INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

SQL2Java

SQL access to a Java based Database System

by

Chris Lambert B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

April 28, 1998

© Copyright

1998, Chris Lambert

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

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.

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 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.

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 autrement reproduits sans son autorisation.

0-612-32389-7

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

SQL2Java: SQL access to a Java based Database System

submitted by
Chris Lambert, B.Sc.
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

April 28, 1998

Abstract

The database field is constantly changing. Advances toward data representations closer to reality are steadily being made. Adoption of object-oriented concepts into the database field has allowed the creation of database schemata which incorporates data and behaviour and which contain complex object relationships.

SQL2Java is a portable Database Management System (DBMS), designed for personal use. SQL2Java's Data Definition Language (DDL) permits inheritance, which enables reuse and specialization, and it allows for the definition of custom methods. SQL2Java's Data Manipulation Language (DML) is based upon the relational query language standard SQL92. SQL2Java is written in Java thereby taking advantage of OO concepts in its implementation and enabling application portability. SQL2Java ensures operation atomicity through the use of a log file which captures file manipulation information before file manipulation begins. Operations whose file manipulation has begun but are not completed are recovered upon start up. SQL2Java's primary advantage over other DBMSs and is that it is a small portable DBMS with SQL access built in.

Acknowledgments

Special thanks is due to my wife Tammy. Completion of this thesis would not have been possible without her support and understanding throughout its preparation.

I would also like to thank Prof. Ekow Otoo for his guidance during the design and implementation of SQL2Java and his assistance during the preparation of this thesis.

I would also like to acknowledge the support of Prof. John Oommen and Marlene Wilson, of the School of Computer Science at Carleton University, for their support during my graduate career.

Table of Contents

ABSTRACT	III
ACKNOWLEDGMENTS	IV
TABLE OF CONTENTS	V
LIST OF TABLES	VII
LIST OF FIGURES	VIII
ABBREVIATIONS	IX
1. INTRODUCTION	1
1.1 THE RELATIONAL DATA MODEL	4
1.2 THE OBJECT-ORIENTED DATA MODEL	5
1.3 OBJECT RELATIONAL DATA MODEL	6
1.4 MOTIVATION	7
1.5 SQL2JAVA FEATURES	11
1.6 CONTRIBUTION OF SQL2JAVA	13
1.7 THESIS OUTLINE	13
2. LITERATURE AND INDUSTRY SURVEY	15
2.1 PERSISTENCE	17
2.1.1 <i>O₂</i>	19
2.1.2 <i>GemStone</i>	20
2.1.3 <i>Illustra</i>	20
2.1.4 <i>PJava</i>	21
2.1.5 <i>Comparison with SQL2Java</i>	22
2.2 QUERY LANGUAGES	23
2.2.1 <i>SQL</i>	23
2.2.2 <i>ODL and OQL</i>	26
2.2.3 <i>Comparison with SQL2Java</i>	28
3. BASIC WORK OUTLINE	30
3.1 STANDARDS BASED	35
3.2 CODING LANGUAGE	36
3.3 APPLICATION PORTABILITY	37
3.4 DATA DEFINITION	38
3.4.1 <i>Meta Data Management</i>	40
3.4.2 <i>Intermediary User_Objects Class</i>	42
3.5 DATA MANIPULATION	43
3.5.1 <i>Data Selection (SQL92-like)</i>	45
3.5.2 <i>Grouping (Having)</i>	50
3.5.3 <i>Sorting (ORDER BY)</i>	52
3.5.4 <i>Functions</i>	53
3.5.5 <i>Operation Atomicity</i>	56
3.6 PERSISTENCE	58
3.6.1 <i>Data File Reorganization</i>	59

4. DESIGN AND IMPLEMENTATION DETAILS	61
4.1 INTERMEDIARY <i>USER_OBJECTS</i> CLASS	61
4.1.1 <i>How the intermediary class is built</i>	63
4.1.2 <i>Intermediary Class Contents</i>	63
4.2 CLASS SETS AND QUERY PROCESSING	67
4.2.1 <i>Rules for determining Class Sets</i>	68
4.2.2 <i>Class Set Processing</i>	70
4.3 PERSISTENCE	72
4.3.1 <i>Application Start Up</i>	74
4.3.2 <i>Operations on Data Files</i>	76
4.3.3 <i>Data File Reorganization</i>	78
4.4 OPERATION ATOMICITY	81
5. APPLICATIONS AND EXTENSIONS	84
5.1 EXAMPLE APPLICATIONS	84
5.1.1 <i>Teaching Tool</i>	85
5.1.2 <i>Personal DBMS</i>	85
5.2 IMPLEMENTING A DATABASE USING SQL2JAVA	85
5.2.1 <i>Installation and Startup</i>	86
5.2.2 <i>Creating a Database Scheme</i>	86
5.2.3 <i>Startup and Use of SQL2Java's DML</i>	89
5.2.4 <i>Operation Recovery following a system crash</i>	90
5.3 EXTENSIONS	91
5.3.1 <i>Indexing Mechanism</i>	91
5.3.2 <i>Backup and Recovery Mechanism</i>	92
5.3.3 <i>Multi-user Capability, Locking Mechanism</i>	92
5.3.4 <i>Utilities for building custom interfaces and reports</i>	93
5.3.5 <i>Data File Packaging Utility</i>	94
5.3.6 <i>Network Access to Data Files</i>	95
5.4 COMPLIMENTARY TECHNOLOGIES	96
5.4.1 <i>Interfaces to other database systems and data sources</i>	96
5.4.2 <i>A Persistence Mechanism built into Java</i>	96
6. CONCLUSION	98
6.1 FUTURE WORK	99
REFERENCES	103
APPENDIX A: SQL2JAVA DML BNF SYNTAX DIAGRAMS	109
APPENDIX B: <i>USER_OBJECTS</i> SAMPLE CLASS DEFINITION	128
APPENDIX C: META CLASS DEFINITIONS	133

List of Tables

Table 3-1: SQL2Java Number Functions	54
Table 3-2: SQL2Java Character Functions	55
Table 3-3: Other SQL2Java Functions	55
Table 3-4: SQL2Java Function Call examples	56
Table 4-1: Sample queries and the Class Sets they produce	70

List of Figures

Figure 3-1: Entity Relationship Diagram for University Database.....	33
Figure 3-2: Java class definition for Taught_By class.....	34
Figure 3-3: <i>Setup</i> Class Instance Attribute Listing for Pentium 133 running Windows 95.....	38
Figure 3-4: Extended Entity Relationship (EER) Diagram for SQL2Java Metadata Classes.....	41
Figure 3-5: Sample query illustrating use of GROUP BY and HAVING	51
Figure 3-6: Sample query illustrating use of ORDER BY clause.....	53
Figure 4-1: Intermediary class attributes and methods	64
Figure 4-2: Data file for the <i>Student</i> user class.....	74
Figure 4-3: Data file for the <i>MetaClass</i> meta data class	74
Figure 4-4: Sample data files before and after Reorganization.....	79
Figure 5-1: Sample SQL2Java DDL Session.....	88
Figure 5-2: Sample SQL2Java DML session.....	90
Figure A-6-1: BNF Symbols	110

Abbreviations

2D	Two Dimensional
3D	Three Dimensional
ADT	Abstract Data Type
ANSI	American National Standards Institute
API	Application Programming Interface
ASC	Ascending
AWT	Abstract Window Toolkit
BLOB	Binary Large Object
CLOB	Character Large Object
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DDL	Data Definition Language
DESC	Descending
DML	Data Manipulation Language
EER	Extended Entity Relationship Diagram
ER	Entity Relationship Diagram
GUI	Graphical User Interface
IDL	Interface Definition Language
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JSQL	Java Structured Query Language
JVM	Java Virtual Machine
MCC	Microelectronics and Computer Technology Corporation
ODBC	Open Database Connectivity
ODL	Object Definition Language

ODMG	Object Data Management Group
OMG	Object Management Group
OO	Object-Oriented
OODBMS	Object-Oriented Database Management System
OOPL	Object-Oriented Programming Language
OQL	Object Query Language
ORDBMS	Object Relational Database Management System
RDBMS	Relational Database Management System
SQL	Structured Query Language
WWW	World Wide Web

1. Introduction

Database Management Systems (DBMS) have become the accepted means of storing and accessing much of the electronic data in the world today. DBMSs have made data storage and access more accurate and less time consuming. Their use has freed organizations, commercial and otherwise, from tedious record keeping which is typically not within their line of expertise. The evolution of DBMSs culminated in the development of the Relational Model and more recently the Object-Oriented (OO) Model. The Relational Model is the most popular approach being used today. From mainframe hierarchical and network systems, to the current client-server and distributed Relational and OO systems [Kim90], this evolution has been anything but smooth. Though the path has not always been clear, the benefits produced by this evolution have been.

This evolution continued with the introduction of Object-Oriented Database Management Systems (OODBMS) into the database field. The degree of integration of the OO Data Model with the current technologies spans the spectrum, from full OODBMSs, based completely on the OO Model (ONTOS, O2, ObjectStore), to Object Relational Database Management Systems (ORDBMS), where the OO approach is blended with the well recognized and accepted Relational Model (UniSQL, Illustra).

I believe the next major step in database system evolution will be the hybrid approach between the Relational Model and the OO Model. Each model has its strengths which should be harnessed. The significant concepts of the OO Model, with respect to the

database field, are Encapsulation, Inheritance and Polymorphism. Use of these concepts allows for the creation of rich data classes whose behaviour is inseparable from their data. A significant strength of the Relational Model is the standard interface language Structured Query Language (SQL). The SQL interface provides a non-procedural means of defining, creating and manipulating data. The SQL language, like a data model, is broken into two parts: the Data Definition Language (DDL) where the structure of the data is defined and the Data Manipulation Language (DML), where the data is accessed and manipulated. This interface language has been prevalent in the database industry since being accepted by the leading Relational Database Management System (RDBMS) vendors.

The recent trend in the RDBMS industry has been to partially or fully integrate or support OO concepts within database products. Informix's plans, for example, are to integrate their Illustra ORDBMS with the Informix RDBMS to produce a Universal Server [III97] [IIM97]. Oracle's newest database server, Oracle8, with the Objects Option, builds on the Oracle7 RDBMS by introducing OO capabilities [Ora98] [OOO98] [Bob97]. ORDBMS vendors have already produced database systems combining the OO and Relational Models. UniSQL, for example, has implemented the yet unapproved query language standard SQL3, which is rooted in the SQL92 standard but which has been extended to incorporate OO concepts [Kim97]. SQL3 is the next version of SQL which the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI) are now considering. SQL92 is the current approved version which is being used by most RDBMS vendors.

Similarly, Object-Oriented Programming Language (OOPL) vendors, have extended their languages to access and manipulate data stored in commercial DBMSs. Sun Microsystems, Inc., for example, has produced standards for Java Database Connectivity (JDBC) [Pat97] and Java Structured Query Language (JSQL) [JSQ97] to provide interfaces between the base Java OOPL and commercial DBMSs. At the time work started on this thesis, JSQL had not yet been defined and could not be used.

For this thesis I have built a DBMS which integrates the approaches taken by both the major DBMS vendors and the OOPL vendors. This system utilizes OO concepts in the design and implementation of the database engine. This utilization allows for the definition of user classes and the storage of user data. To access the user data, this system has a SQL-like interface based upon the current SQL standard SQL92. In keeping with this integrated approach, the DBMS has been named SQL2Java.

SQL2Java cannot be classified as an object-oriented database system since it does not meet a number of the requirements outlined in the paper entitled “The Object-Oriented Database System Manifesto” which provides a definition against which database systems claiming to be object-oriented are compared [KNN90]. SQL2Java, though, does provide a number of the mandatory features of an OODBMS. These features include encapsulation, classes, inheritance, persistence and an ad hoc query facility. Other writings have also attempted to define the required characteristics of an OODBMS [DDB91] [AG92]. In contrast, Andleigh and Gretzinger, in “Distributed Object-Oriented

Data-Systems Design” have a similar definition as that provided in the manifesto referred to above but which includes Multiple Inheritance and a versioning capability which the manifesto lists as optional items. Versioning refers to the ability to store various copies of objects from different times in their life-cycle and is a standard feature of object-oriented systems [Hug91]. Having the ability to store and access various versions of objects is especially useful in design environments, which often use object-oriented technologies.

A major difficulty of the integration of the OO paradigm into the database field has always been ensuring the persistence of data. Typically programs written in an OOP language do not concern themselves with retaining data after the program has stopped or the user session ends inadvertently. Data persistence though is necessary for a database system.

1.1 The Relational Data Model

The Relational Model was introduced in the 1970s and early 1980s and is a model based upon data being stored in well defined structures called tables. These tables are analogous to mathematical relations since each row of a table represents a relationship between the elements of the row. Operations on the Relational Data Model are carried out using the Relational Algebra [AG92]. Relational query languages are based on either the Relational Algebra or the Relational Calculus [KS91].

Connections between relational tables are not explicit, they are determined by the actual data contained within the components of the tables which are called columns. For the

most part, this allows each table to be managed separately and in isolation from the management of other tables.

A standard method of accessing and manipulating data in a RDBMS is through a standard language agreed upon by the major RDBMS vendors and standards bodies including the ISO and ANSI called Structured Query Language (SQL). The current standard being used by most RDBMS vendors was defined in 1992 and is called SQL92. This standard language is non-procedural and is made up of two components. The DDL component allows for the creation, alteration and deletion of database tables. The DML component allows for insertion, deletion, manipulation and retrieval of data stored in the database tables. SQL92 is a later version of the first SQL standard, simply called SQL, which was approved by the same standards bodies in 1986 and 1987. SQL is often used as a generic name to refer to each of the SQL standards.

1.2 The Object-Oriented Data Model

The object-oriented paradigm and the resultant OOPs originated in the 1960s with Simula in 1967. Simula extended ALGOL60 with its class concept [AG92] [Bae80]. Many more have followed including Eiffel, Smalltalk, C++ and Java since then. Their use in Database Management Systems has been limited due to the lack of persistence. The OO paradigm, though, does have characteristics which were very attractive to builders of database systems including the following three. Encapsulation: An object's data and behaviour are hidden within the object. To access or affect the object's state you

must perform operations upon it. These operations have well defined interfaces which are all that is visible to the public. Inheritance: The ability to specialize a class as a subclass of a superclass. Polymorphism: The use of identical methods in the interfaces of different objects that respond according to the semantics of the object.

A number of mature OODBMS products currently exist including O₂, ObjectStore and GemStone. Their producers and other companies involved in this industry have formed a consortium called the Object Database Management Group (ODMG) which has established standards for the development of OODBMSs [ODM97] [SO97]. Though some of the standards borrow from efforts put forward by other standards bodies for Relational databases, their data model is the OO Data Model, not the Relational Data Model. Use of the OO Data Model implies not only the OO concepts listed above, but also that connections between objects are stored directly within the objects. Therefore, data access can potentially be much faster since parsing is reduced. Implementation of the OO concepts given above allow for data representations which are much closer to reality.

1.3 Object Relational Data Model

An integration of the concepts from both the Relational Model and the OO Data Model has been utilized to produce database systems which bridge the gap between the Relational and OO worlds.

These database systems provide an extended Structured Query Language (SQL) interface which allow users, if they wish, to use only the basic Relational capabilities of the

interface or to also use the available OO capabilities. The OO capabilities typically include the ability to create specialized subclasses from existing superclasses (inheritance), the ability to define custom data types or classes with custom methods built in (encapsulation) and the ability to have direct references to other objects within existing objects. SQL3, the yet unapproved version of SQL, defines such an interface. Though this standard is yet to be approved it has already been used by UniSQL and Illustra (now a subsidiary of Informix) for their latest database engines.

1.4 Motivation

The marriage of the Relational Model and the OO Data Model, which would allow users to create and access database objects using techniques with which they are already comfortable, is the motivation for this thesis.

The most popular data model over the past decade or so has been the Relational Model. This trend is changing. Organizations require the ability to store complex data easily without losing any of the complex details and relationships inherent in that data. Data such as financial data, design data and World Wide Web (WWW) application data call for fast access and efficient storage techniques. These techniques are represented better in the OO Model.

Further, a major shift is happening in the programming world toward the OO paradigm. Some describe OO programming as a more natural way of programming since it allows

for data representation which is much closer to reality [Jac93]. This impression is due to the behaviour of objects being programmed directly into the classes and all but the interface being guarded from view to the outside world. Classes programmed in this way can be reusable and expandable, which make them very attractive to businesses who hold major investments in their in-house or commercial software resources.

There are a number of good OOPLs on the market, but like other 3rd and 4th generation languages they typically do not implicitly support data persistence nor do they provide a standardized data access language. These features belong to the database realm.

A relatively new OOPL called Java, which was created by Sun Microsystems Inc., has many attractive features [PSP97]. Perhaps the most valuable one is that Java programs are portable between machines that run a Java Virtual Machine (JVM). The JVM is a piece of software specific to a platform but which will accept and run any Java program. This is a major step forward since the end user applications are truly portable. Another valuable advantage of Java is that users can presently get the required software for free allowing for easier distribution and education in the language.

Sun Microsystems, Inc. has recently added a rudimentary persistence mechanism to the base Java language [OS98]. This functionality, called Object Serialization, has been added to the Input-Output (IO) class and allows the user to store object data to an output stream, typically a disk file, to be read back at a later time to reinstantiate the object. This functionality was introduced with release 1.1 of the Java language and was not available

when design and implementation began on the persistence mechanism used with SQL2Java.

While users would like a more natural and flexible model for representing their data, they do not wish to lose the standards based access tools which they have been using to access their relational data. Though vendors have supplemented their query languages, Oracle's SQL*Plus, for example, has many add-on features, their query languages are typically based on one of the ISO SQL standards. This movement has led to new standards being delivered or under development which are upwardly compatible with ISO's latest SQL version, SQL92. The ODMG have already produced, as part of their standards package, a query language called the Object Query Language (OQL), which is similar to SQL but which is exclusively designed for use with OODBMSs.

Upon reviewing the existing commercial offerings, none were found which were developed exclusively within an OOPL and which provided for data persistence within that environment. As will be described in the Literature and Industry Survey chapter, the PJava project is investigating such a facility for the Java OOPL but it is not yet commercially available. Likewise, the Object Serialization component of the IO class in Java's base language was not yet available when development of SQL2Java began. Not only does SQL2Java allow for persistent data storage, but it also provides a SQL-like interface to access and manipulate the data.

In developing the prototype system for this thesis my objective was to show that it is possible to build a database system based upon the OO Model, with a simple persistence scheme and a standardized DML interface. This system does not contain all the utilities of a production DBMS, nor does it fully realize the OO Data Model but it lays the groundwork for such a system. My development platform was the Java OOPL which also enables portability of SQL2Java.

SQL2Java is a functioning prototype system but without all the necessary components of a commercial database system nor the performance characteristics of a commercial DBMS. The DDL portion of SQL2Java allows for single inheritance, encapsulation, and polymorphism. The DDL, though, does not allow for attributes which are object references nor for attributes which are sets or lists of attributes or objects. The DML portion of SQL2Java is based upon the ISO SQL92 standard. As such it does not implement the OO concepts defined in the not yet approved ISO SQL3 standard. SQL2Java's DML component, though not complete, implements as much of the SQL92 standard as was reasonably possible. This implementation allows for queries requiring list processing, subqueries, grouping and use of the HAVING clause. It also has an extensive set of regular and group functions and allows for result sorting.

Since Java only allows for single inheritance, and SQL2Java database schemata are composed only of Java classes, SQL2Java retains this same convention. Though multiple inheritance would enable the representation of more complex relationships it is also

potentially very confusing since more than one parent class could use common names for attributes and methods.

Within the DDL, SQL2Java also allows for the definition of methods. Though some methods such as get, set, and constructor methods are created by default, the user is able to create custom methods thereby defining class behaviour. Unfortunately, SQL2Java does not have a facility to execute these methods through its DML since it is currently based exclusively upon the SQL92 standard. A future extension to the DML would be to provide such a facility. A strategy to deal with unwanted side effects of these methods must first be defined.

1.5 SQL2Java Features

SQL2Java is a portable database system based upon both the Relational Model and various OO concepts. It is made up of two separate Java applications. The first is used to define the database scheme while the second is used to access and manipulate the user objects. Both the DDL and DML components have Graphical User Interfaces (GUI) with an area for the user to enter input and a separate area where the results are presented.

The SQL2Java DDL allows the user to create, alter or delete user classes. The DDL operates by presenting a series of questions to the user. The answers to these questions supply the information required to perform the DDL request. To ensure data persistence SQL2Java permanently stores its meta and user data in a series of files which either come standard with each system or which are created by the DDL portion of SQL2Java. These

files are then read and manipulated by the DML portion of SQL2Java. The files which come standard with each system are those which hold the metadata or system catalog data for the system.

The SQL2Java DML permits access to user objects. The DML allows the user to submit standard SQL92 requests and to receive the results in return. As SQL2Java's DML component is based upon the SQL92 standard, it is able to perform the vast majority of operations available in the SQL92 standard. This includes the SELECT, INSERT, UPDATE and DELETE statements. A rich set of standard and group functions are available as well as grouping, with or without the HAVING clause, aliases, list processing, subqueries and sorting. As in the DDL, the user is able to get a description of a user class and syntax errors are reported as soon as they are detected.

To ensure the atomicity of operations SQL2Java has a two stage commit mechanism, implemented using a log file. The first stage writes all the necessary data to complete an operation to the log file in case a crash were to occur after modifications to the data files has begun. After an operation has finished the second stage, which is writing its new or updated data from memory to the data files, the operation is removed from the log. This provides SQL2Java with a simple operation recovery mechanism.

1.6 Contribution of SQL2Java

The main contribution of SQL2Java is the design and implementation of a simple DBMS for Java classes that integrates OO concepts with the Relational Database concepts. This integration takes the best from both models to produce a database system which is built in the Java OOPL but which provides a query mechanism based upon the proven and recognized SQL92 standard. SQL2Java also allows for class inheritance and the definition of custom methods. Being that SQL2Java is developed in Java, it is portable and will execute on any machine running a Java Virtual Machine.

1.7 Thesis Outline

This thesis presents, in the next chapter, a literature and industry survey of various OODBMSs and ORDBMSs. It contains a discussion of a number of projects and products with similarities to SQL2Java. Within this discussion these projects and products are broken down into various categories with a focus on data persistence and interface languages. Chapter 3 outlines the design of SQL2Java and the reasons why particular strategies were chosen. The subsequent chapter, chapter 4, describes the Java implementation of SQL2Java. This chapter gives the details of how the code has been structured and the process flow within SQL2Java. Chapter 5 itemizes ways in which SQL2Java can be utilized in its current state and how it can be extended to further its effectiveness. The conclusion is presented in chapter 6 where future work is discussed. Appendix A lists the SQL92 grammar which the DML of SQL2Java is able to interpret.

Appendix B contains a sample class definition for the *User_Objects* class. Appendix C lists the class definitions for the central meta data classes.

2. Literature and Industry Survey

In order for SQL2Java to qualify as a DBMS, it must meet two important requirements [KS91]. The first, and most important, is that it requires a scheme to ensure the persistence of data. The second is that it has to provide a means of defining and accessing the data. All commercial DBMSs on the market today provide these two properties and many more.

The most basic requirement of a DBMS is that the data that it stores must be persistent. Persistence means that the data must be stored in such a way that even after the DBMS shuts down, intentionally or otherwise, the data still exists. The typical method chosen by most DBMS implementers is to store the data on disk. When data is required by a user the data is brought into memory and kept there until that memory is required to hold another piece of data. This swapping technique is necessary since in typical commercial databases the amount of data is too large for it all to fit into the computer's memory.

In order for the DBMS to be usable it must also provide a mechanism for users to create database schemata and to access and manipulate data. A number of query language standards have been agreed upon by standards' bodies and consortia. Commercial DBMS companies tend to adhere to these standards in order to ensure a higher degree of interoperability between their systems and to gain greater acceptance in the user community.

ISO and ANSI have lead the way with the SQL standard agreed upon in 1986 and then later with a revised version called SQL92 which was agreed upon in 1992. They are currently working on the next version called SQL3 which database vendors hope will be approved in the near future. This new standard will include features of benefit to vendors who are moving to incorporate object-oriented techniques into their products. A number of companies including UniSQL and Illustra, have already adopted this latest unapproved version of SQL into their latest database release [Kim97].

The ODMG, a consortium of OODBMS vendors has also defined standards. These are specifically for object-oriented database systems and include a SQL-like query language called OQL which takes into account OO principles. This group has also defined standards for other OO languages wishing to interface with their database systems called language bindings.

In order for a DBMS to be commercially viable it must also provide many other features which have not been included in SQL2Java. Features such as Security, Locking Mechanisms, and Backup and Recovery Procedures are necessary for any mission critical database system. Since this thesis is primarily concerned with the DDL and DML database system components, techniques utilized by commercial OODBMSs and ORDBMSs vendors to handle these other features will not be discussed.

The remainder of this chapter discusses previous work in the area under the headings of Persistence and Query Languages. The first section, which deals with Persistence, describes how various vendors and researchers have implemented persistence. The second section, which deals with Query Languages, discussed the two main query language standards and how vendors have implemented these standards.

2.1 Persistence

Persistence is a necessary feature of database systems. In "Building an Object-Oriented Database System - The Story of O₂" persistence is described as "the ability of data to survive the execution of a process, and to be eventually reusable in another process" [BDK92]. There are different approaches to how persistence is implemented in the DBMS but all eventually result in storing data to permanent storage which, in most cases, is magnetic disk. In most OODBMSs there are two classifications of objects. An object can be either volatile or persistent. If it is volatile, then it is never written to permanent storage and exists only as long as the process that created it. On the other hand, objects can be persistent, meaning that steps are taken to ensure that the object's data is permanently stored for as long as the object exists within the DBMS [Hug91].

For these objects, there are different approaches to defining which are to be persistent and which are to be volatile. Two of the more popular approaches are:

1. Typing - Classes can be declared as being persistent. If an object belongs to a persistent class then it is also persistent.
2. Reachability - If an object is connected, directly or indirectly to a persistent object, which are known as persistent roots, then it is also persistent.

The approach taken in SQL2Java is simpler than these. All objects within SQL2Java are implicitly persistent. This approach stems from the Relational Model where it is assumed that data inserted into a database system is to be stored permanently. OODBMSs and ORDBMSs, though, extend the scope of the database system to allow for the storage of more complex data structures and to more accurately represent real life applications. As a result, this new generation of DBMSs may be used to hold application data which is required only during the execution of a particular process. Therefore, volatile objects are quite useful.

Objects are typically identified with an identifier that is unique throughout the entire system which is called the object identifier or OID. To access an object most OO products use a separate functional layer called an Object Manager. The Object Manager either has a table of the correspondences between the OIDs and the physical memory locations of the objects or has some other scheme whereby it is able to deduce the physical location of an object from the object's OID. Described below are a number of systems which have implemented persistence. I also present PJava, a project underway in the Department of Computing Science at the University of Glasgow. It is investigating a

mechanism to provide for persistence within the Java programming language. The latest release of the Java OOPL now has a mechanism to make objects persistent. This mechanism is called Object Serialization and is part of the IO class of the base Java language. Using this mechanism object data can be stored to and read from data files. SQL2Java does not utilize this facility.

2.1.1 O_2

O_2 is an OODBMS developed within Altair, a French research consortium between the years 1986 and 1991 [BDK92]. O_2 is a member of ODMG and was the first company to produce an OODBMS based on the ODMG-93 standard.

O_2 uses the Reachability approach to implementing persistence. To make an object or value persistent you must name it. Once it is named, the Object Manager stores it, and any objects reachable from it, to permanent storage. The main difference between O_2 and other OODBMSs is that it uses physical object identifiers where as most other systems use logical object identifiers. Physical object identifiers have two purposes in O_2 . The first purpose is that it serves as the object's unique identifier and secondly it identifies exactly where to locate the object on disk. This technique improves O_2 's access performance since there is one less table lookup required. It also removes the storage requirement for the OID table and the table management overhead.

2.1.2 GemStone

GemStone Systems Inc., another member of the ODMG, markets a three tier OODBMS called GemStone. GemStone provides the Object Server which is the middle tier, the top tier being the client applications and the bottom tier being the data repository. GemStone has recently released GemStone/J which is a Java Application Server. Traditionally, GemStone servers were based on Smalltalk. They also provide gateways to leading RDBMS.

GemStone is similar to O₂ in that it uses the Reachability approach to implement persistence. The Object Manager, though, is more in line with standard object-oriented products in that it uses logical object identifiers. To maintain these OIDs GemStone maintains an object table which is organized as a B-tree.

2.1.3 Illustra

The Illustra ORDBMS was developed by Illustra Information Technologies, Inc. and has truly been an evolutionary product with its roots firmly planted in the Relational Model. The most recent version of Illustra introduced a concept which is called DataBlades. DataBlades are add-ons which define how to handle specialized Abstract Data Types such as audio, video or textual information. Illustra utilizes a modified version of SQL which is based on the SQL3 standard that is yet to be approved.

Illustra is included in this survey to illustrate the approach taken by most ORDBMS vendors when it comes to persistence. Since their products are typically a fusion of existing relational technology and new object-oriented technology, they retain many of the techniques which were successful in their RDBMS products. With the exception of UniSQL, which is described in section 2.2.1.1, all other major ORDBMSs have evolved from RDBMSs. The approach taken by Illustra for persistence is that data requests are passed to a Storage Manager. If the data being accessed is of an abstract type, then the Storage Manager uses code contained within the DataBlade corresponding to the abstract data type being accessed to determine how to manage the data. The Storage Manager then gives up much of its autonomy with respect to data storage to the appropriate DataBlade [TPD97].

2.1.4 PJava

PJava is a research project being undertaken by the Department of Computing Science at the University of Glasgow and is investigating the feasibility of providing orthogonal persistence for the Java programming language [Spe97]. The researchers in this project based their implementation on the following three central principles:

1. Orthogonal Persistence - all data elements are to have equal access to persistence regardless of their type
2. Transitive Persistence - object persistence is determined by the Reachability approach described above

3. Persistence Independence - whether a program is operating upon data which is transient or persistent must be transparent

The researchers have produced a prototype system which has been made available for evaluation and research purposes. In an introductory WWW document presented by the research group, it is stated "Collections of objects of any type may be preserved and exchanged between programs without loss of safety, without any change in their semantics and without any complex programming. Executing programs can operate concurrently and transactionally on these objects supported by recovery mechanisms" [Spe97].

No modification to the Java language was required for the implementation of the PJava prototype, the Java core classes or the Java compiler. This was accomplished by the creation of a new API and by making modifications to the JVM.

2.1.5 Comparison with SQL2Java

As with Illustra, persistence in SQL2Java is implicit. All data stored to the database, either meta or user objects, are created both in memory, as Java objects, and are stored to the appropriate data file. No objects stored in SQL2Java are volatile. In essence, SQL2Java is an implementation of a system similar to that outlined for the PJava project. Unlike PJava, which allows for transient objects, SQL2Java does not give the user the choice. All data stored in SQL2Java is persistent.

2.2 Query Languages

A well designed query language based on recognized standards is a necessity for a DBMS to be acceptable to database designers and users. Unless a DBMS's query language is based upon a recognized standard, the learning curve for users of the language will potentially be too steep to ensure the success of the language and consequently the DBMS. Further, interoperability between the DBMS and other DBMSs, which are based upon one of the recognized standards, will be compromised. For these reasons the majority of DBMS vendors choose to adopt one of the standard query languages which are described below.

2.2.1 *SQL*

Structured Query Language (SQL) is a standard query language defined by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI). In 1992 ISO published its first major revision to SQL which is commonly called SQL92. Revisions were to either correct mistakes or ambiguities in the previous standard or to add new features such as new data types and functions. For the most part, the original version is upwardly compatible with SQL92.

Both of these SQL standards are based only on the Relational Model. The next version, which is now commonly called SQL3, will also allow for object-oriented capabilities. The additions include the ability to define different data and reference types, the ability to

define constructors for these new types, the ability to create functions and procedures, and the ability to support large objects including Binary Large Objects (BLOBS) and Character Large Objects (CLOBS). Some of the types which users will be able to define will include Abstract Data Types (ADTs) which will resemble encapsulated OO classes with attributes and behaviour, row types which will be row-encapsulated lists of type value pairs, reference types which will be used to refer to other types and, finally, collection types which will allow for columns to be defined as sets and lists. There will also be subtype and subtable facilities which will allow for ADT and table hierarchies [MS97].

A number of ORDBMS vendors have already implemented the draft SQL3 standard in anticipation of its upcoming release. The first two subsections discuss the approaches taken by UniSQL and Illustra. The third subsection discusses how the JDBC and JSQL facilities may be utilized to access SQL databases and the differences between these two approaches.

2.2.1.1 UniSQL

The UniSQL ORDBMS supports the ISO SQL3 standard which has not yet been approved. UniSQL's query language, which is called SQL/X, extends SQL to incorporate object-oriented capabilities [Kim97]. SQL/X extends SQL in the following four ways:

- 1) Object References. Allow a column to be of any type including user-defined datatypes

- 2) Encapsulation. Allow the definition of methods against user-defined datatypes
- 3) Multiple Inheritance. Allow relations to be subclasses of parent classes
- 4) An attribute can be a set of values rather than just a single value

2.2.1.2 Illustra

Illustra also bases their query language on the not yet released ISO SQL3 draft standard [IDP97]. Illustra allows for the definition of user defined datatypes, the definition of operators, functions and access methods against the new datatypes and for inheritance of data and functions. It also allows for sets and lists of data. To define complex data types the DataBlade mechanism, as described in section 2.1.3, is used [ID97]. A group of related datatypes can then be packaged into a DataBlade and used in other installations. Illustra supplies a number of DataBlades, which must be purchased separately, in various areas such as 2D and 3D spatial data, World Wide Web (WWW) data and textual data.

2.2.1.3 JDBC and JSQL

Java Database Connectivity (JDBC) and Java Embedded Structured Query Language (JSQL) are two facilities which allow database access through a Java program. Both of these facilities require an external database system which has been setup to accept and respond to the requests made to them by the Java program.

The more mature of these two technologies is JDBC [Pat97]. It is based upon the Open Database Connectivity (ODBC) concept. This technique utilizes a middle tier between the Java program and the Database Server. This middle tier, the JDBC drivers and managers, submits the database request to the database and accepts the response back from the database before passing the results back to the Java program. JDBC allows for dynamic SQL calls to the database. This means that the Java program can create SQL statements to submit at runtime. The obvious cost of this type of operation is that there is no validation of the SQL statements at compile time.

JSQL, on the other hand, is more static in nature [JSQ97]. By giving up the ability to create SQL statements at runtime, JSQL is able to validate the embedded SQL at compile time. There is a pre-processor step which validates the SQL code and translates it into standard Java code which is then compiled like any normal Java program. A standard for JSQL has been developed and an effort is now underway to have it approved by ISO and ANSI as embedded SQL standards for other third generation languages have been in the past.

2.2.2 ODL and OQL

The Object Database Management Group (ODMG) is made up of a group of OODBMS vendors committed to the development of a standard for OODBMSs. The purpose of the standard, as pointed out in an overview of the ODMG standard, is "to ensure the portability of applications across different object DBMSs, protecting the user's software

investment while reducing reliance upon a single vendor, and encouraging competitive feature development" [SO97]. The most recent release of the standard is version 2.0 [ODM97].

The standard is composed of four components: the Object Model, the Object Definition Language (ODL), the Object Query Language (OQL) and language bindings for the more popular OOPLs including C++, Smalltalk and Java [SO97].

The ODMG Object Model specifies how the database can be defined. It determines the characteristics of the objects, how objects can be related and how objects can be identified and named. Objects without identifiers are called literals and are not persistent. Those with identifiers are persistent and can be shared between multiple users and applications [ODM97]. The ODL is a definition language which allows for the definition of database schemata. It is an extension of the Object Management Group's (OMG) Interface Definition Language (IDL) standard which was developed as part of the Common Object Request Broker Architecture (CORBA).

The OQL is a functional language which relies upon the ODMG Object Model and is based upon the ISO SQL standard. Though based on the Relational Model standard OQL has been extended to incorporate the object-oriented concepts of complex objects, object identity and polymorphism.

The OOPL language bindings, though slightly different for each language, allow for the definition and manipulation of persistent database objects. They also have a facility to invoke the OQL. O₂ is an original voting member of the ODMG and were instrumental in the OQL definition. O₂'s Object Query Language actually formed the basis for the ODMG OQL [OUF97].

2.2.3 *Comparison with SQL2Java*

SQL2Java's query language can be considered a hybrid of the ODL portion of the ODMG standard and the DML portion of the SQL92 standard. SQL2Java's DDL allows the user to define Java classes which are similar to tables from the Relational Model but with two key diversions. The SQL2Java DDL allows the user to inherit from parent classes and to define custom methods. Though these features are OO in nature, the SQL2Java DDL cannot be considered fully OO in that it does not allow attributes which are object references.

SQL2Java's DML on the other hand is fully relational in that it is modeled after the SQL92 standard. This facility could be extended to implement the proposed SQL3 standard or even the ODMG OQL standard, but no attempt to do that has been done to this point.

SQL2Java differs from JDBC and JSQL in that, for these facilities, external database access is required. SQL2Java's database system is built in. The type of SQL access

available in both JDBC and JSQL is quite similar in that both are based primarily upon the relational SQL92 standard.

3. Basic Work Outline

SQL2Java is a prototype DBMS completely designed and programmed from the ground up in Java. The intention in building SQL2Java was to demonstrate that the integration of relational database and OOPL technologies could result in a product which satisfies changing DBMS requirements while retaining proven Relational Model concepts. Specifically, SQL2Java takes advantage of the ability of a class to inherit attributes and methods from a parent class. Further, SQL2Java is a portable product being that it is programmed in the Java OOPL.

SQL2Java is made up of two independent Java applications: one Java application for defining the database scheme and one Java application for accessing and manipulating the database objects. Together these two independent components make up a database system whose data definition component is a hybrid of the Relational Model and OO concepts and whose data manipulation component is based solely on the SQL92 standard. SQL2Java is intended to be used as an end product and not to be utilized as a utility or library for Java applications or applets.

Like all DBMSs, SQL2Java has a DDL and a DML. The DDL allows users to define Java classes which are analogous to tables in the Relational Model. The major restriction on this definition is that attributes must be of one of the basic data types which SQL2Java currently supports. These types are integer, double(float), and String. An obvious

extension would be to include more of the basic data types such as date. Though this could be done in a future version and would make SQL2Java more usable, the intention in developing the current prototype version was to create a functional, relatively complete, database system. Another obvious extension, which is part of the proposed SQL3 standard, would be to allow attributes to be references to other objects. This would involve modifying the design of SQL2Java's DML to integrate OO concepts. SQL2Java's DML is currently based exclusively upon the DML of the Relational Model.

The SQL2Java DML is based upon the Relational Model and is intended to interpret the majority of the DML portion of the SQL92 standard. It includes the select, insert, update and delete operations. It supports sorting, grouping (with and without the having clause) and queries containing subselect statements. All of the standard arithmetic, character, comparison and logical operators are available. It also has a rich set of both group (COUNT, MIN, MAX, AVG and SUM) and normal functions (SQRT, LENGTH, REPLACE, SUBSTR, etc.).

As SQL2Java is a prototype system, it does not contain all of the supporting utilities which are necessary in a commercial DBMS. Such utilities as a Backup and Recovery Strategy and an indexing mechanism are left for future development.

To better illustrate the design and implementation of SQL2Java a simple database scheme, which will be referred to often throughout this work, is given below. This simplistic Entity Relationship diagram represents a university database intended to

capture Professor, Student and Course information. This database can be used to track a university's employees, its professors, its students, the courses it offers, who is teaching a particular course in a particular semester and which students are taking a particular course in a particular semester. An inheritance relationship exists between the Person entity and the Employee and Student entities. Likewise, an inheritance relationship exists between the Employee entity and the Professor entity. A one-to-many relationship between the Professor and Course entities is given to show that one professor may teach many courses but that each course, in this database, is restricted to being taught by a single professor per semester. To capture the relationship between Professor and a particular Course offered in a particular semester, the Taught_By entity is required. A many-to-many relationship exists between the Student and Course entities showing that one student may take many courses and that a course may be taken by many students. The Taken_By entity is required to resolve the many-to-many relationship and to capture result data.

An implementation of this scheme will result in the *Person*, *Employee*, *Professor*, *Student*, *Course*, *Taught_By* and *Taken_By* Java classes. Each class containing the attributes associated with the corresponding entity in the diagram below.

Figure 3-1: Entity Relationship Diagram for University Database

Section 5.2.2 describes a typical SQL2Java DDL session in which the *Person* class from Figure 3.1 is defined. Figure 3.2 below lists the actual Java class definition file for the *Taught_By* class.

```
public class Taught_By {
    public String Name = "John Doe";
    public String Course_Num = "95.511F1";
    public String Semester = "Fall1997";
    int file_position = 0;
    Taught_By next_Taught_By = null;
    public Taught_By(String Name, String Course_Num, String Semester)
    {
        this.Name = Name;
        this.Course_Num = Course_Num;
        this.Semester = Semester;
        next_Taught_By = null;
    }
    public Taught_By() {
        this.Name = "John Doe";
        this.Course_Num = "95.511F1";
        this.Semester = "Fall1997";
        next_Taught_By = null;
    }
    public String get_Name() {
        return Name;
    }
    public String get_Course_Num() {
        return Course_Num;
    }
    public String get_Semester() {
        return Semester;
    }
    public void set_Name(String Name) {
        this.Name = Name;
    }
    public void set_Course_Num(String Course_Num) {
        this.Course_Num = Course_Num;
    }
    public void set_Semester(String Semester) {
        this.Semester = Semester;
    }
}
```

Figure 3-2: Java class definition for Taught_By class

Each Java class definition contains the attributes' definitions, the default constructors and the default access methods. In the Taught_By class there are five attributes, three of which were specified by the user and two default attributes created by SQL2Java. The Taught_By class has two default constructors. The first constructor accepts values for

each of the user-defined attributes while the second constructor accepts no values and sets all user-defined attributes to the default values specified by the user. For each user-defined attribute there is a default method to get the attribute's value and a default method to set the attribute's value. This chapter describes the design of SQL2Java and why particular strategies were chosen.

3.1 Standards Based

SQL92 is a relational query language which is relatively easy to learn and use and since most relational database vendors provide an implementation of it, interoperability and data exchange between these systems is simplified. OOPs also provide a number of attractive features which could be beneficial when integrated into a database system. One of these features is the ability to inherit attributes and behaviour from a parent class. Another feature is encapsulation. This is the characteristic that access to data and behaviour is available only through well defined interfaces.

The approach in building SQL2Java has been to integrate these two technologies where it would be beneficial and which did not take away from the inherent benefits that each provided. Following that approach, inheritance has been incorporated into the SQL2Java DDL. Through the use of the Java OOP, as my development language, OO concepts have been utilized throughout the implementation.

3.2 Coding Language

In choosing a coding language for SQL2Java there were four main criteria, as follows:

1. A language which had a structure which mapped well to the table structure of the Relational Model
2. A language which allowed for inheritance between those table-like structures
3. A language which produced an end product which was portable between the more popular computer hardware and software platforms
4. A language which provided an automatic means of storing persistent data between executions of the end program

Criterion number two limited the implementation languages to those of the object-oriented paradigm. This worked well for criterion number one, as well, since OOPs are based on the concept of classes which are quite similar to relational tables. Number four was a highly desirable feature but was not as important as application portability.

Fortunately, Sun Microsystems, Inc. produces an object-oriented programming language, which produces portable output and they provide the development toolkit and the execution environment for this language free of charge. Their Java OOP was created from scratch which means that it does not contain many of the features which are contained in other OOPs which evolved from existing languages. Java does not have pointers. Rather Java uses object references. Java supports only single inheritance instead

of the multiple inheritance, which most OOPLs provide. One of the most attractive features of Java is that Java programs can execute on any platform running what Sun calls a Java Virtual Machine (JVM). This code is the intermediary layer between a compiled Java program and the computer on which it is running. Sun provides this JVM code for most popular platforms.

Unfortunately, Java did not provide a built-in means of maintaining data persistence until after implementation began on SQL2Java's persistence mechanism. Sun has sponsored at least one project called PJava, which is investigating the feasibility of a Java extension to provide for data persistence [Spe97]. This research may have been used when developing the Object Serialization mechanism which Sun has built into the IO class of the recent release 1.1 of the Java OOPL.

3.3 Application Portability

Though SQL2Java is a portable program, due to the fact that has been implemented in the Java OOPL, there is still platform specific information which the SQL2Java system must be aware of for SQL2Java to operate correctly. SQL2Java utilizes the file system on the platform that it resides on to store meta and user data; therefore it must know exact paths to data file locations. SQL2Java must also know the name and the location of certain utilities, which it uses to delete data files and to compile new Java classes. Further, since SQL2Java must read lines from and write lines to data files it must know the correct format and length of the new-line sequence on the platform on which it is running.

This platform specific information is retained in each instance of a specific class, which must be modified and recompiled on each platform. The class is called *Setup* and its constructor sets its attributes to the values required for the platform upon which SQL2Java is running.

The following list gives the attributes of the *Setup* class and the values which were used on SQL2Java's development platform.

```
delete_command = "c:\\windows\\command\\deltree /Y "  
compile_command = "c:\\jdk1.1.4\\bin\\javac "  
class_location = "c:\\sql2java\\"  
newline_string = "\\n"  
newline_length = 2
```

Figure 3-3: *Setup* Class Instance Attribute Listing for Pentium 133 running Windows 95

3.4 Data Definition

The objective, with respect to the definition of data, was to provide a means of defining Java classes but with a number of restrictions. These restrictions prevent the definition of attributes which are of a type other than the basic Java types of integer, double or String. This is not only restrictive in OO terms but also in standard SQL terms. With respect to

the standard Java class definition, SQL2Java is restrictive in that it does not permit attributes which are references to other objects.

One significant OO feature SQL2Java does provide for is single inheritance. This feature allows the user to create a hierarchy of classes, which are richer and more specific further down the hierarchy. This is not part of the SQL92 standard but it is not specifically contrary to it either. SQL92 does provide a means of creating a table which is made up of attributes from an existing table through the use of a subselect in the table creation statement. My extension is similar but retains all attributes and methods unless explicitly overwritten.

A typical SQL2Java DDL session in which the *Person* class from the ER diagram in Figure 3.1 is defined is described in Section 5.2.2. The ER diagram in Figure 3.1 is that of the simple example being used throughout this work to illustrate the concepts and workings of SQL2Java.

SQL2Java's DDL also allows for the definition of custom methods with each class. Unfortunately, SQL2Java's DML does not provide a mechanism to allow for the execution of these methods. Two reasons for this are, first, SQL92 does not provide such a facility and, second, side-effects of the methods could cause problems. Since SQL2Java does not restrict, nor should it to attempt to restrict, code placed within the custom methods, it cannot guarantee that a method will not cause program errors.

3.4.1 Meta Data Management

Class definition data entered by users must be maintained. In SQL2Java this information is stored as objects of the following metadata classes:

- ◆ *MetaClass*
- ◆ *MetaAttribute*
- ◆ *MetaMethod*
- ◆ *MetaMethodAttribute*
- ◆ *MetaMethodLine*
- ◆ *MetaParameter*

These meta data classes are represented in the following Extended Entity Relationship (EER) diagram:

Figure 3-4: Extended Entity Relationship (EER) Diagram for SQL2Java Metadata Classes

As a user enters class definition data, two things happen. First, the information entered is used to create objects of these metadata classes. As will be explained later, this information is automatically made persistent by adding this data to data files as the objects are created. Secondly, after all the required class information is entered a class definition file is created and compiled. The DDL automatically creates default constructors for each class, default methods to retrieve an object's attribute values and

default methods to set and object's attribute values. A data file is also created to hold persistent data for the newly created user class.

These metadata classes store all of the information needed to recreate the user classes if they are corrupted for any reason. This information is also used when a user requests a description of a class or when a class file has to be recreated after a user defines an alteration.

Appendix C is a listing of the class definitions of the central meta classes. For each meta class in the listing, the class's attributes and the interface for the class's constructors and methods is given.

3.4.2 Intermediary User_Objects Class

As the SQL2Java DDL is based upon the definition of actual Java classes and since Java applications cannot access classes which do not exist when the application is compiled, it was necessary to separate the DDL module from the DML module. This does not resolve the whole problem though without modifying and recompiling the DML code after every data definition operation to include references to the newly created classes. To get around this obstacle, at the end of each DDL session SQL2Java recreates the class definition file of a SQL2Java system class whose single instance allows the DML to access all of the user classes. This class's constructor will actually instantiate all of the user objects when it is instantiated by the DML upon startup. This class also contains standard broker

methods which the DML can use to call the methods of the user classes. At this point those methods include the user class's constructors, get methods for each attribute and set methods for each attribute.

The broker methods are intermediary methods which may be called by the DML and which in return call methods of the user classes. The DML is able to call these broker methods since they have a standardized interface which is known at the DML application's compile time. These broker methods are required since the interfaces of the methods of the user classes are not known at the DML application's compile time. The broker methods accept, as two of their parameters, the method name of the method to be executed and the class name to which that method belongs. Along with the other relevant parameters, the broker methods call the appropriate method of the current object of the class specified.

Through this mechanism the DML does not have to know about all of the user's classes; it need only know about this intermediary class instance which in turn knows all about the user classes and provides a means of creating user objects and retrieving and setting individual attribute values.

3.5 Data Manipulation

SQL2Java's DML is modeled after the SQL92 standard. Most of the basic data types and features have been implemented while a few notable exceptions, which are listed below,

were excluded. Currently, SQL2Java contains those operations and features which make SQL2Java a usable system.

SQL2Java's DML includes:

1. Data selection (SELECT statement).
2. Data insertion (INSERT statement).
3. Data update (UPDATE statement).
4. Data deletion (DELETE statement).
5. Group operations including the having condition. (GROUP BY and HAVING).
6. Sorting (ORDER BY).
7. Queries containing subqueries.
8. All standard arithmetic operators (+,-,/,*).
9. The concatenation character operator (||).
10. All standard comparison operators (=,!=,<,>).
11. All standard logical operators (AND, OR, NOT).
12. The majority of the number, character and group functions (Tables 3-1, 3-2 and 3-3).
13. Support for the integer, double and String data types.

SQL2Java's DML does not support:

1. Other data types such as the date data type and the corresponding supporting operations and functions

2. The set operations of union, intersect and minus
3. Any indexing mechanism and consequently any means of ensuring uniqueness

The DML component of SQL2Java is an interpreter [Terr86]. The interpretation contains a number of passes or phases including accepting user input, lexical analysis, syntax analysis, semantic analysis and, finally, execution. Accepting user input is performed by reading lines as they are typed in by the user. The lexical analyzer parses the input creating tokens. The syntax analyzer groups these tokens into groups such as the necessary components of a WHERE clause. The semantic analyzer checks the groups formed by the syntax analyzer to check that each token combination formed makes sense in the current context. It checks such things as whether specified classes and attributes exist. Finally, if no errors are detected then execution of the request takes place. If an error is detected at any stage, it is reported immediately and the interpretation is ceased.

In the following subsections the design of the DML is described in more detail.

3.5.1 Data Selection (SQL92-like)

All of the basic DML operations of SELECT, INSERT, UPDATE and DELETE may utilize data selection as part of their operation. SELECT, UPDATE and DELETE do this directly through conditions placed in their WHERE clause whereas INSERT may select data from another class through a subquery.

SQL2Java evaluates an SQL statement by first determining how the classes involved in the query must be joined. A scheme which groups the classes into sets which must be joined and a preliminary result produced prior to performing the final Cartesian product of all such sets was desired. The following rules are used to determine these sets:

1. Classes whose attributes are being compared in the WHERE clause must belong to the same set
2. If the OR logical operator appears anywhere in the condition sequence or the NOT logical operator is used to qualify more than a single condition then a single class set is created for the entire condition sequence
3. Sets containing common classes are coalesced

These sets, which are named Class Sets, represent atomic groups of classes which must be tested separately since no connection is specified in the condition sequence.

Once the results from each class set are determined then a Cartesian product of the class set results is performed. The results can then be used to satisfy the operations involved. For the SELECT statement, the required operations and functions to satisfy the expressions in the SELECT statement can then be executed to produce the final results.

3.5.1.1 Class Sets

Each data selection involves querying data from one or more classes. If there is but a single class involved then the operation is straightforward, you simply test each object of the class against the condition(s) given. When there is more than one class involved it is

advantageous to attempt to reduce the number of comparisons between object attributes as much as possible.

The strategy used with SQL2Java is to divide the classes involved in a query into independent sets. An analysis of the conditions of the WHERE clause is performed and sets are formed such that classes of each set have no connection to the classes of other sets. The classes within each set are joined, the conditions corresponding to the class set are tested, and attribute combinations satisfying the conditions are stored as intermediate results. These result sets are later joined using a Cartesian product operation.

The number of class sets and the number of classes in each set is determined by the WHERE clause's condition sequence. The greater the number of classes in a set, the longer the time required to test all attribute combinations. The majority of queries will reduce to a single class set but in cases where this is not so then a large savings in processing time can be realized by avoiding needless data accesses and comparisons.

The starting point when forming a class set is the WHERE clause's condition list. Initially each condition forms a class set involving one, two or more classes, depending upon the operations and functions contained within the condition. After the original class sets are formed, an attempt is made to coalesce them. The criterion for merging two sets into one is that there must be a class common to both of the sets being joined.

In testing the conditions corresponding to each class set only those attribute combinations which satisfy the set's conditions will be retained to join to the intermediate results from the other class sets. Inherent in this strategy is that if a condition sequence contains any NOTs which qualify more than a single condition or any ORs then there must be only one set containing all classes. In cases where the NOT logical operator is used to qualify a single condition then this does not disqualify the condition from joining with others. The individual condition result is deduced as if there were no NOT qualifier used and that result is reversed. If these restrictions were not used, all intermediate results would have to be retained regardless of whether or not the class set's conditions were met. Results would have to be retained for each individual condition since there is no correspondence between condition location within the condition sequence and the class set to which the individual condition belonged. Final results would have to be determined by processing the full condition sequence, with TRUE or FALSE values replacing the original conditions.

The retention of this extra information, and the added complexity of the final processing, was not practical when the same processing would be performed if all classes were forced into a single class set. Using these criteria to determine class sets results in sets of intermediate values which do not have to be further tested. After all the intermediate results are determined for the class sets then the final results are obtained by forming a Cartesian product of these sets.

To determine whether the conditions of a class set are met, the class set conditions are tested and truth values obtained. A reduction process is then performed on a logic expression made up of these truth values, logical operators and brackets. The result of this reduction determines a final truth value for each attribute combination. Combinations whose final truth value is TRUE are retained. Combinations whose final truth value is FALSE are discarded. This reduction is possible since the conditions attached to a class set will be in the correct order since they were attached to the class set in that way. If there is more than one class set then it must be the case that all conditions are joined with AND logical operators and, thus, the order of the conditions in the condition sequence for each class set is irrelevant. A further discussion of class sets is given in section 4.2.

3.5.1.2 Final Selection Results

After the results for each class set is produced, the intermediate results must be combined to produce a final result. This join of the intermediate results from the class sets obviously requires a full Cartesian product but it is quite probable that the size of the intermediate results is relatively small compared to the full Cartesian products of the classes involved in each class set.

Once these base results are determined, other processing may be required. If the query contains a GROUP BY clause then the results must be grouped. If there is a HAVING clause then group values must be tested. If the query contains an ORDER BY clause then

the results must be sorted. Finally, the expressions in the select list for either regular or group queries must be executed for each attribute or group value sequence.

3.5.2 *Grouping (Having)*

All queries are either standard queries or group queries. Standard queries produce results involving individual objects and cannot use group functions. Group queries produce collective results such as sums, counts and averages which involve collections of objects. The collections are determined by expressions contained within the GROUP BY clause. Group queries necessarily use one or more of the group functions somewhere in the selection expressions, while the other expressions it contains must either be constants or be made up of expressions found in the GROUP BY clause. Group queries can also have a separate condition clause also containing group functions. This clause is called the HAVING clause.

SQL2Java implements the following group functions:

1. AVG (Average)
2. COUNT
3. MAX (Maximum)
4. MIN (Minimum)
5. SUM

The processing strategy for group queries is similar to that of standard queries. First, the whole query is processed as if it were a standard query. This is necessary since group queries can also contain standard conditions. After this subresult is produced, another result is produced which groups the results by the expressions contained in the GROUP BY clause. Finally, this result can be tested against the conditions contained in the HAVING clause. If the subresult passes this final test then the group functions contained within the select expressions, if they are different from those in the HAVING condition list, are executed and the final results returned.

The following figure illustrates an example of how grouping with a HAVING clause can be utilized. The following query which is from the example University Database is used:

Query: List those courses and semesters where the overall results of the course were exceptional (80% or better where course results are marked out of 100%)

```
SELECT course_num, semester, AVG(result) FROM Taken_By
GROUP BY course_num, semester
HAVING AVG(result) >= 80;
```

Figure 3-5: Sample query illustrating use of GROUP BY and HAVING

3.5.3 *Sorting (ORDER BY)*

The SQL92 DML standard requires a method of sorting query results. The order that the results are to appear are specified in the ORDER BY clause of the SELECT statement. To accomplish this ordering the Randomized Quicksort algorithm [CLR92] was chosen since it runs in $O(n^2)$ time in the worst-case and $O(n \lg n)$ time on average. One of the steps of the basic Quicksort algorithm is to divide subresults. By randomizing the pivot position in this division process this algorithm ensures that its complexity is independent of the order of the input data. This makes the algorithm more consistent.

The sort is utilized after the final results, which include the evaluated expressions contained in the ORDER BY clause, are available. The sort may or may not have to be executed for all of the expressions in the ORDER BY clause depending upon whether or not the ordering at one level completely orders the data or not. If not, the sort is recursively called for each of the subsets of data which are not yet fully sorted.

Each sort expression in the ORDER BY clause can be qualified with either ASC (ascending order) or DESC (descending order). Ascending is the default and does not have to be specified.

The following figure illustrates the use of the ORDER BY clause in a query. In this query the students in the “FALL1997” semester of the course “95511F1” are sorted by their

course results. The DESC qualifier on the ORDER BY clause orders the students from highest performer to lowest performer.

```
SELECT name, result FROM Taken_By
WHERE course_num = '95.511F1' AND semester = 'Fall1997'
ORDER BY result DESC;
```

Figure 3-6: Sample query illustrating use of ORDER BY clause

3.5.4 Functions

SQL2Java has the majority of the standard functions defined in the SQL92 standard and a large number of extra functions which DBMS vendors typically provide to increase usability of their systems. The following three tables list those functions which have been included in SQL2Java.

ABS(n)	Absolute Value of n
CEIL(n)	Smallest number greater than or equal to n
FLOOR(n)	Largest number less than or equal to n
MOD(m,n)	Remainder of m divided by n
POWER(m,n)	m raised to the nth power
ROUND(n[,m])	n rounded to m significant digits
SIGN(n)	-1 if n<0, 0 if n=0, 1 if n>0
SQRT(n)	Square root of n

TRUNC(n[,m])	n truncated to m decimal places
--------------	---------------------------------

Table 3-1: SQL2Java Number Functions

INITCAP(char)	char, with first letter of each word capitalized
LOWER(char)	char, with all letters in lowercase
LPAD(char1,n[,char2])	char1, left-padded to length n with the sequence of characters in char2
LTRIM(char[,set])	char, with initial characters removed up to the first character not in set
REPLACE(char,search_string[,replacement_string])	char, with all occurrences of search_string replaced with replacement_string
RPAD(char1,n[,char2])	char1, right-padded to length n with the sequence of characters in char2
RTRIM(char[,set])	char, with final characters removed after the last character not in set
SUBSTR(char,m[,n])	Substring of char, beginning at character m, n characters long
TRANSLATE(char,from,to)	char, translated from the character set from to the character set to
UPPER(char)	char, with all letters in uppercase
INSTR(char1,char2[,n[,m]])	Position of the mth occurrence of char2 in char1, beginning search at position n

LENGTH(char)	Length of char
--------------	----------------

Table 3-2: SQL2Java Character Functions

TO_CHAR(expr)	expr converted to a character value
TO_NUMBER(char)	char converted to a number
GREATEST(expr1,expr2)	greatest of expr1 and expr2
LEAST(expr1,expr2)	least of expr1 and expr2

Table 3-3: Other SQL2Java Functions

The following table gives at least one example of the use of each of the functions listed above. All queries are against the Employee class of the sample University Database. For this example, there exists just a single object in the Employee class. The first entry in the table is the statement which creates the object which all the queries are against.

insert into Employee(Employee_Num,Salary,Name,DOB) values (1234,54321.12,'John Doe','01-JUL-1967');	
select ABS(salary) from Employee;	54321.12
select ABS(salary * -1) from Employee;	54321.12
select CEIL(salary) from employee;	54322.0
select FLOOR(salary) from employee;	54321.0
select MOD(salary,10) from employee;	1.12
select POWER(employee_num,2) from employee;	1522756.0
select ROUND(salary,1) from employee;	54321.1
select SIGN(salary) from employee;	1.0
select SIGN(salary * -1) from employee;	-1.0

select SQRT(employee_num) from employee;	35.12833614050059
select TRUNC(salary,1) from employee;	54321.1
select INITCAP(lower(name)) from employee;	John Doe
select LOWER(name) from employee;	john doe
select LPAD(name,20,'ABC') from employee;	ABCABCABCABCJOHN DOE
select LTRIM(name,'JONABC') from employee;	HN DOE
select REPLACE(name,'DOE','FAWN') from employee;	JOHN FAWN
select RPAD(name,30,'ABC') from employee;	JOHN DOEABCABCABCABC
select RTRIM(name,'ABCDE') from employee;	JOHN DO
select SUBSTR(name,1,2) from employee;	OH
select TRANSLATE(name,'ABCDEF','GHIJKL') from employee;	JOHN JOK
select UPPER(LOWER(name)) from employee;	JOHN DOE
select INSTR(name,'O',1,1) from employee;	1.0
select INSTR(name,'O',1,2) from employee;	6.0
select LENGTH(name) from employee;	8.0
select TO_CHAR(salary) from employee;	54321.12
select TO_NUMBER(TO_CHAR(salary)) from employee;	54321.12
select GREATEST(salary,3) from employee;	54321.12
select LEAST(salary,3) from employee;	3

Table 3-4: SQL2Java Function Call examples

3.5.5 Operation Atomicity

One of the basic tenets of Database System design is that the atomicity of transactions must be ensured. Atomicity refers to the principle that the operations which make up a

transaction must all be executed to completion, or none at all [KS91]. SQL2Java does not provide a facility for the user to group separate operations into a transaction. To complete a single UPDATE or DELETE SQL operation, though, may involve manipulation of more than a single object. In such cases, the atomicity of these operations must be ensured. If a mechanism were not put in place to guarantee this atomicity, a system crash could result in an inconsistent data set.

The way SQL2Java ensures operation atomicity is to utilize a log file mechanism which completes data manipulation operations in two stages. The first stage writes the data modifications which are to be made to the log file. The second stage actually makes those modifications to both the existing Java objects and to the corresponding data in the data files. At most one operation is represented in the log file at any one time. When both stages of this operation are executed to completion, the operation is removed from the log file.

The log file is checked when the DML application starts and an operation, recorded in the log file, which was not completed due to a system crash will be completed before instantiating the user objects. Completion of an operation, in this case, refers to making the data file modifications which were recorded in the log file but were never completed. After the necessary data file modifications have been completed, the correct user objects will be instantiated from the modified data files at the same time as all the other user objects are instantiated. SQL2Java's log file functionality is further described in section 4.4.

3.6 Persistence

Persistence in the SQL2Java system is provided by storing all meta and user data in data files which correspond to each class. For the metadata classes, all the information required to create a single object is stored on a single line of its data file. This is feasible since the types and sizes of the individual data elements within the line are well defined and cannot be modified. Data for the instances of the user classes is stored one attribute per line of its data file since data attribute sizes may vary between instances. This convention eases the data loading process since a whole line of data can be extracted without having to split the data into different attributes.

The first byte of each data file indicates whether or not the data file requires cleanup. Cleanup is required when objects in the data file have been marked for deletion. This is done whenever an object is deleted by a user or when an attribute is updated. Since an insert into a file, other than to the end of the file, cannot be done without overwriting existing information an update is performed by marking the existing object for deletion and adding a new object to the end of the data file. Simply overwriting the existing information cannot be done since it is possible that the new attribute information could be longer than the existing attribute information. Instead of always reserving the maximum space possible for each attribute in the data file and, thereby, wasting vast amounts of disk space update has been implemented in this way.

To mark an object for deletion, both in the data files for the metadata and the user data, one byte per object is reserved. For the metadata data files this is the first byte of the line. For the data files of the user classes, it is the first byte of the first line of data for the object, that is the line containing the data for the first attribute of the object. If the byte's value is '1', this indicates that the object is marked for deletion. The same byte is set to '0' for all other objects.

3.6.1 Data File Reorganization

To ensure that the data files do not get filled up with objects marked for deletion, the last thing done at the end of a DDL session is to reorganize the metadata data files which require it. Likewise, the last operation of a DML session is to reorganize the data files for the user classes which require it.

This reorganization is performed by copying the existing data file to a temporary file, deleting the real data file, recreating it as an empty file and copying back only those objects which have not been marked for deletion.

For the metadata data files, this opportunity is taken to group related data which belongs together. Before the reorganization in the *MetaAttribute* data file, for example, the attributes belonging to a particular class could be spread throughout the data file. Afterwards, the lines containing the attribute information for the one class will be contiguous in the data file. Though this reorganization is not strictly necessary in that the

data will still load correctly, the reorganization serves to reclaim valuable disk space and speeds up data loading since less data file traversal is required when related data is stored together. Data file reorganization is discussed in greater detail in section 4.3.3.

4. Design and Implementation Details

4.1 Intermediary *User_Objects* Class

Since Java programs must be compiled, a running Java application cannot access a class which it creates and compiles itself. For a Java application to be able to reference objects of a Java class, that Java class must exist at compilation time. This is obviously not the case in a database system built using the Java OOPL and using Java classes for both its operational code and for its basic data definition and storage capabilities.

In creating a database system using Java, this fact presented a problem. Since the running application could not directly access a class that it created and compiled into a Java “.class” file, it was necessary for the DDL and the DML portions of SQL2Java to be separate applications. But even making the DDL and the DML separate applications was not yet enough without having to modify and recompile the DML application after every addition to or deletion from the database scheme to add or remove objects references. To get around this problem, an intermediary class was needed which the DDL could redefine and recompile after database scheme changes. If this class was compiled after new or modified class files were compiled, it could contain true references to all user classes and would compile successfully. The implication is that the DML can never access any user

objects directly; it must always go through the methods and attributes made available by the single instance of this intermediary class.

The basic procedure for using SQL2Java is to first use the DDL to define user classes. Upon starting, the DDL loads all metadata from the metadata data files. The DDL can then be used to make modifications to the database scheme. When the user supplies all the proper information to define a new class or to alter an existing class, the DDL will create and compile the actual class files for the user classes and it will make the necessary modifications to the metadata classes and their corresponding data files. The user can make as many modifications to the database scheme as required and, finally, upon exiting the DDL it will create a new version of the intermediary class containing methods which the DML can use to access the instances of all of the user classes including the new ones just defined. Once the DDL has exited successfully, the DML can be started. The DML, upon starting, will first load all the metadata which the DDL did but it also loads all of the user objects. To do this it instantiates the intermediary class which has code in its constructor to parse the data files for each user class and instantiate all user objects. At this point the user is able to begin issuing DML requests. The DML will never modify any of the data of the instances of the metadata classes nor their data files. It will, however, upon request from the user, modify instances of the user classes.

4.1.1 How the intermediary class is built

The DDL has a method which recreates and compiles the intermediary class which is always called prior to exiting. This method gets all of the information that it needs to build the new intermediary class from the objects of the metadata classes. Since this method is always called after the user has completed all required modifications to the user objects, the instances of the metadata classes contain all the necessary information SQL2Java needs to correctly recreate the intermediary class. This method parses the objects of the metadata classes as required to obtain the information necessary to rebuild the class definition file. This file is built and recompiled immediately afterwards.

4.1.2 Intermediary Class Contents

Since the DML can never contain references to any user classes itself, an instance of the intermediary class must contain references to them all and provide a mechanism for the DML to access them. These references are created when the intermediary class is instantiated. The intermediary class is to be instantiated only once per DML session and accesses to user objects are obtained only through this one object.

Appendix B contains a listing of the intermediary class definition after only the Person entity of the sample University Database scheme had been defined in the DDL. The figure below gives the attributes and method interfaces of the same intermediary class definition as is given, in full, in Appendix B.

```

class user_objects {
/* Attributes */
    Person first_Person = null;
    Person current_Person = null;
    Person previous_Person = null;
    Person last_Person = null;
/* Constructor */
    user_objects() {
        first_Person = load_Person();
        current_Person = first_Person;
    }
/* Load Methods */
    private Person load_Person() {...}
/* User Class Constructor Brokers */
    public void constructor_broker(String Method_Name, String
store_string,String PARAMETER1,String PARAMETER2) {...}
    public void constructor_broker(String Method_Name, String
store_string) {...}
/* User Class Method Brokers */
    public String String_broker(String Method_Name, String
Class_Name){...}
    public void void_broker(String Method_Name, String Class_Name,
String
    public int int_broker(String Method_Name, String Class_Name) {...}
    public double double_broker(String Method_Name, String
Class_Name){...}
    public void void_broker(String Method_Name, String Class_Name, int
PARAMETER1) {...}
    public void void_broker(String Method_Name, String Class_Name,
double PARAMETER1) {...}
/* User Class Reset Methods */
    public boolean reset_Person() {...}
/* User Class Advance Methods */
    public boolean advance_Person(int orig_filesize) {...}
/* User Class Delete Methods */
    public boolean delete_Person(int orig_filesize) {...}
/* User Class Master reset, Advance and Delete Methods */
    public boolean master_reset(String class_name) {...}
    public boolean master_advance(String class_name, int
orig_filesize){...}
    public boolean master_delete(String class_name, int
orig_filesize){...}
/* User Class Get File Position Method */
    public int get_fp(String class_name) {...}
}

```

Figure 4-1: Intermediary class attributes and methods

The intermediary class contains one method per user class which loads the data (instantiates the objects) for that class from the class's data file. The intermediary class

also contains a number of references for each class which are utilized by the intermediary class instance, upon request of the DML, for data access or manipulation. The constructor of the intermediary class calls each of the load methods for the user classes and sets all the proper object references.

The next set of methods contained in the intermediary class are the broker methods. Broker methods are required since the DML does not know the names of the individual methods of the user classes when the DML application is compiled. These broker methods, which are called by the DML, in turn call the true methods of the current user object of the class specified. Since the intermediary class can have only one method of a particular name, returning a particular type, and containing the same number and types of parameters, it was necessary for certain broker methods to provide access to more than one method. To specify which method of which class the DML wants to execute, the DML supplies the class name and the method name as the first two parameters to all broker methods. One exception to this rule are the brokers which call the constructors of the user classes. These broker methods require only the name of the class for which an object is to be created plus the parameters required by the constructor.

The broker methods permit the DML to execute and receive the returned values from the get and set methods which were created for each user class by default. This is the mechanism by which an object's attributes are set and how these values are returned to the DML. Currently SQL2Java is creating all the appropriate broker methods for all the methods of the user classes, including those defined by the user and the default get, set

and constructor methods. Since the DML does not know which user classes will exist and which methods they will contain, it must contain code which calls six standard broker methods: one broker to set an attribute of each of the three base data types and one broker to get an attribute of each of the three base data types. When the intermediate class file is built, dummy broker methods are created if there are no attributes for one or more of the base data types. If this were not done, the DML would contain calls to methods in the intermediary class which do not exist.

There are two more types of methods in the intermediary class. The first set is a set of methods which position the object references of a particular class, which delete objects of a class and which return the file position of the current object of a particular class. There is one reset method per class which sets all objects references for a class back to the first object of the class; one advance method per class which advances the "current" object reference forward one object; and one delete method per class which deletes the object referenced by the "current" object reference. These methods are the only methods able to reposition the object references; therefore, they are necessary in order to parse through the objects of a class. There is also a single method which returns the current file position of the current object of the class given as a parameter. This information is required by the UPDATE and DELETE operations to reset specific bytes in the class's data files. The name of each of the reset, advance and delete methods contain the name of the class being affected. This means that since the DML does not know the user class names when it is created, the DML cannot directly call these methods. Finally, this introduces the last set of methods for the intermediary class.

The last three methods of the intermediary class are the master reset, master advance and master delete methods. These methods act as broker methods to the reset, advance and delete methods for each class. To these last three methods the class name must be supplied as a parameter.

The intermediary class is changed by the DDL to reflect any modification to the database scheme. It thereby serves to protect the integrity of both the DDL and the DML code by preventing it from requiring any direct reference to the user objects.

4.2 Class Sets and Query Processing

The simplest technique to join objects of classes in order to test whether they satisfy query conditions is to do a Cartesian product of all classes involved and test all combinations of the object's attributes. This technique is obviously very inefficient.

Commercial DBMSs use a variety of techniques to optimize the query process. The most popular technique is through the use of indexes. Indexes hold object address information which is used to access data without scanning all objects of the class. Indexing is also used to ensure uniqueness of objects. If a particular attribute has an index associated with it and that attribute is used in a condition then the database system knows exactly which objects of that class to test against those of other classes being joined.

Since SQL2Java does not have an indexing mechanism, the challenge was to come up with a data access strategy which was correct and reasonably efficient. The technique used is to parse the condition sequence and produce sets of classes, each of which contain the classes which must be joined. In most cases, this will result in a single set containing all classes involved in the condition sequence. It is possible, though, for classes to be involved in a query but not be mentioned at all in the query's condition sequence. Also, a class or classes can be involved in the condition sequence but be isolated from the other classes due to the absence of any condition involving attributes from both sets of individual conditions. In cases such as these, separate class sets are required to correctly obtain the necessary objects from each class without wasting processing time. By segregating the classes and processing them in isolation, intermediate result lists can be obtained which can be joined afterwards using a Cartesian product.

4.2.1 Rules for determining Class Sets

Class sets are created by parsing the conditions of the query's condition sequence and making sure that any classes which are explicitly joined through an individual condition belong to the same class set. To determine the class sets, one class set is created per condition and, afterwards, these sets are coalesced until each set contains only those classes which join to those of other classes of its class set and none of the classes of other class sets.

Though that is the basic procedure for determining class sets, there is one exception in this procedure. For efficiency reasons, the creation of a single class set is forced if the logical operator OR is used anywhere within the condition sequence or if the logical operator NOT is used to qualify more than individual conditions enclosed in brackets. The reason for this exception is that the objective is to parse all the conditions and handle all condition processing within the class set functionality. If the conditions of class sets which are joined by logical operators other than AND were allowed, then all possible attribute combinations for each class set would have to be retained until the final joining of all the individual condition's results. For instance, if the OR logical operator were used between the conditions of two class sets then only one of the two would need to be successful for the whole sequence to be successful. If only the AND logical operator were used, both would have to be successful for a successful result. Therefore, to prevent the retention of larger than necessary amounts of data which may not be involved in a successful result once the full condition sequence is eventually evaluated, SQL2Java forces the creation of a larger class set. Though the amount of processing time increases as the number of classes in a class set increases, each combination of objects is immediately evaluated when the attributes involved are collected and only those combinations satisfying the conditions of the class set are retained.

The case where the NOT logical operator is used to qualify a single condition is handled relatively easily by reversing the result for that one condition when it is processed. Similar procedures with larger collections of conditions could also be implemented, but the implementation gets steadily more difficult as the complexity of the condition

sequence gets larger. This implementation of SQL2Java does not attempt to do any further class set splitting based upon condition sequence analysis.

To illustrate how SQL2Java determines class sets, the class sets which result from the following examples, which use the sample University Database scheme given previously, are given in the table below.

Query	Class Sets
select p.name, p.faculty, s.name, s.faculty from Professors p, Students s;	1. (Professors) 2. (Students)
select p.name, p.faculty, s.name, s.faculty from Professors p, Students s where p.faculty = 'Computer Science' and s.faculty = 'Computer Science';	1. (Professors) 2. (Students)
select p.name, p.faculty, s.name, s.faculty from Professors p, Students s where p.faculty = 'Computer Science' and NOT(s.faculty = 'Computer Science');	1. (Professors) 2. (Students)
select p.name, p.faculty, s.name, s.faculty from Professors p, Students s where p.faculty = 'Computer Science' or s.faculty = 'Computer Science';	1. (Professors, Students)
select p.name, p.faculty, s.name, s.faculty from Professors, Students s where NOT(p.faculty = 'Computer Science' and s.faculty = 'Computer Science');	1. (Professors, Students)
select p.name, p.faculty, s.name, s.faculty from Professors p, Students s where p.faculty = s.faculty;	1. (Professors, Students)
select s.name, c.course_num, t.mark from Students s, Courses c, Taken_By t where s.student_num = t.student_num and t.course_num = c.course_num;	1. (Students, Taken_By, Courses)

Table 4-1: Sample queries and the Class Sets they produce

4.2.2 Class Set Processing

Within each class set all possible combinations of objects are obtained by parsing through the objects of each class recursively and testing the combination when the data for each

class in the set is obtained. A string containing the logic elements TRUE and FALSE, the logic operators AND, OR and NOT and brackets is created after the value of each individual condition is determined. This sequence is fed to a logic string evaluator which recursively reduces the string until a truth value is obtained. If successful, that attribute combination is added to a subresult list for the class set.

The attributes stored include all attributes required to satisfy further processing. This includes those attributes appearing in the SELECT list, the ORDER BY list, the GROUP BY list, and the HAVING list.

After all class set subresults have been determined, they are all joined using a Cartesian product. In the case where there is but a single class set, this joining process is obviously not required. Following this procedure, all necessary data required to complete the operation is available. If the query is a group query, further processing is performed to group the results by the elements of the GROUP BY clause. If there is also a HAVING clause then the group results are also tested against the conditions of the HAVING clause. Both regular and group queries can also contain an ORDER BY clause. If this query does, the result set is sorted using a Randomized Quicksort algorithm. Finally, the expressions in the SELECT list are determined for each successful result. This information is either presented at the console or is returned to a parent operation if the query was a subquery.

4.3 Persistence

The most basic requirement of a database system is that data stored in the database persist between executions of the database system. This requirement is met by storing database data, organized in a standardized format, to permanent storage. The storage device is usually a hard disk while magnetic tape units are often used for backup purposes. The manner in which the data is organized varies greatly between DBMSs. The timing of when data is written to the various files also varies and is often dependent upon the backup and recovery strategy of the DBMS.

Similarly, SQL2Java stores all of its database data in files. The method used is quite straightforward and is not optimized for speed. The technique used is to create a separate data file for each system and user class. These data files are always kept synchronized with the memory versions of the same data. Therefore, an operation affecting either system or user data will affect the appropriate data files at the same time as the data stored in the computer's memory.

The current implementation of SQL2Java does not have any provision to protect the data files or their data. There is no redundant storage of the data nor are there any special mechanisms to secure the files. However, operations affecting either the system or user data will affect both the memory and data file versions of the objects at the same time. If the system shuts down inadvertently, it can be brought back up without any data loss. This is ensured through the use of a log file mechanism as described in section 4.4. The

operation being executed at the time of the crash may not have been reflected in the appropriate data files, but all previously completed operations will have been. If a user is concerned with this lack of data file security then a manual copy of all data files, for system and user classes, at strategic intervals can be used. If the data files for the database were to become corrupted at a later stage then they could be replaced with those which were manually copied elsewhere. If there were any data manipulation operations completed between the time that the data files were backed up and when they were copied back into production then data loss will occur and operations will have to be resubmitted.

Operating systems typically place limits on the number of files which are allowed to be open at any one time. This could be a concern with SQL2Java if a data manipulation operation involves more classes than the upper limit on open files. To complete data manipulation operations SQL2Java has to open the data file corresponding to each of the classes involved in the operation in order to complete the necessary operations on those data files. Since the limit on open files is usually fairly high, typically 50 or more, this is not a big concern since a data manipulation operation involving that many classes would be quite rare.

Below are example data files. Figure 4.2 shows a possible data file for the *Student* class of the example University Database. It contains the data of two objects of the *Student* class. Neither of those objects are marked for deletion; therefore, the data file itself is not marked for reorganization.

The second example data file, given in Figure 4.3, shows the possible contents of the data file for the meta data class *MetaClass*. It contains the data representing the user classes defined in the example University Database.

```

001
COMPUTER SCIENCE
BACHELOR OF COMPUTER SCIENCE
JOHN DOE
01-JAN-1977
02
COMPUTER SCIENCE
MASTER OF COMPUTER SCIENCE
JANE DOE
14-NOV-1966

```

Figure 4-2: Data file for the *Student* user class

00Person	11	01
0Employee	11Person	01
0Professor	11Employee	01
0Student	11Person	01
0Course	11	01
0Taken_By	11	01
0Taught_By	11	01

Figure 4-3: Data file for the *MetaClass* meta data class

4.3.1 Application Start Up

When either the DDL or the DML starts, a data loading procedure is initiated. These procedures parse the appropriate data files and instantiate Java objects. The start up procedure of the DDL instantiates only the system objects which are those objects which define the database scheme. The DML, on the other hand, instantiates both the system objects and the user objects.

The procedure which instantiates the system objects is quite different from the procedure which instantiates the user objects. This difference is due to the fact that the format of the system classes are constant and well defined. Therefore, the procedure to load this data is hardcoded within the applications. The database scheme, on the other hand, is quite fluid and therefore the DML application requires assistance when instantiating the user objects. This assistance is provided by the single instance of the intermediary *User_Objects* class. This class is built containing one load method corresponding to each user class which, when called parses the data file corresponding to the user class and instantiates the objects for that class. The constructor for the intermediary class calls each of these load methods and, consequently, all user objects are instantiated. When user objects are loaded the exact file position of the object in the data file is also recorded. This information will enable the delete and update procedures to have direct access to the byte which needs to be reset to indicate that the object is deleted and is to be removed from the data file upon data file reorganization.

After the data loading procedures have completed, access to the system objects is direct since the classes which store the system data are known and already defined at the time SQL2Java was implemented. This is obviously not the case for the user classes since database schemata will vary from installation to installation. Therefore, access to user classes and objects must be obtained through the single instance of the intermediary class. This intermediary class provides mechanisms to not only load the user objects but also to access methods which get and set individual attribute values.

4.3.1.1 General Operation Recovery Mechanism

To ensure atomicity of DML operations, the startup procedure of the DML application also processes operations contained in the operation log file. This procedure is initiated after the meta data objects have been instantiated but before the user objects have been instantiated. The meta data objects must be instantiated before beginning the recovery from the log file since the recovery procedure uses some of the meta data objects. If the log file contains any operations which have not been marked as having been completed, then those operations are carried out on the data files of the user classes before those same data files are processed to instantiate the user objects. This step ensures that an operation which was started in the previous DML session, and which had its modifications successfully written to the log file but which was unable to complete due to the session inadvertently ending, will complete upon startup of the new DML session.

4.3.2 Operations on Data Files

The operations SQL2Java is able to perform on data files are quite limited. SQL2Java can read lines or bytes from a data file or write lines or bytes to a data file. With these operations SQL2Java is able to properly manage the data files. SQL2Java users never get direct access to the data files which store SQL2Java system or user data through the SQL2Java DDL or DML Graphical User Interfaces (GUI).

To load objects, SQL2Java reads lines from the data files and uses this information to instantiate either system or user objects. As part of this procedure SQL2Java skips any objects marked for deletion. This marking is indicated with the digit “1” occurring in the first byte of the object in the data file instead of the digit “0”.

Deleting objects from the data file requires two steps. The first step is to mark objects in the data file which need to be removed from the data file. They cannot simply be extracted from the data file and the data file compressed since the attributes containing the physical data file locations for the objects would also have to be adjusted and a great deal of processing time would be wasted in physically reading and writing the lines following the lines being deleted from the data file. The second step, therefore, is to reorganize the data file to physically remove the unwanted lines. This operation is performed only when either the DDL or the DML applications are closed. This procedure is described in section 4.3.3. Marking the lines to be deleted in the data file is performed by simply seeking to the first byte of the object in the data file and writing the digit “1”. The delete operation is used when deleting a class from the database scheme, when deleting a user object or when updating a user object. It is used when updating a user object since it is not possible to simply overwrite the existing data in the data file since the data being written may not be the same length as the data currently in the data file for the object. Without reserving the maximum possible space in the data file for all attributes, it is necessary to implement update by deleting the existing object in the data file and creating a new one at the end of the data file with the updated information. In cases where objects are deleted, the objects in memory are also deleted as part of the

operation which marks the data file(s). Similarly, when an object is updated the object in memory is also updated as part of the operation which marks and inserts into the data file.

Inserting objects involves creating new objects in memory while at the same time creating the data strings containing that data and writing them to the appropriate data file(s). Inserts are always done to the end of data files, but these objects may be moved upon data file reorganization in the case of the system data files.

Inserts are required in the DDL when the user either creates a new class or modifies an existing class. In the DML, insertions are done when the user creates new user objects or updates existing user objects. As with data loading, the physical data file location is recorded when an object is inserted in a data file. This will allow the object to be easily marked for deletion if the user performs an operation which requires its deletion.

4.3.3 Data File Reorganization

Data files are reorganized at the end of each DDL or DML session. This procedure serves two purposes. The first and most important purpose is that it reclaims valuable disk space. The second is that, in the case of the system data files, object lines are ordered such that related data is made contiguous. This eases the data loading process.

The data file reorganization procedure is necessary since as SQL2Java is running lines to be deleted from the data file are not actually deleted, they are only marked for deletion.

To correctly modify the data files while SQL2Java is running would incur a great deal of unnecessary overhead. Delaying the compression and possible ordering of the data file lines until shutdown allows the applications to run faster.

Figure 5.3 below gives a simple data file for the Person class before and after reorganization. In this case six objects were marked for deletion.

Before	After
10CHRIS LAMBERT	00CHRIS LAMBERT
14-NOV-1966	14-NOV-1966
1JOHN DOE	0TAMMY LAMBERT
01-JAN-1960	06-JUN-1966
1JANE DOE	0JESSICA LAMBERT
01-JUL-1967	31-Jul-1997
0TAMMY LAMBERT	
06-JUN-1966	
1FRED SMITH	
14-FEB-1923	
1MARY JONES	
25-DEC-1939	
1JACK HUNT	
05-NOV-1966	
0JESSICA LAMBERT	
31-JUL-1997	
1BILL WHITE	
11-NOV-1945	

Figure 4-4: Sample data files before and after Reorganization

The reorganization is performed by first making a copy of the data file to be reorganized. Then the original data file is deleted and, immediately afterwards, a new file is created with the identical name as the original. The next step is to read lines from the copied file and, if they are not marked for deletion, to write them to the newly created file. For system data files this procedure is slightly different in that the system data in memory is used to organize the data written back to the new data file in a more organized manner.

For example, the objects of the *MetaMethodLine* data file are organized by both class name and method name. This procedure will ease data loading since data from the system data files is loaded in an organized fashion, starting with the class data, then the attribute data, then the method data and so on. If all data for a class is contiguous, this will potentially save data file parsing since the last object in the data file related to the current parent is always marked as such. Therefore, when the last line containing an attribute for a particular class is read the loading procedure knows that it need look no further in the data file for objects for that class.

This reorganization is performed for all data files requiring it. This is indicated in two ways. The first is that the data file itself reserves its first byte to indicate this. If it is set to “1” the data file requires reorganization. If it is set to “0” it does not. Further, an indicator is recorded in the system data. Each class which has a data file has a reference to an object of the *Datafile* class. This class contains data file specific information such as the name of the data file and, possibly, the length of data lines for data files which store system data. An object of the *Datafile* class also stores whether or not the data file requires reorganization. If a system crash were to occur and the data file reorganization procedure was not performed, then this *Datafile* class object will be recreated and its reorganization indicator reset from the first byte read from the physical data file when the application is restarted.

After a data file has been reorganized, the first byte of the data file is reset back to “0” to indicate that it is no longer in need of reorganization.

4.4 Operation Atomicity

SQL2Java ensures operation atomicity through the use of a log file. The log file holds the information required to recover operations following a system crash.

Completion of either of the data manipulation SQL operations (INSERT, UPDATE AND DELETE) is done in two stages. During the first stage enough data is written to the log file for a recovery procedure to be able to complete an operation which has started into its second stage. During that second stage the actual Java objects and their data file representations are modified. Until the second stage starts, the log file does not yet have enough information to complete the whole operation and if a crash occurs during the first stage an attempt to recover the operation is not attempted.

The log file always exists but is constantly changing size. When a data manipulation operation is taking place then the data necessary to complete the data file manipulation corresponding to that operation is written to the log file. This is repeated for all objects affected by the operation. After all the necessary data is added to the log file, the log file is marked as complete and the actual manipulation of the Java objects and the data file is started. As each object has actually been modified, then the entry in the log file corresponding to that object is marked as being completed. When all objects have been modified as required, the entire log file is cleared.

In the event that a system crash occurs during the first phase when data is being written to the log file and the operation has not yet completed, the log file is not marked as complete since there hasn't been enough information written to the log. Upon restarting the DML, application will check the log file but will not use it if it is not complete. In this case it will appear to the user that the operation was never issued.

In the event that a system crash occurs after writing to the log file is complete but the true Java object and data file manipulation is still underway, the objects in the log file which have not been marked as completed will be completed. This is accomplished by making the appropriate modifications to the data files. The exact modifications required to the data files is specified in the log file. After each modification is complete it is marked in the log file as complete just in case there is another system crash. Since the procedure to instantiate all the user objects from the data files has not yet taken place, making the necessary modifications to the data files is sufficient. After the log file is processed and all necessary modifications to the data files are made, user object instantiation begins. This instantiation procedure parses all user class data files, including those just modified, and creates all user objects. The state of the database after this instantiation reflects what it would have been had the original system crash not occurred.

Entries to the log file are made up of either single object deletes or single object inserts. A SQL INSERT operation is represented by a single object insert, a SQL DELETE operation is represented by one or more objects deletes and a SQL UPDATE operation is represented by an object delete and an object insert for each object being updated. To

prevent the possibility of a double write occurring when an insert is being written to the data file but a crash occurring before SQL2Java is able to record this fact to the log file, SQL2Java actually records the position in the data file where both deletes and inserts are to take place. This is possible since exact data sizes are known at the time of writing to the log file. When the recovery procedure uses these same file positions, data partially written to the data file will be overwritten with the full data representation. If this provision were not in place and partial data for an object were written to the end of a data file, the data file could be left in an inconsistent state since the recovery procedure will once again write the complete data representation to the end of the data file. By knowing exact file locations to write to, a partial data representation is overwritten with the complete data representation. Though the use of the log file increases the processing time per operation, it is required to ensure the database is not left in an unrecoverable inconsistent state.

5. Applications and Extensions

As communication networks become more common and computer hardware becomes faster and smaller, users are able to become more mobile in their work. It would be advantageous if such users could run the same programs regardless of the hardware and software platform they happen to be using. This would mean that these users would become more proficient in the use of these programs which would enable them to perform their primary tasks more efficiently.

SQL2Java provides this flexibility. Being a Java application, SQL2Java can be ported to any platform with an installed JVM. For those proficient in SQL, SQL2Java allows access to their data through this recognized interface. Applications developed in SQL2Java in its current state, or utilizing an extended version of SQL2Java, will be able to take advantage of these strengths.

5.1 Example Applications

Possible applications of this software in its present condition would be limited to non-commercial installations due to SQL2Java's lack of utilities necessary to make it

acceptable for mission-critical applications. The following examples illustrate how SQL2Java could be applied in its current state.

5.1.1 Teaching Tool

As SQL2Java's DML is based upon the SQL92 standard, it could be used as a low cost means of providing students with a system on which they can practice SQL language skills.

5.1.2 Personal DBMS

SQL2Java is a small DBMS with SQL access which individuals without stringent commercial requirements may utilize as a personal portable DBMS.

5.2 Implementing a Database using SQL2Java

This section will take you through the necessary steps in implementing a database application using SQL2Java.

5.2.1 Installation and Startup

To prepare for the implementation of any application using SQL2Java, it must first be installed on a platform running a JVM. When this has been done, the *Setup* class must be properly configured and recompiled. This configuration requires platform specific knowledge such as the location of the utility for deleting files and the exact newline string used on your platform. This information must be entered into the *Setup* class definition file (setup.java) using a text editor. When this has been done, it is necessary to compile the setup.java file to produce a setup.class file. The compilation is done using the java compiler (javac). Refer to Figure 3.3 in Section 3.3 for an example of the *Setup* class.

Provided that the information placed into the *Setup* class definition file is correct and that it compiled correctly, SQL2Java is ready to be used. To start SQL2Java the user must first use the DDL to define the database scheme. To start the DDL module the user must execute the DDL application using the following command.

```
java ddl
```

5.2.2 Creating a Database Scheme

The steps required to implement a database application in SQL2Java are the same as with any DBMS. The database scheme must first be defined using the DDL portion of the

database system and then the DML portion of the database system can be used to insert, query and manipulate user data.

Unlike other database systems the DDL and DML components of SQL2Java are separate applications. To use each of these two components its application must be started and stopped before the other is used. This is necessary since the DML component depends upon output from the DDL component.

The SQL2Java DDL component has a GUI interface. This interface has two separate areas. The first area is where the user is able to enter requests, this area is a single line at the top of the SQL2Java DDL window. The second area, where information from SQL2Java for the user is presented, is much larger and is directly below where the user enters requests. This area has multiple lines, is scrollable and is where SQL2Java presents both responses to SQL2Java requests and error messages.

Upon starting, the DDL class instantiates all of the meta data objects, if any exist. If the database system is new, no meta data objects will exist. After this is complete, the user is able to define the database scheme. This is accomplished by issuing create, alter and delete commands. The user is guided through the use of each of these commands with step by step requests for information. If all goes well, the complete database scheme will be defined and ready for use provided that the SQL2Java DDL component is exited properly using the “exit” command. If a crash were to occur then it is necessary to restart the DDL and exit it properly since the intermediary class, needed by the DML component

to access the user classes, must be recreated and recompiled at the end of each DDL session. At this time the data files containing the meta data are reorganized to remove data marked for deletion and to reposition the entries for more efficient loading.

The following figure gives a typical DDL session in which a single class is defined.

```
Please wait while system data loads ...
Please enter DDLBoss Request (create, delete, alter, list, desc, or
exit) : create
Please enter Class Name: Person
Is this class final (Y)es, (N)o: N
Is this class public (Y)es, (N)o: Y
If this class has a super class enter its name, else hit return:
At least one Attribute must be entered.
Please enter the name of the attribute :Name
Please enter the type of the attribute (int, double or char): char
Please specify length of char string: 30
Is this attribute public (Y)es, (N)o: y
Can this attribute be null (Y)es, (N)o: n
Please enter the initial value of this attribute: default
Are there more attributes (Y)es, (N)o: y
Please enter the name of the attribute: DOB
Please enter the type of the attribute (int, double or char): char
Please specify length of char string: 11
Is this attribute public (Y)es, (N)o: y
Can this attribute be null (Y)es, (N)o: y
Please enter the initial value of this attribute: 01-JAN-1900
Are there more attributes (Y)es, (N)o: n
Are there any methods (Y)es, (N)o: n
Please enter DDLBoss Request (create, delete, alter, list, desc, or
exit) : desc
Enter class name of class to be described :Person

Person
char(30)   Name   not null
char(11)   DOB    null
Person   Person(String Name,String DOB)
Person   Person()
String   get_Name()
String   get_DOB()
void     set_Name(String Name)
void     set_DOB(String DOB)
Please enter DDLBoss Request (create, delete, alter, list, desc, or
exit) : exit
```

Figure 5-1: Sample SQL2Java DDL Session

5.2.3 Startup and Use of SQL2Java's DML

SQL2Java's DML module will enable the user to issue standard SQL92 commands. The first ones will presumably be INSERT requests to create instances of the classes the user just defined in the DDL component. Unlike the DDL component where the user is guided through all entries, requests in the DML component are in standard SQL. This means that to utilize the DML component the user must be familiar with SQL2Java's implementation of the SQL92 standard. For information on this standard the user may refer to the Backus-Naur Form (BNF) diagrams of SQL2Java's implementation of the SQL92 standard given in Appendix A.

As with the SQL2Java DDL component, the SQL2Java DML component has a GUI interface with a single line where requests are entered and a much larger display area where results and system information is presented. When the user has inserted and/or manipulated user objects then data can be queried. To do this the user must issue SELECT statements. The results from these statements will be produced in the display area and are the true output from the database system.

When the user is finished using the DML component or wishes to stop it in order to begin another DDL session then it is important that the DML component also be stopped gracefully using the "exit" command. Like the DDL, the DML component also performs a reorganization procedure when it is shut down. This procedure deletes data marked for

deletion from all user class data files. Use the following command to start a SQL2Java DML session:

```
java dml
```

The following figure shows a typical SQL2Java DML session in which an object of the Person class is created, updated and deleted. The object is queried after each data manipulation operation to illustrate the effects of the operation.

```
Please wait for system files to load ...Please enter DMLBoss Request
line, list, or exit): insert into Person values ('Jessica Lambert','31-
JUL-1997');
Please enter DMLBoss Request line, list, or exit): select Name, DOB
from Person;
JESSICA LAMBERT 31-JUL-1997
Please enter DMLBoss Request line, list, or exit): update Person set
Name = 'Jessica Leanne Lambert' where Name like 'Jess%';
Please enter DMLBoss Request line, list, or exit): select * from
Person;
JESSICA LEANNE LAMBERT 31-JUL-1997
Please enter DMLBoss Request line, list, or exit): delete from Person;
Please enter DMLBoss Request line, list, or exit): select * from
Person;
Please enter DMLBoss Request line, list, or exit): exit
```

Figure 5-2: Sample SQL2Java DML session

5.2.4 Operation Recovery following a system crash

To ensure Operation Atomicity, in the event of a system crash during the execution of the DML component, SQL2Java maintains a log file of the required data file modifications. These required modifications are inserted into the log file prior to changes actually being made to the data files or to the objects in memory. If a crash were to occur between the

time when SQL2Java finished writing to the log for an operation but before SQL2Java finished modifying the data in the data files, the log file processing will be completed when the SQL2Java DML is next started. If a crash were to occur during the time when SQL2Java was still adding data to the log file for an operation then that operation will not be recovered when SQL2Java's DML is next started and must be resubmitted by the user. Refer to Section 4.4 for a complete explanation of the workings of the log file mechanism.

5.3 Extensions

The list of possible extensions to the SQL2Java system is too great to completely enumerate. Just a few of those which would dramatically increase the utility of the SQL2Java system are discussed here.

5.3.1 Indexing Mechanism

SQL2Java currently has no mechanism to ensure uniqueness of objects nor does it have a mechanism to aid query performance based upon attribute values. An indexing mechanism could bring both of these to SQL2Java.

5.3.2 Backup and Recovery Mechanism

For a DBMS to be commercially viable, it must provide a transparent backup and recovery mechanism. SQL2Java does not have one at this point. Typical strategies involve redundant data file storage and full database backups to permanent storage as a means of protecting data. Redundant data storage could easily be integrated into SQL2Java but would slow performance. Full database backup can now be done manually simply by copying data files to a permanent storage device. In commercial DBMSs these processes are automated to ease recovery when database system failures occur.

5.3.3 Multi-user Capability, Locking Mechanism

SQL2Java is now a single user system. To make it multi-user would require a major overhaul of the software to integrate locking mechanisms and a security policy. For SQL2Java to be commercially viable, it would require such an extension since it is quite rare to find single user database systems being used for business applications.

As it currently stands, the SQL2Java software can be utilized by different users by having each user define their own *Setup* class. This class is used to identify platform and installation specific information such as where the data files reside. Unfortunately, the *Setup* class must currently reside in the same directory as the rest of the SQL2Java system classes. This means that although different users can use the same system to access different data sources, they cannot do it at the same time with the same software.

For a different user to use the same software but to access a different data set they would have to replace the current *Setup* class with their own or modify the one that is already there.

Further, there are currently no provisions to prevent one user from accessing another user's data files outside the normal security provisions built into the operating system.

With the integration of multi-user capability and a recognized locking mechanism such as two-phase locking [CDK94] into SQL2Java the system will be much easier to deploy as a teaching tool since the source code and data file storage could be managed centrally.

5.3.4 Utilities for building custom interfaces and reports

Another set of utilities which are quite standard with commercial database systems are tools for defining custom form interfaces to user data and for defining custom reports against user data. Recent software versions provide a Graphical User Interface (GUI) for this definition. Interfaces produced by such systems are also GUI based while report definition tools define reports which are static. GUI tools provide such things as buttons, checklists and text areas and are navigable with a mouse or other pointing device. Such tools are typically quite large in themselves and are separate pieces of code developed independently of the DBMS.

Another means of providing similar functionality is to provide a mechanism for external programs to access the data within a SQL2Java database. Standards exist for such an interface, but neither has been implemented for SQL2Java. Third party products which make use of such gateways and which allow for the definition and execution of custom interfaces and reports are available.

5.3.5 Data File Packaging Utility

SQL2Java is a portable DBMS, and as such, it can be expected that users will want to take their data files with them as they move from platform to platform. This would involve the copying of the data files and the intermediary class either across a network or onto a storage device such as a floppy disk or a tape. To reduce the amount of space required, compression technology can be utilized. If the data is being moved over a network, encryption technology may also be utilized to ensure security.

If the information is being transported on a disk or tape, complementary software would be required at the other SQL2Java installation to extract the data files and the intermediary *User_Objects* class.

5.3.6 Network Access to Data Files

With existing client-server technology it should not be necessary for a program to require that data files exist on the same machine as the system code. Currently SQL2Java requires that the user define which directory the data files reside as part of the *Setup* class. SQL2Java does not currently allow for those data files to reside elsewhere unless the path to those files can be defined as a standard path. It does not account for access to data files over the Internet, for example.

Such an extension would be quite beneficial since it would allow users to keep their data files in a single location as long as they have network access to those files. All that may be required when a user moves to another work station would be a modification to their personal *Setup* class.

Another extension continuing on the same theme is the ability to access data files from more than a single location, allowing for the distribution of the database [KS91]. This would require a modification to the DDL to specify where the user would like the data file for a new class to be located and a modification to the data loading mechanism to know where to access data files. Such data distribution is quite attractive and would allow for data sharing with other users. Data sharing would require a further extension of the SQL2Java code to allow for scheme integration.

5.4 Complimentary Technologies

5.4.1 Interfaces to other database systems and data sources

Database systems, when required, should be able to access each other's data. This access is typically achieved through an interface created by the database vendor but defined by a standards body or industry consortium. Such interfaces allow data exchange and, as a result, data within any one database is potentially made richer and more meaningful.

SQL2Java does not have any such external data access and is, therefore, isolated. Such a mechanism could be used to provide data from SQL2Java to other systems and enable SQL2Java to access and even load data from external data systems into SQL2Java.

5.4.2 A Persistence Mechanism built into Java

The persistence model implemented in SQL2Java is rudimentary. Its reliability and security is based upon that of the file system of the platform that it is running upon and the access mechanisms it has defined. A better solution would be if the Java language provided a more extensive persistence mechanism than is currently available. The persistence mechanism that Sun currently provides as part of the IO class, called Object Serialization, allows the user to write object data to a stream and to read it back from a stream. Though this mechanism does provide a means of retaining data between program

executions, it does not incorporate any security measures for retaining sensitive data nor does it have a strategy to swap data to and from persistent storage as the size of the data set gets quite large. Though the scheme put into place by Sun is useful for retaining operational data between program executions, it cannot seriously be considered as a means of persisting large amounts of data.

Sun is currently sponsoring a project at the Department of Computing Science, University of Glasgow, looking into object persistence for Java. The project is entitled “The PJava Project - Orthogonal Persistence for Java” [Spe97] and was described in section 2.1.4. The reachability approach being used in this project has also been taken by various OODBMS vendors including O₂ [BDK92] and ObjectStore [Rao94].

6. Conclusion

The evolution of DBMSs is progressing towards OODBMSs and ORDBMSs. Vendors though should be cautious not to abandon the standards based and time proven technologies of the present in their haste. SQL is a strong and easily understood query language which can be integrated with the newer technologies, thereby producing systems where the various technologies complement each other.

object-oriented techniques are quite beneficial, with respect to database systems, since they provide a means of defining a rich data framework that is easily extendible and presents a clear interface. Traditional RDBMSs are much more transparent. The approach taken with SQL2Java has been to use the strong data definition capabilities of an OOPL while providing access to the data objects through an implementation of SQL92.

The Java programming language was used to implement SQL2Java which gives the added advantage of portability. Although SQL2Java was created on a Pentium 133 running Windows95, it can be ported to any platform running a Java Virtual Machine.

SQL2Java does not provide some of the critical components of a commercial database system such as security, multi-user access or indexing. SQL2Java does demonstrate that instances of classes created using a standard OOPL can efficiently be accessed and

manipulated using a standard SQL-like interface. SQL2Java also ensures persistence of those objects between executions of the program.

SQL2Java is a prototype DBMS developed on the Java platform integrating the Object-Oriented and Relational Data Models. This integration takes the best from both models. The SQL2Java application is developed upon the Java OOPL and uses Java classes and objects to define and store system and user data. The Java implementation provides the added advantage of application portability. SQL2Java's DML is based upon the proven and accepted SQL92 relational query language standard thus making SQL2Java accessible to users familiar with that standard.

6.1 Future Work

SQL2Java is operational and ready for use as a personal database system. SQL2Java, being a prototype DBMS, does not currently contain the required components which would allow it to be used for any sort of mission critical application. A few of the components which are required include locking mechanisms to allow multi-user access, a security mechanism for specifying permissions on classes and a means of accessing the database from external programs.

Other valuable enhancements which are possible within the shorter term are also apparent. Some of these enhancements are discussed below.

An Improved Graphical User Interface (GUI)

Currently SQL2Java's DDL and DML applications have rudimentary GUI interfaces. These were built using Java's Abstract Window Toolkit (AWT) and are each made up of two separate text areas. The first is a single line where the user can enter requests. The user can also cut and paste into this area allowing the copying of a requests which have already been entered. This capability allows the user to modify and resubmit requests. The other text area is much larger and is used to present all output from SQL2Java. Error messages, explanatory text and results are all presented in this area. Although the user cannot write directly to this area, he can cut requests from this area, which are automatically written to this area when submitted, in order to paste them back to the request line.

Many improvements to this interface can be made to make it more user-friendly. AWT provides mechanisms to create graphic elements such as buttons, check boxes and check lists which could all be utilized to produce an interface which would make the use of SQL2Java easier.

An Indexing Mechanism

Currently SQL2Java does not have an indexing mechanism. All data manipulation is performed by parsing all objects of a class. This is obviously very inefficient since in many cases very few objects will meet the conditions in the condition sequence. In such cases it is advantageous to keep an index of the values contained within particular attributes of particular classes and direct references to those objects with

those values. This information can then be used by SQL2Java's DML to go directly to relevant objects instead of parsing complete classes in all cases.

Database Distribution

Many DBMSs today are required to be distributed. People are very mobile and are likely to have access to a variety of hardware and software platforms. To ease access to database data, database systems are being required to allow for the distribution of the database systems and their data. Since SQL2Java is programmed in Java it is portable between various platforms for which there is a Java Virtual Machine available. This portability makes the task of distributing SQL2Java easier. Exactly which distribution mechanism chosen is still to be determined and the complexity depends a great deal on whether or not the system has been made multi-user or not. If it has been, security must also be considered.

A Richer Data Set

Currently SQL2Java allows for only the basic data types of numbers and character strings. Production databases require many more data types such as a date data type and a raw data type which could store large data elements such as sound, picture or video data. One means of providing this capability is to extend the DDL to allow classes to contain references to objects of other classes. If this were done then potentially any data type would be possible and SQL2Java would come much closer to qualifying as a true OODBMS.

Allow custom method calls as part of the DML

Currently SQL2Java's DML allows only standard SQL operations. Although the definition of custom methods is part of the DDL, SQL2Java does not currently give the user access to call these methods. The concern is with the possibility of unwanted and harmful side effects. To restrict custom methods to only affect its object's state or to provide data back from a method call would be too restrictive. The definition of a satisfactory compromise which would allow for user flexibility when writing methods while preventing unwanted side effects is left for future analysis.

An Upgraded DML to include the capabilities defined in the SQL3 standard

This standard has not yet been approved though some ORDBMS vendors, including UniSQL and Illustra, have decided to implement it in their latest versions. The current approved SQL standard, SQL92, is to be upwardly compatible with SQL3. Modifying SQL2Java to interpret the new SQL3 standard will ensure upward compatibility between the current SQL2Java version and possible future versions.

These additions would greatly strengthen SQL2Java and make it much more acceptable to the user community. Continued improvement to SQL2Java would potentially enable it to be a commercially viable ORDBMS.

References

- [AG92] Andleigh, Prabhat K., and Gretzinger, Michael R. 1992. *Distributed Object-Oriented Data-Systems Design*. PTR Prentice-Hall, Inc.
- [Bae80] Baer, Jean-Loup. 1980. *Computer Systems Architecture*. Computer Science Press, Inc.
- [Bob97] Bobrowski, Steve. 1997. Extending Oracle8 with Objects. *Oracle Magazine*, September/October, 123-128.
- [CS94] Chorafas, Dimitris, N. and Steinmann, Heinrich. 1993. *Object-Oriented Databases*. PTR Prentice-Hall, Inc.
- [CDK94] Coulouris, George, Dollimore, Jean and Kindberg, Tim. 1994. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishers Ltd.
- [CLR92] Cormen, Thomas H., Leiserson, Charles E. and Rivest, Ronald L. 1992. *Introduction to Algorithms*. McGraw-Hill
- [CO93] Cannan, Stephen and Otten, Gerard. 1993. *SQL - The Standard Handbook*. McGraw-Hill International (UK) Limited

- [DBK92] Bancilhon, Francois, Delobel, Claude and Kanellakis, Paris. 1992. *Building an Object-Oriented Database System: The Story of O₂*. Morgan Kaufmann Publishers, Inc.
- [DDB91] Dittrich, Klaus R., Dayal, Umeshwar and Buchmann, Alejandro P. eds. 1991. *On Object-Oriented Database Systems*. Springer-Verlag
- [Hug91] Hughes, John G. 1991. *Object-Oriented Databases*. Prentice Hall
- [ID97] "Illustra DataBlades". *About Informix: Informix White Papers*.
"<http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/illustra.htm>" (5 Mar. 1997)
- [III97] "The Integration of Illustra and Informix DSA". *About Informix: Informix White Papers*.
"<http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/intrgrat.htm>". (5 Mar. 1997)
- [IIM97] "Informix and Illustra Merge to Create Universal Server". *About Informix: Informix White Papers*.
"<http://www.informix.com/corpinfo/zines/whitpprs/illustra/ifxillus.htm>". (10 Sep. 97)

- [IPD97] "Illustra Product Description". *About Informix: Informix White Papers*.
"<http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/descr.htm>". (5 Mar. 1997)
- [Jac93] Jacobson, Ivar. 1993. *Object-Oriented Software Engineering: A Use Case Driver Approach*. Addison-Wesley Publishing Company
- [JSQ97] "JSQL: Embedded SQL for Java."
http://www.oracle.com/nca/java_nca/jsql/html/jsql_ansi_proposal_7.html.
(26 Oct. 97)
- [Kim90] Kim, Won. 1990. *Introduction to Object-Oriented Databases*. The MIT Press.
- [Kim97] Kim, Won. "UniSQL: Object-Relational Database Technology."
<http://www.unisql.com/technology/papers/kim/whper.html>. (8 Oct. 1997)
- [KNN90] Kim, Won, Nicolas, Jean-Marie and Nichio, Shojiro, eds. 1990. *Deductive and Object-Oriented Databases*. Elsevier Science Publishers B.V.
- [KS91] Korth, Henry F. and Silberschatz, Abraham. 1991. *Database System Concepts*. McGraw-Hill, Inc.

- [Mey95] Meyer, Bertrand. 1995. *Object Success*. Prentice Hall International (UK) Ltd.
- [MS97] Manola, Frank and Sutherland, Jeff. "SQL3 Object Model". *National Committee for Information Technology, Technical Committee H7, Object Model Features Matrix*. 30 June, 1997. <http://www.objs.com/x3h7/sql3.htm> (26 Oct. 1997)
- [ODM97] "ODMG 2.0: Book Extract",
<http://www.odmg.org/standard/odmgbookextract.htm>. (29 Oct. 1997)
- [OOO98] "Oracle8: Objects Option",
<http://www.oracle.com/st/products/options/objects>. (3 Jan. 1998)
- [Ora98] "Oracle8", <http://www.oracle.com/st/products/uds/oracle8/index.html#matrix>. (3 Jan. 1998)
- [OS98] "Object Serialization",
<http://www.java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html>. (28 Jan, 1998)
- [OUF97] "Objects and the Universal Frontier", *Database Programming and Design*:

- Categories to Watch*. <http://www.dbpd.com/seconddz.htm>. (6 June 1997)
- [Pat97] Patel, Pratik R. "JDBC - Java DataBase Connectivity."
<http://sunsite.unc.edu/prpatel/jdbc/2.html>. (26 Oct. 97)
- [PSP97] Patel, Dilip, Sun, Yuan and Patel, Shushma eds. 1997. 96 OOIS: 1996
International Conference on Object Oriented Information Systems. Springer-Verlag
- [Rao94] Rao, Bindu R. 1994. *Object-Oriented Databases: Technology, Applications, and Products*. McGraw-Hill, Inc.
- [Som92] Sommerville, Ian. 1992. *Software Engineering*. Addison-Wesley Publishing Ltd.
- [SO97] "Standard Overview", *Object Database Management Group*.
"<http://www.odmg.org/standard/standardoverview.htm>" (29 Oct. 1997)
- [Spe97] Spence, Susan. "The PJava Project: Orthogonal Persistence for Java" 25
Sept. 1997. <http://www.dcs.gla.ac.uk/pjava/> (27 Oct. 1997)
- [Terr86] Terry, Patrick D. 1986. *Programming Language Translation: A Practical Approach*. Addison-Wesley Publishers Ltd.

[TPD97] "The Problem of Database Optimization", *About Informix: Informix White Papers*.

<http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/opt.htm> (5 Mar. 1997)

Appendix A: SQL2Java DML BNF Syntax Diagrams

This appendix gives the syntax of those portions of the SQL2Java DML which are modeled after the SQL92 DML syntax in Backus-Naur Form (BNF) [Terr86] [CO93]. A number of functions which are not part of the standard are listed at the end.

Definitions without values to the right of the ::= symbol indicate assignments which are platform specific.

The following symbols are used in the diagrams which follow.

Symbol	Meaning
< >	Angle brackets delimit character strings that are the names of syntactic elements, the non-terminal symbols of the SQL language.
::=	The definition operator. In each rule this symbol separates what is being defined, which appears to the left of the symbol, from the definition, which appears to its right.
[]	Square bracket indicate optional elements in a formula.
{ }	Braces group elements of a formula. The portion of the formula within the braces must be explicitly specified.
	The alternative operator.
...	The ellipsis indicates that the preceding element of a formula may be

	repeated.
--	-----------

Figure A-6-1: BNF Symbols

<actual identifier> ::= <regular identifier>

<as clause> ::= [AS] <column name>

<between predicate> ::=

 <row value constructor> [NOT] BETWEEN

 <row value constructor> AND

 <row value constructor>

<boolean factor> ::=

 [NOT] <boolean test>

<boolean primary> ::=

 <predicate>

 | (<search condition>)

<boolean term> ::=

 <boolean factor>

 | <boolean term> AND <boolean factor>

<boolean test> ::= <boolean primary>

<char length expression> ::=

LENGTH (<string value expression>)

<character primary> ::=

<common primary> | <string value function>

<character representation> ::=

<nonquote character>

| <quote symbol>

<character string literal> ::=

' [<character representation> ...] '

<character substring function> ::=

(SUBSTR <character value expression>

, <start position>

[, <string length>])

<character translation function> ::=

TRANSLATE (<character value expression>

, <character value expression>

, <character value expression>)

<character value expression> ::=

<concatenation>

| <character primary>

<character value function> ::=

<character substring function>

| <fold>

| <character translation function>

| <trim function>

<column name> ::= <identifier>

<column name list> ::=

<column name> [{ , <column name> } ...]

<column reference> ::= [<qualifier> .] <column name>

<common primary> ::=

<unsigned value specification>

| <column reference>

| <set function specification>

- | <scalar subquery>
- | (<value expression>)

<comparison operator> ::=

- =
- | !=
- | <
- | >
- | <=
- | >=

<comparison predicate> ::=

<row value constructor>

<comparison operator> <row value constructor>

<concatenation> ::=

<character value expression>

<concatenation operator>

<character primary>

<concatenation operator> ::= ||

<correlation name> ::= <identifier>

<delete statement: searched> ::=

DELETE [FROM] <table name>
[WHERE <search condition>]

<describe statement> ::=

DESC <table name>

<derived column> ::= <value expression> [<as clause>]

<derived column list> ::= <column name list>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<direct select statement: multiple rows> ::=

<simple table> [<order by clause>]

<direct SQL data statement> ::=

<delete statement: searched>

| <direct select statement: multiple rows>

| <insert statement>

| <update statement: searched>

<direct SQL statement> ::=

 <direct SQL data statement> ;

<exact numeric literal> ::=

 <unsigned integer> [. [<unsigned integer>]]

 | . <unsigned integer>

<exists predicate> ::= EXISTS <table subquery>

<factor> ::=

 [+|-] <numeric primary>

<fold> ::=

 { UPPER | LOWER } (<character value expression>)

<from clause> ::=

 FROM <table reference>

 [{ , <table reference> } ...]

<general literal> ::= <character string literal>

<general set function> ::=

 { AVG | MAX | MIN | SUM | COUNT }

(<value expression>)

<group by clause> ::=

GROUP BY <grouping column reference list>

<grouping column reference> ::= <column reference>

<grouping column reference list> ::=

<grouping column reference>

[{ , <grouping column reference> } ...]

<having clause> ::= HAVING <search condition>

<identifier> ::= <actual identifier>

<identifier body> ::=

<identifier start>

[{ _ | <identifier part> } ...]

<identifier part> ::= <identifier start> | <digit>

<identifier start> ::=

<in predicate> ::=

 <row value constructor>

 [NOT] IN <in predicate value>

<in predicate value> ::=

 <table subquery>

 | (<in value list>)

<in value list> ::=

 <value expression> { , <value expression> } ...

<insert column list> ::= <column name list>

<insert columns and source> ::=

 [(<insert column list>)] <query expression>

 | VALUES (<value expression> [{ , <value expression> } ...])

<insert statement> ::=

 INSERT INTO <table name>

 <insert columns and source>

<length> ::= <unsigned integer>

<length expression> ::= <char length expression>

<like predicate> ::=

 <match value> [NOT] LIKE <pattern>

<match value> ::= <character value expression>

<nonquote character> ::=

<numeric primary> ::=

 <common primary> | <numeric value function>

<numeric value expression> ::=

 <term>

 | <numeric value expression> + <term>

 | <numeric value expression> - <term>

<numeric value function> ::=

 <position expression>

 | <length expression>

<object column> ::= <column name>

<order by clause> ::=

ORDER BY <sort specification list>

<ordering specification> ::= ASC | DESC

<pattern> ::= <character value expression>

<position expression> ::=

INSTR (<character value expression> , <character value expression>

[, <unsigned integer> [, <unsigned integer>]])

<precision> ::= <unsigned integer>

<predicate> ::=

<comparison predicate>

| <between predicate>

| <in predicate>

| <like predicate>

| <qualified comparison predicate>

| <exists predicate>

<qualified identifier> ::= <identifier>

<qualified name> ::= <qualified identifier>

<qualifier> ::=

 <table name>

 | <correlation name>

<qualified comparison predicate> ::=

 <row value constructor> <comparison operator> <quantifier>

 <table subquery>

<quantifier> ::= ALL | ANY

<query expression> ::= <simple table>

<query specification> ::=

 SELECT <select list>

 <table expression>

<quote> ::= ' '

<quote symbol> ::= <quote><quote>

<regular identifier> ::= <identifier body>

<row subquery> ::= <subquery>

<row value constructor> ::=

 <row value constructor element>

 | (<row value constructor list>)

 | <row subquery>

<row value constructor element> ::= <value expression>

<row value constructor list> ::=

 <row value constructor element>

 [{ , <row value constructor element> } ...]

<scalar subquery> ::= <subquery>

<search condition> ::=

 <boolean term>

 | <search condition> OR <boolean term>

<select list> ::=

 *

 | <select sublist> [{ , <select sublist> } ...]

<select statement: single row> ::=

SELECT <select list>
 <table expression>

<select sublist> ::=

<derived column>
| <qualifier>.*

<set clause> ::=

<object column> = <update source>

<set clause list> ::=

<set clause> [{ , <set clause> } ...]

<set function specification> ::=

COUNT(*)
| <general set function>

<signed integer> ::= [+|-] <unsigned integer>

<simple table> ::=

<query specification>

| <table value constructor>

<sort key> ::=

<column name> | <unsigned integer>

<sort specification> ::=

<sort key> [<ordering specification>]

<sort specification list> ::=

<sort specification>

[{ , <sort specification> } ...]

<space> ::=

<SQL special character> ::=

. | ; | (|) | , | % | _ | ' | " | + | -

| * | / | <space> | < | > | ! | = | &

| <vertical bar>

<start position> ::= <numeric value expression>

<string length> ::= <numeric value expression>

<string value expression> ::= <character value expression>

<string value function> ::= <character value function>

<subquery> ::= (<simple table>)

<table expression> ::=

 <from clause>

 [<where clause>]

 [<group by clause>]

 [<having clause>]

<table name> ::= <qualified name>

<table reference> ::=

 <table name> [[AS] <correlation name>]

<table subquery> ::= <subquery>

<table value constructor> ::=

 VALUES <table value constructor list>

<table value constructor list> ::=

<row value constructor>

[{ , <row value constructor> } ...]

<term> ::=

<factor>

| <term> * <factor>

| <term> / <factor>

<trim character> ::= <character value expression>

<trim function> ::=

LTRIM (<trim character> [, <trim source>])

| RTRIM (<trim character> [, <trim source>])

<trim source> ::= <character value expression>

<unsigned integer> ::= <digit> ...

<unsigned literal> ::=

<unsigned numeric literal>

| <general literal>

<unsigned numeric literal> ::=

<exact numeric literal>

<unsigned value specification> ::=

<unsigned literal>

<update source> ::= <value expression>

<update statement: searched> ::=

UPDATE <table name>

SET <set clause list>

[WHERE <search condition>]

<value expression> ::=

<numeric value expression>

| <string value expression>

<vertical bar> ::= |

<where clause> ::= WHERE <search condition>

- The functions CEIL, FLOOR, MOD, POWER, ROUND, SIGN, SQRT, TRUNC, INITCAP, LPAD, REPLACE, RPAD, TO_CHAR, TO_NUMBER, GREATEST and

LEAST are not part of the SQL92 standard and are thus not included in the BNF diagrams above.

Appendix B: *User_Objects* sample class definition

The class definition given below is that of the *User_Objects* class after the *Person* class has been defined for the university application.

```
import java.io.*;
import java.util.*;
import java.lang.*;
class user_objects {

    Person first_Person = null;
    Person current_Person = null;
    Person previous_Person = null;
    Person last_Person = null;
    user_objects() {
        first_Person = load_Person();
        current_Person = first_Person;
    }

    private Person load_Person() {
        Person    first_object = null;
        Person    next_object = null;
        Person    current_object = null;
        String X1 = "";
        String X2 = "";
        try {
            RandomAccessFile datafile = new
RandomAccessFile("Person.data", "rw");
            datafile.seek(1);
            while(datafile.length() > datafile.getFilePointer()+1) {
                int file_position = (int)datafile.getFilePointer();
                X1 = datafile.readLine();
                X2 = datafile.readLine();
                if(X1.charAt(0) == '0') {
                    X1 = X1.substring(1,X1.length());
                    next_object = new Person(
                        X1.trim()
                        ,X2.trim()
                    );
                    next_object.file_position = file_position;
                    if(first_object == null) {
                        first_object = next_object;
                    }
                    else {
                        current_object.next_Person = next_object;
                    }
                    current_object = next_object;
                }
            }
            datafile.close();
        }catch(IOException e) {
            System.out.println("IOException : " + e);
        }
    }
}
```

```

    }
    return(first_object);
}

public void constructor_broker(String Method_Name, String
store_string,String PARAMETER1,String PARAMETER2) {
    if(Method_Name.equals("Person") == true) {
        try {
            RandomAccessFile datafile = new
RandomAccessFile("Person.data","rw");
            int filesize = (int)datafile.length();
            datafile.seek(filesize);
            datafile.writeBytes(store_string);
            datafile.close();
            Person dummy_object = new Person(PARAMETER1,PARAMETER2);
            dummy_object.file_position = filesize;
            if(this.first_Person == null) {
                this.first_Person = dummy_object;
            }
            else {
                Person current_object = this.first_Person;
                while(current_object.next_Person != null) {
                    current_object = current_object.next_Person;
                }
                current_object.next_Person = dummy_object;
            }
            last_Person = dummy_object;
        } catch (IOException e) {
            System.err.println("IOException: " + e);
        }
    }
}

public void constructor_broker(String Method_Name, String
store_string) {
    if(Method_Name.equals("Person") == true) {
        try {
            RandomAccessFile datafile = new
RandomAccessFile("Person.data","rw");
            int filesize = (int)datafile.length();
            datafile.seek(filesize);
            datafile.writeBytes(store_string);
            datafile.close();
            Person dummy_object = new Person();
            dummy_object.file_position = filesize;
            if(this.first_Person == null) {
                this.first_Person = dummy_object;
            }
            else {
                Person current_object = this.first_Person;
                while(current_object.next_Person != null) {
                    current_object = current_object.next_Person;
                }
                current_object.next_Person = dummy_object;
            }
            last_Person = dummy_object;
        } catch (IOException e) {
            System.err.println("IOException: " + e);
        }
    }
}

```

```

    }
}
public String String_broker(String Method_Name, String Class_Name) {
    if((Method_Name.equals("get_Name") == true) &&
        (Class_Name.equals("Person") == true)) {
        return(current_Person.get_Name());
    }
    if((Method_Name.equals("get_DOB") == true) &&
        (Class_Name.equals("Person") == true)) {
        return(current_Person.get_DOB());
    }
    return("");
}
public void void_broker(String Method_Name, String Class_Name,String
PARAMETER1) {
    if((Method_Name.equals("set_Name") == true) &&
        (Class_Name.equals("Person") == true)) {
        last_Person.set_Name(PARAMETER1);
    }
    if((Method_Name.equals("set_DOB") == true) &&
        (Class_Name.equals("Person") == true)) {
        last_Person.set_DOB(PARAMETER1);
    }
}
public int int_broker(String Method_Name, String Class_Name) {
    return(0);
}
public double double_broker(String Method_Name, String Class_Name) {
    return(0.0);
}
public void void_broker(String Method_Name, String Class_Name, int
PARAMETER1) {
}
public void void_broker(String Method_Name, String Class_Name,
double PARAMETER1) {
}
public boolean reset_Person() {
    if(first_Person != null) {
        current_Person = first_Person;
        previous_Person = null;
        return(true);
    }
    else {
        current_Person = null;
        previous_Person = null;
        return(false);
    }
}
public boolean advance_Person(int orig_filesize) {
    previous_Person = current_Person;
    if(current_Person.next_Person != null) {
        current_Person = current_Person.next_Person;
        if(orig_filesize < current_Person.file_position + 1) {
            return(false);
        }
    }
    else {
        return(true);
    }
}

```

```
    }
  }
  else {
    previous_Person = null;
    current_Person = first_Person;
    return(false);
  }
}

public boolean delete_Person(int orig_filesize) {
  try {
    RandomAccessFile datafile = new
RandomAccessFile("Person.data","rw");
    datafile.seek(current_Person.file_position);
    datafile.writeBytes("1");
    datafile.seek(0);
    datafile.writeBytes("1");
    datafile.close();
    if(current_Person == first_Person) {
      if(current_Person.next_Person != null) {
        first_Person = current_Person.next_Person;
        current_Person = current_Person.next_Person;
        if(orig_filesize < current_Person.file_position + 1) {
          return(false);
        }
      }
      else {
        return(true);
      }
    }
    else {
      first_Person = null;
      current_Person = null;
      return(false);
    }
  }
  else {
    if(current_Person.next_Person != null) {
      previous_Person.next_Person =
current_Person.next_Person;
      current_Person = current_Person.next_Person;
      if(orig_filesize < current_Person.file_position + 1) {
        return(false);
      }
    }
    else {
      return(true);
    }
  }
  else {
    current_Person = previous_Person;
    last_Person = previous_Person;
    previous_Person.next_Person = null;
    return(false);
  }
}
} catch (IOException e) {
  System.err.println("IOException: " + e);
}
return(false);
```



```
    }
    public boolean master_reset(String class_name) {
        if(class_name.equals("Person") == true) {
            return(reset_Person());
        }
        return(false);
    }
    public boolean master_advance(String class_name, int orig_filesize)
{
    if(class_name.equals("Person") == true) {
        return(advance_Person(orig_filesize));
    }
    return(false);
}
    public boolean master_delete(String class_name, int orig_filesize) {
        if(class_name.equals("Person") == true) {
            return(delete_Person(orig_filesize));
        }
        return(false);
    }
    public int get_fp(String class_name) {
        if(class_name.equals("Person") == true) {
            return(current_Person.file_position);
        }
        return(0);
    }
}
```

Appendix C: Meta Class definitions

This appendix contains the definitions of the main meta classes including the constructors and methods that each contain.

```
class MetaClass {

    /* Attributes */

    String Class_Name;
    MetaAttribute first_attribute;
    MetaMethod first_method;
    MetaClass super_class = null;
    boolean final_indicator = false;
    boolean public_indicator = false;
    int file_position;
    static Setup setup_data = new Setup();
    static int object_data_length = 45 + setup_data.newline_length;
    static boolean datafile_created = false;
    static DataFile MetaClass_datafile = new
DataFile("MetaClass_DD",object_data_length);
    static int next_insertion_point = 1;
    String super_Class_Name = "";

    /* Constructors */

    public MetaClass(String class_name, MetaClassReference first_class,
DDLBOSS boss) {
        ...
    }
    public MetaClass() {
        ...
    }

    /* Methods */

    private MetaMethod create_default_methods(String Class_Name, int
more) {
        ...
    }
    public void recreate_method_data(MetaClass true_class) {
        ...
    }
    public String get_Class_Name() {
        ...
    }
    public int get_file_position() {
        ...
    }
    public DataFile get_datafile() {
        ...
    }
    private void save_metadata() {
```

```

    ...
}
public String get_class() {
    ...
}
public void make_class() {
    ...
}
private void make_ref_class() {
    ...
}
public void make_data_file() {
    ...
}
public void delete_MetaClass() {
    ...
}
public String desc_class() {
    ...
}
}

class MetaAttribute {

    /* Attributes */

    String    Attribute_Name;
    String    Attribute_Type;
    int       Attribute_Size;
    boolean   Attribute_Null;
    String    Attribute_Value;
    MetaAttribute next_attribute;
    String    Class_Name;
    boolean   public_indicator = false;
    boolean   private_indicator = false;
    boolean   extended_indicator = false;
    int       file_position;
    static Setup setup_data = new Setup();
    static int object_data_length = 151 + setup_data.newline_length;
    static boolean datafile_created = false;
    static DataFile MetaAttribute_datafile = new
DataFile("MetaAttribute_DD",object_data_length);
    static int datafile_baseline = 1;
    static int next_insertion_point = 1;
    static boolean dont_move_baseline = false;
    static int class_baseline = 1;
    static String last_class = "";

    /* Constructors */

    public MetaAttribute(String class_name, String attribute_name,
MetaAttribute first_attribute, DDLBoss boss, boolean ask_for_more) {
        ...
    }
    public MetaAttribute(String class_name, String dummy) {
        ...
    }
}

```

```

    public MetaAttribute(String Attribute_Name, String Attribute_Type,
        int Attribute_Size,
            boolean Attribute_Null, String Attribute_Value,
        String Class_Name,
            boolean Public_Indicator, boolean Private_Indicator,
            boolean Extended_Indicator, boolean
write_to_datafile) {
    ...
}

/* Methods */

private void save_metadata(int more_attributes) {
    ...
}
public String get_attributes() {
    ...
}
public int print_attributes(DataOutputStream dos) {
    ...
}
public String get_attribute_name() {
    ...
}
public String get_attribute_value() {
    ...
}
public MetaAttribute get_next_attribute() {
    ...
}
public String get_attribute_type() {
    ...
}
public int get_file_position() {
    ...
}
public DataFile get_datafile() {
    ...
}
public void delete_MetaAttribute() {
    ...
}
}

class MetaMethodAttribute {

    /* Attributes */

    String    Attribute_Name;
    String    Attribute_Type;
    int       Attribute_Size;
    String    Attribute_Value;
    MetaMethodAttribute next_attribute;
    String    Method_Name;
    String    Class_Name;
    int file_position;
    static Setup setup_data = new Setup();

```

```

static int object_data_length = 177 + setup_data.newline_length;
static boolean datafile_created = false;
static DataFile MetaMethodAttribute_datafile = new
DataFile("MetaMethodAttribute_DD",object_data_length);
static int next_insertion_point = 1;
static int datafile_baseline = 1;
static boolean dont_move_baseline = false;
static int method_baseline = 1;
static String last_class = "";
static String last_method = "";

/* Constructors */

public MetaMethodAttribute(String class_name, String method_name,
boolean reorg, DDLBoss boss) {
...
}
public MetaMethodAttribute(String class_name, String method_name,
String dummy) {
...
}

/* Methods */

private void save_metadata(int more_attributes) {
...
}
public String get_attributes() {
...
}
public int print_attributes(DataOutputStream dos) {
...
}
public String get_attribute_type() {
...
}
public String get_attribute_name() {
...
}
public String get_attribute_value() {
...
}
public MetaMethodAttribute get_next_attribute() {
...
}
public void delete_MetaMethodAttribute() {
...
}
}

class MetaParameter {

/* Attributes */

String    Parameter_Name;
String    Parameter_Type;

```

```

    MetaParameter next_parameter;
    String Method_Name;
    String Class_Name;
    int file_position;
    static Setup setup_data = new Setup();
    static int object_data_length = 92 + setup_data.newline_length;
    static boolean datafile_created = false;
    static DataFile MetaParameter_datafile = new
DataFile("MetaParameter_DD",object_data_length);
    static int next_insertion_point = 1;
    static int datafile_baseline = 1;
    static boolean dont_move_baseline = false;
    static int method_baseline = 1;
    static String last_class = "";
    static String last_method = "";

    /* Constructors */

    public MetaParameter(String class_name, String method_name, boolean
reorg, DDLBoss boss) {
        ...
    }
    public MetaParameter(String class_name, String method_name, String
dummy) {
        ...
    }
    public MetaParameter(String class_name, String method_name, String
parameter_name, String parameter_type, int more) {
        ...
    }

    /* Methods */

    private void save_metadata(int more_parameters) {
        ...
    }
    public String get_parameters() {
        ...
    }
    public void print_parameters(DataOutputStream dos) {
        ...
    }
    public void delete_MetaParameter() {
        ...
    }
}

class MetaMethod {

    /* Attributes */

    String Method_Name;
    String Method_Type;
    MetaParameter First_Parameter;
    MetaMethodAttribute First_Attribute;
    MetaMethodLine First_Method_Line;
    MetaMethod next_method;

```

```

String Class_Name;
boolean processed = false;
boolean public_indicator = false;
boolean private_indicator = false;
boolean extended_indicator = false;
int file_position;
static Setup setup_data = new Setup();
static int object_data_length = 68 + setup_data.newline_length;
static boolean datafile_created = false;
static DataFile MetaMethod_datafile = new
DataFile("MetaMethod_DD",object_data_length);
static int next_insertion_point = 1;
static int datafile_baseline = 1;
static boolean dont_move_baseline = false;
static int class_baseline = 1;
static String last_class = "";

/* Constructors */

public MetaMethod(String class_name,boolean reorg,DDLBoss boss) {
...
}
public MetaMethod(String class_name, String dummy) {
...
}
public MetaMethod(String class_name, String method_name, String
method_type) {
...
}

/* Methods */

public void save_metadata(int more_methods) {

...
}
public void print_methods(DataOutputStream dos) {
...
}
public void delete_MetaMethod() {
...
}
public String get_method_name() {
...
}
public MetaMethod get_next_method() {
...
}
public String get_method_type() {
...
}
public int get_file_position() {
...
}
public DataFile get_datafile() {
...
}

```

```

    }
}

class MetaMethodLine {

    /* Attributes */

    String Method_Line;
    MetaMethodLine next_method_line;
    String Class_Name;
    String Method_Name;
    int line_number;
    int file_position;
    static Setup setup_data = new Setup();
    static int object_data_length = 136 + setup_data.newline_length;
    static boolean datafile_created = false;
    static DataFile MetaMethodLine_datafile = new
    DataFile("MetaMethodLine_DD",object_data_length);
    static int next_insertion_point = 1;
    static int datafile_baseline = 1;
    static boolean dont_move_baseline = false;
    static int method_baseline = 1;
    static String last_class = "";
    static String last_method = "";

    /* Constructors */

    public MetaMethodLine(String class_name, String method_name, int
line_num, boolean reorg, DDLBoss boss) {
        ...
    }
    public MetaMethodLine(String class_name, String method_name, String
dummy) {
        ...
    }
    public MetaMethodLine(String class_name, String method_name, String
method_line, int line_number, int more) {
        ...
    }

    /* Methods */

    private void save_metadata(int more_method_lines) {
        ...
    }
    public void print_method_lines(DataOutputStream dos) {
        ...
    }
    public void delete_MetaMethodLine() {
        ...
    }
}

class DataFile {

    /* Attributes */

```



```
String Class_Name;
boolean requires_reorg;
int object_length;
int num_of_objects;
int num_marked_for_deletion;
int next_insertion_point;
int current_position = 1; // To jump over the reorg flag
Setup setup_data = new Setup();

/* Constructors */

public DataFile(String class_name, int object_len) {
    ...
}

/* Methods */

public void store_object(String object_data) {
    ...
}
public String retrieve_next_object() {
    ...
}
public String retrieve_object(int file_position) {
    ...
}
public void delete_object(int file_position) {
    ...
}
public void reorg(DDLBoss boss) {
    ...
}
public void mark_for_reorg() {
    ...
}
public void reset_byte(int file_position, String new_value) {
    ...
}
public int get_file_size() {
    ...
}
}
```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved