Database Application Development
Part 2

Chapter 6
Alternative Approach

- Abstract Database interface layer
- Better dynamic integration into host language
- Somewhat Independent of DBMS
  - Database engine need not even understand SQL
  - Depends on a “Driver” layer to translate generic commands into a DBMS-specific call.
- Two Java-embedded efforts
  - ODBS (Open Database Connectivity)
  - JDBC (Java Database Connectivity)
- We’ll examine JDBC
JDBC: Architecture

- Four architectural components:
  - *Application* (initiates and terminates connections, submits SQL statements)
  - *Driver Manager* (JDBC initialization, loads JDBC drivers dynamically, delegates calls from the application to the appropriate driver, provides status and logs)
  - *Driver* (connects to data source, transmits requests and returns/translated results and error codes)
  - *Data Sources* (processes SQL statements)
Drivers are like Translators

- Drivers are analogous to translators at the UN
- Person who speaks language X want to communicate with a person who speaks language Y
- Different strategies
  - Hire a translator for every X-Y combination (expensive)
  - Translate X to a Universal language, U, and then translate U to language Y (only needs as many translators as there are languages)
  - Hybrids (Service bureaus provide translators as needed)
Four Driver Types

**Bridge:**
- Translates SQL commands into non-native API.
  Example: JDBC-ODBC bridge. Code for ODBC and JDBC driver needs to be available on each client.

**Direct translation to native API, non-Java driver:**
- Translates SQL commands to native API of data source.
  Need OS-specific binary on each client.

**Network bridge:**
- Send commands over the network to a middleware server that talks to the data source.
  Needs only small JDBC driver at each client.

**Direction translation to native API via Java driver:**
- Converts JDBC calls directly to network protocol used by DBMS. Needs DBMS-specific Java driver at each client.
What lives where

Java Application

JDBC driver manager

JDBC/native bridge

JDBC/ODBC bridge

JDBC Driver (DBMS Specific)

JDBC middleware (various DBMS)

Native driver (DBMS specific)

ODBC Driver

DBMS
JDBC Classes and Interfaces

Steps to submit a database query:
- Load the JDBC driver
- Connect to the data source
- Execute SQL statements

- Part of "import java.sql.*" package
JDBC Driver Management

- All drivers are managed by the DriverManager class
- Loading a JDBC driver:
  - In the Java code:
    ```java
    Class.forName("oracle/jdbc.driver.OracleDriver");
    ```
  - When starting the Java application:
    ```java
    java -Djdbc.drivers=oracle/jdbc.driver appName
    ```
Connections in JDBC

We interact with a data source through sessions. Each connection identifies a logical session.

- **JDBC URL:**
  jdbc:<subprotocol>:<otherParameters>

**Example:**

```java
String url = "jdbc:oracle:www.bookstore.com:3083";

Connection con;

try {
    con = DriverManager.getConnection(url, userId, password);
} catch (SQLException excpt) {
    ...}
```
Connection Class Interface

- public boolean getAutoCommit() and void setAutoCommit(boolean b)
  If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using commit(), or aborted using rollback().

- public int getTransactionIsolation() and void setTransactionIsolation(int level)
  Gets/Sets isolation level level for the current connection.

- public boolean getReadOnly() and void setReadOnly(boolean b)
  Specifies if transactions in this connection are read-only

- public boolean isClosed()
  Checks whether connection is still open.
Executing SQL Statements

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatement (stored procedures)

- PreparedStatement class:
  - Precompiled, parametrized SQL statements:
    - Structure is fixed
    - Values of parameters are determined at run-time
Executing SQL Statements (Contd.)

String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatement pstmt=con.prepareStatement(sql);

// instantiate parameters with values
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4, age);

// we know that no rows are returned,
// thus we use executeUpdate()
int numRows = pstmt.executeUpdate();
ResultSets

- `PreparedStatement.executeUpdate` only returns the number of affected records
- `PreparedStatement.executeQuery` returns data, encapsulated in a `ResultSet` object (a cursor)

```java
ResultSet cursor = pstmt.executeQuery(sql);

while (cursor.next()) {
    // process the data
}
```
ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- `previous()`: moves one row back
- `absolute(int num)`: moves to the row with the specified number
- `relative (int num)`: moves forward (positive ints) or backward (negative ints)
- `first()` and `last()`
Retrieving Query Results

- Type-specific Accessor methods allow us to retrieve the query results
- Two forms
  - By column index
  - By column name

```java
ResultSet cursor = pstmt.executeQuery(sql);
while (cursor.next()) {
    sailorname = cursor.getString(2)
    rating = cursor.getFloat("rating")
}
```
### 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>
Examining Database Metadata

DatabaseMetaData object gives information about the database system and the catalog.

DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
    “Name:” + md.getDriverName() +
    “version: ” + md.getDriverVersion());
Database Metadata (Contd.)

DatabaseMetaData md = con.getMetaData();
ResultSet trs = md.getTables(null, null, null, null);
String tableName;
While(trs.next()) {
    tableName = trs.getString("TABLE_NAME");
    System.out.println("Table: " + tableName);
    //print all attributes
    ResultSet crs = md.getColumns(null, null, tableName, null);
    while (crs.next()) {
        System.out.println(crs.getString("COLUMN_NAME" + ", ");
    }
}
import java.sql.*;

/**
 * This is a sample program with jdbc odbc  Driver
 **/
public class localdemo {

    public static void main(String[] args) {
        try {
            // Register JDBC/ODBC Driver in jdbc DriverManager
            // On some platforms with some java VMs,
            // newInstance() is necessary...
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

            // Test with MS Access database (sailors ODBC data source)
            String url = "jdbc:odbc:mysailors";

            java.sql.Connection c = DriverManager.getConnection(url);
        }
    }
}
String query = "select * from Sailors";
java.sql.Statement st = c.createStatement();
java.sql.ResultSet rs = st.executeQuery(query);

java.sql.ResultSetMetaData md = rs.getMetaData();
while(rs.next()) {
    System.out.print("\nTUPLE: | ");
    for(int i=1; i<= md.getColumnCount(); i++) {
        System.out.print(rs.getString(i) + " | ");
    }
}
rs.close();
} catch(Exception e) {
e.printStackTrace();
}
SQLJ

Complements JDBC with a (semi-)static query model: Compiler can perform syntax checks, strong type checks, consistency of the query with the schema

- All arguments always bound to the same variable:
  ```
  sql x = {
    SELECT name, rating INTO :name, :rating
    FROM Books WHERE sid = :sid
  }
  ```

- Compare to JDBC:
  ```
  sid=rs.getInt(1);
  if (sid==1) {sname=rs.getString(2);} else {sname2=rs.getString(2);} 
  ```

SQLJ (part of the SQL standard) versus embedded SQL (vendor-specific)
```java
Int sid;
String name;
Int rating;
// named iterator
#sql iterator Sailors(Int sid, String name, Int rating);
Sailors sailors;
// assume that the application sets rating
#sailors = {
    SELECT sid, sname INTO :sid, :name
    FROM Sailors WHERE rating = :rating
};

// retrieve results
while (sailors.next()) {
    System.out.println(sailors.sid + " " + sailors.sname);
}
sailors.close();
```
**SQLJ Iterators**

Two types of iterators ("cursors"):

- **Named iterator**
  - Need both variable type and name, and then allows retrieval of columns by name.
  - See example on previous slide.

- **Positional iterator**
  - Needs only variable type; uses FETCH .. INTO construct:

```sql
#sql iterator Sailors(Int, String, Int);  
Sailors sailors;  
#sailors = ...  
while (true) {  
  #sql {FETCH :sailors INTO :sid, :name} ;  
  if (sailors.endFetch()) { break; }  
  // process the sailor  
}
```
 Stored Procedures

- What is a stored procedure:
  - Program executed through a single SQL statement
  - Executed in the process space of the server

- Advantages:
  - Can encapsulate application logic while staying “close” to the data
  - Reuse of application logic by different users
  - Avoid tuple-at-a-time return of records through cursors
Stored Procedures: Examples

CREATE PROCEDURE ShowNumReservations
    SELECT S.sid, S.sname, COUNT(*)
    FROM Sailors S, Reserves R
    WHERE S.sid = R.sid
    GROUP BY S.sid, S.sname

Stored procedures can have parameters:
   Three different modes: IN, OUT, INOUT

CREATE PROCEDURE IncreaseRating(
    IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
    SET rating = rating + increase
WHERE sid = sailor_sid
Stored procedures don’t have to be written in SQL:

```sql
CREATE PROCEDURE TopSailors(
    IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```
Calling Stored Procedures

EXEC SQL BEGIN DECLARE SECTION
Int sid;
Int rating;
EXEC SQL END DECLARE SECTION

// now increase the rating of this sailor
EXEC CALL IncreaseRating(:sid,:rating);
Calling Stored Procedures (Contd.)

JDBC:
CallableStatement cstmt =
   con.prepareCall("{call ShowSailors}");
ResultSet rs =
cstmt.executeQuery();
while (rs.next()) {
   ...
}

SQLJ:
#sql iterator ShowSailors(...);
ShowSailors showsailors;
#sql showsailors={CALL ShowSailors};
while (showsailors.next()) {
   ...
}
**SQL/PSM**

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) \(\rightarrow\) 

SQL/PSM standard is a representative

**Declare a stored procedure:**

CREATE PROCEDURE name(p1, p2, ..., pn)  
local variable declarations  
procedure code;

**Declare a function:**

CREATE FUNCTION name (p1, ..., pn) RETURNS sqlDataType  
local variable declarations  
function code;
Simple SQL/PSM Example

CREATE FUNCTION rateSailor
  (IN sailorId INTEGER)
  RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
               FROM Reserves R
               WHERE R.sid=sailorId)
IF (numRes > 10) THEN rating=1;
ELSE rating=0;
END IF;
RETURN rating;
Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
    ELSEIF (condition) statements;
    … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without “EXEC SQL”
Summary

- Embedded SQL allows execution of parametrized static queries within a host language
- Dynamic SQL allows execution of completely ad-hoc queries within a host language
- Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL
- APIs such as JDBC introduce a layer of abstraction between application and DBMS
Summary (Contd.)

- SQLJ: Static model, queries checked at compile-time.
- Stored procedures execute application logic directly at the server.
- SQL/PSM standard for writing stored procedures.