Database Systems

Lecture 11:
Advanced SQL (part 1)

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Outline

- **Accessing SQL From a Programming Language**
- Functions
- Triggers
- Advanced Aggregation Features
- OLAP

Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables

Accessing SQL From a Programming Language

- Possible approaches:
  - Dynamic SQL
    - JDBC (Java Database Connectivity) works with Java
    - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic.
      - Other API's such as ADO.NET sit on top of ODBC
  - Embedded SQL

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.

JDBC

- 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

JDBC Example

```
public static void JDBCexample(String userid, String passwd)
{
        try
        {
                Class.forName ("oracle.jdbc.driver.OracleDriver");
                Connection conn = DriverManager.getConnection(
                                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                                userid, passwd);
                Statement stmt = conn.createStatement();
                try {
                        stmt.executeUpdate(
                                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
                } catch (SQLException sqle)
                {
                        System.out.println("Could not insert tuple. " + sqle);
                }
                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));
                }
                stmt.close();
                conn.close();
        }
        catch (Exception sqle)
        {
                System.out.println("Exception : " + sqle);
        }
}
```

JDBC Example

```
public static void JDBCexample(String userid, String passwd)
{
        try
        {
                Class.forName ("oracle.jdbc.driver.OracleDriver");
                Connection conn = DriverManager.getConnection(
                                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                                userid, passwd);
                Statement stmt = conn.createStatement();
                try {
                        stmt.executeUpdate(
                                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
                } catch (SQLException sqle)
                {
                        System.out.println("Could not insert tuple. " + sqle);
                }
```

JDBC Example

```
        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));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

Database Connection

- Each database product that supports JDBC provides a JDBC driver that must be dynamically loaded in order to access the database from Java.
  - This is done by invoking *Class.forName* with one argument specifying a concrete class implementing the java.sql.Driver interface
- Connecting to the Database: A connection is opened using the getConnection method of the DriverManager class (within java.sql) using 3 parameters:
  - a string that specifies the URL, or machine name, where the server runs
  - a database user identifier, which is a string
  - a password, which is also a string.
    - Note: the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.

Shipping SQL Statements

- Methods for executing a statement:
  - *executeQuery*
    - When the SQL statement is a query
    - It returns a result set
  - *executeUpdate*
    - When the SQL statement is nonquery (DDL or DML)
      - Update
      - Insert
      - Delete
      - Create table
      - …
    - It returns an integer giving the number of tuples inserted, updated, or deleted.
    - For DDL statements, the return value is zero.

Retrieving the Results of a Query

- Retrieving the set of tuples in the result into a ResultSet object
- Fetching the results one tuple at a time
- Using the *next* method on the result set to test whether there remains at least one unfetched tuple in the result set and if so, fetches it.
- Attributes from the fetched tuple are retrieved using various methods whose names begin with *get*
  - *getString:* can retrieve any of the basic SQL data types
  - *getFloat*
- Possible argument to the *get* methods
  - The attribute name specified as a string
  - An integer indicating the position of the desired attribute within the tuple

Database Connection

- The statement and connection are both closed at the end of the Java program.
- It is important to close the connection because there is a limit imposed on the number of connections to the database
- Unclosed connections may cause that limit to be exceeded.
- If this happens, the application cannot open any more connections to the database.

Prepared Statements

- Creating a prepared statement in which some values are replaced by “?”
- Specifying that actual values will be provided later
- Compiling the query by the database system when it is prepared
- Reusing the previously compiled form of the query and apply the new values whenever the query is executed
  - (with new values to replace the “?”s),

Prepared Statements

```
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();
```

Figure 5.2 Prepared statements in JDBC code.

- insert into instructor values (“88877”, “Perry”, “Finance”, 125000)
- insert into instructor values (“88878”, “Perry”, “Finance”, 125000)

Prepared Statements

- Prepared statements allow for more efficient execution
  - Where the same query can be compiled once and then run multiple times with different parameter values
  - Whenever a user-entered value is used, even if the query is to be run only once

Metadata Features

- Capturing metadata about
  - Database
  - ResultSet (relations)

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
        System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnTypeName(i));
}
```

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.

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**
- The SQL structures permitted in the host language comprise *embedded* SQL
- An embedded SQL program must be processed by a special preprocessor prior to compilation.
- The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses.
- Then, the resulting program is compiled by the host-language compiler.

Embedded SQL

- **EXEC SQL** statement is used 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 { …. };

Database Connection

- Before executing any SQL statements, the program must first connect to the database. This is done using:

EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

- Here, *server* identifies the server to which a connection is to be established.

Variables

- Variables of the host language can be used within embedded SQL statements.
- They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount)*

- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

```
EXEC-SQL BEGIN DECLARE SECTION;
        int  credit-amount ;
EXEC-SQL END DECLARE SECTION;
```

SQL Query

- To write an embedded SQL query, we use the following statement:

  declare *c* **cursor for <SQL query>**

- The variable *c* is used to identify the query

- Example:

```
EXEC SQL
        declare c cursor for
            select ID, name
            from student
            where tot_cred > :credit_amount
END_EXEC
```

SQL Query

- The open statement is then used to evaluate the query
- The open statement for our example is as follows:

  EXEC SQL **open** *c* ;

- This statement causes the database system to execute the query and to save the results within a temporary relation.
- The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

SQL Query

- The fetch statement
  - Placing the values of one tuple in the query result into host language variables
  - Requiring one host-language variable for each attribute of the result relation;

    e.g., we need one variable to hold the ID value (si) and another to hold the name value (sn) which have been declared within a DECLARE section

```
EXEC SQL
        fetch c into :si, :sn
END_EXEC
```

Repeated calls to fetch get successive tuples in the query result

SQL Query

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* ;

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)

- Example for updating tuples fetched by cursor by declaring that the cursor is for update

```
EXEC SQL
    declare c cursor for
        select *
        from instructor
        where dept_name = 'Music'
        for update;
```

Updates Through Embedded SQL

- Iterating through the tuples by performing **fetch** operations on the cursor
- Executing the following statement

```
EXEC SQL
    update instructor
        set salary = salary + 1000
        where current of c;
```

Outline

- Accessing SQL From a Programming Language
- **Functions**
- Triggers
- Advanced Aggregation Features
- OLAP

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

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
```

SQL Functions

- The function *dept_*count can be used in a query that returns names and budget of all departments with more that 12 instructors.

select *dept_name, budget*
from *department*
where *dept_*count (*dept_name*) > 12

SQL Functions

- Compound statement: **begin … end**
  - May contain multiple SQL statements between **begin** and **end**.

- **returns** -- indicates the variable-type that is returned (e.g., integer)

- **return** **--** specifies the values that are to be returned as result of invoking the function

- SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters.

Table Functions

- SQL:2003 added functions that return a relation as a result
- 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'))
```

The picture can't be displayed.

Questions?