9.3.10 Exercises for Section 9.3

Exercise 9.3.1: Write the following embedded SQL queries, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1. You may use any host language with which you are familiar, and details of host-language programming may be replaced by clear comments if you wish.

a) Ask the user for the maximum price and minimum values of the speed, RAM, hard disk, and screen size that they will accept. Find all the laptops that satisfy these requirements. Print their specifications (all attributes of `Laptop`) and their manufacturer.

b) Ask the user for a manufacturer, model number, speed, RAM, hard-disk size, and price of a new PC. Check that there is no PC with that model number. Print a warning if so, and otherwise insert the information into tables `Product` and `PC`.

c) Ask the user for a price and find the PC whose price is closest to the desired price. Print the maker, model number, and RAM of the PC.

! d) Ask the user for a manufacturer. Print the specifications of all products by that manufacturer. That is, print the model number, product-type, and all the attributes of whichever relation is appropriate for that type.

!! e) Ask the user for a "budget" (total price of a PC and printer), and a minimum speed of the PC. Find the cheapest "system" (PC plus printer) that is within the budget and minimum speed, but make the printer a color printer if possible. Print the model numbers for the chosen system.

Exercise 9.3.2: Write the following embedded SQL queries, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3.

a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore of the guns. Find the class with the largest firepower.

```
1)  CREATE FUNCTION GetYear(t VARCHAR(255)) RETURNS INTEGER

2)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
3)  DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

    BEGIN
4)      DECLARE EXIT HANDLER FOR Not_Found, Too_Many
5)          RETURN NULL;
6)      RETURN (SELECT year FROM Movies WHERE title = t);
    END;
```

Figure 9.18: Handling exceptions in which a single-row select returns other than one tuple

becomes the return-value. Also, since the handler is an `EXIT` handler, control next passes to the point after the `END`. Since that point is the end of the function, `GetYear` returns at that time, with the return-value `NULL`. □

9.4.8 Using PSM Functions and Procedures

As we mentioned in Section 9.4.2, we can call a PSM procedure anywhere SQL statements can appear, e.g., as embedded SQL, from PSM code itself, or from SQL issued to the generic interface. We invoke a procedure by preceding it by the keyword `CALL`. In addition, a PSM function can be used as part of an expression, e.g., in a `WHERE` clause. Here is an example of how a function can be used within an expression.

Example 9.17: Suppose that our schema includes a module with the function `GetYear` of Fig. 9.18. Imagine that we are sitting at the generic interface, and we want to enter the fact that Denzel Washington was a star of *Remember the Titans*. However, we forget the year in which that movie was made. As long as there was only one movie of that name, and it is in the `Movies` relation, we don't have to look it up in a preliminary query. Rather, we can issue to the generic SQL interface the following insertion:

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES('Remember the Titans', GetYear('Remember the Titans'),
    'Denzel Washington');
```

Since `GetYear` returns `NULL` if there is not a unique movie by the name of *Remember the Titans*, it is possible that this insertion will have `NULL` in the middle component. □

9.4.9 Exercises for Section 9.4

Exercise 9.4.1: Using our running movie database:

Movies(tit
StarsIn(mo
MovieStar
MovieExec
Studio(nar

write PSM proce

a) Given the n
movies from

b) Given the n

c) Given a nar
a movie sta
both, and 0

d) Given an ac
there is exa

! e) Given a star
120 minute
return the y

! f) Given a stu
shortest mo
there is no
is no "secon

Exercise 9.4.2:
the database sche

Product(ma
PC(model,
Laptop(mo
Printer(m

of Exercise 2.4.1.

a) Take a mak
type of pro

b) Take a pric
whose price

! c) Take model
and insert
ready a PC
a key const
to '25000'
model numb

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write PSM procedures or functions to perform the following tasks:

a) Given the name of a star, delete them from `MovieStar` and delete all their movies from `StarsIn` and `Movies`.

b) Given the name of a movie studio, produce the address of its president.

c) Given a name and address, return 1 if the person is an executive but not a movie star, 2 if the person is a movie star but not an executive, 3 if both, and 0 if neither.

d) Given an address, find the name of the unique star with that address if there is exactly one, and return `NULL` if there is none or more than one.

! e) Given a star name, find the most recent (highest year) movie of more than 120 minutes length in which they appeared. If there is no such movie, return the year 0.

! f) Given a studio name, assign to output parameters the titles of the two shortest movies by that studio. Assign `NULL` to one or both parameters if there is no such movie (e.g., if there is only one movie by a studio, there is no "second-longest").

Exercise 9.4.2: Write the following PSM functions or procedures, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1.

a) Take a maker and model as arguments, and return the price of whatever type of product that model is.

b) Take a price as argument and return the model number of the laptop whose price is closest.

! c) Take model, speed, ram, hard-disk, and price information as arguments, and insert this information into the relation `PC`. However, if there is already a PC with that model number (tell by assuming that violation of a key constraint on insertion will raise an exception with `SQLSTATE` equal to `'25000'`), then keep adding 1 to the model number until you find a model number that is not already a PC model number.

! d) Given a price, produce the number of PC's, the number of laptops, and the number of printers selling for less than that price.

Exercise 9.4.3: Write the following PSM functions or procedures, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3.

a) Take as arguments a new class name, type, country, number of guns, bore, and displacement. Add this information to `Classes` and also add the ship with the class name to `Ships`.

b) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore. Given a class, find its firepower.

! c) Given a ship name, determine if the ship was in a battle with a date before the ship was launched. If so, set the date of the battle and the date the ship was launched to -1.

! d) Given the name of a battle, produce the two countries whose ships were involved in the battle. If there are more or fewer than two countries involved, produce `NULL` for both countries.

! **Exercise 9.4.4:** In Fig. 9.15, we used a tricky formula for computing the variance of a sequence of numbers $x_1, x_2, \ldots, x_n$. Recall that the variance is the average square of the deviation of these numbers from their mean. That is, the variance is $\left(\sum_{i=1}^{n}(x_i - \overline{x})^2\right)/n$, where the mean $\overline{x}$ is $\left(\sum_{i=1}^{n} x_i\right)/n$. Prove that the formula for the variance used in Fig. 9.15, which is

$$\left(\sum_{i=1}^{n}(x_i)^2\right)/n - \left(\left(\sum_{i=1}^{n} x_i\right)/n\right)^2$$

yields the same value.

9.5 Using a Call-Level Interface

When using a *call-level interface* (CLI), we write ordinary host-language code, and we use a library of functions that allow us to connect to and access a database, passing SQL statements to that database. The differences between this approach and embedded SQL programming are, in one sense, cosmetic, since the preprocessor replaces embedded SQL by calls to library functions much like the functions in the standard SQL/CLI.

in a loop, then each time around the loop, a new tuple, with a new name and address for a studio, is inserted into `Studio`. □

9.5.5 Exercises for Section 9.5

Exercise 9.5.1: Repeat the problems of Exercise 9.3.1, but write the code in C with CLI calls.

Exercise 9.5.2: Repeat the problems of Exercise 9.3.2, but write the code in C with CLI calls.

9.6 JDBC

Java Database Connectivity, or JDBC, is a facility similar to CLI for allowing Java programs to access SQL databases. The concepts resemble those of CLI, although Java's object-oriented flavor is evident in JDBC.

9.6.1 Introduction to JDBC

The first steps we must take to use JDBC are:

1. include the line:

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

   to make the JDBC classes available to your Java program.

2. Load a "driver" for the database system we shall use. The driver we need depends on which DBMS is available to us, but we load the needed driver with the statement:

   `Class.forName(<driver name>);`

   For example, to get the driver for a MySQL database, execute:

   ```
   Class.forName("com.mysql.jdbc.Driver");
   ```

   The effect is that a class called `DriverManager` is available. This class is analogous in many ways to the environment whose handle we get as the first step in using CLI.

3. Establish a connection to the database. A variable of class Connection is created if we apply the method `getConnection` to `DriverManager`.

The Java statement to establish a connection looks like:

9.6.4 Parameter Passing

As in CLI, we can use a question-mark in place of a portion of a query, and then bind values to those *parameters*. To do so in JDBC, we need to create a prepared statement, and we need to apply to that PreparedStatement object methods such as `setString(`i`, `v`)` or `setInt(`i`, `v`)` that bind the value v, which must be of the appropriate type for the method, to the ith parameter in the query.

Example 9.26: Let us mimic the CLI code in Example 9.21, where we prepared a statement to insert a new studio into relation `Studio`, with parameters for the name and address of that studio. The Java code to prepare this statement, set its parameters, and execute it is shown in Fig. 9.22. We continue to assume that connection object `myCon` is available to us.

```
1)  PreparedStatement studioStat = myCon.prepareStatement(
2)      "INSERT INTO Studio(name, address) VALUES(?, ?)");
    /* get values for variables studioName and studioAddr
       from the user */
3)  studioStat.setString(1, studioName);
4)  studioStat.setString(2, studioAddr);
5)  studioStat.executeUpdate();
```

Figure 9.22: Setting and using parameters in JDBC

In lines (1) and (2), we create and prepare the insertion statement. It has parameters for each of the values to be inserted. After line (2), we could begin a loop in which we repeatedly ask the user for a studio name and address, and place these strings in the variables `studioName` and `studioAddr`. This assignment is not shown, but represented by a comment. Lines (3) and (4) set the first and second parameters to the strings that are the current values of `studioName` and `studioAddr`, respectively. Finally, at line (5), we execute the insertion statement with the current values of its parameters. After line (5), we could go around the loop again, beginning with the steps represented by the comment. □

9.6.5 Exercises for Section 9.6

Exercise 9.6.1: Repeat Exercise 9.3.1, but write the code in Java using JDBC.

Exercise 9.6.2: Repeat Exercise 9.3.2, but write the code in Java using JDBC.

9.7 PHP

PHP is a scripting language for helping to create HTML Web pages. It provides support for database operations through an available library, much as JDBC

9.7.8 Exercises for Section 9.7

Exercise 9.7.1: Repeat Exercise 9.3.1, but write the code using PHP.

Exercise 9.7.2: Repeat Exercise 9.3.2, but write the code using PHP.

! **Exercise 9.7.3:** In Example 9.31 we exploited the feature of PHP that strings in double-quotes have variables expanded. How essential is this feature? Could we have done something analogous in JDBC? If so, how?

9.8 Summary of Chapter 9

- *Three-Tier Architectures*: Large database installations that support large-scale user interactions over the Web commonly use three tiers of processes: web servers, application servers, and database servers. There can be many processes active at each tier, and these processes can be at one processor or distributed over many processors.

- *Client-Server Systems in the SQL Standard*: The standard talks of SQL clients connecting to SQL servers, creating a connection (link between the two processes) and a session (sequence of operations). The code executed during the session comes from a module, and the execution of the module is called a SQL agent.

- *The Database Environment*: An installation using a SQL DBMS creates a SQL environment. Within the environment, database elements such as relations are grouped into (database) schemas, catalogs, and clusters. A catalog is a collection of schemas, and a cluster is the largest collection of elements that one user may see.

- *Impedance Mismatch*: The data model of SQL is quite different from the data models of conventional host languages. Thus, information passes between SQL and the host language through shared variables that can represent components of tuples in the SQL portion of the program.

- *Embedded SQL*: Instead of using a generic query interface to express SQL queries and modifications, it is often more effective to write programs that embed SQL queries in a conventional host language. A preprocessor converts the embedded SQL statements into suitable function calls of the host language.

- *Cursors*: A cursor is a SQL variable that indicates one of the tuples of a relation. Connection between the host language and SQL is facilitated by having the cursor range over each tuple of the relation, while the components of the current tuple are retrieved into shared variables and processed using the host language.