CHAPTER 5

Advanced SQL

Practice Exercises

5.1 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.

Embedded SQL has the advantage of programs being less complicated since it avoids the clutter of the ODBC or JDBC function calls, but requires a specialized preprocessor.

5.2 Write a Java function 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:

```
public class ResultSetTable implements TabelModel {
    ResultSet result;
    ResultSetMetaData metadata;
    int num_cols;

    ResultSetTable(ResultSet result) throws SQLException {
        this.result = result;
        metadata = result.getMetaData();
        num_cols = metadata.getColumnCount();

        for(int i = 1; i <= num_cols; i++) {
            System.out.print(metadata.getColumnName(i) + ‘‘ ‘‘);
            }
            System.out.println();
            while(result.next()) {
                for(int i = 1; i <= num_cols; i++) {
                    System.out.print(result.getString(
                        metadata.getColumnName(i) + '' ''));
                }
                System.out.println();
            }
        }
    }
```

5.3 Write a Java function using JDBC metadata features that prints a list of all relations in the database, displaying for each relation the names and types of its attributes.
Answer:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getTables();
while (rs.next()) {
    System.out.println(rs.getString(''TABLE_NAME'');
    ResultSet rs1 = dbmd.getColumns(null, ''schema-name'',
                rs.getString(''TABLE_NAME''), ''%'');
    while (rs1.next()) {
        System.out.println(rs1.getString(''COLUMN_NAME''),
                rs.getString(''TYPE_NAME'');
    }
}
```

5.4 Show how to enforce the constraint "an instructor cannot teach in two different classrooms 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: FILL

5.5 Write triggers to enforce the referential integrity constraint from *section* to *time_slot*, on updates to *section*, and *time_slot*. Note that the ones we wrote in Figure 5.8 do not cover the **update** operation.
Answer: FILL

5.6 To maintain the *tot_cred* attribute of the *student* relation, carry out the following:

a. Modify the trigger on updates of *takes*, to handle all updates that can affect the value of *tot_cred*.

b. Write a trigger to handle inserts to the *takes* relation.

c. Under what assumptions is it reasonable not to create triggers on the *course* relation?

Answer: FILL

5.7 Consider the bank database of Figure 5.25. Let us define a view *branch_cust* as follows:

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

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 and deletions from *depositor* or *account*. Do not bother about updates.
Answer: For inserting into the materialized view *branch_cust* we must set a database trigger on an insert into *depositor* and *account*. We assume that the database system uses *immediate* binding for rule execution. Further, assume that the current version of a relation is denoted by the relation name itself, while the set of newly inserted tuples is denoted by qualifying the relation name with the prefix – **inserted**.
The active rules for this insertion are given below –

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

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

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch_cust*.
The deletion of a tuple from *branch_cust* is similar to insertion, exce pt that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly

deleted set of tuples by qualifying the relation name with the keyword **deleted**.

```
define trigger delete_from_branch_cust_via_depositor
after delete on depositor
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from deleted, account
    where deleted.account_number = account.account_number

define trigger delete_from_branch_cust_via_account
after delete on account
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from depositor, deleted
    where depositor.account_number = deleted.account_number
```

5.8 Consider the bank database of Figure 5.25. Write an SQL trigger to carry out the following action: On **delete** of an account, for each owner of the 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
```

5.9 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)

5.10 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.
Answer: We assume that multiple students do not have the same marks since otherwise the question is not deterministic; the query below deterministically returns all students with the same marks as the n student, so it may return more than n students.

```
select student, sum(marks) as total,
        rank() over (order by (total) desc ) as trank
from S
groupby student
having trank ≤ n
```

5.11 Consider the *sales* relation from Section 5.6. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 5.21. Do not use the **cube** construct.

Answer:

```
(select color, size, sum(number)
 from sales
 groupby color, size
)
union
(select color, 'all', sum(number)
 from sales
 groupby color
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', 'all', sum(number )
 from sales
)
```