Interactive vs. Non-Interactive SQL

- **Interactive SQL:**
  - SQL statements input from terminal; DBMS outputs to screen
  - Inadequate for most uses
    - It may be necessary to process the data before output
    - SQL has very limited expressive power (not computationally complete)
- **Non-interactive SQL:**
  - SQL statements are included in an application program written in a host language, like C, Java, COBOL

Application Program

- **Host language:**
  - A conventional language (e.g., C, Java) that supplies control structures, computational capabilities, interaction with physical devices, support for GUI
- **SQL:**
  - supplies ability to interact with database.
- **Using the facilities of both:** the application program can act as an intermediary between the user at a terminal and the DBMS
Preparation & Execution

• Before an SQL statement is executed, it must be prepared by the DBMS:
  – Translation of SQL into relational algebra based expression
  – What indices can be used?
  – In what order should tables be accessed?
  – What constraints should be checked?
• Decisions are based on schema, table sizes, etc.
• Result is a query execution plan
• Preparation is a complex activity, usually done at run time
  – Time taken is justified by the complexity of query processing

Introducing SQL Into the Application

• SQL statements can be incorporated into an application program in two different ways:
  1. Statement Level Interface (SLI):
     – Application program is a mixture of host language statements and SQL statements and directives
  2. Call Level Interface (CLI):
     – Application program is written entirely in host language
     • SQL statements are values of string variables that are passed as arguments to host language (library) procedures

Statement Level Interface

• SQL statements and directives in the application have a special syntax that sets them off from host language constructs
  – e.g., EXEC SQL SQL_statement
• Precompiler scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS
• Host language compiler then compiles program
• An example of an SLI is SQLJ
  – Embed SQL statements in Java
  – Precompiler translates SQL stmts into Java calls

Call Level Interface

• Application program written entirely in host language (no precompiler)
  – Examples: JDBC, ODBC
• SQL statements are values of string variables constructed at run time using host language
• Application uses string variables as arguments of library routines that communicate with DBMS
  – e.g. executeQuery("SELECT ...")
Topics

- Introduction
- JDBC
- Stored Procedures

JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS
- Can be interfaced to any DBMS that has a JDBC driver

JDBC Architecture

- Four architectural components:
  1. Application:
     - Initiates and terminates connections, submits SQL statements
  2. Driver manager
  3. Driver
     - Connects to data source, transmits requests and returns/translations results and error codes
  4. Data source/database system
     - Processes SQL statements

JDBC Run-Time Architecture
JDBC API

- Package: java.sql
  - DriverManager
  - Connection
  - Statement
  - ResultSet
  - ResultSetMetaData

JDBCExample.java

```java
public class JDBCExample {
    public static void main(String args[]) throws Exception {
        try {
            // load driver
            Class.forName("oracle.jdbc.OracleDriver");
            // create connection to the database
            Connection conn = DriverManager.getConnection(
                s_oraURL, s_userName, s_password);
        } catch (SQLException se) { ...}
```
JDBCExample.java

// URL for the database connection
private static String s_oraURL = "jdbc:oracle:thin:@carrolton:cs4208";
private static String s_userName = "nauman";
private static String s_password = "temp";
}

JDBC URLs

• To send a connection request to a DB server, a URL is used.
• The exact syntax of these URLs depends on the DBMS
• For Oracle, such an URL looks like
  jdbc:oracle:thin:@carrolton:1521:dbrs
  jdbc JDBC protocol is used
  oracle sub-protocol
  thin sub-protocol
  @carrolton hostname (can also use IP address)
  1521 Port
  dbrs DB name (also called SID, or Service ID)

Registering Driver

• First, the appropriate JDBC driver class has to be loaded by the virtual machine
• This is achieved by the statement
  Class.forName("oracle.jdbc.OracleDriver");
• Once loaded, this class registers itself with class
  DriverManager as an available JDBC driver
• Multiple drivers can be loaded, but you should avoid loading more drivers than are needed by your application

Connection Establishment

• Using the method getConnection of
  DriverManager, an instance of Connection
  is actually created:
  Connection conn =
    DriverManager.getConnection(
      url, login, pwd);
• When the getConnection() method is called, the
  DriverManager queries registered drivers
  asking if the driver understands the URL
• Login/password are the login and password to your database account
Connection Object

- **Connection** object encapsulates a single connection to a particular database
  - An application may maintain multiple connections to same/different databases
- When a **Connection** object has outlived its usefulness, be sure to explicitly close it by calling the **close()** method
  - The **Connection** object also acquires certain database resources which can be much more valuable than the memory in the JVM
  - Don’t just rely on Java’s garbage collection mechanism
  - Best to place the **close()** statements (for **Connection**, **Statement**, **ResultSet**) in a finally block of 'try-catch-finally'
  - Note: For an application repeatedly connecting/disconnecting consider connection caching

**Interface java.sql.Connection**

- A variety of methods in the **Connection** interface are available for setting properties of the current connection
- **public int getTransactionIsolation()** and **void setTransactionIsolation(int level)**
  - Sets isolation level for the current connection
  - TRANSACTION_NONE
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
- **public boolean isClosed()**
  - Checks whether connection is still open.

Statement Object

- In turn, this **Connection** instance can be used to create an instance of **Statement**:  
  ```java
  Statement stmt = conn.createStatement();
  ```
- This statement object can repeatedly be used to submit a SQL statement to the DB server
- Create a **String** containing the actual SQL statement, e.g.,  
  ```java
  String sqlStmt = "SELECT name, gpa FROM student WHERE gpa > 3";
  ```

Executing SQL Statements

- To perform a query, use the method **executeQuery(String sql)**:
  ```java
  ResultSet rs = stmt.executeQuery(sqlStmt);
  ```
- To perform an update operation (such as UPDATE, DELETE, INSERT) use the method **executeUpdate(String sql)**:
  ```java
  int rowsAffected = stmt.executeUpdate(sqlStmt);
  ```
- You can execute another query with the same statement object
  - This execution will implicitly close any active ResultSet associated with the object, so be sure that you are done with the processing before calling execute methods again
Processing the Query Results

- When a SQL query executes, the results form a pseudo-table that contains:
  - All the rows that match the query criteria
  - All the columns specified in the SELECT clause
- JDBC uses the `java.sql.ResultSet` to encapsulate the query result
- You can think of `ResultSet` as an object that represents the underlying result table and provides methods to navigate between rows and retrieve particular column values

Retrieving Rows

- For `executeQuery`, the return value is `ResultSet`
  ```java
  ResultSet rs = stmt.executeQuery(query);
  ```
- We can iterate through the result set in a loop row by row:
  ```java
  while (rs.next()) {
    // process the next row
  }
  ```

Retrieving Column Values: `getXXX`-methods

- A column in a database table will have a specific SQL data type
  - E.g., DATE, INTEGER, ...
- `ResultSet` implements `getXXX`-methods, where `XXX` is the desired result type
  - E.g., `getInt` to get the value of a column with SQL data type INTEGER, `getDate`, etc.

Matching Java and SQL Data Types

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java class</th>
<th>ResultSet get method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Boolean</td>
<td>getBoolean()</td>
</tr>
<tr>
<td>CHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer</td>
<td>getInt()</td>
</tr>
<tr>
<td>REAL</td>
<td>Double</td>
<td>getFloat()</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>getDate()</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>getTime()</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>getTimestamp()</td>
</tr>
</tbody>
</table>
Retrieving Column Values: $getXXX$-methods – contd.

- The $getXXX$ methods come in two forms:
  1. With an int corresponding to the attribute index as argument
     - **Index starts from 1 not 0 !!!!!**
     - $rs.getString(1)$; // to get NAME in our ex
     - $rs.getDouble(2)$; // to get GPA in our ex
  2. With a String corresponding to the attribute name as argument
     - $rs.getString("NAME")$; // to get NAME
     - $rs.getDouble("GPA")$; // to get GPA

java.sql.PreparedStatement

- If you use a Statement object multiple times to execute the same query, every call to execute can result in the query being both prepared and executed
- Preparation can be costly and should be avoided if possible
- Alternative for better performance:
  - public interface PreparedStatement extends Statement
- This avoids unnecessary preparations for multiple executions of the statement
  - can cause significant performance improvement

Example: PreparedStatement

- conn represents an active connection:
  PreparedStatement pstmt = conn.prepareStatement(
  "SELECT balance FROM account WHERE account_num = ?");
  pstmt.setInt(1, 98751);
  pstmt.executeQuery();
- We use '?' to indicate parameters for the statement
- These parameters are termed bind variables in Oracle
- To reuse the same PreparedStatement object multiple times, call:
  - Pstmt.clearParameters()
  - before calling the setXXX methods

JDBC & Transactions

- Recall that a transaction is a group of several operations that must be executed atomically
- By default, a new Connection object starts out in transaction auto-commit mode
  - Every SQL statement is executed as an individual transaction
  - Every statement gets immediately committed to the database
- To carry out a multi-statement transaction you must set auto-commit off
JDBC & Transactions: Example

- When you set auto-commit off, you must call commit when your transaction is done:

  ```java
  try{
      conn.setAutoCommit(false);
      // run your statements
      stmt.executeUpdate("INSERT INTO ...");
      stmt.executeUpdate("UPDATE student SET ...");
      conn.commit();
  }catch(SQLException e)
  { conn.rollback(); // undo the results }
  ```

MetaData: Information about the Database

- You can retrieve information about the database you are connected to by using java.sql.DatabaseMetaData interface
  - `DatabaseMetaData dbMeta = conn.getMetaData();`
- `DatabaseMetaData` provides an overwhelming number of methods that can be called to get information on:
  - configuration, database product name, database version, database support for various functionality, ...
    - `dbMeta.getDatabaseProductName()`
    - `dbMeta.getMaxConnections()`

Uses of Class `MetaData`

- By making use of this class, a program can tailor its SQL and use of JDBC on the fly to accommodate different levels of database and JDBC support
- In a generic application, the structure of the resulting table may be unknown
  - E.g., A user enters SQL in a text area and your application needs to display the result in an HTML table
  - An instance of `ResultSetMetaData` can be retrieved from the `ResultSet`, containing the structural information of the result
    - Number of attributes
    - Attributes
    - Domains

Example: Retrieving the Attributes

- To get the `ResultSetMetaData` instance, use `ResultSetMetaData meta= rs.getMetaData();`
- To get the number of attributes, use`int numAttr= meta.getColumnName();`
- To get the attributes and their domains, use a loop such as the following:
Example: Retrieving the Attributes

```java
for (i=1; i<= numAttr; i++) {
    out.print(
        meta.getColumnName(i) + " (" + meta.getColumnTypeName(i) + ")");
}
```

- This will display a row containing all rows and their domains
- The `getColumnTypeName()` method returns the SQL type of the column listed in `java.sql.Types`.

Result Sets and Scrolling

- JDBC supports three types of result sets:
  1. **Forward-only**:
     - not scrollable
  2. **Scroll-insensitive**:
     - Scrollable
     - `rs.absolute(5);` // moves the cursor to the fifth row of rs
     - Changes made to underlying tables after the creation of the result set are not visible through that result set
  3. **Scroll-sensitive**:
     - Scrollable
     - Updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the set

Insensitive Result Set

```
key1 t t t t t t t t            key1    t t t t qqt t t t
key3 yyyyyyyyy            key2    xxxxxxxxx
key4 zzzzzzzzz            key3    yyyrryyyy
key4    zzzzzzzzzz
key5    uuuuuuuuu
key6   vvvvvvvvv
```

Base Table

Result Set

Changes made after opening cursor not seen in the cursor

Tuples added after opening the cursor

Updatable Result Sets

```java
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
```

- Any result set type can be declared **read-only** or **updatable** – `CONCUR_UPDATABLE`
  - assuming SQL query satisfies the conditions for updatable views
- Current row of an updatable result set can be updated or deleted, and a new row can be inserted, causing changes in base table
  - `res.updateString ("Name", "John");` // update attribute "Name" of current row in row buffer.
  - `res.updateRow ();` // install current row buffer in rs and underlying table
Handling Exceptions

```java
try {
    //Java/JDBC code...
} catch (SQLException ex) {
    //exception handling code...
}
```

- try/catch is the basic structure within which an SQL statement should be embedded.
- If an exception is thrown, an exception object, `ex`, is created and the catch clause is executed.
- The exception object has methods to:
  - Print an error message
  - Return the vendor specific error code

Topics

- Introduction
- JDBC
- Stored Procedures

Stored Procedures

- Many DBMS vendors allow stored procedures to be included as schema elements.
- Procedure – written in a conventional programming language with all the control instructions.
- Stored in DBMS.
- Can be invoked by the application.
Stored Procedures: Advantages

- Intermediate data need not be communicated to application (time and cost savings)
- Procedure’s SQL statements prepared in advance
- Authorization can be done at procedure level
- Added security since procedure resides in server
- Applications that call the procedure need not know the details of database schema – all database access is encapsulated within the procedure

Language for Coding Stored Procedures

- SQL standard specifies a language called SQL Persistent Stored Methods (SQL/PSM) for writing procedures
- DBMS vendors provide their own languages similar to SQL/PSM
  - E.g., Oracle has PL/SQL, Microsoft has Transact-SQL
- Certain DBMS also support other languages
  - E.g., in Oracle 9i with Java option, methods written in Java can be stored and executed in the DBMS
  - E.g., support for procedures in C is provided by many DBMS

PL/SQL Example

```sql
CREATE TABLE emp (emp_id NUMBER, name VARCHAR2(100));
-- We create a PL/SQL procedure to load up this table
CREATE OR REPLACE PROCEDURE emp_load(start_id INTEGER, end_id INTEGER)
AS
BEGIN
  FOR i IN start_id..end_id LOOP
    INSERT INTO emp VALUES (i, MOD(i, 20) || 'this is a loooooooooooong name');
    IF MOD(i, 1000) = 0 THEN
      COMMIT;
    END IF;
  END LOOP;
END;
/
```

Calling Stored Procedures

- To call a stored procedure,
  `java.sql.CallableStatement` can be used
```java
cStmt = conn.prepareCall("{call emp_assign(?,?,?,?)}")
cStmt.setInt(1, empId); // IN param
cStmt.setInt(2, deptId); // IN param
cStmt.registerOutParameter(3, Types.VARCHAR); // OUT param
cStmt.registerOutParameter(4, Types.INTEGER); // OUT param
cStmt.execute();
String txnMessage = cStmt.getString(3);
int successCode = cStmt.getInt(4);
```
Topics

• Introduction
• JDBC
• Stored Procedures

Summary

• For most application SQL needs to be embedded in a host language
• Statement Level and Call Level Interfaces have been developed for various languages
• We looked at JDBC which is a Call Level Interface for embedding SQL in Java
  – For more details look at Sun’s JDBC web site and your DBMS vendor’s JDBC manual
    • http://java.sun.com/products/jdbc/
• Stored procedures provide a very powerful mechanism to embed SQL in a computationally complete programming language
  – Should be used where feasible