# Advanced SQL: Accessing SQL From a Programming Language

CMPT 354

Jian Pei

jpei@cs.sfu.ca

# 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
- 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 – provide 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

# 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

# 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**

# Update and Query Execution

- 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));
}
```

# Getting Results

- 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

- 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();
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')“
  - What if name is “D'Souza”?

# SQL Injection

- Suppose a 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'
  - User could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared stament internally uses:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - **Always use prepared statements, with user inputs as parameters**

# Retrieve Attribute Names and Data Types

- ResultSet metadata
- Example: after executing query to get a ResultSet rs:

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

# Metadata

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
 ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
while( rs.next()) {
      System.out.println(rs.getString("COLUMN_NAME"),
                              rs.getString("TYPE_NAME");
 }
```

- Where is this useful?

# Another Way

- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
//    TABLE means only regular tables
 ResultSet rs = dbmd.getTables ("", "", "%", new String[] {"TABLES"});
while( rs.next()) {
      System.out.println(rs.getString("TABLE_NAME"));
 }
```

- May use wildcards

# Finding Primary Keys

```
DatabaseMetaData dmd = connection.getMetaData();

// Arguments below are:  Catalog, Schema, and Table
// The value “”  for Catalog/Schema indicates current catalog/schema
//  The value null indicates all catalogs/schemas
ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

while(rs.next()){
   // KEY_SEQ indicates the position of the attribute in
   // the primary key, which is required if a primary key has multiple
   // attributes
   System.out.println(rs.getString(“KEY_SEQ”),
                              rs.getString("COLUMN_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

- Calling functions and procedures
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - Get data from these objects by getBytes()
  - Associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream)
- Resource: JDBC Basics Tutorial at https://docs.oracle.com/javase/tutorial/jdbc/index.html

# SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
  - #sql iterator deptInfoIter ( String dept name, int avgSal);
    deptInfoIter iter = null;
    #sql iter = { select dept_name, avg(salary) from instructor
    group by dept name };
    while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
    }
    iter.close();

# 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

# 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 { …. };

# Embedded SQL

- 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 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;

# Embedded SQL

- To write an embedded SQL query, we use the

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

  statement. The variable *c* is used to identify the query
- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue
  - Specify the query in SQL as follows:

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

# Embedded SQL

- 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.
- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

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

  Repeated calls to fetch get successive tuples in the query result

# Embedded SQL

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* ;

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples

# Updates Through Embedded SQL

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

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

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