CHAPTER 5

Advanced SQL

In this chapter we address the issue of how to access SQL from a general-purpose programming language, which is very important for building applications that use a database to store and retrieve data. We describe how procedural code can be executed within the database, either by extending the SQL language to support procedural actions, or by allowing functions defined in procedural languages to be executed within the database. We describe triggers, which can be used to specify actions that are to be carried out automatically on certain events such as insertion, deletion, or update of tuples in a specified relation. We discuss recursive queries and advanced aggregation features supported by SQL. Finally, we describe online analytic processing (OLAP) systems, which support interactive analysis of very large datasets.

Given the fact that the JDBC and ODBC protocols (and variants such as ADO.NET) are have become the primary means of accessing databases, we have significantly extended our coverage of these two protocols, including some examples. However, our coverage is only introductory, and omits many details that are useful in practise. Online tutorials/manuals or textbooks covering these protocols should be used as supplements, to help students make full use of the protocols.

Exercises

5.12 Consider the following relations for a company database:

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

and the Java code in Figure 5.26, 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.)

```
import java.sql.*;
public class Mystery {
   public static void main(String[] args) {
      try {
         Connection con=null;
         Class.forName("oracle.jdbc.driver.OracleDriver");
         con=DriverManager.getConnection(
            "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
         Statement s=con.createStatement();
         String q;
         String empName = "dog";
         boolean more;
         ResultSet result;
         do {
            q = "select mname from mgr where ename = '" + empName + "'";
            result = s.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 5.26 Java code for Exercise 5.12.

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, since Star, crafty cat that she is, changes her password.

5.13 Suppose you were asked to define a class MetaDisplay in Java, containing a method static void printTable(String r); the method takes a relation name *r* as input, executes the query "**select** * **from** *r*", and prints the result out in nice tabular format, with the attribute names displayed in the header of the table.

a. What do you need to know about relation *r* to be able to print the result in the specified tabular format.

b. What JDBC methods(s) can get you the required information?

c. Write the method printTable(String r) using the JDBC API.

Answer:

a. We need to know the number of attributes and names of attributes of r to decide the number and names of columns in the table.

b. We can use the JDBC methods getColumnCount() and getColumnName(int) to get the required information.

c. The method is shown below.

```
static void printTable(String r)
{
    try
    {
        Class.forName("oraclejdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:univdb",user,passwd);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.ExecuteQuery(r);
        ResultSetMetaData rsmd = rs.getMetaData();
        int count = rsmd.getColumnCount();
        System.out.println("<tr>");
        for(int i=1;i<=count;i++){
            System.out.println("<td>"+rsmd.getColumnName(i)+"</td>");
        }
        System.out.println("</tr>");
        while(rs.next(){
            System.out.println("<tr>");
            for(int i=1;i<=count;i++){
                System.out.println("<td>"+rs.getString(i)+"</td>");
            }
            System.out.println("</tr>");
        }
        stmt.close();
        conn.close();
    }
    catch(SQLException sqle)
    {
        System.out.println("SQLException : " + sqle);
    }
}
```

5.14 Repeat Exercise 5.13 using ODBC, defining void printTable(char *r) as a function instead of a method.

Answer:

a. Same as for JDBC.

b. The function SQLNumResultCols(hstmt, &numColumn) can be used to find the number of columns in a statement, while the function SQLColAttribute() can be used to find the name, type and other information about any column of a result set. set, and the names

c. The ODBC code is similar to the JDBC code, but significantly longer. ODBC code that carries out this task may be found online at the URL http://msdn.microsoft.com/en-us/library/ms713558.aspx (look at the bottom of the page).

5.15 Consider an employee database with two relations

employee (<u>*employee_name*</u>, *street*, *city*)
works (<u>*employee_name*</u>, *company_name*, *salary*)

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

a. Using SQL functions as appropriate.

b. Without using SQL functions.

Answer:

a.

```
create function avg_salary(cname varchar(15))
    returns integer
    declare result integer;
        select avg(salary) into result
        from works
        where works.company_name = cname
    return result;
end
select company_name
from works
where avg_salary(company_name) > avg_salary("First Bank Corporation")
```

b.

```
select company_name
from works
group by company_name
having avg(salary) > (select avg(salary)
                      from works
                      where company_name="First Bank Corporation")
```

5.16 Rewrite the query in Section 5.2.1 that returns the name and budget of all departments with more than 12 instructors, using the **with** clause instead of using a function call.

Answer:

```
with instr_count (dept_name, number) as
    (select dept_name, count (ID)
        from instructor
        group by dept_name)
select dept_name, budget
    from department, instr_count
    where department.dept_name = instr_count.dept_name
    and number > 12
```

5.17 Compare the use of embedded SQL with the use in SQL of functions defined in a general-purpose programming language. Under what circumstances would you use each of these features?

Answer: SQL functions are primarily a mechanism for extending the power of SQL to handle attributes of complex data types (like images), or to perform complex and non-standard operations. Embedded SQL is useful when imperative actions like displaying results and interacting with the user are needed. These cannot be done conveniently in an SQL only environment. Embedded SQL can be used instead of SQL functions by retrieving data and then performing the function's operations on the SQL result. However a drawback is that a lot of query-evaluation functionality may end up getting repeated in the host language code.

5.18 Modify the recursive query in Figure 5.15 to define a relation

$$prereq\_depth(course\_id, prereq\_id, depth)$$

where the attribute *depth* indicates how many levels of intermediate prerequisites are there between the course and the prerequisite. Direct prerequisites have a depth of 0.

Answer:

```
with recursive prereq_depth(course_id, prereq_id, depth) as
    (select course_id, prereq_id, 0
        from prereq
    union
    select prereq.course_id, prereq_depth.prereq_id, (prereq_depth.depth + 1)
        from prereq, prereq_depth
        where prereq.prereq_id= prereq_depth.course_id)

select *
from prereq_depth
```

5.19 Consider the relational schema

part(<u>*part_id*</u>, *name*, *cost*)
subpart(<u>*part_id*</u>, <u>*subpart_id*</u>, *count*)

A tuple (p_1, p_2, 3) in the *subpart* relation denotes that the part with part-id p_2 is a direct subpart of the part with part-id p_1, and p_1 has 3 copies of p_2. Note that p_2 may itself have further subparts. Write a recursive SQL query that outputs the names of all subparts of the part with part-id "P-100".
Answer:

```
with recursive total_part(name) as
    (select part.name
        from subpart, part
        where subpart.part_id = "P-100" and
        subpart.part_id = part.part_id
    union
    select p2.name
        from subpart s, part p1, part p2
        where s.part_id = p1.part_id
        and p1.name total_part.name
        and s.subpart_id = p2.part_id)

    select *
        from total_part
```

5.20 Consider again the relational schema from Exercise 5.19. Write a JDBC function using non-recursive SQL to find the total cost of part "P-100", including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.
Answer: The SQL function 'total_cost' is called from within the JDBC code.
SQL function:

```
create function total_cost(id char(10))

returns table(number integer)

begin
  create temporary table result (name char(10), number integer);
  create temporary table newpart (name char(10), number integer);
  create temporary table temp (name char(10), number integer);
  create temporary table final_cost(number integer);

  insert into newpart
    select subpart_id, count
```

```
        from subpart
        where part_id = id
    repeat
        insert into result
        select name, number
        from newpart;

        insert into temp
        (select subpart.subpart_id, count
            from newpart, subpart
            where newpart.subpart_id = subpart.part_id;
        )
        except(
            select subpart_id, count
                from result;
            );

        delete from newpart;
        insert into newpart
            select *
            from temp;
        delete from temp;

    until not exists(select * from newpart)
    end repeat;

    with part_cost(number) as
        select (count*cost)
            from result, part
            where result.subpart_id = part.part_id);
    insert into final_cost
        select *
        from part_cost;
    return table final_cost;
end
```

JDBC function:

```
Connection conn = DriverManager.getConnection(
    ”jdbc:oracle:thin:@db.yale.edu:2000:bankdb”,
    userid,passwd);
Statement stmt = conn.createStatement();

ResultSet rset = stmt.executeQuery(
    "SELECT SUM(number) FROM TABLE(total_cost("P-100"))");

System.out.println(rset.getFloat(2));
```

5.21 Suppose there are two relations r and s, such that the foreign key B of r references the primary key A of s. Describe how the trigger mechanism can be used to implement the **on delete cascade** option, when a tuple is deleted from s.

Answer: We define triggers for each relation whose primary-key is referred to by the foreign-key of some other relation. The trigger would be activated whenever a tuple is deleted from the referred-to relation. The action performed by the trigger would be to visit all the referring relations, and delete all the tuples in them whose foreign-key attribute value is the same as the primary-key attribute value of the deleted tuple in the referred-to relation. These set of triggers will take care of the **on delete cascade** operation.

5.22 The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.

Answer: It is possible that a trigger body is written in such a way that a non-terminating recursion may result. An example of such a trigger is a *before insert* trigged on a relation that tries to insert another record into the same relation.

In general, it is extremely difficult to statically identify and prohibit such triggers from being created. Hence database systems, at runtime, put a limit on the depth of nested trigger calls.

5.23 Consider the relation, r, shown in Figure 5.27. Give the result of the following query:

> **select** *building*, *room_number*, *time_slot_id*, **count**(*)
> **from** r
> **group by rollup** (*building*, *room_number*, *time_slot_id*)

Answer:

Garfield	359	P	1
Garfield	359	*null*	1
Garfield	*null*	*null*	1
Painter	705	N	1
Painter	705	*null*	1
Painter	*null*	*null*	1
Saucon	550	D	1
Saucon	550	*null*	1
Saucon	651	N	1
Saucon	651	*null*	1
Saucon	*null*	*null*	2

5.24 For each of the SQL aggregate functions **sum, count, min**, and **max**, show how to compute the aggregate value on a multiset $S_1 \cup S_2$, given the aggregate values on multisets S_1 and S_2.

On the basis of the above, give expressions to compute aggregate values with grouping on a subset S of the attributes of a relation $r(A, B, C, D, E)$, given aggregate values for grouping on attributes $T \supseteq S$, for the following aggregate functions:

a. **sum, count, min,** and **max**

b. **avg**

c. Standard deviation

Answer: Given aggregate values on multisets S_1 and S_2, we can calculate the corresponding aggregate values on multiset $S_1 \cup S_2$ as follows:

- $\textbf{sum}(S_1 \cup S_2) = \textbf{sum}(S_1) + \textbf{sum}(S_2)$
- $\textbf{count}(S_1 \cup S_2) = \textbf{count}(S_1) + \textbf{count}(S_2)$
- $\textbf{min}(S_1 \cup S_2) = \textbf{min}(\textbf{min}(S_1), \textbf{min}(S_2))$
- $\textbf{max}(S_1 \cup S_2) = \textbf{max}(\textbf{max}(S_1), \textbf{max}(S_2))$

Let the attribute set $T = (A, B, C, D)$ and the attribute set $S = (A, B)$. Let the aggregation on the attribute set T be stored in table *aggregation_on_t* with aggregation columns *sum_t*, *count_t*, *min_t*, and *max_t* storing **sum, count, min** and **max** resp.

a. The aggregations *sum_s*, *count_s*, *min_s*, and *max_s* on the attribute set S are computed by the query:

select *A*, *B*, **sum**(*sum_t*) **as** *sum_s*, **sum**(*count_t*) **as** *count_s*,
  min(*min_t*) **as** *min_s*, **max**(*max_t*) **as** *max_s*
from *aggregation_on_t*
groupby *A*, *B*

b. The aggregation *avg* on the attribute set S is computed by the query:

select *A*, *B*, **sum**(*sum_t*)/**sum**(*count_t*) **as** *avg_s*
from *aggregation_on_t*
groupby *A*, *B*

c. For calculating standard deviation we use an alternative formula:

$$stddev(S) = \frac{\sum_{s \in S} s^2}{|S|} - avg(S)^2$$

which we get by expanding the formula

$$stddev(S) = \frac{\sum_{s \in S} (s^2 - avg(S))^2}{|S|}$$

If S is partitioned into n sets $S_1, S_2, \ldots S_n$ then the following relation holds:

$$stddev(S) = \frac{\sum_{S_i} |S_i|(stddev(S_i)^2 + avg(S_i)^2)}{|S|} - avg(S)^2$$

Using this formula, the aggregation **stddev** is computed by the query:

select *A*, *B*,
(**sum**(*count_t* ∗ (*stddev_t***stddev_t*+ *avg_t** *avg_t*))/**sum**(*count_t*)) -
(**sum**(*sum_t*)/**sum**(*count_t*))
from *aggregation_on_t*
groupby *A*, *B*

5.25 In Section 5.5.1, we used the *student_grades* view of Exercise 4.5 to write a query to find the rank of each student based on grade-point average. Modify that query to show only the top 10 students (that is, those students whose rank is 1 through 10).

Answer:

```
with s_grades as
    select ID,rank() over (order by (GPA)desc) as s_rank
    from student_grades
select ID,s_rank
from s_grades
where s_rank <= 10
```

5.26 Give an example of a pair of groupings that cannot be expressed by using a single **group by** clause with **cube** and **rollup**.

Answer: Consider an example of hierarchies on dimensions from Figure 5.19. We can not express a query to seek aggregation on groups (*City*, *Hour of day*) and (*City*, *Date*) using a single **group by** clause with **cube** and **rollup**.

Any single **groupby** clause with **cube** and **rollup** that computes these two groups would also compute other groups also.

5.27 Given relation $s(a, b, c)$, show how to use the extended SQL features to generate a histogram of c versus a, dividing a into 20 equal-sized partitions (that is, where each partition contains 5 percent of the tuples in s, sorted by a).

Answer:

```
select tile20, sum(c)
from (select c, ntile(20) over (order by (a)) as tile20
      from r) as s
groupby tile20
```

5.28 Consider the bank database of Figure 5.25 and the *balance* attribute of the *account* relation. Write an SQL query to compute a histogram of *balance* values, dividing the range 0 to the maximum account balance present, into three equal ranges.

Answer:

```
(select 1, count(*)
 from account
 where 3* balance <= (select max(balance)
                      from account)
)
union
(select 2, count(*)
 from account
 where 3* balance > (select max(balance)
                     from account)
       and 1.5* balance <= (select max(balance)
                            from account)
)
union
(select 3, count(*)
 from account
 where 1.5* balance > (select max(balance)
                       from account)
)
```