Database Systems
Lecture07 – JDBC and PL/SQL

Beomseok Nam (남범석)
bnam@skku.edu

Using SQL in Other Programming Languages

- Need to use a general-purpose programming languages (e.g., C/C++/Java) along with SQL
  - Not all queries can be expressed in SQL
    - Some queries can be written more easily with general-purpose programming languages
  - Non-declarative actions cannot be done in SQL
    - e.g., printing a report
    - interacting with a user
    - sending the results of a query to a GUI

JDBC and ODBC

- API for a program to interact with a database server
- Applications make calls to
  - Connect with the database server
  - Send SQL queries to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

JDBC

- JDBC is a Java API for SQL.
- Model for communicating with the database:
  - [1] Open a connection
  - [2] Create a “Statement” object
  - [3] Execute queries using the “Statement” object to send queries and fetch results
  - [4] Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid,
                               String passwd)
{
  try {
   Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://localhost/db_name",
             userid, passwd);
    Statement stmt = conn.createStatement();
        /*… Do Actual Work … shown in the next slide */
    stmt.close();
    conn.close();
  }
  catch (SQLException sqle) {
    System.out.println("SQLException : " + sqle);
  }
}
```

To change your psql password, run the following stmt in psql
ALTER USER your_userid WITH PASSWORD your_password;

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));
}
```

JDBC Code Details

- Getting result fields:
  - **`rset.getString(“dept_name”)`** and **`rset.getString(1)`** are equivalent if dept_name is the first argument of select result.

- Dealing with Null values

  -
    ```
    if (rset.wasNull())
        Systems.out.println(“Got null value”);
    ```

Warning: Statement is not safe

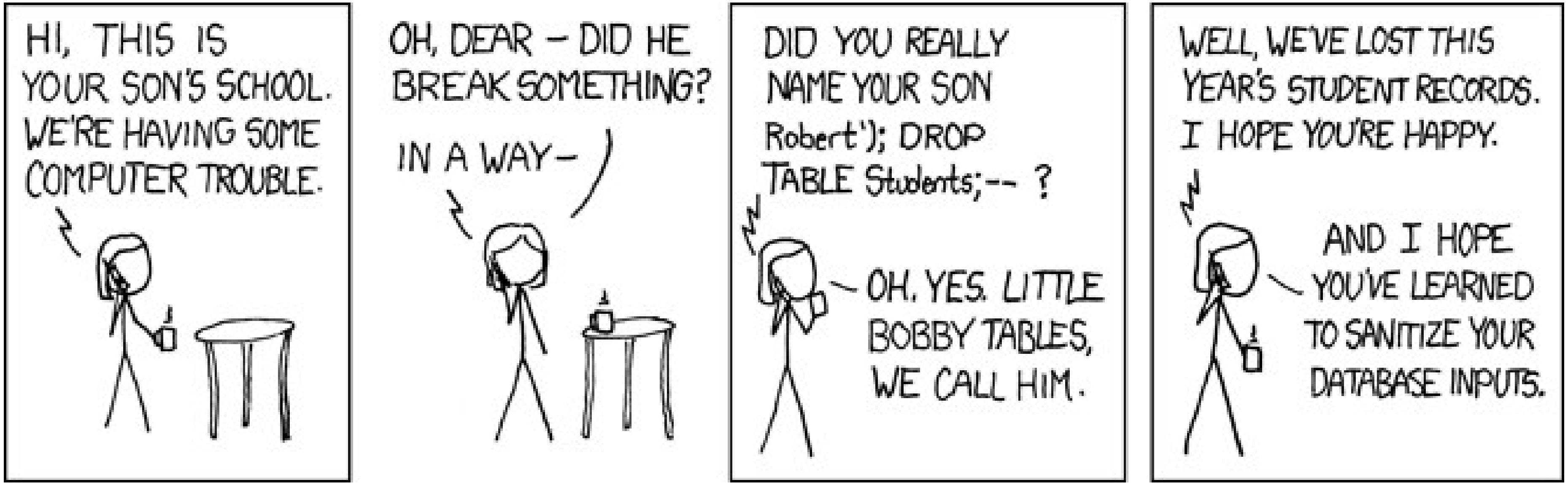
- WARNING:
  NEVER create a query by concatenating strings which you get as inputs
  - e.g.,

```
stmt.executeUpdate(“SELECT dept_name FROM students " +
                   “WHERE name= ’ " + name + " ’) " );
```

→ This line will put your database in danger. Why?

SQL Injection

- Suppose a user entered `“Robert’; DROP TABLE students; --”` in a ‘name’ text input box in GUI.
  - " SELECT dept_name FROM students WHERE name= ’ " + name + " ’) "
- then the resulting statement becomes:
  - " SELECT dept_name FROM students WHERE name= ‘Robert’;
    DROP TABLE students;
    -- "


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Prepared Statement

- Instead, use PreparedStatements when taking an input from the user

```
PreparedStatement pStmt = conn.prepareStatement(
                    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");  /*parameter index, value*/
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- For SELECT queries, use pStmt.executeQuery() to get results, i.e.,
  ResultSet rset = pStmt.executeQuery("…");
- Prepared statement internally uses escaped quotes:
- e.g.

```
SELECT dept_name FROM students
WHERE name= ‘Robert\’; DROP TABLE students; --’
```

Metadata Features

- ResultSet metadata
- E.g., after executing query to get a **`ResultSet`** rset:

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

Metadata (Cont)

- DatabaseMetaData
  - provides methods to get metadata about database

```
DatabaseMetaData dbmd = conn.getMetaData();
      ResultSet rset = dbmd.getColumns(null, "univdb",
                                  "department", "%");
      // Returns: One row for each column;
      // row has a number of attributes, e.g., COLUMN_NAME,
      // TYPE_NAME, etc
      while( rset.next()) {
         System.out.println(rset.getString("COLUMN_NAME"),
                       rset.getString("TYPE_NAME");
}
```

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.

Other JDBC Features

- Functions and procedures can be implemented in procedural PL
  - e.g., Oracle PL/SQL and MS TransactSQL

  -
    ```
    CallableStatement cStmt1 =
             conn.prepareCall("{? = call_some_function(?)}");
    ```

  -
    ```
    CallableStatement cStmt2 =
             conn.prepareCall("{call_some_procedure(?,?)}");
    ```

- Handling large object types
  - `getBlob()` and `getClob()` are similar to the `getString()` method, but return objects of type `Blob` and `Clob`, respectively
  - get data from these objects by `getBytes()`
  - associate a stream with Java Blob or Clob object to update large objects

    -
      ```
      pstmt.setBlob(int parameterIndex,
                    InputStream inputStream).
      ```

ODBC

- Open DataBase Connectivity(ODBC) standard
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Was defined originally for Basic and C, versions available for many languages.

ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect().
- Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password

ODBC Code

```
int ODBCexample()
 {
     RETCODE error;
     HENV    env;      /* environment */
     HDBC    conn;     /* database connection */
     SQLAllocEnv(&env);
     SQLAllocConnect(env, &conn);
     SQLConnect(conn, "localhost", SQL_NTS,
         "bnam", SQL_NTS, "changethis", SQL_NTS);
     {                                          // SQL_NTS: NULL Terminated String

        …. Do actual work …
     }
     SQLDisconnect(conn);
     SQLFreeConnect(conn);
     SQLFreeEnv(env);
 }
```

ODBC Code (Cont.)

- Program sends SQL commands to DBMS by using `SQLExecDirect`
- Result tuples are fetched using `SQLFetch()`
- `SQLBindCol()` binds variables to attributes of the query result
  - When a tuple is fetched, its attribute values are stored in corresponding C variables.
  - Arguments to `SQLBindCol()`
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

ODBC Code (Cont.)

- Main body of program

```
char deptname[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select dept_name, sum (salary)
                     from instructor
                     group by dept_name";
SQLAllocStmt(conn, &stmt);
ret = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (ret == SQL_SUCCESS) {
   SQLBindCol(stmt, 1, SQL_C_CHAR, deptname, 80, &lenOut1);
   SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
   while (SQLFetch(stmt) == SQL_SUCCESS) {
           printf (" %s %g\n", deptname, salary);
   }
}
SQLFreeStmt(stmt, SQL_DROP);
```

ODBC Prepared Statements

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- To prepare a statement

```
SQLPrepare(stmt, <SQL String>);
```

- To bind parameters

```
SQLBindParameter(stmt, <parameter#>,
    … type information and value omitted for
      simplicity..)
```

- To execute the statement

```
retcode = SQLExecute(stmt);
```

- To avoid SQL injection security risk, do not create SQL strings directly using user input; instead use prepared statements to bind user inputs

Autocommit in ODBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - `SQLSetConnectOption(conn,SQL_AUTOCOMMIT,0)}`
  - Transactions must then be committed or rolled back explicitly by
    - `SQLTransact(conn, SQL_COMMIT) or`
    - `SQLTransact(conn, SQL_ROLLBACK)`

Procedural Extensions and Stored Procedures

- SQL is a declarative language
  - each query declares what it wants, but does not tell the logic
  - Convenient, but too restrictive
  - Sometimes imperative features are needed
    - if-then-else
    - for loop
    - while loop
    - etc.

- Stored Procedures
  - Can implement and store procedures inside the database
  - then execute them using the **call** statement
  - Run procedures inside DBMS (unlike JDBC/ODBC)

Function (PL/pgSQL)

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $$
  DECLARE
    declaration;
    [...]
  BEGIN
    < function_body >
    [...]
    RETURN { variable_name | value }
  END;
  $$
LANGUAGE plpgsql;
```

Function (PL/pgSQL)

- Define a function that returns the total count of the number of students

```
CREATE OR REPLACE FUNCTION total_students()
RETURNS integer AS $$
declare
    total integer;
BEGIN
   SELECT count(*) into total FROM STUDENT;
   RETURN total;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT dept_name, count(ID)
FROM department NATURAL JOIN student
GROUP BY dept_name
HAVING count(ID) > total_students()/4;
```

Table Function (PL/pgSQL)

- functions can return a relation as a result
- Example: Return all accounts owned by a given customer

```
CREATE OR REPLACE FUNCTION instructors_of (dname char(20))
            RETURNS TABLE (     ID varchar(5),  name varchar(20),
                              dept_name varchar(20), salary numeric(8,2)
            ) AS $$
            BEGIN
               RETURN QUERY
                  SELECT INS.ID, INS.name, INS.dept_name, INS.salary
                   FROM instructor AS INS
                   WHERE INS.dept_name = instructors_of.dname;
           END; $$
           LANGUAGE plpgsql;
```

- Usage

```
select *
from table (instructors_of (‘Finance’))
```

If-Else Statement (PL/pgSQL)

- Imperative conditional branch

```
IF <condition> then
    <statements>
ELSEIF <condition> then
    <statements>
ELSE
    <statements>
END IF
```

- Note: <condition> is a generic Boolean expression
- Note: END IF has an embedded blank, but ELSEIF does not.

If-Else Statement (PL/pgSQL)

- Define a function that returns the total count of the number of students

```
DO $$
DECLARE std_age INT:= 20;
BEGIN
    IF std_age <= 18 THEN
        RAISE NOTICE ‘student under 18’;
    ELSE
        RAISE NOTICE ‘student over 18’;
    END IF;
END $$;
```

Case Statement (PL/pgSQL)

- Case syntax:

```
CASE <expression>
    WHEN <value> then
      <statements>
    WHEN <value> then
      <statements>
    …
    ELSE
      <statements>
END CASE;
```

```
CASE
    WHEN <condition> then
        <statements>
    WHEN <condition> then
        <statements>
    …
    ELSE
        <statements>
END CASE;
```

Case Statement (PL/pgSQL)

```
DO $$
DECLARE
    letter VARCHAR(10);
    grade_value VARCHAR(10);
BEGIN
    FOR letter IN SELECT grade FROM takes
    LOOP
        grade_value := CASE letter
                            WHEN 'A' THEN '4'
                            WHEN 'B' THEN '3'
                            WHEN 'C' THEN '2’
                            ELSE 'other'
                        END;
        RAISE NOTICE 'Grade: %, Value: %', letter, grade_value;
    END LOOP;
END $$;
```

Simple Loop and While Loop (PL/pgSQL)

- Repeat until terminated by an EXIT or RETURN statement.

```
LOOP
     -- some computations
    IF count > 0 THEN
        EXIT; -- exit loop
    END IF;
END LOOP;
```

- repeats a sequence of statements so long as the boolean-expression evaluates to true

```
WHILE var1 > 0 AND var2 > 0 LOOP
    -- some computations here
END LOOP;
```

For Loop (PL/pgSQL)

- Loop that iterates over a range of integer values

```
DO $$
DECLARE i INT;
BEGIN
    FOR i IN 1..10 LOOP
        RAISE NOTICE ‘i = %’, i;
    END LOOP;
END $$;
```

- iterate through the results of a query

```
DO $$
DECLARE s RECORD;
BEGIN
    FOR s IN
        SELECT id, name FROM student
    LOOP
        RAISE NOTICE 'id= %, name = %', s.id, s.name;
    END LOOP;
END $$;
```

Foreach Loop (PL/pgSQL)

- FOREACH iterates through slices of the array rather than single elements.

```
CREATE FUNCTION scan_rows(int[])
RETURNS void AS $$
DECLARE
    x int[];
BEGIN
    FOREACH x SLICE 1 IN ARRAY $1
    LOOP
        RAISE NOTICE 'row = %', x;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Triggers (PL/pgSQL)

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
  - Examples:
    - Charge $10 overdraft fee if an account balance drops below $500
    - Limit the salary increase of an employee to no more than 5% raise

```
CREATE TRIGGER trigger-name
trigger-time trigger-event ON table-name
FOR EACH ROW
        trigger-action;
```

```
trigger-time ∈ {BEFORE, AFTER}
trigger-event ∈ {INSERT,DELETE,UPDATE}
```

e.g.) AFTER INSERT ON
BEFORE UPDATE ON
…

Trigger Example (PL/pgSQL)

- Create a trigger to update the budget of a department when a new instructor is hired:

```
CREATE OR REPLACE FUNCTION update_budget()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.dept_name IS NOT NULL THEN
        UPDATE department
        SET budget = budget + NEW.salary
        WHERE dept_name = NEW.dept_name;
    END IF;
    RETURN NEW;  -- new refers to the new row inserted
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_budget
AFTER INSERT ON instructor
FOR EACH ROW
EXECUTE PROCEDURE update_budget();
```

Trigger Example (PL/pgSQL)

```
bnam=> select * from department
where dept_name = 'Comp. Sci.';
 dept_name   | building |  budget
------------+----------+-----------
 Comp. Sci. | Taylor   | 100000.00
(1 row)

bnam=> insert into instructor
values (88888, 'Nam', 'Comp. Sci.’, 30000.00);
Query OK, 1 row affected (0.02 sec)


bnam=> select * from department
where dept_name = 'Comp. Sci.';
 dept_name   | building |  budget
------------+----------+-----------
 Comp. Sci. | Taylor   | 130000.00
(1 row)
```