Chapter 5: Advanced SQL

Database System Concepts, 7th Ed.

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers
- ※ Recursive Queries
- ※ Advanced Aggregation Features

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative 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.

Accessing SQL from a Programming Language (Cont.)

There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- 可參考課本圖5.5的ODBC範例。
- 在Windows裡，有 “ODBC資料來源管理員”，可看到電腦裡已經灌好的driver library。

JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- ※ Callable Statements
- ※ Metadata Features
- ※ Other Features
- ※ Database Access from Python

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors
- 以下的程式範例必須先import 「java.sql.*」.

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
    {
     try (Connection conn = DriverManager.getConnection(
           "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
           Statement stmt = conn.createStatement();
         )
     {
           … Do Actual Work ….
     }
     catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
     }
}
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
Resources opened in “try (….)” syntax (“try with resources”) are automatically closed at the end of the try block

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
                        "select dept_name, avg (salary)
                         from instructor
                         group by dept_name");
while (rset.next()) {
      System.out.println(rset.getString("dept_name") + " " +
                                 rset.getFloat(2));
}
```

dept_name	*avg(salary)*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

JDBC Code Details

- Getting result fields:
  - **rs.getString(“dept_name”) and rs.getString(1) equivalent if dept_name is the first argument of select result.**
- Dealing with Null values

  int a = rs.getInt(“a”);

  if (rs.wasNull()) Systems.out.println(“Got null value”);

Prepared Statement

- We can create a prepared statement in which some values are replaced by "?", thereby specifying that actual values will be provided later.
- The database system compiles the query when it is **prepared**. Each time the query is **executed**, the database system reuses the compiled form of the query and applies the new values as parameters.
- Prepared statements allow for more **efficient** execution in cases where the same query can be compiled once and then run multiple times with different parameter values.
- Example:
  - **PreparedStatement pStmt = conn.prepareStatement(**
    "insert into instructor values(?,?,?,?)");
    pStmt.setString(1, "88877");
    pStmt.setString(2, "Perry");
    pStmt.setString(3, "Finance");
    pStmt.setInt(4, 125000);
    pStmt.executeUpdate();
    pStmt.setString(1, "88878");
    pStmt.executeUpdate();

Prepared Statement (Cont.)

- Example:
  - **“insert into instructor values(‘ ” + ID + “ ’, ‘ ” + name + “ ’, ‘ “ + dept_ name + ” ‘,” + balance + “)”**
- What if name is “Perry”?
  - “insert into instructor values(‘ ” + “88877” + " ', ' " + “Perry” + “ ’, ‘ “ + …

  ⇒ insert into instructor values(‘88877 ', ‘Perry’,’…….
- What if name is “D'Souza”?
  - “insert into instructor values(‘ ” + “88877” + " ', ' " + “D’Souza” + “ ’, ‘ “ + …

  ⇒ insert into instructor values(‘88877 ', ' D’Souza ’,’……

    單引號配對出錯!!
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - 因為單引號會失去特殊功能 (看下一頁)
  - NEVER create a query by concatenating strings

SQL Injection

- Suppose query is constructed using
  - **“select * from instructor where name = ‘” + name + “ ‘ "**
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = 'Y
- then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = ‘X’ or ‘Y’ = ‘Y‘ =>看到全部
  - User could have even used
    - X‘; update instructor set salary = salary + 10000; -- => 更改資料
- Prepared statement internally uses:
  “select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - **Always use prepared statements, with user inputs as parameters**
- SQL injection 實例:
  https://news.ltn.com.tw/news/life/breakingnews/2971370

補充PDO (PHP) prepared statements example

```
<?php
 //連接資料庫
 $db = new
PDO('mysql:host=localhost;dbname=test;charset=utf8',$user,$password);
 //設定想要新增入資料庫的資料內容如下
            $dept = "資工";
            $no = "00557888";
            $name = "發發發";
            $club = "桌遊社";
 //設定要使用的SQL指令
            $query = ("insert into student values(?,?,?,?)");
            $stmt = $db->prepare($query);
 //執行SQL語法
            $result = $stmt->execute(array($dept,$no,$name,$club));
?>
```

補充 PDO – query example

```
<?php
        include "db_conn.php";
        $no = "00557888";
        $query = ("select * from student where no = ?");
        $stmt =  $db->prepare($query);
        $error = $stmt->execute(array($no));
        $result = $stmt->fetchAll();
        for($i=0; $i<count($result); $i++){
        echo "dept:".$result [$i]['dept'].' '.
                "no:". $result [$i]['no'].' '.
                "name:".$result[$i]['name'].' '.
                "club:". $result[$i]['club'].' '.
                '<br>';
    }
```

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

※ Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;

  Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

Functions and Procedures

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
    returns integer
    begin
    declare d_count  integer;
        select count (* ) into d_count
        from instructor
        where instructor.dept_name = dept_name;
    return d_count;
    end
```

instructor (ID, name, dept_name, salary)
department (dept_name, building, budget)

- The function *dept_*count can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12;
```

- 函數可直接用在select和where裡。

※ Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
        returns table  (
                ID varchar(5),
                 name varchar(20),
                dept_name varchar(20),
                 salary numeric(8,2))
        return table
                (select ID, name, dept_name, salary
                 from instructor
                 where instructor.dept_name = instructor_of.dept_name);
```

- Usage

```
select *
from table (instructor_of ('Music'));
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
  begin
     select count(*) into d_count
     from instructor
     where instructor.dept_name = dept_count_proc.dept_name
  end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

- 補充: SQL Server和MySQL支援stored procedure.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin** … **end,**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- While and repeat statements:
  - **while** boolean expression **do**
    sequence of statements ;
    end while
  - **repeat**
    sequence of statements ;
    until boolean expression
    end repeat

※ Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
      select budget from department
do
            set n = n + r.budget
end for
```

Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

  if *boolean expression*
  then *statement or compound statement*
  elseif *boolean expression*
  then *statement or compound statement*
  else *statement or compound statement*
  end if

- 複雜的例子請參考課本圖5.8
- ※ MariaDB function: https://mariadb.com/kb/en/create-function/
- ※ MariaDB procedure: https://mariadb.com/kb/en/create-procedure/

※ External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out count integer)
language C
external name   '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

- Benefit: efficiency; drawback: security

Triggers

Triggers (觸發程序)

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to a relation or a specific attribute; e.g.,
  - **after update on** *takes* <- 較多軟體支援
  - **after update of** *grade* **on** *takes*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update on takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
    begin atomic
        set nrow.grade = null;
    end;
```

} declare a transaction

Trigger Example

- E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
    rollback
end;
```

- 完整範例請參考課本圖5.9

section (course_id, sec_id, semester, year, building, room_number, time_slot_id)
time_slot (time_slot_id, day, start_time, end_time)

練習

- Write triggers to enforce the referential integrity constraint from *section* to *time_slot*, on deletion to *time_slot*.

(hint: You need to check that the *time_slot_id* of the tuple being deleted is either still present in *time_slot*, or that no tuple in *section* contains that particular time_slot_id value; otherwise, referential integrity would be violated.)

Answer:

Trigger to Maintain credits_earned value

```
create trigger credits_earned after update of grade on takes
referencing new row as nrow
referencing old row as orow
for each row
when (nrow.grade <> 'F' and nrow.grade is not null)
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
        (select credits
         from course
         where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

takes

ID	course_id	sec_id	semester	year	grade
1	CS-101	1	Spring	2010	F
					A

orow

nrow

student

ID	name	dept_name	tot_cred
1	John	Comp. Sci.	100
			103

course

course_id	..	..	credits
CS-101			3

ERROR 1054

- Two tables: animals (ID, name); animal_count (animals);
- CREATE TRIGGER increment_animal AFTER INSERT ON animals
  FOR EACH ROW
  UPDATE animal_count SET animal_count.animals = animal_count.animals+1;
- CREATE TRIGGER the_mooses_are_loose AFTER INSERT ON animals
  FOR EACH ROW
  BEGIN
  IF NEW.name = 'Moose' THEN
  UPDATE animal_count SET
  animal_count.animals = animal_count.animals+100;
  ELSE
  UPDATE animal_count SET
  animal_count.animals = animal_count.animals+1;
  END IF;
  END;
- 若判斷有錯誤發生可寫 SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =‘……’;
- 參考網址 : https://mariadb.com/kb/en/trigger-overview/

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

※ 補充MS-SQL 的Trigger範例

This example creates a trigger that, when an employee job level is inserted or updated, checks that the specified employee job level (**job_lv**) is within the range defined for the job. To get the appropriate range, the **jobs** table must be referenced.

```
CREATE TRIGGER employee_insupd ON employee FOR INSERT, UPDATE AS
/* Get the range of level for this job type from the jobs table. */
DECLARE @min_lv tinyint, @max_lv tinyint,  @emp_lv tinyint,  @job_id smallint
SELECT @min_lv = min_lv,  @max_lv = max_lv,  @emp_lv = i.job_lv,  @job_id = i.job_id
FROM employee e INNER JOIN inserted i ON e.emp_id = i.emp_id INNER JOIN jobs j ON
    j.job_id = i.job_id
IF (@job_id = 1) and (@emp_lv <> 10)
BEGIN
   RAISERROR ('Job id 1 expects the default level of 10.')
   ROLLBACK TRANSACTION
END
ELSE
IF NOT (@emp_lv BETWEEN @min_lv AND @max_lv)
BEGIN
  RAISERROR ('The level for job_id:%d should be between %d and %d.', @job_id, @min_lv,
    @max_lv)
  ROLLBACK TRANSACTION
END
```

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication