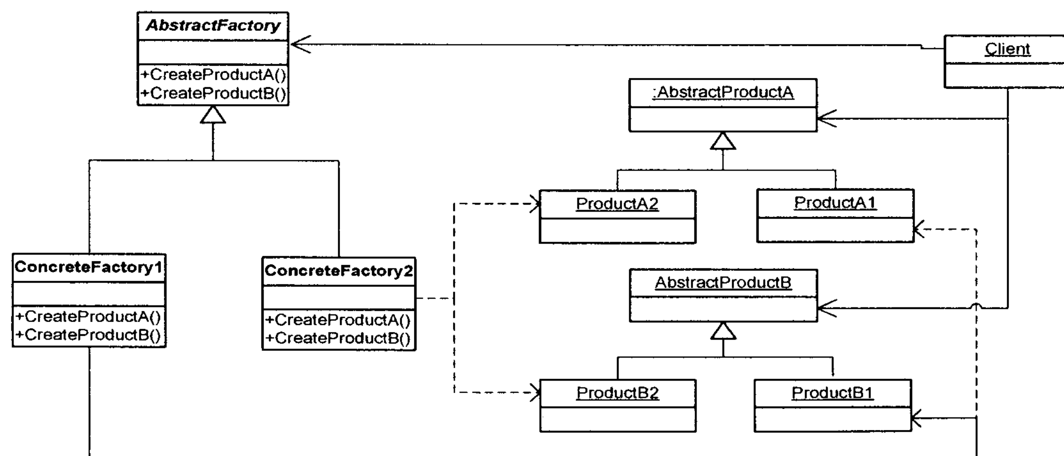

Figure 2-2 Abstract Factory Pattern [6]

Abstract factory is an object creational pattern to provide an interface for creating families of related objects without declaring their concrete classes. It isolates the concrete classes and makes exchanging product families easy. Moreover, consistency among products is promoted by this pattern because it is easy to make product objects in a family

work together. However, since an AbstractFactory interface fixes the set of products that can be initialized, it is difficult to support new kinds of products. Figure 2.2 shows the concept structure of this pattern.

2.4.2 Façade

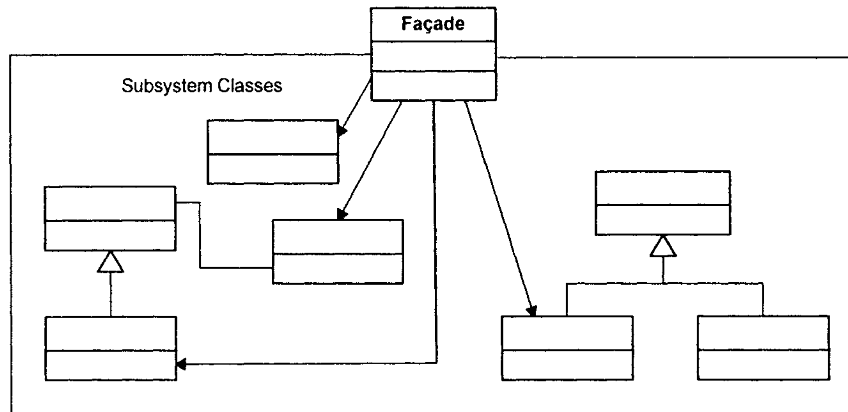

Figure 2-3 Façade Pattern [6]

Façade is an object structural pattern to provide a unified interface to a set of interfaces in a subsystem in order to make the subsystem easier to use. As Figure 2-3 shows, this pattern isolates the subsystem and its clients. It hides details of the subsystem, thereby reducing the number of objects that clients deal with and making the subsystem easier to operate. On the other hand, this pattern does not prevent applications from using subsystem classes directly. Thus, users can choose between ease of use and generality.

2.4.3 Command

Command is an object behavioral pattern to encapsulate a request as an object so as to parameterize clients with different requests, queue, or log requests. Its structure is shown in Figure 2.4. The Command pattern separates the object that invokes the operation from

the one that performs it. New commands are easily to be added because the existing class structure does not have to be changed.

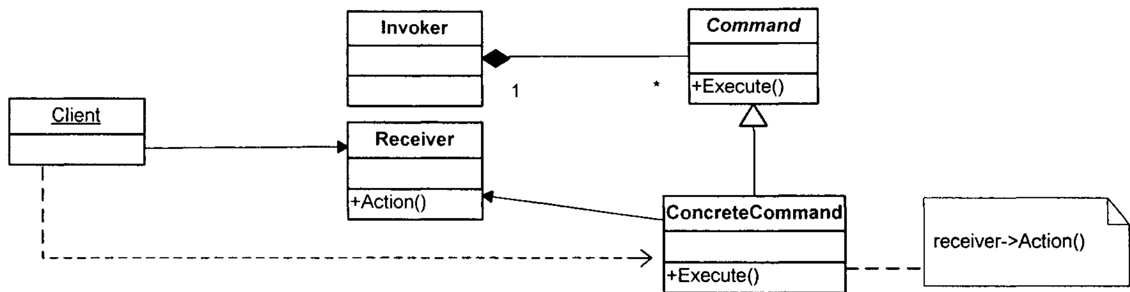

Figure 2-4 Command Pattern [6]

2.4.4 Iterator

Iterator, as illustrated in Figure 2-5, is an object behavioral pattern to provide a sequential access route to the elements in an aggregate object without exposing its underlying structure. It supports variations while traversing an aggregate and simplifies the interface for the aggregate. In addition, more than one traversal can be performed on an aggregate because an iterator keeps track of its own state.

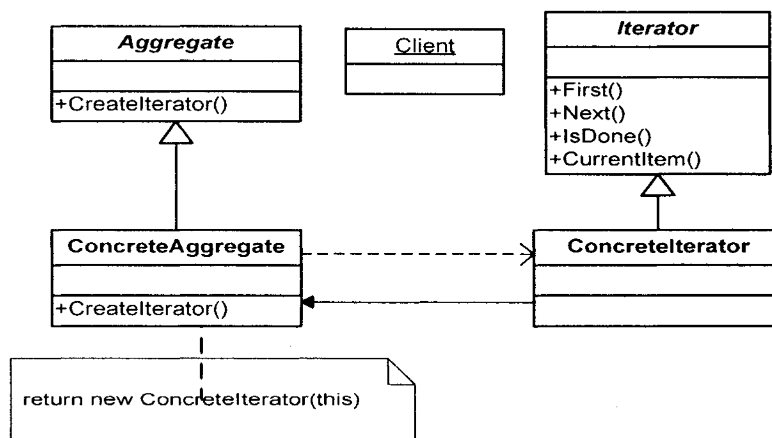

Figure 2-5 Iterator Pattern [6]

2.4.5 Template method

The Template method is a class behavioral pattern to declare the skeleton of an algorithm and defer some steps to the subclasses. It permits subclass define the algorithm without changing the algorithm's structure. Template method is a fundamental technique for reusing code. It calls concrete operations, concrete abstract class operations, abstract operations and factory methods.

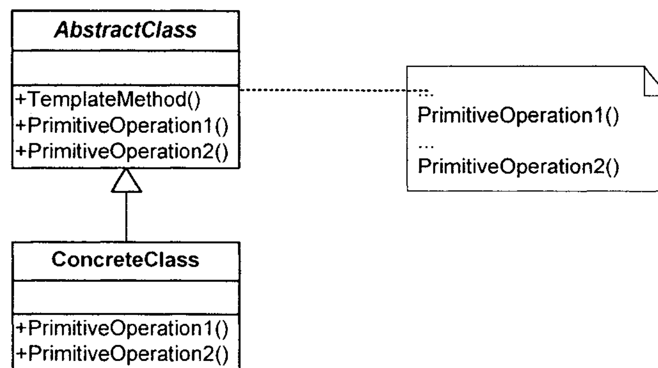

Figure 2-6 Template Method Pattern [6]

Chapter 3 Requirement

This chapter mainly specifies the functions and behaviors CORAL needs to communicate with MySQL and PostgreSQL. Use cases and flows of events are utilized to describe these requirements. Each problem is defined and described in detail. Non-functional requirements, such as hardware and implementation requirements, are also stipulated at the end of the chapter.

3.1 Requirement Definition

As mentioned in 2.1.3, CORAL provides a Rdb command set for relational database access and the beta version code was released as a compilation option. In order to abide by this command set and make it work with MySQL and PostgreSQL databases, the requirement is defined based on features of this command set and the two backend database systems.

There are three main requirements for CORAL when it connects to MySQL or PostgreSQL database management system. First of all, CORAL must provide an easy way to establish the connection. Namely, by using specific CORAL commands, users can access tables and data residing in MySQL or PostgreSQL. Second, from the user point of view, CORAL must process the data from the backend database in the same way as the in memory data. CORAL must use the same commands to delete, add, and query data no matter if it is in the memory or in the hard disk. Third, for the convenience of users,

CORAL should work as a client program for the connected database system. In other words, CORAL should act as the mysql in MySQL or psql in PostgreSQL. Any DML (Data Manipulate Language) and DDL (Data Definition Language) supported by the database system should be executed directly in CORAL interface.

These requirements are converted to the use case diagram shown in Figure 3.1. The actors represent roles interact with CORAL during this procedure. The use cases describe what CORAL system does when linking to backend databases.

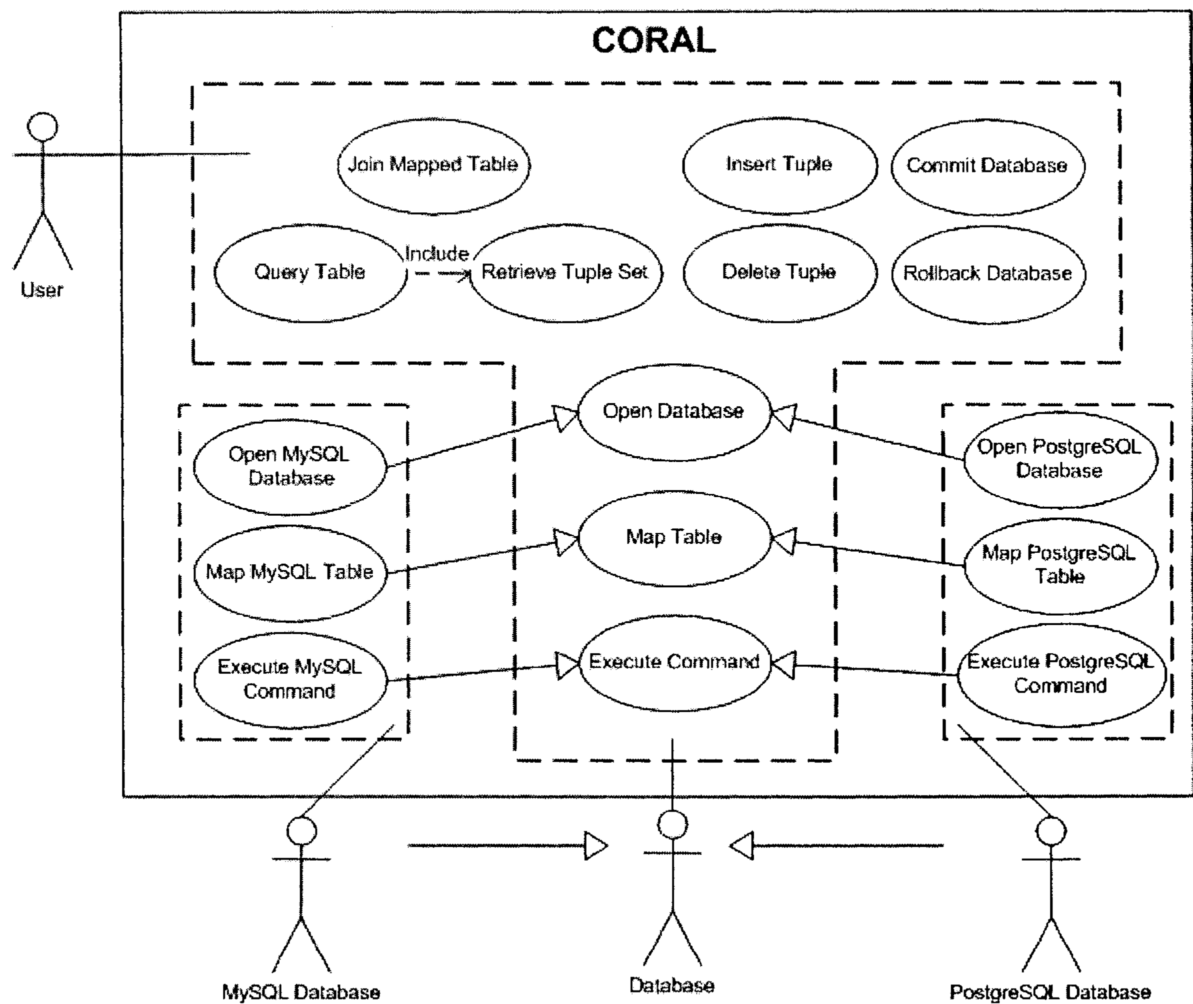

Figure 3-1 Use Case Diagram

3.2 Functional Requirement and Specification

Figure 3.1 is composed of four actors and sixteen use cases. Actor *User* represents a human being or a CORAL client process that talks with CORAL. Actor *Database* represents the database that CORAL can connect to. It has two children: one is MySQL *Database*; the other is *PostgreSQL Database*. Actor *MySQL Database* stands for the databases in MySQL database system requested by *User* from the CORAL side. Like actor *MySQL Database*, actor *PostgreSQL Database* consists of those PostgreSQL databases connected with CORAL.

The sixteen use cases fall into four functional groups. The first group is called connection establishment group, which is composed of seven use cases. In this group, *Open MySQL Database* and *Open PostgreSQL Database* are child use cases of *Open Database*; *Map MySQL Table* and *Map PostgreSQl Table* inherit behavior and meaning of *Map Table*; *Join Mapped Table* is a special use case because it only treats the CORAL relations created by *Map Table* and does not communicate with extensional databases. The second group is the relation process group, which consists of four use cases. Among these use cases, *Retrieve Tuple Set* is included by *Query Table* and never stands alone, so it is functionally connected with *Query Table*. From the *User* point of view, *Insert Tuple* and *Delete Tuple* communicate with *MySQL Database* tables and *PostgreSQl Database* tables in the same way as with in memory CORAL relations. Therefore, they do not have derived use cases. Thirdly, since *Execute Command* can be substituted by *Execute MySQL Command* and *Execute PostgreSQl Command* because of their generalization relationship, they compose another functional group, execute command group. This group mainly deals with the relational database commands not supported by CORAL.

Finally, the last two use cases, *Commit Database* and *Rollback Database*, form the fourth group, transactional support group. They are used to support database transactions. All these use cases are described in the following sections based on the group to which they belong.

3.2.1 Connection establishment

3.2.1.1 Open database

Use Case: *Open Database*

ID : UC01

Description

This use case is for identifying backend extensional databases to CORAL. By sending connection request to the backend databases, the *User* can connect CORAL with the database and store the database information as a CORAL database in CORAL environment.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*.

Flow of events

Main flow of events:

1. (start)The *User* enters the open database request through the keyboard.
2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format, new CORAL database name, database system name, and extensional database parameters.

4. CORAL sends the open database request to the *Database*.

5. The *Database* returns the execution result to CORAL. Connection is established.

6. The connection information is stored in CORAL as a new CORAL database under the name that the *User* provided.

7. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the open database request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the open database request to the *Database*.

4. The *Database* returns the execution result to CORAL.

5. Connection is established.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, system will display an error message to the *User*. The use case terminates.

Condition 2:

If the database system name is wrong, the use case terminates with an error message.

Condition 3:

If the new CORAL database name has been already used in CORAL, an error message will be shown to the *User*. The use case terminates.

Condition 4:

If the *User* provides wrong information about the *Database*, such as wrong database name, user name, password, or host location. The use case terminates after sending an error message to the *User*.

Condition 5:

If the *Database* is not started or can not be accessed by CORAL, the use case terminates.

Post-conditions

Added to the CORAL database list is a new CORAL database, which stores the connection information.

3.2.1.2 Open MySQL database

Use Case: *Open MySQL Database*

ID : UC011

Description

As a child use case of *Open Database*, this use case is for identifying backend MySQL databases to CORAL. By sending a connection request to the *MySQL Database*, the *User* can connect CORAL with the *MySQL Database* and store the database information as a CORAL database in CORAL environment.

Actors

User, MySQL Database

Pre-conditions

Same as that of *Open Database*.

Flow of events

Same as that of *Open Database*.

Post-conditions

Added to the CORAL database list is a new CORAL database, which includes the connected MySQL database information.

3.2.1.3 Open PostgreSQL database

Use Case: *Open PostgreSQL Database*

ID : UC012

Description

As a child use case of *Open Database*, this use case is for identifying backend PostgreSQL databases to CORAL. By sending a connection request to the *PostgreSQL Database*, the *User* can connect CORAL with the *PostgreSQL Database* and store the database information as a CORAL database in CORAL environment.

Actors

User, PostgreSQL Database

Pre-conditions

Same as that of *Open Database*.

Flow of events

Same as that of *Open Database*.

Post-conditions

Added to the CORAL database list is a new CORAL database, which includes the connected PostgreSQL database information.

3.2.1.4 Map table

Use Case: *Map Table*

ID : UC02

Description

This use case is for mapping relational tables in the extensional databases as CORAL relations. By means of the use case, CORAL creates special relations, which are based on relational tables and treated as regular CORAL relations. All facts of these relations are persistently stored in the backend databases as tuples and can be loaded into memory if required.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*. The connection with the *Database* must be made.

Flow of events

Main flow of events:

1. (start)The *User* enters the map table request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes the command format, CORAL database name, and the new CORAL relation name.

4. CORAL sends the map table request to the *Database*. Relative database information is retrieved from the established connection.

5. The *Database* returns the execution result to CORAL.

6. Mapping is made to the relational table. A CORAL relation is created and added to the CORAL environment.

7. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the map table request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the map table request to the *Database*.

4. The *Database* returns the execution result to CORAL.

5. Mapping is made to the relational table.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, system will display an error message to the *User*. The use case terminates.

Condition 2:

If the CORAL database name is wrong, which means it is not a database connected to the *Database*, an error message will be shown to the *User*. The use case terminates.

Condition 3:

If the *User* provided a CORAL relation name has been already used in the current workspace, the use case terminates with an error message.

Condition 4:

If the *User* provides wrong information about the *Database*, such as wrong table names, the use case terminates after sending an error message to the *User*.

Condition 5:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

CORAL relations are created based on the backend relational tables.

3.2.1.5 Map MySQL table

Use Case: *Map MySQL Table*

ID : UC021

Description

This use case is for mapping relational tables in the MySQL databases as CORAL relations. It is a child use case of *Map Table*. CORAL creates special relations, which are based on MySQL relational tables, and treated as regular CORAL relations. All facts in these relations are persistently stored in the MySQL databases and can be loaded into memory, if required.

Actors

User, MySQL Database

Pre-conditions

Same as that of *Map Table*.

Flow of events

Same as that of *Map Table*.

Post-conditions

CORAL relations are created based on the MySQL relational tables.

3.2.1.6 Map PostgreSQL table

Use Case: *Map PostgreSQL Table*

ID : UC022

Description

This use case is for mapping relational tables in the PostgreSQL databases as CORAL relations. It is a child use case of *Map Table*. CORAL creates special relations, which are based on PostgreSQL relational tables and treated as regular CORAL relations. All facts in these relations are persistently stored in the PostgreSQL databases and can be loaded into memory, if required.

Actors

User, PostgreSQL Database

Pre-conditions

CORAL is started and accessible to the *User*.

Flow of events

Main flow of events:

Same as that of *Map Table*.

Alternative flow of events:

Same as that of *Map Table*.

Exceptional Flow of Events:

Same as that of *Map Table*.

Post-conditions

CORAL relations are created based on the PostgreSQL relational tables.

3.2.1.7 Join mapped table

Use Case: *Join Map Table*

ID : UC03

Description

This use case is for mapping two or more backend relations into the current workspace as a new relation. Each backend relation is a CORAL relation that has been mapped to the relational tables by *Map Table*. The use case is solely an optimization because it allows the equi-join to be performed within the backend database rather than within CORAL.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*. The CORAL relation has been already created by the *Map Table*.

Flow of events

Main flow of events:

1. (start)The *User* enters the join table request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes the command format and the new CORAL relation name.

4. CORAL retrieves information about the backend relations.

5. The relation is created based on the existing backend relations and stored in the CORAL environment.

6. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the join table request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL retrieve information about the backend relations.

4. The relation is created based on the existing backend relations and stored in the CORAL environment.

5. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User*. The use case terminates.

Condition 2:

If the *User* provided a CORAL relation name that has been already used in the current workspace, the use case terminates with an error message.

Condition 3:

If the backend CORAL relation name is wrong, an error message will be shown to the *User*. The use case terminates.

Post-conditions

CORAL relations are created based on the backend CORAL tables.

3.2.2 Relation process

3.2.2.1 Query table

Use Case: *Query Table*

ID : UC04

Description

This use case is for retrieving results from the backend databases through the CORAL relations created during *Map Table* or *Join Mapped Table* use case. The *User* will use CORAL's standard query command to operate the use case, like querying an in-memory CORAL relation. The query result will be displayed to the *User* in the way stipulated by CORAL run-time parameters.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*. The CORAL relation has been already created.

Flow of events

Main flow of events:

1. (start)The *User* enters the query through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the query validity, which includes command format, CORAL relation name, and relation attributes.

4. Includes the *Retrieve Tuple Set* use case, which interacts with the *Database* and returns the result.

5. CORAL deduces the result based on the facts created by the *Retrieve Tuple Set* use case.

6. (end) CORAL returns the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the query through a running process.

2. CORAL receives the request and checks the query validity.

3. Includes the *Retrieve Tuple Set* use case.

4. CORAL deduces the result based on the facts created by the *Retrieve Tuple Set* use case.

5. (end) CORAL returns the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the query is entered in a wrong format, the system will display an error message. The use case terminates.

Condition 2:

If the *User* inputs the wrong relation name, an error message will be shown to the *User*. The use case terminates.

Condition 3:

If the *User* provided wrong attributes, such as wrong attribute type and wrong attribute sequence, the use case terminates with an error message.

Post-conditions

Query result is sent to the *User*.

3.2.2.2 Retrieve tuple set

Use Case: *Retrieve Tuple Set*

ID : UC041

Description

This use case is for retrieving facts from the backend database and storing them in memory. The use case is included in *Query Table* use case. It communicates with the backend database and creates a set of facts based on the request from *Query Table*. The facts will be used by *Query Table*.

Actors

Database

Pre-conditions

Query Table use case is started and the *User* inputs a command.

Flow of events

Main flow of events:

1. (start) The *Database* is called to retrieve the required facts.
2. The facts are created and stored in memory.
3. (end) The Database acknowledges the execution result.

Alternative flow of events:

None.

Exceptional Flow of Events:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

A fact set is created and stored in memory.

3.2.2.3 Insert tuple

Use Case: *Insert Tuple*

ID : UC05

Description

This use case is for storing new facts of mapped tables to the hard disk while CORAL is running. It acts as the insert command in SQL.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*. The CORAL relation has been already created.

Flow of events

Main flow of events:

1. (start)The *User* enters the insert tuple request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format, CORAL relation name, and relation attributes.

4. CORAL sends the insert tuple request to the *Database*.

5. New tuple is inserted into the relational table.

6. The *Database* returns the execution result to CORAL.

7. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the insert tuple request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the insert tuple request to the *Database*.

4. New tuple is inserted into the relational table.

5. The *Database* returns the execution result to CORAL.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User*. The use case terminates.

Condition 2:

If the CORAL relation name is wrong, which means that it is not a mapped table from the backend database, an error message will be shown to the *User*. The use case terminates.

Condition 3:

If the *User* provided wrong attributes, such as wrong attribute type and wrong attribute sequence, the use case terminates with an error message.

Condition 4:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

New tuples are inserted into the relational tables and stored in hard disk.

3.2.2.4 Delete tuple

Use Case: *Delete Tuple*

ID : UC06

Description

This use case is for deleting a specific fact or a set of facts from the mapped tables and storing the result to the hard disk while CORAL is running. It acts as the delete command in SQL.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*.

Flow of events

Main flow of events:

1. (start)The *User* enters the delete tuple request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format, CORAL relation name, and relation attributes.

4. CORAL sends the delete tuple request to the *Database.*

5. The tuples are deleted from the relational table.

6. The *Database* returns the execution result to CORAL.

7. (end) CORAL acknowledges the result to the *User.*

Alternative flow of events:

1. (start)The *User* enters the delete tuple request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the delete tuple request to the *Database.*

4. The tuples are deleted from the relational table.

5. The *Database* returns the execution result to CORAL.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User*. The use case terminates.

Condition 2:

If the CORAL relation name is wrong, which means that it is not a mapped table from the *Database*, an error message will be shown to the *User*. The use case terminates.

Condition 3:

If the *User* provided wrong attributes, such as wrong attribute type and wrong attribute sequence, the use case terminates with an error message.

Condition 4:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

Required tuples are deleted from the relational tables.

3.2.3 Execute command

3.2.3.1 Execute command

Use Case: *Execute Command*

ID : UC07

Description

This use case is for executing DDL and DML commands, such as create table and update table, on the backend extensional databases from the CORAL interface. CORAL acts as a client process of the underlying extensional database system. All requests will be written in the supported format and sent to the database system through the connection built by *Open Database.* The command is allowed to produce output, such as the command select. The result will be displayed to the *User* when execution finishes.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*. The *Database* has been connected already.

Flow of events

Main flow of events:

1. (start)The *User* enters the command request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format and CORAL database name.

4. CORAL sends the command to the *Database*.

5. The *Database* executes the command and returns the execution result to CORAL.

6. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the command request through a running process.

2. CORAL receives the request and checks the command validity, which includes command format and CORAL database name.

3. CORAL sends the command to the *Database.*

4. The *Database* executes the command and returns the execution result to CORAL.

5. (end) CORAL acknowledges the result to the *User.*

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User.* The use case terminates.

Condition 2:

If the CORAL database name is wrong, an error message will be shown to the *User.* The use case terminates.

Condition 3:

If the *Database* is not started or can not be accessed by CORAL, the use case terminates with an error message.

Post-conditions

The command has been executed in the *Database.* The result is displayed to the *User.*

3.2.3.2 Execute MySQL command

Use Case: *Execute MySQL Command*

ID : UC071

Description

This use case is for executing regular DDL and DML commands on the backend MySQL database from the CORAL interface. CORAL acts as a client process of the MySQL server. The use case is a child use case of *Execute Command.*

Actors

User, MySQL Database

Pre-conditions

CORAL is started and accessible to the *User*. The backend MySQL database connection has been established already.

Flow of events

Same as in Execute Command.

Post-conditions

The command has been executed in the MySQL database system. The result is displayed to the *User*.

3.2.3.3 Execute PostgreSQL command

Use Case: *Execute PostgreSQL Command*

ID : UC072

Description

This use case is for executing regular DDL and DML commands on the backend PostgreSQL database from the CORAL interface. CORAL acts as a client process of the PostgreSQL server. The use case is a child use case of *Execute Command.*

Actors

User, PostgreSQL Database

Pre-conditions

CORAL is started and accessible to the *User*. The backend PostgreSQL database connection has been established already.

Flow of events

Main flow of events:

Same as in *Execute Command.*

Post-conditions

The command has been executed in the PostgreSQL database system. The result is displayed to the *User*.

3.2.4 Transactional support

3.2.4.1 Commit database

Use Case: *Commit Database*

ID : UC08

Description

This use case is for committing all changes to the backend database system. The backend database system must be set in the transactional support mode.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*.

Flow of events

Main flow of events:

1. (start)The *User* enters the commit database request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format and the CORAL database name.

4. CORAL sends the request to the *Database*.

5. All uncommitted actions on the *Database* are committed and stored permanently.

6. The *Database* returns the result to CORAL.

7. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the commit database request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the request to the *Database*.

4. All uncommitted actions on the *Database* are committed and stored.

5. The *Database* returns the result to CORAL.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User*. The use case terminates.

Condition 2:

If the CORAL database name is wrong, an error message will be shown to the *User*. The use case terminates.

Condition 4:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

All uncommitted actions are committed and stored in the *Database*.

3.2.4.2 Rollback database

Use Case: *Rollback Database*

ID : UC08

Description

This use case is for rolling back all changes to the backend database system. The backend database system must be set in the transactional support mode.

Actors

User, Database

Pre-conditions

CORAL is started and accessible to the *User*.

Flow of events

Main flow of events:

1. (start)The *User* enters the rollback database request through the keyboard.

2. The *User* commits the entry by pressing the Enter key.

3. CORAL receives the request and checks the command validity, which includes command format and the CORAL database name.

4. CORAL sends the request to the *Database*.

5. All uncommitted actions on the *Database* are undone and the *Database* is rolled back to the previous committed state.

6. The *Database* returns the result to CORAL.

7. (end) CORAL acknowledges the result to the *User*.

Alternative flow of events:

1. (start)The *User* enters the rollback database request through a running process.

2. CORAL receives the request and checks the command validity.

3. CORAL sends the request to the *Database*.

4. All uncommitted actions on the *Database* are undone.

5. The *Database* returns the result to CORAL.

6. (end) CORAL acknowledges the result to the *User*.

Exceptional Flow of Events:

Condition 1:

If the request is entered in a wrong format, the system will display an error message to the *User*. The use case terminates.

Condition 2:

If the CORAL database name is wrong, an error message will be shown to the *User*. The use case terminates.

Condition 4:

If the *Database* can not be accessed by CORAL, the use case terminates.

Post-conditions

All uncommitted actions are rolled back permanently.

3.3 Non-functional Requirement and Specification

3.3.1 Computer hardware and software requirement

The system should run on the Unix server. The computer hardware, such as CPU, physical memory, and hard disk, must satisfy the requirements of the CORAL, MySQL, and PostgreSQL software packages.

The program will be developed in C++ in order to extend CORAL's original structure. GNU Make will be used to compile the source code. The MySQL database system and PostgreSQL database system must be installed and running properly.

3.3.2 System performance and reusability

The system will provide a convenient way for CORAL users to manage facts, i.e. retrieving and storing data. It will abide by all CORAL commands. For the concern of reusability, object-oriented design will be utilized in the system development. Such C++ techniques as inheritance, polymorphism, and design patterns will be applied during the design, as done in CORAL.

Chapter 4 Design

4.1 Architecture Design

This chapter presents an object-oriented design for linking CORAL to MySQL and PostgreSQL. The structural aspects of this link are discussed, including the overall structure and detailed design of each class. CORAL's rules of data management are studied and described as a prerequisite. Class diagrams, object diagrams, and interaction diagrams are drawn to show the class relationships and the runtime system snapshot at a given moment.

4.1.1 CORAL system structure

As an extension of CORAL's present system, the design for linking to MySQL and PostrgreSQL is based on CORAL's default structure and abides by all CORAL rules. Therefore, the CORAL system is studied and its methods to manage workspaces, relations, tuples, and extensional database connections are described in this section.

4.1.1.1 Classes for creating CORAL workspaces, relations, and tuples

In CORAL, a set of classes are used to create workspaces and relations. As Figure 4-1 shows, class Cor_DatabaseStruct is designed for creating CORAL's built-in workspace (builtin_ws), default workspace (default_ws), and any runtime workspaces created by the ws_create command. The Cor_CRdbDatabase class is assumed to construct the object for

an extensional database connection, although CORAL has not reached this step yet. It is a child class of Cor_DatabaseStruct. Both of them are concrete classes.

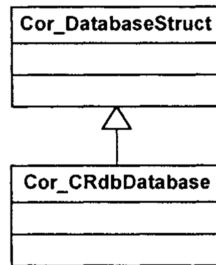

Figure 4-1 CORAL Workspace Classes

Figure 4-2 shows the high-level classes for creating and managing relations. The Cor_Relation class is an abstract class that provides a common interface for its children classes. Class Cor_BuiltinRelation is used to create predicate relations, which are used to create CORAL's predicates. Class Cor_DerivedRelation and Cor_StorageRelation are designed for regular in-memory relations. Cor_BuiltinRelation and Cor_Derivedrelation are concrete classes, whereas Cor_StorageRelation is an abstract class. All of them have derived classes, which are ignored in this diagram. The fourth child class, Cor_CRdbRelation, is for extensional database connections. It is a concrete class and a main focus in the thesis.

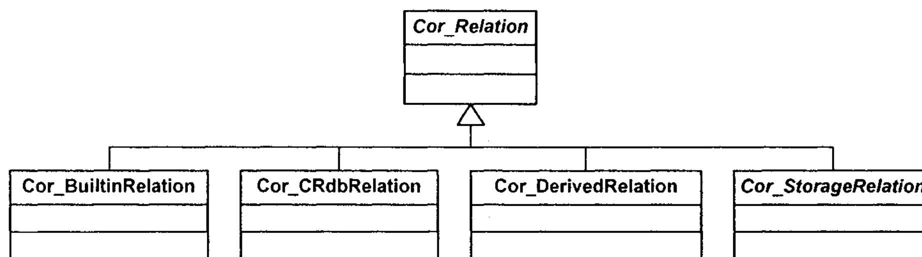

Figure 4-2 CORAL Relation Classes

CORAL uses the Cor_Tuple class to create in memory tuple objects. The Cor_Tuple class has an attribute Cor_ArgList and some public flags. Cor_ArgList is used to hold

attributes of tuples. Each attribute is represented by a Cor_Arg object, which is from the class that CORAL uses to create all its arguments in the runtime. The relationship of these classes is shown in Figure 4-3.

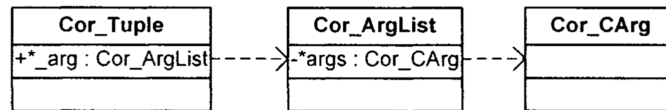

Figure 4-3 CORAL Tuple Classes

4.1.1.2 Classes for managing CORAL workspaces and relations

Two main classes, Cor_SymTable and Cor_SymTabElement, are used to manage the runtime CORAL workspaces and relations. Class Cor_SymTable is a container for Cor_SymTabElement objects and uses a hash table to store these objects. Cor_SymTabElement is the class to store information of a workspace or a relation. A void pointer is defined in this class in order to point objects of any type.

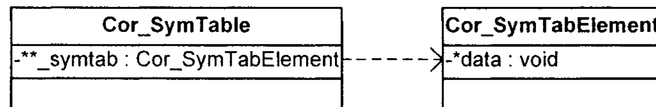

Figure 4-4 CORAL Workspace and Relation Management Classes

4.1.1.3 Runtime workspaces and relations structure

When CORAL starts, a tree style memory structure is created to hold workspaces and relations. At first, a Cor_Symtable type object named Cor_DatabaseTable is created. While the two Cor_DatabaseStruct type workspaces, builtin_ws and default_ws, are created in the initialization period, the default_ws is added to the Cor_DatabaseTable as a Cor_SymTabElement. Like default_ws, if a new Cor_DatabaseStruct type workspace is created in CORAL, it will be inserted into the Cor_DatabaseTable, too. The only exception is the builtin_ws that is solely managed by the CORAL system and never

added to the Cor_DatabaseTable. In each Cor_DatabaseStruct object, there is an attribute called RelationTable, which is a Cor_Symtable object for holding relations. Although CORAL defines the Cor_CRdbDatabse class intending to create workspaces for extensional database connections, it is not utilized yet. The solution for managing these connections will be described in 4.1.1.4.

When relations are created in CORAL, they will be stored in the workspaces they belong to. Because the Cor_BuiltinRelation objects are for CORAL predicates, all these objects is stored in the Cor_Symtable of the builtin_ws workspace. All other types of relations, such as extensional database relations and in memory relations, are stored in the Cor_Symtable of the current workspace.

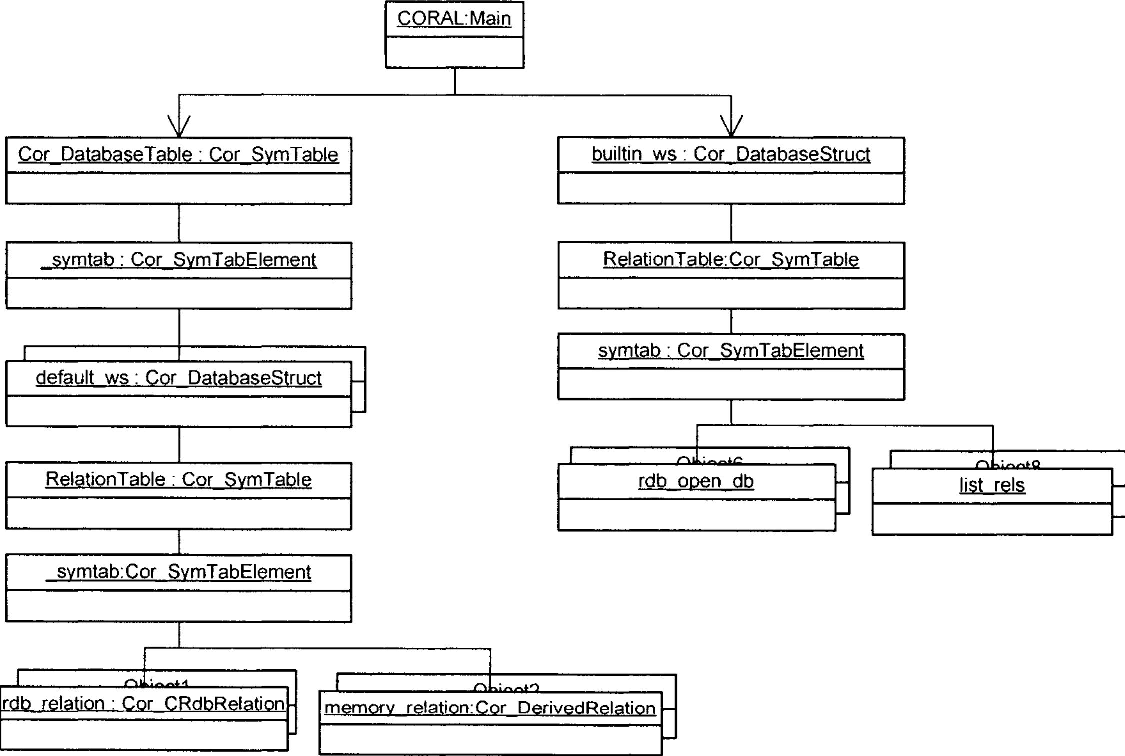

Figure 4-5 CORAL System Runtime Snapshot

Figure 4-5 illustrates a runtime snapshot of the CORAL system. At the moment this object diagram is drawn, the CORAL system has been initialized, i.e. default_ws,

builtin_ws, and all predicates have been established. At least one or more regular workspaces have been created and at least one extensional relation, rdb_relation, and one in-memory relation, memory_relation, have been constructed.

4.1.1.4 Relational database interface

From version 1.2, CORAL tried to access data stored in relational databases. A class hierarchy has been used to interact with relational databases and CORAL's in-memory structure. CORAL called these classes RDB classes and named them starting with Rdb. They consist of thirteen classes. The relations among these classes are illustrated in Figures 4-6, 4-7, 4-8, and 4-9. Although Cor_CRdbManger, Cor_CRdbRelation, DList, and Cor_DBEntry are not RDB classes, they are drawn in the figures because of their important roles related to the RDB classes. Their functions will be discussed with RDB classes, too.

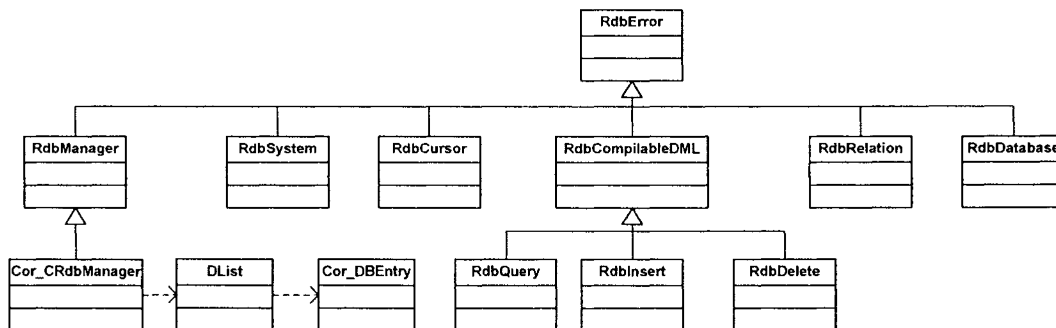

Figure 4-6 CORAL Main RDB Classes Inheritance Hierarchy

Figure 4-6 describes the generalization relationship of the main RDB classes. Class RdbError is located at the highest level. This class is used to manage all possible errors generated by RDB classes.

The RdbManager class and Cor_CRdbManager class perform a key role to control data interaction between CORAL and extensional databases. Both of them are concrete

classes. RdbManager is used to manage RdbSystems and Cor_CRdbManager is used to store connections to RdbDatabases. Cor_CRdbManager inherits all features of RdbManager. It has an object of DList, which is a template class that can contain Cor_DBEntry objects. The Cor_DBEntry object records the extensional database connection names and the information about the connections. It acts as the Cor_CRdbDatabase and solely controlled by Cor_CRdbManger.

Class RdbSystem represents particular DBMSs. It is an abstract class. Each specific DBMS should be managed by a concrete class derived from it. All database connections to this DBMS must be solely created and controlled by this class.

Class RdbDatabase is the most important class among all RDB classes because it works as the functional center to treat all requests related to the extensional databases. It is an abstract class and provides a virtual interface for child concrete database classes that deal with the extensional databases. RdbDatabase does not belong to the CORAL database hierarchy and it is managed by RdbSystem.

Class RdbRelation stands for all connections to extensional tables. Each RdbRelation object stores such information as the backend relation name, a collection of column names, and the database in which it resides.

Class RdbCompilableDML is a concrete class for any database statements that can be "prepared" and stored temporarily. These statements include insert, delete, and all queries. Class RdbInsert, RdbDelete, and RdbQuery are derived from it. These three classes are for inserting a tuple, deleting a tuple, and querying on specific tables in the backend databases. They are all abstract classes and have concrete child classes for any DBMS that CORAL supports.

Class RdbCursor is an iterator class that will be used during queries on the backend tables. It stores some state of the query. Figure 4-7 describes its relation to the RdbQuery.

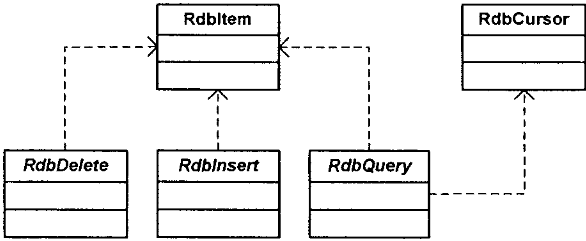

Figure 4-7 CORAL RDB Classes Dependency Relationship

Figure 4-7 introduces another RDB class, RdbItem, which is a collection of types of attribute that the RDB interface supports. Three data types are currently supported by CORAL. They are integer, real number, and string. RdbDelete, RdbInsert, and RdbQuery are dependent on RdbItem because all objects of these three classes deal with RdbItem type objects. This class is designed according to the Cor_Arg class, which is much more complex and belongs to the CORAL kernel.

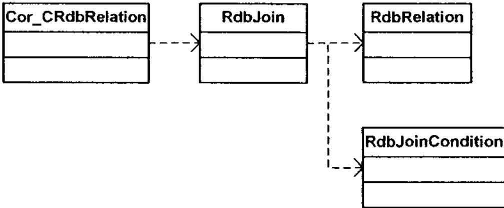

Figure 4-8 CORAL RDB Classes for Join Actions

Two more RDB classes, RdbJoin and RdbJoinCondition, must be mentioned because they are extremely important while mapping extensional tables to CORAL relations. As Figure 4-8 shows, class RdbJoinCondition records the equal join condition between any two different or same relations. Class RdbJoin is based on RdbRelation and RdbJoinCondition. These objects store the information of relations that are involved in

the join and their join conditions. The Cor_CRdbRelation objects are created based on the RdbJoin. One Cor_CRdbRelation may be dependent on one or more RdbRelations.

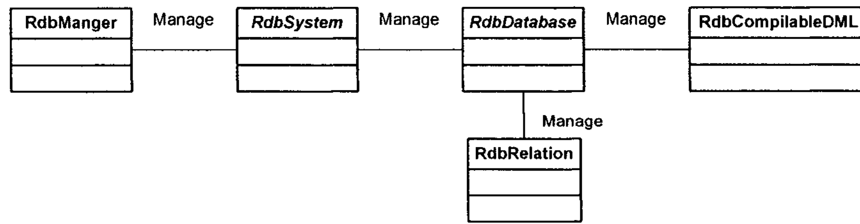

Figure 4-9 CORAL Relational Database Interface

In Figure 4-9, five RDB classes are shown. During CORAL runtime, only one RdbManager object will be created (as the parent of Cor_CRdbManger), which will manage one or more RdbSystem classes. Each RdbSystem object is for a specific DBMS that CORAL can link to. One or more RdbDatabase objects will be created based on the RdbSystem they reside in and stored in the RdbSystem. Each RdbDatabase object will contain one RdbCompilableDML, which is the start point of RdbQuery, RdbInsert and RdbDelete. The RdbRelation objects are based on the RdbDatabase they belong to and as components of the corresponding Cor_CRdbRelation objects.

4.1.2 Classes for MySQL and PostgreSQL

In order to connect CORAL to MySQL and PostgreSQL, classes are designed to extend CORAL's current class structure. The overall class diagram is in Figure 4-10. Twelve concrete classes are added to perform specific tasks for communicating with extensional databases. All these classes follow CORAL's rules mentioned above and the C API of MySQL and PostgreSQL. These classes are divided in two groups, one for MySQL and the other for PostgreSQL.

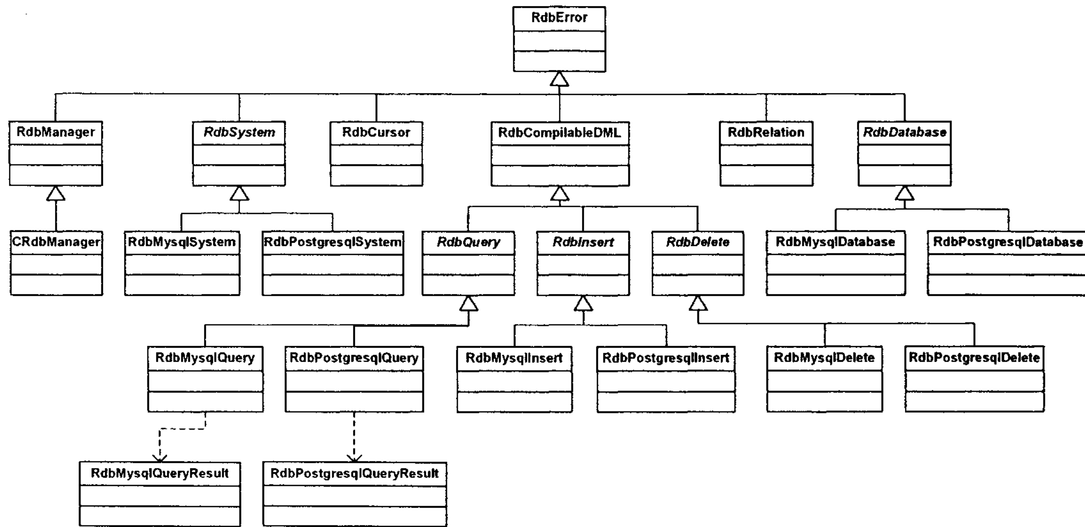

Figure 4-10 Extended CORAL RDB Class Diagram

4.1.2.1 Classes for MySQL

Classes for MySQL include RdbMysqlSystem, RdbMysqlDatabase, RdbMysqlInsert, RdbMysqlDelete, RdbMysqlQuery, and RdbMysqlQueryResult. Class RdbMysqlSystem is derived from RdbSystem. It is responsible for creating a concrete instance for MySQL. It is initialized from the RdbManager. All MysqlDatabase objects are created by this class. Class RdbMysqlDatabase is used to establish the connection with a specific MySQL database. Any DDL and DML operation on the backend MySQL database will be processed by this class. Class RdbMysqlInsert and class RdbMysqlDelete perform the tuple insert and delete tasks for the backend database. They are derived from their abstract parent classes and perform the opposite functions from the databases' point of view. Class RdbMysqlQuery serves as the iterator while processing a query on the backend relations. All facts it processes are from the RdbMysqlQueryResult collection. Class RdbMysqlQueryResult is a special class for holding the query result set returned from the backend database and the status of the result set. It always binds with a RdbCursor when initialized.

4.1.2.2 Classes for PostgreSQL

Similar to MySQL, there are six classes, RdbPostgresqlSystem, RdbPostgresqlDatabase, RdbPostgresqlInsert, RdbPostgresqlDelete, RdbPostgresqlQuery, and RdbPostgresqlQueryResult, designed for PostgreSQL. Class RdbPostgresqlSystem is derived from RdbSystem, too. It is used to create a concrete instance for PostgreSQL. PostgresqlDatabase objects are initialized and stored by this class. Class RdbPostgresqlDatabase is used to establish the connection with a specific PostgreSQL database. Any operation on the backend PostgreSQL database will be processed by this class. Class RdbPostgresqlInsert and class RdbPostgresqlDelete perform the tuple insert and delete tasks for the backend database. They are derived from their abstract parent classes. Class RdbPostgresqlQuery serves as the iterator while processing a query on the backend relations. It retrieves all the required facts from RdbPostgresqlQueryResult. Class RdbPostgresqlQueryResult is designed for holding the query result set returned from the backend database and the status of the result set. It always binds with a RdbCursor when created.

4.1.3 Behavioral modeling

This section describes the dynamic aspects of the system. It is according to the use cases mentioned in section 3.2. All solutions for these use cases are discussed by means of classes designed above. Since MySQL and PostgreSQL are both extensional databases and share similar behaviors when CORAL connects to them, only MySQL is used for the example.

4.1.3.1 Establish database connections

Use cases concerned are *Open Database* (UC01), *Open MySQL Dtabase*(UC011), and *Open PostgreSQL Database*(UC012). Figure 4-11 shows the sequence when connecting CORAL with a MySQL database.

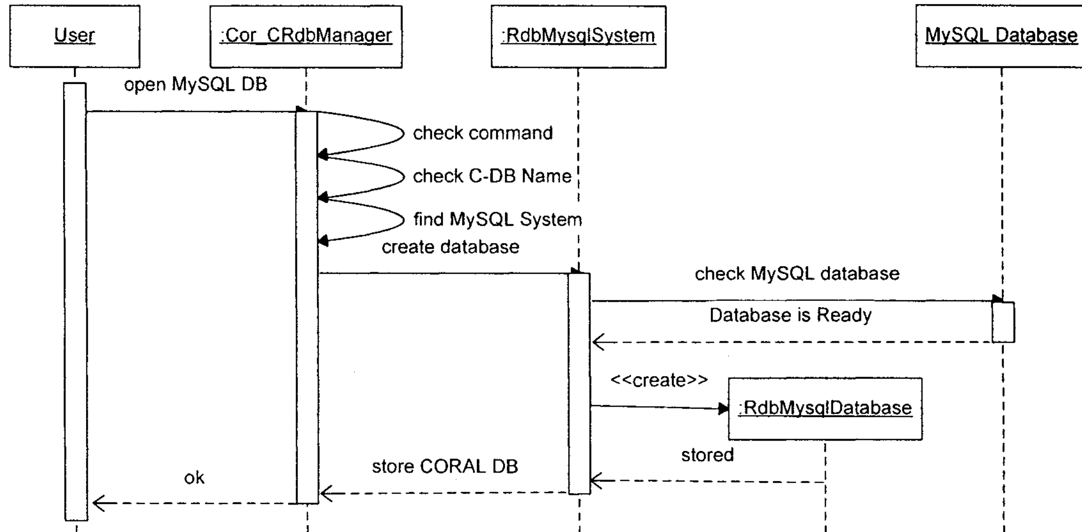

Figure 4-11 Open a MySQL Database

First of all, the user types in request to establish a connection with MySQL database. Cor_CRdbManager will catch the request and check its syntax. Then, CORAL will check whether the name for this CORAL RDB database has already been used. If not, RdbMysqlSystem will be called to create the database. After RdbMysqlSystem receives the command, it will contact with the MySQL Database to ensure that the backend database is exist and ready for connection. The next step is that one RdbMysqlDatabase object is created and stored in RdbMysqlSystem. As a result, this connection is represented by a CORAL RDB name and stored in Cor_CRdbManger. Result message will be displayed to the user before the sequence ends.

4.1.3.2 Establish table connections

As described in use case *Map Table*(UC02), *Map MySQL Table*(UC021), and *Map PostgreSQL Table*(UC022), backend relational tables can be mapped as CORAL relations. Figure 4-12 illustrates the process for mapping a MySQL table to a CORAL relation.

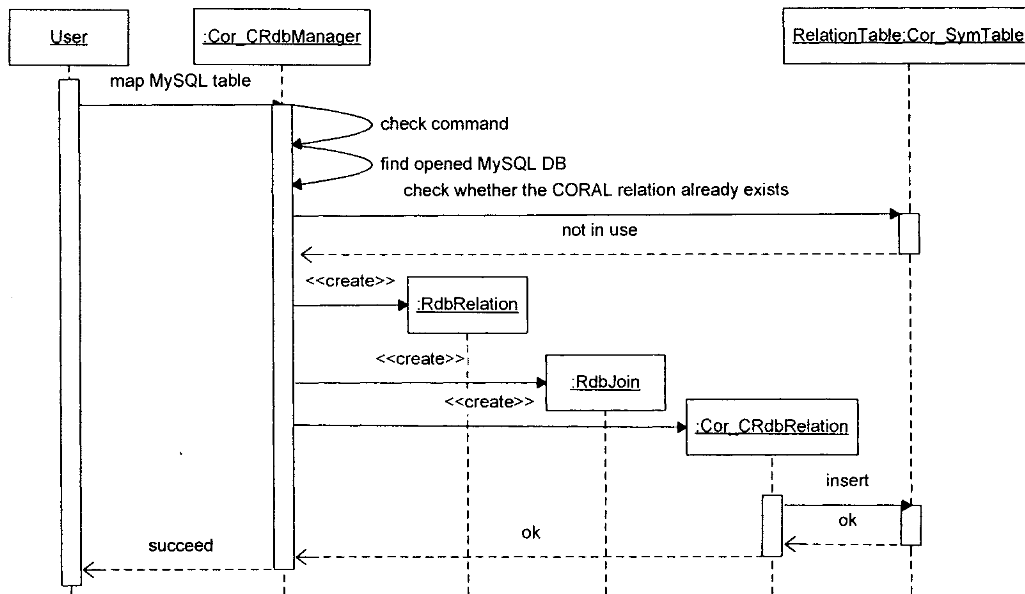

Figure 4-12 Map a MySQL Table

When the user issues the map table request, Cor_CRdbManger first catches the request and validates the syntax of the request. Then, the opened RDB database list is checked for the CORAL RDB database name mentioned in the command. If it exists, the new CORAL relation name will be checked by referring to the RelationTable. If the provided relation name is not used in current workspace, Cor_CRdbManager will create two objects of type RdbRelation and RdbJoin in preparation to create the desired CORAL relation of type Cor_CRdbRelation. The RdbRelation object stores information about the backend database and the table column names. The RdbJoin object stores the join

conditions between the relations, if there are any conditions. The Cor_CRdbRelation object is created using the above two objects. The next step is to store the created Cor_CRdbRelation object into the RelationTable. This step ensures that the CORAL relation can be utilized by the CORAL system directly. Finally, a message will be return to the user to confirm the success of the process.

4.1.3.3 Create relations based on RDB relations

When relational tables are mapped as CORAL relations, they can be joined to create CORAL relations, as described in use case *Join Mapped Table* (UC03). The main steps for this procedure are shown in Figure 4-13. It is merely an optimization behavior and has no interaction with the backend database.

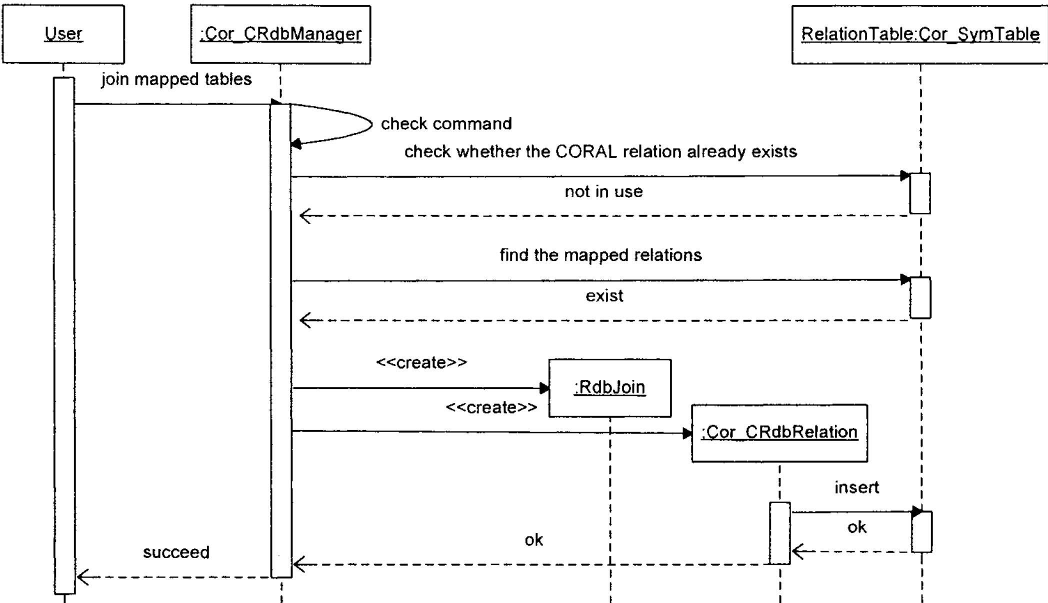

Figure 4-13 Join Mapped Tables

When the user inputs the join mapped tables command, the Cor_CRdbManager will check the command validity and refer to the RelationTable to find the relations mentioned in the command. If they exist, a RdbJoin object will be created using the

relations and their join conditions defined in the command. Then, the Cor_CRdbRelation object will be created and added to the RelationTable. At last, the end message will be displayed to the user.

4.1.3.4 Relation process

Once relational tables are mapped to the CORAL, they can be queried directly from the CORAL environment. CORAL's standard delete and insert commands can also delete or insert tuples for the backend tables. Use cases *Query Table*(UC04), *Retrieve Tuple Set*(UC041), *Insert Tuple*(UC05), and *Delete Tuple*(UC06) discuss these behaviors. Figure 4-14 describes the main activities when querying a RDB table. In this scenario, only one result is obtained.

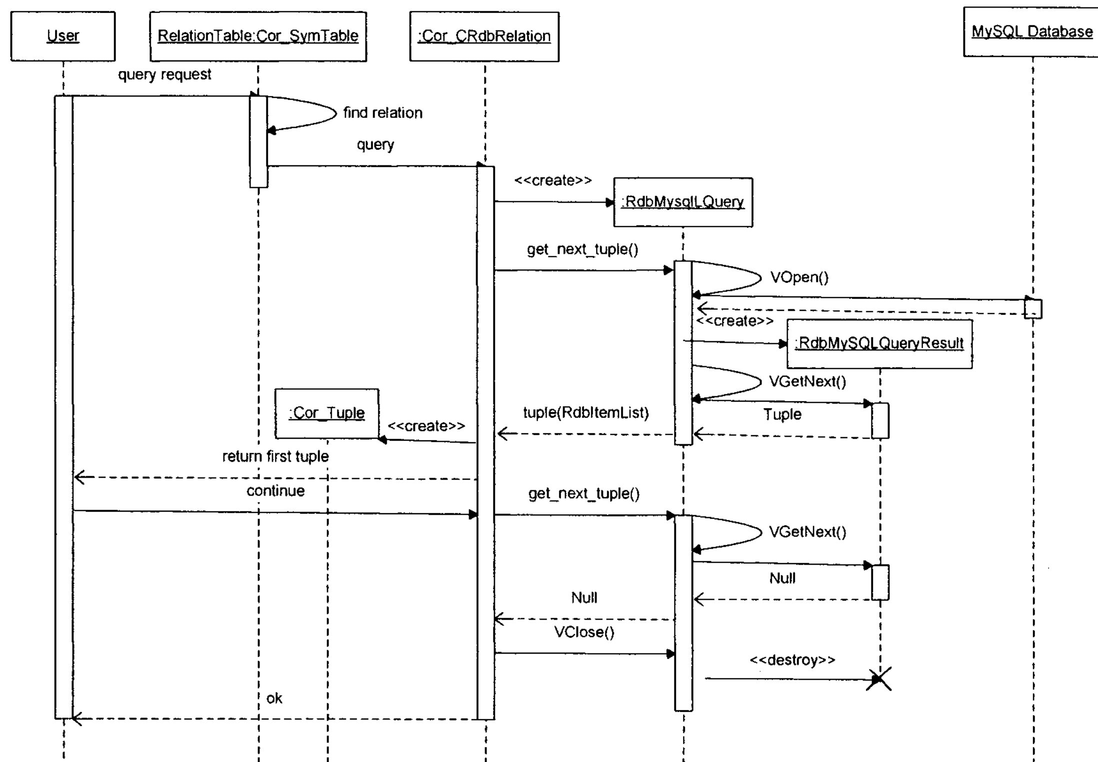

Figure 4-14 Query a Mapped Table

When a user inputs the query, CORAL will find the relation from RelationTable and use its "get-next-tuple" interface [3] to retrieve facts. During this period, a RdbMysqlQuery object will be created. It is an iterator to query the result set that is stored in object RdbMysqlQueryResult. The RdbMysqlQueryResult is created when VOpen() is called. It will be destroyed when the query finishes, namely, when VClose() is called. All tuples from the backend database will be converted to CORAL tuples (Cor_Tuple) by Cor_CRdbRelation. The result will be shown to the user.

The insert and delete tuple operations are similar to the query operation. When inserting a tuple to the backend table, the corresponding CORAL relation will be located first. Then, a RdbMysqlInsert object will be created to communicate with the backend MySQL Database and insert the tuple into the specified table. On the other hand, when the delete command is called, a RdbMysqlDelete object will be created to delete the tuple as required. Figure 4-15 illustrates the behavior for insert.

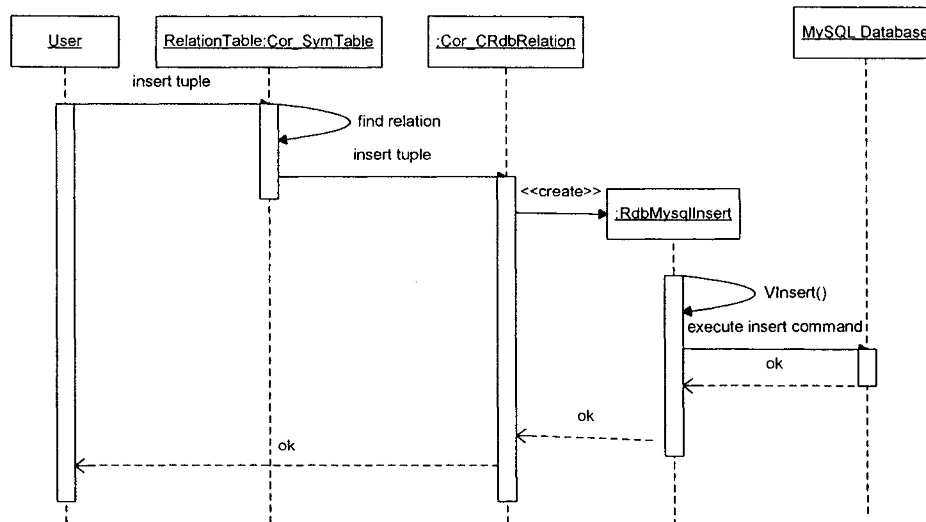

Figure 4-15 Insert a Tuple to the Backend Table

4.1.3.5 Execute command

CORAL supports all backend database commands by acting like a client process for the database. All the database commands can be execute by this method. Use cases *Execute Command*(UC07), *Execute MySQL Command*(UC071), and *Execute PostgreSQL Command*(UC072) were defined for this function. Figure 4-16 illustrates the solutions for the MySQL database system.

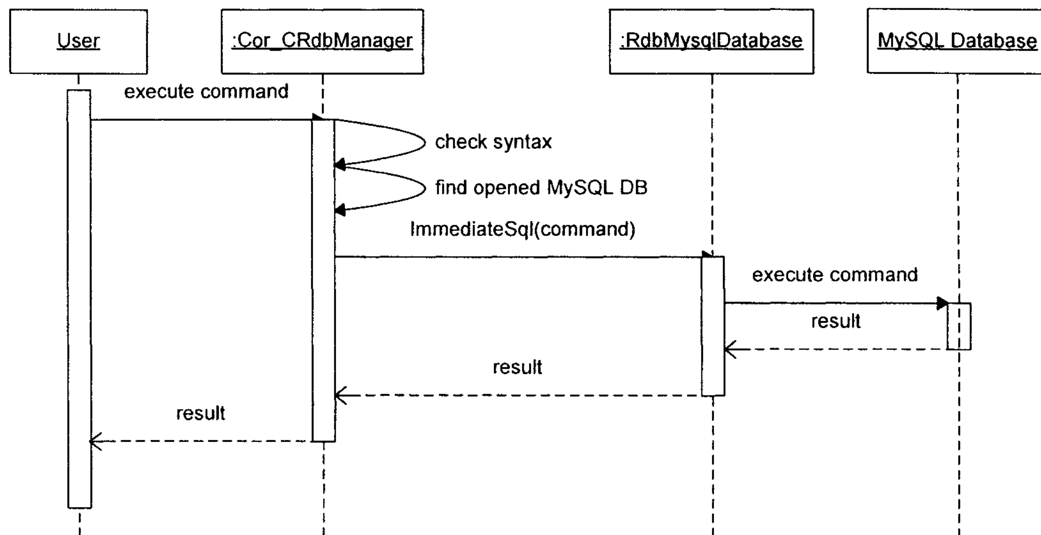

Figure 4-16 Execute Command at the Backend MySQL

As soon as a user enters the command that will be executed at the backend MySQL database system, Cor_CRdbManager will first check the syntax of the user input. Then, the database mentioned in the command will be found from the database list and set as the current database. The database object has a template method called ImmediateSql, which is used to execute the command at the backend MySQL. By calling ImmediateSql(), the command input by user is passed to the MySQL server and executed. The result is returned to the Cor_CRdbManager and finally shown to the user.

4.1.3.6 Transactional support

CORAL provides an abstract interface for transaction support in the backend database. In order to support these functions while linking to MySQL and PostgreSQL, concrete classes must be designed to provide specific methods for each database. In the thesis, the tasks described in the use cases *Commit Database* (UC08) and *Rollback Database* (UC09) are solved by the virtual functions, VCommit() and VRollback(), in RdbMysqlDatabase and RdbPostgresqlDatabase. Although these two functions perform totally different tasks on the backend database, CORAL executes them in the same algorithm. Figure 4-17 shows the sequence when committing a MySQL database.

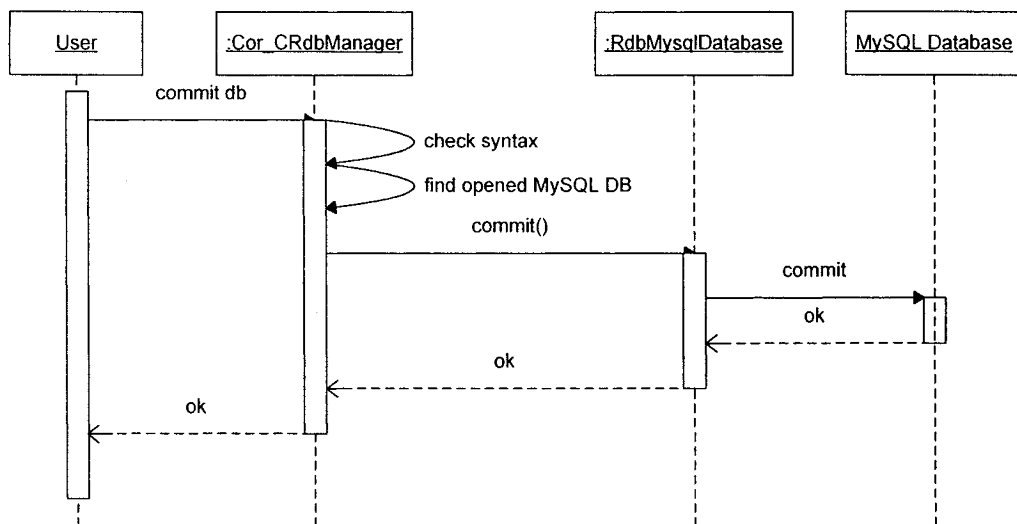

Figure 4-17 Commit the Backend MySQL Database

4.1.4 Design patterns used

Many patterns, such as composite and strategy, can be found in CORAL. In this thesis, only those design patterns that have been followed during the design are illustrated. Examples are shown in the following diagrams. As introduced in 2.4, five design patterns

have been used in the design. These patterns are chosen based on CORAL's high-level class structure and the requirement of the extensional databases.

4.1.4.1 Abstract factory pattern

Figure 4-18 shows the usage of abstract factory pattern in the RDB classes. From the diagram we can see that RdbDatabase performs as the abstract factory, which declares an interface for operations that create abstract product objects, RdbQuery, RdbInsert, and RdbDelete. RdbMysqlDatabase and RdbPostgresqlDatabase are concrete factories that implement the operation to create concrete product objects, RdbMysqlQuery, RdbPostgresqlQuery, RdbMysqlInsert, RdbPostgresqlInsert, and so on. RdbQuery and RdbInsert are the abstract products. RdbMysqlQuery and RdbPostgresqlQuery are the concrete products. RdbSystem acts as the Client because it only calls the interface declared by the abstract factory.

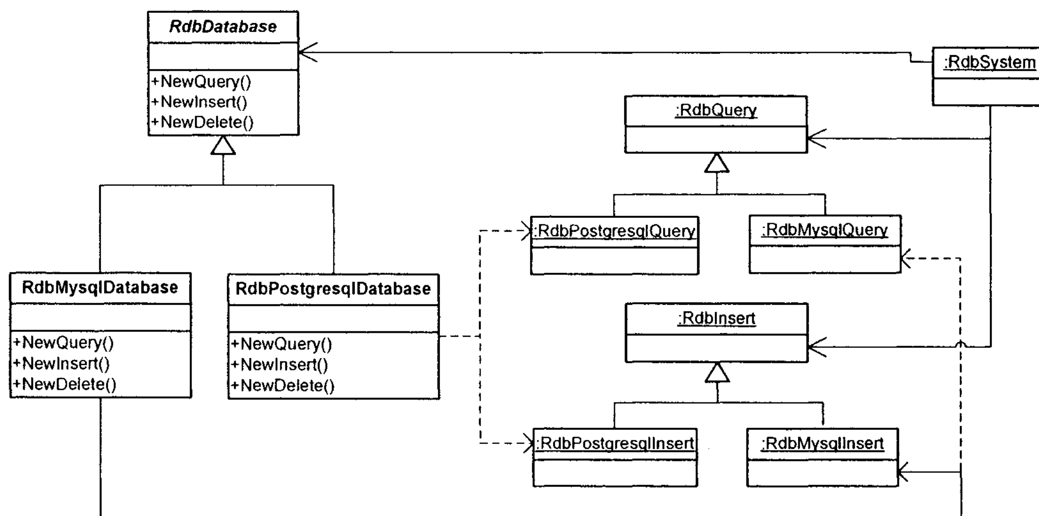

Figure 4-18 Abstract Factory Pattern in RDB Classes

4.1.4.2 Façade pattern

As Figure 4-19 shows, the façade pattern provides a unified interface to all the interfaces in the RDB classes. The Cor_CRdbManager, as well as its parent RdbManager, is the

façade that knows which subsystem classes are responsible for a request. It delegates a client request to the appropriate subsystem objects. RDB objects and RDB functions of each object can be accessed through it. On the other hand, all the RDB classes can be called directly if required.

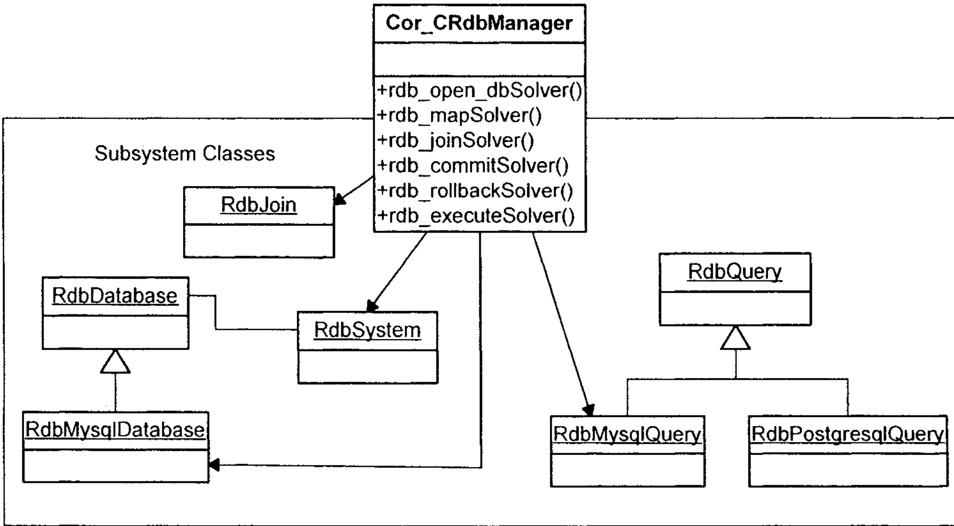

Figure 4-19 Façade Pattern in RDB Classes

4.1.4.3 Command pattern

The Command pattern is used among classes for querying data from extensional databases. Figure 4-20 describes the roles of these classes. RdbQuery acts as the command that declares an interface for executing the 'get-next-tuple' operation. The RdbMysqlQuery class defines this function and binds the GetQSResult() function of the RdbMysqlQueryResult, which is the receiver in this case. When CORAL performs the query on the MySQL database, it uses the 'get-next-tuple' interface defined in Cor-CRdbRelation and invokes the action on RdbMysqlQueryResult.

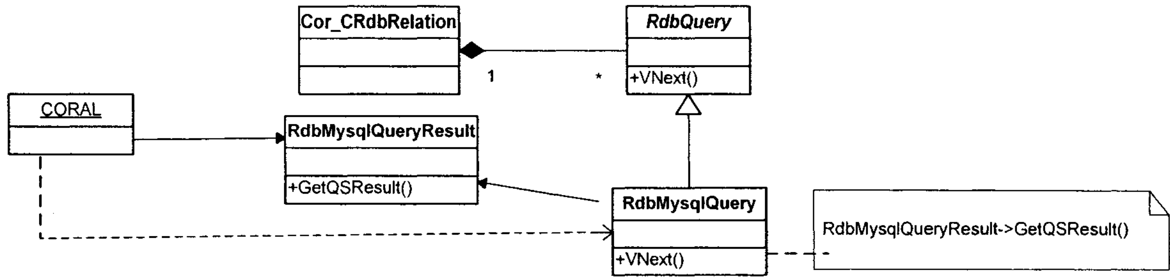

Figure 4-20 Command Pattern in RDB Classes

4.1.4.4 Iterator pattern

Many Iterators are used in CORAL, such as RdbCursor and Cor_TupleIterator. Figure 4-21 explains the iterator pattern performed by RdbQuery. As an iterator, RdbQuery is an abstract class that defines an interface for accessing and traversing tuples. This interface consists of Open(), GetNext(), and Close(). The concrete iterator, RdbMysqlQuery, implements this interface. RdbDatabase acts as the aggregate class that defines NewQuery() for creating the RdbQuery object. The real implementation is dynamically bound to into RdbMysqlDatabase so as to return an instance of the RdbMysqlQuery.

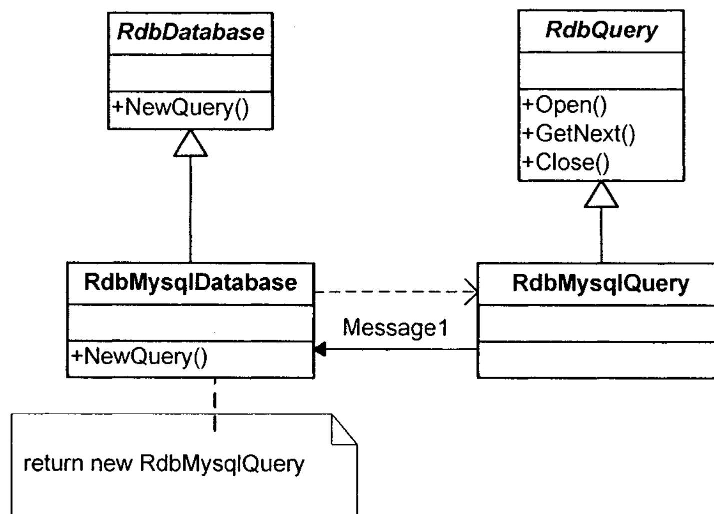

Figure 4-21 Iterator Pattern in RDB Classes

4.1.4.5 Template method pattern

Many template methods are defined in the RdbDatabase class. From Figure 4-22 we can see that RdbDatabase defines VDisplay() and VCommit() as pure virtual functions. The template methods, Display() and Commit() are declared to call these abstract functions individually. As the subclass, RdbMysqlDatabase implements the abstract functions defined in its parent class so as to display the required stuff fro MySQL or commit MySQL database when the template methods are called.

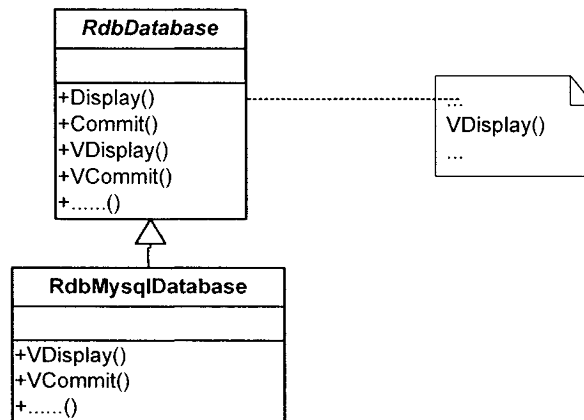

Figure 4-22 Template Method Pattern in RDB Classes

4.2 Major Classes Specifications

In 4.1, the architecture of the main classes for linking CORAL to extensional databases was illustrated. The interactions of these classes were discussed according to the use cases defined in chapter 3. In this section, the concrete classes that were added to CORAL's class hierarchy are described in detail. Their abstract parent classes are also drawn and partially explained in order to make the concrete classes clear. Other classes, such as Cor_DBEntry and Cor_CRdbRelation, are part of CORAL and can be traced from CORAL's source code. They are ignored in this section. As classes for MySQL and

PostgreSQL are similar and in the same structure, explanations for PostgreSQL are concise in order to avoid too much repetition.

4.2.1 RDB system classes

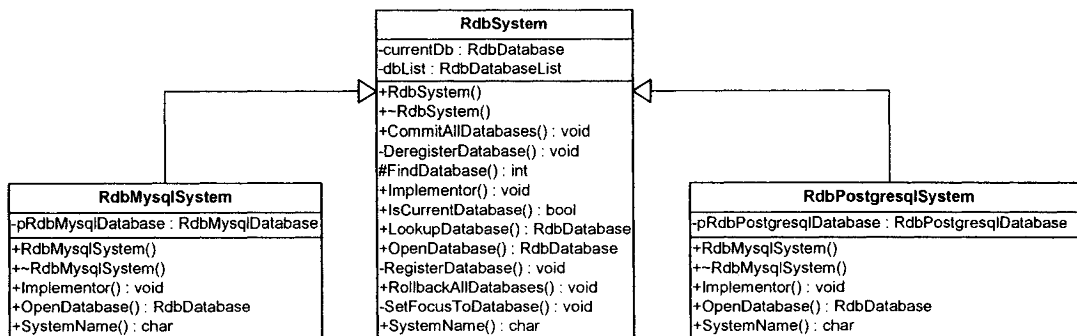

Figure 4-23 RDB System Classes

4.2.1.1 The RdbSystem class

This class is an abstract class that defines the common interface for RdbMysqlSystem and RdbPostgresqlSystem. It can not be initialized independently and must be created inside a RdbMysqlSystem or RdbPostgresqlSystem object. Two attributes and five main operations of it are described because of their importance for the child classes.

Attributes:

currentDb: RdbDatabase type pointer that points to the current active RdbMysqlDatabase or RdbPostgresqlDatabase object.

dbList: RdbDatabase type list that holds all RdbMysqlDatabase or RdbPostgresqlDatabase objects.

Operations:

LookupDatabase(): searches for the required database object by its name and server name. If found, returns it to caller; if not, returns null.

RegisterDatabase() and DeregisterDatabase(): points or de-points the currentDb to the target database object.

CommitAllDatabase() and RollbackAllDatabase(): commits or rolls back all database in the dbList at one time. They are invoked by rdb_commit() or rdb_rollback()

4.2.1.2 The RdbMysqlSystem class

This class is used to create an instance representing the MySQL database system. Only one such object is created in the CORAL runtime and all RdbMysqlDatabase objects will be managed by this instance. It consists of one attribute and five member functions.

Attribute:

pRdbMysqlDatabase: RdbMysqlDatabase type pointer used to hold the created database object.

Operations:

OpenDatabase(): creates a RdbMysqlDatabase object based on the parameters provided by the caller.

SystemName(): returns string 'mysql'.

Implementor(): returns this. This function is for virtual access to current object.

4.2.1.3 The RdbPostgresqlSystem class

This class is used to create an instance representing the PostgreSQL database system. Only one such object is created in the CORAL runtime and all RdbPostgresqlDatabase objects will be managed by this instance. One attribute and three member functions will be explained as follows:

Attribute:

pRdbPostgresqlDatabase: RdbPostgresqlDatabase type pointer used to hold the created database object.

Operations:

OpenDatabase(): creates a RdbPostgresqlDatabase object.

SystemName(): returns string 'postgresql'.

Implementor(): returns this.

4.2.2 RDB database classes

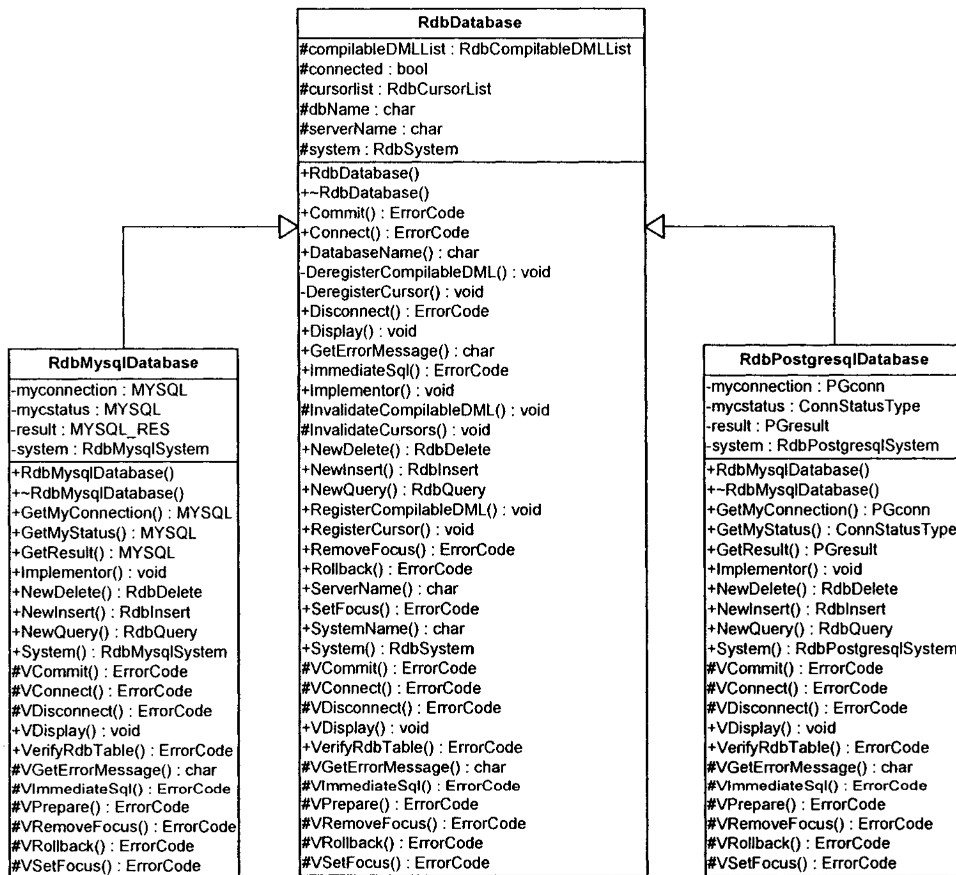

Figure 4-24 RDB Database Classes

4.2.2.1 The RdbDatabase class

This class is used to communicate with the backend database and invoke any operations on it. Template methods, such as commit(), immediateSql(), and Display() , are provided by this class. Fifteen pure virtual functions are defined in this class. The implementations of them are deferred to the concrete child classes.

4.2.2.2 The RdbMysqlDatabase class

This class is used to establish the connection with the backend MySQL database. It consists of four attributes and twenty one operations. The main attributes and operations are introduced as follows:

Attributes:

myconnection: MYSQL type handler for MySQL database connections.

mycstatus: type handler for MySQL database connection status.

result: MYSQL_RES type handler for holding the result set from MySQL.

system: RdbMysqlSystem type pointer pointing to the system the database is in.

Operations:

RdbMysqlDatabase(): constructs the object and initializes each attribute in it. During construction, a connection will be established to MySQL through its C API. The connection will be assigned to myconnection. All MySQL commands called in this procedure were introduced in chapter 2.

~RdbMysqlDatabase(): destroys the connection with MySQL and releases memory held by it.

NewQuery(): initializes a RdbMysqlQuery object for the query request from caller.

NewInsert(): initializes a RdbMysqlInsert object to insert a tuple to the backend database.

NewDelete(): creates a RdbMysqlDelete object to delete tuples from the backend database.

VCommit(): virtual function, commits the backend database if it supports transactions.

VDisplay(): displays the result returned from the MySQL database.

VerifyRdbTab(): checks whether the table exists in the MySQL database before creating a Cor_CRdbRelation on the table.

VImmediateSQL(): executes the query through a connection to MySQL.

4.2.2.3 The RdbPostgresqlDatabase class

This class is used to establish the connection with the backend PostgreSQL database. Like RdbMysqlDatabase, it consists of four attributes and twenty one operations. The main attributes and operations are illustrated as follows:

Attributes:

myconnection: PostrgreSQL type handler for PostgreSQL database connections.

mycstatus: type handler for database connection status.

result: PGresult type handler for holding the result set from PostgreSQL.

system: RdbPostgresqlSystem type pointer pointing to the system the database is in.

Operations:

RdbPostgreSQLDatabase(): constructor, uses the C API to make the connection.

~RdbPostgreSQLDatabase(): destroys the connection with PostgreSQL and releases memory held by it.

NewQuery(): initializes a RdbPostgreSQLQuery object.

NewInsert(): initializes a RdbPostgreSQLInsert object.

NewDelete(): creates a RdbPostgreSQLDelete object.

VCommit(): . commits the backend database if it supports transactions.

VDisplay(): displays the result returned from the PostgreSQL database.

VerifyRdbTab(): checks whether the table exists in the PostgreSQL database before creating a Cor_CRdbRelation on the table.

VImmediateSQL(): executes the query through a connection to PostgreSQL.

4.2.3 RDB query classes

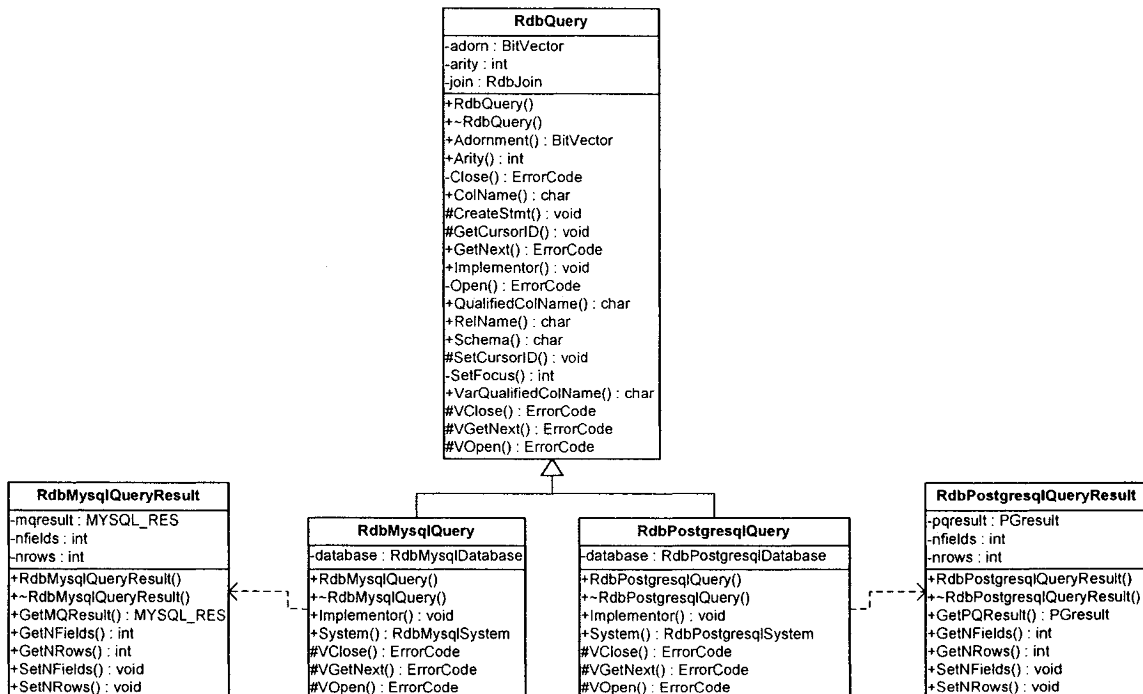

Figure 4-25 RDB Query Classes

4.2.3.1 The RdbQuery class

As an abstract class, RdbQuery consists of three attributes and twenty operations. The SQL statement that will be executed in the backend database is created by this class. Its attributes and main functions are described below:

Attributes:

adorn: the attributes that will be known at open.

arity: total number of attributes in the result set.

join: a RdbJoin object that maintains the set of relations, and any join conditions between those columns.

Operations:

Adornment():states which attributes will have selection criteria on them when a cursor is opened on the query.

Arity(): returns the number of attributes in the query.

CreateStmt(): creates the standard SQL query for the backend database.

QualifiedColName(): creates the relationName.columnName style string for the query

4.2.3.2 The RdbMysqlQuery class

This class mainly implements the virtual operations derived from the RdbQuery class. It executes the SQL command through the connection with the MySQL database and retrieve tuples as required. The attributes and main operations in it follow:

Attributes:

database: a reference of the RdbMysqlDatabase this query object belongs to.

Operations:

VOpen(): runs the query at the backend database and creates a RdbMysqlQueryResult object to store the result set.

VGetNext(): retrieves one tuple for the result set and converts each attribute to the data type that CORAL accepts. Currently, CORAL only supports three data types, integer, real number, and string. As mentioned in 2.2.3, all MySQL data type will be changed to CORAL's data type according to Table 2-4. The result will be returned to the Cor_CRdbRelation object and coverts to a CORAL tuple at end.

VClose(): destroys the RdbMysqlQueryResult object created by VOpen and releases the memory held by the MySQL result set structure.

4.2.3.3 The RdbMysqlQueryResult class

This class is used to manage the query result set created by the MySQL database. It is composed of three attributes and seven operations:

Attributes:

mqresult: a MYSQL_RES type pointer points to the result set returned from MySQL.

nfields: the number of fields the result set has.

nrows: the position of the row that VGetNext last visits.

Operations:

RdbMysqlQueryResult(): constructor

~ RdbMysqlQueryResult(): releases the memory held by the result set and destroys the RdbMysqlQueryResult object.

GetMQResult(): returns the handler to the result set.

GetNFields(): returns the value of nfields.

GetNRows(): return the value of nrows.

SetNFields(): assigns a new value to nfields.

SetNRows(): assigns a new value to nrows.

4.2.3.4 The RdbPostgresqlQuery class

This class mainly implements the virtual operations to create and traverse the result set from a PostgreSQL database. The specification of it is as follows:

Attribute:

database: a handler of the RdbPostgresqlDatabase this query object belongs to.

Operations:

VOpen(): runs the query at the backend database and creates a RdbPostgresqlQueryResult object to manage the result set.

VGetNext(): retrieves one tuple for the result set and converts each attribute to the data type that CORAL accepts. All conversion is based on Table 2-6.The result will be converted to a CORAL tuple by the Cor_CRdbRelation object.

VClose(): destroys the RdbPostgresqlQueryResult object.

4.2.3.5 The RdbPostgreSQLQueryResult class

This class is used to manage the query result set created by the PostgreSQL database. It is composed of three attributes and seven operations:

Attributes:

pqresult: a PGresult type handler points to the result set returned from PostgreSQL.

nfields: the number of fields the result set has.

nrows: the position of the row that VGetNext last visits.

Operations:

RdbPostgreSQLQueryResult(): constructor

~ RdbPostgreSQLQueryResult(): releases the memory held by the result set and destroys the RdbPostgresqlQueryResult object.

GetPQResult(): returns the handler to the result set.

GetNFields(): returns the value of nfields.

GetNRows(): return the value of nrows.

SetNFields(): assigns a new value to nfields.

SetNRows(): assigns a new value to nrows.

4.2.4 RDB insert classes

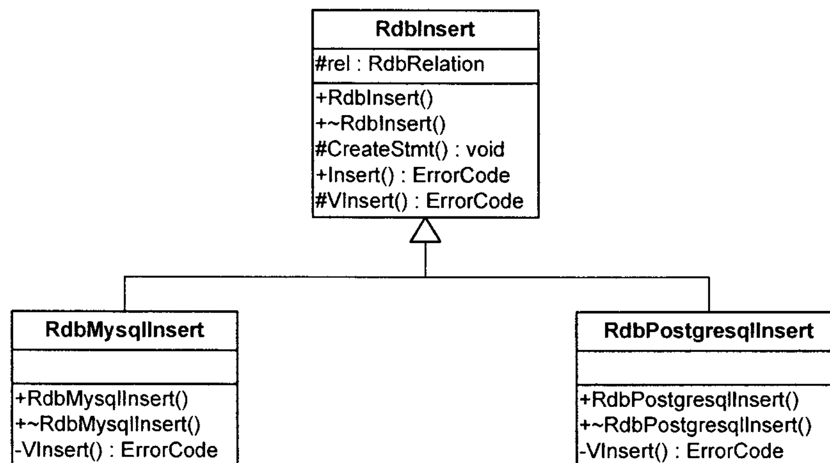

Figure 4-26 RDB Insert Classes

4.2.4.1 The RdbInsert class

This class defines a template method, Insert(), to execute the insert request on the backend database. VInsert() is an abstract operation that will be implemented in the derived concrete classes. The operation CreateStmt() will be used to create the insert query in the SQL format. It will use the rel attribute to the get the backend table name and attributes names.

4.2.4.2 The RdbMysqlInsert class

This class consists of three operations. The functions of these operations are described below:

Operations:

RdbMysqlInsert(): constructor, initlializes the RdbMysqlInsert object and its parent instance RdbInsert.

~RdbMysqlInsert(): destructor.

VInsert(): creates the insert query using the CreateStmt() operation and runs it at the backend database. The result is displayed to the caller.

4.2.4.3 The RdbPostgresqlInsert class

This class performs the same functions on PosgreSQL database as the RdbMysqlInsert on MySQL database. Descriptions of the three operations are ignored since they work in the same way as those described in 4.2.4.2.

4.2.5 RDB delete classes

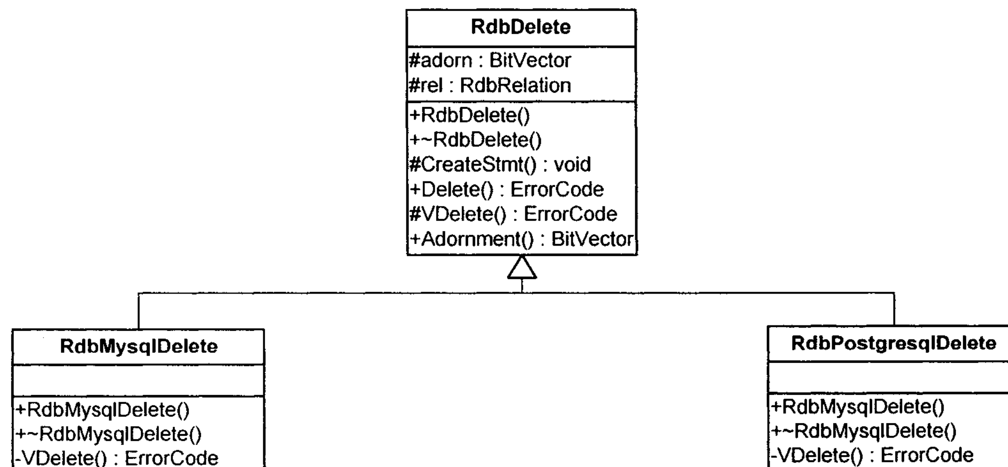

Figure 4-27 RDB Delete Classes

4.2.5.1 The RdbDelete class

This class consists of two attributes and six operations. The description follows:

Attributes:

adorn: stores the expected binding for the relation.

rel: a pointer to the RdbRelation object that will perform the delete operation.

Operations:

RdbDelete(): constructor, initializes its parent instance and attributes.

~RdbDelete(): destructor.

CreateStmt(): creates the delete query based on the values of adorn and rel.

Delete(): template method that calls the VDelete() operation.

VDelete(): abstract operation.

Adornment(): return the value of adorn.

4.2.5.2 The RdbMysqlDelete class

This class consists of three operations. The description follows:

Operations:

RdbMysqlDelete(): constructor, initializes its parent instance and attributes.

~RdbMysqlDelete(): destructor.

VDelete(): calls CreateStmt() in the parent class to create the delete query and executes it on the backend MySQL database. The result will be shown to the caller.

4.2.5.3 The RdbPostgresqlDelete class

This class consists of three operations that are described as below:

Operations:

RdbPostgresqlDelete(): constructor, initializes its parent instance and attributes.

~RdbPostgresqlDelete(): destructor.

VDelete(): calls CreateStmt() in the parent class to create the delete query and executes it on the backend PostgreSQL database. The result will be shown to the caller.

4.3 Database Design

In the thesis, MySQL and PostgreSQL are used as the backend databases system. They must be installed and run on a location that can be reached by CORAL. Corresponding databases are created on both systems for test purposes. Currently, a user 'gwang_1' is created on MySQL. The user's working database is 'mydb'. On postgreSQL, user 'wang' is activated and performs operations on database 'pgsqldb'.

The tables created in the MySQL and PostgreSQL databases are based on a standard University Model benchmark [2]. Twenty tables as well as the sample data are created. In order to support transactions and reference constraints, INNODB tables are used in MySQL. Extra indexes are created to support the foreign keys. This University Model is also the test benchmark for the Graph Database System introduced in Chapter 6.

All these operations are performed in standard query language supported by each system individually. The script files for creating these tables in MySQL and PostgreSQL can be found in appendix A and appendix B. Commands for other operations, such as creating users, creating databases, and assigning privileges, can be found in the online documents at http://www.mysql.org and http://www.postgresql.org.

Chapter 5 Test

In this chapter, the implementation of CORAL's interfaces to MySQL and PostgreSQL is tested. At the beginning, the test requirement is specified. Then, test methods are illustrated in order to make the test procedure clear. Lastly, test cases are used to describe step-by-step test procedures and to verify the compliance with the requirements in chapter 3.

5.1 Test Requirement

All functional requirements stipulated in chapter 2 are tested to ensure that the system can provide these functions properly. These requirements include:

- Connections from CORAL to MySQL and from CORAL to PostgreSQL can be established and work correctly as required;
- Tables in MySQL and PostgreSQL can be mapped correctly to the current workspace of CORAL;
- Regular operations, such as query, insert tuple, and delete tuple, can be performed properly on the mapped tables with CORAL's standard commands;
- Both DDL and DML commands other than those stipulated by CORAL can be performed precisely on the backend MySQL and PostgreSQL through CORAL;
- Transactions can be supported if the backend database is set so.

5.2 Test Methods

In this these, all the requirements, design, and implementation are based on CORAL's structure. The test methods are selected based on this condition.

The unit test is mainly a static test by checking syntax and inspecting code. A white box method is used to focus on the internal structure of each class. During the integration test, the black-box method is emphasized in order to test the behavior and functionality of the system. These system tests ensure that the system performs as required.

5.3 Test Cases

All these tests fall into ten test cases. Each test case consists of a title, ID, rationale, description, and the reference to the requirement and solution mentioned in the previous chapters. All conditions are stipulated following each test case.

5.3.1 Connect CORAL to MySQL

Title: Connect CORAL to MySQL

ID: TC01

Rationale: to check whether the connection from CORAL to MySQL can be established properly.

Description: Connect to MySQL database 'mydb' as the user 'gwang_1' identified by password 'sundy'. The MySQL server is running on 'jeeves'. The created CORAL database name is 'mysqldb'.

Reference: UC01, UC011, D4.2.1.2, D4.2.2.2

Condition 1: Pass: connect correctly

Input data: rdb_open_db(mysqldb, mysql, mydb, gwang_1, sundy, jeeves).

rdb_open_db(mysqldb, mysql, mydb, gwang_1,sundy).

Condition 2: Fail: connect in wrong format.

Input data: rdb_open_db(, mysql, mydb, gwang_1, sundy, jeeves).

rdb_open_db(mysqldb , , mydb, gwang_1, sundy, jeeves).

rdb_open_db(mysqldb, mysql, ,gwang_1, sundy, jeeves).

Condition 3: Fail: connect with wrong MySQL identifier.

Input data: rdb_open_db(mysqldb, MySQL, mydb, gwang_1,sundy,jeeves).

Condition 4: Fail: connect with wrong MySQL database name.

Input data: rdb_open_db(mysqldb, mysql, mydatabase, gwang_1, sundy,jeeves).

Condition 5: Fail: connect with wrong username.

Input data: rdb_open_db(mysqldb, mysql, mydb, guangwang, sundy,jeeves).

Condition 6: Fail: connect with wrong password.

Input data: rdb_open_db(mysqldb, mysql, mydb, gwang_1,sunday,jeeves).

Condition 7: Fail: connect with wrong server name.

Input data: rdb_open_db(mysqldb, mysql, mydb, gwang_1,sundy,jee).

Condition 8: Fail: the CORAL database already exists (create mysqldb twice).

Input data: rdb_open_db(mysqldb, mysql, mydb, gwang_1,sundy,jeeves).

5.3.2 Connect CORAL to PostgreSQL

Title: Connect CORAL to PostgreSQL

ID: TC02

Rationale: to check whether the connection from CORAL to PostgreSQL can be established properly.

Description: Connect to PostgreSQL database 'pgsqldb' as the user 'wang' identified by password 'guang'. The created CORAL database name is 'mypgdb'.

Reference: UC01, UC012, D4.2.1.3, D4.2.2.3

Condition 1: Pass: connect correctly

Input data: rdb_open_db(mypgdb, postgresql, pgsqldb, wang, guang).

Condition 2: Fail: connect in wrong format.

Input data: rdb_open_db(, postgresql, pgsqldb, wang, guang).

rdb_open_db(mypgdb, , pgsqldb, wang, guang).

rdb_open_db(mypgdb, postgresql, , wang, guang).

Condition 3: Fail: connect with wrong PostgreSQL identifier.

Input data: rdb_open_db(mypgdb, PostgreSQL, pgsqldb, wang, guang).

Condition 4: Fail: connect with wrong PostgreSQL database name.

Input data: rdb_open_db(mypgdb, postgresql, pgdb, wang, guang).

Condition 5: Fail: connect with wrong username.

Input data: rdb_open_db(mypgdb, postgresql, pgsqldb, guangwang, guang).

Condition 6: Fail: connect with wrong password.

Input data: rdb_open_db(mypgdb, postgresql, pgsqldb, wang, sunday).

Condition 7: Fail: the CORAL database already exists (create mypgdb twice).

Input data: rdb_open_db(mypgdb, postgresql, pgsqldb, wang, guang).

5.3.3 Create CORAL mapped tables based on MySQL tables

Title: Create CORAL Mapped tables based on MySQL tables

ID: TC03

Rationale: to check whether the MySQL tables can be mapped to CORAL through the database connections

Description: Map a MySQL table 'person (ID, Name)' to CORAL current workspace under name 'cperson'. The database connection used is 'mysqldb'.

Reference: UC02, UC021, D4.2.1.2, D4.2.2.2

Condition 1: Pass: mapped correctly

Input data: rdb_map(mysqldb, cperson(person("ID", "Name"))).

Condition 2: Fail: input in wrong format.

Input data: rdb_map(mysqldb, cperson(person(ID, Name))).

rdb_map(mysqldb, cperson(person(ID, Name)).

rdb_map(mysqldb, cperson(person)).

Condition 3: Fail: input in a wrong MySQL based CORAL database name

Input data: rdb_map(mysqld, cperson(person(ID, Name))).

Condition 4: Fail: create cperson twice

Input data: rdb_map(mysqldb, cperson(person(ID, Name))).

Condition 5: Fail: wrong back end table name

Input data: rdb_map(mysqldb, cperson(pers (ID, Name))).

5.3.4 Create CORAL mapped tables based on PostgreSQL tables

Title: Create CORAL Mapped tables based on PostgreSQL tables

ID: TC04

Rationale: to check whether the PostgreSQL tables can be mapped to CORAL through the database connections

Description: Map a PostgreSQL table 'person (ID, Name)' to CORAL current workspace under name 'cperson'. The database connection used is 'mypgdb'.

Reference: UC02, UC022, D4.2.1.3, D4.2.2.3

Condition 1: Pass: mapped correctly

Input data: rdb_map(mypgdb, cperson (person("ID", "Name"))).

Condition 2: Fail: input in wrong format.

Input data: rdb_map(mypgdb, cperson (person(ID, Name))).

rdb_map(mypgdb, cperson (person(ID, Name)).

rdb_map(mypgdb, cperson (person)).

Condition 3: Fail: input in a wrong PostgreSQL based CORAL database name

Input data: rdb_map(mysqld, cperson (person(ID, Name))).

Condition 4: Fail: create cperson twice

Input data: rdb_map(mypgdb, cperson (person(ID, Name))).

Condition 5: Fail: wrong back end table name

Input data: rdb_map(mypgdb, cperson (pers (ID, Name))).

5.3.5 Create CORAL joined table

Title: Create CORAL Joined table

ID: TC05

Rationale: to check whether the mapped tables can be joined together to create a CORAL table. This test case is for both MySQL and PostgreSQL.

Description: Join two CORAL mapped tables, cperson(based on 'person (ID, Name)') and cstudent(based on 'student(ID)'), and create a CORAL table newstudent.

Reference: UC03, D4.2.1.2, D4.2.1.3, D4.2.2.2, D4.2.2.3

Condition 1: Pass: joined table created correctly

Input data: rdb_join(newstudent(cperson(X,Y), cstudent(X))).

Condition 2: Fail: input in wrong format.

Input data: rdb_join(newstudent(cperson, cstudent)).

rdb_join(newstudent(cperson(x,y), cstudent(x))).

Condition 3: Fail: newstudent already exists in current workspace.

Input data: rdb_join(newstudent(cperson(X,Y), cstudent(X))).

Condition 4: Fail: the source relations do not exist or the names are wrong names.

Input data: rdb_join(newstudent(cpers(X,Y), cstudent)).

5.3.6 Query mapped tables

Title: Query mapped tables

ID: TC06

Rationale: to check whether the mapped tables accept CORAL query command. This test case is for both MySQL and PostgreSQL.

Description: Query cperson, which is mapped from table 'person (ID,Name)'.

Reference: UC04, UC041, D4.2.3.2, D4.2.3.3, D4.2.3.4, D4.2.3.5

Condition 1: Pass: query result returned correctly.

Input data: ?cperson(X,Y).

Condition 2: Fail: input in wrong format.

Input data: ?cperson(X,Y.

Condition 3: Fail: input wrong table name.

Input data: ?person(X,Y).

Condition 4: Fail: input wrong table attribute.

Input data: ?cperson(X,Y,Z).

5.3.7 Insert tuple

Title: Insert tuple

ID: TC07

Rationale: to check whether tuples can be inserted to the backend database by standard CORAL command. This test case is for both MySQL and PostgreSQL.

Description: Insert tuple ("001", "John English") to table 'person(ID,Name)' through the mapped table cperson.

Reference: UC05, D4.2.4.2, D4.2.4.3

Condition 1: Pass: a tuple is inserted to the backend database.

Input data: insert (cperson("001", "John English")).

cperson("001", "John English").

Condition 2: Fail: input in wrong format.

Input data: insert (cperson(001,John English)

Condition 3: Fail: input wrong table name.

Input data: insert (person("001", "John English")).

Condition 4: Fail: input wrong table attribute.

Input data: insert (cperson("001")).

5.3.8 Delete tuple

Title: Delete tuple

ID: TC08

Rationale: to check whether tuples can be deleted from the backend database by standard CORAL command. This test case is for both MySQL and PostgreSQL.

Description: Delete tuple ("001", "John English") from table 'person(ID,Name)' through the mapped table cperson.

Reference: UC06, D4.2.5.2, D4.2.5.3

Condition 1: Pass: the tuple is deleted from the backend database.

Input data: delete (cperson("001", "John English")).

Condition 2: Fail: input in wrong format.

Input data: delete (cperson("001", "John English")

Condition 3: Fail: input wrong table name.

Input data: delete (person("001", "John English")).

Condition 4: Fail: input wrong table attribute.

Input data: delete (cperson(_,"John English")).

5.3.9 Execute command

Title: Execute command

ID: TC09

Rationale: to check whether DDL and DML commands can be executed on the backend database through CORAL interface. This test case is for both MySQL (mysqldb) and PostgreSQL(mypgdb).

Description: Show all tuples in table 'person', then drop a table called 'nouse'.

Reference: UC07, UC071, UC072, D4.2.1.2, D4.2.1.3, D4.2.2.2, D4.2.2.3

Condition 1: Pass: the command is executed on the backend database.

Input data: rdb_execute(mysqldb, "select * from person").

rdb_execute(mypgdb, "select * from person").

rdb_execute(mysqldb, "drop table nouse").

rdb_execute(mypgdb, "drop table nouse").

Condition 2: Fail: input in wrong format.

Input data: rdb_execute(mysqldb, select * from person).

rdb _execute(mypgdb, select * from person).

rdb _execute(mysqldb, drop table nouse).

rdb _execute(mypgdb, drop table nouse).

Condition 3: Fail: input wrong database connection name.

Input data: rdb_execute(mydb, "select * from person").

rdb_execute(pgsqldb, " select * from person").

5.3.10 Transactional support

Title: Transactional support

ID: TC10

Rationale: to check whether CORAL's rdb_commit and rdb_rollback commands can affect the transaction of the backend databases. This test case is for both MySQL (mysqldb) and PostgreSQL(mypgdb).

Description: The backend database is committed or rolled back as required.

Reference: UC08, UC09, D4.2.1.2, D4.2.1.3, D4.2.2.2, D4.2.2.3

Condition 1: Pass: the command is executed on the backend database.

Input data: rdb_commit(mysqldb).

rdb_commit(mypgdb).

rdb_rollback(mysqldb).

rdb_rollback(mypgdb).

Condition 2: Fail: input in wrong format.

Input data: rdb_commit(mysqldb

rdb_commit(mypgdb

rdb_rollback(mysqldb

rdb_rollback(mypgdb

Condition 3: Fail: input wrong database connection name.

Input data: rdb_commit(wrongdb).

rdb_rollback(wrongdb).

Chapter 6 Application

The work demonstrated in this thesis is mainly for the Graph Database System developed at Concordia University. This System intends to apply the benefits of deductive query language, diagrammatic queries, and visualized results more broadly in genomics. JAVA, XML, C/C++, CORAL, MySQL and PostgreSQL are used to implement this system in UNIX environment.

The Graph Database System provides a visual query mechanism to manage genomics data. Its JAVA based interface allows scientists to construct diagrams to express the query, which shows the entities of their interest and the relationships among these entities. The supported graphical query language is GraphLog [7]. This system manages the translation from a query diagram to a textual CORAL query program. Then, CORAL is called to process the query and deduce the result. All raw data are stored in MySQL and PostgreSQL databases and loaded to the memory as required. At last, the query result set is also visualized as diagrams with the same icons and style as in the query and displayed to the user.

Eventually, this system will interface directly to our Know-It-All framework for databases [8].

6.1 System Architecture

As shown in Figure 6-1, this system consists of five layers, GUI, TGL Translator[9], CORAL Client, CORAL Server and Data Storage (MySQL & PostgreSQL). The description of responsibilities for each layer follows:

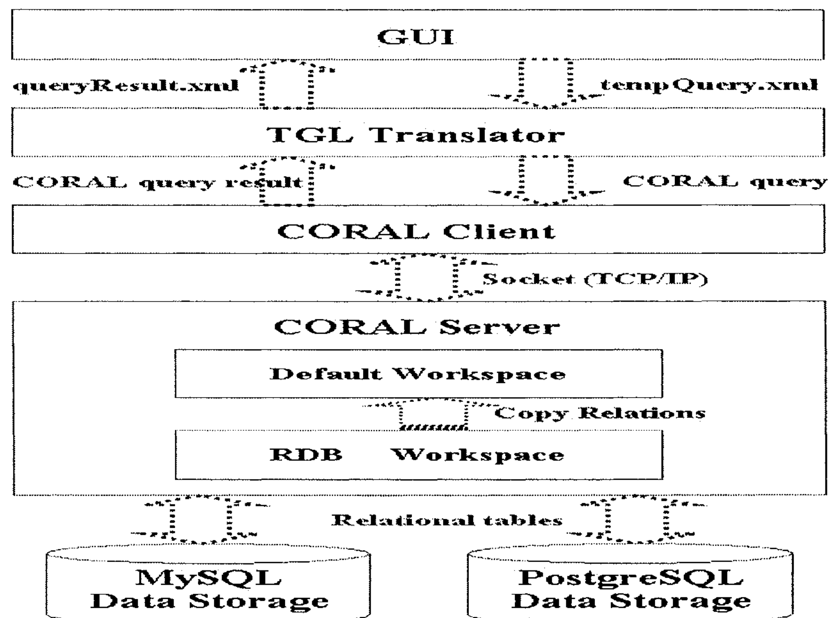

Figure 6-1 Graph Database System Architecture

GUI: The GUI is the system's interface to end users. End users may draw a query in the query editor. The GUI component translates user's query that is defined as a diagram into XML format and sends it to the next layer of the system: TGL Translator. TGL (*T*ransferable *G*raphic *L*anguage) is an XML format that defines the communication protocol between GUI layer and TGL translator layer. The GUI is also responsible for visualizing the query result set into a graph.

TGL Translator: The TGL translator is the transformation engine that transforms an XML formatted query, which it receives from the GUI layer, to a CORAL query

program. A set of translations rules is defined in TGL translator to regulate the translation from an XML formatted query to a CORAL program. The TGL translator calls up the CORAL client. TGL translator is also responsible for transforming the CORAL query result into XML and pass the XML-format query result to the upper GUI layer.

CORAL Client: The CORAL client is responsible for two tasks: one is to receive a query plan from TGL translator and to send the CORAL query programs in the query plan to the CORAL Server; the other is to receive the query result from the CORAL server, and pass it to the TGL translator.

CORAL Server: During the CORAL server initialization, two workspaces, default workspace and RDB workspace, are created and work interactively to manage incoming and outgoing data. The default workspace is responsible for maintaining CORAL's relations and executing queries. The RDB workspace is in charge of connecting with MySQL and PostgreSQL as well as manipulating relational data. At the RDB workspace, a dictionary describing the mapping between relational tables and corresponding relations in CORAL database is constructed first. Then the data in the relational tables are loaded into CORAL's default workspace according to this dictionary. A CORAL program sent by a CORAL client is evaluated and executed in CORAL server and the query result is returned to the CORAL client.

Data Storage: The data source is stored physically in MySQL and PostgreSQL databases. The conventional data manipulations can be performed on data in these databases. In current version, during the CORAL server initialization, the connections between these extensional databases and CORAL are set up. All the stored tuples in the

target database are loaded into the CORAL server's computer main memory as a runtime database for CORAL system.

6.2 Integrating CORAL with the TGL Translator

For the five layers described in 6.1, the work in this thesis is mainly focusing on the last three layers. The original TGL Translator starts a stand-alone CORAL process each time and terminates it when completes. Facts need to be loaded in memory each time CORAL runs. This method is inefficient, especially when loading huge amount of genomic data. Therefore, the client-server model CORAL is utilized to avoid extra data load time.

CORAL supports client-server model. CORAL client connects with CORAL server by a socket under TCP/IP protocol. In this system, a CORAL client was designed and integrated with the TGL Translator to perform the required tasks. It is invoked by the TGL Translator and communicates with the CORAL server. From the TGL Translator point of view, it performs as same as the previous stand-alone CORAL process.

The main functions of the CORAL client are as follows:

•Connects CORAL server with a socket;

•Sends requests received from TGL Translator to CORAL server;

•Receives result from CORAL server and stores it to the target file;

•Disconnects and terminates when receiving the EXIT signal from CORAL server.

The CORAL client terminates when the query finishes, whereas the CORAL server will live until the user requires to shut it down.

6.3 Optimization Experiment

In this system, CORAL works as the deductive engine to evaluate queries and deduce results. As mentioned in 2.1.5, although CORAL developed a number of query evaluation strategies, it still uses heuristic programming rather than a cost estimation package to choose evaluation methods [1]. Annotations are utilized to guide query optimization and control query evaluation. There are system-level annotations and user-level annotations. User-level annotations can be added directly to the source code and they give the programmer freedom to control query's optimization as well as evaluation.

CORAL's user-level annotations are divided into Rewriting Annotations, Execution Annotations, and Per-Predicate Annotations. Presently, CORAL's Rewriting Annotations, which include *Supplementary Magic Templates* [10], *Magic Templates* [11], *Context Factoring* [12], *naïve backtracking* [13], and *Without Rewriting* method, have been tested. The test platform was a SunFire 280R with two 900MHz UltraSparc-III+ CPUs and 4GB physical memory. The operating system is Solaris 9. A standard benchmark consisting of the University Model and a set of well documented queries were used rather than invented our own genomic DB benchmark at this stage [2]. The test data set was based on 100,000 person facts. The total number of ground facts was 635,813. The 20 queries used in the test were fully documented in [9]. All tests were made on the MySQL solely.

These five annotations were tried individually on each query and the query execution time was recorded on the CORAL server. Then the comparative speed was calculated for each method relative to CORAL's default optimization strategy, *Supplementary Magic*. The experiment results are listed in Table 6-1.

	No rewriting	Sup. magic	Magic	Context factoring	Naïve backtracking
Q1	0.92	1	0.96	0.96	0.96
Q2	0.90	1	0.93	1.01	0.96
Q4	139.56	1	0.89	0.89	0.89
Q6	1.08	1	1	0.92	1
Q7	4.93	1	1.01	0.96	0.97
Q8	1.03	1	1.03	1	1
Q9	1.06	1	1.14	1.10	1.09
Q10	0.91	1	0.91	0.87	0.95
Q13	0.99	1	1.01	1.02	1
Q14	1.13	1	1	1.13	1.13
Q17	1.02	1	0.98	1.04	1.05
Q19	0.58	1	0.65	1	1
Q20	1.01	1	1	0.99	0.99
Q21	0.90	1	1	0.90	1
Q22	60.85	1	1	1	1.08
Q23	23.53	1	0.8	1.07	0.93
Q24	1	1	1	1	1.08
Q25	1.11	1	1.11	1	1.11
Q26	1.1	1	1	1.1	1
Q27	49.44	1	0.94	0.94	0.94

Table 6-1. Relative Query Execution Time

From the result we can see that there is no single method that outperforms the others. Each method works the best with a subset of the queries. For example, the *Magic Templates* have the best performance in Q23; the *Context Factoring* runs fastest in Q6 and Q10. However, the variation relative to the default optimization strategy is no more than 10%. Clearly, one of the optimization strategies should be used rather than no strategy (the "no rewriting" strategy column in Table 1). These test results will be the inital guidelines to design an efficient cost estimation package based on CORAL's methods for the requirements of genomics.

Chapter 7 Conclusion

In this thesis, CORAL's extensional database interface as well as CORAL's data management classes have been studied and explained. The connections from CORAL to MySQL and from CORAL to PostgreSQL have been designed, implemented, and tested based on CORAL's class structure and rules. A client process has been designed and coded according to CORAL's client-server protocols. During the design and implementation, design patterns have been applied to the classes for connecting MySQL and PostgreSQL. UML [15] has been used to illustrate system architecture, runtime behavior, and interaction of these classes.

As an application of these solutions, the Graph Database System has been introduced and its architecture has been demonstrated. Integration of the CORAL client with the JAVA based TGL translator has been explained in order to specify the roles of the CORAL client. CORAL's optimization methods have also been tested on the Graph Database System in order to find a better way to process genomics data.

The future work of the thesis may focus on adding new data types to both CORAL and the extensional databases, i.e. MySQL and PostgreSQL, based on the characteristics of genomics data. The update of CORAL system, both its class structure and methods, is also a big future work. Moreover, to design an index for the result set retrieved from the extensional database is also a potential improvement for CORAL's data management.

References

[1] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri, "The CORAL Deductive System", *VLDB Journal*, vol 3, no. 2, pp. 161–210, 1994.

[2] K. C. Chan, P. W. Trinder, R. Welland, "Evaluating Object-Oriented Query Languages", *Computing Journal*, vol 37, no. 10, pp. 858–872, 1994.

[3] R. Ramakrishnan, D.Srivastava, S. Sudarshan, P. Seshadri, "Implementation of the CORAL Deductive Database System", *SIGMOD Conference*, pp. 167-176, 1993.

[4] M. Carey, D. DeWitt, J. Richardson, and E. Shekita, "Object and file management in the EXODUS extensible database system", *In Proceedings of the International Conference on Very Large Databases*, pp. 91-100, August 25-28, 1986.

[5]. M. Widenius, D. Axmark, *MySQL Reference Manual*, O'Reilly & Associates, Incorporated, June 2002.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] M. P. Consens, F. Ch. Eigler, M. Z. Hasan, A. O. Mendelzon, E. G. Noik, A. G. Ryman, and D. Vista, "Architecture and Applications of the Hy+ Visualization System", *IBM Systems Journal*,vol 33, no. 3, pp. 458– 476, 1994.

[8] G. Butler, L. Chen, X. Chen, A. Gaffar, J. Li, L. Xu, "The Know-It-All project: A Case Study in Framework Development and Evolution", *Domain Oriented Systems Development: Perspectives and Practices*, Taylor and Francis Publishers, UK, 2002.

[9] L. Zou, "GraphLog: Its Representation in XML and Translation to CORAL", *Master Thesis*, Dept. of Computer Science, Concordia University, 2003.

[10] C. Beeri and R. Ramakrishnan, "On the Power of Magic", *Procs. of the ACM Symposium on Principles of Database Systems*, pp. 269–283, 1987.

[11] R. Ramakrishnam, "Magic Templates: A Spellbinding Approach to Logic Programs", *Procs. of the International Conference on Logic Programming*, pp. 140–159, 1988.

[12] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman, "Argument Reduction Through Factoring", *Procs. of the Fifteenth International Conference on Very Large Databases*, pp. 173–182, 1989.

[13] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, "Rule ordering in bottom-up fixpoint evaluation of logic programs", *Procs. of the International Conference on VeryLarge Data Bases*, pp. 359- -371, 1990.

[14] R. Ramakrishnan, P. Seshadri, D. Srivastava, S. Sudarshan, *The CORAL User Manual, A Tutorial Introduction to CORAL*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[15] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

Appendix

A. University Data Model Schema (script files) for MySQL

How to run:

Shell> mysql –u gwang_1 –p < University-INNODB.sql
When loading data from file:
Shell> mysql –u gwang_1 –p < University-INNODB.sql **--local-infile=1**

University-INNODB.sql

##Tips: For FOREIGN KEY CONSTRAINT, the foreign key values must be in the first place of index. For example, in table works_in, although compound indexes have been created for primary keys ID and Dept, a separate index must be created for Dept in order to make it the first place of the index.
INNODB supports transaction and reference constraint from version 3.23.34. It is the default setting from version 4.0.

```
# 1. Select default MYSQL DB
use mydb;
# 2. Initialize the database
drop table if exists resides,second_supervisor,first_supervisor,lives_in,assessment;
drop table if exists prerequisites,run_by,takes,supervises,majors_in,teaches,works_in;
drop table if exists address,course,dept,visiting_staff,tutor,student,staff,person;
# 3. Create tables
create table person
(
    ID  varchar(10) NOT NULL,
    Name        varchar(50),
        PRIMARY KEY (ID)
)
TYPE = InnoDB;

create table staff
(
        ID      varchar(10) NOT NULL,
        Salary  int,
        PRIMARY KEY (ID),
        FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table student
(
        ID      varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table tutor
(
```

```
        ID        varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table visiting_staff
(
        ID        varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table dept
(
        No        varchar(10) NOT NULL,
        Name      varchar(50),
    PRIMARY KEY (No)
)
TYPE = InnoDB;

create table course
(
        Code      varchar(10) NOT NULL,
        Title     varchar(50),
        Credit    int,
    PRIMARY KEY (Code)
)
TYPE = InnoDB;

create table address
(
        AID       varchar(10) NOT NULL,
        Street    varchar(50),
        District  varchar(20),
        City      varchar(20),
    PRIMARY KEY (AID)
)
TYPE = InnoDB;

create table works_in
(
        ID        varchar(10) NOT NULL,
        Dept      varchar(10) NOT NULL,
        PRIMARY KEY (ID,Dept),
        FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
        INDEX (Dept),
        FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE
)
TYPE = InnoDB;

create table teaches
(
        ID        varchar(10) NOT NULL,
        Course    varchar(10) NOT NULL,
```

```
    PRIMARY KEY (ID,Course),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (Course),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
)
TYPE = InnoDB;

create table majors_in
(
        ID      varchar(10) NOT NULL,
        Dept    varchar(10) NOT NULL,
    PRIMARY KEY (ID,Dept),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (Dept),
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE

)
TYPE = InnoDB;

create table supervises
(
        StaffID varchar(10) NOT NULL,
        StudentID       varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (StudentID),
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table takes
(
        ID      varchar(10) NOT NULL,
        Course  varchar(10) NOT NULL,
    PRIMARY KEY (ID,Course),
    FOREIGN KEY (ID) REFERENCES student (ID) ON DELETE CASCADE,
    INDEX (Course),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
)
TYPE = InnoDB;

create table run_by
(
        Course  varchar(10) NOT NULL,
        Dept    varchar(10) NOT NULL,
    PRIMARY KEY (Course,Dept),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE,
    INDEX (Dept),
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE
)
TYPE = InnoDB;

create table prerequisites
(
        Course  varchar(10) NOT NULL,
        PreCourse       varchar(10) NOT NULL,
```

```
    PRIMARY KEY (Course,PreCourse),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE,
    INDEX (Precourse),
    FOREIGN KEY (Precourse) REFERENCES course (Code) ON DELETE CASCADE
)
TYPE = InnoDB;

create table assessment
(
        Course  varchar(10) NOT NULL,
        AssName         varchar(10) NOT NULL,
        Percent double(3,2),
    PRIMARY KEY (Course,AssName),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
)
TYPE = InnoDB;

create table lives_in
(
        ID      varchar(10) NOT NULL,
        AID     varchar(10) NOT NULL,
    PRIMARY KEY (ID,AID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (AID),
    FOREIGN KEY (AID) REFERENCES address (AID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table first_supervisor
(
        StaffID varchar(10) NOT NULL,
        StudentID       varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (StudentID),
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table second_supervisor
(
        StaffID varchar(10) NOT NULL,
        StudentID       varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    INDEX (StudentID),
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
)
TYPE = InnoDB;

create table resides
(
        Dept    varchar(10) NOT NULL,
        AID     varchar(10) NOT NULL,
    PRIMARY KEY (Dept,AID),
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE,
```

```
    INDEX (AID),
    FOREIGN KEY (AID) REFERENCES address (AID) ON DELETE CASCADE
)
TYPE = InnoDB;

#4. Populate tables
insert into person values ("cs0001", "William Atwood");
... ...
insert into resides values ("eg","addr003");
#   OR by load data file
load data LOCAL infile '~/data/university-person.txt' REPLACE into table person fields terminated by ','
ignore 1 lines;
... ...
```

B. University Data Model Schema (script files) for PostgreSQL

How to run:

```
pgsql/bin/dropdb pgsqldb;
pgsql/bin/createdb pgsqldb;
pgsql/bin/psql -d pgsqldb -f university-POSTGRESQL.sql;
```

university-POSTGRESQL.sql

```
/*1. Create tables */
create table person
(
    ID  varchar(10) NOT NULL,
    Name        varchar(50),
        PRIMARY KEY (ID)
);
create table staff
(
        ID      varchar(10) NOT NULL,
        Salary  int,
        PRIMARY KEY (ID),
        FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
);
create table student
(
        ID      varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
);
create table tutor
(
        ID      varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
);
create table visiting_staff
(
        ID      varchar(10) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE
);
```

```
create table dept
(
        No      varchar(10) NOT NULL,
        Name    varchar(50),
    PRIMARY KEY (No)
);
create table course
(
        Code    varchar(10) NOT NULL,
        Title   varchar(50),
        Credit  int,
    PRIMARY KEY (Code)
);
create table address
(
        AID     varchar(10) NOT NULL,
        Street  varchar(50),
        District varchar(20),
        City    varchar(20),
    PRIMARY KEY (AID)
);
create table works_in
(
        ID      varchar(10) NOT NULL,
        Dept    varchar(10) NOT NULL,
        PRIMARY KEY (ID,Dept),
        FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
        FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE
);
create table teaches
(
        ID      varchar(10) NOT NULL,
        Course  varchar(10) NOT NULL,
    PRIMARY KEY (ID,Course),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
);
create table majors_in
(
        ID      varchar(10) NOT NULL,
        Dept    varchar(10) NOT NULL,
    PRIMARY KEY (ID,Dept),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE

);
create table supervises
(
        StaffID  varchar(10) NOT NULL,
        StudentID         varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
);
create table takes
(
```

```
        ID       varchar(10) NOT NULL,
        Course   varchar(10) NOT NULL,
    PRIMARY KEY (ID,Course),
    FOREIGN KEY (ID) REFERENCES student (ID) ON DELETE CASCADE,
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
);
create table run_by
(
        Course   varchar(10) NOT NULL,
        Dept     varchar(10) NOT NULL,
    PRIMARY KEY (Course,Dept),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE,
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE
);
create table prerequisites
(
        Course   varchar(10) NOT NULL,
        PreCourse        varchar(10) NOT NULL,
    PRIMARY KEY (Course,PreCourse),
    FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE,
    FOREIGN KEY (Precourse) REFERENCES course (Code) ON DELETE CASCADE
);
create table assessment
(
        Course   varchar(10) NOT NULL,
        AssName          varchar(10) NOT NULL,
        Percent  decimal(3,2),
    PRIMARY KEY (Course, AssName),
        FOREIGN KEY (Course) REFERENCES course (Code) ON DELETE CASCADE
);
create table lives_in
(
        ID       varchar(10) NOT NULL,
        AID      varchar(10) NOT NULL,
    PRIMARY KEY (ID,AID),
    FOREIGN KEY (ID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (AID) REFERENCES address (AID) ON DELETE CASCADE
);
create table first_supervisor
(
        StaffID  varchar(10) NOT NULL,
        StudentID        varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
);
create table second_supervisor
(
        StaffID  varchar(10) NOT NULL,
        StudentID        varchar(10) NOT NULL,
    PRIMARY KEY (StaffID,StudentID),
    FOREIGN KEY (StaffID) REFERENCES person (ID) ON DELETE CASCADE,
    FOREIGN KEY (StudentID) REFERENCES student (ID) ON DELETE CASCADE
);
create table resides
(
```

```
        Dept     varchar(10) NOT NULL,
        AID      varchar(10) NOT NULL,
    PRIMARY KEY (Dept,AID),
    FOREIGN KEY (Dept) REFERENCES dept (No) ON DELETE CASCADE,
    FOREIGN KEY (AID) REFERENCES address (AID) ON DELETE CASCADE
);

/*2. Populate tables */
insert into person values ('cs0001', 'William Atwood');
... ...
insert into resides values ('eg','addr003');

/*3. Create User: */
drop user wang;
create user wang password 'guang';

/*4. Grant-Revoke privileges */
grant all on person,staff,student,tutor,visiting_staff to wang;
grant all on dept,course,address,works_in,teaches,majors_in to wang;
grant all on takes,run_by,prerequisites,assessment,lives_in to wang;
grant all on supervises,takes,first_supervisor,second_supervisor,resides to wang;
```