Fundamentals of Database Systems

Laboratory Manual[1]

Rajshekhar Sunderraman

Georgia State University

August 2010

[1] To accompany Elmasri and Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley, 2010.

Preface

This laboratory manual accompanies the popular database textbook *Elmasri and Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley, 2010*. It provides supplemental materials to enhance the practical coverage of concepts in an introductory database systems course. The material presented in this laboratory manual complement many of the chapters of the Elmasri/Navathe text typically covered in most introductory database systems courses.

Chapter Mappings

The laboratory manual consists of 8 chapters and the following table shows the mapping to the chapters in the Elmasri/Navathe textbook:

Laboratory Manual Chapter	Elmasri/Navathe 6th Edition Chapter(s)
Chapter 1	Chapters 7, 8, and 9
Chapter 2	Chapters 3, 6, and 26
Chapter 3	Chapters 4, 5, and 13
Chapter 4	Chapters 4, 5, and 14
Chapter 5	Chapters 15 and 16
Chapter 6	Chapter 11
Chapter 7	Chapter 12
Chapter 8	Chapters 13 and 14

Chapter 1 presents ERWin, a popular data modeling software that allows database designers to represent Entity-Relationship diagrams and automatically generate relational SQL code to create the database in one of several commercial relational database management systems such as Oracle or Microsoft SQLServer. The material presented in this chapter is tutorial in nature and covers the COMPANY database design of the Elmasri/Navathe text in detail.

Chapter 2 presents three interpreters that can be used to execute queries in Relational Algebra, Domain Relational Calculus, and Datalog. These interpreters are part of a Java package that includes a rudimentary database engine capable of storing relations and able to perform basic relational algebraic operations on these relations. It is hoped that these interpreters will allow the student to get a better understanding of abstract query languages.

Chapter 3 presents techniques to interact and program with Oracle database management system. A popular data-loading tool for Oracle databases called SQL Loader is introduced and the COMPANY database of the Elmasri/Navathe text is extended with additional data to make it more interesting to program with. Programming applications that access Oracle databases is then introduced in Java using the JDBC interface. Several non-trivial example programs are discussed.

Chapter 4 covers MySQL database management system, a popular open source database system that is increasing used by small and medium sized organizations. Programming Web applications in PhP that accesses MySQL databases is introduced with a complete database browser application for the COMPANY database as well as a complete Online Address Book application.

Chapter 5 introduces a Prolog-based toolkit for relational database design. The toolkit, called Database Designer (DBD), allows the student to work with numerous concepts and algorithms that deal with functional dependency theory and data normalization. The student may use DBD to verify answers to many questions related to functional dependency theory and normalization algorithms.

Chapter 6 presents programming with a popular open source Object-Oriented Database Management system, db4o. Creating and populating objects in db4o is covered as well various methods to query and retrieve data from the object-oriented database is introduced. Db4o supports various object-oriented programming interfaces, but the Java interface is covered in the lab manual.

Chapter 7 presents XML and its related technologies. Query languages XPath and XQuery are covered as well as schema specification language XML Schema. Numerous examples are presented including a complete specification of the company database in XML along with a XML Schema.

Chapter 8 presents several semester-long projects for students in introductory database courses to complete. These projects may be implemented in Java, PhP or any other favorite programming language and may access Oracle, MySQL or any other relational database management system.

Code

The laboratory manual comes with all the code and data presented in the different chapters. The software for the relational query interpreters as well as the database designer (DBD) also accompanies the laboratory manual.

Software

The software systems discussed and used in the laboratory manual are ERWin from Computer Associates, Oracle DBMS from Oracle, and MySQL, PhP, db4o, and SWI-Prolog from open source. Both Computer Associates and Oracle have educational pricing for their software and we expect the individual universities and colleges that use this laboratory manual to provide the software for use by their students.

Rajshekhar Sunderraman
Atlanta, Georgia
August 2010

Contents

CHAPTER 1

ER Modeling Tools

This chapter introduces ERWin, a popular data-modeling tool used in the industry. ERWin is a powerful tool that allows database designers to enter their Entity Relationship (ER) diagrams in a graphical form and produce physical database designs for popular relational database management systems such as Oracle and Microsoft SQLServer.

The use of ERWin is illustrated in this chapter using the ER schema diagram for the COMPANY database shown in Figure 7.2 of the Elmasri/Navathe text.

1.1 Starting with ERWin

The ERWin Data Modeler workspace is shown in Figure 1.1.

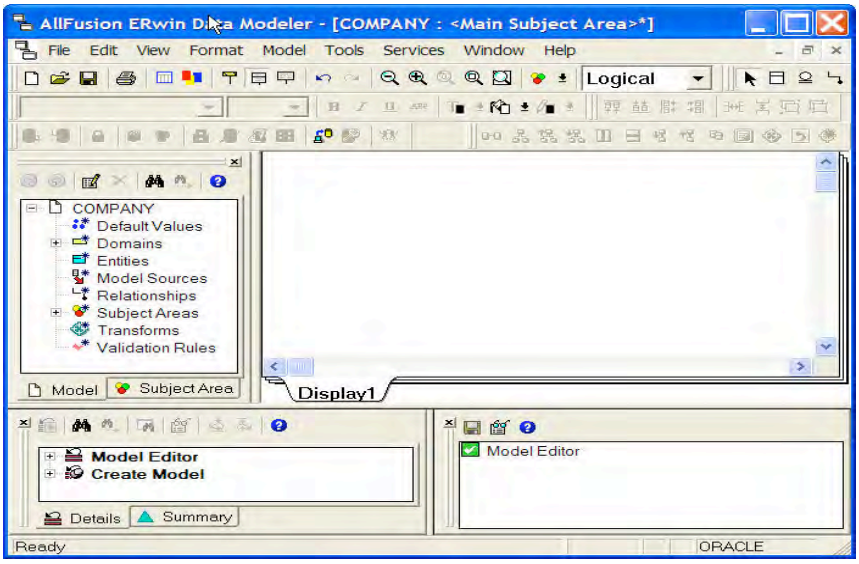

Figure 1.1: ERWin Data Modeler Workspace

The top part of the workspace consists of Menu and Toolbars. The middle part of the workspace consists of two panes: the *model explorer panel* on the left providing a text based view of the data model and the *diagram window panel* on the right providing a graphical view of the data model. The lower part of the workspace consists of two panes: the *action log panel* on the left that displays a log of all changes made to the data model under design and the *advisories panel* that displays messages associated with the actions performed on the data model under design.

ERWin supports three model types for use by the database designer:

1. Logical: A conceptual model that includes entities, relationships, and attributes. This model type is essentially at the ER modeling level.

2. Physical: A database specific model that contains relational tables, columns and associated data types.
3. Logical/Physical: A single model that includes both the conceptual level objects as well as physical level tables. In this chapter we will use this model type.

To create a model in ERWin, one should launch the program and then choose the "New" option from the File menu. The Create Model dialog appears as shown in Figure 1.2.

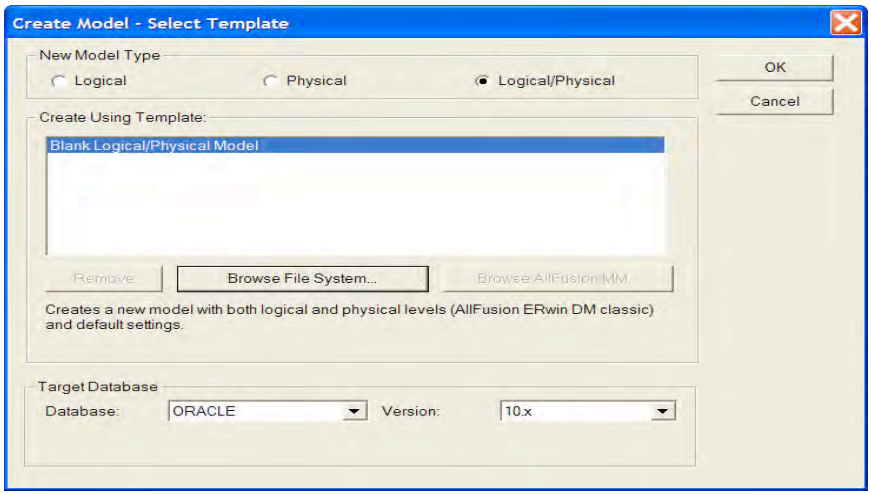

Figure 1.2: Create Model dialog window

In this dialog window, the user should choose the type of model. Typically the Logical/Physical model type should be chosen if the final goal is to produce a relational design for the database. The target database may also be chosen. In this case, Oracle 10.x version is chosen as the target database. In a future step, we will illustrate how ERWin can be used to generate SQL code to create the database objects in Oracle 10.x database.

The workspace for the new model will be populated by the system with a default name of Model_n. This name may be changed in the model explorer pane by right clicking the model name and choosing the Properties option. This brings up a new window in which the name and other properties of the model may be changed. Besides changing the model name, the "Transform" options should be checked. This would allow for many-to-many relationships to be transformed correctly into separate relational tables in the physical model. In addition any sub-type/super-type relationships will also be transformed correctly in the physical model.

1.2 Adding Entity Types

To add an entity type to the database design, the user may either right click the "Entities" entry in the model explorer pane and choose "New" or choose the "Entity" icon in the Menus and Toolbars section of the workspace and click in the diagram window panel. An entity box shows up in the diagram window panel with a default entity name (E/n) that can be changed either in the diagram window panel or in the model explorer pane. Figure 1.3 shows the addition of the EMPLOYEE entity type in the COMPANY database.

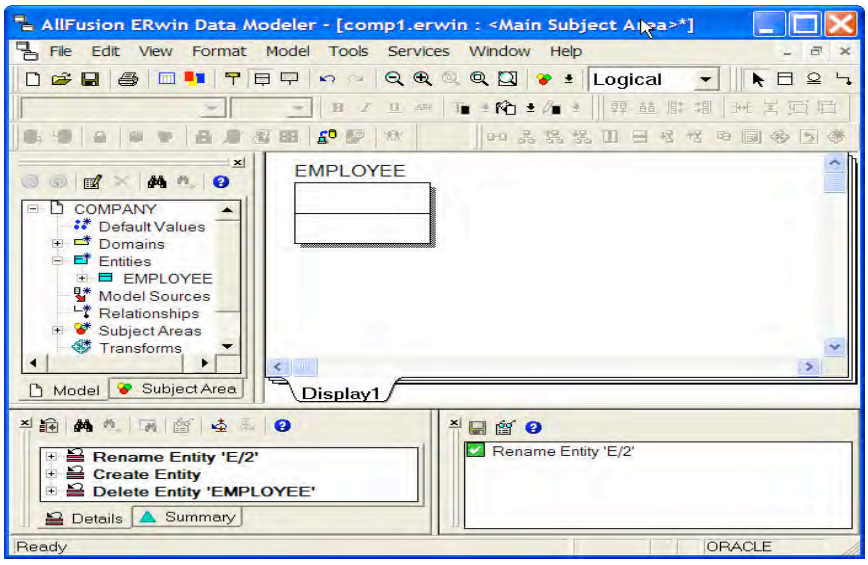

Figure 1.3: Add EMPLOYEE entity to the COMPANY database

To add attributes to the EMPLOYEE entity type, the user may right click within the EMPLOYEE entity box in the diagram window panel and choose "Attributes". This brings up a separate window using which new attributes may be added. The attribute window is shown in Figure 1.4.

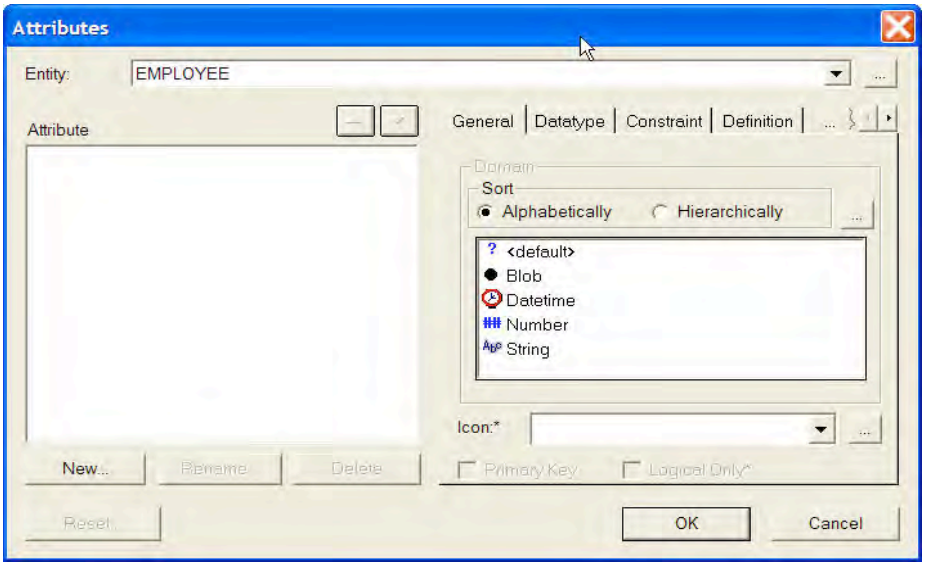

Figure 1.4: Attribute Window

The user may now add attributes one at a time by clicking the "New" button. A separate window pops up as shown in Figure 1.5.

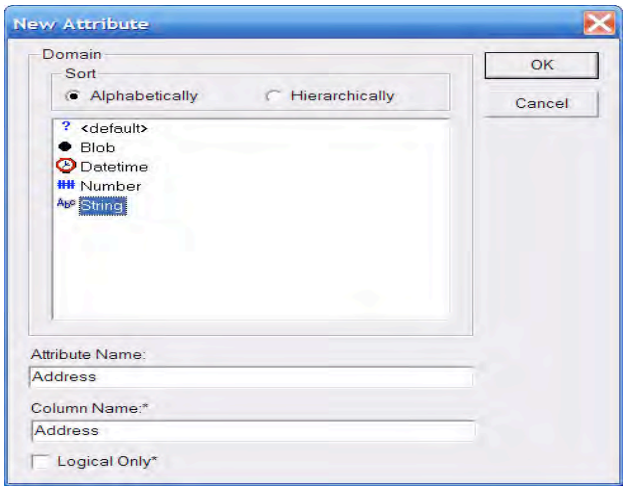

Figure 1.5: New Attribute Window

The user may choose an appropriate Domain (data type) and enter the Attribute Name and click OK. The data type may be further refined in the Attribute Window by choosing the Datatype tab and entering a precise data type. The user may also choose to designate this attribute as a primary key by selecting this option in the Attribute window.

After adding a few attributes to the EMPLOYEE entity type the Attributes window is shown in Figure 1.6.

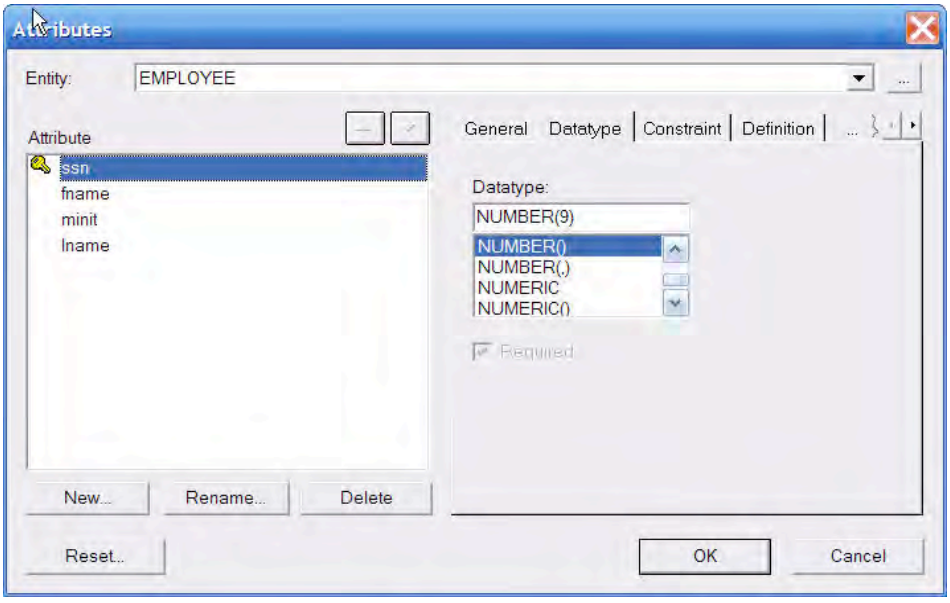

Figure 1.6: Attribute Window with four attributes

In this way, we can create each of entity types: EMPLOYEE, DEPARTMENT, PROJECT, and DEPENDENT for the COMPANY database.

Weak Entity Sets

By default any entity type created as discussed so far is classified as an independent entity type. ERWin will classify an entity type as "weak" as soon as it participates in an identifying relationship. For example, the entity type DEPENDENT will be classified as "weak" in a subsequent step when we add the identifying relationship from EMPLOYEE to DEPENDENT in the next section. Weak entity types are denoted by rounded rectangles in the diagram window panel.

Multi-Valued Attributes

Multi-valued attributes such as the locations attribute for the DEPARTMENT entity type cannot be modeled easily with ERWin. To handle such attributes, a separate entity type LOCATIONS is created and a many-to-many relationship between DEPARTMENT and LOCATIONS will be established in the next section.

1.3 Adding Relationships

Three types of relationships are supported in ERWin: identifying, non-identifying, and many-to-many. ERWin classifies the child entity type in an identifying relationship as "weak". To add a relationship, the user may simply right click the Relationships entry in the model explorer pane and choose "New". This pops up a new relationship window as shown in Figure 1.7.

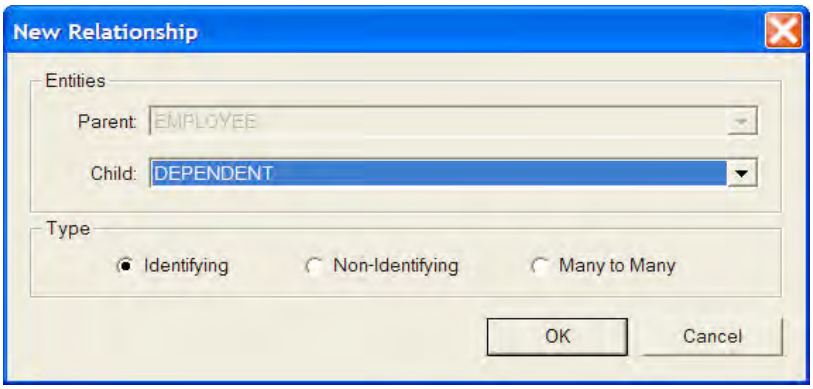

Figure 1.7: New Relationship Window

After choosing the parent and child entity types and the type of relationship and clicking OK, the new relationship is added and is reflected by a line connecting the two entity types in the diagram window panel. The many-to-many relationships are denoted by solid connecting lines, with two black dots at the two ends. Non-identifying relationships are denoted by a dashed connecting line with a black dot at many-end and a square-shaped symbol at the one-end. Identifying relationships are denoted by a solid connecting line with a black dot at the many-end and nothing special at the one-end.

After creating a new relationship, the user may add verb phrases and other properties of the relationship by right clicking the connecting line in the diagram and choosing properties.

In the case of the COMPANY database, we create the following relationships:

- One identifying relationship from EMPLOYEE to DEPENDENT.
- Two many-to-many relationships, one from EMPLOYEE to PROJECT and the other from DEPARTMENTS to LOCATIONS, and
- Four non-identifying relationships: from EMPLOYEE to DEPARTMENT (one-to-one for manages), from DEPARTMENT to EMPLOYEE (one-to-many for works for relationship), from EMPLOYEE to EMPLOYEE (one-to-many for supervisor/supervisee relationship), and from DEPARTMENT to PROJECT (one-to-many for the controls relationship).

The final logical ER diagram from the diagram window panel is shown in Figure 1.8.

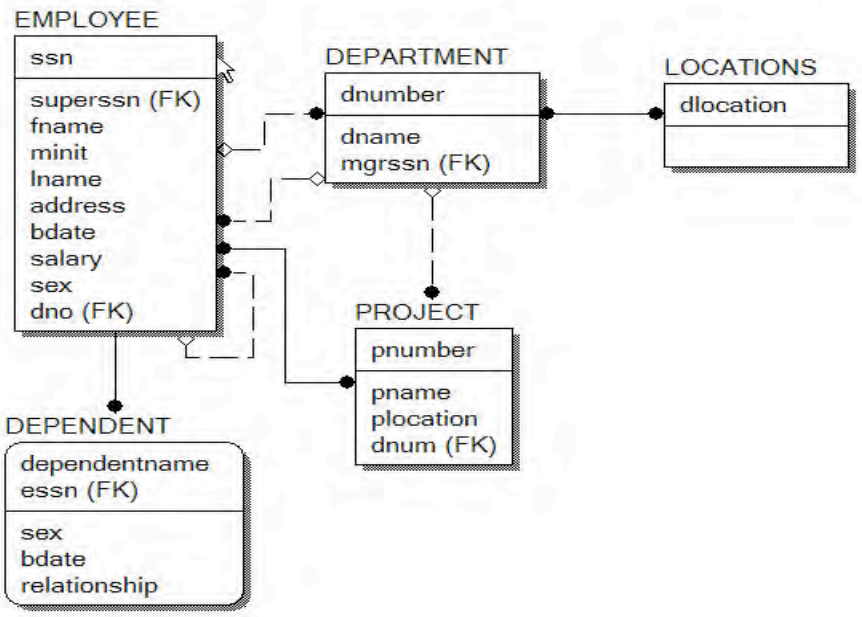

Figure 1.8: Final Logical ER Diagram

Notice that the two many-to-many relationships do not have the transforms applied yet. The transforms are shown in the physical ER diagram (obtained by switching from Logical to Physical in the Menu and Toolbar section) in Figure 1.9. Notice the introduction of the two new "entity types" for the two many-to-many relationships. These entity types are introduced because the transforms are defined at the model level.

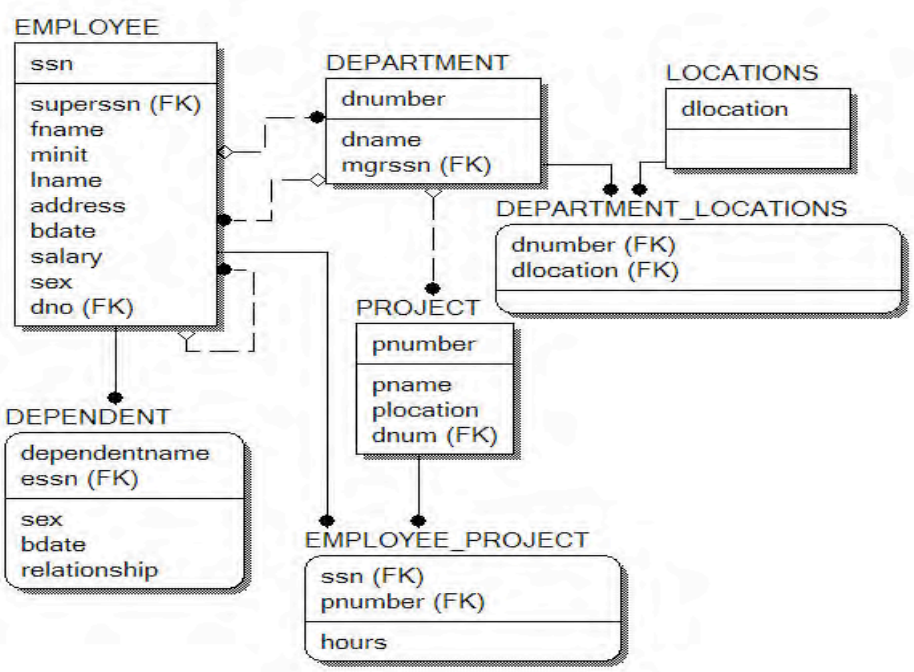

Figure 1.9: Final Physical ER Diagram

1.4 Forward Engineering

ERWin provides a powerful feature called forward engineering that allows the database designer to convert the ER design into a schema generation SQL script for one or more target relational databases. The following SQL script is obtained for the COMPANY database by choosing Tools→Forward Engineering→Schema-Generation option in the Menus and Toolbars section and clicking the "Preview" button.

```
CREATE TABLE DEPARTMENT
(
      dname VARCHAR2(20)  NOT NULL ,
      dnumber INTEGER  NOT NULL ,
      mgrssn NUMBER(9)  NULL
);

ALTER TABLE DEPARTMENT
      ADD PRIMARY KEY (dnumber);

CREATE TABLE DEPARTMENT_LOCATIONS
(
      dnumber  INTEGER  NOT NULL ,
      dlocation  VARCHAR2(20)  NOT NULL
);


ALTER TABLE DEPARTMENT_LOCATIONS
      ADD  PRIMARY KEY (dnumber,dlocation);
```

```
CREATE TABLE DEPENDENT
(
      dependentname  VARCHAR2(20)  NOT NULL ,
      sex  CHAR  NULL ,
      bdate  DATE  NULL ,
      relationship  VARCHAR2(20)  NULL ,
      essn  NUMBER(9)  NOT NULL
);


ALTER TABLE DEPENDENT
      ADD  PRIMARY KEY (dependentname,essn);


CREATE TABLE EMPLOYEE
(
      ssn  NUMBER(9)  NOT NULL ,
      superssn  NUMBER(9)  NULL ,
      fname  VARCHAR2(20)  NULL ,
      minit  CHAR  NULL ,
      lname  VARCHAR2(20)  NOT NULL ,
      address  VARCHAR2(50)  NULL ,
      bdate  DATE  NULL ,
      salary  NUMBER(8)  NULL ,
      sex  CHAR  NULL ,
      dno  INTEGER  NULL
);


ALTER TABLE EMPLOYEE
      ADD  PRIMARY KEY (ssn);


CREATE TABLE EMPLOYEE_PROJECT
(
      ssn  NUMBER(9)  NOT NULL ,
      pnumber  INTEGER  NOT NULL ,
      hours  NUMBER(3)  NULL
);


ALTER TABLE EMPLOYEE_PROJECT
      ADD  PRIMARY KEY (ssn,pnumber);


CREATE TABLE LOCATIONS
(
      dlocation  VARCHAR2(20)  NOT NULL
);


ALTER TABLE LOCATIONS
      ADD  PRIMARY KEY (dlocation);
```

```
CREATE TABLE PROJECT
(
      pnumber  INTEGER  NOT NULL ,
      pname  VARCHAR2(20)  NULL ,
      plocation  VARCHAR2(20)  NULL ,
      dnum  INTEGER  NULL
);


ALTER TABLE PROJECT
      ADD  PRIMARY KEY (pnumber);


ALTER TABLE DEPARTMENT
      ADD ( FOREIGN KEY (mgrssn) REFERENCES EMPLOYEE(ssn) ON DELETE SET NULL);


ALTER TABLE DEPARTMENT_LOCATIONS
      ADD ( FOREIGN KEY (dnumber) REFERENCES DEPARTMENT(dnumber));


ALTER TABLE DEPARTMENT_LOCATIONS
      ADD ( FOREIGN KEY (dlocation) REFERENCES LOCATIONS(dlocation));


ALTER TABLE DEPENDENT
      ADD ( FOREIGN KEY (essn) REFERENCES EMPLOYEE(ssn));


ALTER TABLE EMPLOYEE
      ADD  ( FOREIGN  KEY  (superssn)  REFERENCES  EMPLOYEE(ssn)  ON  DELETE  SET
NULL);


ALTER TABLE EMPLOYEE
      ADD  ( FOREIGN  KEY  (dno)  REFERENCES  DEPARTMENT(dnumber)  ON  DELETE  SET
NULL);


ALTER TABLE EMPLOYEE_PROJECT
      ADD ( FOREIGN KEY (ssn) REFERENCES EMPLOYEE(ssn));


ALTER TABLE EMPLOYEE_PROJECT
      ADD ( FOREIGN KEY (pnumber) REFERENCES PROJECT(pnumber));


ALTER TABLE PROJECT
      ADD  ( FOREIGN  KEY  (dnum)  REFERENCES  DEPARTMENT(dnumber)  ON  DELETE  SET
NULL);
```

The above SQL script contains table definitions and basic primary and foreign key constraints definitions. ERWin does provide a number of options to generate views, triggers, indices etc. and these can be set in the forward engineering schema generation window.

1.5 Supertype/Subtype Example

ERWin supports the creation of sub-type/super-type relationships between entity types. Consider the example in Figure 8.3 of the Elmasri/Navathe text in which a super-type entity VEHICLE consists of two sub-types CAR and TRUCK. To create this design in ERWin, the three entity types are created first. Then, the user may click the sub-type button (a circle with two parallel lines below the circle) in the Menus and Toolbars section, followed by clicking the super-type entity (VEHICLES) in the diagram window pane, followed by clicking the sub-type entity (CAR) in the diagram window pane. This process may be repeated for adding other sub-types (TRUCK in this example). The logical model for this example is shown in Figure 1.10.

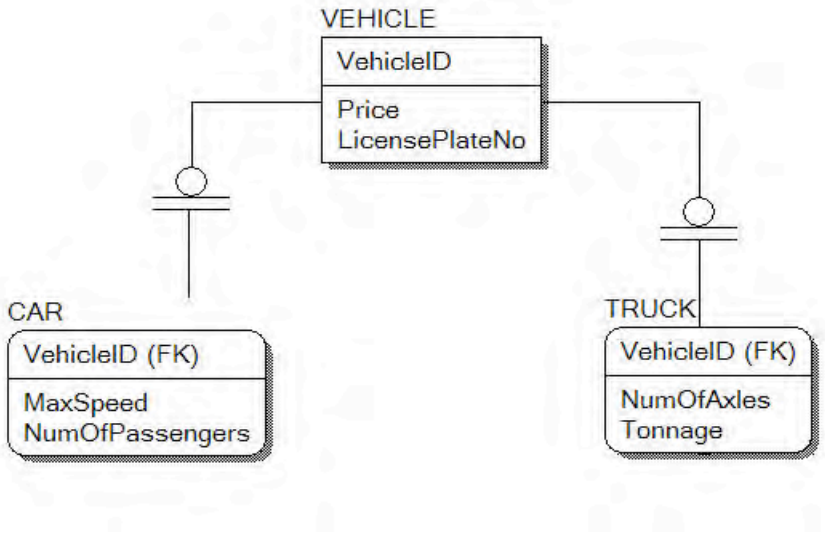

Figure 1.10: Sub-type/Super-type Logical ER Diagram

To customize the properties of the sub-type/super-type relationship, the user may right click the relationship symbol (circle with two parallel lines) and choose Subtype Relationship. This brings up a window shown in Figure 1.11. The user may choose "Complete" subtype (when all categories are known) or "Incomplete" subtype (when all categories may not be known). The user may also add verb phrases etc by right-clicking the relationship line and choosing properties as was done for ordinary relationships. ERWin also allows the user to choose a "discriminator" attribute for the sub-types (an attribute in the super-type whose values determine the sub-type object). If no discriminator attribute is defined, the user may choose "-- -- --".

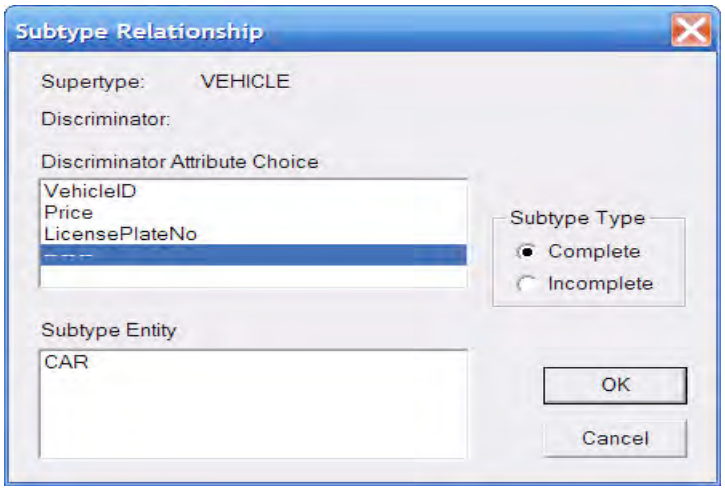

Figure 1.11: Subtype Relationship Properties

The following SQL script is produced using the forward engineering feature of ERWin for the Vehicles example:

```
CREATE TABLE CAR
(
      MaxSpeed  INTEGER  NULL ,
      NumOfPassengers  INTEGER  NULL ,
      VehicleID  INTEGER  NOT NULL
);


ALTER TABLE CAR
      ADD  PRIMARY KEY (VehicleID);


CREATE TABLE TRUCK
(
      NumOfAxles  INTEGER  NULL ,
      Tonnage  INTEGER  NULL ,
      VehicleID  INTEGER  NOT NULL
);


ALTER TABLE TRUCK
      ADD  PRIMARY KEY (VehicleID);


CREATE TABLE VEHICLE
(
      VehicleID  INTEGER  NOT NULL ,
      Price  NUMBER(8,2)  NULL ,
      LicensePlateNo  VARCHAR2(20)  NULL
);


ALTER TABLE VEHICLE
      ADD  PRIMARY KEY (VehicleID);
```

```
ALTER TABLE CAR
      ADD ( FOREIGN KEY (VehicleID) REFERENCES VEHICLE(VehicleID));


ALTER TABLE TRUCK
      ADD ( FOREIGN KEY (VehicleID) REFERENCES VEHICLE(VehicleID));
```

Exercises

ER Modeling Problems

1. Consider the *university* database described in Exercise 7.16 of the Elmasri/Navathe text. Enter the ER schema for this database using a data-modeling tool such as ERWin.

2. Consider a *mail order* database in which employees take orders for parts from customers. The data requirements are summarized as follows:

   - The mail order company has employees identified by a unique employee number, their first and last names, and a zip code where they are located.
   - Customers of the company are uniquely identified by a customer number. In addition, their first and last names and a zip code where they are located are recorded.
   - The parts being sold by the company are identified by a unique part number. In addirion, a part name, their price, and quantity in stock are recorded.
   - Orders placed by customers are taken by employees and are given a unique order number. Each order may contain certain quantities of one or more parts and their received date as well as a shipped date is recorded.

   Design an Entity-Relationship diagram for the mail order database and enter the design using a data-modeling tool such as ERWin.

3. Consider a *movie* database in which data is recorded about the movie industry. The data requirements are summarized as follows:
   - Movies are identified by their title and year of release. They have a length in minutes. They also have a studio that produces the movie and are classified under one or more genres (such as horror, action, drama etc). Movies are directed by one or more directors and have one or more actors acting in them. The movie also has a plot outline. Each movie also has zero or more quotable quotes that are spoken by a particular actor acting in the movie.
   - Actors are identified by their names and date of birth and act in one or more movies. Each actor has a role in the movie.
   - Directors are also identified by their names and date of birth and direct one or more movies. It is possible for a director to act in a movie (not necessarily in a movie they direct).

- Studios are identified by their names and have an address. They produce one or more movies.

Design an Entity-Relationship diagram for the movie order database and enter the design using a data-modeling tool such as ERWin.

4. Consider a *conference review* system database in which researchers submit their research papers for consideration. The database system also caters to reviewers of papers who make recommendations on whether to accept or reject the paper. The data requirements are summarized as follows:
   Authors of papers are uniquely identified by their email id. Their first and last names are also recorded.
   - Papers are assigned unique identifiers by the system and are described by a title, an abstract, and a file name containing the actual paper.
   - Papers may have multiple authors, but one of the authors is designated as the contact author.
   - Reviewers of papers are uniquely identified by their email id. Their first and last names are also recorded.
   - Each paper is assigned between two and four reviewers. The reviewer rates the paper assigned to him on a scale of 1 to 10.
   - Each review contains two types of written comments: one to be seen by the review committee only and the other by the author(s) as well.

   Design an Entity-Relationship diagram for the conference review database and enter the design using a data-modeling tool such as ERWin.

5. Consider the ER diagram for the AIRLINE database shown in Figure 7.20 of the Elmasri/Navathe text. Enter this design using a data-modeling tool such as ERWin.

Enhanced ER Modeling Problems

6. Consider a *grade book* database in which instructors within an academic department maintain scores/points obtained by individual students in their classes. The data requirements are summarized as follows:

   - Students are identified by a unique student id, their first and last names, and an email address.
   - The instructor teaches certain courses each term. The courses are uniquely identified by a course number, a section number, and the term in which they are taught. The instructor also assigns grade cutoffs (example 90, 80, 70, and 60) for letter grades A, B, C, D, and F for each course he or she teaches.
   - Students are enrolled in courses taught by the instructor.
   - Each course being taught by the instructor has a number of grading components (such as mid-term, final exam, project, etc.). Each grading component has a maximum number of points (such as 100 or 50) and a weight (such as 20% or 10%). The weights of all the grading components of a course usually add up to 100.

- Finally, the instructor records the points earned by each student in each of the grading components in each of the courses. For example, student with id=1234 earns 84 points for the grading component mid-term for the course CSc 2310 section 2 in the fall 2005 term. The mid-term grading component may have been defined to have a maximum of 100 points and a weight of 20% of the course grade.

Design an enhanced Entity-Relationship diagram for the grade book database and enter the design using a data-modeling tool such as ERWin.

7. Consider an *online auction* database system in which members (buyers and sellers) participate in the sale of items. The data requirements for this system are summarized as follows:

   - The online site has members who are identified by a unique member id and are described by an email address, their name, a password, their home address, and a phone number.
   - A member may be a buyer or a seller. A buyer has a shipping address recorded in the database. A seller has a bank account number and routing number recorded in the database.
   - Items are placed by a seller for sale and are identified by a unique item number assigned by the system. Items are also described by an item title, an item description, a starting bid price, bidding increment, the start date of the auction, and the end date of the auction.
   - Items are also categorized based on a fixed classification hierarchy (for example a modem may be classified as /COMPUTER/HARDWARE/MODEM).
   - Buyers make bids for items they are interested in. A bidding price and time of bid placement is recorded. The person at the end of the auction with the highest bid price is declared the winner and a transaction between the buyer and the seller may proceed soon after.
   - Buyers and sellers may place feedback ratings on the purchase or sale of an item. The feedback contains a rating between 1 and 10 and a comment. Note that the rating is placed by the buyer or seller involved in the completed transaction.

   Design an Entity-Relationship diagram for the auction database and enter the design using a data-modeling tool such as ERWin.

8. Consider a database system for a baseball organization such as the major leagues. The data requirements are summarized as follows:

   - The personnel involved in the league include players, coaches, managers, and umpires. Each is identified by a unique personnel id. They are also described by their first and last names along with the date and place of birth.
   - Players are further described by other attributes such as their batting orientation (left, right, or switch) and have a lifetime batting average (BA).
   - Within the players group is a subset of players called pitchers. Pitchers have a life-time ERA (earned run average) associated with them.

- Teams are uniquely identified by their names. Teams are also described by the city in which they are located and the division and league in which they play (such as Central division of the American league).
- Teams have one manager, a number of coaches, and a number of players.
- Games are played between two teams with one designated as the home team and the other the visiting team on a particular date. The score (runs, hits, and errors) are recorded for each team. The team with more number of runs is declared the winner of the game.
- With each finished game, a winning pitcher and a losing pitcher are recorded. In case there is a save awarded, the save pitcher is also recorded.
- With each finished game, the number of hits (singles, doubles, triples, and home runs) obtained by each player is also recorded.

Design an Entity-Relationship diagram for the baseball database and enter the design using a data-modeling tool such as ERWin.

9. Consider the ER diagram for the university database shown in Figure 8.9 of the Elmasri/Navathe text. Enter this design using a data-modeling tool such as ERWin.

10. Consider the ER diagram for the small airport database shown in Figure 8.12 of the Elmasri/Navathe text. Enter this design using a data-modeling tool such as ERWin.

CHAPTER 2

Abstract Query Languages

This chapter introduces Java-based interpreters for three abstract query languages: Relational Algebra (RA), Domain Relational Calculus (DRC), and Datalog. The interpreters have been implemented using the parser generator tools JCup and JFlex. In order to use these interpreters, one needs to only download two jar files: `dbengine.jar` and `aql.jar` and include them in the Java CLASSPATH. The JCup libraries are included as part of the jar files and hence the only other software that is required to use the interpreters is a standard Java environment.

The system is simple to use and comes with a database engine that implements a set of basic relational algebraic operators. The interpreter reads a query from the terminal and performs the following three steps:

(1) **Syntax Check**: The query is checked for any syntax errors. If there are any syntactic errors, the interpreter reports these to the terminal and waits to read another query; otherwise the interpreter proceeds to the second step.

(2) **Semantics Check**: The syntactically correct query is checked for semantic errors including type mismatches, invalid column references, and invalid relation names. In addition, the DRC and Datalog interpreters check the queries for safety. If there are any semantic errors or if the DRC/Datalog query is unsafe, the interpreter reports these to the terminal and waits to read another query; otherwise the interpreter proceeds to the third step.

(3) **Query Evaluation**: The query is evaluated using the primitives provided by the database engine and the results are displayed.

2.1 Creating the Database

Before the user can start using the interpreters, they must create a database against which they will submit queries. The database consists of several text files all stored within a directory. The directory is named after the database name. For example, to create a database identified with the name `db1` and containing two tables:

```
student(sid:integer,sname:varchar,phone:varchar,gpa:decimal)
skills(sid:integer,language:varchar)
```

a directory called `db1` should be created along with the following three files (one for the catalog description and the remaining two for the data for the two tables):

```
catalog.dat
STUDENT.dat
SKILLS.dat
```

The file names are case sensitive and should strictly follow the convention used, i.e. `catalog.dat` should be all lower case and the data files should be named after their relation name in upper case followed by the file suffix, `.dat`, in lower case.

The `catalog.dat` file contains the number of relations in the first line followed by the descriptions of each relation. The description of each relation begins with the name of the relation in a separate line followed by the number of attributes in a separate line followed by attribute descriptions. Each attribute description includes the name of the attribute in a separate line followed by the data type (`VARCHAR`, `INTEGER`, or `DECIMAL`) in a separate line. All names and data types are in upper case. There should be no leading or trailing white space in any of the lines. The `catalog.dat` file for database `db1` is shown below:

```
2
STUDENT
4
SID
INTEGER
SNAME
VARCHAR
PHONE
VARCHAR
GPA
DECIMAL
SKILLS
2
SID
INTEGER
LANGUAGE
VARCHAR
```

The `db1` directory must include one data file for each relation. In the case of `db1`, they should be named `STUDENT.dat` and `SKILLS.dat`. The data file for relations contains the number of tuples in the first line followed by the description of each tuple. Tuples are described by the values under each column with each vale in a separate line. For example, let the `SKILLS` relation have three tuples:

```
(111,Java)
(111,C++)
(222,Java)
```

These tuples will be represented in the `SKILLS.dat` data file as follows:

```
3
111
Java
```

```
111
C++
222
Java
```

Again, there should be no leading or trailing white spaces in any of the lines. Some pre-defined databases are available along with this laboratory manual. New data may be added to existing databases as well as new databases may be created when needed.

2.2 Relational Algebra Interpreter

The RA interpreter is invoked using the following terminal command:

```
$ java edu.gsu.cs.ra.RA company
```

Here `$` is the command prompt and `company` is the name of the database (as well as the name of the directory where the database files are stored). This command assumes that the `company` directory is present in the same directory where this command is issued. Of course, one can issue this command in a different directory by providing the full path to the database directory.

The interpreter responds with the following prompt:

```
RA>
```

At this prompt the user may enter a Relational Algebra query or type the exit command. Every query is terminated by a ";". Even the exit command must end with a semi-colon. Queries may span more than one line; upon typing the ENTER key the interpreter prints the RA> prompt and waits for further input unless the ENTER key is typed after a semi-colon, in which case the query is processed by the interpreter.

2.2.1 Relational Algebra Syntax

A subset of Relational Algebra that includes the union, minus, intersect, Cartesian product, natural join, select, project, and rename operators is implemented in the interpreter. The context-free grammar for this subset is shown below:

```
<Query> ::= <Expr> SEMI;
<Expr>  ::= <ProjExpr>  | <RenameExpr>    | <UnionExpr> |
            <MinusExpr> | <IntersectExpr> | <JoinExpr> |
            <TimesExpr> | <SelectExpr>    | RELATION
<ProjExpr>   ::= PROJECT [<AttrList>] (<Expr>)
<RenameExpr> ::= RENAME [<AttrList>] (<Expr>)
<AttrList>   ::= ATTRIBUTE | <AttrList> , ATTRIBUTE
<UnionExpr>  ::= (<Expr> UNION <Expr>)
<MinusExpr>  ::= (<Expr> MINUS <Expr>)
<IntersectExpr>  ::= (<Expr> INTERSECT <Expr>)
```

```
<JoinExpr>   ::= (<Expr> JOIN <Expr>)
<TimesExpr>  ::= (<Expr> TIMES <Expr>)
<SelectExpr> ::= SELECT [<Condition>](<Expr>)
<Condition>  ::= <SimpleCondition> |
                 <SimpleCondition> AND <Condition>
<SimpleCondition> ::= <Operand> <Comparison> <Operand>
<Operand>    ::= ATTRIBUTE | STRING-CONST | NUMBER-CONST
<Comparison> ::= < | <= | = | <> | > | >=
```

The terminal strings in the grammar include

- Keywords for the relational algebraic operators: `PROJECT`, `RENAME`, `UNION`, `MINUS`, `INTERSECT`, `JOIN`, `TIMES`, and `SELECT`. These keywords are case-insensitive.
- Logical keyword `AND` (case-insensitive).
- Miscellaneous syntactic character strings such as (,), <, <=, =, <>, >, >=, ;, and comma (,).
- Name strings: `RELATION` and `ATTRIBUTE` (case-insensitive names of relations and their attributes).
- Constant strings: `STRING-CONST` (a string enclosed within single quotes; e.g. 'Thomas') and `NUMBER-CONST` (integer as well as decimal numbers; e.g. 232 and -36.1).

An example of a well-formed syntactically correct query for the `company` database of the Elmasri/Navathe text is:

```
( project[ssn](select[lname='Jones'](employee))
   union
  project[superssn](select[dno=5](employee))
);
```

All relational algebra queries must be terminated by a ";".

A relational algebra query in the simplest form is a "relation name". For example the following terminal session with the interpreter illustrates the execution of this simple query form:

```
$ java edu.gsu.cs.ra.RA company
RA> departments;
SEMANTIC ERROR in RA Query: Relation DEPARTMENTS does not exist
RA> department;
DEPARTMENT(DNAME:VARCHAR,DNUMBER:INTEGER,MGRSSN:VARCHAR,MGRSTARTDA
TE:VARCHAR)

Number of tuples = 6
Research:5:333445555:22-MAY-1978:
Administration:4:987654321:01-JAN-1985:
Headquarters:1:888665555:19-JUN-1971:
Software:6:111111100:15-MAY-1999:
Hardware:7:444444400:15-MAY-1998:
```

```
Sales:8:555555500:01-JAN-1997:

RA> exit;
$
```

In response to a query, the interpreter displays the schema of the result followed by the answer to the query. Individual values within a tuple are terminated by a ":". The simplest query form is useful to display the database contents.

More complicated relational algebra queries involve one or more applications of one or more of the several operators such as select, project, times, join, union, etc. For example, consider the query "*Retrieve the names of all employees working for Dept. No. 5*". This would be expressed by the query execution in the following RA session:

```
RA> project[fname,lname](select[dno=5](employee));
temp1(FNAME:VARCHAR,LNAME:VARCHAR)

Number of tuples = 4
Franklin:Wong:
John:Smith:
Ramesh:Narayan:
Joyce:English:

RA>
```

2.2.2 Naming of Intermediate Relations and Attributes

The RA interpreter assigns temporary relation names such as `temp0, temp1`, etc. to each intermediate relation encountered in the execution of the entire query. The RA interpreter also employs the following rules as far as naming of attributes/columns of intermediate relations:

1. Union, Minus, and Intersect: The attribute/column names from the left operand are used to name the attributes of the output relation.
2. Times (Cartesian Product): Attribute/Column names from both operands are used to name the attributes of the output relation. Attribute/Column names that are common to both operands are prefixed by relation name (`tempN`).
3. Select: The attribute names of the output relation are the same as the attribute/column names of the operand.
4. Project, Rename: Attribute/Column names present in the attribute list parameter of the operator are used to name the attributes of the output relation. Duplicate attribute/column names are not allowed in the attribute list.
5. Join (Natural Join): Attribute/Column names from both operands are used to name the attributes of the output relation. Common attribute/column names appear only once.

As another example, consider the query "*Retrieve the social security numbers of employees who either work in department 5 or directly supervise an employee who works in department 5*". The query is illustrated in the following RA session:

```
RA> (project[ssn](select[dno=5](employee))
RA> union project[superssn](select[dno=5](employee)));
temp4(SSN:VARCHAR)

Number of tuples = 5
333445555:
123456789:
666884444:
453453453:
888665555:

RA>
```

2.2.3 Relational Algebraic Operators Supported by the RA Interpreter

Select: As can be noted from the grammar, the select operator supported by the interpreter has the following syntax:

```
select[condition](expression)
```

where `condition` is a conjunction of one or more simple conditions involving comparisons of attributes or constants with other attributes or constants. The attributes used in the condition must be present in the attributes of the relation corresponding to `expression`.

Project: The project operator supported by the interpreter has the following syntax:

```
project[attribute-list](expression)
```

where `attribute-list` is a comma separated list of attributes, each of which is present in the attributes of the relation corresponding to `expression`.

Rename: The syntax for the rename operator is

```
rename[attribute-list](expression)
```

where `attribute-list` is a comma separated list of attribute names. The number of attributes mentioned in the list must be equal to the number of attributes of the relation corresponding to `expression`.

Join: The syntax for the join operator is

```
(expression1 join expression2)
```

There is no restriction on the schemas of the two expressions.

Times: The syntax for the times operator is

```
(expression1 times expression2)
```

There is no restriction on the schemas of the two expressions.

Union: The syntax for the union operator is

```
(expression1 union expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

Minus: The syntax for the minus operator is

```
(expression1 minus expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

Intersect: The syntax for the intersect operator is

```
(expression1 intersect expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

2.2.4 Examples

The queries from Section 6.5 of the Elmasri/Navathe text modified to work with the RA interpreter are shown below:

Query 1: Retrieve the name and address of employees who work for the "`Research`" department.

```
project[fname,lname,address](
  (rename[dname,dno,mgrssn,mgrstartdate](
      select[dname='Research'](department))
   join
   employee
  )
);
```

Query 2: For every project located in "Stafford", list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
project[pnumber,dnum,lname,address,bdate](
  (
   (select[plocation='Stafford'](projects)
    join
    rename[dname,dnum,ssn,mgrstartdate](department)
   )
   join employee
  )
);
```

Query 3: Find the names of employees who work on all the projects controlled by department number 5.

```
project[lname,fname](
 (employee
  join
  (project[ssn](employee)
   minus
   project[ssn](
     (
       (project[ssn](employee)
        times
        project[pnumber](select[dnum=5](projects))
       )
       minus
       rename[ssn,pnumber](project[essn,pno](works_on))
     )
   )
  )
 )
);
```

Query 4: Make a list of project numbers for projects that involve an employee whose last name is "Smith", either as a worker or as a manager of the department that controls the project.

```
( project[pno](
   (rename[essn](project[ssn](select[lname='Smith'](employee)))
    join
    works_on
   )
  )
 union
  project[pnumber](
   ( rename[dnum](project[dnumber](select[lname='Smith'](
```

```
        (employee
         join
         rename[dname,dnumber,ssn,mgrstartdate](department)
        )
        )
        )
      )
      join
      projects
     )
  )
);
```

Query 5: List the names of all employees with two or more dependents.

```
project[lname,fname](
(rename[ssn](
 project[essn1](
  select[essn1=essn2 and dname1<>dname2](
   (rename[essn1,dname1](project[essn,dependent_name](dependent))
    times
    rename[essn2,dname2](project[essn,dependent_name](dependent)))
   )
  )
 )
join
employee)
);
```

Query 6: Retrieve the names of employees who have no dependents.

```
project[lname,fname](
 ( ( project[ssn](employee)
     minus project[essn](dependent)
   )
   join
   employee
 )
);
```

Query 7: List the names of managers who have at least one dependent.

```
project[lname,fname](
  ((rename[ssn](project[mgrssn](department))
    join
    rename[ssn](project[essn](dependent))
   )
```

```
  join
  employee
  )
);
```

Important Tip: Since many of the queries shown above are long and span multiple lines, the best way to use the interpreter is to create a text file in which the queries are typed. These queries are then cut and pasted into the interpreter prompt. Any errors in syntax or semantics should be corrected in the text file and then the process of cut and paste should be repeated until a correct solution is reached.

2.3 Domain Relational Calculus Interpreter

The DRC interpreter is invoked using the following terminal command:

```
$ java edu.gsu.cs.drc.DRC company
```

Here `$` is the command prompt and `company` is the name of the database (as well as the name of the directory where the database files are stored). This command assumes that the `company` directory is present in the same directory where this command is issued. Of course, one can issue this command in a different directory by providing the full path to the database directory.

The interpreter responds with the following prompt:

```
DRC>
```

At this prompt the user may enter a Domain Relational Calculus query or type the `exit` command. Each DRC query is expressed in a set-notation using a pair of curly brackets as follows:

```
{ variable-list | P(variable-list) }
```

where `variable-list` is a comma-separated list of variables which must all be present in the body predicate of the query `P(variable-list)` as free-variables.

The `exit` command must end with a semi-colon. Queries may span more than one line; upon typing the `ENTER` key the interpreter prints the `DRC>` prompt and waits for further input unless the `ENTER` key is typed after a right curly bracket (`}`), in which case the query is processed by the interpreter.

2.3.1 Domain Relational Calculus Syntax

The context-free grammar for DRC queries implemented within the DRC interpreter is shown below:

```
Query ::= LBRACE VarList BAR Formula RBRACE;
```

```
VarList ::= NAME | VarList COMMA NAME;
Formula ::= AtomicFormula |
            Formula AND Formula |
            Formula OR Formula |
            NOT LPAREN Formula RPAREN |
            LPAREN EXISTS VarList RPAREN LPAREN Formula RPAREN |
            LPAREN FORALL VarList RPAREN LPAREN Formula RPAREN;
AtomicFormula ::=
     NAME LPAREN ArgList RPAREN | Arg Comparison Arg;
ArgList ::= Arg | ArgList COMMA Arg;
Arg ::= NAME | STRING | NUMBER;
Comparison ::= < | <= | = | <> | > | >=
```

The terminal strings in the grammar include

- Keywords for the logical operators: `AND`, `OR`, and `NOT`. These keywords are case-insensitive.
- Quantifier keywords `EXISTS` and `FORALL` (case-insensitive).
- Miscellaneous syntactic character strings such as (,), <, <=, =, <>, >, >=, and comma (,).
- `NAME` strings: used for named relations and variables (case-insensitive).
- Constant strings: `STRING` (a string enclosed within single quotes; e.g. 'Thomas') and `NUMBER` (integer as well as decimal numbers; e.g. 232 and -36.1).

An example of a well-formed syntactically correct query on the `company` database of the Elmasri/Navathe text is:

```
{ x | (exists a1,a2,a3,a4,a5,a6,a7,a8)(
          employee(a1,a2,'Jones',x,a3,a4,a5,a6,a7,a8)) or
      (exists a1,a2,a3,a4,a5,a6,a7,a8)(
          employee(a1,a2,a3,x,a4,a5,a6,a7,a8,5)) }
```

All DRC queries must be enclosed within a pair of matching curly brackets.

The simplest DRC query displays the contents of a relation. For example the following terminal session with the interpreter illustrates the execution of this simple query form that displays the contents of the `DEPARTMENT` relation:

```
$ java edu.gsu.cs.drc.DRC company
DRC> { a,b,c,d | department(a,b,c,d) }
ANSWER(A:VARCHAR,B:INTEGER,C:VARCHAR,D:VARCHAR)

Number of tuples = 6
Research:5:333445555:22-MAY-1978:
Administration:4:987654321:01-JAN-1985:
Headquarters:1:888665555:19-JUN-1971:
Software:6:111111100:15-MAY-1999:
```

```
Hardware:7:444444400:15-MAY-1998:
Sales:8:555555500:01-JAN-1997:

DRC>
```

In response to a query, the interpreter displays the schema of the result followed by the answer to the query.

2.3.2 Safe DRC Queries

The DRC interpreter checks for the "safety" of queries and evaluates only those that are determined to be safe. An error message is generated for unsafe queries.

For the discussion of safe DRC queries, we will assume that the formula defining the query does not contain the `forall` quantifier. If the `forall` quantifier does appear in the formula, the user can convert such a formula to an equivalent one without the `forall` quantifier using the logical equivalence:

```
(forall X)(F) ≡ NOT ((exists X)( NOT (F)))
```

It is almost always the case that the `F` in the `forall` quantified formula above is of the form

```
NOT (P) or Q
```

In case the user does not eliminate the `forall` quantifier, the DRC interpreter would automatically convert all `forall` quantified formulas into equivalent `exists` quantified formulas using the above equivalence. In addition, the interpreter would also apply the DeMorgan's law:

```
NOT (P or Q) ≡ NOT (P) and NOT (Q)
```

to push the `NOT` further inside the formula.

As an example of this automatic transformation, consider the following query provided by the user:

```
{a,b |(exists c)(r(a,b,c) and
          (forall d,e)(not(s(a,d,e)) or (exists f)(t(d,f)))) }
```

The DRC interpreter would convert the above query to:

```
{a,b |(exists c)(r(a,b,c) and
        not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f)))) }
```

Definition: A DRC query (without `forall` quantifiers) is defined to be *safe* if it satisfies the following three conditions:

(a) For every sub-formula in the query connected with an "`or`", the two operand formulas have the same set of free variables, i.e. the "`or`" formula is of the form:

```
F(X1,…,Xn) or G(X1,…,Xn)
```

(b) All free variables appearing in "maximal sub-conjuncts", `F1 and … and Fn`, must be "limited" in that they either appear in (i) a positive sub-formula `Fi` or (ii) as `X` in an sub-formula of the form `X=a` or `a=X` or (iii) as `X` in a sub-formula of the form `X=Y` where `Y` is determined to be "limited".

(c) The NOT operator may be applied only to a term in a maximal sub-conjunct of type discussed in (b), i.e. all free variables in the NOT term must be shown to be "limited" in the positive terms of the maximal sub-conjunct.

Some examples follow. The following query would be considered safe as it satisfies condition (a).

```
{a,b | (exists c)(r(a,b,c)) or s(a,b) }
```

But the following would not be safe:

```
{a,b | (exists b,c)(r(a,b,c)) or s(a,b) }
```

This is because the free variables on the left operand of the "`or`" formula consists of only one variable, `a`, and the free variables on the right operand consists of two variables, `a` and `b`.

The query formula from an earlier query:

```
(exists c)(r(a,b,c) and
  not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f))))
```

is safe. The formula has the following two maximal sub-conjuncts (ignoring atomic formulas which are maximal sub-conjuncts of size 1):

(1) `s(a,d,e) and not(exists f)(t(d,f))`
all three free variables `a`, `d`, and `e` are limited as they appear in `s(a,d,e)`

(2) `r(a,b,c) and`
`not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f)))`
all three free variables `a, b,` and `c` are limited as they appear in `r(a,b,c).`

The free variables in each of the maximal sub-conjuncts are shown to be "limited" and hence the overall query is safe.

The following query formula is unsafe:

```
p(a,b) and not ((exists c)(q(b,c,d)))
```

This is because the free variable `d` is not “limited” as it is not grounded in a positive term in the maximal sub-conjunct.

2.3.3 DRC Query Examples

The queries from Section 6.7 of the Elmasri/Navathe text modified to work with DRC interpreter are shown below:

Query 0: Retrieve the birthdate and address of the employees whose name is "`John B. Smith`".

```
{ u,v | (exists t,w,x,y,z)(
            employee('John','B','Smith',t,u,v,w,x,y,z) ) }
```

Query 1: Retrieve the name and address of all employees who work for the "`Research`" department.

```
{ q,s,v | (exists r,t,u,w,x,y,z,n,o)(
            employee(q,r,s,t,u,v,w,x,y,z) and
            department('Research',z,n,o) ) }
```

Query 2: For every project located in "`Stafford`", list the project number, the controlling department number, and the department manager's last name, birth date, and address.

```
{ i,k,s,u,v | (exists h,q,r,t,w,x,y,z,l,o)(
                projects(h,i,'Stafford',k) and
                employee(q,r,s,t,u,v,w,x,y,z) and
                department(l,k,t,o) ) }
```

Query 6: List the names of employees who have no dependents.

```
{ q,s | (exists r,t,u,v,w,x,y,z)(
          employee(q,r,s,t,u,v,w,x,y,z) and
          not ((exists m,n,o,p)(dependent(t,m,n,o,p))) ) }
```

The following is not SAFE and would not work

```
{ q,s | (exists r,t,u,v,w,x,y,z)(
          employee(q,r,s,t,u,v,w,x,y,z) and
         (forall l,m,n,o,p)(not (dependent(l,m,n,o,p)) or t<>l) )}
```

Query 7: List the names of managers who have at least one dependent.

```
{ s,q | (exists r,t,u,v,w,x,y,z,h,i,k,m,n,o,p)(
           employee(q,r,s,t,u,v,w,x,y,z) and
           department(h,i,t,k) and
           dependent(t,m,n,o,p) ) }
```

2.4 Datalog Interpreter

The DLOG interpreter is invoked using the following terminal command:

$ `java edu.gsu.cs.dlg.DLOG company`

Here `$` is the command prompt and `company` is the name of the database (as well as the name of the directory where the database files are stored). This command assumes that the `company` directory is present in the same directory where this command is issued. Of course, one can issue this command in a different directory by providing the full path to the database directory.

The interpreter responds with the following prompt:

```
DLOG>
```

At this prompt the user may enter the query execution command `@file-name` or type the `exit` command, where file-name contains the Datalog query. Each command is to be terminated by a semi-colon. Even the `exit` command must end with a semi-colon.

2.4.1 Datalog Syntax

Datalog is a rule-based logical query language for relational databases. The syntax of Datalog is defined below:

An *atomic formula* is of one of the following two forms:

1. `p(x1, ..., xn)` where `p` is a relation name and `x1`, ..., `xn` are either constants or variables, or
2. `x <op> y` where `x` and `y` are either constants or variables and `<op>` is one of the six comparison operators: `<, <=, >, >=, =, !=.`

A *Datalog rule* is of the form:

```
p :- q1, ..., qn.
```

Here `p` is an atomic formula and `q1, ..., qn` are either atomic formulas or negated atomic formulas (i.e. atomic formula preceded by `not`). `p` is referred to as the head of the rule, and `q1, ..., qn` are referred to as sub-goals.

A Datalog rule `p :- q1, ..., qn.` is said to be *safe* if

1. Every variable that occurs in a negated sub-goal also appears in a positive sub-goal, and
2. Every variable that appears in the head of the rule also appears in the body of the rule.

A *Datalog query* is set of safe Datalog rules with at least one rule having the `answer` predicate in the head. The `answer` predicate collects all answers to the query.

Note: Variables that appear only once in a rule can be replaced by anonymous variables (represented by underscores). Every anonymous variable is different from all other variables.

2.4.2 Datalog Query Examples

The following are examples of Datalog queries against the company database:

Query 1: Get names of all employees in department 5 who work more than 10 hours/week on the ProductX project.

```
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,5),
  works_on(S,P,H),
  projects('ProductX',P,_,_),
  H >= 10.
```

Query 2: Get names of all employees who have a dependent with the same first name as their own first names.

```
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_),
  dependent(S,F,_,_,_).
```

Query 3: Get the names of all employees who are directly supervised by Franklin Wong.

```
answer(F,M,L) :-
  employee(F,M,L,_,_,_,_,_,S,_),
  employee('Franklin',_,'Wong',S,_,_,_,_,_,_).
```

Query 4: Get the names of all employees who work on every project.

```
temp1(S,P) :-
  employee(_,_,_,S,_,_,_,_,_,_),
  projects(_,P,_,_).
temp2(S,P) :-
  works_on(S,P,_).
temp3(S) :-
  temp1(S,P), not temp2(S,P).
answer(F,M,L) :-
```

```
    employee(F,M,L,S,_,_,_,_,_,_), not temp3(S).
```

In this query, `temp1(S,P)` collects all combinations of employees, `S`, and projects, `P`; `temp2(S,P)` collects only those pairs where employee `S` works on project `P`; `temp3(S)` collects employees, `S`, who do not work for a particular project (these employees should not be in the answer). A second negation in the final rule gets the answers to the query.

Query 5: Get the names of employees who do not work on any project.

```
temp1(S) :-
  works_on(S,_,_).
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_), not temp1(S).
```

Query 6: Get the names and addresses of employees who work for at least one project located in Houston but whose department does not have a location in Houston.

```
temp1(S) :-
  works_on(S,P,_), project(_,P,'Houston',_).
temp2(S) :-
  employee(_,_,_,S,_,_,_,_,_,D),
  not dept_locations(D,'Houston').
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp1(S), temp2(S).
```

`temp1(S)` collects employee `S` who work for a project located in `Houston`; `temp2(S)` collects employees `S` whose department do not have a location in `Houston`; the final rule intersects the two temp predicates to get the answer to the query.

Query 7: Get the names and addresses of employees who work for at least one project located in Houston or whose department does not have a location in Houston. (Note: this is a slight variation of the previous query with 'but' replaced by 'or').

```
temp1(S) :-
  works_on(S,P,_),
  project(_,P,'Houston',_).
temp2(S) :-
  employee(_,_,_,S,_,_,_,_,_,D),
  not dept_locations(D,'Houston').
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp1(S).
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp2(S).
```

Query 8: Get the last names of all department managers who have no dependents.

```
temp1(S) :-
  dependent(S,_,_,_,_).
answer(L) :-
  employee(_,_,L,S,_,_,_,_,_,_),
  department(_,_,S,_),
  not temp1(S).
```

To execute the above queries using the Datalog interpreter, each must be placed in a separate file with a `$` symbol appearing at the end of the file. Assume that the queries are placed in files named `q1, q2, …, q8`. The following is a terminal session showing the execution of the above queries:

```
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q1;
-----------------------------------
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,5),
  works_on(S,P,H), H >= 10,
  projects('ProductX',P,_,_).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 2
John:B:Smith:
Joyce:A:English:

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q2;
-----------------------------------
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_),
  dependent(S,F,_,_,_).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 1
Alec:C:Best:

DLOG> exit;
Exiting...
```

```
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q3;
-----------------------------------
answer(F,M,L) :-
  employee(F,M,L,_,_,_,_,_,S,_),
  employee('Franklin',_,'Wong',S,_,_,_,_,_,_).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 3
John:B:Smith:
Ramesh:K:Narayan:
Joyce:A:English:

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q4;
-----------------------------------
temp1(S,P) :-
  employee(_,_,_,S,_,_,_,_,_,_),
  projects(_,P,_,_).
temp2(S,P) :-
  works_on(S,P,_).
temp3(S) :-
  temp1(S,P), not temp2(S,P).
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_), not temp3(S).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 0

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q5;
-----------------------------------
```

```
temp1(S) :-
  works_on(S,_,_).
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_), not temp1(S).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 2
Bob:B:Bender:
Kate:W:King:

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q6;
-----------------------------------
temp1(S) :-
  works_on(S,P,_), projects(_,P,'Houston',_).
temp2(S) :-
  employee(_,_,_,S,_,_,_,_,_,D),
  not dept_locations(D,'Houston').
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp1(S), temp2(S).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR,A:VARCHAR)

Number of tuples = 1
Jennifer:S:Wallace:291 Berry, Bellaire, TX:

DLOG> exit;
Exiting...
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q7;
-----------------------------------
temp1(S) :-
  works_on(S,P,_),
  projects(_,P,'Houston',_).
temp2(S) :-
  employee(_,_,_,S,_,_,_,_,_,D),
  not dept_locations(D,'Houston').
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp1(S).
```

```
answer(F,M,L,A) :-
  employee(F,M,L,S,_,A,_,_,_,_), temp2(S).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR,A:VARCHAR)

Number of tuples = 38
James:E:Borg:450 Stone, Houston, TX:
Franklin:T:Wong:638 Voss, Houston, TX:
Jennifer:S:Wallace:291 Berry, Bellaire, TX:
Ramesh:K:Narayan:971 Fire Oak, Humble, TX:
Alicia:J:Zelaya:3321 Castle, Spring, TX:
Ahmad:V:Jabbar:980 Dallas, Houston, TX:
Jared:D:James:123 Peachtree, Atlanta, GA:
Alex:D:Freed:4333 Pillsbury, Milwaukee, WI:
John:C:James:7676 Bloomington, Sacramento, CA:
Jon:C:Jones:111 Allgood, Atlanta, GA:
Justin:null:Mark:2342 May, Atlanta, GA:
Brad:C:Knight:176 Main St., Atlanta, GA:
Evan:E:Wallis:134 Pelham, Milwaukee, WI:
Josh:U:Zell:266 McGrady, Milwaukee, WI:
Andy:C:Vile:1967 Jordan, Milwaukee, WI:
Tom:G:Brand:112 Third St, Milwaukee, WI:
Jenny:F:Vos:263 Mayberry, Milwaukee, WI:
Chris:A:Carter:565 Jordan, Milwaukee, WI:
Kim:C:Grace:6677 Mills Ave, Sacramento, CA:
Jeff:H:Chase:145 Bradbury, Sacramento, CA:
Bonnie:S:Bays:111 Hollow, Milwaukee, WI:
Alec:C:Best:233 Solid, Milwaukee, WI:
Sam:S:Snedden:987 Windy St, Milwaukee, WI:
Nandita:K:Ball:222 Howard, Sacramento, CA:
Bob:B:Bender:8794 Garfield, Chicago, IL:
Jill:J:Jarvis:6234 Lincoln, Chicago, IL:
Kate:W:King:1976 Boone Trace, Chicago, IL:
Lyle:G:Leslie:417 Hancock Ave, Chicago, IL:
Billie:J:King:556 Washington, Chicago, IL:
Jon:A:Kramer:1988 Windy Creek, Seattle, WA:
Ray:H:King:213 Delk Road, Seattle, WA:
Gerald:D:Small:122 Ball Street, Dallas, TX:
Arnold:A:Head:233 Spring St, Dallas, TX:
Helga:C:Pataki:101 Holyoke St, Dallas, TX:
Naveen:B:Drew:198 Elm St, Philadelphia, PA:
Carl:E:Reedy:213 Ball St, Philadelphia, PA:
Sammy:G:Hall:433 Main Street, Miami, FL:
Red:A:Bacher:196 Elm Street, Miami, FL:

DLOG> exit;
Exiting...
```

```
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q8;
-----------------------------------
temp1(S) :-
  dependent(S,_,_,_,_).
answer(L) :-
  employee(_,_,L,S,_,_,_,_,_,_),
  department(_,_,S,_),
  not temp1(S).$
-----------------------------------
ANSWER(L:VARCHAR)

Number of tuples = 2
Borg:
James:

DLOG> exit;
Exiting...
[raj@tinman ch2]$
```

Exercises

1. Specify and execute the following queries using the RA interpreter on the COMPANY database schema.

    a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the 'ProductX' project.
    b. List the names of all employees who have a dependent with the same first name as themselves.
    c. Find the names of employees who are directly supervised by 'Franklin Wong'.
    d. Retrieve the names of employees who work on every project.
    e. Retrieve the names of employees who do not work on any project.
    f. Retrieve the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.
    g. Retrieve the last names of all department managers who have no dependents.

2. Consider the following MAILORDER relational schema describing the data for a mail order company:

```
parts(pno,pname,qoh,price,olevel)
customers(cno,cname,street,zip,phone)
employees(eno,ename,zip,hdate)
zipcodes(zip,city)
```

```
orders(ono,cno,eno,received,shipped)
odetails(ono,pno,qty)
```

The attribute names are self-explanatory. "qoh" stands for quantity on hand. Specify and execute the following queries using the RA interpreter on the MAILORDER database schema.

a. Retrieve the names of parts that cost less than $20.00.
b. Retrieve the names and cities of employees who have taken orders for parts costing more than $50.00
c. Retrieve the pairs of customer number values of customers who live in the same zip code.
d. Retrieve the names of customers who have ordered parts only from employees living in the city of Wichita.
e. Retrieve the names of customers who have ordered all parts costing less than $20.00.
f. Retrieve the names of customers who have not placed a single order.
g. Retrieve the names of customers who have placed exactly two orders.

3. Consider the following GRADEBOOK relational schema describing the data for a grade book of a particular instructor (Note: The attributes `A`, `B`, `C`, and `D` store grade cutoffs.)

```
catalog(cno,ctitle)
students(sid,fname,lname,minit)
courses(term,secno,cno,A,B,C,D)
enrolls(sid,term,secno)
```

Specify and execute the following queries using the RA interpreter on the GRADEBOOK database schema.

a. Retrieve the names of students enrolled in the '`Automata`' class in the term of `Fall 1996`.
b. Retrieve the SID values of students who have enrolled in `CSc226` as well as `CSc227`.
c. Retrieve the SID values of students who have enrolled in `CSc226` or `CSc227`.
d. Retrieve the names of students who have not enrolled in any class.
e. Retrieve the names of students who have enrolled in all courses in the `catalog` table.

4. Consider the database consisting of the following relations:

```
supplier(sno, sname)
part(pno, pname)
project(jno, jname)
supply(sno,pno,jno)
```

The database records information about suppliers, parts, and projects and includes a ternary relationship between suppliers, parts, and projects. This relationship is a many-many-many relationship. Specify and execute the following queries using the RA interpreter.

a. Retrieve part numbers of parts that are supplied to exactly two projects.
b. Retrieve supplier names of suppliers who supply more than two parts to project '*J1*'.
c. Retrieve part numbers of parts that are supplied by every supplier.
d. Retrieve project names of projects that are supplied only by suppliers '*S1*'.
e. Retrieve supplier names of suppliers who supply at least two different parts each to at least two different projects.

5. Specify and execute the following queries for the database in Exercise 5.16 of the Elmasri/Navathe text using the RA interpreter.

    a. Retrieve the names of students who have enrolled in a course that uses a textbook published by Addison Wesley.
    b. Retrieve the course names of courses which have changed their textbook at least once.
    c. Retrieve the names of departments that only adopt textbooks from Addison Wesley.
    d. Retrieve the names of departments that have adopted all textbooks written by Navathe and published by Addison Wesley in their courses.
    e. Retrieve the names of students who have never used a book (in a course) written by Navathe and published by Addison Wesley.

6. Repeat Exercises 1 through 5 in domain relational calculus (DRC) by using the DRC interpreter.

7. Repeat Exercises 1 through 5 in Datalog (DLOG) by using the DLOG interpreter.

CHAPTER 3

Relational Database Management System: Oracle™

This chapter introduces the student to the basic utilities used to interact with Oracle DBMS. The chapter also introduces the student to programming applications in Java and JDBC that interact with Oracle databases. The discussion in this chapter is not specific to any version of Oracle and all examples would work with any version of Oracle higher than Oracle 8.

The company database of the Elmasri/Navathe text is used throughout this chapter. In Section 3.1 a larger data set is introduced for the company database. In Section 3.2 and 3.3, the SQL*Plus and the SQL*Loader utilities are introduced using which the student can create database tables and load the tables with data. Finally, in Section 3.4, programming in Java using the JDBC API and connecting to Oracle databases is introduced through three complete application programs.

3.1 COMPANY Database

Consider the COMPANY database state shown in Figure 5.6 of the Elmasri/Navathe text. Let us assume that the company is expanding with 3 new departments: Software, Hardware, and Sales. The Software department has 2 locations: Atlanta and Sacramento, the Hardware department is located in Milwaukee, and the Sales department has 5 locations: Chicago, Dallas, Philadelphia, Seattle, and Miami.

The Software department has 3 new projects: OperatingSystems, DatabaseSystems, and Middleware and the Hardware department has 2 new projects: InkjetPrinters and LaserPrinters. The company has added 32 new employees in this expansion process.

The updated `COMPANY` database is shown in the following tables.

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	22-May-78
	Administration	4	987654321	1-Jan-85
	Headquarters	1	888665555	19-Jun-71
	Software	6	111111100	15-May-99
	Hardware	7	444444400	15-May-98
	Sales	8	555555500	1-Jan-97

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4
	OperatingSystems	61	Jacksonville	6
	DatabaseSystems	62	Birmingham	6
	Middleware	63	Jackson	6
	InkjetPrinters	91	Phoenix	7
	LaserPrinters	92	LasVegas	7

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston
	6	Atlanta
	6	Sacramento
	7	Milwaukee
	8	Chicago
	8	Dallas
	8	Philadephia
	8	Seattle

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATION
	333445555	Alice	F	5-Apr-76	Daughter
	333445555	Theodore	M	25-Oct-73	Son
	333445555	Joy	F	3-May-48	Spouse
	987654321	Abner	M	29-Feb-32	Spouse
	123456789	Michael	M	1-Jan-78	Son
	123456789	Alice	F	31-Dec-78	Daughter
	123456789	Elizabeth	F	5-May-57	Spouse
	444444400	Johnny	M	4-Apr-97	Son
	444444400	Tommy	M	7-Jun-99	Son
	444444401	Chris	M	19-Apr-69	Spouse
	444444402	Sam	M	14-Feb-64	Spouse

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40
	453453453	1	20
	453453453	2	20
	333445555	2	10
	333445555	3	10
	333445555	10	10
	333445555	20	10
	999887777	30	30
	999887777	10	10
	987987987	10	35
	987987987	30	5
	987654321	30	20
	987654321	20	15
	888665555	20	null
	111111100	61	40
	111111101	61	40
	111111102	61	40
	111111103	61	40
	222222200	62	40
	222222201	62	48
	222222202	62	40
	222222203	62	40
	222222204	62	40
	222222205	62	40
	333333300	63	40
	333333301	63	46
	444444400	91	40
	444444401	91	40
	444444402	91	40
	444444403	91	40
	555555500	92	40
	555555501	92	44
	666666601	91	40
	666666603	91	40
	666666604	91	40
	666666605	92	40
	666666606	91	40
	666666607	61	40
	666666608	62	40
	666666609	63	40
	666666610	61	40
	666666611	61	40
	666666612	61	40
	666666613	61	30
	666666613	62	10
	666666613	63	10

EMPLOYEE									
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DN
James	E	Borg	888665555	10-Nov-27	450 Stone, Houston, TX	M	55000	null	
Franklin	T	Wong	333445555	8-Dec-45	638 Voss, Houston, TX	M	40000	888665555	
Jennifer	S	Wallace	987654321	20-Jun-31	291 Berry, Bellaire, TX	F	43000	888665555	
John	B	Smith	123456789	9-Jan-55	731 Fondren, Houston, TX	M	30000	333445555	
Alicia	J	Zelaya	999887777	19-Jul-58	3321 Castle, Spring, TX	F	25000	987654321	
Ramesh	K	Narayan	666884444	15-Sep-52	971 Fire Oak, Humble, TX	M	38000	333445555	
Joyce	A	English	453453453	31-Jul-62	5631 Rice, Houston, TX	F	25000	333445555	
Ahmad	V	Jabbar	987987987	29-Mar-59	980 Dallas, Houston, TX	M	25000	987654321	
Jared	D	James	111111100	10-Oct-66	123 Peachtree, Atlanta, GA	M	85000	null	
Alex	D	Freed	444444400	9-Oct-50	4333 Pillsbury, Milwaukee, WI	M	89000	null	
John	C	James	555555500	30-Jun-75	7676 Bloomington, Sacramento, (	M	81000	null	
Jon	C	Jones	111111101	14-Nov-67	111 Allgood, Atlanta, GA	M	45000	111111100	
Justin	null	Mark	111111102	12-Jan-66	2342 May, Atlanta, GA	M	40000	111111100	
Brad	C	Knight	111111103	13-Feb-68	176 Main St., Atlanta, GA	M	44000	111111100	
Evan	E	Wallis	222222200	16-Jan-58	134 Pelham, Milwaukee, WI	M	92000	null	
Josh	U	Zell	222222201	22-May-54	266 McGrady, Milwaukee, WI	M	56000	222222200	
Andy	C	Vile	222222202	21-Jun-44	1967 Jordan, Milwaukee, WI	M	53000	222222200	
Tom	G	Brand	222222203	16-Dec-66	112 Third St, Milwaukee, WI	M	62500	222222200	
Jenny	F	Vos	222222204	11-Nov-67	263 Mayberry, Milwaukee, WI	F	61000	222222201	
Chris	A	Carter	222222205	21-Mar-60	565 Jordan, Milwaukee, WI	F	43000	222222201	
Kim	C	Grace	333333300	23-Oct-70	6677 Mills Ave, Sacramento, CA	F	79000	null	
Jeff	H	Chase	333333301	7-Jan-70	145 Bradbury, Sacramento, CA	M	44000	333333300	
Bonnie	S	Bays	444444401	19-Jun-56	111 Hollow, Milwaukee, WI	F	70000	444444400	
Alec	C	Best	444444402	18-Jun-66	233 Solid, Milwaukee, WI	M	60000	444444400	
Sam	S	Snedder	444444403	31-Jul-77	987 Windy St, Milwaukee, WI	M	48000	444444400	
Nandita	K	Ball	555555501	16-Apr-69	222 Howard, Sacramento, CA	M	62000	555555500	
Bob	B	Bender	666666600	17-Apr-68	8794 Garfield, Chicago, IL	M	96000	null	
Jill	J	Jarvis	666666601	14-Jan-66	6234 Lincoln, Chicago, IL	F	36000	666666600	
Kate	W	King	666666602	16-Apr-66	1976 Boone Trace, Chicago, IL	F	44000	666666600	
Lyle	G	Leslie	666666603	9-Jun-63	417 Hancock Ave, Chicago, IL	M	41000	666666601	
Billie	J	King	666666604	1-Jan-60	556 Washington, Chicago, IL	F	38000	666666603	
Jon	A	Kramer	666666605	22-Aug-64	1988 Windy Creek, Seattle, WA	M	41500	666666603	
Ray	H	King	666666606	16-Aug-49	213 Delk Road, Seattle, WA	M	44500	666666604	
Gerald	D	Small	666666607	15-May-62	122 Ball Street, Dallas, TX	M	29000	666666602	
Arnold	A	Head	666666608	19-May-67	233 Spring St, Dallas, TX	M	33000	666666602	
Helga	C	Pataki	666666609	11-Mar-69	101 Holyoke St, Dallas, TX	F	32000	666666602	
Naveen	B	Drew	666666610	23-May-70	198 Elm St, Philadelphia, PA	M	34000	666666607	
Carl	E	Reedy	666666611	21-Jun-77	213 Ball St, Philadelphia, PA	M	32000	666666610	
Sammy	G	Hall	666666612	11-Jan-70	433 Main Street, Miami, FL	M	37000	666666611	

3.2 *SQL*Plus* Utility

Oracle provides a command line utility, called SQL*Plus, that allows the user to execute SQL statements interactively. The user may enter the statements directly on the SQL*Plus prompt or may save the statements in a file and have them executed by entering the "start" command. We shall now look at how one can use this utility to create the database tables for the COMPANY database.

Creating Database Tables

Let us assume that all the SQL statements to create the COMPANY database tables are present in a file called company_schema.sql. The part that defines the DEPARTMENT table is shown below:

```
CREATE TABLE department (
  dname         varchar(25),
  dnumber       number(4),
  mgrssn        char(9),
  mgrstartdate date,
  primary key (dnumber),
  foreign key (mgrssn) references employee
);
```

The following SQL*Plus session illustrates how these statements would be executed resulting in the creation of the COMPANY database tables:

```
$ sqlplus
SQL>start company_schema

Table created

SQL>exit
$
```

The $ symbol used above is the command line prompt on a Unix system or a Windows command line interface.

In general, when several tables are defined in a single file, these definitions should be arranged in a particular order of foreign key references. For example, the definition of the WORKS_ON table should follow the definition of the EMPLOYEE table and the PROJECT table since it has foreign keys referencing the EMPLOYEE and the PROJECT tables.

Sometimes, it is possible for two tables to have a circular reference between them. For example, the foreign key constraints defined for the EMPLOYEE and the DEPARTMENT tables indicate a mutual dependency between the two tables. The EMPLOYEE table contains an attribute DNO which

refers to the DNUMBER attribute of the DEPARTMENT table and the MGRSSN attribute of the DEPARTMENT table refers to the SSN attribute of the EMPLOYEE table. This mutual dependency causes some trouble while creating the tables as well as loading the data. To avoid these problems, it may be simplest to omit the forward references, i.e. omit the DNO foreign key in the EMPLOYEE table.

A more appropriate way to handle the circular reference problem is as follows: Create the tables by omitting the forward references. After the tables are created, use the ALTER TABLE statement to add the omitted reference. In the case of the EMPLOYEE table, use the following ALTER command to add the forward reference after the DEPARTMENT table is created:

```
ALTER TABLE employee ADD (
  foreign key (dno) references department(dnumber)
);
```

The SQL*Plus utility is very versatile program and allows the user to submit any SQL statement for execution. In addition, it allows users to format query results in different ways. Please consult the SQL*Plus documentation that comes with any Oracle installation for further details on its use.

3.3 *SQL*Loader* Utility

Oracle provides a data loading utility called SQL*Loader[2] (sqlldr) that allows the user to populate the tables with data. To use sqlldr, we need a control file that indicates how that data is to be loaded and a data file that contains the data to be loaded. A typical control file is shown below:

```
LOAD DATA
INFILE <dataFile>
APPEND INTO TABLE <tableName>
FIELDS TERMINATED BY '<separator>'
(<attribute-list>)
```

Here, <datafile> is the name of the file containing the data, <tableName> is the database table into which the data is to be loaded, <separator> is a string that does not appear as part of the data, and <attribute-list> is the list of attributes or columns under which data is to be loaded. The attribute list need not be the entire list of columns and the attributes need not appear in the order in which they were defined in the create table statement,

As a concrete example, the following control file (department.ctl) will load data into the department table:

[2] For more details on this utility, please refer to Oracle's online documentation.

```
LOAD DATA
INFILE department.csv
APPEND INTO TABLE department
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(dname,dnumber,mgrssn,mgrstartdate DATE 'yyyy-mm-dd')
```

The `OPTIONALLY ENCLOSED` phrase is useful in situations when one of the data fields contained the separator symbol. In this case, the entire data value would be enclosed with a pair of double quotes. The `TRAILING NULLCOLS` phrase is useful in case there is a `null` value for the last data field. `Null` values are indicated by an empty string followed by the separator symbol. The corresponding data file, `department.csv` is:

```
Research,5,333445555,1978-05-22
Administration,4,987654321,1985-01-01
Headquarters,1,888665555,1971-06-19
Software,6,111111100,1999-05-15
Hardware,7,444444400,1998-05-15
Sales,8,555555500,1997-01-01
```

The following command will load the data into the department table:

```
$ sqlldr OracleId/OraclePassword control=department.ctl
```

Here, `OracleId/OraclePassword` is the Oracle user id/password. Upon execution of this command, a log file under the name `department.log` is created by `sqlldr`. This log file contains information such as the number of rows successfully loaded into the table, and any errors encountered while loading the data.

The loading of the data can be verified by the following `sqlplus` session:

```
$ sqlplus OracleId

SQL*Plus: Release 9.2.0.1.0 - Production on Wed Apr 6 20:35:45
2005

Copyright (c) 1982, 2002, Oracle Corporation.  All rights
reserved.

Enter password:

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
```

```
SQL> select * from department;

DNAME                          DNUMBER MGRSSN    MGRSTARTD
------------------------- ---------- --------- ---------
Research                            5 333445555 22-MAY-78
Administration                      4 987654321 01-JAN-85
Headquarters                        1 888665555 19-JUN-71
Software                            6 111111100 15-MAY-99
Hardware                            7 444444400 15-MAY-98
Sales                               8 555555500 01-JAN-97

6 rows selected.

SQL> exit
Disconnected from Oracle9i Enterprise Edition Release 9.2.0.1.0 -
64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 – Production
$
```

It is possible to load the data without a separate data file by using the following control file:

```
LOAD DATA
INFILE *
APPEND INTO TABLE department
FIELDS TERMINATED BY ','
(dname,dnumber,mgrssn,mgrstartdate DATE 'yyyy-mm-dd')
BEGINDATA
Research,5,333445555,1978-05-22
Administration,4,987654321,1985-01-01
Headquarters,1,888665555,1971-06-19
Software,6,111111100,1999-05-15
Hardware,7,444444400,1998-05-15
Sales,8,555555500,1997-01-01
```

Note that the filename has been replaced by a "`*`" in the `INFILE` clause of the control file and the data is separated from the control information by the line containing the string `BEGINDATA`.

In the case of foreign key dependencies between tables, data for the tables that are being referenced should be loaded/created first before the data for the tables that refer to values in other tables. For example, the `EMPLOYEE` and `PROJECT` table data should be loaded before the data for the `WORKS_ON` table is loaded.

The circular references are trickier to handle as far as bulk loading is concerned. To successfully load such data, we may use NULL values for forward references and once the referenced data is loaded, we may use the UPDATE statement to add values for the forward references.

3.4 Programming with Oracle using the JDBC API

This section presents Oracle JDBC programming through three complete applications. The first example illustrates basic query processing via PreparedStatement object in Java. The query result has at most one answer. The second example illustrates basic query processing using Statement object. In this example, the query may have more than one answer. The final example is a more involved one in which recursive query as well as aggregate query processing is discussed.

To be able to program with Oracle databases using Java/JDBC, one must have access to an Oracle database installation with a listener program for JDBC connections. One must also have access to the JDBC drivers (classes12.zip or classes12.jar) that comes with any standard Oracle distribution. The driver archive file must be included in the Java CLASSPATH variable. Any standard Java environment is sufficient to compile and run these programs.

The COMPANY database of the Elmasri/Navathe text is used in each of the three examples that follow.

Example 1: A simple Java/JDBC program that prints the last name and salary of an employee given his or her social security number is shown below:

```
import java.sql.*;
import java.io.*;

class getEmpInfo {
  public static void main (String args [])
      throws SQLException, IOException {

    // Load Oracle's JDBC Driver
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
        System.out.println ("Could not load the driver");
      }

    //Connect to the database
    String user, pass;
    user = readEntry("userid  : ");
    pass = readEntry("password: ");
    Connection conn =
```

```
      DriverManager.getConnection
        ("jdbc:oracle:thin:@tinman.cs.gsu.edu:1521:sid9ir2",
         user,pass);

    //Perform query using PreparedStatement object
    //by providing SSN at run time
    String query =
      "select LNAME,SALARY from EMPLOYEE where SSN = ?";
    PreparedStatement p = conn.prepareStatement (query);
    String ssn = readEntry("Enter a Social Security Number: ");
    p.clearParameters();
    p.setString(1,ssn);
    ResultSet r = p.executeQuery();

    //Process the ResultSet
    if (r.next ()) {
      String lname = r.getString(1);
      double salary = r.getDouble(2);
      System.out.println(lname + "  " + salary);
    }

    //Close objects
    p.close();
    conn.close();
  }

  //readEntry function -- to read input string
  static String readEntry(String prompt) {
     try {
       StringBuffer buffer = new StringBuffer();
       System.out.print(prompt);
       System.out.flush();
       int c = System.in.read();
       while(c != '\n' && c != -1) {
         buffer.append((char)c);
         c = System.in.read();
       }
       return buffer.toString().trim();
     } catch (IOException e) {
       return "";
       }
   }

}
```

In the above program, the user is prompted for a database user id and password. The program uses this information to connect[3] to the database. The user is then prompted for the social security number of an employee. The program supplies this social security number as a parameter to an SQL query used by a `PreparedStatement` object. The query is executed and the resulting `ResultSet` is then processed and the last name and salary of the employee is printed to the console.

The JDBC API is a very simple API to learn as there are an few important classes and methods that are required to manipulate relational data. After the driver is loaded and a connection is made (quite standard code to do this), queries and updates are submitted to the database via one of three objects: `Statement`, `PreparedStatement`, and `CallableStatement`.

The `PreparedStatement` method is illustrated in Example 1 where the SQL statement is sent to the database server with possible place-holders for pluggable values for compilation using the `prepareStatement` method of the `Connection` object. After this is done, the same SQL statement may be executed a number of times with values provided for the place holder variables using the `setString` or similar methods. In Example 1, the prepared statement is executed only once.

The `Statement` method is more commonly used and is illustrated in the following example.

Example 2: Given a department number, the following Java/JDBC program prints the last name and salary of all employees working for the department.

```
import java.sql.*;
import java.io.*;

class printDepartmentEmps {
  public static void main (String args [])
      throws SQLException, IOException {

    //Load Oracle's JDBC Driver
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
        System.out.println ("Could not load the driver");
      }

    //Connect to the database
    String user, pass;
    user = readEntry("userid  : ");
    pass = readEntry("password: ");
    Connection conn =
    DriverManager.getConnection
```

[3] The connect string will be different in different installations of Oracle. Please consult your instructor/administrator for the exact connect string used in `getConnection` method.

```
        ("jdbc:oracle:thin:@tinman.cs.gsu.edu:1521:sid9ir2",
         user,pass);

    //Perform query using Statement object
    String dno = readEntry("Enter a Department Number: ");
    String query =
      "select LNAME,SALARY from EMPLOYEE where DNO = " + dno;
    Statement s = conn.createStatement();
    ResultSet r = s.executeQuery(query);

    //Process ResultSet
    while (r.next ()) {
      String lname = r.getString(1);
      double salary = r.getDouble(2);
      System.out.println(lname + "  " + salary);
    }

    //Close objects
    s.close();
    conn.close();
  }
```

The query is executed via a `Statement` object by including the department number as part of the query string at run time (using string concatenation). This is in contrast to the `PreparedStatement` method used in Example 1. The main difference between the two methods is that in the `PreparedStatement` method, the query string is first sent to the database server for syntax checking and then for execution subsequently. This method may be useful when the same query string is executed a number of times in a program with only a different parameter each time. On the other hand, in the `Statement` method, syntax checking and execution of the query happens at the same time.

The third method for executing SQL statements is to use the CallableStatement object. This is useful only in situations where the Java program needs to call a stored procedure or function. To learn about this method, the reader is referred to any standard JDBC textbook or the Oracle 9i Programming: A Primer (2004) published by Addison Wesley.

Example 3: In this next program, the user is presented with a menu of 3 choices:

(a) **Find supervisees at all levels:** In this option, the user is prompted for the last name of an employee. If there are several employees with the same last name, the user is presented with a list of social security numbers of employees with the same last name and asked to choose one. The program then proceeds to list all the supervisees of the employee and all levels below him or her in the employee hierarchy.

(b) **Find the top 5 highest paid employees**: In this option, the program finds five employees who rank in the top 5 in salary and lists them.

(c) **Find the top 5 highest worked employees:** In this option, the program finds five employees who rank in the top 5 in number of hours worked and lists them.

A sample interaction with the user is shown below:

```
$ java example3
Enter userid: book
Enter password: book

        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.

Type in your option: a

Enter last name of employee : King
King,Kate 666666602
King,Billie 666666604
King,Ray 666666606
Select ssn from list : 666666602
     SUPERVISEES
FNAME          LNAME          SSN
---------------------------------------

Gerald         Small          666666607
Arnold         Head           666666608
Helga          Pataki         666666609
Naveen         Drew           666666610
Carl           Reedy          666666611
Sammy          Hall           666666612
Red            Bacher         666666613

        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.

Type in your option: b

    HIGHEST PAID WORKERS
-------------------------------------------------

666666600 Bob            Bender          96000.0
222222200 Evan           Wallis          92000.0
444444400 Alex           Freed           89000.0
111111100 Jared          James           85000.0
```

```
555555500 John            James           81000.0

        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.

Type in your option: c

    MOST WORKED WORKERS
-------------------------------------------------

666666613 Red             Bacher          50.0
222222201 Josh            Zell            48.0
333333301 Jeff            Chase           46.0
555555501 Nandita         Ball            44.0
111111100 Jared           James           40.0

        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.

Type in your option: q

$
```

The program for option (a) is discussed now. Finding supervisees at all levels below an employee is a recursive query in which the data tree needs to be traversed from the employee node all the way down to the leaves in the sub-tree. One common strategy to solve this problem is to use a temporary table of social security numbers. Initially, this temporary table will store the next level supervisees. Subsequently, in an iterative manner, the supervisees at the "next" lower level will be computed with a query that involves the temporary table. These next supervisees are then added to the temporary table. This iteration is continued as long as there are new social security numbers added to the temporary table in any particular iteration. Finally, when no new social security numbers are found, the iteration is stopped and the names and social security numbers of all supervisees in the temporary table are listed.

The section of the main program that (a) creates the temporary table, (b) calls the `findSupervisees` method, and (c) drops the temporary table is shown below:

```
    /* create new temporary table called tempSSN */
    String sqlString = "create table tempSSN (" +
                       "ssn char(9) not null, " +
                       "primary key(ssn))";
```

```
   Statement stmt1 = conn.createStatement();
   try {
      stmt1.executeUpdate(sqlString);
   } catch (SQLException e) {
       System.out.println("Could not create tempSSN table");
       stmt1.close();
       return;
     }

…
…
       case 'a': findSupervisees(conn);
…
…
   /* drop table called tmpSSN */
   sqlString = "drop table tempSSN";
   try {
      stmt1.executeUpdate(sqlString);
   } catch (SQLException e) {
     }
```

The `findSupervisees` method is divided into four stages.

Stage 1: The program prompts the user for input data (last name of employee) and queries the database for the social security number of employee. If there are several employees with same last name, the program lists all their social security numbers and prompts the user to choose one. At the end of this stage, the program would have the social security number of the employee whose supervisees the program needs to list. The code for this stage is shown below.

```
private static void findSupervisees(Connection conn)
     throws SQLException, IOException {

   String sqlString = null;

   Statement stmt = conn.createStatement();

   // Delete tuples from tempSSN from previous request.
   sqlString = "delete from tempSSN";
   try {
     stmt.executeUpdate(sqlString);
   } catch (SQLException e) {
       System.out.println("Could not execute Delete");
       stmt.close();
       return;
     }

   /* Get the ssn for the employee */
```

```
sqlString = "select lname, fname, ssn " +
            "from   employee " +
            "where  lname = '";
String lname =
   readEntry( "Enter last name of employee : ").trim();
sqlString += lname;
sqlString += "'";
ResultSet rset1;
try {
  rset1 = stmt.executeQuery(sqlString);
} catch (SQLException e) {
    System.out.println("Could not execute Query");
    stmt.close();
    return;
}
String samelName[] = new String[40];
String fName[] = new String[40];
String empssn[] = new String[40];
String ssn;
int nNames = 0;
while (rset1.next()) {
  samelName[nNames] = rset1.getString(1);
  fName[nNames] = rset1.getString(2);
  empssn[nNames] = rset1.getString(3);
  nNames++;
}
if (nNames == 0) {
  System.out.println("Name does not exist in database.");
  stmt.close();
  return;
}
else if (nNames > 1) {
  for(int i = 0; i < nNames; i++) {
    System.out.println(samelName[i] + "," +
                       fName[i] + " " + empssn[i]);
  }
  ssn = readEntry("Select ssn from list : ");
  ResultSet r = stmt.executeQuery(
       "select ssn from employee where ssn = '" + ssn + "'");
  if( !r.next()) {
    System.out.println("SSN does not exist in database.");
    stmt.close();
    return;
  }
}
else {
  ssn = empssn[0];
```

```
}
```

Stage 2: In the second stage, the `findSupervisees` method finds the immediate supervisees, i.e. supervisees who directly report to the employee. The social security numbers of immediate supervisees are then inserted into the temporary table.

```
/* Find immediate supervisees for that employee */
sqlString =
  "select distinct ssn from employee where superssn = '";
sqlString += ssn;
sqlString += "'";
try {
  rset1 = stmt.executeQuery(sqlString);
} catch (SQLException e) {
    System.out.println("Could not execute query");
    stmt.close();
    return;
}

/* Insert result into tempSSN table*/
Statement stmt1 = conn.createStatement();
while (rset1.next()) {
  String sqlString2 = "insert into tempSSN values ('";
  sqlString2 += rset1.getString(1);
  sqlString2 += "')";
  try {
    stmt1.executeUpdate(sqlString2);
  } catch (SQLException e) {
   }
}
```

Stage 3: In the third stage, the `findSupervisees` method iteratively calculates supervisees at the next lower level using the query:

```
select employee.ssn
from   employee, tempSSN
where  superssn = tempSSN.ssm;
```

The results of this query are inserted back into the `tempSSN` table to prepare for the next iteration. A boolean variable, called `newrowsadded`, is used to note if any new tuples were added to the `tempSSN` table in any particular iteration. Since the `tempSSN` table's only column (`ssn`) is also defined as a primary key, duplicate social security numbers would cause an SQLException which is simply ignored in this program. The code for this stage is shown below:

```
/* Recursive Querying */
ResultSet rset2;
```

```
boolean newrowsadded;
sqlString = "select employee.ssn from employee, tempSSN " +
            "where superssn = tempSSN.ssn";
do {
  newrowsadded = false;
  try {
    rset2 = stmt.executeQuery(sqlString);
  } catch (SQLException e) {
      System.out.println("Could not execute Query");
      stmt.close();
      stmt1.close();
      return;
  }
  while ( rset2.next()) {
    try {
      String sqlString2 = "insert into tempSSN values ('";
      sqlString2 += rset2.getString(1);
      sqlString2 += "')";
      stmt1.executeUpdate(sqlString2);
      newrowsadded = true;
    } catch (SQLException e) {
      }
  }
} while (newrowsadded);
stmt1.close();
```

Stage 4: In the final stage, the `findSupervisees` method prints names and social security numbers of all employees whose social security number is present in the `tempSSN` table. The code is shown below:

```
/* Print Results */
sqlString = "select fname, lname, e.ssn from " +
            "employee e, tempSSN t where e.ssn = t.ssn";
ResultSet rset3;
try {
  rset3 = stmt.executeQuery(sqlString);
} catch (SQLException e) {
    System.out.println("Could not execute Query");
    stmt.close();
    return;
 }
 System.out.println("     SUPERVISEES ");
 System.out.print("FNAME");
 for (int i = 0; i < 10; i++)
   System.out.print(" ");
 System.out.print("LNAME");
```

```
    for (int i = 0; i < 10; i++)
      System.out.print(" ");
    System.out.print("SSN");
    for (int i = 0; i < 6; i++)
      System.out.print(" ");
    System.out.println("\n---------------------------------\n");

    while(rset3.next()) {
      System.out.print(rset3.getString(1));
      for (int i = 0;
           i < (15 - rset3.getString(1).length()); i++)
        System.out.print(" ");
      System.out.print(rset3.getString(2));
      for (int i = 0;
           i < (15 - rset3.getString(2).length()); i++)
        System.out.print(" ");
      System.out.println(rset3.getString(3));
    }
    stmt.close();
  }
```

This concludes the discussion of the `findSupervisees` method. The code to implement options (b) and (c) is quite straightforward and is omitted. The entire code for all the examples is available along with this lab manual.

Exercises

1. Consider the ER-schema generated for the `UNIVERSITY` database in Laboratory Exercise 7.31 of the Elmasri/Navathe text. Convert the schema to a relational database and solve the following:
    a. Create the database tables in Oracle using the `SQL*Plus` interface.
    b. Create comma separated data files containing data for at least 3 departments (CS, Math, and Biology), 20 students, and 20 courses. You may evenly distribute the students and the courses among the three departments. You may also choose to assign minors to some of the students. For a subset of the courses, create sections for the Fall 2006 and Spring 2007 terms. Make sure that you assign multiple sections for some courses. Assuming that the grades are available for Fall 2006 term, add enrollments of students in sections and assign numeric grades for these enrollments. Do not add any enrollments for the Spring 2007 sections.
    c. Using the `SQL*Loader` utility of Oracle, load the data created in (b) into the database.
    d. Write SQL queries for the following and execute them within an `SQL*Plus` session.
        i. Retrieve student number, first and last names, and major department names of students who have a minor in the Biology department.

ii. Retrieve the student number and the first and last names of students who have never taken a class with instructor named "King".
iii. Retrieve the student number and the first and last names of students who have taken classes only with instructor named "King".
iv. Retrieve department names of departments along with the numbers of students with that department as their major (sorted in decreasing order of the number of students).
v. Retrieve department names of departments along with the numbers of students with that department as their major (sorted in decreasing order of the number of students).
vi. Retrieve the instructor names of all instructors teaching CS courses along with the sections (course number, section, semester, and year) they are teaching and the total number of students in these sections.
vii. Retrieve the student number, first and last names, and major departments of students who do not have a grade of "A" in any of the courses they have enrolled in.
viii. Retrieve the student number, first and last names, and major departments of all straight-A students (students who have a grade of "A" in all of the courses they have enrolled in).
ix. For each student in the CS department, retrieve their student number, first and last names, and their GPA.

2. Consider the ER-schema generated for the `MAIL_ORDER` database in Laboratory Exercise 7.32 of the Elmasri/Navathe text. Convert the schema to a relational database and solve the following:
    a. Create the database tables in Oracle using the `SQL*Plus` interface.
    b. Create comma separated data files containing data for at least 6 customers, 6 employees, and 20 parts. Also create data for 20 orders with an average of 3 to 4 parts in each order.
    c. Using the `SQL*Loader` utility of Oracle, load the data created in (b) into the database.
    d. Write SQL queries for the following and execute them within an `SQL*Plus` session.
        i. Retrieve the names of customers who have ordered at least one part costing more than $30.00.
        ii. Retrieve the names of customers who have ordered all parts that cost less than $20.00.
        iii. Retrieve the names of customers who order parts only from employees who live in the same city as they live in.
        iv. Retrieve a list of part number, part name, and total quantity ordered for that part. Produce a listing sorted in decreasing order of the total quantity ordered.
        v. Retrieve the average waiting time (number of days) for all orders. The waiting time is defined as the difference between shipping date and order received date rounded up to the nearest day.

vi. For each employee, retrieve his number, name, and total sales (in terms of dollars) in a given year, say 2005.

3. Consider the relational schema for the GRADE_BOOK database in Laboratory Exercise 7.3 of the Elmasri/Navathe text augmented with the following two tables:

```
components(Term,Sec_no,Comp_name,Max_points,Weight)
scores(Sid,Term,Sec_no,Comp_name,Points)
```

where the components table records the grading components for a course and the scores table records the points earned by a student in a grading component of the course.

a. Create the database tables in Oracle using the SQL*Plus interface.
b. Create comma separated data files containing data for at least 20 students and 3 courses with an average of 8 students in each course. Create data for 3 to 4 grading components for each course. Then, create data to assign points to each student in every course they are enrolled in for each of the grading components.
c. Using the SQL*Loader utility of Oracle, load the data created in (b) into the database.
d. Write SQL queries for the following and execute them within an SQL*Plus session.
    i. Retrieve the names of students who have enrolled in 'CSc 226' or 'CSc 227'.
    ii. Retrieve the names of students who enrolled in some class during fall 2005 but no class in spring 2006.
    iii. Retrieve the titles of courses that have had enrollments of five or fewer students.
    iv. Retrieve the student ids, their names, and their course average (weighted average of points from all components) for the CSc 226 course offering with section number 2 during Fall 2005.

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

4. Consider the UNIVERSITY database of Exercise 1. Write and test a Java program that performs the functions illustrated in the following terminal session:

```
$ java p1
Student Number: 1234
Semester: Fall
Year: 2005

Main Menu
            (1) Add a class
            (2) Drop a class
            (3) See my schedule
            (4) Exit
```

```
Enter your choice: 1

Course Number: CSC 1010
Section: 2
Class Added


Main Menu
            (1) Add a class
            (2) Drop a class
            (3) See my schedule
            (4) Exit
Enter your choice: 1

Course Number: MATH 2010
Section: 1
Class Added


Main Menu
            (1) Add a class
            (2) Drop a class
            (3) See my schedule
            (4) Exit
Enter your choice: 3

Your current schedule is:

CSC 1010 Section 2, Introduction to Computers, Instructor: Smith
MATH 2010 Section 1, Calculus I, Instructor: Jones


Main Menu
            (1) Add a class
            (2) Drop a class
            (3) See my schedule
            (4) Exit
Enter your choice: 2

Course Number: CSC 1010
Section: 2
Class dropped

Main Menu
            (1) Add a class
            (2) Drop a class
            (3) See my schedule
            (4) Exit
```

```
Enter your choice: 3

Your current schedule is:

MATH 2010 Section 1, Calculus I, Instructor: Jones


Main Menu
            1. Add a class
            2. Drop a class
            3. See my schedule
            4. Exit
Enter your choice: 4
$
```

As the terminal session shows, the student signs into this program with an id number and the term and year of registration. The `Add a class` option allows the student to add a class to his/her current schedule and the `Drop a class` option allows the student to drop a class in his/her current schedule. The `See my schedule` option lists the classes in which the student is registered.

5. Consider the `MAIL_ORDER` database of Exercise 2. Write and test a Java program that prints an invoice for a given order number as illustrated in the following terminal session:

```
$java p2
Order Number: 1020

Customer:      Charles Smith
Customer No:   1111
Zip:           67226

Order No:      1020
Taken By:      John Jones (emp. No. 1122)
Received on:   10-DEC-1994
Shipped on:    13-DEC-1994

Part No.  Part Name Quantity  Price     Sum
-------------------------------------------
10506     Nut       200       1.99      398.00
10509     Bolt      100       1.55      155.00
-------------------------------------------
                              Total:    553.00
```

6. Consider the GRADE_BOOK database of Exercise 3. Write and test a Java program that interacts with the Oracle database and prints a grade count report for each course. The report should list the number of A's, B's, C's, D's, and F's given in each of the courses sorted on course number. The format for the report is:

```
                    Grade Count Report
CNO       Course Title   #A's #B's #C's #D's #F's
-------------------------------------------------
CSc2010   Intro to CS    12   8    15   5    2
CSc2310   Java           15   2    12   8    1
-------------------------------------------------
```

CHAPTER 4

Relational Database Management System: MySQL

This chapter introduces the student to the MySQL database management system and PHP, the programming language used to program applications that access a MySQL database. The discussion in this chapter is not specific to any version of MySQL and all examples would work with MySQL 4.0 or higher version.

The `COMPANY` database of the Elmasri/Navathe text is used throughout this chapter. In Section 4.1, a larger data set is introduced for the `COMPANY` database. In Section 4.2, the `mysql` utility is introduced which allows the users to interact with the database including running commands, executing SQL statements, and running MySQL scripts. In Section 4.3 PHP programming with MySQL is introduced through a complete Web browsing application for the `COMPANY` database. Finally, in Section 4.4 an online address book application is discussed with interfaces for adding, deleting, listing and searching a collection of contacts coded in HTML as well as in PhP.

4.1 COMPANY Database

Consider the COMPANY database state shown in Figure 5.6. Let us assume that the company is expanding with 3 new departments: Software, Hardware, and Sales. The Software department has 2 locations: Atlanta and Sacramento, the Hardware department is located in Milwaukee, and the Sales department has 5 locations: Chicago, Dallas, Philadelphia, Seattle, and Miami.

The `Software` department has 3 new projects: `OperatingSystems`, `DatabaseSystems`, and `Middleware` and the `Hardware` department has 2 new projects: `InkjetPrinters` and `LaserPrinters`. The company has added 32 new employees in this expansion process.

The updated `COMPANY` database is shown in the following tables.

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	22-May-78
	Administration	4	987654321	1-Jan-85
	Headquarters	1	888665555	19-Jun-71
	Software	6	111111100	15-May-99
	Hardware	7	444444400	15-May-98
	Sales	8	555555500	1-Jan-97

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4
	OperatingSystems	61	Jacksonville	6
	DatabaseSystems	62	Birmingham	6
	Middleware	63	Jackson	6
	InkjetPrinters	91	Phoenix	7
	LaserPrinters	92	LasVegas	7

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston
	6	Atlanta
	6	Sacramento
	7	Milwaukee
	8	Chicago
	8	Dallas
	8	Philadephia
	8	Seattle

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATION
	333445555	Alice	F	5-Apr-76	Daughter
	333445555	Theodore	M	25-Oct-73	Son
	333445555	Joy	F	3-May-48	Spouse
	987654321	Abner	M	29-Feb-32	Spouse
	123456789	Michael	M	1-Jan-78	Son
	123456789	Alice	F	31-Dec-78	Daughter
	123456789	Elizabeth	F	5-May-57	Spouse
	444444400	Johnny	M	4-Apr-97	Son
	444444400	Tommy	M	7-Jun-99	Son
	444444401	Chris	M	19-Apr-69	Spouse
	444444402	Sam	M	14-Feb-64	Spouse

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40
	453453453	1	20
	453453453	2	20
	333445555	2	10
	333445555	3	10
	333445555	10	10
	333445555	20	10
	999887777	30	30
	999887777	10	10
	987987987	10	35
	987987987	30	5
	987654321	30	20
	987654321	20	15
	888665555	20	null
	111111100	61	40
	111111101	61	40
	111111102	61	40
	111111103	61	40
	222222200	62	40
	222222201	62	48
	222222202	62	40
	222222203	62	40
	222222204	62	40
	222222205	62	40
	333333300	63	40
	333333301	63	46
	444444400	91	40
	444444401	91	40
	444444402	91	40
	444444403	91	40
	555555500	92	40
	555555501	92	44
	666666601	91	40
	666666603	91	40
	666666604	91	40
	666666605	92	40
	666666606	91	40
	666666607	61	40
	666666608	62	40
	666666609	63	40
	666666610	61	40
	666666611	61	40
	666666612	61	40
	666666613	61	30
	666666613	62	10
	666666613	63	10

EMPLOYEE									
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DN
James	E	Borg	888665555	10-Nov-27	450 Stone, Houston, TX	M	55000	null	
Franklin	T	Wong	333445555	8-Dec-45	638 Voss, Houston, TX	M	40000	888665555	
Jennifer	S	Wallace	987654321	20-Jun-31	291 Berry, Bellaire, TX	F	43000	888665555	
John	B	Smith	123456789	9-Jan-55	731 Fondren, Houston, TX	M	30000	333445555	
Alicia	J	Zelaya	999887777	19-Jul-58	3321 Castle, Spring, TX	F	25000	987654321	
Ramesh	K	Narayan	666884444	15-Sep-52	971 Fire Oak, Humble, TX	M	38000	333445555	
Joyce	A	English	453453453	31-Jul-62	5631 Rice, Houston, TX	F	25000	333445555	
Ahmad	V	Jabbar	987987987	29-Mar-59	980 Dallas, Houston, TX	M	25000	987654321	
Jared	D	James	111111100	10-Oct-66	123 Peachtree, Atlanta, GA	M	85000	null	
Alex	D	Freed	444444400	9-Oct-50	4333 Pillsbury, Milwaukee, WI	M	89000	null	
John	C	James	555555500	30-Jun-75	7676 Bloomington, Sacramento, (	M	81000	null	
Jon	C	Jones	111111101	14-Nov-67	111 Allgood, Atlanta, GA	M	45000	111111100	
Justin	null	Mark	111111102	12-Jan-66	2342 May, Atlanta, GA	M	40000	111111100	
Brad	C	Knight	111111103	13-Feb-68	176 Main St., Atlanta, GA	M	44000	111111100	
Evan	E	Wallis	222222200	16-Jan-58	134 Pelham, Milwaukee, WI	M	92000	null	
Josh	U	Zell	222222201	22-May-54	266 McGrady, Milwaukee, WI	M	56000	222222200	
Andy	C	Vile	222222202	21-Jun-44	1967 Jordan, Milwaukee, WI	M	53000	222222200	
Tom	G	Brand	222222203	16-Dec-66	112 Third St, Milwaukee, WI	M	62500	222222200	
Jenny	F	Vos	222222204	11-Nov-67	263 Mayberry, Milwaukee, WI	F	61000	222222201	
Chris	A	Carter	222222205	21-Mar-60	565 Jordan, Milwaukee, WI	F	43000	222222201	
Kim	C	Grace	333333300	23-Oct-70	6677 Mills Ave, Sacramento, CA	F	79000	null	
Jeff	H	Chase	333333301	7-Jan-70	145 Bradbury, Sacramento, CA	M	44000	333333300	
Bonnie	S	Bays	444444401	19-Jun-56	111 Hollow, Milwaukee, WI	F	70000	444444400	
Alec	C	Best	444444402	18-Jun-66	233 Solid, Milwaukee, WI	M	60000	444444400	
Sam	S	Snedder	444444403	31-Jul-77	987 Windy St, Milwaukee, WI	M	48000	444444400	
Nandita	K	Ball	555555501	16-Apr-69	222 Howard, Sacramento, CA	M	62000	555555500	
Bob	B	Bender	666666600	17-Apr-68	8794 Garfield, Chicago, IL	M	96000	null	
Jill	J	Jarvis	666666601	14-Jan-66	6234 Lincoln, Chicago, IL	F	36000	666666600	
Kate	W	King	666666602	16-Apr-66	1976 Boone Trace, Chicago, IL	F	44000	666666600	
Lyle	G	Leslie	666666603	9-Jun-63	417 Hancock Ave, Chicago, IL	M	41000	666666601	
Billie	J	King	666666604	1-Jan-60	556 Washington, Chicago, IL	F	38000	666666603	
Jon	A	Kramer	666666605	22-Aug-64	1988 Windy Creek, Seattle, WA	M	41500	666666603	
Ray	H	King	666666606	16-Aug-49	213 Delk Road, Seattle, WA	M	44500	666666604	
Gerald	D	Small	666666607	15-May-62	122 Ball Street, Dallas, TX	M	29000	666666602	
Arnold	A	Head	666666608	19-May-67	233 Spring St, Dallas, TX	M	33000	666666602	
Helga	C	Pataki	666666609	11-Mar-69	101 Holyoke St, Dallas, TX	F	32000	666666602	
Naveen	B	Drew	666666610	23-May-70	198 Elm St, Philadelphia, PA	M	34000	666666607	
Carl	E	Reedy	666666611	21-Jun-77	213 Ball St, Philadelphia, PA	M	32000	666666610	
Sammy	G	Hall	666666612	11-Jan-70	433 Main Street, Miami, FL	M	37000	666666611	

4.2 `mysql` Utility

MySQL database system provides an interactive utility, called `mysql`, which allows the user to enter SQL commands interactively. One can also include one or more SQL statements in a file and have them executed within `mysql` using the `source` command.

In the following discussion, we will assume that your MySQL administrator has created a database called `company`[4], created a MySQL user[5] called `book`, and has granted all rights to the `company` database to the `book` user.

The following `mysql` session creates the department table:

```
$ mysql -u book -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1485 to server version: 4.1.9-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use company
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> source create-department.sql;
Query OK, 0 rows affected (0.04 sec)

mysql> show tables;
+------------------+
| Tables_in_company |
+------------------+
| department       |
| foo              |
+------------------+
2 rows in set (0.00 sec)

mysql> exit;
Bye
$
```

[4] The administrator command to create a MySQL user is: `create user book identified by 'book';` Here the user id is `book` and the password is also `book`.

[5] The administrator command to create a database called `company` and to assign all rights to the `book` user is: `grant all on company.* to 'book'@'hostname.domain.edu';`

In the above `mysql` session, the user invokes the `mysql` program with the userid `book` and the `-p` option to specify the password in a separate line. After connecting to the database server, the `use company` command is executed to start using the company database. Then, the `source` command is executed on the file `create-department.sql` that contains the following SQL statement:

```
CREATE TABLE department (
  dname        varchar(25) not null,
  dnumber      integer(4),
  mgrssn       char(9) not null,
  mgrstartdate date,
  primary key (dnumber),
  key (dname)
);
```

Note: The data types supported in different DBMSs may have slightly different names; Please consult MySQL documentation to learn about all data types supported.

Bulk Loading of Data

MySQL provides a "`load`" command that can load data stored in a text file into a table. The following `mysql` session illustrates the loading of the data located in `department.csv` file into the `department` table. In the following example, the load command is directly entered at the `mysql` prompt. It may be a good idea to create a script file with the `load` command and use the `source` command to execute the `load` command. This way, if there are any syntax errors, these can be corrected in the script file and the `load` command can be re-executed.

```
$ mysql -u raj -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1493 to server version: 4.1.9-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use company;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> LOAD DATA LOCAL INFILE "department.csv"
INTO TABLE department FIELDS TERMINATED BY ","
OPTIONALLY ENCLOSED BY '"';
Query OK, 6 rows affected (0.00 sec)
Records: 6  Deleted: 0  Skipped: 0  Warnings: 0
```

```
mysql> select * from department;
+----------------+---------+-----------+--------------+
| dname          | dnumber | mgrssn    | mgrstartdate |
+----------------+---------+-----------+--------------+
| Research       |       5 | 333445555 | 1978-05-22   |
| Administration |       4 | 987654321 | 1985-01-01   |
| Headquarters   |       1 | 888665555 | 1971-06-19   |
| Software       |       6 | 111111100 | 1999-05-15   |
| Hardware       |       7 | 444444400 | 1998-05-15   |
| Sales          |       8 | 555555500 | 1997-01-01   |
+----------------+---------+-----------+--------------+
6 rows in set (0.00 sec)

mysql> exit;
Bye
$
```

The `LOAD DATA` command takes the name of the data file as parameter and other information such as field terminator symbols (in this case the comma) and loads the data into the table. For more details on the `load` command, please consult the MySQL documentation.

4.3 MySQL and PHP Programming

PHP is a very popular Web scripting language that allows the programmers to rapidly develop Web applications. In particular, PHP is most suited to develop Web applications that access a MySQL database. In this section, we illustrate the ease of programming with PHP and provide several examples of Web access to MySQL databases.

Example 1: Consider the problem of finding employee names given their social security number. To implement this problem as a Web application, we can design two Web pages:

1. The first Web page would contain a HTML form that contains a select list of social security numbers of employees and a submit button.

2. Upon choosing a social security number and submitting the form in the first Web page produces the second Web page that lists the name of the employee.

The two Web pages are shown in Figures 4.1 and 4.2.

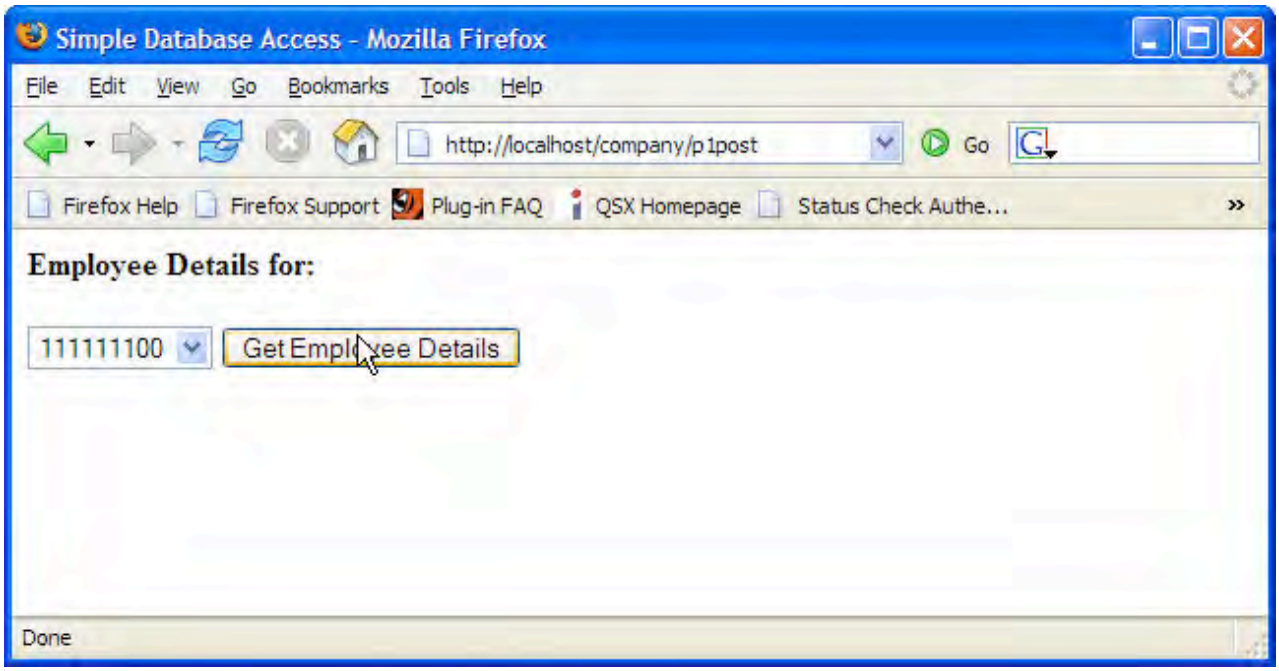

Figure 4.1: Initial Web Page - Example 1

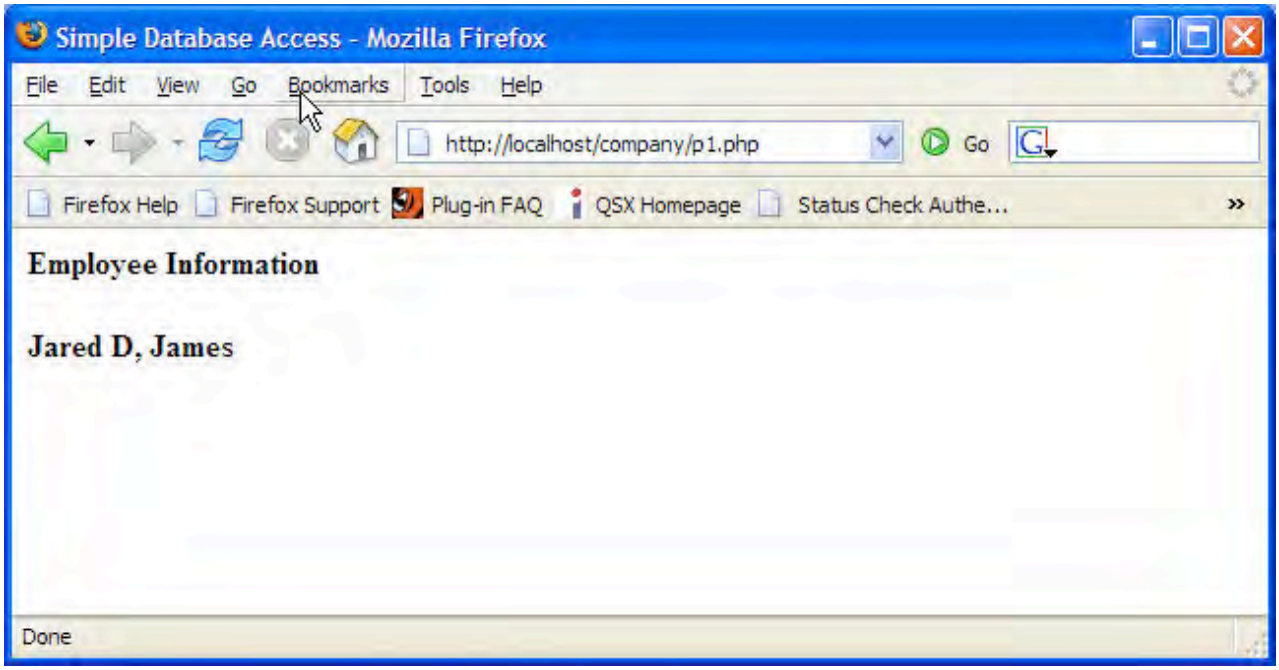

Figure 4.2: Second Web Page - Example 1

Both these Web pages contain dynamic information (obtained from the database) and therefore can easily be produced by PHP scripts. The PHP script (`p1post.php`) that produces the first Web page is shown below.

```
<html>
<head>
<title>Simple Database Access</title>
</head>
<body>

<?
$username="user";
$password="password";
$database="company";
mysql_connect(localhost,$username,$password);
@mysql_select_db($database) or die( "Unable to select database");
$query="SELECT ssn FROM employee";
$result=mysql_query($query);
$num=mysql_numrows($result);
mysql_close();
?>

<h4>Employee Details for:</h4>
<form method="post" action="p1.php">
<select name="ssn">

<?
$i=0;
while ($i < $num) {
  $ssn=mysql_result($result,$i,"ssn");
  echo "<option>",$ssn,"\n";
  $i++;
}
?>

</select>
<input type="submit" value="Get Employee Details">
</form>
</body>
</html>
```

A PHP script typically consists of HTML code to display "static" parts of the Web page interspersed with procedural PHP statements that produce "dynamic" parts of the Web page. The dynamic content may come from a database such as MySQL and hence most of the PHP procedural code involves connecting to the database, running queries and using the query result to produce parts of the Web page.

In the above example script, a simple HTML page is produced which has:

- Some static content such as text headers and a HTML form with a submit button. The HTML form when submitted invokes the second PHP script called "`p1.php`".

- A dynamic “select” GUI element within the HTML form which contains a list of employee social security numbers for the users to choose from.

PHP statements are enclosed within <? And >?. HTML code can be produced within PHP code using the “echo” command as is seen in several places in the code. As can be seen, there are two blocks of PHP code in the example: one to connect to the database and execute a query and the second to use the results of the query to produce the HTML “select” list options.

The PHP script (p1.php) to produce the second Web page is shown next.

```
<html>
<head>
<title>Simple Database Access</title>
</head>
<body>
<h4>Employee Information</h4>

<?
$username="user";
$password="password";
$database="company";
$ssn=$_POST['ssn'];
mysql_connect(localhost,$username,$password);
@mysql_select_db($database) or die( "Unable to select database");
$query="SELECT * FROM employee where ssn=$ssn";
$result=mysql_query($query);
$num=mysql_numrows($result);
mysql_close();
if ($num == 1) {
  $fname=mysql_result($result,$i,"fname");
  $minit=mysql_result($result,$i,"minit");
  $lname=mysql_result($result,$i,"lname");
  echo "<b>$fname $minit, $lname</b>";
}?>

</body>
</html>
```

In this script, the employee social security number posted in the first Web page is retrieved using the PHP statement

```
$ssn=$_POST['ssn'];
```

This social security number is then used in an SQL query to retrieve employee name. The script contains one block of PHP code surrounded by some HTML code.

Example 2: In this example, a PHP script that lists all employees in a given department is shown. The script takes as input the department number as a "GET" parameter in the URL itself as follows:

```
http://localhost/company/p2?dno=4
```

The Web page produced by the script is shown in Figure 4.3.

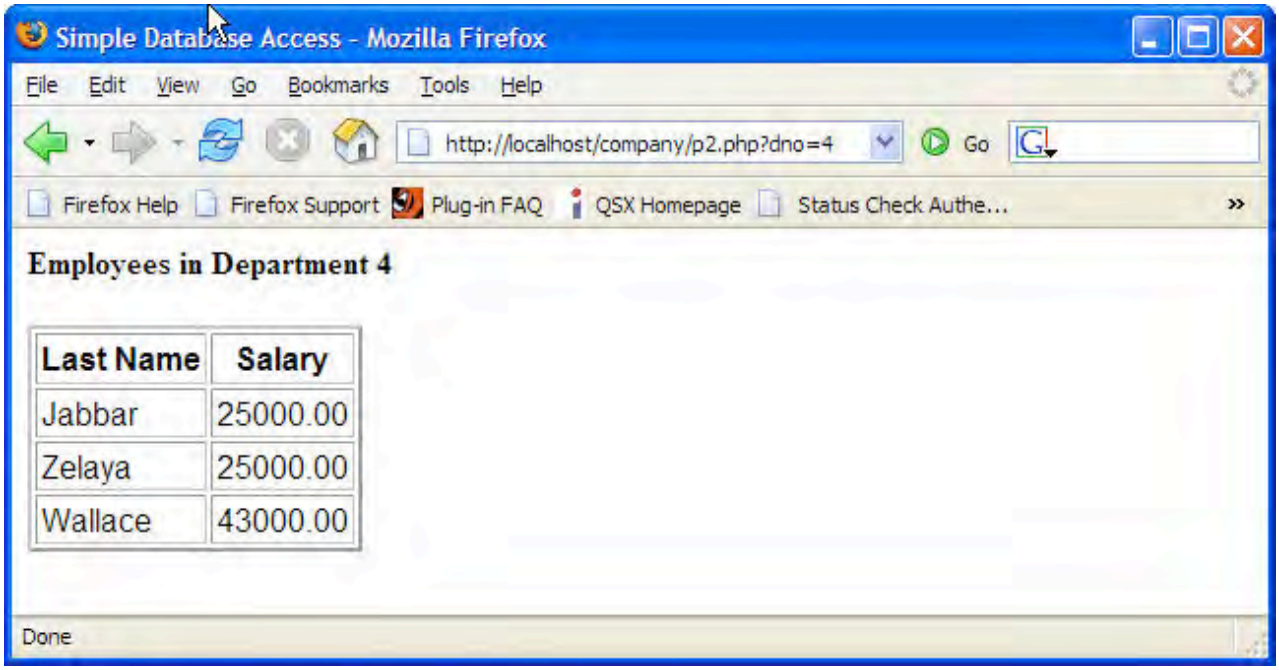

Figure 4.3: Web Page for Example 2

The PHP code (`p2.php`) is shown below:

```
<html>
<head>
<title>Simple Database Access</title>
</head>
<body>

<?
$username="user";
$password="password";
$database="company";
$dno=$_GET['dno'];
mysql_connect(localhost,$username,$password);
@mysql_select_db($database) or die( "Unable to select database");
$query="SELECT lname,salary FROM employee where dno=$dno";
$result=mysql_query($query);
```

```
$num=mysql_numrows($result);
mysql_close();
?>

<table border="2" cellspacing="2" cellpadding="2">
<tr>
<th><font face="Arial,Helvetica,sans-serif">Last Name</font></th>
<th><font face="Arial,Helvetica,sans-serif">Salary</font></th>
</tr>

<?
echo "<h4>Employees in Department $dno</h4>";
$i=0;
while ($i < $num) {
  $lname=mysql_result($result,$i,"lname");
  $salary=mysql_result($result,$i,"salary");
?>

<tr>
<td><font face="Arial, Helvetica, sans-serif">
<? echo $lname; ?>
</font></td>
<td><font face="Arial, Helvetica, sans-serif">
<? echo $salary; ?>
</font></td>
</tr>

<?
  $i++;
}
?>

</table>
</body>
</html>
```

The PHP script performs an SQL query to retrieve employee names and salaries who work for the given department. This information is then formatted neatly into an HTML table for display. This example illustrates more intricate embedding of PHP code within HTML code as is seen in the "`while`" loop towards the end of the script.

Example 3: A COMPANY database browser application is shown in this example. The initial Web page in this application lists all the departments in the company. By following hyperlinks, the user may see more details of departments, employees, and projects in three separate Web pages. The browser program is implemented using four PHP scripts:

(a) `companyBrowse.php`: This script lists all the departments in the company in a tabular form as shown in Figure

(b) deptView.php:
(c) empView.php:
(d) projectView.php:

The code for companyBrowse script is shown below:

```
<html>
<head>
<title>All Departments</title>
</head>
<body>

<?
$username="user";
$password="password";
$database="company";

mysql_connect("host.domain.edu",$username,$password);
@mysql_select_db($database) or die( "Unable to select database");
$query="SELECT dnumber,dname FROM department order by dnumber";
$result=mysql_query($query);
$num=mysql_numrows($result);
mysql_close();
?>

<h4>Departments of Company</h4>
<table border="2" cellspacing="2" cellpadding="2">
<tr>
<th><font face="Arial, Helvetica, sans-serif">
    Department Number</font></th>
<th><font face="Arial, Helvetica, sans-serif">
    Department Name</font></th>
</tr>

<?
$i=0;
while ($i < $num) {
  $dno=mysql_result($result,$i,"dnumber");
  $dname=mysql_result($result,$i,"dname");
?>

<tr>
<td><font face="Arial, Helvetica, sans-serif">
  <a href="deptView?dno=<? echo $dno; ?>">
  <? echo $dno; ?></a></font></td>
<td><font face="Arial, Helvetica, sans-serif">
  <? echo $dname; ?></font></td>
```

```
</tr>

<?
   $i++;
}
?>

</table>
</body>
</html>
```

The script performs a simple query on the `DEPARTMENT` table and outputs the list of department numbers and names formatted as an HTML table as shown in Figure 4.4.

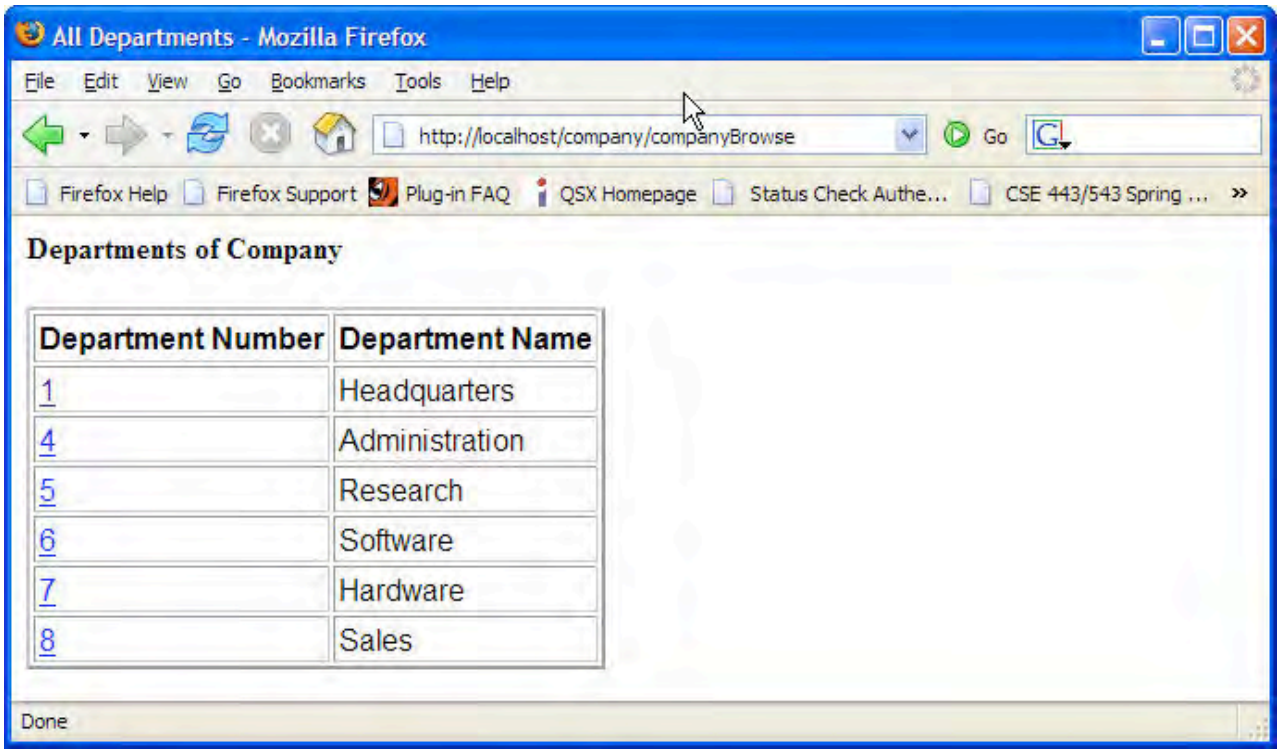

Figure 4.4: All Departments Web Page – COMPANY database browser

The department numbers in this list are formatted as HTML hyperlinks that, when traversed by the user, will produce a Web page containing more details of the chosen department. The detailed department view Web page is shown in Figure 4.5.

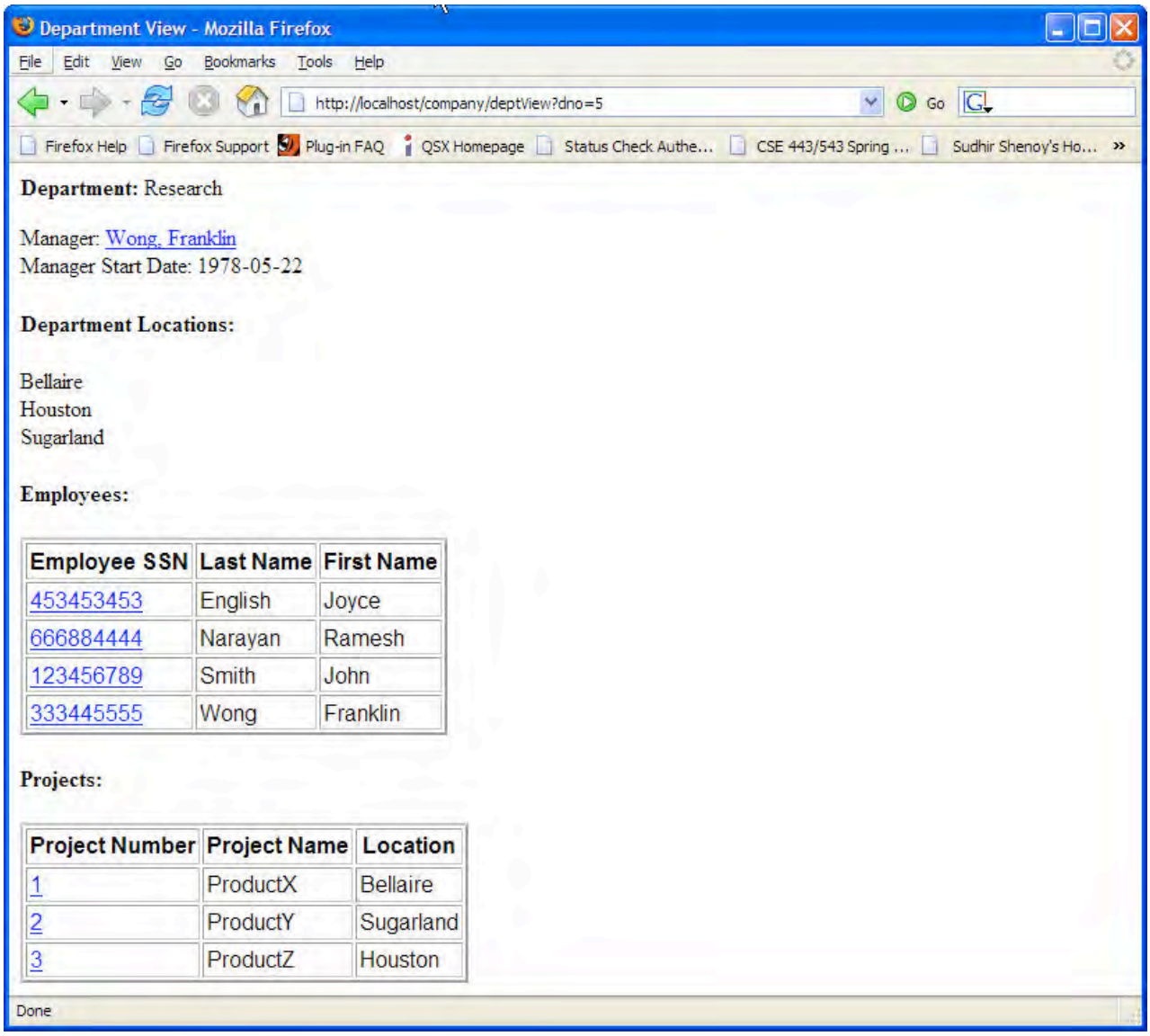

Figure 4.5: Department Detail Web Page – COMPANY database browser

The PHP script (`deptView.php`) that produces the detailed department view is shown below.

```
<html>
<head>
<title>Department View</title>
</head>
<body>

<?
$username="user";
$password="password";
$database="company";
```

```
$dno=$_GET['dno'];
mysql_connect("host.domain.edu",$username,$password);
@mysql_select_db($database) or die( "Unable to select database");

$query="SELECT dname,mgrssn,mgrstartdate,lname,fname FROM
department,employee where dnumber=$dno and mgrssn=ssn";
$result=mysql_query($query);
$num=mysql_numrows($result);

$dname=mysql_result($result,0,"dname");
$mssn=mysql_result($result,0,"mgrssn");
$mstart=mysql_result($result,0,"mgrstartdate");
$mlname=mysql_result($result,0,"lname");
$mfname=mysql_result($result,0,"fname");

echo "<b>Department: </b>", $dname;
echo "<P>Manager: <a href=\"empView?", $mssn, "\">", $mlname, ",
", $mfname, "</a></BR>";
echo "Manager Start Date: ", $mstart;

echo "<h4>Department Locations:</h4>";
$query="SELECT dlocation FROM dept_locations where dnumber=$dno";
$result=mysql_query($query);
$num=mysql_numrows($result);
$i=0;
while ($i < $num) {
  $dloc=mysql_result($result,$i,"dlocation");
  echo $dloc, "<BR>\n";
  $i++;
}

echo "<h4>Employees:</h4>";
$query="SELECT ssn,lname,fname FROM employee where dno=$dno";
$result=mysql_query($query);
$num=mysql_numrows($result);
?>

<table border="2" cellspacing="2" cellpadding="2">
<tr>
<th><font face="Arial, Helvetica, sans-serif">
     Employee SSN</font></th>
<th><font face="Arial, Helvetica, sans-serif">
     Last Name</font></th>
<th><font face="Arial, Helvetica, sans-serif">
     First Name</font></th>
</tr>
```

```
<?
$i=0;
while ($i < $num) {
  $essn=mysql_result($result,$i,"ssn");
  $elname=mysql_result($result,$i,"lname");
  $efname=mysql_result($result,$i,"fname");
?>
<tr>
  <td><font face="Arial, Helvetica, sans-serif">
  <a href="empView?ssn=<? echo $essn; ?>">
     <? echo $essn; ?></a></font></td>
<td><font face="Arial, Helvetica, sans-serif">
     <? echo $elname; ?></font></td>
<td><font face="Arial, Helvetica, sans-serif">
     <? echo $efname; ?></font></td>
</tr>
<?
  $i++;
}
?>

</table>

<?
echo "<h4>Projects:</h4>";
$query="SELECT pnumber,pname,plocation FROM project where
dnum=$dno";
$result=mysql_query($query);
$num=mysql_numrows($result);
?>

<table border="2" cellspacing="2" cellpadding="2">
<tr>
<th><font face="Arial, Helvetica, sans-serif">Project
Number</font></th>
<th><font face="Arial, Helvetica, sans-serif">Project
Name</font></th>
<th><font face="Arial, Helvetica, sans-serif">Location</font></th>
</tr>

<?
$i=0;
while ($i < $num) {
  $pnum=mysql_result($result,$i,"pnumber");
  $pname=mysql_result($result,$i,"pname");
  $ploc=mysql_result($result,$i,"plocation");
?>
```

```
<tr>
  <td><font face="Arial, Helvetica, sans-serif">
  <a href="projView?ssn=<? echo $pnum; ?>"><? echo $pnum;
?></a></font></td>
<td><font face="Arial, Helvetica, sans-serif"><? echo $pname;
?></font></td>
<td><font face="Arial, Helvetica, sans-serif"><? echo $ploc;
?></font></td>
</tr>
<?
  $i++;
}

mysql_close();
?>

</body>
</html>
```

The `deptView` script executes the following four queries and formats the results of the queries as shown in the Web page.

```
SELECT dname,mgrssn,mgrstartdate,lname,fname
FROM   department,employee
WHERE  dnumber=$dno and mgrssn=ssn

SELECT dlocation
FROM   dept_locations
WHERE  dnumber=$dno

SELECT ssn,lname,fname
FROM   employee
WHERE  dno=$dno

SELECT pnumber,pname,plocation
FROM   project
WHERE  dnum=$dno
```

Each of these queries uses the PHP variable `$dno` containing the department number for which the view is generated.

The department view also contains hyperlinks for employees and projects which when traversed by the user produces detailed employee and project Web pages. The code for PHP scripts that produce these Web pages is omitted as they are similar to the `deptView` script.

4.4 Online Address Book

In this section an online address/contact book application is illustrated. The application is coded in PhP and accesses a MySQL database which has the following table:

```
create table contacts (
  lname  varchar(30),
  fname  varchar(30),
  email  varchar(30),
  homePhone varchar(20),
  cellPhone varchar(20),
  officePhone varchar(20),
  address varchar(100),
  comment varchar(100),
  primary key (lname,fname)
);
```

The application is capable of the following functions:

(1) ADD a new contact.
(2) DELETE one or more contacts.
(3) SEARCH contacts by substring match on name.
(4) LIST all contacts.

The initial Web page is shown in Figure 4.6. On the left frame, the menu options are listed and the right frame contains space to display results.

Figure 4.6: Address Book – Main Page

The HTML code that produces this (`index.html`) page is shown below:

```
<html>
<head>
<title>Addressbook</title>
</head>
<frameset rows="*" cols="18%,*" framespacing="0"
          frameborder="no" border="0">
  <frame src="mainMenu.html" name="leftFrame" scrolling="No"
         noresize="noresize" id="leftFrame" />
  <frame src="Welcome.html" name="mainFrame" id="mainFrame" />
</frameset>
</html>
```

The HTML code that produces the left frame (`mainMenu.html`) is shown below:

```
<html>
<head>
<title>Addressbook</title>
</head>
<body>
  <p>
  <h2>   Address Book</h2>
  <h3><a href="add.html" target="mainFrame">Add</a></h3>
  <h3><a href="delete.php" target="mainFrame">Delete</a></h3>
  <h3><a href="search.html" target="mainFrame">Search</a> </h3>
  <h3><a href="list.php" target="mainFrame">List All</a></h3>
</body>
</html>
```

The HTML code (`Welcome.html`) to produce the right frame is shown below:

```
<html>
  <head><title>Addressbook</title></head>
  <body>
    <BR><BR><BR><BR><BR><BR><BR>
    <center>
    <h2>Welcome to the Addressbook App</h2>
    </center>
  </body>
</html>
```

Add

Upon clicking the “Add” menu option, the form shown in Figure 4.7 is displayed on the right frame:

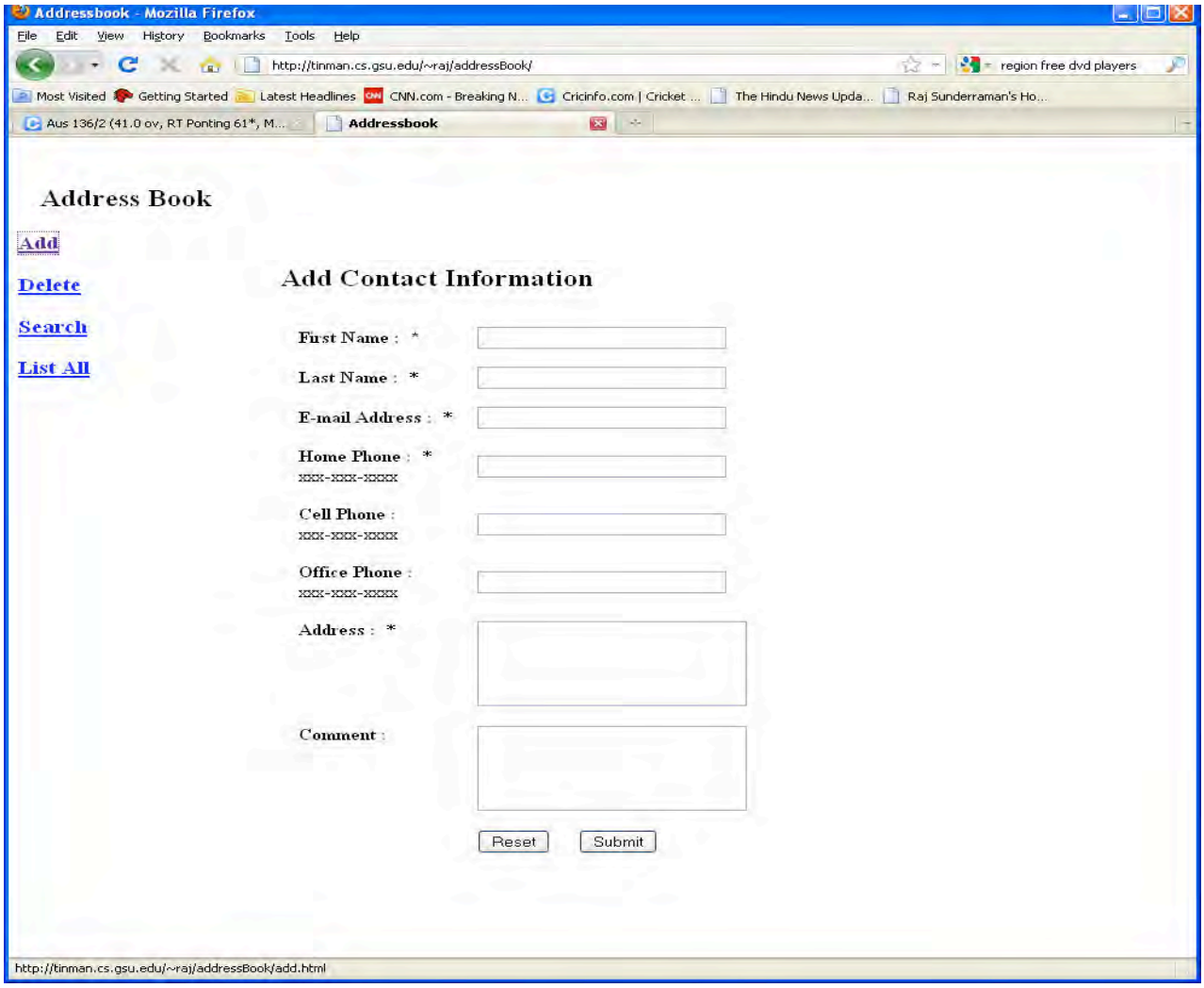

Figure 4.7: Address Book – Add Form

Mandatory fields are marked with an "*". The HTML code (`add.html`) that produces this Web page is shown below:

```
<html>
<head>
<title>Add</title>
</head>
<body>
<h2>Add Contact Information</h2>
<form name="addform" action="add.php" method=post" >
   <table width="60%" border="0" cellpadding="3" cellspacing="12">
   <tr>
      <td width="130"><strong>First Name:</strong>*</td>
      <td><input name="fname" type="text" size="30" maxlength="30" /></td>
   </tr>
   <tr>
      <td width="130"><strong>Last Name:</strong>*</td>
      <td><input name="lname" type="text" size="30" maxlength="30" /></td>
    </tr>
```

```
    <tr>
      <td width="130"><strong>E-mail Address:</strong>*</td>
      <td><input name="email" type="text" size="30" maxlength="30" /></td>
    </tr>
    <tr>
      <td width="130"><p><strong>Home Phone:</strong>*<br />xxx-xxx-xxxx</p>
      </td>
      <td><input name="homePhone" type="text" size="30" maxlength="20" /></td>
    </tr>
    <tr>
      <td width="130"><strong>Cell Phone : <br />
      </strong>xxx-xxx-xxxx</td>
      <td><input name="cellPhone" type="text" size="30" maxlength="20" /></td>
    </tr>
    <tr>
      <td width="130"><strong>Office Phone : <br />
      </strong>xxx-xxx-xxxx</td>
      <td><input name="officePhone" type="text" size="30"
                 maxlength="20" /></td>
    </tr>
    <tr valign="top">
      <td width="130"><strong>Address:</strong>*</td>
      <td><textarea name="address" cols="25" rows="4"></textarea></td>

    </tr>
    <tr valign="top">
      <td width="130"><strong>Comment:</strong> </td>
      <td><textarea name="comment" cols="25" rows="4"></textarea></td>
    </tr>
    <tr>
      <td width="130"> </td>
      <td><input type="reset" name="Reset" value="Reset" />
            
        <input type="submit" name="Submit" value="Submit" /></td>
    </tr>
  </table>
  </form>
</body>
</html>
```

The code to process the submission of the “ADD” form (`add.php`) is shown below:

```
<?php
// connect to my sql
$host="localhost"; // Host name
$username="id";         // Mysql username
$password="pwd";      // Mysql password
$db_name="db";         // Database name
$tbl_name="contacts"; // Table name

// Connect to server and select database.
mysql_connect("$host", "$username", "$password")or die("cannot connect");
mysql_select_db("$db_name")or die("cannot select DB");

// retrieve all variables
$fname = @$_POST["fname"];
```

```
$lname = @$_POST["lname"];
$email = @$_POST["email"];
$homePhone = @$_POST["homePhone"];
$cellPhone = @$_POST["cellPhone"];
$officePhone = @$_POST["officePhone"];
$address = @$_POST["address"];
$comment = @$_POST["comment"];

// insert information to database
$sql="insert into $tbl_name values".
     "('$lname','$fname','$email','$homePhone','$cellPhone', ".
     "'$officePhone','$address','$comment')";
$result = mysql_query($sql);
mysql_close();
?>

<html>
<head>
<title>Add processed</title>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
  <p>
  <h3>Your information is added to database. </h3>
  <body>
<p> </p>
<p> </p>
<p> </p>
<p> </p>
</body>
</html>
```

The program first connects to the database, then retrieves all submitted data into variables and then constructs an SQL insert statement. Finally, it executes the insert statement and prints a message.

List All

Upon clicking the "List ALL" option, the Web page shown in Figure 4.8 is shown on the right frame. This is an alphabetical listing of all contacts, each hyper-linked to go to a detail page.

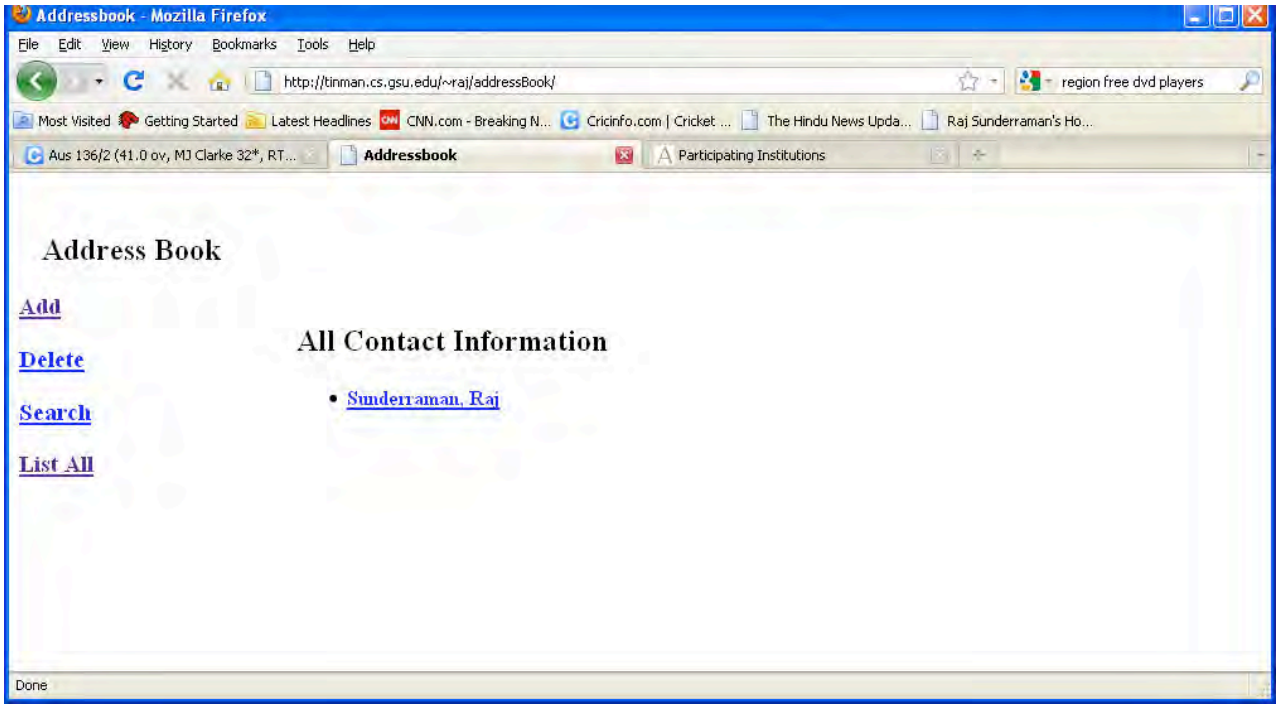

Figure 4.8: Address Book – List All

The code to display the list is shown below:

```
<?php
// connect to my sql
$host="localhost"; // Host name
$username="id";  // Mysql username
$password="pwd";  // Mysql password
$db_name="db";  // Database name
$tbl_name="contacts"; // Table name

// Connect to server and select database.
mysql_connect("$host", "$username", "$password")or die("cannot connect");
mysql_select_db("$db_name")or die("cannot select DB");

// show all contact information
$sql="select * from $tbl_name order by lname";
$result = mysql_query($sql);
mysql_close();
?>

<html>
<head>
<title>List</title>
</head>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
  <p>
  <h2>All Contact Information</h2>
```

```
   <?php
       if (mysql_num_rows($result)==0){
      echo "<h4>No data<h4>";
    } else {
       while($row = mysql_fetch_assoc($result)) {
         $lname = $row['lname'];
         $fname = $row['fname'];
         echo "<ul><li><h4><a href=\"detail.php? ".
              "lname=$lname&fname=$fname\">$lname, $fname</a><h4></li></ul>";
      }
    }
  ?>
</blockquote>
</body>
</html>
```

The program connects to the database and executes a simple query and displays the results on the Web page. Each contact is hyper-linked to the `detail.php` program.

Detail

Upon clicking the link, the detail page shown in Figure 4.9 is shown.

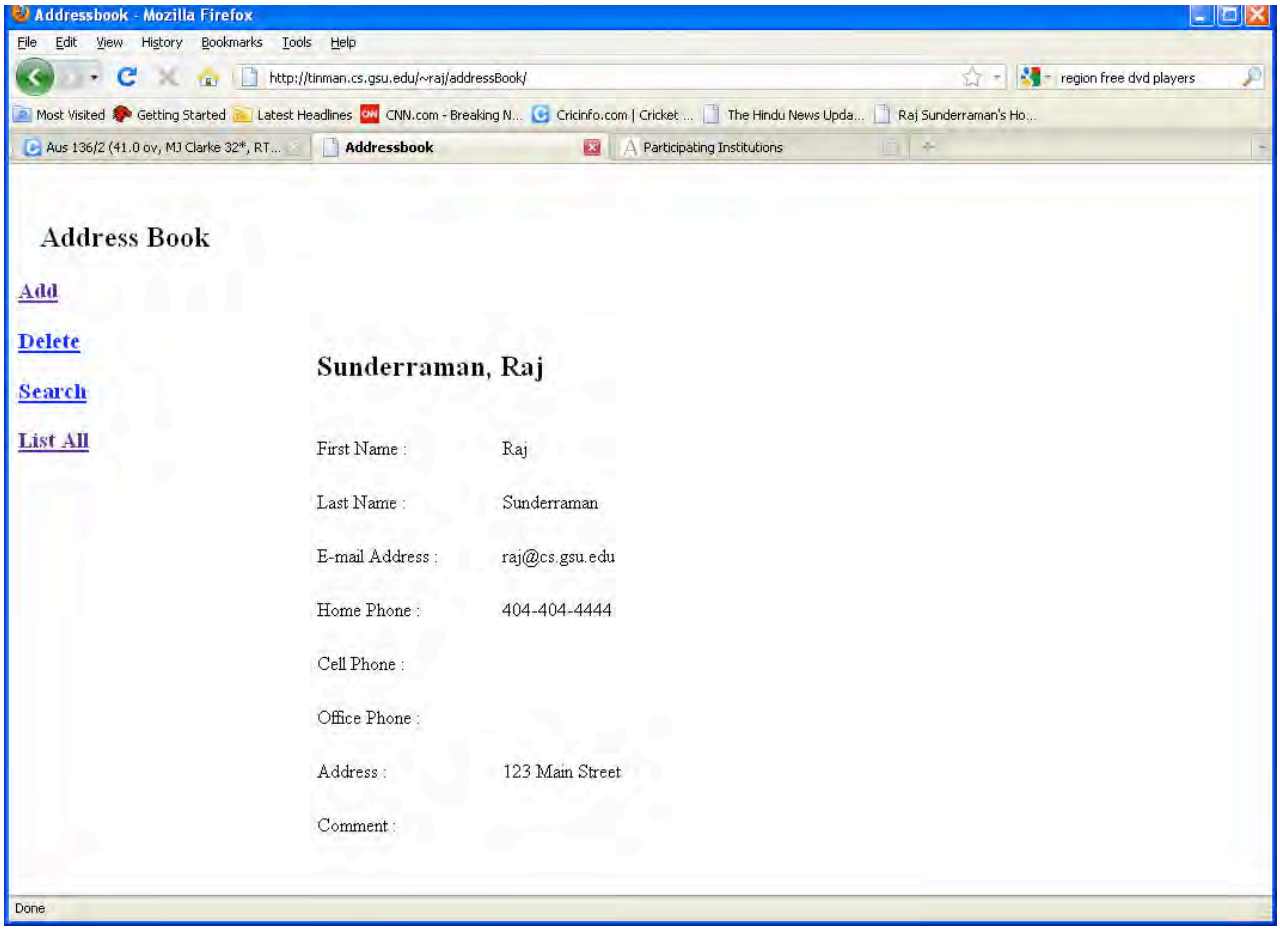

Figure 4.9: Address Book – Detail

The code that produces the detail page is shown below:

```
<?php
// connect to my sql
$host="tinman.cs.gsu.edu"; // Host name
$username="cms";        // Mysql username
$password="cms123";     // Mysql password
$db_name="conf";        // Database name
$tbl_name="contacts"; // Table name

// Connect to server and select database.
mysql_connect("$host", "$username", "$password")or die("cannot connect");
mysql_select_db("$db_name")or die("cannot select DB");

// retrieve all variables
$fname = @$_GET["fname"];
$lname = @$_GET["lname"];

// show all contact information
$sql="select * from $tbl_name where fname='$fname' and lname='$lname'";
$result = mysql_query($sql);
$row = mysql_fetch_assoc($result);
mysql_close();
?>

<html>
<head>
<title>Detail</title>
</head>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
    <table width="60%" border="0" cellpadding="5" cellspacing="15">
        <tr>
            <td colspan="2"><p><h2><?php echo "$lname, $fname"; ?></h2></td>
          </tr>
    <tr>
      <td width="130">First Name :</td>
      <td><?php echo $row['fname']; ?></td>
    </tr>
    <tr>
      <td width="130">Last Name :</td>
      <td><?php echo $row['lname']; ?></td>
    </tr>
    <tr>
      <td width="130">E-mail Address :</td>
      <td><?php echo $row['email']; ?></td>
    </tr>
    <tr>
      <td width="130"><p>Home Phone :<br />
      </p>        </td>
      <td><?php echo $row['homePhone']; ?></td>
    </tr>
```

```
    <tr>
      <td width="130">Cell Phone :</td>
      <td><?php echo $row['cellPhone']; ?></td>
    </tr>
    <tr>
      <td width="130">Office Phone :</td>
      <td><?php echo $row['officePhone']; ?></td>
    </tr>
    <tr valign="top">
      <td width="130">Address :</td>
      <td><?php echo $row['address']; ?></td>
    </tr>
    <tr valign="top">
      <td width="130">Comment :</td>
      <td><?php echo $row['comment']; ?></td>
    </tr>
  </table>
   </p>
</blockquote>
</body>
</html>
```

The above code connects to the database and performs a query to get details of the requested contact. It then displays the data for the contact in a tabular format.

Delete

The delete contact interface is shown in Figure 4.10. The list of contacts is presented to the user along with a check box to the left of each contact. The user may choose one or more contacts and submit for deletion.

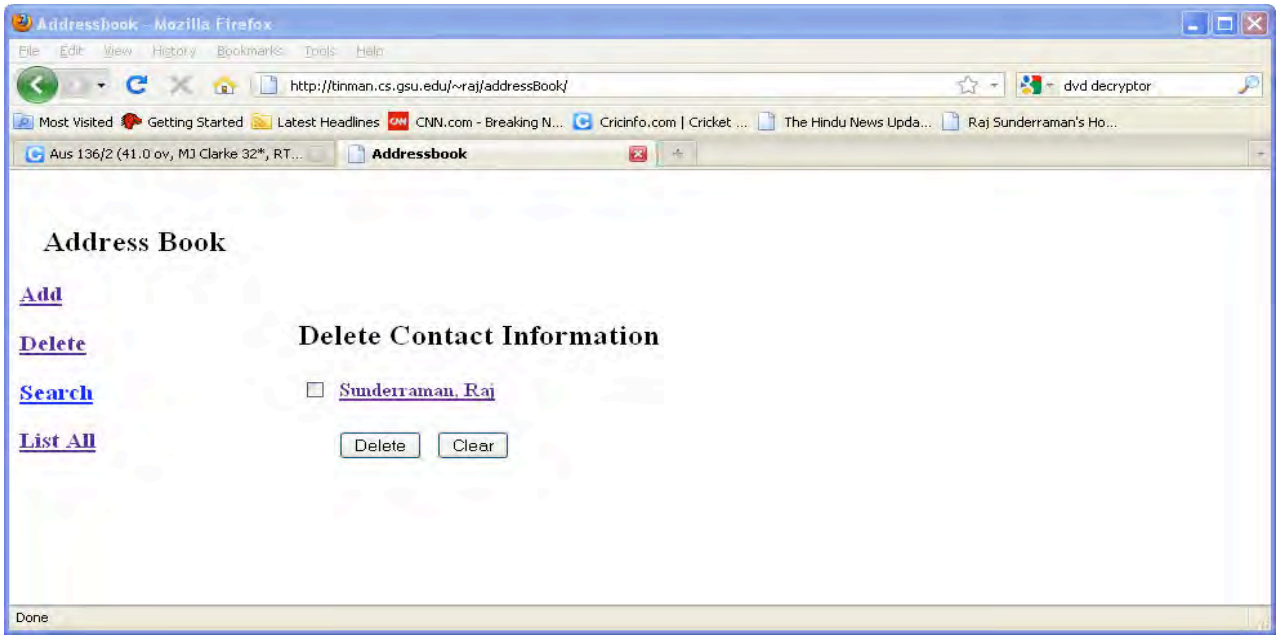

Figure 4.10: Address Book – Delete

The code to display the delete interface as well as to process the delete request (`delete.php`) is shown below:

```
<?php
// connect to my sql
$host="localhost"; // Host name
$username="id";  // Mysql username
$password="pwd";  // Mysql password
$db_name="db";  // Database name
$tbl_name="contacts"; // Table name

// Connect to server and select database.
mysql_connect("$host", "$username", "$password")or die("cannot connect");
mysql_select_db("$db_name")or die("cannot select DB");

// delete record
$delete = @$_POST["delete"];
if ($delete == "Delete") {
  $dfname = @$_POST["dfname"];
  $dlname = @$_POST["dlname"];
  $checkbox = @$_POST["checkbox"];
  // delete record
  foreach ($checkbox as $index){
    $sql = "delete from $tbl_name where fname='$dfname[$index]' ".
           "and lname='$dlname[$index]'";
    $result = mysql_query($sql);
  }
}

// show all contact information
$sql="select * from $tbl_name order by lname";
$result = mysql_query($sql);
mysql_close();
?>

<html
<head>
<title>Delete</title>
</head>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
<p>
<h2>Delete Contact Information</h2>
<form name="myform" method="post" action="delete.php">
<table width="40%" border="0">
<?php
  $index=0;
  while($row = mysql_fetch_assoc($result)) {
    $lname = $row['lname'];
    $fname = $row['fname'];
    echo "<tr><td width=\"25\" valign=\"top\">".
         "<input type=\"checkbox\" name=\"checkbox[]\" value=\"$index\" />";
    echo "</td><td valign=\"bottom\"><h4><a href=\"detail.php? ".
```

```
          "lname=$lname&fname=$fname\">$lname, $fname</a><h4></td></tr>";
    echo "<input type=\"hidden\" name=\"dfname[]\" value=\"$fname\" />";
    echo "<input type=\"hidden\" name=\"dlname[]\" value=\"$lname\" />";
    $index++;
  }

  echo "<tr><td width=\"25\" valign=\"top\"> </td>".
       "<td valign=\"bottom\"><input type=\"submit\" name=\"delete\"";
  echo "value=\"Delete\" />    ".
       "<input type=\"reset\" name=\"Submit2\" value=\"Clear\" /></td></tr>";
?>
</table>
</form>

<?php
  if (mysql_num_rows($result)==0)
    echo "<h4>No data<h4>";
?>
</p>
</blockquote>
</body>
</html>
```

In the first part of the code, the database connection is established and if the delete submission is detected, the corresponding record is deleted from the database. Then, the listing of all contacts is produced along with the check boxes within an HTML form.

Search

The search interface is shown in Figure 4.11.

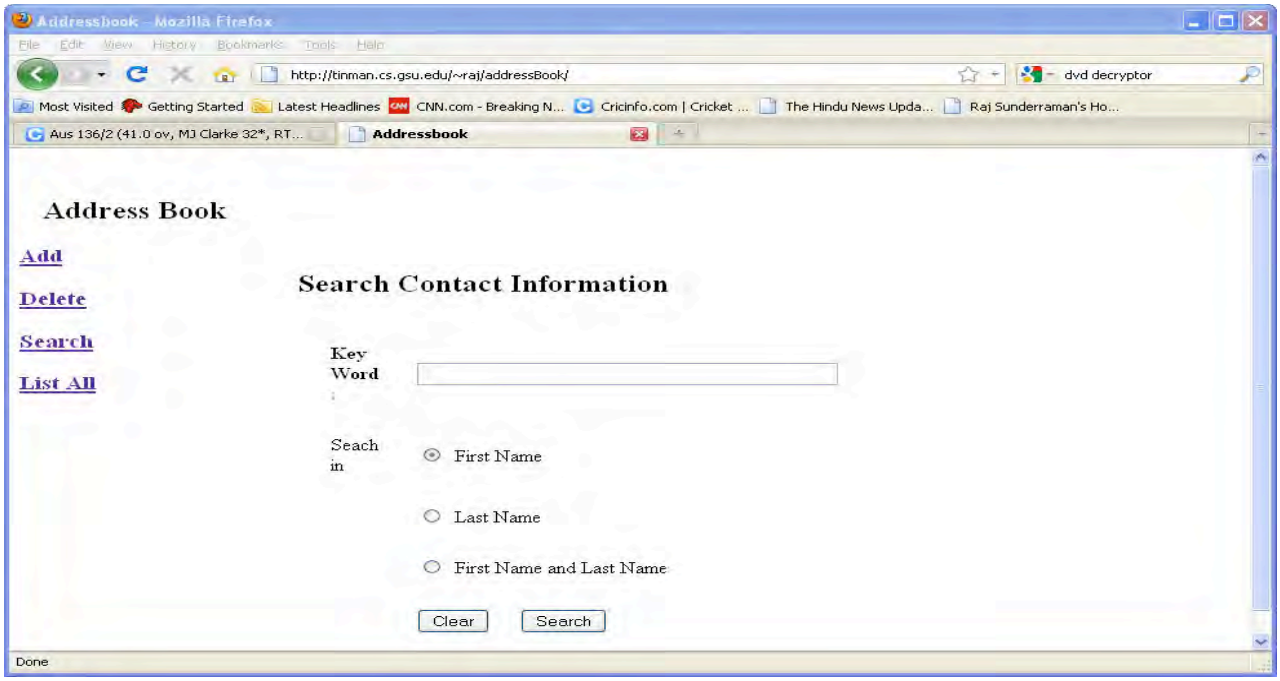

Figure 4.11: Address Book – Search

The search interface consists of a keyword text box along with options for choosing first name, last name or both for the search. The code (`search.html`) to produce the interface is shown below:

```
<html>
<head>
<title>Search HTML</title>
</head>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
<p>
<h2>Search Contact Information</h2>
<form action="result.php" method=post>
  <table width="60%" border="0" cellpadding="5" cellspacing="20">
  <tr>
    <td width="115"><strong>Key Word :  </strong></td>
    <td><input name="keyword" type="text" size="50" maxlength="50" /></td>
  </tr>
  <tr>
    <td width="115">Seach in  </td>
    <td><input name="searchin" type="radio" value="fname" checked/>
         First Name </td>
  </tr>
  <tr>
    <td width="115"> </td>
    <td><input name="searchin" type="radio" value="lname" />
         Last Name </td>
  </tr>
  <tr>
    <td width="115"> </td>
    <td><input name="searchin" type="radio" value="both" />
     First Name and Last Name </td>
  </tr>

  <tr>
    <td> </td>
    <td><input type="reset" name="Reset" value="Clear" />
            
        <input type="submit" name="Submit" value="Search" /></td>
  </tr>
  </table>
</form>
</p>
</blockquote>
</body>
</html>
```

The above is a straightforward coding of an HTML form with the relevant form elements enclosed within it. Upon submission of the search form, the following program (`result.php`) is called.

```
<?php
// connect to mysql
$host="localhost"; // Host name
```

```
$username="id";  // Mysql username
$password="pwd";  // Mysql password
$db_name="db";  // Database name
$tbl_name="contacts"; // Table name

// Connect to server and select database.
mysql_connect("$host", "$username", "$password")or die("cannot connect");
mysql_select_db("$db_name")or die("cannot select DB");

// retrieve all variables
$keyword = @$_POST["keyword"];
$searchin = @$_POST["searchin"];

// execute query
$sql="select * from $tbl_name";
$result = mysql_query($sql);
mysql_close();
?>

<html>
<head>
<title>Results</title>
</head>
<body>
<p> </p>
<p> </p>
<p> </p>
<blockquote>
<p>
<h2>Result</h2>
<?php
  $i =0;
  while($row = mysql_fetch_assoc($result)) {
    $lname = $row['lname'];
    $fname = $row['fname'];
    if ($searchin == "both"){
      // search in last name & first name
      if ((preg_match("/$keyword/i",$fname))||
          (preg_match('/$keyword/',$lname))){
        echo "<ul><li><h4><a href=\"detail.php? ".
             "lname=$lname&fname=$fname\">$lname, $fname</a><h4></li></ul>";
        $i++;
      }
    } else if($searchin == "fname"){
      // search in first name
      if (preg_match("/$keyword/i",$fname)){
        echo "<ul><li><h4><a href=\"detail.php?.
             "lname=$lname&fname=$fname\">$lname, $fname</a><h4></li></ul>";
        $i++;
      }
    } else {
      // search in last name
      if (preg_match("/$keyword/i",$lname)){
        echo "<ul><li><h4><a href=\"detail.php? ".
             "lname=$lname&fname=$fname\">$lname, $fname</a><h4></li></ul>";
        $i++;
      }
```

```
    }
  }
  if ($i == 0)
    echo "<ul><h4>No match result.<h4></ul>";
?>
</blockquote>
</body>
</html>
```

The above code connects to the database and executes a query to retrieve all contacts. Then, for each contact the appropriate filter is applied and only those contacts that pass the filter test are displayed in the Web page.

Exercises

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

1. Consider the ER-schema generated for the `CONFERENCE_REVIEW` database in Laboratory Exercise 7.34 for which the relational database was created in Laboratory Exercise 9.13.
    a. Create the database tables in MySQL.
    b. Create comma separated data files containing data for at least 10 papers, 15 authors, and 8 reviewers. You must assign authors to papers, assign a contact author, and assign reviewers to papers. It is possible for some authors to be reviewers. In such a case, the reviewer should not be assigned to a paper in which he or she is an author. You should also create data for the individual reviews for each paper. The values for various rankings must be between 1 and 10.
    c. Using the MySQL loader command, load the data created in (b) into the database.
    d. Write SQL queries for the following and execute then in a MySQL interactive session.
        i. Retrieve the email and names of authors who have submitted more than one paper for review.
        ii. Retrieve the email id of the contact author, the title of paper submitted by that author, and the average rating received on its reviews such that the average rating was above 8.0.
        iii. Retrieve the paper id, title, and average ratings received for all papers sorted in decreasing order of average rating. Do not show any papers below an average rating of 3.0.
        iv. Retrieve the email and names of reviewers along with the average ratings they gave to papers assigned to them. Sort the result in increasing order of average rating.
        v. Retrieve the paper id and title of papers along with the author name and email id such that the author is also a reviewer.

2. Consider the SQL schema generated for the `GRADE_BOOK` database in Laboratory Exercise 8.28 for which the relational database was created in Laboratory Exercise 9.13.

a. Create the database tables in MySQL.
b. Create comma separated data files containing data for at least 20 students and 3 courses with an average of 8 students in each course. Create data for 3 to 4 grading components for each course. Then, create data to assign points to each student in every course they are enrolled in for each of the grading components.
c. Using the MySQL loader command, load the data created in (b) into the database.
d. Write SQL queries for the following and execute them in a MySQL interactive session.
    i. Retrieve the course numbers, titles, and term of courses in which student with id=1234 has enrolled.
    ii. Retrieve the term, section number, and course numbers along with their total enrollments.
    iii. Retrieve the term, section number, and course numbers of courses in which there is a grading component that has a weight of more than 50 and has fewer than 10 students.
    iv. Retrieve the names of students who have enrolled in all courses taught by the instructor in the Fall 2005 term.

3. Consider the SQL schema generated for the `ONLINE_AUCTION` database in Laboratory Exercise 8.29 for which the relational database was created and populated in Laboratory Exercise 9.14.
    a. Create the database tables in MySQL.
    b. Create comma separated data files containing data for at least 20 items with at least 10 buyers and at least 10 sellers. Observe the queries below and make sure that your data will be appropriate for those queries.
    c. Using the MySQL loader command, load the data created in (b) into the database.
    d. Write SQL queries for the following and execute then in a MySQL interactive session.
        i. Retrieve item numbers and descriptions of items that are still active (i.e. bidding has not closed) along with their current bidding price.
        ii. Retrieve the item numbers and descriptions of all items in which the member with member id `abc24` is the winner.
        iii. Retrieve the member ids and names of members who have an average rating of 8.0 or above.
        iv. Retrieve the bidding history for the item with item number `i246`. The bidding history should contain a chronological listing of all bids till the current time including the member id of the bidder and the bidding price.
        v. Retrieve the item number and titles of items that fall under the `/COMPUTER/HARDWARE/PRINTER` category and have "`HP`" in their titles and on which bidding has not yet closed sorted by time of bid closing.

4. Consider the `CONFERENCE_REVIEW` database of Laboratory Exercise 7.34 for which the relational database was created and populated in Laboratory Exercise 9.12. There are three types of users of the system: *contact author*, *reviewer*, and *administrator*. The contact author should be able to sign in and submit a paper to the conference; the reviewer should be able to submit reviews of papers assigned to him or her; and the administrator should be

able to assign reviewers to papers (at least three reviewers to each paper) as well as print a report of all papers sorted by average rating assigned by reviewers. The reviewers and administrator have pre-assigned user ids and passwords, however, contact authors must register themselves first before they can start using the system. Write a PhP-based application that implements the following functions:

a. A user registration page that allows a contact author to register their email and other information with the system. They choose a password in the registration page.
b. A user login page with GUI textbox elements to accept user id and password and radio buttons to choose between contact author, reviewer, and administrator. A separate menu for each user should be displayed after authentication.
c. The contact author should be able to enter details of his paper submission and upload a file.
d. The reviewer should be able to choose one of his assigned papers and submit a review for the paper.
e. The administrator should be able to assign at least three reviewers to each paper. The administrator should also be able to display a report of all papers sorted by average rating (high to low).

5. Consider the `GRADE_BOOK` database of Laboratory Exercise 8.28 for which the relational database was created and populated in Laboratory Exercise 9.13. Write and test PhP scripts that access the MySQL database that implements the following two functions for a given course section (the course number, term, and section information will be provided as GET parameters in the URL itself):
    i. Allows the instructor to enter the points/scores for a particular grading component. The instructor should be able to choose a grading component using a pull down list.
    ii. Allows the instructor to view the scores for all components for all students in a tabular form with one row per student and one column per grading component.

6. Consider the `ONLINE_AUCTION` database of Laboratory Exercise 8.29 for which the relational database was created and populated in Laboratory Exercise 9.14 in MySQL. Write PhP scripts to implement an online auction Web site. A member may register himself/herself and choose his/her password. After that, they may sign into the system. The following functions should be implemented:
    a. A buyer should be able to search items by entering a keyword. The result of the search should be a listing of items, each with a hyper-link. Upon clicking on the hyper-link, a detailed page describing the item should be displayed. On this page, the user may place a bid on the item.
    b. A seller should be able to place an item for sale by providing all details of the item along with auction deadlines etc.
    c. Buyers and sellers should be able to look at a list of all items on which bidding has ended.
    d. Buyers and sellers should be able to place a rating as well as view ratings on transactions they are involved in.

CHAPTER 5

Database Design (DBD) Toolkit

This chapter introduces a toolkit written in SWI-Prolog (a freely available Prolog system for the Unix as well as the Windows operating systems) that allows the student to learn and test out the various concepts, definitions, and algorithms associated with relational database design using functional dependency theory.

The DBD system is available as SWI-Prolog source code in a single file, `dbd.pl`. This file needs to be consulted (read into the Prolog interpreter's memory) before invoking any of the DBD predicates.

5.1 Coding Relational Schemas and Functional Dependencies

The DBD system uses lists to code the relational schemas and functional dependencies. Variables in Prolog begin with an upper-case letter and hence attributes of relational schemes are coded as Prolog constants – terms that begin with a lower-case letter. Consider the following relational schema and associated set of functional dependencies:

```
R(A,B,C,D,E)
F = { A → B, AC → D, E → AB, AD → E, ABC → D }
```

These will be coded as the following Prolog predicates in DBD:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
```

5.2 Invoking the SWI-Prolog Interpreter

Once the SWI-Prolog system is installed in your server or local computer, the interpreter is invoked using the following terminal command:

$ **pl**

Here `$` is the command prompt. In the rest of the chapter, all user inputs are shown in **bold** font and system displays in regular font.

After printing a welcome message, the interpreter responds with the following prompt:

```
?-
```

At this prompt, the user may enter a Prolog query to invoke a predicate. In order to invoke one of the predicates of the DBD system, the user must first load Prolog interpreter's memory with the DBD program using the following command/query:

```
?- ['dbd.pl'].
```

Note that all commands/queries to Prolog end with a period (".") symbol. To exit the Prolog system, the halt command/query is entered at the Prolog prompt as follows:

```
?- halt.
```

To solve a particular problem using the DBD system, the student will usually create a file containing the code to represent the relational schema and the functional dependencies. In addition, one or more Prolog rules may be included to execute predicates to solve the problem at hand. For example, consider the problem of finding one or more candidate keys for the relational schema and functional dependencies mentioned earlier in the section. The student will create a text file with the following code:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
answer(K) :- schema(R), fds(F), candkey(R,F,K).
```

The first two predicates (`schema` and `fds`) code the relational schema and functional dependencies. The third rule invokes the `candkey` predicate of the DBD system to find the candidate keys for `R` and `F`. The answer is captured in the `answer` predicate with one argument. Let us assume that the above code is present in a file called `example.pl`. The following interaction with the SWI-Prolog system shows the steps necessary to find the answer(s) to the problem:

```
$ pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.7)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['example.pl'].
% example.pl compiled 0.00 sec, 1,116 bytes

Yes
?- answer(K).

K = [c, e] ;
```

```
K = [a, c] ;

No
?- answer(K).

K = [c, e]

Yes
?- halt.
$
```

The user inputs are shown in **bold** font. After invoking the interpreter, the user first loads the DBD system code (`dbd.pl`). The user then loads the code to solve the problem at hand (`example.pl`). Finally, the user submits the query, **answer(K)**, to determine the candidate keys. By default, every Prolog system responds with the first answer to the query and waits for user input. At this point the user has two choices: press the `ENTER` key to stop looking at further answers to the query or press the semi-colon key to request Prolog to display the next answer to the query. In the above screen capture, both options are shown.

5.3 DBD System Predicates

The DBD system predicates are discussed in this section. Almost all of the predicates take as input the relation schema R and a set of functional dependencies F. R and F must be coded properly as indicated in earlier sections. In particular, the attribute names must begin with a lower-case letter. The predicate names themselves also must begin with a lower-case letter.

5.3.1 xplus(R,F,X,Xplus)

The `xplus` predicate takes as input a relation schema `R`, a set of functional dependencies `F`, and a subset `X` of `R` and produces as output the attribute-closure of `X` under `F`, i.e. the set of attributes that are functionally dependent on `X`. The following shows a sample interaction with the Prolog system to compute the attribute closure of `{A, C}` for the relational schema and functional dependencies mentioned earlier in this chapter (the file `e2.pl` is assumed to contain the `schema` and `fds` predicates):

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e2.pl'].
% e2.pl compiled 0.00 sec, 1,096 bytes

Yes
?- schema(R),fds(F),xplus(R,F,[a,c],Xplus).

R = [a, b, c, d, e]
F = [[[a], [b]], [[a, c], [d]], [[e], [a, b]], [[a, d], [e]], [[a, b, c], [d]]]
Xplus = [a, b, c, d, e]
```

```
Yes
?- halt.
$
```

5.3.2 finfplus(R,F,[X,Y])

The `finfplus` predicate takes as input a relation schema `R`, a set of functional dependencies `F`, and a functional dependency `X → Y` (in list coded form `[X,Y]`) and returns "yes" if `X → Y` is in the closure of `F` and "no" otherwise. An example using the same `R` and `F` as before follows:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e2.pl'].
% e2.pl compiled 0.00 sec, 1,096 bytes

Yes
 ?- schema(R), fds(F), finfplus(R,F,[[a,b],[c]]).

No
?- schema(R), fds(F), finfplus(R,F,[[a,d],[b]]).

R = [a, b, c, d, e]
F = [[[a], [b]], [[a, c], [d]], [[e], [a, b]], [[a, d], [e]], [[a, b, c], [d]]]

Yes
```

In the above screen capture we observe that the functional dependency `AB → C` is not in the closure of `F` whereas the functional dependency `AD → B` is in the closure of `F`.

5.3.3 fplus(R,F,Fplus)

The `fplus` predicate takes as input a relational schema `R` and a set of functional dependencies `F` and returns as output the closure of `F` in the third parameter. Typically the number of functional dependencies in the closure of `F` is large and in the worst case can be exponential in the number of attributes of `R`.

Consider the following Prolog code in file `e3.pl`:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
displayFDs([]) :- nl.
displayFDs([[X,Y]|Rest]) :-
  write(X),write('-->'), write(Y),nl,displayFDs(Rest).
go :- schema(R), fds(F), fplus(R,F,Fplus), displayFDs(Fplus).
```

This program adds the display rules to the `e2.pl` file and invokes the `displayFDs` predicate on the closure of `F`. All functional dependencies that are implied by `F` are displayed by the following invocation:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e3.pl'].
% e3.pl compiled 0.01 sec, 1,700 bytes

Yes
?- go.
[a]-->[a, b]
[a]-->[b]
...
...
```

Note: The `fplus` predicate does not generate any trivial functional dependencies (dependencies whose RHS attributes are also present in the LHS attributes).

5.3.4 implies(R,F1,F2) and equiv(R,F1,F2)

The `implies` predicate takes as input a relational schema `R` and two sets of functional dependencies `F1` and `F2`. It returns "yes" if `F1` implies `F2`, i.e. if each functional dependency in `F2` is in the closure of `F1`; returns "no" otherwise.

The `equiv` predicate takes as input a relational schema `R` and two sets of functional dependencies `F1` and `F2`. It returns "yes" if `F1` is equivalent to `F2`, i.e. if each functional dependency in `F2` is in the closure of `F1` and if each functional dependency in `F1` is in the closure of `F2`; returns "no" otherwise.

Consider the relational schema `R(A,B,C)` and the two sets of functional dependencies:

```
F1 = {A →BC, B →A}
F2 = {A →B, A →C}
```

The above schema and functional dependencies can be coded into a file `e4.pl` as follows:

```
schema([a,b,c]).
fds1([[[a],[b,c]], [[b],[a]]]).
fds2([[[a],[b]], [[a],[c]]]).
```

The following screen capture illustrates the `implies` and `equiv` predicates:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes
```

```
Yes
?- ['e4.pl'].
% e4.pl compiled 0.00 sec, 1,120 bytes

Yes
?- schema(R), fds1(F1), fds2(F2), implies(R,F1,F2).

R = [a, b, c]
F1 = [[[a], [b, c]], [[b], [a]]]
F2 = [[[a], [b]], [[a], [c]]]

Yes
?- schema(R), fds1(F1), fds2(F2), implies(R,F2,F1).

No
?- schema(R), fds1(F1), fds2(F2), equiv(R,F1,F2).

No
?-
```

5.3.5 superkey(R,F,K) and candkey(R,F,K)

The `superkey` and `candkey` predicates both take a relation schema `R` and a set of functional dependencies as input in the first two parameters. The third parameter may be a constant (i.e. an instantiated list of attributes of `R`) or a variable.

In case the third parameter is instantiated to a list of attributes the `superkey` (`candkey`) predicate with return "yes" if the instantiated third parameter is a super key (candidate key) of R; "no" otherwise.

In case the third parameter is a variable, the predicate `superkey` (`candkey`) will generate all super keys (candidate keys) one at a time.

Consider the following `e5.pl` file containing a relation schema and associated functional dependencies:

```
schema([a,b,c]).
fds([[[a],[b,c]], [[b],[a]]]).
```

The following screen capture illustrates usage of the `superkey` and `candkey` predicates:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.01 sec, 30,428 bytes

Yes
?- ['e5.pl'].
% e5.pl compiled 0.00 sec, 1,124 bytes

Yes

?- schema(R), fds(F), candkey(R,F,K).
```

```
R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a] ;

No
?- schema(R), fds(F), superkey(R,F,K).

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, b, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, b] ;

No
?- schema(R), fds(F), candkey(R,F,[a,b]).

No
?- schema(R), fds(F), superkey(R,F,[a,b]).

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]

Yes
?-
```

5.3.6 mincover(R,F,FC)

The `mincover` predicate takes as input a relation schema `R` and a set of functional dependencies `F` and returns as output in the third parameter the minimal cover for `F`.

Consider the following relation schema and functional dependencies in file `e6.pl`:

```
schema([a,b,c,d,e]).
fds([[[a],[b,c]], [[b],[a]], [[a,b],[d]], [[a],[d]], [[b,c],[a]]]).
```

The functional dependencies coded in the file are:

F = {A → BC, B → A, AB → D, A → D, BC → A}

The following screen capture illustrates the `mincover` predicate usage:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e6.pl'].
% e6.pl compiled 0.00 sec, 1,072 bytes

Yes
?- schema(R),fds(F),mincover(R,F,MinF).

R = [a, b, c, d, e]
F = [[[a], [b, c]], [[b], [a]], [[a, b], [d]], [[a], [d]], [[b, c], [a]]]
MinF = [[[a], [b, c, d]], [[b], [a]]]

Yes
?-
```

The minimum cover for the set of functional dependencies is

MinF = {A → BCD, B → A}.

5.3.7 ljd(R,F,R1,R2), ljd(R,F,D), and fpd(R,F,D)

The predicates discussed in this sub-section are related to decompositions of relation schemas. We represent decompositions using lists of relational schemas. For example, the decomposition D = {ABC, ABD, BDE} of R = ABCDE is represented as the following Prolog term:

```
D = [[a,b,c],[a,b,d],[b,d,e]]
```

Testing for lossless join decomposition

There are two versions of the `ljd` predicate.

The first version of the `ljd` predicate takes as input a relation scheme `R`, a set of functional dependencies `F`, and two relation schemas `R1` and `R2` that are decompositions of `R`. The predicate returns "yes" if `(R1, R2)` is a lossless join decomposition of `R`; "no" otherwise.

As an example, consider the R = ABC and F = {A → B}. Consider the decomposition of R into two sub-schemas R1 = AB and R2 = AC. The following screen capture illustrates the use of the first version of ljd predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ljd([a,b,c],[[[a],[b]]],[a,b],[a,c]).

Yes
?-
```

The second version of the ljd predicate takes as input a relation scheme R, a set of functional dependencies F, and a decomposition D into two or more sub-schemas. D is represented as a list of relation schemas where each schema is itself a list of attributes. The predicate returns "yes" if D is a lossless join decomposition of R; "no" otherwise. This version implements the "matrix" method (Algorithm 11.1 of the Elmasri/Navathe text) to test for lossless join and prints the final matrix.

As an example, consider the following file (named e7a.pl) containing a relational schema, a set of functional dependencies and a decomposition into more than two sub-schemas:

```
schema([a,b,c,d,e]).
fds([[[a],[c]], [[b],[c]], [[c],[d]], [[d,e],[c]], [[c,e],[a]]]).
decomp([[a,d],[a,b],[b,e],[c,d,e],[a,e]]).
```

The following screen capture illustrates the usage of the more general version of the ljd predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e7a.pl'].
% e7a.pl compiled 0.01 sec, 1,440 bytes

Yes
?- schema(R), fds(F), decomp(D), ljd(R,F,D).
[[a, 1], [b, 1, 2], [a, 3], [a, 4], [b, 1, 5]]
[[a, 1], [a, 2], [a, 3], [a, 4], [b, 2, 5]]
[[a, 1], [a, 2], [a, 3], [a, 4], [a, 5]]
[[a, 1], [b, 4, 2], [a, 3], [a, 4], [a, 5]]
[[a, 1], [b, 5, 2], [a, 3], [a, 4], [a, 5]]


R = [a, b, c, d, e]
F = [[[a], [c]], [[b], [c]], [[c], [d]], [[d, e], [c]], [[c, e], [a]]]
D = [[a, d], [a, b], [b, e], [c, d, e], [a, e]]

Yes
?-
```

Note that the `ljd` predicate execution prints the final matrix to the screen; In this case it can be observed that the third row is turned into an "all-a" row.

Testing for FD-preserving decomposition

The `fpd` predicate takes as input a relation scheme `R`, a set of functional dependencies `F`, and a decomposition `D` into two or more sub-schemas. `D` is represented as a list of relation schemas where each schema is itself a list of attributes. The predicate returns "yes" if `D` is a functional dependency preserving decomposition of `R`; "no" otherwise.

As an example, consider the following file (named `e7b.pl`) containing a relational schema, a set of functional dependencies and a decomposition into sub-schemas:

```
schema([a,b,c,d]).
fds([[[a],[b]], [[b],[c]], [[c],[d]], [[d],[a]]]).
decomp([[a,b],[b,c],[c,d]]).
```

The following screen capture illustrates the usage of the `fpd` predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.03 sec, 30,740 bytes

Yes
?- ['e7b.pl'].
% e7b.pl compiled 0.00 sec, 1,260 bytes

Yes
?- schema(R), fds(F), decomp(D), fpd(R,F,D).
Considering [a]-->[b]
Xplus=[a, b, c, d]
Considering [b]-->[c]
Xplus=[a, b, c, d]
Considering [c]-->[d]
Xplus=[a, b, c, d]
Considering [d]-->[a]
Xplus=[a, b, c, d]

R = [a, b, c, d]
F = [[[a], [b]], [[b], [c]], [[c], [d]], [[d], [a]]]
D = [[a, b], [b, c], [c, d]]

Yes
?-
```

Note that the `fpd` predicate execution prints each functional dependency in `F` and displays the attribute closure of the LHS of the functional dependency with respect to the "projected" functional dependencies. The algorithm implemented in this predicate is from "*Jeffrey D. Ullman, Principles of Database Systems, Second Edition, Computer Science Press, 1982*".

5.3.8 is3NF(R,F) and threenf(R,F,D)

The `is3NF` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns "yes" if the relation schema is in 3NF; "no" otherwise.

The `threenf` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns in the third parameter a decomposition of `R` into 3NF. The `threenf` predicate implements Algorithm 11.4 of the Elmasri/Navathe text.

As an example, consider the following file (named `e8.pl`) containing a relational schema and a set of functional dependencies:

```
schema([a,b,c,d]).
fds([[[a,b],[c]],[[b],[d]],[[b,c],[a]]]).
```

The following screen capture illustrates the 3NF test as well as a 3NF decomposition of the relation schema.

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e8.pl'].
% e8a.pl compiled 0.01 sec, 932 bytes

Yes
?- schema(R), fds(F), is3NF(R,F).

No
?- schema(R), fds(F), threenf(R,F,R3NF).

R = [a, b, c, d]
F = [[[a, b], [c]], [[b], [d]], [[b, c], [a]]]
R3NF = [[a, b, c], [b, d]]

Yes
?-
```

As can be seen in the above interaction with the DBD system, the schema `R = ABCD` is not in 3NF and is decomposed into 3NF sub-schemas `ABC` and `BD`.

5.3.9 isBCNF(R,F) and bcnf(R,F,D)

The `isBCNF` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns "yes" if the relation schema is in BCNF; "no" otherwise.

The `bcnf` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns in the third parameter a decomposition of `R` into BCNF. The `bcnf` predicate implements Algorithm 11.3 of the Elmasri/Navathe text.

As an example, consider the same `e8.pl` file used in the previous sub-section.

The following screen capture illustrates the BCNF test as well as a BCNF decomposition of the relation schema.

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e8.pl'].
% e8.pl compiled 0.00 sec, 928 bytes

Yes
?- schema(R), fds(F), isBCNF(R,F).

No
?- schema(R), fds(F), bcnf(R,F,D).
Scheme to decompose = [a, b, c, d] Offending FD: [b]-->[d]
Final Result is:
[a, b, c]
[b, d]

R = [a, b, c, d]
F = [[[a, b], [c]], [[b], [d]], [[b, c], [a]]]
D = [[a, b, c], [b, d]];
```

As can be seen in the above interaction with the DBD system, the schema `R = ABCD` is not in BCNF and is decomposed into BCNF sub-schemas `ABC` and `BD`. Note that the `bcnf` predicate implementation also prints out the schema that is being decomposed along with the "offending" functional dependency.

Exercises

For all the problems in this set of exercises, use the predicates in the DBD system to answer the questions.

Chapter 10 (Elmasri/Navathe Text) Problems:

1. Consider the following two sets of functional dependencies:

   $F = \{A \rightarrow C,\ AC \rightarrow D,\ E \rightarrow AD,\ E \rightarrow H\}$
   $G = \{A \rightarrow CD,\ E \rightarrow AH\}$

   Check whether they are equivalent.

2. Consider the relation schema

   `EMP_DEPT(ename,ssn,bdate,address,dnumber,dname,dmgrssn)`

and the following set G of functional dependencies on EMP_DEPT:

```
ssn → ename, bdate, address, dnumber
dnumber → dname, dmgrssn
```

Calculate the closures {ssn}$^+$ and {dnumber}$^+$.

3. Is the set of functional dependencies G in Exercise 2 minimal? If not, find a minimal set of functional dependencies that is equivalent to G.

4. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

   ```
   F = {AB → C, A → DE, B → F, F → GH, D → IJ}
   ```

   What is the key for R? Decompose R into 3NF relations.

5. Repeat Exercise 4 for the same R but a different set of functional dependencies

   ```
   G = {AB → C, BD → EF, B → F, F → GH, D → IJ}
   ```

6. Consider a relation R(A,B,C,D,E) with the following functional dependencies:

   ```
   F = {AB → C, CD → E, DE → B}
   ```

   Is AB a candidate key of this relation? If not, is ABD?

7. Consider the relation R, which has attributes that hold schedules of courses and sections at a university;

   ```
   R = {courseno,secno,offeringdept,credithours,courselevel,
        instructorssn,semester,year, Dayshours,roomno,
        noofstudents}
   ```

   Suppose the following functional dependencies hold on R:

   ```
   courseno → offeringdept,credithours,courselevel

   courseno,secno,semester,year →
             dayshours,roomno,noofstudents,instructorssn

   roomno,dayshours,semester,year →
             instructorssn,courseno,secno
   ```

   Find the candidate keys for R and decompose the relation into 3NF relations.

Chapter 11 (Elmasri/Navathe Text) Problems:

8. Consider the relation schema

   LOTS(propertyid,countyname,lotno,area,price,taxrate)

   and associated set of functional dependencies:

   propertyid → countyname,lotno,area,price,taxrate
   countyname,lotno → propertyid,area,price,taxrate
   countyname → taxrate
   area → price

   Determine if the decomposition of LOTS into

   LOTS1AX(propertyis,area,lotno)
   LOTS1AY(area,countyname)
   LOTS1B(area,price)
   LOTS2(countyname,taxrate)

   is a lossless join decomposition?

9. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

   F = {AB → C, A → DE, B → F, F → GH, D → IJ}

   Determine the candidate keys for R, find a minimal cover for F, and decompose R into 3NF relations.

10. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

    G = {AB → C, BD → EF, B → F, F → GH, D → IJ}

    Determine the candidate keys for R, find a minimal cover for F, and decompose R into 3NF relations.

11. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

    F = {AB → C, A → DE, B → F, F → GH, D → IJ}

    Decompose R into BCNF relations. Repeat the exercise for

    G = {AB → C, BD → EF, B → F, F → GH, D → IJ}

12. Consider a relation R(A,B,C,D,E) with the following functional dependencies:

    F = {AB → C, CD → E, DE → B}

    Determine the candidate keys for R and decompose R into 3NF relations and into BCNF relations.

    Repeat the exercise for

    R = {courseno,secno,offeringdept,credithours,courselevel,
         instructorssn,semester,year, Dayshours,roomno,
         noofstudents}

    and F containing the following functional dependencies:

    courseno → offeringdept,credithours,courselevel,

    courseno,secno,semester,year →
              dayshours,roomno,noofstudents,instructorssn

    roomno,dayshours,semester,year →
              instructorssn,courseno,secno

13. Consider the following decompositions for the relation schema

        R(a,b,c,d,e,f,g,h,i)

    and the set of associated functional dependencies:

        F = {ab → c, a → de, b → f, f → gh, d → ij}.

    Determine whether each of the following decompositions has the (a) dependency preservation property, and (b) the lossless join property with respect to F:

i. D1 = {R1,R2,R3,R4,R5};
   R1 = {a,b,c}, R2 = {a,d,e}, R3 = {b,f}, R4 = {f,g,h},
   R5 = {d,i,j}.

ii. D2 = {R1,R2,R3};
    R1 = {a,b,c,d,e}, R2 = {b,f,g,h}, R3 = d,i,j}

iii. D3 = {R1,R2,R3,R4,R5};
     R1 = {a,b,c,d}, R2 = {d,e}, R3 = {b,f}, R4 = {f,g,h},
     R5 = {d,i,j}.

14. Consider the relation REFRIG(modelno,year,price,manufplant,color) which is abbreviated as REFRIG(m,y,p,mp,c) and the set of functional dependencies

    F = {m → mp, {m,y} → p, mp → c}

iv. Evaluate each of the following as candidate keys: {m}, {m,y}, {m,c}.
v. Test if REFRIG is in 3NF? In BCNF?
vi. Test if the decomposition D = {R1(m,y,p), R2(m,mp,c)} is lossless.

CHAPTER 6

Object-Oriented Database Management Systems: db4o

This chapter introduces the student to db4o, a very popular open-source object-oriented database management. db4o provides programming APIs in Java as well as several .Net languages, but this chapter focuses only on the Java API.

The `COMPANY` database of the Elmasri-Navathe text is used throughout this chapter. In Section 6.1, db4o installation as well as an overview of the API packages is introduced. Section 6.2 presents an elementary example of creating and querying a database. Section 6.3 presents database updates and deletes. The company database is defined in Section 6.4. Database querying is covered in Section 6.5. A complete Java application that creates and queries the company database is introduced in Section 6.6. Data about objects is read from text files and various persistent objects are created and queried. A Web application to access the company database is illustrated in Section 6.7.

6.1 db4o Installation and Getting Started

The db4o object database system can be obtained from www.db4o.com. The installation of the system is quite straightforward. All it requires is the placement of `db4o.jar` in the CLASSPATH of your favorite Java installation.

The main packages of the Java API are: `com.db4o` and `com.db4o.query`. The important classes/interfaces in the `com.db4o` package are:

1. `com.db4o.Db4o`: the factory class that provides methods to configure the database environment and to open a database file.
2. `com.db4o.ObjectContainer`: the database interface that provides methods to store, query and delete database objects as well as commit and rollback database transactions.

The `com.db4o.query` package contains the Predicate class that allows for "Native Queries" to be executed against the database. Native queries provide the ability to run one or more lines of code to execute against all instances of a class and return those that satisfy a predicate.

A typical sequence of statements to work with the database is shown below:

```
Configuration config = Db4o.configure();
ObjectContainer db = Db4o.openFile(config,"student.db4o");
…
…
db.close();
```

Initially, a configuration object is created. Various database configuration settings can be made. In this example, none of the settings are shown, however, in later examples some of the important

settings will be illustrated. The configuration object along with a database file name is provided to the `openFile()` method of the factory class. If the database file does not exist, a new database is created. Otherwise, the database present in the file is opened. A database transaction also begins at this point. The `ObjectContainer` object, `db`, is then used to perform database operations such as inserting new objects, updating or deleting existing objects, and querying the database. Transactions can also be committed or rolled back. At the end, the database is closed.

6.2 A Simple Example

In this section, a simple example is introduced in which a database of student objects is created and retrieved. Consider the following Student class:

```
public class Student {
  int sid;
  String lname;
  String fname;
  float gpa;

  public Student(int sid, String lname, String fname, float gpa) {
    this.sid = sid;
    this.lname = lname;
    this.fname = fname;
    this.gpa = gpa;
  }

  public int getSid() {
    return sid;
  }

  public void setSid(int sid) {
    this.sid = sid;
  }

  public String getLname() {
    return lname;
  }

  public void setLname(String lname) {
    this.lname = lname;
  }

  public String getFname() {
    return fname;
  }

  public void setFname(String fname) {
    this.fname = fname;
  }

  public float getGpa() {
```

```
    return gpa;
  }

  public void setGpa(float gpa) {
    this.gpa = gpa;
  }

  public String toString() {
    return sid+" "+fname+" "+lname;
  }
}
```

The class defines a student object with four simple attributes: `sid`, `lname`, `fname`, and `gpa`. There are the usual getter and setter methods along with a standard constructor and a `toString()` method. The following is a sample Java code that creates a few student objects and stores them in the database:

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.config.Configuration;

public class CreateStudent {
  public static void main(String[] args) {
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,"student.db4o");
    createFewStudents(db);
    db.close();
  }

  public static void createFewStudents(ObjectContainer db) {
    //Create few student objects and store them in the database
    Student s1 = new Student(1000,"Smith","Josh", (float) 3.00);
    Student s2 = new Student(1001,"Harvey", "Derek", (float) 4.00);
    Student s3 = new Student(1002,"Lemon", "Don", (float) 3.50);
    db.store(s1);
    db.store(s2);
    db.store(s3);
  }
}
```

As can be seen in the above code, the program opens a database file and calls the `createFewStudents()` method which creates three student objects and stores each one of them into the database using the `store()` method. At the end, the main method closes the database.

The following Java code prints the contents of the database that was just created:

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
```

```
import com.db4o.config.Configuration;

public class PrintStudents {
  public static void main(String[] args) {
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,"student.db4o");
    printStudents(db);
    db.close();
  }

  public static void printStudents(ObjectContainer db) {
    ObjectSet result = db.queryByExample(Student.class);
    System.out.println("Number of students: " + result.size()+"\n");
    while (result.hasNext()) {
      Student s = (Student) result.next();
      System.out.println(s);
    }
  }
}
```

The above program opens the database and calls `printStudents()` method to print all student objects in the database. The program illustrates one of the many ways to query the database: the query by example method. In this method, the `queryByExample()` method is called on the `ObjectContainer db`. By providing `Student.class` as input the method returns all student objects. db4o also provides Java 5 generics shortcut using the `queryByExample()` method; so the `printStudents()` code can be rewritten as follows:

```
List <Student> result1 = db.queryByExample(Student.class);
System.out.println("Number of students: " + result1.size()+"\n");
for (int i=0; i<result1.size(); i++) {
  Student s = result1.get(i);
  System.out.println(s);
}
```

The `queryByExample()` method can also be invoked with a template object and the method returns all objects that match the template object in the non-default values provided. For example, the following sequence of statements will query the database for all students whose last names is "`Smith`":

```
Student s = new Student(0,"Smith",null,(float) 0.0);
ObjectSet result = db.queryByExample(s);
```

Default values are `0` for `int`, `0.0` for `float`/`double`, and `null` for `String` and other objects. Default values act as wild cards in the template object and objects that match the non-default values in the template are returned by the method.

6.3 Database Updates and Deletes

Updating and deleting objects is quite straightforward in db4o. In either case, the first step is to retrieve the object to be updated or deleted. To update the object, an appropriate method defined in the class for the object is then invoked, which performs the necessary update in memory. Then, the `store()` method is invoked to make the update persistent on disk. Here is a code fragment that changes the GPA of student with `sid=1000`:

```
Student s = new Student(1000,null,null,(float) 0.0);
List <Student> result1 = db.queryByExample(s);
s = (Student) result1.get(0);
s.setGpa((float) 3.67);
db.store(s);
```

Here, the student object corresponding to `sid=1000` is retrieved in the first three lines of code. Then, the `setGpa()` method is called to update the GPA of the student. Finally, the `store()` method is called to make the change permanent.

To delete an object, the `delete()` method is called on the object container database object with the object to be deleted as the argument. For example, if the student with `sid=1002` is deleted by executing the following code:

```
s = new Student(1002,null,null,(float) 0.0);
List <Student> result2 = db.queryByExample(s);
s = (Student) result2.get(0);
db.delete(s);
```

6.4 Company Database

In this section, the company database of the Elmasri-Navathe textbook is represented as an object-oriented database. The design includes classes for entity sets *Employee*, *Department*, *Project*, and *Dependent*. To represent the many-to-many relationship *worksOn*, a separate class is designed. The class definitions showing only the attributes are defined as follows:

```
public class Department {
  // attributes
  private int dnumber;
  private String dname;
  private Vector<String> locations;
  // relationships
  private Vector<Employee> employees;
  private Employee manager;
  private Vector<Project> projects;
  // one-to-many relationship (manager) attribute
  private String mgrStartDate;
}
```

```
public class Employee {
  // attributes
  private int ssn;
  private String fname;
  private char minit;
  private String lname;
  private String address;
  private String bdate;
  private float salary;
  private char sex;
  //relationships
  private Department worksFor;
  private Department manages;
  private Vector<WorksOn> worksOn;
  private Vector<Dependent> dependents;
  private Employee supervisor;
  private Vector<Employee> supervisees;
}

public class Project {
  // attributes
  private int pnumber;
  private String pname;
  private String location;
  // relationships
  Department controlledBy;
  Vector<WorksOn> worksOn;
}

public class Dependent {
  // attributes
  private String name;
  private char sex;
  private String bdate;
  private String relationship;
  // relationships
  private Employee dependentOf;
}

public class WorksOn {
  // attribute
  float hours;
  //owner attributes
  Employee employee;
  Project project;
}
```

The above design is a straightforward translation of the ER diagram for the Company database shown in page 225 of the 6th edition of the Elmasri-Navathe textbook. Simple attributes of entity sets are represented as instance variables of primitive types (e.g. `ssn` in `Employee`) and multiple-valued attributes as Java Vectors (e.g. `locations` in `Department`). Single-valued

relationships are represented as references to objects (e.g. `manager` in `Department`) and many-valued relationships are represented as Vectors of object references (`employees` in `Department`). The only many-to-many relationship, `worksOn`, is represented by a separate class with a simple attribute, hours, and two object references, one to `Employee` object and the other to `Project` object involved in the relationship. The usual constructors, getter and setter methods are also defined in the classes.

6.5 Database Querying

There are three main methods to query and retrieve objects in db4o: Query by Example, Native Queries, and SODA (Simple Object Database Access).

6.5.1 Query by Example

The query by example method has already been discussed. In this approach, an object template is provided as input to the `queryByExample()` method, which then retrieves all objects that match the non-default values of the template. This method works well in many cases, but has its limitations. For example, one cannot constrain objects on default values such as 0, null etc as these are treated as default values; one cannot perform advanced query operations such as "and", "or", and "not".

6.5.2 Native Queries

To avoid the limitations of the query by example method, db4o provides the native queries system. Native queries provide the ability to execute one or more lines of code on all instances of a class and select a subset of the instances that satisfy a criterion specified by the code. Here is a simple example in Java 1.5 to retrieve the department object for department with `dnumber = 5`.

```
List<Department> depts = db.query(new Predicate<Department>() {
  public boolean match(Department dept) {
    return (dept.getDnumber() == 5);
  }
});
Department d = depts.get(0);
```

The `query()` method takes a `Predicate` object as its argument. The `Predicate` object encapsulates a Boolean method called `match()` which will be applied to all instances of the class on which the predicate is defined. All instances that evaluate to `True` will be returned as the value of the `query()` method.

As a more complicated query, consider "*Find departments that have a location in Houston or have less than 4 employees or controls a project located in Phoenix*". The following Native query code prints such departments:

```
List<Department> depts = db.query(new Predicate<Department>() {
  public boolean match(Department dept) {
```

```
      int nEmps = dept.getEmployees().size();
      Vector<Project> prjs = dept.getProjects();
      boolean foundPhoenix = false;
      for (int i=0; i<prjs.size(); i++) {
        Project p = prjs.get(i);
        if (p.getLocation().equals("Phoenix")) {
          foundPhoenix = true;
          break;
        }
      }
      return dept.getLocations().contains("Houston") ||
             (nEmps < 4) || foundPhoenix;
    }
  });
  for (int i=0; i<depts.size(); i++)
    System.out.println("Department: "+depts.get(i));
```

The `match()` method checks for each of the three conditions (department has a location in `Houston`, department has less than 4 employees, and department controls a project located in `Phoenix`) and returns true if any one or more of the conditions is satisfied by the department. The example illustrates the power of Native queries where complex conditions can be coded in Java to be tested on instances of the class.

6.5.3 SODA (Simple Object Database Access) Queries

SODA API is db4o's low-level access to the nodes of the data graph of the underlying object-oriented database. It gives flexibility in expressing dynamic queries and therefore is an important too to use to build object-oriented database applications. For most routine situations, however, native queries are an excellent choice.

For the next set of examples, let us assume the following method to print the results of the query is available:

```
public static void printResult(ObjectSet result) {
    System.out.println(result.size());
    while(result.hasNext()) {
        System.out.println(result.next());
}
```

Query 1: Find employees with last name `King`.

```
Query query = db.query();
query.constrain(Employee.class);
query.descend("lname").constrain("King");
ObjectSet result=query.execute();
printResult(result);
```

The above code fragment first creates a `Query` object that represents a query graph with nodes denoting database objects that have constraints associated with them. The `Query` object is created by invoking the `query()` method on the `ObjectContainer db`. The `constrain()` method allows us to attach a constraint to the node in the query graph. A commonly used special argument to the method is a class, e.g. `Employee.class` in the above example, which constrains the node to objects of the class. The `descend()` method creates a child node in the query graph and associates the node with the specified field name parameter. In the example, a child node is created for the `lname` field, which is constrained by the string "`King`". Once the query graph is created with all the constraints, the query can be executed with the execute() method, which returns all objects for the root node of the query graph that satisfy the various constraints imposed. The above code fragment will print the employees with last name `King`.

Query 2: Find employees with salary = 25000.

```
Query query = db.query();
query.constrain(Employee.class);
query.descend("salary").constrain(new Integer(25000));
ObjectSet result = query.execute();
printResult(result);
```

This example is similar to the previous one except that the constraint is on the salary field with an `Integer` object.

Query 3: Find projects not located in Houston.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("location").constrain("Houston").not();
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `not()` modifier on a constraint. Any constraint that is associated with a query graph node is first evaluated and then negated.

Query 4: Find employees with last name King and salary = 44000.

```
Query query = db.query();
query.constrain(Employee.class);
Constraint constr = query.descend("lname").constrain("King");
query.descend("salary").constrain(new Integer(44000)).and(constr);
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `and()` modifier. A separate Constraint object for constraining the last name is created and is combined with the salary constraint using the `and()` modifier.

Query 5: Find projects pnumber > 90.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("pnumber").constrain(new Integer(90)).greater();
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of `greater()` modifier to make comparisons other than "equality", the default comparison in the constrain() method. Other methods available are `smaller()`, `like()`, `startsWith()`, and `equal()`.

Query 6: Find projects with pnumber > 90 or with location in Houston.

```
Query query = db.query();
query.constrain(Project.class);
Constraint constr = query.descend("location").constrain("Houston");
query.descend("pnumber").constrain(new Integer(90))
                        .greater().or(constr);
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `or()` modifier.

Query 7: Find employees in sorted order.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("pname").orderAscending();
ObjectSet result = query.execute();
printResult(result);
query.descend("pname").orderDescending();
result = query.execute();
printResult(result);
```

This example illustrates the `orderAscending()` and `orderDescending()` methods available to sort the results of a query.

Query 8: Find departments whose manager's last name is Wong.

```
Query query = db.query();
query.constrain(Department.class);
query.descend("manager").descend("lname").constrain("Wong");
ObjectSet result = query.execute();
Department d = (Department) result.next();
System.out.println("Wong managed department: "+d);
```

This example illustrates SODA querying which requires traversing an object reference. Here, the `manager` field of `Department` object is descended in the SODA query.

6.6 Company Database Application

In this section, a complete application that creates and queries the company database is introduced. It is assumed that the five classes: `Employee.java`, `Department.java`, `Project.java`, `Dependent.java`, and `WorksOn.java` are created as defined in Section 6.4. Included in their definitions are standard constructor, setter, getter, and toString methods. Once these class definitions are compiled the following application modules can be created.

6.6.1 CreateDatabase.java

The first module in the company application is the `CreateDatabase` program that reads data from text files and creates the object-oriented database. The main program along with the import statements is shown below:

```
import java.util.*;

import com.db4o.*;
import com.db4o.config.Configuration;
import com.db4o.query.*;

public class CreateDatabase {
  public static void main(String[] args) {
    String DB4OFILENAME = args[0];
    Configuration config = Db4o.configure();
    config.objectClass(Employee.class).cascadeOnUpdate(true);
    config.objectClass(Department.class).cascadeOnUpdate(true);
    config.objectClass(Project.class).cascadeOnUpdate(true);
    config.objectClass(Dependent.class).cascadeOnUpdate(true);
    config.objectClass(WorksOn.class).cascadeOnUpdate(true);
    config.updateDepth(1000);
    ObjectContainer db = Db4o.openFile(config,DB4OFILENAME);

    try {
      createEmployees(db);
      createDependents(db);
      createDepartments(db);
      createProjects(db);
      setManagers(db);
      setControls(db);
      setWorksfor(db);
      setSupervisors(db);
      createWorksOn(db);
      db.commit();
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    }
    finally {
```

```
        db.close();
      }
      System.out.println("DONE");
    }
  }
```

The main program takes as command line argument the name of the database file, `company.db4o`. This file will contain the object-oriented database at the termination of the program. Initially, a `Configuration` object, `config`, is created and several properties are set. For each object class, the `cascadeOnUpdate` property is set to `true` using the following Java statement (shown for the `Employee class`):

```
config.objectClass(Employee.class).cascadeOnUpdate(true);
```

This property indicates to the db4o database engine that all updates should be made persistent including chained references. By default, in response to the `store()` method call, db4o makes only the top-level object persistent and any chained references to objects such as Vector of references etc. are not made persistent. By setting the property to `true`, db4o makes all chained references persistent as well. The other property setting is:

```
config.updateDepth(1000);
```

This sets the depth of the chained references to be a large number. After these settings, the database object is created. Once the database object is created, it can be used to create objects within each class by reading data from text files by calling various methods. Each of these methods is discussed next.

6.6.2 createEmployees

This method reads data about employees from a text file which contains the number of employees in the first line and details about each employee in subsequent lines. The individual fields describing the employee are separated by a colon. The first few lines of the text file, `employee.dat`, are shown below:

```
40
James:E:Borg:888665555:10-NOV-27:450 Stone, Houston, TX:M:55000
Franklin:T:Wong:333445555:08-DEC-45:638 Voss, Houston, TX:M:40000
Jennifer:S:Wallace:987654321:20-JUN-31:291 Berry, Bellaire, TX:F:43000
```

The method makes use of a text file reading class called `InputFile.java` which is provided along with the source code of the lab manual. The methods of this class are self explanatory. The code for the `createEmployees` method is shown below:

```
public static void createEmployees(ObjectContainer db)
                                    throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/employee.dat")) {
```

```
      int nEmps = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nEmps; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        String fname = fields[0];
        char minit = fields[1].charAt(0);
        String lname = fields[2];
        int ssn = Integer.parseInt(fields[3]);
        String bdate = fields[4];
        String address = fields[5];
        char sex = fields[6].charAt(0);
        float salary = Float.parseFloat(fields[7]);
        Employee e = new Employee(
          ssn, fname, minit, lname, address, bdate, salary, sex);
        db.store(e);
      }
    }
  }
```

The method starts off by opening the text file and reading the first line into an integer variable. Then, for each employee, it reads the details into a string variable, uses the `split()` method to break up the individual fields, and finally creates the `Employee` object by calling its constructor and makes the object persistent by invoking the `store()` method. Note that at this point, none of the employee objects have their object references to other objects set. These will be done in subsequent method calls.

6.6.3 createDependents

The data file (`dependent.dat`) from which the `createDependent` method reads information about dependent objects is shown below:

```
11
333445555,Alice,F,05-APR-1976,Daughter
333445555,Theodore,M,25-OCT-1973,Son
333445555,Joy,F,03-MAY-1948,Spouse
987654321,Abner,M,29-FEB-1932,Spouse
123456789,Michael,M,01-JAN-1978,Son
123456789,Alice,F,31-DEC-1978,Daughter
123456789,Elizabeth,F,05-MAY-1957,Spouse
444444400,Johnny,M,04-APR-1997,Son
444444400,Tommy,M,07-JUN-1999,Son
444444401,Chris,M,19-APR-1969,Spouse
444444402,Sam,M,14-FEB-1964,Spouse
```

It has a similar format as the `employee.dat` file. The first line contains the number of dependents in the file and the remaining lines contain the individual fields describing the dependent. The first field is the social security number of the employee owner of the dependent. The following is the code for creating the dependent objects:

```
public static void createDependents(ObjectContainer db)
          throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/dependent.dat")) {
    int nDeps = Integer.parseInt(fin.readLine());
    for (int i = 0; i < nDeps; i++) {
      String line = fin.readLine();
      String[] fields = line.split(",");
      final int essn = Integer.parseInt(fields[0]);
      String name = fields[1];
      char sex = fields[2].charAt(0);
      String bdate = fields[3];
      String relationship = fields[4];
      List<Employee> emps = db.query(new Predicate<Employee>() {
        public boolean match(Employee emp) {
          return (emp.getSsn() == essn);
        }
      });
      Employee e = emps.get(0);
      Dependent d = new Dependent(name, sex, bdate, relationship);
      d.setDependentOf(e);
      db.store(d);
      e.addDependent(d);
      db.store(e);
    }
  }
}
```

The above method proceeds in a similar manner as `createEmployees`. One main difference is that a native query is executed to obtain a reference to the employee object corresponding to the social security member. The dependent object is created and the reference to the employee object is set by calling the `setDependentOf()` method. At the same time, a reference to the dependent object is set in the employee object using the method `addDependent()`. Both updates are made persistent by calling the `store()` method.

6.6.4 createDepartment

The data file (department.dat) consists of information about departments. The locations for the department listed at the end of the line, separated by commas. A sample file is shown below:

```
6
1:Headquarters:Houston
4:Administration:Stafford
5:Research:Bellaire,Sugarland,Houston
6:Software:Atlanta,Sacramento
7:Hardware:Milwaukee
8:Sales:Chicago,Dallas,Philadephia,Seattle,Miami
```

The code for the method is shown below:

```
  public static void createDepartments(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/department.dat")) {
      int nDepts = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nDepts; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        int dnumber = Integer.parseInt(fields[0]);
        String dname = fields[1];
        String[] ls = fields[2].split(",");
        Vector<String> locs = new Vector<String>();
        for (int j = 0; j < ls.length; j++)
          locs.add(ls[j]);
        Department d = new Department(dnumber, dname, locs);
        db.store(d);
      }
    }
  }
```

The code is similar to previous methods. One difference is that a vector of string objects is created to store the different locations for the department. Again, references to other objects are not stored in this method. This will be done in separate methods to follow.

6.6.5 createProjects

The data file for creating project objects is shown below:

```
11
1,Product X,Bellaire
2,Product Y,Sugarland
3,Product Z,Houston
10,Computerization,Stafford
20,Reorganization,Houston
30,New Benefits,Stafford
61,Operating Systems,Jacksonville
62,Database Systems,Birmingham
63,Middleware,Jackson
91,Inkjet Printers,Phoenix
92,Laser Printers,Las Vegas
```

The code to read this data and create project objects is shown below:

```
  public static void createProjects(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/project.dat")) {
      int nProjs = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nProjs; i++) {
```

```
        String line = fin.readLine();
        String[] fields = line.split(",");
        int pnumber = Integer.parseInt(fields[0]);
        String pname = fields[1];
        String loc = fields[2];
        Project p = new Project(pnumber, pname, loc);
        db.store(p);
      }
    }
  }
```

The code is self-explanatory as it is very similar to previous examples.

6.6.6 createWorksOn

The `worksOn` relationship is a many-to-many relationship between `Employee` and `Project`. The relationship also contains an attribute, `hours`. This has been modeled by a separate class, `WorksOn` which has only one ordinary attribute, `hours`. It also consists of two object reference, one points to the employee object and the other to the project object involved in the relationship. A portion of the data file is shown below:

```
48
123456789,1,32.5
123456789,2,7.5
666884444,3,40.0
453453453,1,20.0
```

There are three fields describing each worksOn relationship: social security number of employee, project number, and number of hours. The code to read the file and create the objects is shown below:

```
  private static void createWorksOn(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/worksOn.dat")) {
      int nWorksOn = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nWorksOn; i++) {
        String line = fin.readLine();
        String[] fields = line.split(",");
        final int essn = Integer.parseInt(fields[0]);
        final int pno = Integer.parseInt(fields[1]);
        float hours = Float.parseFloat(fields[2]);
        List<Employee> emps = db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = emps.get(0);
        List<Project> prjs = db.query(new Predicate<Project>() {
```

```
          public boolean match(Project prj) {
            return (prj.getPnumber() == pno);
          }
        });
        Project p = prjs.get(0);
        WorksOn won = new WorksOn(hours);
        won.setEmployee(e);
        won.setProject(p);
        db.store(won);
        e.addWorksOn(won);
        p.addWorksOn(won);
        db.store(e);
        db.store(p);
      }
    }
  }
```

For each `worksOn` relationship entry, the method retrieves the `Employee` object for the given social security number and the `Project` object for the given project number using native query approach. Once these object references are obtained, a `worksOn` object is created with the given hours value and the two object references are set. Finally, all objects are made persistent using the `store()` method call.

6.6.7 setManagers

This method sets the object references for the one-to-one relationship, `Manages`. The data file contains the department number, its manager's social security number and the start date for each department. The data file contents are shown below:

```
6
1,888665555,19-JUN-1971
4,987654321,01-JAN-1985
5,333445555,22-MAY-1978
6,111111100,15-MAY-1999
7,444444400,15-MAY-1998
8,555555500,01-JAN-1997
```

The following method reads the data present in the file and sets the appropriate object references in the already created employee and department objects.

```
  public static void setManagers(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/manager.dat")) {
      int nMgrs = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nMgrs; i++) {
        String line = fin.readLine();
        String[] fields = line.split(",");
        final int dno = Integer.parseInt(fields[0]);
```

```
        final int essn = Integer.parseInt(fields[1]);
        String startDate = fields[2];
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        List<Employee> emps = db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = emps.get(0);
        d.setMgrStartDate(startDate);
        e.setManages(d);
        d.setManager(e);
        db.store(d);
        db.store(e);
      }
    }
  }
```

The method finds the `Employee` object given the social security number and the `Department` object given the department number using native query approach. Then, it sets the appropriate object references in the two objects to point to the other. It also sets the start date field in the `Department` object. Finally, the method makes the changes persistent by calling `store()`.

6.6.8 setControls

This method sets the object references for the one-to-many relationship, controls, between Department and Project. The data file consists of the department number followed by a list of project numbers of projects controlled by the department separated by commas. The data file is shown below:

```
5
1:20
4:10,30
5:1,2,3
6:61,62,63
7:91,92
```

The following method read the data file and sets the object references for the relationship.

```
  public static void setControls(ObjectContainer db)
              throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/controls.dat")) {
      int nControls = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nControls; i++) {
```

```
        String line = fin.readLine();
        String[] fields = line.split(":");
        final int dno = Integer.parseInt(fields[0]);
        String[] projects = fields[1].split(",");
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        for (int j = 0; j < projects.length; j++) {
          final int pno = Integer.parseInt(projects[j]);
          List<Project> prjs = db.query(new Predicate<Project>() {
            public boolean match(Project prj) {
              return (prj.getPnumber() == pno);
            }
          });
          Project p = prjs.get(0);
          p.setControlledBy(d);
          db.store(p);
          d.addProject(p); // add p to the "projects" vector of d
        }
        db.store(d);
      }
    }
  }
```

The above code retrieves the Department object for the given department number. Then, for each project number it retrieves the `Project` object and then sets the appropriate object references in both these objects. Finally, the updates are made persistent using the `store()` method.

6.6.9 setWorksFor

This method sets the object references for the one-to-many relationship, `worksFor`, between `Employee` and `Department`. It reads the data from a text file whose first few lines are shown below:

```
6
1:888665555
4:987654321,987987987,999887777
5:123456789,333445555,666884444,453453453
```

The department number is followed by a list of the social security numbers of employees working for the department. The following method reads this data and sets the object references for the relationship.

```
  private static void setWorksFor(ObjectContainer db) throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/worksFor.dat")) {
```

```
      int nWorksFor = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nWorksFor; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        final int dno = Integer.parseInt(fields[0]);
        String[] emps = fields[1].split(",");
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        for (int j = 0; j < emps.length; j++) {
          final int ssn = Integer.parseInt(emps[j]);
          List<Employee> es = db.query(new Predicate<Employee>() {
            public boolean match(Employee emp) {
              return (emp.getSsn() == ssn);
            }
          });
          Employee e = es.get(0);
          e.setWorksFor(d);
          db.store(e);
          d.addEmployee(e); // add e to "employees" vector of d
        }
        db.store(d);
      }
    }
  }
```

This method is identical to the `setControls()` method discussed before.

6.6.10 setSupervisors

This method sets the object references for the one-to-many relationship, `supervisor`, between `Employee` and `Employee`. It reads the data from a text file whose first few lines are shown below:

```
19
888665555:333445555,987654321
333445555:123456789
987654321:999887777,987987987
333445555:666884444,453453453
111111100:111111101,111111102,111111103
222222200:222222201,222222202,222222203
222222201:222222204,222222205
```

The supervisor number is followed by a list of the social security numbers of employees working under the supervisor. The following method reads this data and sets the object references for the relationship.

```
private static void setSupervisors(ObjectContainer db)
                throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/sups.dat")) {
    int nSups = Integer.parseInt(fin.readLine());
    for (int i = 0; i < nSups; i++) {
      String line = fin.readLine();
      String[] fields = line.split(":");
      final int superssn = Integer.parseInt(fields[0]);
      String[] subs = fields[1].split(",");
      List<Employee> emps = db.query(new Predicate<Employee>() {
        public boolean match(Employee emp) {
          return (emp.getSsn() == superssn);
        }
      });
      Employee s = emps.get(0);
      for (int j = 0; j < subs.length; j++) {
        final int essn = Integer.parseInt(subs[j]);
        List<Employee> subworkers=db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = subworkers.get(0);
        e.setSupervisor(s);
        db.store(e);
        s.addSupervisee(e); // add e to "supervisees" vector of s
      }
      db.store(s);
    }
  }
}
```

This method is identical to the `setControls()` method discussed before.

6.6.11 Complex Retrieval Example

In this example, a complex retrieval is illustrated. Consider the request: *retrieve departments that have a location in Houston or have less than 4 employees or controls a project located in Phoenix*. The code to solve this problem is shown below in its entirety.

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.config.Configuration;
import com.db4o.query.Predicate;

import java.io.*;
import java.util.List;
import java.util.Vector;
```

```
public class DisplayDept2 {
  public static void main(String[] args) {
    String DB4OFILENAME = args[0];
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,DB4OFILENAME);
    try {
      List<Department> depts = db.query(new Predicate<Department>() {
        public boolean match(Department dept) {
          int nEmps = dept.getEmployees().size();
          Vector<Project> prjs = dept.getProjects();
          boolean foundPhoenix = false;
          for (int i=0; i<prjs.size(); i++) {
            Project p = prjs.get(i);
            if (p.getLocation().equals("Phoenix")) {
              foundPhoenix = true;
              break;
            }
          }
          return dept.getLocations().contains("Houston") ||
                       (nEmps < 4) ||
                       foundPhoenix;
        }
      });
      for (int i=0; i<depts.size(); i++)
        System.out.println("Department: "+depts.get(i));
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
      }
    finally {
      db.close();
    }
  }
```

The main part of this code is the native query predicate specification. The `match` method checks for the number of employees by examining the size of the employees vector in the department object (`dept.getEmployees().size()`). It also check for the "Houston" location of the department by the expression: `dept.getLocations().contains("Houston")`. To check for the location of the projects, a loop is set up for all projects controlled by the department and each project's location is checked. Rest of the code is self-explanatory.

6.7 Web Application

This section introduces methodology to access db4o databases from the Web. The company browser example of Chapter 4 is duplicated in Java with the company database that is already created in this chapter.

6.7.1 Client-Server Configuration

For a Web application to access the db4o database, a client-server configuration of the database is more appropriate. The following class, `Db4oServletContextListener`, is associated with a Servlet Context:

```
import java.io.File;
import javax.servlet.*;
import com.db4o.*;

public class Db4oServletContextListener
                implements ServletContextListener {

  public static final String KEY_DB4O_FILE_NAME = "db4oFileName";
  public static final String KEY_DB4O_SERVER = "db4oServer";
  private ObjectServer server=null;

  public void contextInitialized(ServletContextEvent event) {
    close();
    ServletContext context=event.getServletContext();
    String filePath =
      context.getRealPath("WEB-INF/db/"+
                context.getInitParameter(KEY_DB4O_FILE_NAME));
    server = Db4o.openServer(filePath,0);
    context.setAttribute(KEY_DB4O_SERVER,server);
    context.log("db4o startup on "+filePath);
  }

  public void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.removeAttribute(KEY_DB4O_SERVER);
    close();
    context.log("db4o shutdown");
  }

  private void close() {
    if(server!=null
      server.close();
    server=null;
  }
}
```

The `contextInitialized()` method is invoked when the Context is activated; The method creates a db4o server object by associating it with a db4o database available in the `WEB-INF/db` directory of the Web application; the name of the file is available in the `web.xml` file of the Web application as a Context parameter, `db4oFileName`. The server reference is saved in the Context parameter `db4oServer` for other servlets to look up and use. The `contextDestroyed()` method is invoked when the Context is shut down. This method destroys the db4o server object.

The `web.xml` file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
  <web-app>
    <context-param>
      <param-name>db4oFileName</param-name>
      <param-value>company.db4o</param-value>
    </context-param>
    <listener>
      <listener-class>Db4oServletContextListener</listener-class>
    </listener>
    <servlet>
      <servlet-name>DisplayDepartment</servlet-name>
      <servlet-class>DisplayDepartment</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>DisplayDepartment</servlet-name>
      <url-pattern>/DisplayDepartment</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

The company database browser application consists of the following servlets:

(1) `AllDepartments.java`: This servlet displays all departments in the database in a HTML table format with two columns, one for department number and he other for department name. The department number values are hyper-linked to the `DisplayDepartment` servlet; the department number is sent as a `GET` argument under the name `dno`.
(2) `AllEmployees.java`: This servlet produces a listing of all employees and is similar to `AllDepartments`.
(3) `AllProjects.java`: This servlet poduces a listing of all projects and is similar to `AllDepartments`.
(4) `DisplayDepartment.java`: This servlet accepts the `dno` parameter and produces a detailed listing of all the details of the given department. It lists the name of the department, it's manager name (hyper-linked to employee detail) and start date, a listing of all department locations, a tabular listing of all employees who work for the department with the employee `ssn` hyper-linked to the servlet that produces the employee details, and a listing of all projects controlled by the department with the project number hyper-linked to the servlet that produces the project details.
(5) `DisplayEmployee.java`: This servlet is similar to `DisplayDepartment` and produces a listing of the given employee's details.
(6) `DisplayProject.java`: This servlet is similar to `DisplayDepartment` and produces a listing of the given project details.

The code for the `AllDepartments` servlet is given below:

```
  import com.db4o.*;
  import com.db4o.query.*;
```

```
import java.io.*;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class AllDepartments extends HttpServlet {

  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
  doPost(request,response);
  }

  public void doPost (HttpServletRequest request,
                      HttpServletResponse response)
      throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    ServletContext context = getServletContext();
    ObjectServer server =
        (ObjectServer)context.getAttribute("db4oServer");
    ObjectContainer db = server.openClient();

    out.println("<html>");
    out.println("<head>");
    out.println("<title>All Departments</title>");
    out.println("</head>");
    out.println("<body>");

    out.println("<h2>Departments of Company</h2>");
    out.println("<table border=2>");
    out.println("<tr>");
    out.println("<th>Department Number</th>");
    out.println("<th>Department Name</th>");
    out.println("</tr>");

    try {
      Query query = db.query();
      query.constrain(Department.class);
      query.descend("dnumber").orderAscending();
      ObjectSet results = query.execute();
      while (results.hasNext()) {
        Department d = (Department) results.next();
        out.println("<tr>");
        out.println("<td><a href=\"DisplayDepartment?dno="+
                    d.getDnumber()+"\">"+
                    d.getDnumber()+"</a></td>");
        out.println("<td>"+d.getDname()+"</td>");
        out.println("</tr>");
      }
```

```
      } catch (Exception e) {
          out.println("Exception: " + e.getMessage());
        }

      out.println("</body>");
      out.println("</html>");
      out.close();

      db.close();
    }
  }
```

To connect to the db4o server object, the servlet retrieves the servlet Context using the following statement:

```
ServletContext context = getServletContext();
```

Then, it retrieves the db4o server object that was created at the start of the Context using the statement:

```
ObjectServer server =
        (ObjectServer)context.getAttribute("db4oServer");
```

Finally, the `ObjectContainer` object is obtained using the `openClient()` method as follows:

```
ObjectContainer db = server.openClient();
```

The `ObjectContainer` object is used to perform various database transactions and at the end it is closed. Notice that a SODA query is used to retrieve all the `Department` objects in a sorted manner. Rest of the servlet code is self-explanatory.

The `DisplayDepartment` servlet code is shown next.

```
import com.db4o.*;
import com.db4o.query.Predicate;
import java.io.*;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class DisplayDepartment extends HttpServlet {
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doPost(request,response);
  }

  public void doPost (HttpServletRequest request,
```

```
                HttpServletResponse response)
  throws ServletException, IOException {
final int dno = Integer.parseInt(request.getParameter("dno"));

response.setContentType("text/html");
PrintWriter out = response.getWriter();

ServletContext context = getServletContext();
ObjectServer server =
    (ObjectServer)context.getAttribute("db4oServer");
ObjectContainer db = server.openClient();

out.println("<html>");
out.println("<head>");
out.println("<title>Department View</title>");
out.println("</head>");
out.println("<body>");
try {
  List<Department> depts = db.query(new Predicate<Department>() {
    public boolean match(Department dept) {
      return (dept.getDnumber() == dno);
    }
  });

  Department d = depts.get(0);
  out.println("<B>Department: </B>" + d.getDname());
  out.println("<P>Manager: <a href=\"DisplayEmployee?ssn="+
              d.getManager().getSsn()+"\">"+
              d.getManager().getLname()+", "+
              d.getManager().getFname()+"</a></br>");
  out.println("Manager Start Date: "+d.getMgrStartDate());

  out.println("<h4>Department Locations:</h4>");
  for (int i=0; i<d.getLocations().size(); i++)
    out.println(d.getLocations().get(i)+"<BR>");

  out.println("<h4>Employees:</h4>");
  out.println("<table border=2>");
  out.println("<tr>");
  out.println("<th>Employee SSN</th>");
  out.println("<th>First Name</th>");
  out.println("<th>Last Name</th>");
  out.println("</tr>");

  for (int i=0; i<d.getEmployees().size(); i++) {
    Employee e = d.getEmployees().get(i);
    out.println("<tr>");
    out.println("<td><a href=\"DisplayEmployee?ssn="+
                e.getSsn()+"\">"+e.getSsn()+"</a></td>");
    out.println("<td>"+e.getFname()+"</td>");
    out.println("<td>"+e.getLname()+"</td>");
    out.println("</tr>");
```

```
        }
        out.println("</table>");

        out.println("<h4>Projects:</h4>");
        out.println("<table border=2 cellspacing=2 cellpadding=2>");
        out.println("<tr>");
        out.println("<th>Project Number</th>");
        out.println("<th>Project Name</th>");
        out.println("</tr>");

        for (int i=0; i<d.getProjects().size(); i++) {
          Project p = d.getProjects().get(i);
          out.println("<tr>");
          out.println("<td><a href=\"DisplayProject?pno="+
                      p.getPnumber()+"\">"+p.getPnumber()+"</a></td>");
          out.println("<td>"+p.getPname()+"</td>");
          out.println("</tr>");
        }
        out.println("</table>");
      } catch (Exception e) {
          out.println("Exception: " + e.getMessage());
        }

      out.println("</body>");
      out.println("</html>");
      out.close();

      db.close();
    }
}
```

The above code connects to the db4o server and performs a native query to find the department object corresponding to the `dno` value that is sent as a parameter. The details of the department object are then printed. First the department name, manager name and start date are printed, then a list of all employees working for the department are printed in tabular format, and finally a list of all projects controlled by the department is printed in tabular format. All employee `ssn` values as well as project numbers are hyper-linked to the detail pages.

Exercises

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

1. Consider the following class definitions (only instance variables are shown) for a Portfolio Management database:

```
public class Security {
  // Attribute Data members
  private String symbol;
```

```
  private String companyName;
  private float currentPrice;
  private float askPrice;
  private float bidPrice;
  // Relationship Data Members
  // Set of transactions for this security
  private Vector<Transaction> transactions;
}

public class Member {
  // Attribute Data members
  private String mid;
  private String password;
  private String fname;
  private String lname;
  private String address;
  private String email;
  private double cashBalance;
  // Relationship Data Members
  // Set of transactions for this member
  private Vector<Transaction> transactions;
}

public class Transaction {
  // Attribute Data members
  private Date transDate;
  private String transType; // Buy or Sell
  private float quantity;
  private float pricePerShare;
  private float commission;
  // Relationship Data Members
  private Member member; // member for this transaction
  private Security security; // security for this transaction
}
```

(a) Write a Java program that reads data from a text file (security.dat) and creates security objects in a db4o database. The first line of the text file contains a number denoting the number of securities that are described. Every subsequent five lines describe one security (symbol, company name, current price, ask price, and bid price). A sample data file follows:

```
3
ORCL
Oracle Corporation
15.75
15.25
15.45
SUNW
Sun Microsystems
14.75
14.25
```

```
14.45
MSFT
Microsoft
55.75
55.25
55.45
```

(b) Write a Java program that implements the following menu of functions on the database:

```
(1) Member Log In
(2) New Member Sign In
(3) Quit
```

A new member may use option (2) to create a new account. This option prompts the new member for the member id, a password, first name, last name, address, email, and initial cash balance. A new member object should be created if no existing member has the same member id, otherwise an error message should be printed. An existing member can use option (1) to login to their account. The member is prompted for the member id and password. Invalid member id or password should result in an error message. Upon successful login, the following menu of options should be presented:

```
(1)  View Portfolio
(2)  Print Monthly Report
(3)  Update Account Data
(4)  Price Quote
(5)  Buy
(6)  Sell
(7)  Quit
```

View Portfolio option should produce a well-formatted report of all current holdings and their values, with a total value displayed at the end. Print Monthly Report should prompt the user for the month and year and a monthly report of all transactions in that month should be displayed. Update Account should prompt user for new values of password, address, and email. Price Quote should prompt the user for a security symbol and then display the current, ask, and bid prices. Buy and Sell should prompt the user for stock symbol and number of shares and perform the action if possible. Otherwise, an error message should be generated.

2. Convert the application described in the previous problem into a Web application with appropriate user interfaces.

3. Consider the GRADEBOOK database described in Exercise 6.36 (Page 193) of the ElMasri/Navathe textbook. In addition to the tables described there, consider the following two additional tables:

```
component(Term,Sec_no,Compname,Maxpoints,Weight)
score(Sid,Term,Sec_no,Compname,Points)
```

The component table records all the grading components for a course offering such as exams, quizzes, home work assignments etc. with each component given a name, a maximum points and a weight (between 0 and 100). The sum of weights of all components for a course offering should normally be 100. The score table records the points awarded to a student for a particular course component for a course offering in which he or she is enrolled.

(a) Design a db4o database schema for the GRADEBOOK database.
(b) Write a Java program that implements the following menu of options for a user (some user interaction is shown – for other menu options you may introduce appropriate user program-user interactions).

```
GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 1
Course Number: 6710
Course Title: Database Systems
Added Catalog Entry

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 2
Term: sp02
Section Number:
5556
Course Number: 6710
A Cutoff: 90
B Cutoff: 80
C Cutoff: 70
D Cutoff:
60
Course was added! Success!

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
```

```
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 3
ID (0 to stop): 1111
Last Name: Jones
First Name: Tony
Middle Initial: A
ID (0 to stop): 2222
Last Name: Smith
First Name: Larry
Middle Initial: B
ID (0 to stop): 0

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 4
Term: sp02
5556 6710 Database Systems

Select a course line number: 5556

SELECT COURSE SUB-MENU

(1) Add Students to Course
(2) Add Course Components
(3) Add Student Scores
(4) Modify Student Score
(5) Drop Student from Course
(6) Print Course Report
(q) Quit

Type in your option: 1
Student Id (0 to stop): 1111
Student Id (0 to stop): 2222
Student Id (0 to stop): 0

SELECT COURSE SUB-MENU

(1) Add Students to Course
(2) Add Course Components
(3) Add Student Scores
(4) Modify Student Score
(5) Drop Student from Course
(6) Print Course Report
```

```
(q) Quit

Type in your option: q

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: q
```

(c) Write a Java program that reads a text file with the following format

```
sp02
5556
1111 Jones Tony A
2222 Smith Larry B
0000
```

The first line in the text file contains the term, the second line contains the Sec_no and subsequent lines contain the student id, last name, first name, and middle initial per line for each student enrolled in the course. The program should enroll the students in the course. If the student object already exists, only the enrollment should take place, however, if the student object does not exist, it must be created and then the enrollment should take place. You may assume that the course object already exists.

4. Convert the application described in the previous problem into a Web application with appropriate user interfaces.

5. Consider the following class definitions for a geographical database of states and cities of the United States:

```
public class State {
  // Attribute Data members
  private String stateCode;
  private String stateName;
  private String nickname;
  private int population;
  // Relationship Data Members
  // Capital city
  private City capitalCity;
  // Set of cities
  private Vector<City> cities;
}

public class City {
  // Attribute Data members
```

```
  private String cityCode;
  private String cityName;
  // Relationship Data Members
  // State city lelongs to
  private State state;
}
```

(a) Write a Java program to read data from a text file containing data about states and cities and populate the db4o database. The format of the text file is:

```
50
GA:Georgia:Peach State:6478216
Atlanta(ATL),Columbus(CLB),Savannah(SVN),Macon(MCN)
IL:Illinois:Prairie State:12128370
Springfield(SPI),Bloomington(BMI),Chicago(ORD),Peoria(PIA)
…
…
```

(b) Write a program in Java which implements the following menu-based system:

```
MAIN MENU

(1)    Given the first letter of a state, print all states along
   with their capital city names.
(2)    Print the top 5 populated states in descending order of
   population.
(3)    Given a state name, list all information about that state,
   including capital city, population, state nickname, major
   cities, etc. The report should be formatted nicely.
(4)    Print an alphabetical count of number of states for each
   letter of the English alphabet. The list should be nicely
   formatted; 4 letter counts to a   line. The count is the
   number of states whose names begin with the letter.
(5)    Quit
```

6. Convert the program of part (b) of the previous problem into a Web application with appropriate user interfaces.

CHAPTER 7

XML

XML (extensible Markup Language) is a data representation standard adopted by the World Wide Web Consortium (W3C) many years ago as the mechanism to share and exchange data on the Web. This chapter introduces the student to basic XML concepts including syntax, querying using XPath and XQuery, and schema specifications using XML Schema.

7.1 XML Basics

XML represents data in text format and encloses data items between user understandable and meaningful tags. For example, the address of a student is described as follows:

```
<address>123 Main Street, Atlanta, GA 30002</address>
```

The data item in this case is "`123 Main Street, Atlanta, GA 30002`" and is enclosed between the start tag `<address>` and the end tag `</address>`. The tag name "`address`" is user-defined and is descriptive of the data item that it is enclosing.

The entire string starting with the start tag, including the data item, and ending with the end tag is referred to as an XML *element*. XML elements can be nested as in the following example which includes sub-components of the address:

```
<address>
  <street>123 Main Street</street>
  <city>Atlanta</city>
  <state>GA</state>
  <zipcode>30002</zipcode>
</address>
```

Here, the `<address>` XML element has four sub-elements, `<street>`, `<city>`, `<state>`, and `<zipcode>`. Notice that all sub-elements are completely enclosed with the start and end tags of the main element.

Occasionally, there is a need to have an element without any contents. Such elements are called *empty* elements. For example, the following empty element may be used in a situation where the phone number is not known:

```
<phone></phone>
```

The above empty element can also be represented in the shorter notation:

```
<phone/>
```

There is no restriction in XML for repeating the sub-element tags. So, a list of phone numbers for an individual can be described as follows:

```
<person>
  <name>John Smith</name>
  <phone>111-1234</phone>
  <phone>212-2121</phone>
</person>
```

In addition to the element syntactic structure, XML also provides an additional mechanism to describe data. XML *attributes* are name-value pairs that are introduced in the start tag of elements. The following is an example of attributes in use within an element description:

```
<cost currency=”USD”>25.20</cost>
```

Here, the cost of some item is being described. The cost (25.20) is being described using an XML element <cost>. The start tag includes a name-value pair `currency=”USD”` indicating that the currency of the cost is in US dollars. In contrast to XML elements, attribute names cannot be repeated within the same start-tag. So, the following will be an error:

```
<cost currency=”USD” currency=”INR”>25.20</cost>
```

XML syntax also allows mixing of elements and text outside of the element context. This is a left over feature from the document world in mark up languages are still used. For example, consider the following XM fragment code:

```
<person>
  <name>John</name>
  This is my cousin!
  <age>22</age>
</person>
```

In this description of a person, the text “`This is my cousin!`” appears outside of the context of any element of sub-element. Such annotations are usually ignored by XML parsers.
Comments in XML are introduced as follows:

```
<!--This is a comment -->
```

XML documents usually begin with a version statement as follows:

```
<?xml version=”1.0”>
```

The `CDATA` construct allows one to include special characters such as the < or > symbols as part of text. For example, to include XML tags as part of the content of elements, the `CDATA` construct can be used as follows:

```
<![CDATA[<age>22</age>]]>
```

7.2 Company Database in XML

One possible XML representation of the COMPANY database is discussed in this section. The overall structure of the XML document is as follows:

```
<?xml version="1.0">
<companyDB>
  <departments>
  …
  …
  </departments>
  <employees>
  …
  …
  </employees>
  <projects>
  …
  …
  </projects>
</companyDB>
```

The document contains three main sections, one each for the list of departments, employees, and projects. Each section describes the individual entities within the classes along with relationships with other entities.

The <departments> element contains one or more <department> elements that describe the individual department along with its relationships with other entities. The following XML code fragment shows the details for department 5:

```
<departments>
  …
  …
  <department dno="5">
    <dname>Research</dname>
    <locations>
      <location>Bellaire</location>
      <location>Sugarland</location>
      <location>Houston</location>
    </locations>
    <manager mssn="333445555">
      <startDate>22-MAY-1978</startDate>
    </manager>
    <employees essns="123456789 333445555 666884444 453453453"/>
    <projectsControlled pnos="1 2 3"/>
  </department>
  …
  …
</departments>
```

As can be seen, the department element contains an attribute dno and several sub-elements, each describing either a simple attribute or a relationship. The <dname> sub-element describes the department name, the <locations> sub-element encloses one or more <location> elements, each describing a department location. The <manager>, <employees>, and <projectsControlled> sub-elements describe relationships with other entities and all these are represented by XML attributes.

The <employees> element contains one or more <employee> elements that describe the individual employees along with its relationships with other entities. The following XML code fragment shows the details for employee 333445555:

```
<employees>
  …
  …
  <employee ssn="333445555" worksFor="5"
            supervisor="888665555" manages="5">
    <fname>Franklin</fname>
    <minit>T</minit>
    <lname>Wong</lname>
    <dob>08-DEC-45</dob>
    <address>638 Voss, Houston, TX</address>
    <sex>M</sex>
    <salary>40000</salary>
    <dependents>
      <dependent>
        <dependentName>Alice</dependentName>
        <sex>F</sex>
        <dob>05-APR-1976</dob>
        <relationship>Daughter</relationship>
      </dependent>
      <dependent>
        <dependentName>Theodore</dependentName>
        <sex>M</sex>
        <dob>25-OCT-1973</dob>
        <relationship>Son</relationship>
      </dependent>
      <dependent>
        <dependentName>Joy</dependentName>
        <sex>F</sex>
        <dob>03-MAY-1948</dob>
        <relationship>Spouse</relationship>
      </dependent>
    </dependents>
    <supervisees essns="123456789 666884444 453453453"/>
    <projects>
      <worksOn pno="2" hours="10.0"/>
      <worksOn pno="3" hours="10.0"/>
      <worksOn pno="10" hours="10.0"/>
      <worksOn pno="20" hours="10.0"/>
    </projects>
```

```
  </employee>
  …
  …
</employees>
```

The <projects> element contains one or more <project> elements that describe the individual projects along with its relationships with other entities. The following XML code fragment shows the details for project 1:

```
<projects>
  …
  …
  <project pnumber="1" controllingDepartment="5">
    <pname>ProductX</pname>
    <plocation>Bellaire</plocation>
    <workers>
      <worker essn="123456789">32.5</worker>
      <worker essn="453453453">20.0</worker>
    </workers>
  </project>
  …
  …
</projects>
```

We shall use the above XML description of the COMPANY database in subsequent sections to discuss XML querying.

7.3 XML Editor EditiX

A free version of an XML editor can be downloaded from http://free.editix.com/. This software is available for all platforms including PC, Mac, and Linux. The free version of the editix has built-in support for editing and validating XML documents against DTDs and XML Schemas and also provides for XPath and XQuery support. The initial window for editix is shown in Figure 7.1.

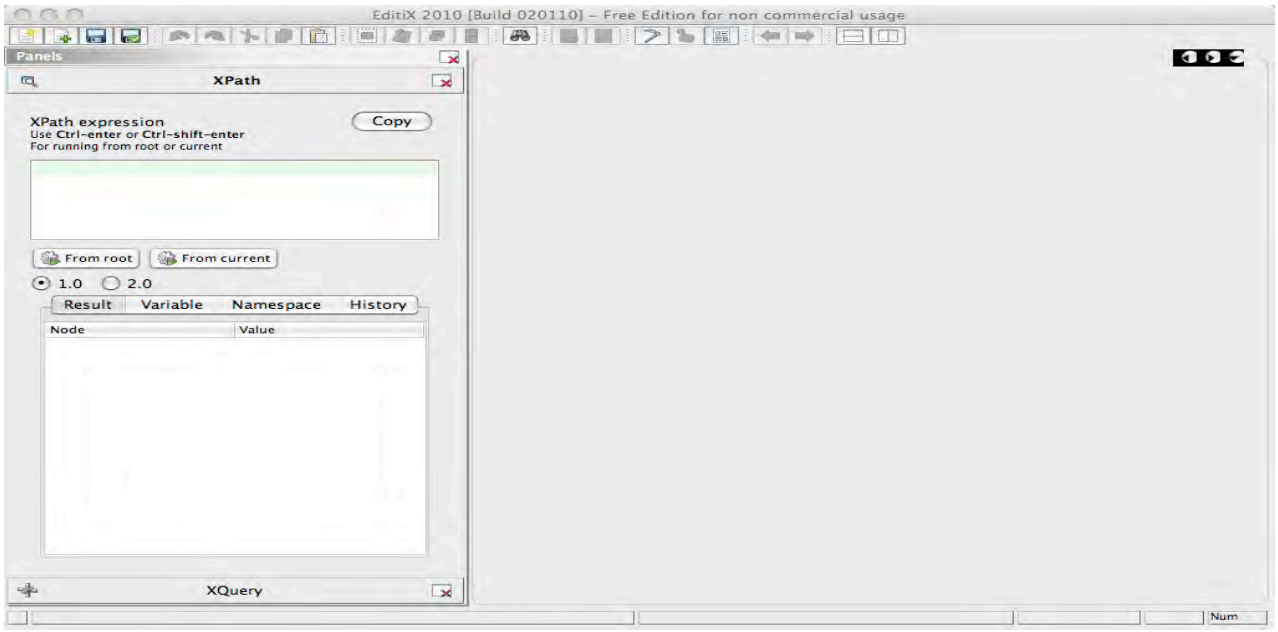

Figure 7.1 editix Window on startup

The left panel provides space to specify XPath as well as XQuery expressions to be evaluated against an XML document. The right panel is initially blank. This space is used to display the XML document. The File menu provides the ability to assign DTD or XML Schema files to the XML document as well as to validate XML documents against the schema files. Figure 7.2 shows the window after the `company.xml` document is opened in editx.

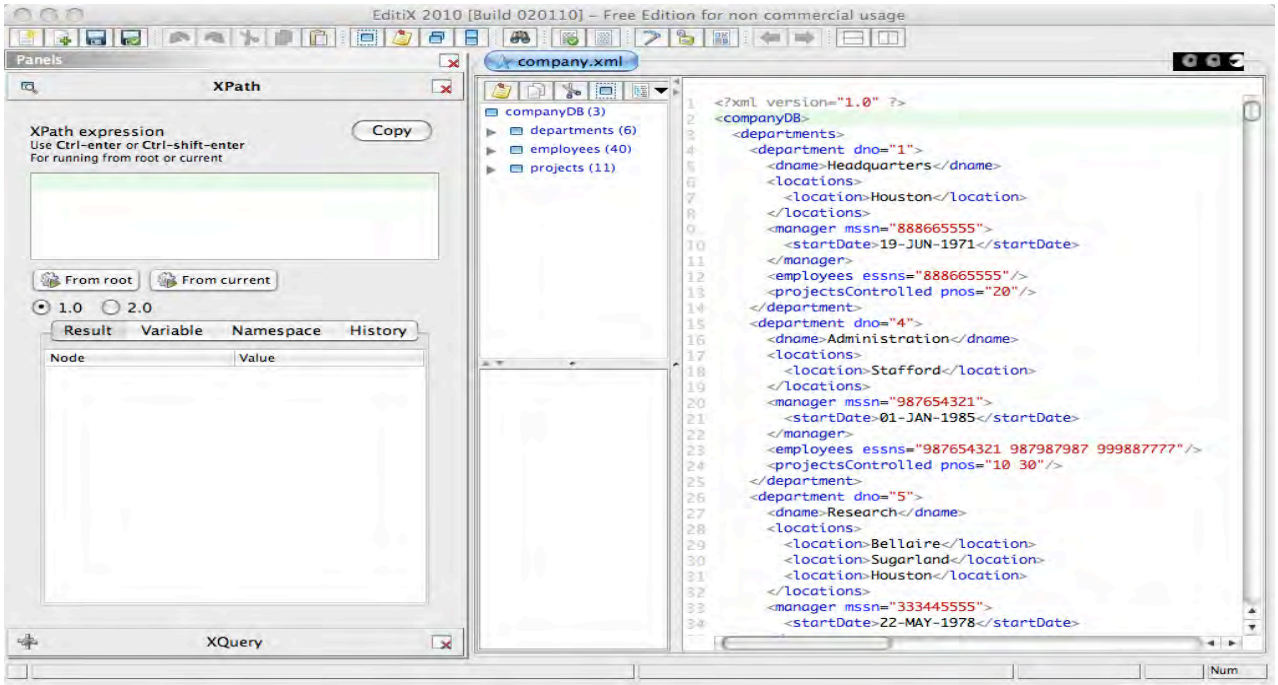

Figure 7.2 editix Window after opening XML document

The right panel contains two parts: a navigation pane that shows top-level elements along with counts of sub-elements and the display pane showing the entire XML document. The navigation pane can be used to go to specific portions of the XML file. The editor itself is quite straightforward to use.

7.4 XPath

XPath is an important specification language used to traverse and locate nodes in an XML document. It is not a full-fledged query language, but is used to specify a set of nodes in the XML tree structure, very much like the file specification in the directory hierarchy of a Unix operating system. XPath expressions are used in other query languages such as XQuery. The `company.xml` document will be used to illustrate all examples in the rest of the chapter.

There are several types of nodes in the XML tree: root node (e.g. `<companyDB>`), element node (e.g. `<salary>80000</salary>`), attribute node (e.g. `dno="4"`), and text node (e.g. `Stafford`). The nodes are related to each other in the XML tree via the parent, child, sibling, ancestor, and descendant relationships.

The latest XPath specification is available from the Web site http://www.w3.org/TR/xpath and the list of built-in functions for use in XPath and XQuery is available at http://www.w3.org/TR/xquery-operators/.

Basic XPath expressions

XPath expressions are of two types: absolute and relative. An absolute XPath expression begins with a `/` and is followed by a sequence of XML element names each separated by a `/`. For example, the expression:

```
/companyDB/departments/department
```

denotes the set of all `<department>` elements that are sub-elements of `<departments>` elements which in turn are sub-elements of the root element `<companyDB>` in the XML document. To execute XPath expressions on editix, simply enter the XPath expression in the XPath panel on the left and click “From root” button. This will bring up the matching nodes in the results box below. Upon clicking one of the results, the corresponding section in the XML tree display on the right is highlighted. The window after executing the above XPath expression is shown in Figure 7.3.

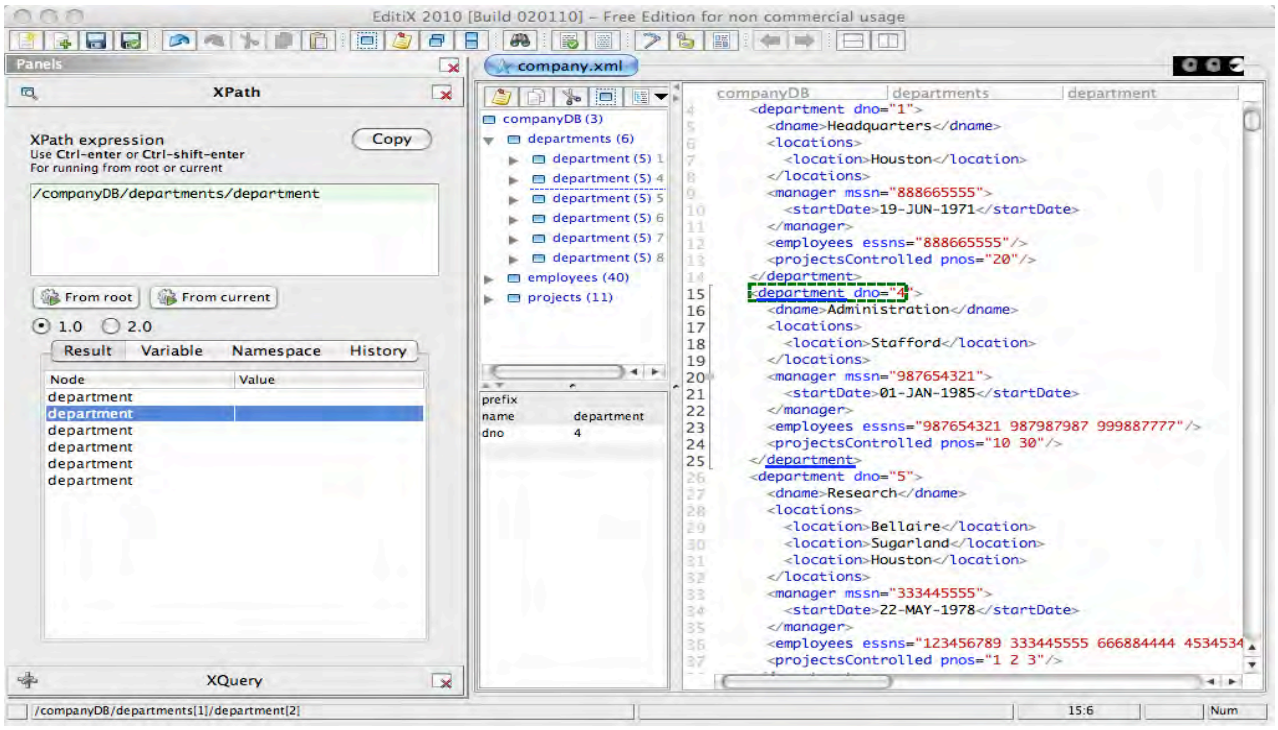

Figure 7.3 editix Window after XPath expression is executed

The expression / denotes the root node, . denotes the current node, and .. denotes the parent node of the current node. Expressions that do not begin with the / symbol are called relative XPath expressions and have a meaning only with respect to a current node. Attribute values are accessed using the @ expression. For example, the following XPath expression can be used to access the supervisor attribute of employee elements in the XML document:

```
/companyDB/employees/employee/@supervisor
```

Text nodes in the XML document can be accessed using the built-in function `text()`. For example, to access all the `lname` values for employees, the following XPath expression can be used:

```
/companyDB/employees/employee/lname/text()
```

Advanced XPath expressions

XPath allows the * wildcard to match any node in the path. For example, to select all children nodes of <employee> element, the following XPath expression can be used:

```
/companyDB/employees/employee/*
```

The wildcard can also be used to retrieve all attributes of an element. For example, to retrieve all the attributes of the <employee> element, the following XPath expression can be used:

```
/companyDB/employees/employee/@*
```

Another wildcard provided by XPath is `//`, the descendant-or-self wildcard. This allows access to nodes at any level in the tree. For example, to access all the <dob> elements in the XML document regardless of the level it appears, the following XPath expressions can be used:

```
//dob
```

XPath allows access to the children nodes based on their positions. For example, the 7th <employee> sub-element of <employees> element can be accessed using the XPath expression:

```
/companyDB/employees/employee[7]
```

To access the second dependent of the first employee, the following XPath expression can be used:

```
/companyDB/employees/employee[1]/dependents/dependent[2]
```

Using the built-in function last(), the last dependent of the first employee can be accessed using the XPath expression:

```
/companyDB/employees/employee[1]/dependents/dependent[last()]
```

The [] notation used to get positional access to the children of a node can be used to express general predicates as well. A logical predicate involving the attributes and child node values of the current node can be specified along with any built-in functions. Several examples of the use of general predicate are presented next.

The `position()` built-in function can be used within the predicate to access the first three <employee> elements in the XML document as follows:

```
/companyDB/employees/employee[position()<=3]
```

To access all <employee> elements that have a <minit> sub-element with a value of "E", the following XPath expression can be used:

```
/companyDB/employees/employee[minit="E"]
```

To access all <project> elements that have a `controllingDepartment` attribute value greater than 6, the following XPath expression can be used:

```
/companyDB/projects/project[@controllingDepartment>6]
```

The `starts-with()` and `contains()` built-in functions work on string values and can be useful to access elements based on substring matches. To access all <employee> elements whose last name starts with "S", the following XPath expression can be used:

```
/companyDB/employees/employee[starts-with(lname,"S")]
```

To access all <employee> elements who address contains the sub-string "Philadelphia", the following XPath expression can be used:

```
/companyDB/employees/employee[contains(address,"Philadelphia")]
```

Complex search conditions can be specified using logical connectives "and", "or" and "not". For example, to access all <employee> elements who work for department 7 and who are males and who have a male dependent, the following XPath expression can be used:

```
//employee[@worksFor=7 and sex="M" and dependents/dependent[sex="M"]]
```

XPath allows a path expression to be used in place of a predicate with the [] notation. The node is selected if the expression within the square brackets evaluates to a non-empty set if nodes. For example, the XPath expression:

```
/companyDB/employees/employee[dependents]
```

returns all employee nodes that have dependents.

XPath also provides support for the "union" of two sub-expressions using the "|" operator. For example, to access all project names and department names, the following XPath expression can be used:

```
/companyDB/departments/department/dname/text() |
/companyDB/projects/project/pname/text()
```

The general form of an XPath expression is one of the following:

`/locationStep1/locationStep2/…` for absolute path expressions

or

`locationStep1/locationStep2/…` for relative path expressions.

In either case, the location step is of the form:

```
axis::nodeSelector[selectionPredicate]
```

where axis is one of the following: `child`, `parent`, `descendant`, `ancestor`, `descendant-or-self` (`//`), or `ancestor-or-self`. The default axis is `child`. The `nodeSelector` is either the name of an element or an attribute or a wild card symbol. The `selectionPredicate` is a predicate as discussed in various examples previously. An example of an XPath expression in which the axis is explicitly mentioned is:

```
/companyDB/employees/employee[@worksFor=ancestor::companyDB/depart
ments/department[dname="Administration"]/@dno]
```

This expression accesses all employee nodes of employees who work for the "`Administration`" department. The use of the `ancestor` axis enables traversing up the XML tree to look for a particular department.

7.5 XQuery

XQuery is the official W3C (World Wide Web Consortium) standard query language for XML data. It is a high-level functional language for formulating ad hoc queries on XML data. The latest XQuery specifications are available at http://www.w3.org/TR/xquery.

Editix provides support for XQuery execution in its user interface. Figure 7.4 shows the editx window with the XQuery tab opened on the left panel.

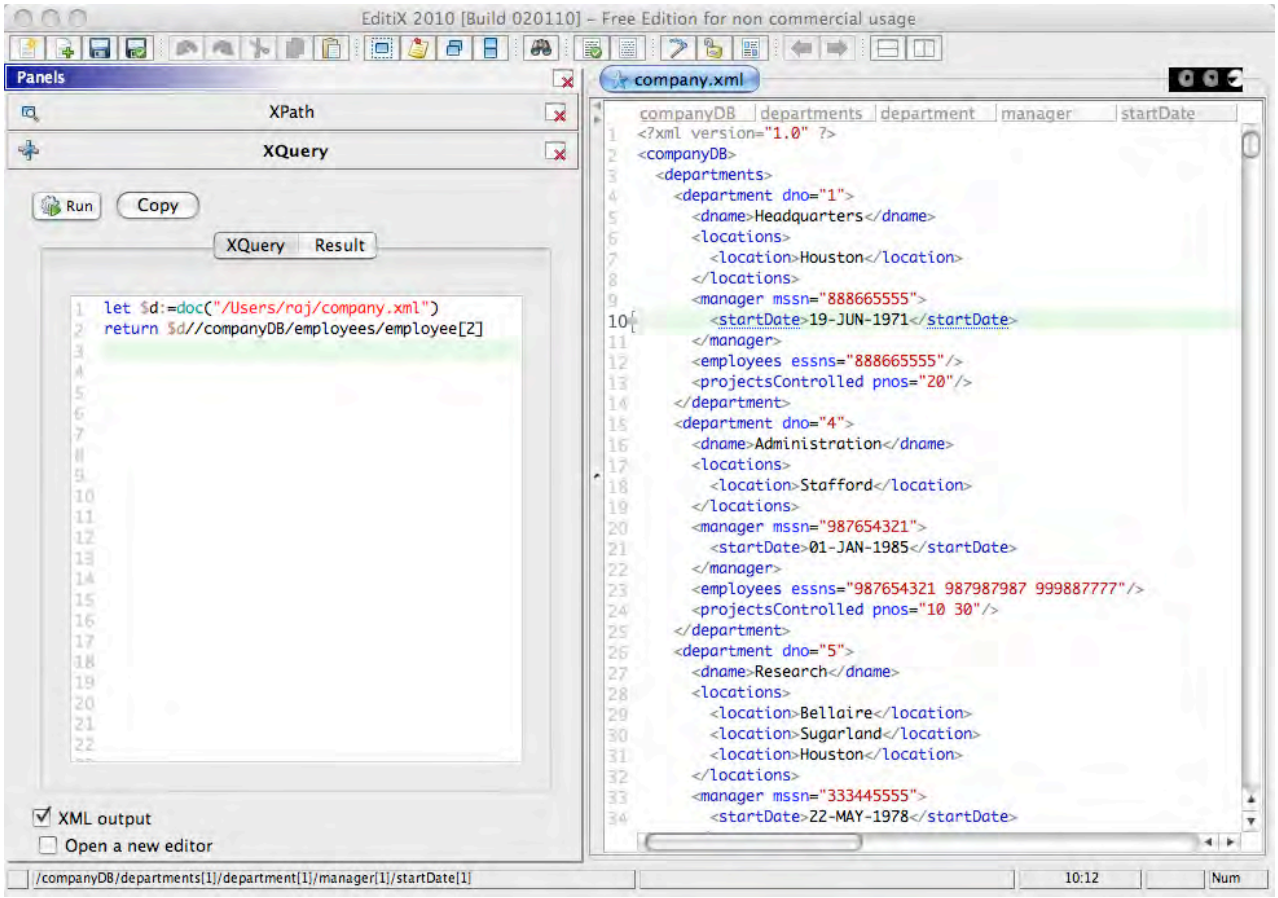

Figure 7.4 editix Window with XQuery tab in left panel

The user first enters the query in the text box and then clicks the "Run" button. In the figure the following query is shown:

```
let $d:=doc("/Users/raj/company.xml")
return $d//companyDB/employees/employee[2]
```

The results are displayed under the "Result" tab. To see the results the user needs to click the "Results" tab. A screenshot of the results tab is shown in Figure 7.5.

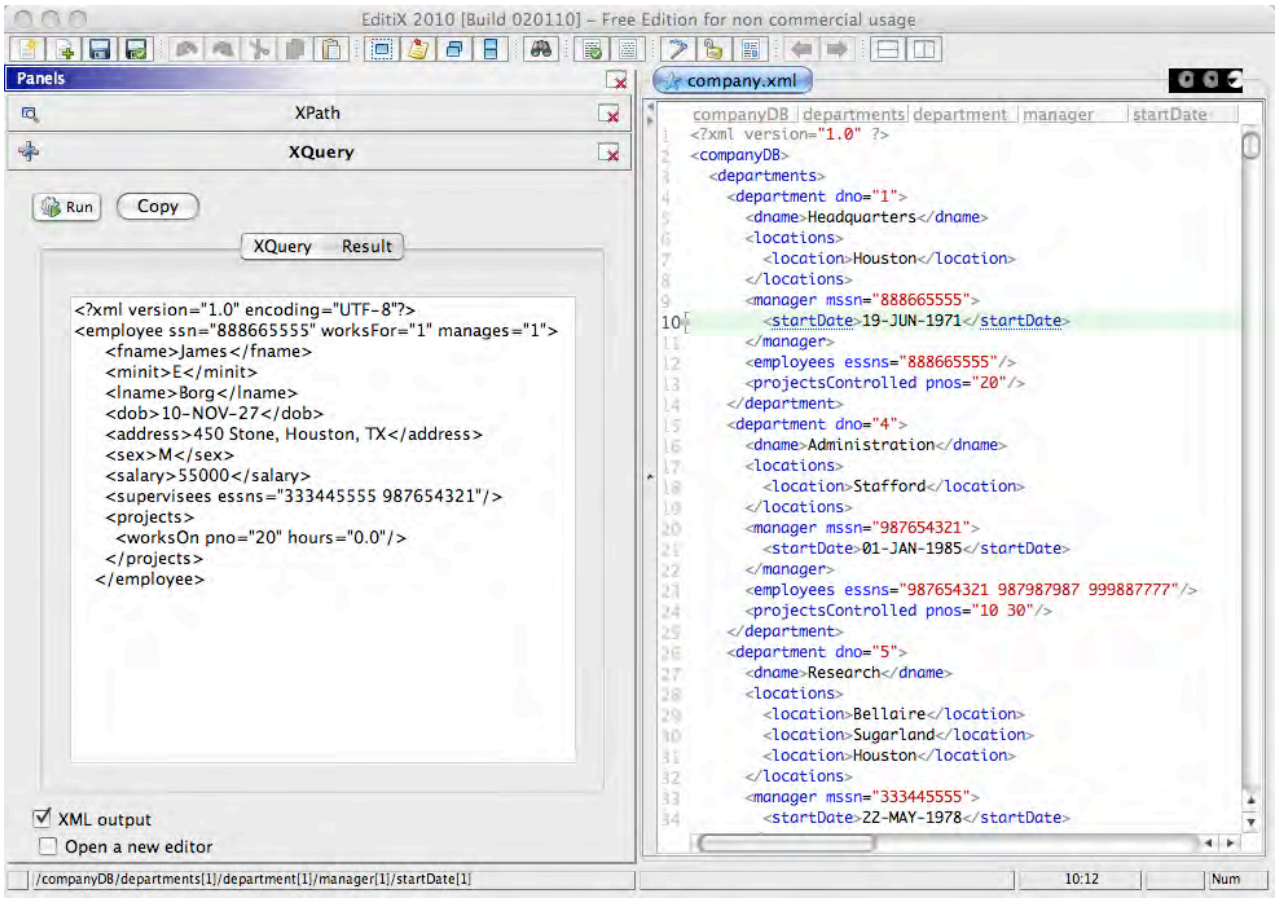

Figure 7.5 editix Window with query results in left panel

Note that the XML file is displayed on the right panel, but unlike XPath, the results are not highlighted there. This is because XQuery is a general query language and the results are constructed from the XML document but may not have the same structure as the XML document.

Simple XQuery Expressions

XQuery is a high-level functional language that evaluates expressions of different kinds. Simple arithmetic expressions such as:

```
(2*3)-(8*7)
```

are evaluated by XQuery. Built-in functions can be invoked as shown in the following three examples:

```
concat("Hello"," World")
matches("Monday","^.*da.*$")
current-time()
```

The `concat()` function takes in one or more string arguments and returns the concatenation of all strings. The `matches()` function takes a string input and a second string input representing a regular expression. It returns `true` if the first string matches the regular expression. The `current-time()` function returns the current time of day. All XPath/XQuery built-in functions are documented at http://www.w3.org/TR/xquery-operators/.

XQuery provides the `let` clause to assign values to variables and the `return` clause to construct the output and return the value of the output. Lists in XQuery are flat objects and are constructed by surrounding comma-separated values by parentheses. The following example illustrates list values, the `let` and the `return` clauses, and list aggregate operations:

```
let $list:=(1,5,10,12,15)
return count($list)
```

The above example returns the number of elements in the list. Other list aggregate operations include `avg`, `sum`, `max`, and `min`.

Value comparison operators for primitive data types are: `eq, ne, lt, gt, ge, and le`; general objects (including the primitive types) such as lists etc. can be compared using `<, <=, =, !=, >, >=`. XML nodes comparisons are done with one of the three operators: `<<, >>,` and `is`. The "is" operator compares for exact identity. The document order of nodes is verified by << (appears before) and >> (appears after).

XPath expressions can be directly introduced in the return clause of queries as follows:

```
let $d:=doc("/Users/raj/company.xml")
return $d//companyDB/employees/employee[2]
```

In fact, XPath expressions form the building clocks for constructing XQuery queries and can be used in several clauses as will be discussed later in this section.

Raw XML content also are treated as expressions and simply printed to the output by XQuery. For example, the following expression is simply sent to output when evaluated by XQuery:

```
<item ino="222"><iname>Nut</iname><price>22.50</price></item>
```

Curly braces can enclose sub-expressions in XQuery that need to be evaluated before sending to output. The following example employs the curly braces around an expression to print the salaries of employees who work for department number `6`:

```
let $d:=doc("/Users/raj/company.xml")//employee[@worksFor=6]
return
  <dept6Salary>{$d/salary}
  </dept6Salary>
```

FLWOR Expressions

The most important and powerful construct in XQuery is the FLWOR expression. FLWOR, pronounced "flower" stands for `for`, `let`, `where`, `order by`, and `return`, the individual clauses allowed in a query. A FLWOR expression starts with one or more `for` or `let` clauses, followed by an optional `where` clause, followed by an optional `order by` clause, and ending with a `return` clause. FLWOR expressions are illustrated via a series of queries on the `company.xml` document.

Query 1: Get all projects.

```
let $d:=doc("/Users/raj/company.xml")
for $p in $d/companyDB/projects/project
return $p
```

This query uses the `let` expressions to assign the root element of the `company.xml` document to the variable `$d`. Then, in the `for`-clause, the query iterates through all the nodes corresponding to the XPath expression `$d/companyDB/projects/project`. In the `return`-clause the value of the iterator variable `$p` is returned as the result of the query. The result will be a forest of all `<project>` elements.

The next query uses the distinct-values function to remove duplicates:

Query 2: Get distinct project numbers of projects in which employees work.

```
<projects>
{
let $d:=doc("/Users/raj/company.xml")
for $p in distinct-values(
        $d/companyDB/employees/employee/projects/worksOn/@pno)
return
<project>{$p}</project>
}
</projects>
```

In this query, the overall expression is a XML construct which includes FLWOR query within curly braces. The `for`-clause uses the `distinct-values()` function to eliminate duplicates in project numbers found within the `projects/worksOn` sub-element of the `employee` element. The query results for this query are shown in Figure 7.6.

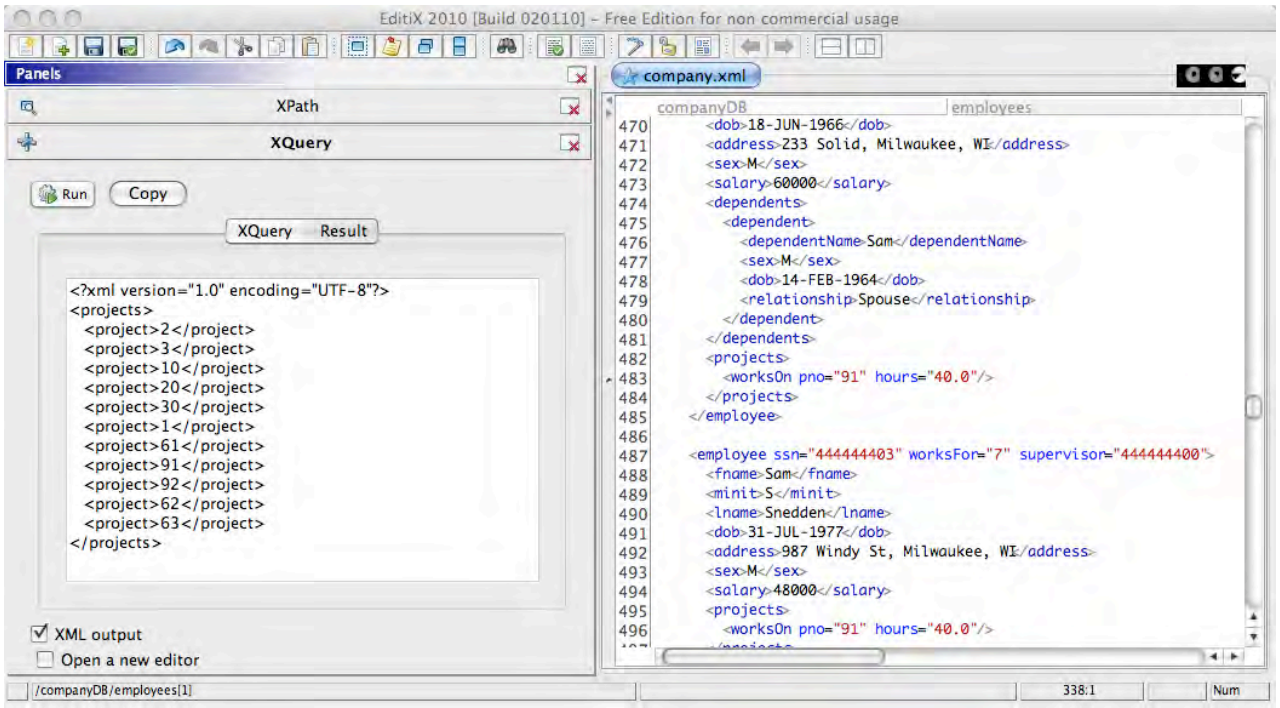

Figure 7.6 editix Window with Query 2 results in left panel

Notice that the project numbers are not ordered. The order by clause allows results to be ordered. By adding the order by clause in Query 2 as shown below, the results can be ordered.

```
<projects>
{
let $d:=doc("/Users/raj/company.xml")
for $p in distinct-values(
            $d/companyDB/employees/employee/projects/worksOn/@pno)
order by number($p)
return
<project>{$p}
</project>
}
</projects>
```

Note the use of the `number()` function in the order by clause. This is necessary because by default the order by clause treats the list as strings and as such the number 100 will appear before 99. The `number()` function converts the string to a number, thereby allowing the ordering to be done on a numeric basis.

Query 3: Get social security numbers of employees whose last name starts with "S".

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee
```

```
where starts-with($e/lname,"S")
return <sssn>{$e/@ssn}</sssn>
```

This query calls the built-in function `starts-with()` to determine if the last name of the employee begins with "`S`" and employs the `where`-clause to filter the results as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<sssn ssn="123456789"/>
<sssn ssn="444444403"/>
<sssn ssn="666666607"/>
```

Query 4: Get last names and first names of employees in the "Research" department.

```
let $d:=doc("/Users/raj/company.xml")
let $r:=$d/companyDB/departments/department[dname="Research"]
for $e in $d/companyDB/employees/employee
where $e/@worksFor=$r/@dno
return
<ResearchEmp>{$e/lname}{$e/fname}</ResearchEmp>
```

This query implements a "join" operation. The query assigns the "`Research`" department object to the variable `$r`. The, in the `for`-clause, the variable `$e` iterates over all employees. The `where`-clause selects only those employees who work for the department denoted by `$r`. The equality comparison works fine since there is only one department with name "Research", i.e. the value for `$r` is a singleton set. The `return`-clause constructs the answer to the query as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ResearchEmp>
   <lname>Wong</lname>
   <fname>Franklin</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>Smith</lname>
   <fname>John</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>Narayan</lname>
   <fname>Ramesh</fname>
</ResearchEmp>
<ResearchEmp>
   <lname>English</lname>
   <fname>Joyce</fname>
</ResearchEmp>
```

Query 5: Get employees who work more than 40 hours.

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee
where sum($e/projects/worksOn/@hours)>40.0
return
<OverWorkedEmp>{$e/lname}
{$e/fname}<TotalHours>{sum($e/projects/worksOn/@hours)}
</TotalHours>
</OverWorkedEmp>
```

This query illustrates the aggregate operation, `sum`. The for-clause introduces an iterator $e over all employees and the where-clause sums the hours worked on various projects by the employee and check to see if it exceeds 40. The return-clause constructs the results as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<OverWorkedEmp>
   <lname>Zell</lname>
   <fname>Josh</fname>
   <TotalHours>48</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Chase</lname>
   <fname>Jeff</fname>
   <TotalHours>46</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Ball</lname>
   <fname>Nandita</fname>
   <TotalHours>44</TotalHours>
</OverWorkedEmp>
<OverWorkedEmp>
   <lname>Bacher</lname>
   <fname>Red</fname>
   <TotalHours>50</TotalHours>
</OverWorkedEmp>
```

Query 6: Get department names and the total number of employees working in the department.

```
let $d:=doc("/Users/raj/company.xml")
for $r in $d/companyDB/departments/department
return
<deptNumEmps>{$r/dname}
<numEmps>{count(tokenize($r/employees/@essns,"\s+"))}
</numEmps>
</deptNumEmps>
```

This example is similar to the previous example, but it illustrates how to tokenize a string of social security numbers separated by a space. The `tokenize()` function takes as its first argument the string of social security numbers of employees working for a particular department and a regular expression denoting the separator. In this case, the regular expression is "`s\+`" indicating one or more spaces. Rest of the query is similar to the previous one. The results are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<deptNumEmps>
   <dname>Headquarters</dname>
   <numEmps>1</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Administration</dname>
   <numEmps>3</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Research</dname>
   <numEmps>4</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Software</dname>
   <numEmps>8</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Hardware</dname>
   <numEmps>10</numEmps>
</deptNumEmps>
<deptNumEmps>
   <dname>Sales</dname>
   <numEmps>14</numEmps>
</deptNumEmps>
```

Query 7: Get last names of employees who work for a project located in "Houston".

```
<empsWorkingOnHoustonProjects>
{
distinct-values(
let $d:=doc("/Users/raj/company.xml")
for $r in $d/companyDB/projects/project[plocation="Houston"]
return
 for $e in $d/companyDB/employees/employee
 where exists(index-of($r/workers/worker/@essn,$e/@ssn))
 return $e/lname
)
}
</empsWorkingOnHoustonProjects>
```

This query illustrates nesting of FLWOR expressions as well as additional built-in functions. The outer FLWOR expression sets up an iterator `$r` over all "`Houston`" projects. The nested FLWOR expression is present in the `return`-clause and iterates over all `employee` nodes. The `where`-clause checks to see if the social security number of the employee matches one of the worker social security numbers of the "`Houston`" project. The `index-of()` function returns a list of indices where the second argument (search item) appears in the first argument (list to be searched). The `exists()` function returns `true` if its input is non-empty. The results of executing the query are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<empsWorkingOnHoustonProjects>
Wong Narayan Borg Wallace
</empsWorkingOnHoustonProjects
```

Query 8: Get last names of employees with dependents.

```
let $d:=doc("/Users/raj/company.xml")
for $e in $d/companyDB/employees/employee[dependents]
return $e/lname
```

This query illustrates the use of an XPath-expression (`[dependents]`) as a predicate; employee nodes that have sub-element `<dependents>` are selected and the last names of such employees are returned as the answer to the query.

Query 9: Get last names of employees without dependents.

```
let $d:=doc("/Users/raj/company.xml")
let $empsWithDeps := $d/companyDB/employees/employee[dependents]
for $e in $d/companyDB/employees/employee
where empty(index-of($empsWithDeps,$e))
return $e/lname
```

This query defined a variable `$empsWithDeps` that holds all employee nodes with dependents. Then, the `for`-clause iterates over all employees and selects only those employees who do not appear in the list `$empsWithDeps`.

Query 10: get last names of employees from Milwaukee along with their income group: "Low Income Group" (earning < 40000), "Middle Income Group" (earning between 40000 and 60000), and "High Income Group" (earning more than 80000).

```
<IncomeGroup>
{
let $d:=doc("E:/company.xml")
for $e in
  $d/companyDB/employees/employee[contains(address,"Milwaukee")]
return
```

```
<emp>{$e/lname}
<income>
{if  ($e/salary >= 80000) then "High Income"
else if ($e/salary >=60000) then "Middle Income"
else "Low Income"
}
</income>
</emp>
}
</IncomeGroup>
```

This query illustrates the use of conditional expressions in FLWOR queries. The `for`-clause sets up an iterator on all "`Milwaukee`" employee nodes. The `return`-clause constructs the output XML using a conditional expression. The output to the query looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<IncomeGroup>
  <emp>
    <lname>Freed</lname>
    <income>High Income</income>
  </emp>
…
…
</IncomeGroup>
```

Query 11: Get employee names of employees who work on all projects located in "Houston".

```
let $d:=doc("/Users/raj/company.xml")
let $houstonProjs :=
 $d/companyDB/projects/project[plocation="Houston"]
for $e in $d/companyDB/employees/employee
where every $p in $houstonProjs satisfies
  (some $q in $e/projects/worksOn satisfies
    $p/@pnumber = $q/@pno)
return concat($e/fname,", ",$e/lname)
```

This query illustrates the use of the "`every`" and "`some`" quantifier constructs in XQuery. The query first computes the list of all projects located in "`Houston`" in the variable `$houstonProjs`. The `for`-clause sets up an iterator on `employee` nodes. The `where`-clause employs the quantifier constructs to verify if every "`Houston`" project is present in the `projects/worksOn` sub-element of the `employee` node. The results are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
Franklin, Wong
```

The general syntax of the quantifier expressions is:

```
some $var1 in $expr1, …, $varN in $exprN
satisfies $boolExpr

every $var1 in $expr1, …, $varN in $exprN
satisfies $boolExpr
```

The "some" expression evaluates to true if at least one assignment of values to the variables result in the Boolean expression being evaluated to true.

The "every" expression evaluates to true if all assignment of values to the variables result in the Boolean expression being evaluated to true.

7.6 XML Schema

XML Schema is a schema language for XML documents with a rich set of primitive data types as well as an extensive set of type constructs. The syntax of XML Schema is XML itself, i.e. XML Schemas are themselves well-formed and valid XML documents. The structure of XML documents is defined in XML Schema by the constructing a type system of simple and complex types that describe the elements and sub-elements of the document.

The schema language features are introduced in this section by considering the `company.xml` document introduced earlier in this chapter and creating a schema for it.

Primitive Types

XML Schema provides a host of primitive types including `xs:string`, `xs:integer`, `xs:decimal`, `xs:boolean`, and `xs:date`. Simple elements in the XML document can be defined in the schema as having one of these primitive types. For example,

```
<xs:element name=dname" type="xs:string"/>
```

defines the structure for XML element:

```
<dname>Research</dname>
```

Simple Types

Simple types are used for elements that do not have attributes and that do not have sub-elements. They are also used for attributes. Starting with the basic primitive types, XML Schema provides several constructs to impose restrictions on the values that a particular type can allow from the domain of the primitive types.

Consider the XML element `<dno>6</dno>` and the restriction that the department number be in the range 1 through 50. The XML Schema code that describes the type of the element is shown below:

```
<xs:simpleType name="dnoType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="50"/>
  </xs:restriction>
</xs:simpleType>
```

Here, the simple type `dnoType` is being defined as a restriction on the base type `xs:integer` with the minimum value being `1` and the maximum value being `50`. The <dno> element is then described in the schema as:

```
<xs:element name=dno" type="dnoType"/>
```

The social security number is restricted to be a 9-digit number and the simple type for it is shown below:

```
<xs:simpleType name="ssnType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{9}"/>
  </xs:restriction>
</xs:simpleType>
```

Here, the base type is `xs:string` and the restriction is based on a regular expression pattern which indicates that there should be exactly `9` digits. The regular expressions that describe the pattern are very similar to that of Unix regular expressions. Detailed descriptions of the various features of XML Schema including that of the regular expressions can be found at the W3C website: http://www.w3.org/XML/Schema.html

The number of hours per week an employee of the company may work for a project is a decimal number with two digits after the decimal point and occupying a total of 5 spaces. The type is defined as follows:

```
<xs:simpleType name="hoursType">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="5"/>
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Enumerated types are possible in XML Schema where the values are chosen from a given set. For example, the `genderType` used in the company example for employees as well as dependents can be defined as follows:

```
<xs:simpleType name="genderType">
   <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="M"/>
      <xs:enumeration value="F"/>
   </xs:restriction>
</xs:simpleType>
```

Lists of Simple Types

XML Schema provides rich support for creating lists of simple types and enforcing several constraints in lists. The following is a list type definition for a list of social security numbers:

```
<xs:simpleType name="listOfSSNType">
  <xs:list itemType="ssnType"/>
</xs:simpleType>
```

Restrictions on lists can be expressed using the facets (a fancy name for restrictions!) `xs:length`, `xs:minLength`, and `xs:maxLength`. For example, to define lists of social security numbers of length `8`, the following code can be used:

```
<xs:simpleType name="eightListOfSSNType">
  <xs:restriction base="listOfSSNType">
    <xs:length value=”8”/>
  </xs:restriction>
</xs:simpleType>
```

Complex Types

Elements that have attributes or those that have sub-elements are said to have complex types and XML Schema provides the ability to compose complex types from simple types. For example, the following element found within the `<employee>` element describes a dependent:

```
<dependent>
  <dependentName>Abner</dependentName>
  <sex>M</sex>
  <dob>29-FEB-1932</dob>
  <relationship>Spouse</relationship>
</dependent>
```

The complex type, `dependentType`, shown below describes the structure of the `<dependent>` element:

```
<xs:complexType name="dependentType">
  <xs:sequence>
    <xs:element name="dependentName" type="xs:string"/>
    <xs:element name="sex" type="genderType"/>
    <xs:element name="dob" type="xs:string"/>
    <xs:element name="relationship" type="xs:string"/>
```

```
  </xs:sequence>
</xs:complexType>
```

The complex type describes the structure as containing four sub-elements in a particular order. The <xs:sequence> construct specifies that the order of the sub-element is as specified in the type definition. To define the <dependents> element that includes one or more <dependent> elements, another complex type can be defined as follows:

```
<xs:complexType name="dependentsType">
  <xs:sequence>
    <xs:element name="dependent" type="dependentType"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Note the minOccurs and maxOccurs attributes that specify that there can be one or more occurrences of the <dependent> sub-elements within the <dependents> element.

A similar type definition is made for the <locations> element within the <department> element as follows:

```
<xs:complexType name="locationsType">
  <xs:sequence>
    <xs:element name="location" type="xs:string"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

A sample XML fragment that conforms to the above type definition is:

```
<locations>
  <location>Atlanta</location>
  <location>Sacramento</location>
</locations>
```

Consider the following XML fragment from the company XML file:

```
<manager mssn="111111100">
  <startDate>15-MAY-1999</startDate>
</manager>
```

This describes a manager and includes an attribute as well as a sub-element. A complex type to describe such a structure is shown below:

```
<xs:complexType name="managerType">
  <xs:sequence>
```

```
    <xs:element name="startDate" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="mssn" type="ssnType"/>
</xs:complexType>
```

Attributes are described after all the sub-elements are described. In this case there is only one sub-element, `startDate`.

Elements that have attributes and have empty content (i.e. no sub-elements) are also classified under complex types. For example, consider the following element in the `<department>` element.

```
<projectsControlled pnos="61 62 63"/>
```

The complex type definition for this element is shown below:

```
<xs:complexType name="projsControlType">
  <xs:attribute name="pnos" type="listOfPnoType"/>
</xs:complexType>
```

Elements with Simple Content and Attributes

Describing the structure of an element with simple content and with attributes is done by using the `<xs:simpleContent>` construct in XML Schema. Such an element appears in the company XML document as follows:

```
<worker essn="555555500">40.0</worker>
```

The complex type to describe this element is shown below:

```
<xs:complexType name="workerType">
  <xs:simpleContent>
    <xs:extension base="hoursType">
      <xs:attribute name="essn" type="ssnType"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Within the `<simpleContent>` construct, the `hoursType` type is extended by adding an attribute. Now, to enclose one or more `<worker>` elements within the `<workers>` element such as:

```
<workers>
  <worker essn="555555500">40.0</worker>
  <worker essn="555555501">44.0</worker>
  <worker essn="666666605">40.0</worker>
```

```
</workers>
```

the following complex type is defined:

```
<xs:complexType name="workersType">
  <xs:sequence>
    <xs:element name="worker" type="workerType"
                minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Complete XML Schema File

The complete XML Schema file for the company document is constructed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  definitions of all simple and complex types
  …
  …
  <xs:element name="companyDB" type="companyDBType"/>
</xs:schema>
```

The file starts with the XML version statement followed by the `<xs:schema>` element. Within this element, all simple and complex types are defined. Finally the root element `<companyDB>` is defined. The entire schema file (`company.xsd`) along with the XML document file (`company.xml`) is available along with this lab manual.

Exercises

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

1. Write and execute XQuery expressions for the following queries on the company.xml document:
    a. Retrieve the names and addresses of employees who work for the "Research" department.
    b. For every project located in "Stafford", retrieve the project number, the controlling department number, and the department's manager's last name, address, and birth date.
    c. Retrieve the names of all employees who have two or more dependents.
    d. Retrieve the names of managers who have at least one dependent.
    e. Retrieve the names of employees who work on all projects controlled by department "5".
2. Write and execute XQuery expressions for all queries of laboratory exercise 6.34 on page 192 of the Elmasri/Navathe textbook (6^{th} edition).

3. Consider the mail order database described in problem 2 of the Exercises in Chapter 1 of this lab manual.
    a. Create a XML representation of the data described there (you should invent your own data instances).
    b. Write a XML Schema specification for the XML document constructed in part a.
    c. Write expressions in XQuery to answer all queries of problem 2 of the Exercises in Chapter 2 of this lab manual.
4. Consider the grade book database described in problem 6 of the Exercises in Chapter 1 of this lab manual.
    a. Create a XML representation of the data described there (you should invent your own data instances).
    b. Write a XML Schema specification for the XML document constructed in part a.
    c. Write expressions in XQuery to answer all queries of problem 3 of the Exercises in Chapter 2 of this lab manual.
5. Consider the bibliography XML document, bib.xml, provided along with this lab manual.
    a. Write a XML Schema specification for the XML document.
    b. Write expressions in XQuery to answer the following queries:
        i. Find articles that have "Temporal" in their titles.
        ii. Find all articles authored by "Raghu Ramakrishnan".
        iii. Find number of articles with more than 3 authors.
        iv. Produce a listing of URLs of articles for each author. Output should consist of author names followed by list of URLS, sorted by author names.
        v. Find all articles that are over 40 pages long.

CHAPTER 8

Projects

This chapter presents several projects to be completed by the student using Java/JDBC and Oracle database or using PHP and MySQL database. These projects may be assigned as group projects with teams of two or three students. A written documentation as well as a class presentation of the project may be required.

8.1 STUDENT REGISTRATION System (GoLunar)

Consider the following relational database and sample data for the student registration database (written in Oracle SQL):

```
drop table students cascade constraints;
create table students (
  sid      number(4) primary key,
  password number(5),
  fname    varchar2(20),
  lname    varchar2(20),
  sType    varchar2(5) check (sType in ('GRAD','UGRAD')),
  major    char(4) check (major in ('CSC','MATH','POLS','HIST')),
  gradAssistant char(1) check (gradAssistant in ('Y','N')),
  inState char(1) check (inState in ('Y','N'))
);
insert into students values
  (1111,1111,'John','Davison','UGRAD','CSC','N','Y');
insert into students values
  (2222,2222,'Jacob','Oram','UGRAD','CSC','N','N');
insert into students values
  (3333,3333,'Ashish','Bagai','GRAD','CSC','Y','N');
insert into students values
  (4444,4444,'Joe','Harris','GRAD','CSC','N','Y');
insert into students values
  (5555,5555,'Andy','Blignaut','GRAD','CSC','N','Y');
insert into students values
  (6666,6666,'Pommie','Mbangwa','GRAD','CSC','N','Y');
insert into students values
  (7777,7777,'Ian','Healy','GRAD','CSC','N','Y');
insert into students values
  (8888,8888,'Dougie','Marillier','GRAD','CSC','N','Y');
--
drop table staff cascade constraints;
create table staff (
  tid  number(4) primary key,
  password number(5),
  fname varchar2(20),
  lname varchar2(20),
  staffType varchar2(10) check (staffType in ('REGISTRAR','DEPARTMENT'))
```

```
);
insert into staff values
  (1000,1000,'Venette','Rice','DEPARTMENT');
insert into staff values
  (2000,2000,'Alison','Payne','REGISTRAR');
--
create or replace view lunarUsers as
  (select sid uid, password, 'STUDENT' uType
   from   students) union
  (select tid uid, password, staffType uType
   from   staff);
--
drop table courses cascade constraints;
create table courses (
  cprefix  char(4),
  cno      number(4),
  ctitle   varchar2(50),
  chours   number(2),
  primary key (cprefix,cno)
);
insert into courses values ('CSC',1010,'Computers and Applications',3);
insert into courses values ('CSC',2010,'Introduction to Computer Science',3);
insert into courses values ('CSC',2310,'Intro to Programming in Java',3);
insert into courses values ('CSC',2311,'Introduction to Programming in C++',3);
insert into courses values ('CSC',3410,'Data Structures',3);
insert into courses values ('CSC',3210,'Computer Organization',3);
insert into courses values ('CSC',3320,'Systems Programming in Unix and C',3);
insert into courses values ('MATH',2211,'Calculus I',5);
insert into courses values ('MATH',2212,'Calculus II',5);
insert into courses values ('MATH',2420,'Discrete Mathematics',3);
insert into courses values ('CSC',6220,'Networks',4);
insert into courses values ('CSC',8220,'Advanced Networks',4);
insert into courses values ('CSC',6710,'Database',4);
insert into courses values ('CSC',8710,'Advanced Database',4);
insert into courses values ('CSC',6820,'Graphics',4);
insert into courses values ('CSC',8820,'Advanced Graphics',4);
insert into courses values ('POLS',1200,'Intro Political Sci',3);
--
drop table sections cascade constraints;
create table sections (
  term        char(2) check (term in ('FA','SP','SU')),
  year        number(4),
  crn         number(5),
  cprefix     char(4),
  cno         number(4),
  section     number(2),
  days        char(6),
  startTime   char(5), -- example 08.15, 13.30 etc.
  endTime     char(5),
  room        varchar2(10),
  cap         number(3),
  instructor  varchar2(30),
  auth        char(1) check (auth in ('Y','N')),
  primary key (term,year,crn),
  foreign key (cprefix,cno) references courses
);
--
```

```
insert into sections values
('SU',2002,10101,'CSC',1010,1,'MWF','09.00','09.50','105G',35,'Bhola','N');
insert into sections values
('SU',2002,10701,'POLS',1200,1,'TR','09.00','09.50','205Sp',25,'Jones','N');
--
insert into sections values
('FA',2002,10101,'CSC',2010,1,'MWF','09.00','09.50','105G',35,'Bhola','N');
insert into sections values
('FA',2002,10102,'CSC',2010,2,'MWF','10.00','10.50','105CS',40,'Henry','N');
insert into sections values
('FA',2002,10103,'CSC',2310,1,'MWF','12.00','12.50','106G',30,'Henry','N');
insert into sections values
('FA',2002,10104,'CSC',2311,1,'MWF','15.00','15.50','205G',35,'Liu','N');
insert into sections values
('FA',2002,10201,'CSC',6220,1,'TR','19.00','20.40','405G',25,'Hundewale','N');
insert into sections values
('FA',2002,10202,'CSC',6710,1,'TR','16.00','17.15','115CS',25,'Madiraju','N');
insert into sections values
('FA',2002,10203,'CSC',8820,1,'MWF','09.00','09.50','605G',25,'Owen','N');
insert into sections values
('FA',2002,10301,'MATH',2211,1,'TR','11.00','12.50','305G',35,'Li','N');
insert into sections values
('FA',2002,10302,'MATH',2211,2,'MWF','09.00','10.50','106GB',35,'Davis','N');
--
--This data will be loaded into the database in your application program
--insert into sections values
--('SP',2003,10101,'CSC',2010,1,'MWF','09.00','09.50','105G',35,'Bhola','N');
--insert into sections values
--('SP',2003,10102,'CSC',2010,2,'MWF','10.00','10.50','105CS',40,'Henry','N');
--insert into sections values
--('SP',2003,10103,'CSC',2310,1,'MWF','12.00','12.50','106G',30,'Henry','N');
--insert into sections values
--('SP',2003,10104,'CSC',2311,1,'MWF','15.00','15.50','205G',35,'Liu','N');
--insert into sections values
--('SP',2003,10201,'CSC',6220,1,'TR','19.00','20.40','405G',25,'Hundewale','N');
--insert into sections values
--('SP',2003,10202,'CSC',6710,1,'TR','16.00','17.15','115CS',25,'Madiraju','N');
--insert into sections values
--('SP',2003,10203,'CSC',8220,1,'MWF','09.00','09.50','605G',25,'Bourgeois','Y');
--insert into sections values
--('SP',2003,10301,'MATH',2211,1,'TR','11.00','12.50','305G',35,'Li','N');
--insert into sections values
--('SP',2003,10302,'MATH',2211,2,'MWF','09.00','10.50','606GB',35,'Miller','N');
--insert into sections values
--('SP',2003,10303,'MATH',2212,1,'MWF','09.00','10.50','706GB',35,'Davis','N');
--insert into sections values
--('SP',2003,10304,'MATH',2420,1,'TR','14.00','14.50','106GB',35,'Domke','N');
--insert into sections values
--('SP',2003,10405,'CSC',8710,1,'MW','17.30','18.45','206GB',35,'Dogdu','N');
--insert into sections values
--('SP',2003,10406,'CSC',8820,1,'TR','19.15','20.55','306GB',3,'Owen','N');
--
drop table enrolls cascade constraints;
create table enrolls (
  sid    number(4),
  term   char(2) check (term in ('FA','SP','SU')),
  year   number(4),
  crn    number(5),
  grade  char(2) check (grade in ('A','B','C','D','F','I','IP','S','U')),
  primary key (sid,term,year,crn),
```

```
  foreign key (sid) references students,
  foreign key (term,year,crn) references sections
);
--
insert into enrolls values (1111,'SU',2002,10101,'A');
insert into enrolls values (1111,'SU',2002,10701,'C');
--
insert into enrolls values (1111,'FA',2002,10101,null);
insert into enrolls values (1111,'FA',2002,10103,null);
insert into enrolls values (1111,'FA',2002,10301,null);
insert into enrolls values (3333,'FA',2002,10201,null);
insert into enrolls values (3333,'FA',2002,10202,null);
insert into enrolls values (3333,'FA',2002,10203,null);
--
drop table authorizations cascade constraints;
create table authorizations (
  term   char(2) check (term in ('FA','SP','SU')),
  year   number(4),
  crn    number(5),
  sid    number(4),
  authType char(4) check (authType in ('OVFL','AUTH')),
  primary key (term,year,crn,sid,authType),
  foreign key (sid) references students,
  foreign key (term,year,crn) references sections
);
--
drop table fixedFee cascade constraints;
create table fixedFee (
  feeName varchar2(30) primary key,
  fee number(5,2)
);
--
insert into fixedFee values ('Technology Fee',75.00);
insert into fixedFee values ('Health Fee',30.00);
insert into fixedFee values ('Activity Fee',65.00);
insert into fixedFee values ('Transportation Fee',25.00);
--
drop table variableFeeRate cascade constraints;
create table variableFeeRate (
  sType varchar2(6)
        check (sType in ('GRAD','UGRAD')),
  inOrOutOfState varchar2(10)
        check (inOrOutOfState in ('INSTATE','OUTOFSTATE')),
  fee number(6,2),
  primary key (sType,inOrOutOfState)
);
--
insert into variableFeeRate values ('GRAD','INSTATE',125.00);
insert into variableFeeRate values ('GRAD','OUTOFSTATE',500.00);
insert into variableFeeRate values ('UGRAD','INSTATE',100.00);
insert into variableFeeRate values ('UGRAD','OUTOFSTATE',400.00);
```

The database consists of the following tables:

1. `Students`: This table records information about students. The `gradAssistant` attribute records whether the student is a graduate assistant or not. The graduate assistants

automatically qualify for a full tuition waiver (they still have to pay the fixed fee). The `instate` attribute records whether the student is an in-state student or not. Again, this has an impact on the fees the student would pay.

2. `Staff`: This table records information about staff users of the system. There are two categories of staff: "`Registrar`" and "`Department`". These users would have different capabilities and functions in the application to be developed.

   Note: A view called `lunarUsers` is created to provide a simple way to authenticate users of the system.

3. `Courses`: This table records information about courses in the university catalog which includes course number, title and credit hours. The credit hours value will be used in calculating the GPA in the student's transcript.
4. `Sections`: This table records the course offerings for each term and includes the term, year, and course record number (`crn`), a unique number assigned to course offerings for a specific term and year. The table also includes start time and end time and meeting days as well as the name of the instructor. Finally, this table records a boolean value (yes or no) called `auth` to indicate if the registration for this course is open to all or is done only by authorization.
5. `Enrolls`: This table records information about which student has registered for which course offering.
6. `Authorizations`: This table records authorizations given to students for specific course offerings. Two types of authorizations are given: `OVFL` for overflow, i.e. allows students to register in a course offering that does not have any open seats, and `AUTH` for authorization to register in a course offering that is designated as a authorization only course offering.
7. `FixedFee`: This table records information about all fixed fees a student is required to pay each term they register.
8. `VariableFeeRate`: This table records per credit hour fee rate for different categories of students (graduate vs undergraduate students and in-state vs out-of-state students).

You will implement a University Registration System in Java using JDBC.

There are 3 kinds of database users:

1. **Registrar Staff:** These users will have the ability to load the database tables, make changes to courses, sections, fee details etc.
2. **Department Staff:** These users will have the ability to authorize students into sections, overflow students into sections, add assistantship information to the system, generate class lists etc.
3. **Student:** These users will be able to register for classes, see their schedules, see fee detail, see transcripts etc.

The following real-world constraints need to be enforced by your Java program:

1. Undergraduate students are not allowed to register for graduate courses numbered 6000 and above.
2. Students should not be allowed to register for a class which is FULL unless they have an overflow.
3. Students should not be allowed to register for a class which is listed as AUTHORIZATION ONLY unless they have an authorization.
4. Undergraduate students are not allowed to register for more than 20 hours in a semester and the limit for graduate students is 15.
5. Students cannot register for two classes that overlap in meeting time.

The Java application will be a terminal-based program that has the following interactions with the users. Based on the username, the program should determine the type of user and provide the appropriate menu.

Department Staff Menu:

```
$ java GoLunar OracleId
Oracle Password:xxxxxx
Semester (e.g. FA2003,SP2003,SU2003): SP2003
Username: 1000
Password:

*********************************************************************
***                                                               ***
***    Welcome to the GoLunar - Online Registration System        ***
***        Venette Rice - Department Staff                        ***
***                                                               ***
*********************************************************************

                         1. Authorize Student into Section

                         2. Overflow Student into Section

                         3. Add Assistantship on System

                         4. Generate Class List

                         q. Quit

Type in your option:
```

Option 1 Interface:

```
CRN:10101
SID:1111
Student John Davison authorized into CRN 10101, CSC 2010.
OR
No need to authorize - This section does not need authorization.
```

Option 2 Interface:

```
CRN:10101
SID:1111
No need to overflow - Space still available in this section.
OR
Student John Davison overflowed into CRN 10101, CSC 2010.
```

Option 3 Interface:

```
Student Id: 3333
Ashish Bagai (3333) has been added to the Assistantship List.
```

Option 4 Interface:

```
CRN: 10101

CSC 2010, Introduction to Computer Science
SP 2003
Instructor: Bhola

SID           LNAME           FNAME
-------------------------------------------
1111          Davison         John
2222          Oram            Jacob
...
...
```

Student Menu:

```
$ java GoLunar OracleId
Oracle Password:xxxxxx
Semester (e.g. FA2003,SP2003,SU2003): SP2003
Username: 1111
Password:
*********************************************************************
***                                                               ***
***    Welcome to the GoLunar - Online Registration System        ***
***        John Davison - Student                                 ***
***                                                               ***
*********************************************************************

                          1. Add a Section

                          2. Drop a Section

                          3. See Schedule for a Term

                          4. See Fee detail

                          5. See Transcript

                          q. Quit

Type in your option: 1
```

Option 1 Interface:

```
CRN: 10101
CSC2010, Introduction to Computer Science ADDED.
OR
Appropriate Error Message.
```

Option 2 Interface:

```
CRN: 10101
CSC2010, Introduction to Computer Science DROPPED.
OR
Appropriate Error Message.
```

Option 3 Interface:

```
Term: FA2002

CRN   Course  Title                                Days Time        Room Instructor
-------------------------------------------------------------------------------
10101 CSC2010 Introduction to Computer Science MWF  09.00-09.50 105G Bhola
...
...
```

Option 4 Interface:

```
Term: sp2003

Spring 2003

  Tuition - InState
      (12 hours)     1,500.00
  Technology Fee        75.00
  Health Fee            30.00
  Activity Fee          65.00
  Transportation Fee    25.00
                     --------
                     1,695.00
                     --------
```

Option 5 Interface:

```
Summer 2002
  CSC  1010 10101 Computers and Applications          3 A 12.00
  POLS 1200 10701 Intro Political Sci                 3 C  6.00
       Semester GPA: 3.00   GPA: 3.00

Fall 2002
  CSC  2010 10101 Introduction to Computer Science    3 A 12.00
  CSc  2310 Introduction to Programming in Java       3 A 12.00
  Math 2211 Calculus I                                5 B 15.00
       Semester GPA: 3.54    GPA: 3.35
...
...
```

Registrar Staff Menu:

```
$ java GoLunar OracleId
Oracle Password:xxxxxx
Semester (e.g. FA2003,SP2003,SU2003): SP2003
Username: 2000
Password:

*********************************************************************
***                                                               ***
***    Welcome to the GoLunar - Online Registration System        ***
***        Alison Payne - Registrar Staff                         ***
***                                                               ***
*********************************************************************

                        1. Load Sections from File

                        2. Load Grades from File

                        3. Increase Section Cap

                        4. Display Term Schedule

                        5. Display Student Transcript

                        6. Display Student Schedule and Fee Detail

                        q. Quit


Type in your option:
```

Option 1 Interface:

```
File Name: sections.dat
Sections Loaded
```

Option 2 Interface:

```
File Name: grades.dat
Grades Loaded
```

Option 3 Interface:

```
CRN: 10101
Old Capacity is 35
New Capacity: 45
Cap Updated for CRN 10101.
```

Option 4 Interface:

```
CRN   Course  Sec  Days  Time        Room  Cap Cur Avail Instructor Auth
-------------------------------------------------------------------------
10101 CSC2010 1    MWF   09.00-09.50 105G  35   3   32   Bhola      N
10102 CSC2010 2    MWF   10.00-10.50 105CS 40   5   35   Henry      N
...
```

```
...
```

Option 5 is similar to student option except here the system should accept student id as input and display that student's transcript.

Option 6 is similar to student options except here the system should accept student id (in addition to term) as input and display that student's term schedule and fee detail for the particular term.

Sample files for loading data in the Registrar's options are available in

sections.dat

```
SP
2003
10101,CSC,2010,1,MWF,09.00,09.50,105G,35,Bhola,N
10102,CSC,2010,2,MWF,10.00,10.50,105CS,40,Henry,N
10103,CSC,2310,1,MWF,12.00,12.50,106G,30,Henry,N
10104,CSC,2311,1,MWF,15.00,15.50,205G,35,Liu,N
10201,CSC,6220,1,TR,19.00,20.40,405G,25,Hundewale,N
10202,CSC,6710,1,TR,16.00,17.15,115CS,25,Madiraju,N
10203,CSC,8220,1,MWF,09.00,09.50,605G,25,Bourgeois,Y
10301,MATH,2211,1,TR,11.00,12.50,305G,35,Li,N
10302,MATH,2211,2,MWF,09.00,10.50,606GB,35,Miller,N
10303,MATH,2212,1,MWF,09.00,10.50,706GB,35,Davis,N
10304,MATH,2420,1,TR,14.00,14.50,106GB,35,Domke,N
10405,CSC,8710,1,MW,17.30,18.45,206GB,35,Dogdu,N
10406,CSC,8820,1,TR,19.15,20.55,306GB,3,Owen,N
```

grades.dat

```
FA
2002
1111,10101,A
1111,10103,A
1111,10301,B
3333,10201,B
3333,10202,B
3333,10203,A
```

8.2 Online Book Store Database System

Consider the following relational database schema written in Oracle SQL for an online book store application:

```
drop table books cascade constraints;
create table books (
  isbn char(10),
  author varchar2(100) not null,
  title varchar2(128) not null,
  price number(7,2) not null,
  subject varchar2(30) not null,
  primary key (isbn)
);
```

```
drop table members cascade constraints;
create table members (
  fname varchar2(20) not null,
  lname varchar2(20) not null,
  address varchar2(50) not null,
  city varchar2(30) not null,
  state varchar2(20) not null,
  zip number(5) not null,
  phone varchar2(12),
  email varchar2(40),
  userid varchar2(20),
  password varchar2(20),
  creditcardtype varchar2(10)
    check(creditcardtype in ('amex','discover','mc','visa')),
  creditcardnumber char(16),
  primary key (userid)
);
drop table orders cascade constraints;
create table orders (
  userid varchar2(20) not null,
  ono number(5),
  received date not null,
  shipped date,
  shipAddress varchar2(50),
  shipCity varchar2(30),
  shipState varchar2(20),
  shipZip number(5),
  primary key (ono),
  foreign key (userid) references members
);
drop table odetails cascade constraints;
create table odetails (
  ono number(5),
  isbn char(10),
  qty number(5) not null,
  price number(7,2) not null,
  primary key (ono,isbn),
  foreign key (ono) references orders,
  foreign key (isbn) references books
);
drop table cart cascade constraints;
create table cart (
  userid varchar2(20),
  isbn char(10),
  qty number(5) not null,
  primary key (userid,isbn),
  foreign key (userid) references members,
  foreign key (isbn) references books
);
```

The database consists of five tables:

1. `Books`: This table records information about the books on sale in the book store. Each book is classified under a “subject” to enable subject searches.

2. `Members`: This table records information about members of the application. Each member chooses their own user id and password at the time of registration.
3. `Orders`: This table records information about orders placed by members place orders. The orders may contain one or more books and the details of the order are kept in a separate table. A unique order number is generated by the system.
4. `OrderDetails`: This table records information about each order including the isbn and quantity of books in the order.
5. `Cart`: This table contains isbn and quantity of each book placed in the shopping cart of a member. Once a member checks out, the shopping cart is emptied and an order is created.

The book store application should be developed as a terminal application in Java and should be implemented in three phases:

Phase I of the project requires:

(a) Each student to create data for approximately 10 books for two different subjects (the subjects may be assigned by the instructor of the class to each student). The instructor may then consolidate the data into a large data set and give it out to the entire class. This is an easy way to create a large data set of books.

(b) Each student to build program the following interface implementing only the member registration and member login functions:

```
$ java OnlineBookStore

**********************************************************************
***                                                                ***
***             Welcome to the Online Book Store                   ***
***                                                                ***
**********************************************************************
                          1. Member Login

                          2. New Member Registration

                          q. Quit

Type in your option: 2

Welcome to the Online Book Store
    New Member Registration

Enter first name: Raj
Enter last name: Sunderraman
Enter street address: 123 Main Street
Enter city: Atlanta
Enter state: GA
Enter zip: 30303
Enter phone: 555-1212
Enter email address: raj@cs.gsu.edu
Enter userID: raj
Enter password: raj
```

```
Do you wish to store credit card information(y/n): y
Enter type of Credit Card(amex/visa): amex
Enter Credit Card Number: 121212121212121
Invalid Entry
Enter Credit Card Number: 1212121212121212
Invalid Entry
Enter Credit Card Number: 12121212121212

You have registered successfully.
Name:                Raj Sunderraman
Address:             123 Main Street
City:                Atlanta GA 30303
Phone:               555-1212
Email:               raj@cs.gsu.edu
UserID:              raj
Password:            raj
CreditCard Type:     amex
CreditCard Number:   12121212121212
Press Enter to go back to Menu

**************************************************************************
***                                                                    ***
***             Welcome to the Online Book Store                       ***
***                                                                    ***
**************************************************************************
                         1. Member Login

                         2. New Member Registration

                         q. Quit
Type in your option: 1

Enter userID: raj
Enter password: raj
**************************************************************************
***                                                                    ***
***                          Welcome to Online Book Store              ***
***                               Member Menu                          ***
***                                                                    ***
**************************************************************************
                     1. Browse by Subject

                     2. Search by Author/Title/Subject

                     3. View/Edit Shopping Cart

                     4. Check Order Status

                     5. Check Out

                     6. One Click Check Out

                     7. View/Edit Personal Information

                     8. Logout
Type in your option: 8
You have successfully logged out.
```

Phase II of the project requires the student to implement the following member functions:

1. Browse by Subject: This option should first list all subjects alphabetically; It then allows user to choose one subject; Upon choosing a subject, the program displays book details (2 books at a time on a screen); The option allows user to
(a) enter isbn to put in cart;
(b) press `ENTER` to return to main menu
(c) press `n ENTER` to continue browsing

User Interface follows:

```
Type in your option: 1

1. Cooking
2. Jokes
3. Sports

Enter your choice: 3

5 books available on this Subject

Author: Dom Parker
Title:  1,001 Baseball Questions Your Friends Can't Answer
ISBN:   0451191323
Price:   22.46
Subject Sports

Author: Timothy Jacobs
Title:  100 Atheletes Who Shaped Sports History
ISBN:   0912517131
Price:   32.56
Subject Sports


Enter ISBN to add to Cart or
n Enter to browse or
ENTER to go back to menu:
0451191323
Enter quantity: 2


Author: Michael Dregni
Title:  100 Years of Fishing
ISBN:   0896584305
Price:   15.95
Subject Sports

Author: David Claerbaut
Title:  The 1999 NBA Analyst: The Science of Hoops Magic
ISBN:   0878332103
Price:   20.95
Subject Sports
```

```
Enter ISBN to add to Cart or
n Enter to browse or
ENTER to go back to menu: n

Author: Sports Collectors Digest
Title:  1999 Sports Collectors Almanac (Serial)
ISBN:   0987654234
Price:   17.56
Subject Sports

Enter ISBN to add to Cart or
n Enter to browse or
ENTER to go back to menu:
0987654234
Enter quantity: 1
```

6. One Click Check Out

This option should move items in the cart to the `order` and `odetails` tables. Cart is emptied in the process and an invoice is printed. Shipping address is same as member address in this option. User Interface follows:

```
                    Invoice for Order no.117

     Shipping Address                    Billing address
     Name:     Raj Sunderraman           Name:     Raj Sunderraman
     Address:  123 Main Street           Address:  123 Main Street
               Atlanta                             Atlanta
               GA 33333                            GA 33333

-------------------------------------------------------------------------------
ISBN        Title                                              $  Qty   Total
-------------------------------------------------------------------------------
0451191323  1,001 Baseball Questions Your Friends Can't Answer 22.45  1   22.45
0987654234  1999 Sports Collectors Almanac(Serial)             17.55  1   17.55
-------------------------------------------------------------------------------
Total =                                                                  $40.01
-------------------------------------------------------------------------------

Press enter to go back to Menu
```

4. Check Order Status

This option should list all orders for member and should allow user to choose one order to see details. User Interface follows:

```
Orders placed by Raj Sunderraman
-----------------------------------------------------------
ORDER NO    RECEIVED DATE                SHIPPED DATE
-----------------------------------------------------------
117         3-1-2001                     3-3-2001
-----------------------------------------------------------

Enter the Order No to display its details or (q) to quit: 117
```

```
                    Details for Order no.117

     Shipping Address                    Billing address
     Name:     Raj Sunderraman           Name:     Raj Sunderraman
     Address:  123 Main Street           Address:  123 Main Street
               Atlanta                             Atlanta
               GA 33333                            GA 33333

-------------------------------------------------------------------------------
ISBN        Title                                              $  Qty   Total
-------------------------------------------------------------------------------
0451191323  1,001 Baseball Questions Your Friends Can't Answer 22.45  1   22.45
0987654234  1999 Sports Collectors Almanac(Serial)             17.55  1   17.55
-------------------------------------------------------------------------------
Total =                                                                 $40.01

Press Enter to go back to Menu
```

Phase III of the project requires the student to implement the following member functions:

2. Search by Author/Title

This option should provide 3 sub-options:

1. Author Search
2. Title Search
3. Go Back to Member Menu

In the Author or Title search sub-option, the user may enter a substring and the system should respond with all books which contain the substring in the title/author. The display should be done 2 books at a time on a screen.

The system should also allow user to enter isbn to put in cart;

to press `ENTER` to return to main menu

to press `n ENTER` to continue browsing

User Interface follows:

```
1. Author Search
2. Title Search
3. Go Back to Member Menu

Type in your option: 2

Enter title or part of the title: cook
2 books found


Author: Irma S. Rambauer
Title:  Joy of Cooking
ISBN:   0452279232
Price:   15.25
Subject Cooking


Author: Jennifer E. Darling
Title:  Better Homes and Gardens New Cook Book
```

```
ISBN:   0696201887
Price:   21.96
Subject Cooking


Enter ISBN to add to Cart or
      Enter to browse or
      n ENTER to return to menu: 0696201887
Enter quantity: 1


1. Author Search
2. Title Search
3. Go Back to Member Menu

Type in your option: 2

Enter title or part of the title: Computer
0 books found

Enter ISBN to add to Cart or
      Enter to browse or
      n ENTER to return to menu: n

1. Author Search
2. Title Search
3. Go Back to Member Menu

Type in your option: 1

Enter name or part of the name: am
1 books found

Author: Irma S. Rambauer
Title:  Joy of Cooking
ISBN:   0452279232
Price:   15.25
Subject Cooking

Enter ISBN to add to Cart or
      Enter to browse or
      n ENTER to return to menu: 0452279232
Enter quantity: 2


1. Author Search
2. Title Search
3. Go Back to Member Menu

Type in your option: 3
```

3. View/Edit Shopping Cart

This option should show the contents of the cart; It should then provide options to delete items or edit (change quantity) items. User Interface (for delete and update cart) follows:

```
Current Cart Contents:

ISBN        Title                                                    $  Qty   Total
-------------------------------------------------------------------------------
0696201887  Better Homes and Gardens New Cook Book             21.95  1   21.95
0452279232  Joy of Cooking                                     15.25  2   30.50
-------------------------------------------------------------------------------
Total =                                                                  $52.45

Enter d to delete item
e to edit cart or
q to go back to Menu: d
Enter isbn of item: 0452279232
Delete Item Completed
Press enter to go back to Menu

Type in your option: 3

Current Cart Contents:

ISBN        Title                                                    $  Qty   Total
-------------------------------------------------------------------------------
0696201887  Better Homes and Gardens New Cook Book             21.95  1   21.95
-------------------------------------------------------------------------------
Total =                                                                  $21.95

Enter d to delete item
e to edit cart or
q to go back to Menu: e
Enter isbn of item: 0696201887
Enter new Quantity: 2
Edit Item Completed
Press enter to go back to Menu
```

5. Check Out

This option should display invoice; request user of they want to provide shipping address (if no use current address in file for shipping); Also this option should ask if a new credit card should be used. Finally, an invoice should be printed. User Interface follows:

```
Current Cart Contents:

ISBN        Title                                                    $ Qty   Total
-------------------------------------------------------------------------------
0696201887  Better Homes and Gardens New Cook Book             21.95  2   43.91
-------------------------------------------------------------------------------
Total                                                                    $43.91
-------------------------------------------------------------------------------

Proceed to check out(Y/N): y
Do you want to enter new shipping address(y/n): y
Enter first name: John
Enter last name: Smith
```

```
Enter street: 123 Elm Street
Enter city: Atlanta
Enter state: GA
Enter zip: 11111
Do you want to enter new CreditCard Number(y/n): n

                    Invoice for Order no.118

     Shipping Address                    Billing address
     Name:     John Smith                Name:     Raj Sunderraman
     Address:  123 Elm Street            Address:  123 Main Street
               Atlanta                             Atlanta
               GA 11111                            GA 33333

-------------------------------------------------------------------------------
ISBN        Title                                          $  Qty   Total
-------------------------------------------------------------------------------
0696201887  Better Homes and Gardens New Cook Book      21.95  2   43.91
-------------------------------------------------------------------------------
Total =                                                          $43.91
-------------------------------------------------------------------------------

Press enter to go back to Menu
```

8.3 Online Shopping System

Using PhP and MySQL, implement a Web-based application for an online video-store. The online video-store maintains an inventory of DVDs. Customers become member of this online store. They are able to search for DVDs of their interest and add DVDs to their shopping cart. At any time, they are able to edit the shopping cart and also are able to check out. The initial login screen shown in the Figure 8.1 allows an existing customer to sign in or a new customer to register. The new customer registration screen is shown in Figure 8.2. Upon successful sign-in, the 3-frame page shown in Figure 8.3 is displayed. As one can see, there are six different options for the customer:

1. **Search by Keyword:** This option allows the customer to perform a keyword search of the DVD titles (substring; case insensitive comparison). Successful matches are shown on the right frame (Figure 8.4). The customer may then choose certain quantities of the DVDs and add them to the shopping cart. Upon successful addition to the shopping cart, a message should be shown on the right frame.
2. **View/Edit Cart:** Upon clicking this option, the system should display the shopping cart on the right side frame (Figure 8.5). Here, the customer may edit the shopping cart by changing quantities including replacing a value with a zero. Upon submission, the cart should be updated and a message should be displayed.
3. **Update Profile:** This option brings up the customers profile (Figure 8.6) on the right frame. The user may modify any field and submit. Upon successful update, a message should be displayed.
4. **Check Order Status:** This option allows the customer to see all their orders (Figure 8.7). Upon clicking the order number link the details for that order should be displayed in a tabular format.

5. **Check Out:** Upon clicking this link, the system should empty the cart and move the items into the `orders` and `odetails` tables. An invoice (Figure 8.8) should be printed on the right frame.
6. **Logout:** Upon logout, the system should display 3 Options (Figure 8.9) to the user. The user may check out, save cart and logout or empty cart and logout. Upon checkout a similar action should take place as earlier. Upon the other two options, appropriate action should take place and a message should be displayed. If the cart was empty to begin with these 3 options should not be shown and the customer should be logged out.

The database schema for the online shopping cart example is shown below:

```
create table parts(
  pno      integer(5) not null,
  pname    varchar(30),
  qoh      integer,
  price    decimal(6,2),
  olevel   integer,
  primary key (pno));

create table customers (
  cno      integer(10) not null auto_increment=100,
  cname    varchar(30),
  street   varchar(50),
  city     varchar(30),
  state    varchar(30),
  zip      integer(5),
  phone    char(12),
  email    varchar(50),
  password varchar(15),
  primary key (cno));

create table cart(
  cartno   integer(10) not null auto_increment,
  cno      integer(10) not null,
  pno      integer(5) not null,
  qty      integer,
  primary key (cartno, pno),
  foreign key (cno) references customers,
  foreign key (pno) references parts);

create table orders (
  ono      integer(5) not null auto_increment=1000,
  cno      integer(10),
  received date,
  shipped  date,
  primary key (ono),
  foreign key (cno) references customers);

create table odetails (
  ono      integer(5) not null,
  pno      integer(5) not null,
  qty      integer,
  primary key (ono,pno),
  foreign key (ono) references orders,
```

```
foreign key (pno) references parts);
```

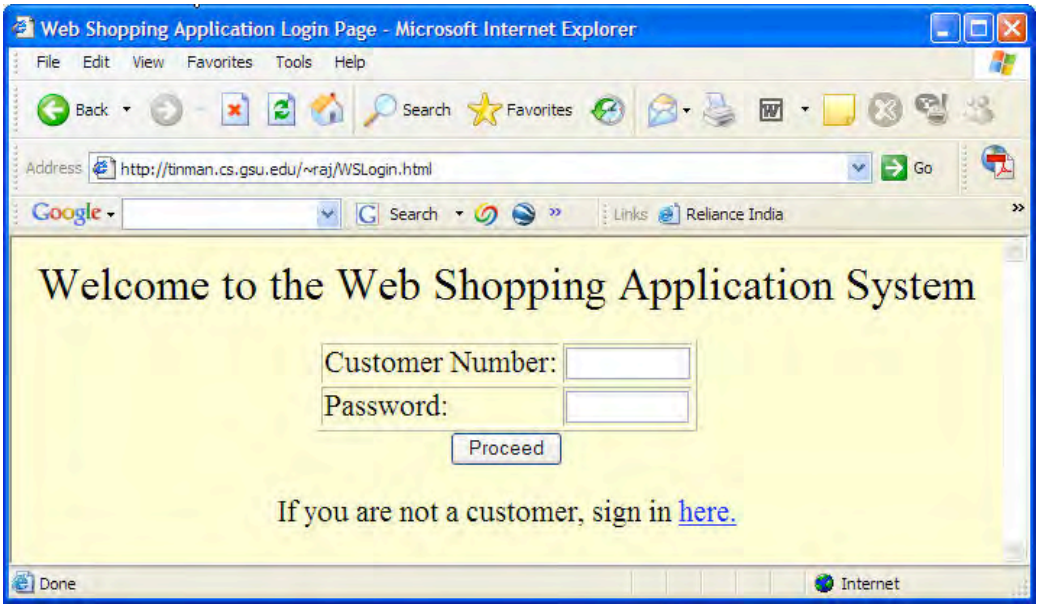

Figure 8.1 Web Shopping – Initial Screen

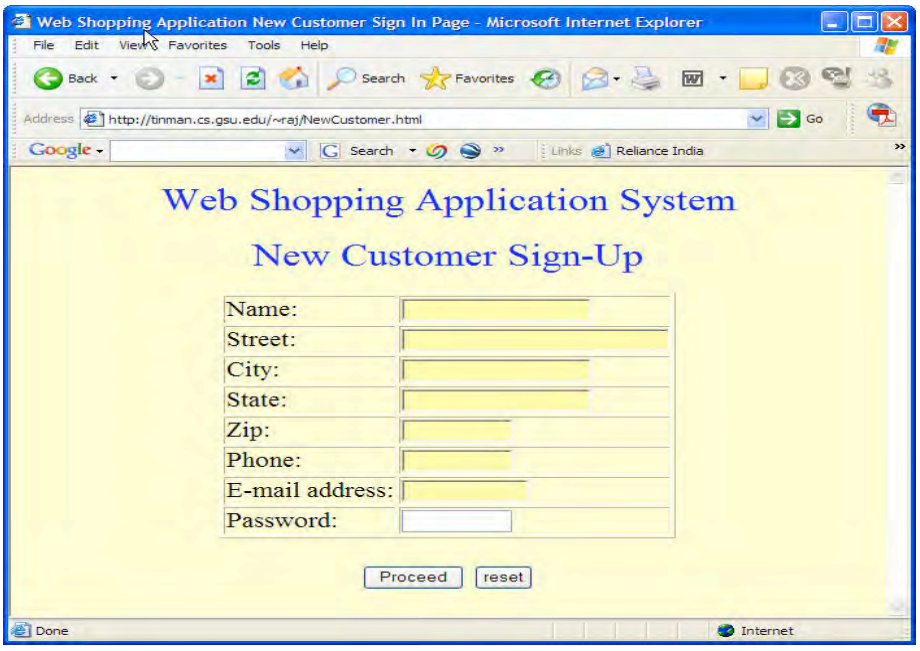

Figure 8.2 Web Shopping – New Customer Registration

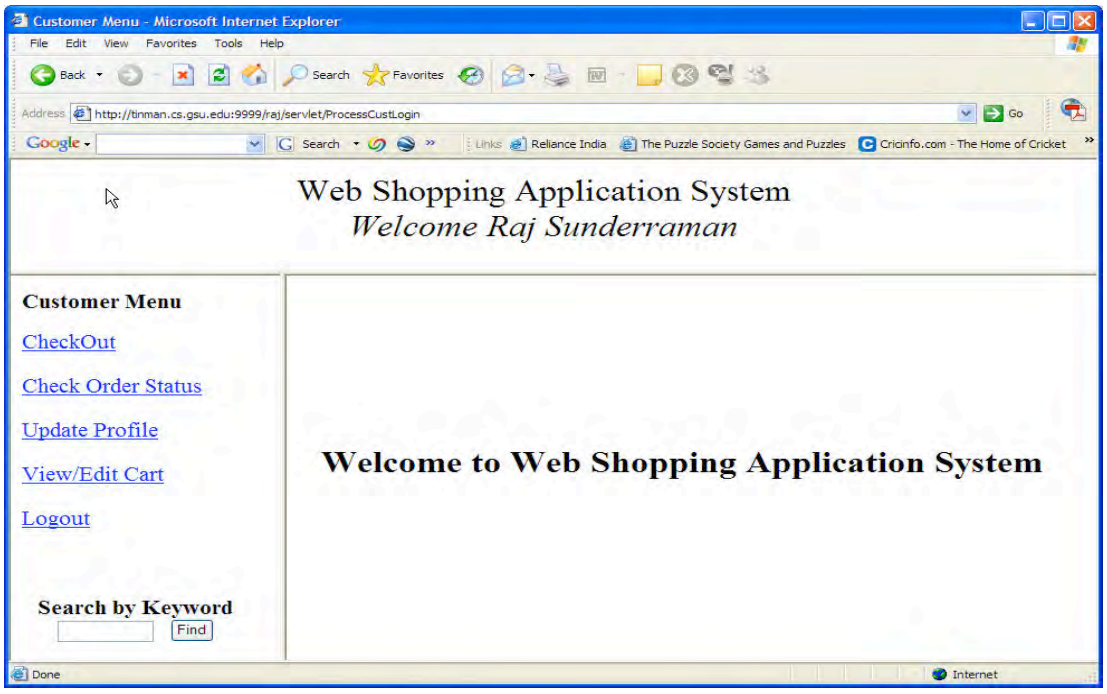

Figure 8.3: Web Shopping – Successful Sign-In 3-Frame Page

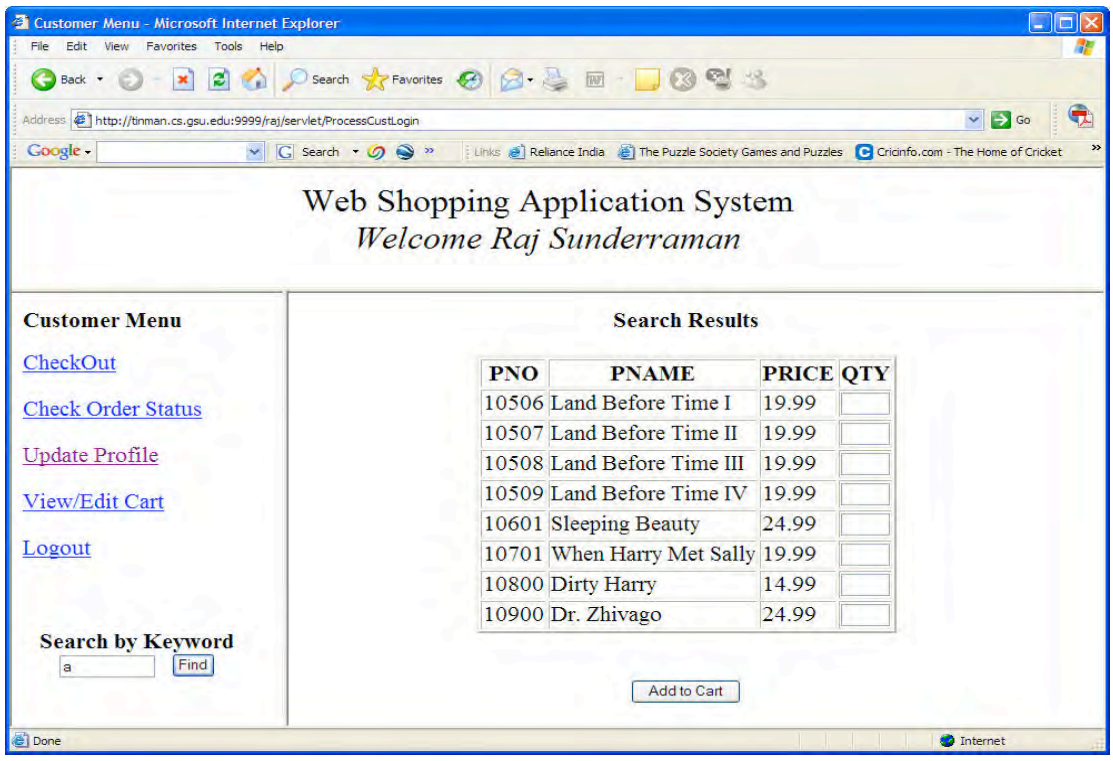

Figure 8.4: Web Shopping – Search Result Page

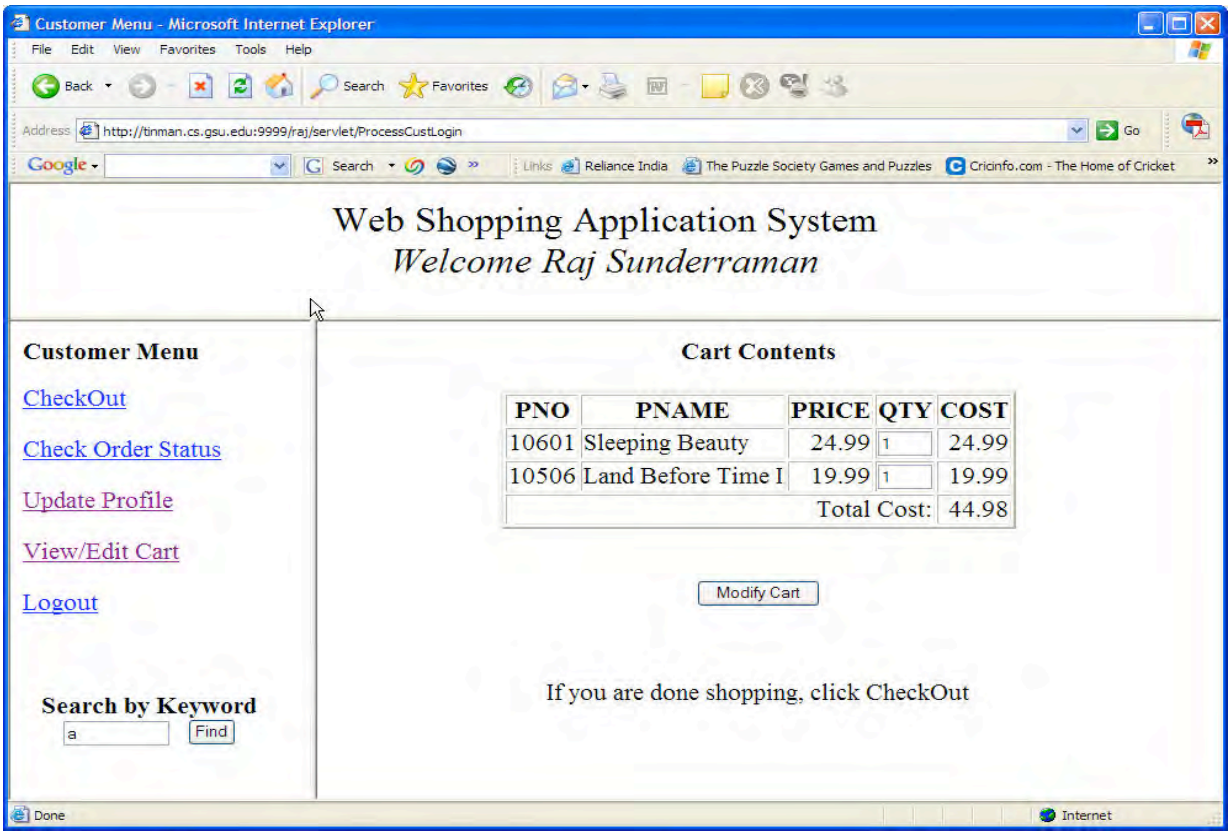

Figure 8.5: Web Shopping – View/Edit Cart

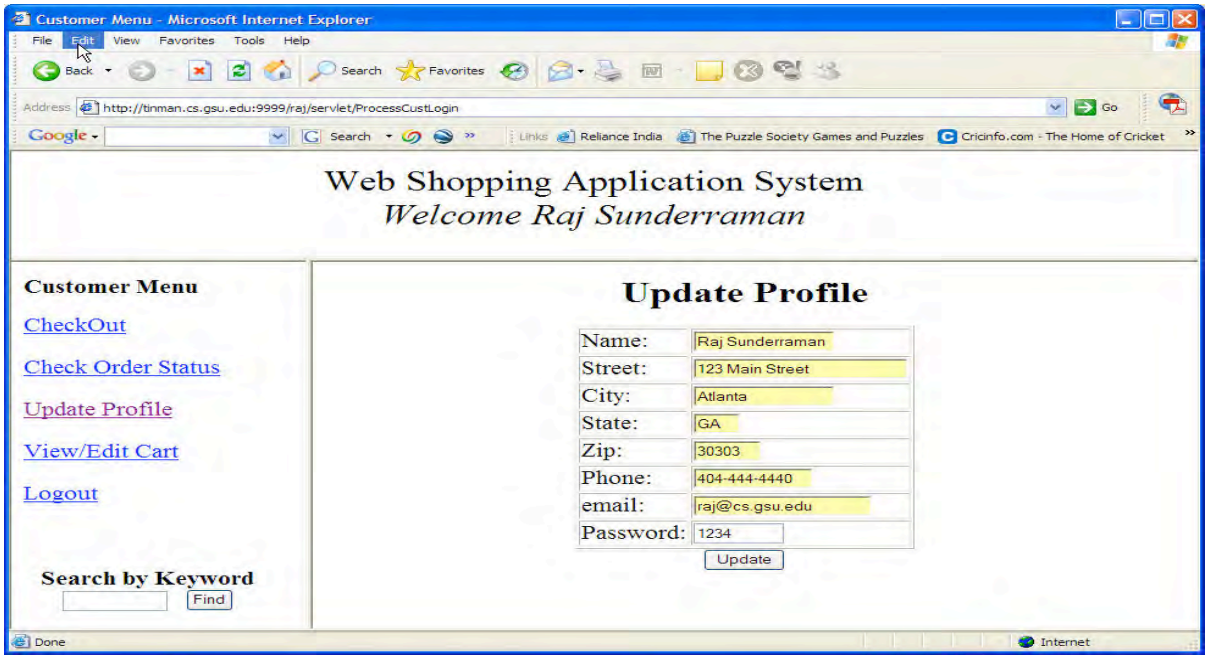

Figure 8.6: Web Shopping – Update Profile

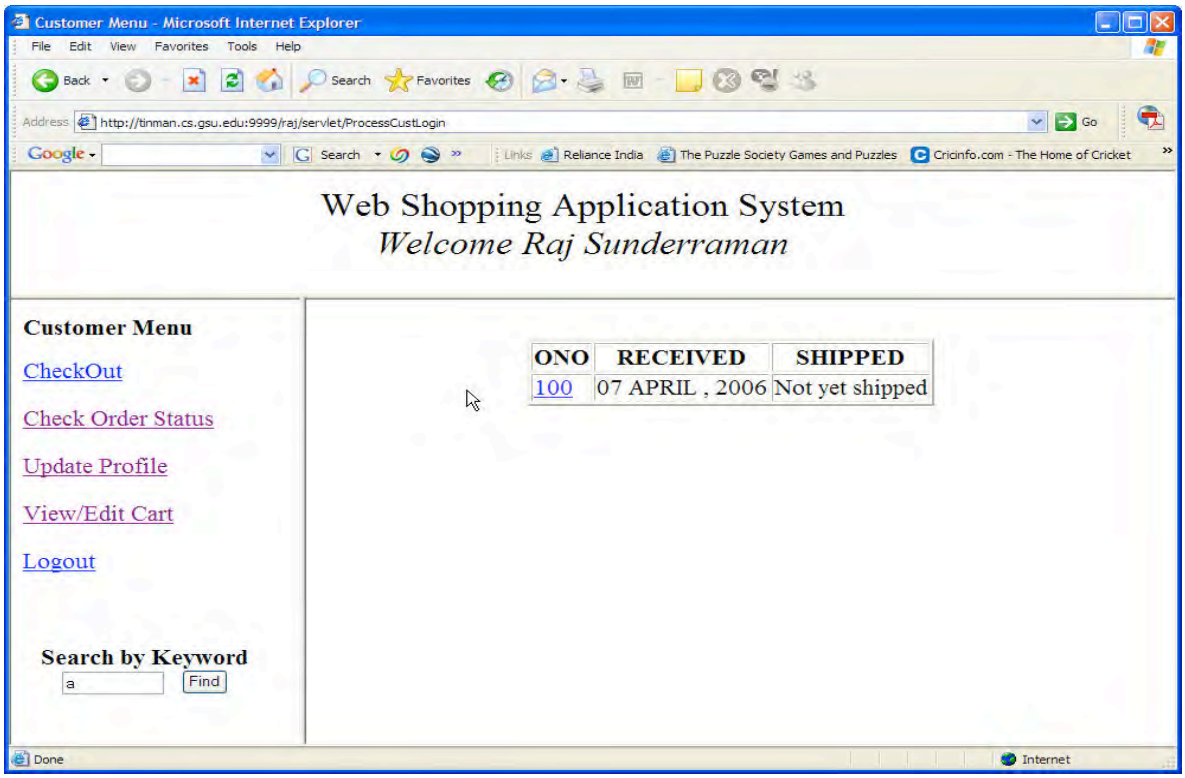

Figure 8.7: Web Shopping – Check Order Status

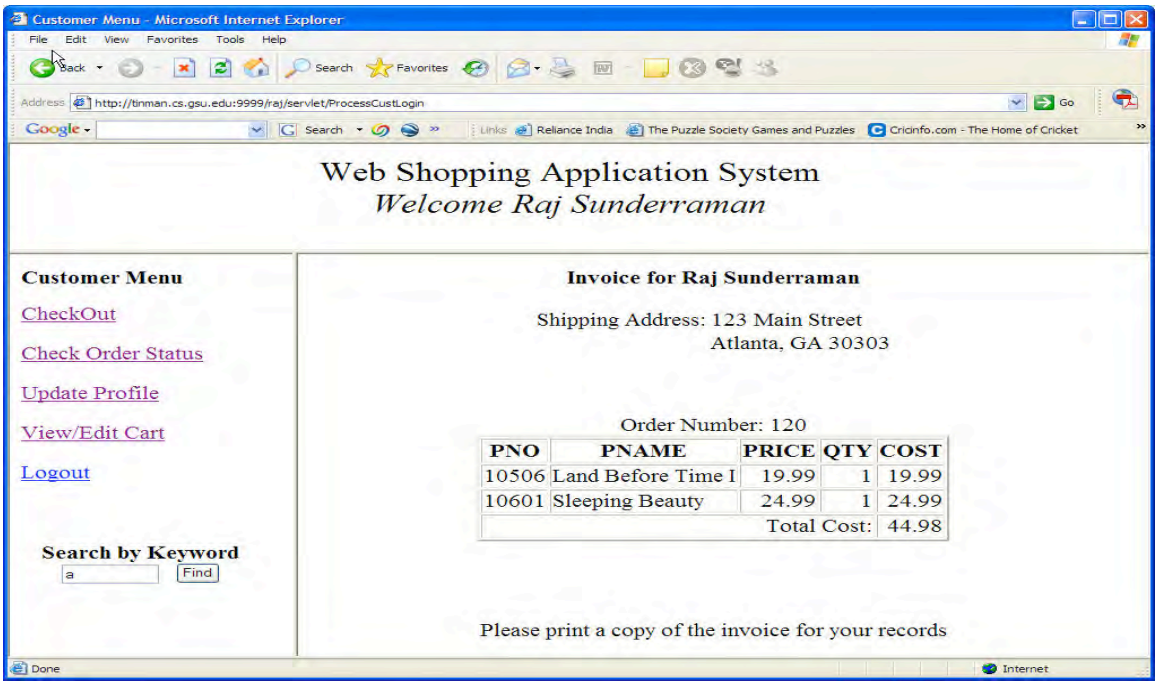

Figure 8.8: Web Shopping – Check Out

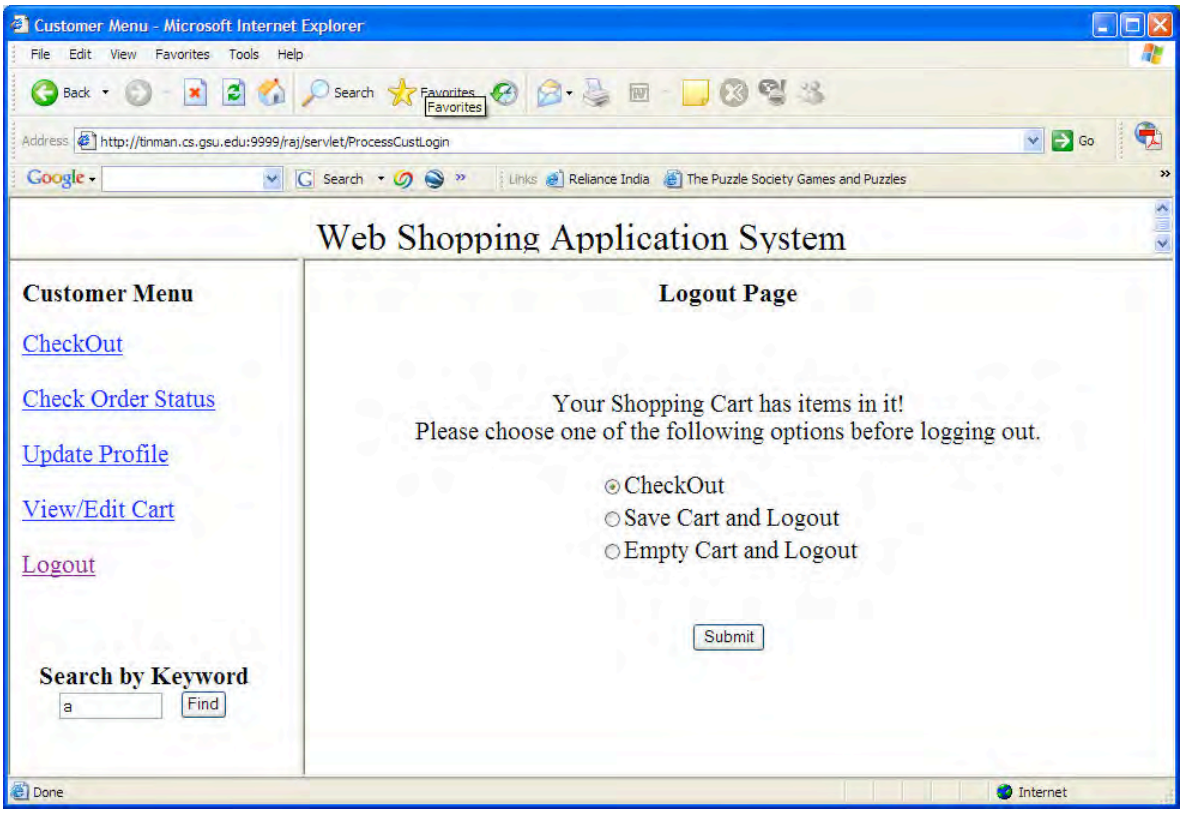

Figure 8.9: Web Shopping – Log Out

8.4 Online Bulletin Board System

Using PhP and MySQL implement an online bulletin board system that allows a set of authorized users to participate in an online discussion forum. The data for the bulletin board system should be stored in a MySQL database with the following schema:

```
create table bbusers (
  email    varchar(50),
  name     varchar(30),
  password varchar(10),
  nickname varchar(30),
  primary key (email)
);

create table postings (
  postId       integer(5) auto_increment,
  postDate     datetime,
  postedBy     varchar(50),
  postSubject  varchar(100),
  content      varchar(512),
  ancestorPath varchar(100),
  primary key (postId),
```

```
  foreign key (postedBy) references bbusers
);
```

The database has two tables:

1. `bbusers`: This table records information about users of the bulletin board. The `email` and `password` fields are used for signing into the system.

2. `postings`: This table records information about all postings as well as follow-up postings of the bulletin board. Each posting is assigned a unique `postId`. To keep track of the "tree-structure" generated by follow-up postings, the system keeps track of the path from root message to the posting in the `ancestorPath` attribute. The path is recorded as a colon separated list of posting Ids; for example the ancestor path `1:5:6:12` would indicate that the current posting has a parent posting with `postId=12`, a grand-parent posting with `postId=6`, a great-grand-parent with `postId=5`, and a great-great-grandparent with `postId=1`. With this structure, the entire bulletin board messages can be viewed as a collection (forest) of trees.

The Web application should implement the following basic functions:

1. User sign-in and sign-out.
2. Default display of messages in reverse chronological order and properly indented follow-up messages.
3. Post message and post follow-up message by user.

Figure 8.10 and 8.11 show possible user interfaces for the main display page and the follow-up display page.

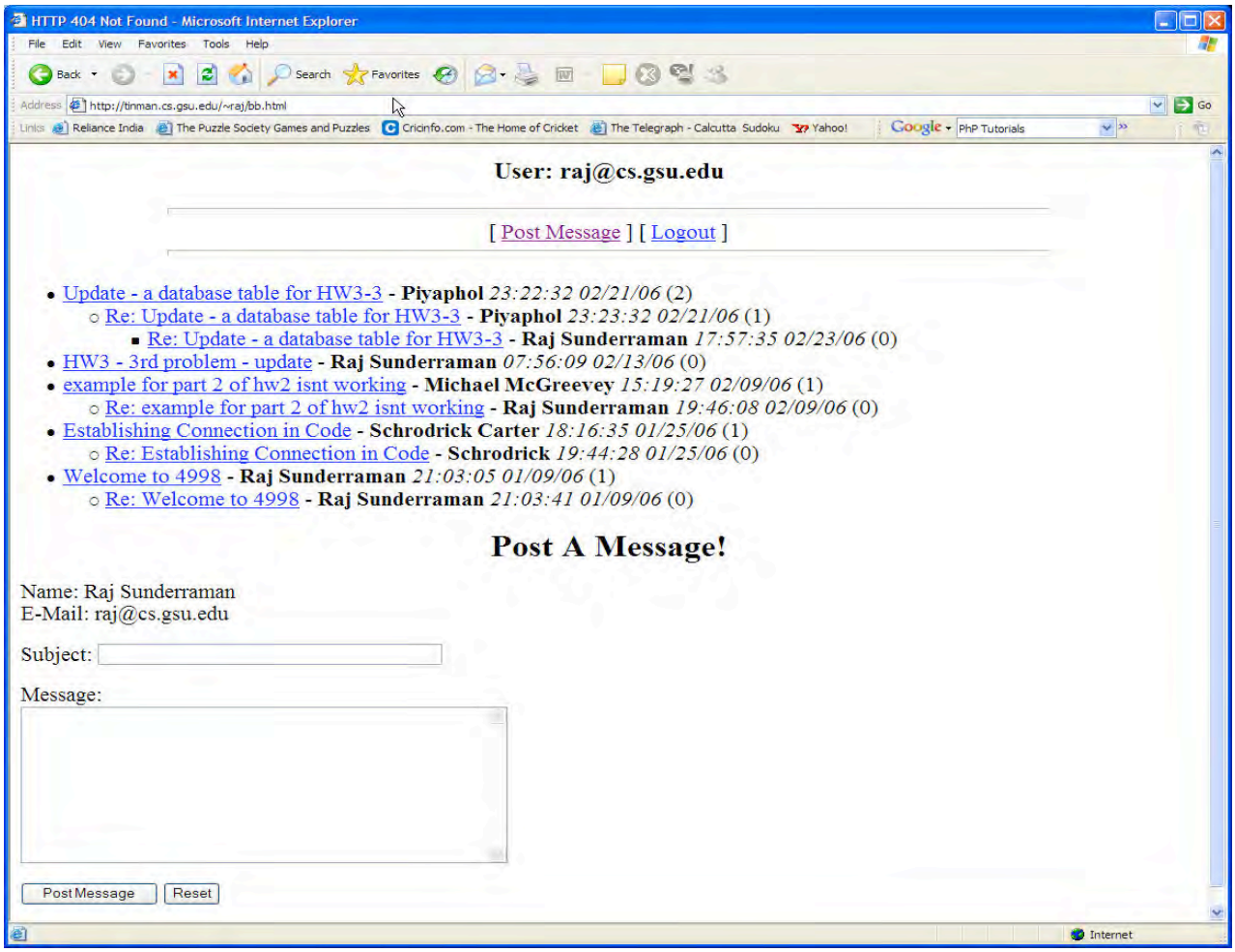

Figure 8.10: Bulletin Board – Main Display Page

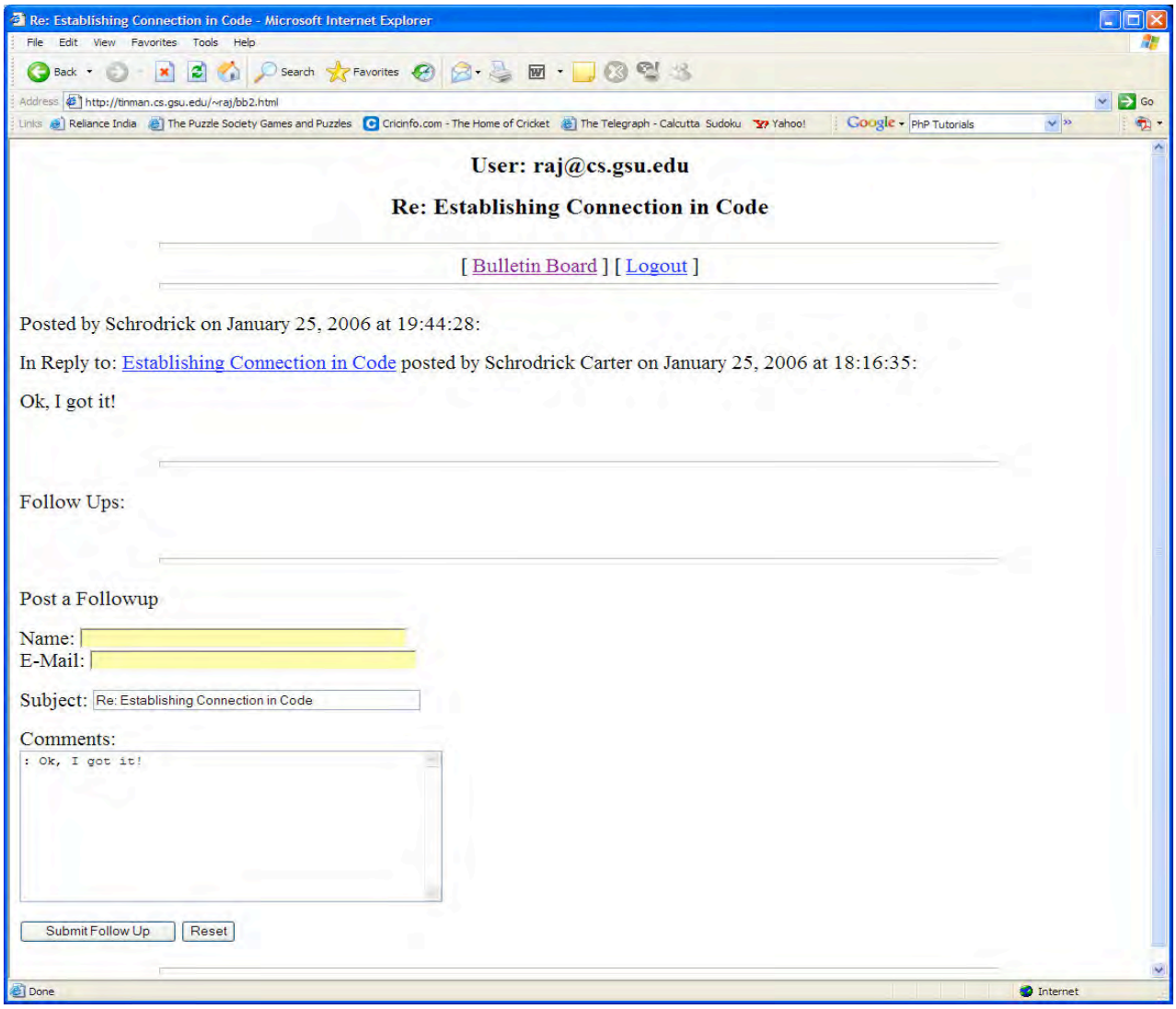

Figure 8.11: Bulletin Board – Follow-up Listing Page

8.5 Online Exam Management System

Using PhP and MySQL implement an online exam management system that allows (a) an administrator to create/delete/edit online multiple-choice exams and (b) student users to take these exams and view the results. The relational schema for the system is already designed and is shown below:

```
drop table exam cascade constraints;
create table exam (
  eno number(5),
  etitle varchar2(50),
  timeAllowed  number(8), -- minutes
  numberOfQuestionsPerPage number(3),
  primary key (eno)
```

```
);

drop table question cascade constraints;
create table question (
  eno number(5),
  qno number(5),
  qtext varchar2(2048), -- maybe be CLOB object
  correctAnswer char(1),  -- must be one of the options
  foreign key (eno) references exam,
  primary key (eno,qno)
);

drop table answerOption cascade constraints;
create table answerOption (
  eno number(5),
  qno number(5),
  ono char(1) check (ono in ('A','B','C','D','E')),
  optionText varchar2(256),
  foreign key (eno,qno) references question,
  primary key (eno,qno,ono)
);

drop table users cascade constraints;
create table users (
  uno number(5), -- primary key; system generated starting at 1
                 -- first user gets 1 and subsequent users get max+1
  email varchar2(64),  -- unique key used for signing in
  password varchar2(64),
  fname varchar2(64) not null,
  lname varchar2(64) not null,
  address1 varchar2(64),
  address2 varchar2(64),
  city varchar2(64),
  state varchar2(64),
  zip number(5),
  primary key (uno)
);

drop table enrolls cascade constraints;
create table enrolls (
  uno number(5),
  eno number(5),
  startTime date,
  finishTime date,
  foreign key (uno) references users,
  foreign key (eno) references exam,
  primary key (uno,eno)
);

drop table userResponse cascade constraints;
create table userResponse (
  uno number(5),
  eno number(5),
  qno number(5),
  response char(1)
    check (response in ('A','B','C','D','E','N')), -- N for No Answer
  foreign key (uno,eno) references enrolls,
```

```
  foreign key (eno,qno) references question,
  primary key (uno,eno,qno)
);
```

Here is some sample data for the `exam`, `question`, and `answerOption` tables:

```
insert into exam values (3,'Elementary History',10,3);

insert into question values
  (3,1,'The Battle of Gettysburg was fought during which war?','C');
insert into answerOption values (3,1,'A','World War II');
insert into answerOption values (3,1,'B','The Revolutionary War');
insert into answerOption values (3,1,'C','The Civil War');
insert into answerOption values (3,1,'D','World War I');

insert into question values
  (3,2,'Neil Armstrong and Buzz Aldrin walked how many \n' ||
       'minutes on the moon in 1696?','B') ;
insert into answerOption values (3,2,'A','123');
insert into answerOption values (3,2,'B','None');
insert into answerOption values (3,2,'C','10');
insert into answerOption values (3,2,'D','51');

insert into question values
  (3,3,'Which Presidents held office during World War II?','D');
insert into answerOption values (3,3,'A','Franklin D. Roosevelt');
insert into answerOption values (3,3,'B','Dwight D. Eisenhower');
insert into answerOption values (3,3,'C','Harry Truman');
insert into answerOption values (3,3,'D','Both A and C');

insert into question values
  (3,4,'In a communist economic system, people:','B');
insert into answerOption values (3,4,'A','Are forced to work as slaves');
insert into answerOption values (3,4,'B','Work for the common good');
insert into answerOption values (3,4,'C','Work from home computers');
insert into answerOption values (3,4,'D','Don''t work');


insert into question values
  (3,5,'Which president did not die while in office?','D');
insert into answerOption values (3,5,'A','John F. Kennedy');
insert into answerOption values (3,5,'B','Franklin D. Roosevelt');
insert into answerOption values (3,5,'C','Abraham Lincoln');
insert into answerOption values (3,5,'D','Ronald Reagan');
insert into answerOption values (3,5,'E','James A. Garfield');

insert into question values
  (3,6,'Which state refused to attend the Constitutional Convention \n' ||
       'in 1787 because it didn''t want the United States government \n' ||
       'to interfere with already established state affairs?','A');
insert into answerOption values (3,6,'A','Rhode Island');
insert into answerOption values (3,6,'B','New Hampshire');
insert into answerOption values (3,6,'C','New Jersey');
insert into answerOption values (3,6,'D','New York');

insert into question values
```

```
  (3,7,'Who founded Buddhism?','A');
insert into answerOption values (3,7,'A','Siddharta Gautama');
insert into answerOption values (3,7,'B','Jesus Christ');
insert into answerOption values (3,7,'C','Mahatma Gandhi');
insert into answerOption values (3,7,'D','Muhammad');

insert into question values
  (3,8,'Where is India?','D');
insert into answerOption values (3,8,'A','Australia');
insert into answerOption values (3,8,'B','America');
insert into answerOption values (3,8,'C','Africa');
insert into answerOption values (3,8,'D','Asia');


insert into question values
  (3,9,'What is the dominant religion in India?','B');
insert into answerOption values (3,9,'A','Islam');
insert into answerOption values (3,9,'B','Hinduism');
insert into answerOption values (3,9,'C','Christianity');
insert into answerOption values (3,9,'D','Buddhism');

insert into question values
  (3,10,'Near which river did archaeologists find India''s \n' ||
        'first civilization?','B');
insert into answerOption values (3,10,'A','The Tiber River');
insert into answerOption values (3,10,'B','The Indus River');
insert into answerOption values (3,10,'C','The Yellow River');
insert into answerOption values (3,10,'D','The Nile River');
```

The project should be implemented in two separate modules:

1. **Admin Module:** The admin module should allow administrators to create multiple-choice exams. This is basically a data input/update module that allows admin user to
    - **Create Exam**: After collecting top level exam details such as exam title etc, the user should be allowed to add questions one at a time. The add question screen should contain input boxes and text areas for top level question data and a pull down list for number of options (2 through 5). Using Javascript, you should create the right number of answer option data input text areas for answer option text along with a check box that can be checked for "correct answer" option. Once all information is given, the user can submit the question to be added to the database. The user should be presented with the add question screen in case they want to add the next question.
    - **Delete Exam**: Given a list of exams, the user chooses exam to be deleted. Only exams in which no one has signed up should be presented. A "confirm delete" screen should be presented before the exam is deleted.
    - **Edit Exam**: The user should be able to add new questions at a particular position and delete a question.
2. **User Module:** The user module should allow ordinary users to register, sign in, update profile, sign up for exams, take exams, and see their results.

- **Register, Change Password, Sign In, and Sign Out**: A standard login page (with email and password text boxes) along with a "If you do not have an account, register here" link. Once logged in successfully, the user should be presented with several options including "Update Profile" in which they can change some of the data about themselves such as password, address etc.
- **Enroll in Exam(s)**: A menu option for the user - used to enroll in a particular exam (you may present a select list of all available exams and ask the user to choose one). Note: If the student is already enrolled in the exam and has finished taking the exam or is currently taking the exam (i.e. has started taking the exam but not yet finished), a warning should be issued stating that his answers will be reset. You may confirm that the user wishes to reset the old exam. Once enrolled, you should present the user with a confirmation which includes details about the exam he or she has just signed up for.
- **Take an Exam**: The user should be presented questions from where they left off the last time they signed on to take the exam. Questions should be presented in order using the pre-defined number of questions per page. Once answers are submitted, they cannot be revisited. The user is then presented the next set of questions until time runs out or there are no more questions.
- **View their Grade Report(s)**: The user chooses the exam for which they like to view results. Only list of exams that have been completed should be presented. The format of the grade report is up to you, but must include number of questions answered correctly, total number of questions, percentage coorect, and a detailed listing of user responses and correct answers.

8.6 Online Auctions

Using PhP and MySQL implement an online auction website (AuctionBase). The relational schema for the system is already designed and is shown below:

```
drop table member cascade constraints;
create table member (
  mid varchar2(10) not null,
  email varchar2(40) not null,
  fname varchar2(20) not null,
  lname varchar2(20) not null,
  street varchar2(50) not null,
  city varchar2(30) not null,
  state varchar2(20) not null,
  zip number(5) not null,
  phone varchar2(12),
  password varchar2(20),
  primary key (mid)
);
--
drop table category cascade constraints;
create table category (
  cname varchar2(120),
  primary key (cname)
);
--
```

```
drop table item cascade constraints;
create table item (
  ino number(5),
  title varchar2(128) not null,
  category varchar2(120) not null,
  description varchar2(2000),
  openDateTime date,
  sellerId varchar2(10) not null,
  startingBid number(7,2) not null,
  bidIncrement number(7,2) not null,
  closeDateTime Date,
  winnerId varchar2(10),
  primary key (ino),
  foreign key (category) references category,
  foreign key (sellerId) references member,
  foreign key (winnerId) references member
);
--
drop table bid cascade constraints;
create table bid (
  ino number(5),
  buyerId varchar2(10),
  bidPrice number(7,2),
  timeOfBid date,
  primary key (ino,buyerId,timeOfBid),
  foreign key (ino) references item,
  foreign key (buyerId) references member
);
--
drop table rating cascade constraints;
create table rating (
  ino number(5),
  buyerRating number(1) check (buyerRating between 1 and 5),
  buyerComment varchar2(100),
  sellerRating number(1) check (sellerRating between 1 and 5),
  sellerComment varchar2(100),
  primary key (ino),
  foreign key (ino) references item
);
```

Initial data is given below:

```
insert into member values
  ('a100','a@cs.gsu.edu','Tom','Jones','120 Main Street','Atlanta','GA',30303,
   '404-111-1110','a123');
insert into member values
  ('m100','m@cs.gsu.edu','Jim','Smith','121 Main Street','Atlanta','GA',30303,
   '404-111-1111','m123');
insert into member values
  ('p100','p@cs.gsu.edu','Don','Fleming','122 Main
Street','Atlanta','GA',30303,
   '404-111-1112','p123');
insert into member values
  ('q100','q@cs.gsu.edu','James','John','123 Main Street','Atlanta','GA',30303,
   '404-111-1113','q123');
insert into member values
```

```
  ('s100','s@cs.gsu.edu','Monty','Jones','124 Main
Street','Atlanta','GA',30303,
   '404-111-1114','s123');
--
insert into category values ('Books:Biology');
insert into category values ('Books:Computers');
insert into category values ('Books:Economics');
insert into category values ('Books:Fiction');
insert into category values ('Computers:Apple:Desktops');
insert into category values ('Computers:Apple:Laptops');
insert into category values ('Computers:PCs:Desktops');
insert into category values ('Computers:PCs:Laptops');
insert into category values ('Computers:Storage:Hard Drives');
insert into category values ('Computers:Storage:Flash Drives');
insert into category values ('DVDs:Action');
insert into category values ('DVDs:Comedy');
insert into category values ('Music:Blues');
insert into category values ('Music:Jazz');
insert into category values ('Music:World');
insert into category values ('Video Games:Systems:XBox 360');
insert into category values ('Video Games:Systems:Wii');
insert into category values ('Video Games:Systems:Playstation');
insert into category values ('Video Games:Systems:Nintendo DS');
insert into category values ('Video Games:Games:XBox 360');
insert into category values ('Video Games:Games:Wii');
insert into category values ('Video Games:Games:Playstation');
insert into category values ('Video Games:Games:Nintendo DS');
--
insert into item values
  (1000,'Mario Party IV','Video Games:Games:Wii','Excellent Condition ' ||
      'Best Seller; Super Graphics',
      to_date('21-APR-2008 1700','DD-MON-YYYY HH24MI'),
      'a100',20.00,2.00,
      to_date('28-APR-2008 1700','DD-MON-YYYY HH24MI'),
      null);
```

The project should be implemented in the following stages:

Stage I: Implement the "Browse/Search" part of the AuctionBase website.

Each Web page in this part should have a "Top" portion which contains:

- A "Bread Crumb" indicating the level of the category being browsed. For example, the initial page should have HOME as the bread crumb. Lower levels of categories would have bread crumbs such as HOME::DVDS::FICTION, HOME::DVDS, HOME::BOOKS, HOME::BOOKS::ECONOMICS, etc. Each of these terms in the bread crumbs should be hyper-linked so that when they are clicked, the page refreshes and shows the level that is clicked.
- A "Search" text box and a pull-down list of top-level categories and a submit button. This should allow users to search for items using keyword. The results of the search should be shown in list form.

The Web page should contain a "Bottom" portion which lists either the sub-categories for the rightmost category in the bread crumb or a list of items if the rightmost category in the bread crumb is the lowest level category. These sub-categories and items should be hyper-linked as well. The sub-categories should be hyper-linked to the next level page and the items should be hyper-linked to a "Detail" page for the item.

The "Detail Page" for the item should list all details of the item and should provide a text box for the user to enter a bid and a submit button.

Stage II: Implement the following functions:

1. Login/logout.
2. Update member profile.
3. Place a bid.
4. View closed items along with open items. This should be be displayed along with browse/search options, but with no text box/submit button for "bid".
5. Place feedback.
6. View feedback/ratings for a particular member.

BIBLIOGRAPHY

1. R. Elmasri and S. Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley, 2010.

2. M. Fisher, J. Ellis , and J. Bruce, JDBC API Tutorial and Reference, 3rd Edition, Addison-Wesley Professional, 2003.

3. R. Sunderraman, Oracle 10g Programming: A Primer, Addison-Wesley, 2008.

4. J.D. Ullman, Principles of Database and Knowledge-Base Systems, Volume 1, Computer Science Press, 1988.

5. J.D. Ullman and J. Widom, A First Course in Database Systems, 3rd Edition, Prentice Hall 2007.

6. H. Williams and D. Lane, Web Database Applications with PhP and MySQL, O'Reilley, 2004.

Useful URLs:

1. Computer Associates: http://www.ca.com/

2. Java: http://java.sun.com

3. JFlex: http://www.jflex.de/

4. JCup: http://www.cs.princeton.edu/~appel/modern/java/CUP/

5. SWI Prolog: http://www.swi-prolog.org/

6. MySQL: http://www.mysql.com/

7. Oracle: http://www.oracle.com/index.html

8. PhP: http://www.php.net/

9. SWI Prolog: http://www.swi-prolog.org/

10. db4o: http://www.db4o.com/

11. XML: http://www.w3.org/standards/xml/