CHAPTER 5

Advanced SQL

Practice Exercises

5.1 Consider the following relations for a company database:

- *emp* (*ename*, *dname*, *salary*)
- *mgr* (*ename*, *mname*)

and the Java code in Figure 5.20, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like "It finds the manager of the toy department," not a line-by-line description of what each Java statement does.)

Answer:
It prints out the manager of "dog," that manager's manager, etc., until we reach a manager who has no manager (presumably, the CEO, who most certainly is a cat). Note: If you try to run this, use your own Oracle ID and password.

5.2 Write a Java method using JDBC metadata features that takes a `ResultSet` as an input parameter and prints out the result in tabular form, with appropriate names as column headings.

Answer:
Please see **??**

5.3 Suppose that we wish to find all courses that must be taken before some given course. That means finding not only the prerequisites of that course, but prerequisites of prerequisites, and so on. Write a complete Java program using JDBC that:

- Takes a *course_id* value from the keyboard.
- Finds prerequisites of that course using an SQL query submitted via JDBC.

```
import java.sql.*;
public class Mystery {
  public static void main(String[] args) {
      try (
          Connection con=DriverManager.getConnection(
              "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
          q = "select mname from mgr where ename = ?";
          PreparedStatement stmt=con.prepareStatement();
      )
      {
          String q;
          String empName = "dog";
          boolean more;
          ResultSet result;
          do {
              stmt.setString(1, empName);
              result = stmt.executeQuery(q);
              more = result.next();
              if (more) {
                  empName = result.getString("mname");
                  System.out.println (empName);
              }
          } while (more);
          s.close();
          con.close();
      }
      catch(Exception e){
          e.printStackTrace();
      }
  }
}
```

Figure 5.20 Java code for Exercise 5.1 (using Oracle JDBC).

- For each course returned, finds its prerequisites and continues this process iteratively until no new prerequisite courses are found.
- Prints out the result.

For this exercise, do not use a recursive SQL query, but rather use the iterative approach described previously. A well-developed solution will be robust to the error case where a university has accidentally created a cycle of prerequisites (that is, for example, course *A* is a prerequisite for course *B*, course *B* is a prerequisite for course *C*, and course *C* is a prerequisite for course *A*).

```
printTable(ResultSet result) throws SQLException {
    metadata = result.getMetaData();
    num_cols = metadata.getColumnCount();
    for(int i = 1; i <= num_cols; i++) {
        System.out.print(metadata.getColumnName(i) + '\t');
    }
    System.out.println();
    while(result.next()) {
        for(int i = 1; i <= num_cols; i++) {
            System.out.print(result.getString(i) + '\t'
        }
        System.out.println();
    } }
```

Figure 5.101 Java method using JDBC for Exercise 5.2.

Answer:
Please see **??**

5.4 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.

Answer:
Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However, not all kinds of queries can be written in SQL. Also, nondeclarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.

5.5 Show how to enforce the constraint "an instructor cannot teach two different sections in a semester in the same time slot." using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).

Answer:
Please see **??**

5.6 Consider the bank database of Figure 5.21. Let us define a view *branch_cust* as follows:

```
import java.sql.*;
import java.util.Scanner;
import java.util.Arrays;
public class AllCoursePrereqs {
  public static void main(String[] args) {
    try (
         Connection con=DriverManager.getConnection
           ("jdbc:oracle:thin:@edgar0.cse.lehigh.edu:1521:cse241","star","pw");
         Statement s=con.createStatement();
         ){
           String q;
           String c;
           ResultSet result;
           int maxCourse = 0;
           q = "select count(*) as C from course";
           result = s.executeQuery(q);
           if (!result.next()) System.out.println ("Unexpected empty result.");
           else maxCourse = Integer.parseInt(result.getString("C"));
           int numCourse = 0, oldNumCourse = -1;
           String[] prereqs = new String [maxCourse];
           Scanner krb = new Scanner(System.in);
           System.out.print("Input a course id (number): ");
           String course = krb.next();
           String courseString = "" + '\'' + course + '\'';
           while (numCourse != oldNumCourse) {
             for (int i = oldNumCourse + 1; i < numCourse; i++) {
               courseString += ", " + '\'' + prereqs[i] + '\'' ;
             }
             oldNumCourse = numCourse;
             q = "select prereq_id from prereq  where course_id in ("
               + courseString + ")";
             result = s.executeQuery(q);
             while (result.next()) {
               c = result.getString("prereq_id");
               boolean found = false;
               for (int i = 0; i < numCourse; i++)
                    found |= prereqs[i].equals(c);
               if (!found) prereqs[numCourse++] = c;
             }
             courseString = "" + '\'' + prereqs[oldNumCourse] + '\'';
           }
           Arrays.sort(prereqs,0,numCourse);
           System.out.print("The courses that must be taken prior to "
             + course + " are: ");
           for (int i = 0; i < numCourse; i++)
                System.out.print ((i==0?" ":", ") + prereqs[i]);
           System.out.println();
         } catch(Exception e){e.printStackTrace();
} }
```

Figure 5.102 Complete Java program using JDBC for Exercise 5.3.

```
create trigger onesec before insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id in (
   select time_slot_id
   from teaches natural join section
   where ID in (
            select ID
            from teaches natural join section
            where sec_id = nrow.sec_id and course_id = nrow.course_id and
                     semester = nrow.semester and year = nrow.year
)))
begin
   rollback
end;


create trigger oneteach before insert on teaches
referencing new row as nrow
for each row
when (exists (
            select time_slot_id
            from teaches natural join section
            where ID = nrow.ID
   intersect
            select time_slot_id
            from section
            where sec_id = nrow.sec_id and course_id = nrow.course_id and
                     semester = nrow.semester and year = nrow.year
))
begin
   rollback
end;
```

Figure 5.103 Trigger code for Exercise 5.5.

```
create view branch_cust as
       select branch_name, customer_name
       from depositor, account
       where depositor.account_number = account.account_number
```

branch (<u>*branch_name*</u>, *branch_city, assets*)
customer (<u>*customer_name*</u>, *customer_street, cust omer_city*)
loan (<u>*loan_number*</u>, *branch_name, amount*)
borrower (<u>*customer_name*</u>, <u>*loan_number*</u>)
account (<u>*account_number*</u>, *branch_name, balance*)
depositor (<u>*customer_name*</u>, <u>*account_number*</u>)

Figure 5.21 Banking database for Exercise 5.6.

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *maintain* the view, that is, to keep it up-to-date on insertions to *depositor* or *account*. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

Answer:
Please see **??**

5.7 Consider the bank database of Figure 5.21. Write an SQL trigger to carry out the following action: On **delete** of an account, for each customer-owner of the

```
create trigger insert_into_branch_cust_via_depositor
after insert on depositor
referencing new row as inserted
for each row
insert into branch_cust
    select branch_name, inserted.customer_name
    from account
    where inserted.account_number = account.account_number

create trigger insert_into_branch_cust_via_account
after insert on account
referencing new row as inserted
for each statement
insert into branch_cust
    select inserted.branch_name, customer_name
    from depositor
    where depositor.account_number = inserted.account_number
```

Figure 5.22 Trigger code for Exercise 5.6.

account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.

Answer:

```
create trigger check-delete-trigger after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
    ( select customer_name from depositor
    where account_number <> orow.account_number )
end
```

5.8 Given a relation *S*(*student*, *subject*, *marks*), write a query to find the top 10 students by total marks, by using SQL ranking. Include all students tied for the final spot in the ranking, even if that results in more than 10 total students.

Answer:

```
select *
from (
    select student, total, rank() over (order by (total) desc) as t_rank
    from (
        select student, sum(marks) as total
        from S group by student
    )
)
where t_rank <= 10
```

5.9 Given a relation *nyse*(*year, month, day, shares_traded, dollar_volume*) with trading data from the New York Stock Exchange, list each trading day in order of number of shares traded, and show each day's rank.

Answer:

```
select year, month, day, shares_traded,
    rank() over (order by shares_traded desc ) as mostshares
from nyse
```

5.10 Using the relation from Exercise 5.9, write an SQL query to generate a report showing the number of shares traded, number of trades, and total dollar volume broken down by year, each month of each year, and each trading day.

Answer:

```
select year, month, day, sum(shares_traded) as shares,
        sum(num_trades) as trades, sum(dollar_volume) as total_volume
from nyse
group by rollup (year, month, day)
```

5.11 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.

Answer:

groupby rollup(a), **rollup**(b), **rollup**(c), **rollup**(d)