ICS 321 Fall 2012

SQL in a Server Environment (ii)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa
Alternative to Embedded SQL

• What if we want to compile an application without the need for a DBMS-specific pre-compiler?
• Use a library of database calls
  – Standardized (non-DBMS-specific) API
  – Pass SQL-strings from host language and presents result sets in a language friendly way
  – Eg. ODBC for C/C++ and JDBC for Java
  – DBMS-neutral
    • A driver traps the calls and translates them into DBMS-specific code
ODBC/JDBC Architecture

• Application
  – Initiates connections
  – Submits SQL statements
  – Terminates connections

• Driver Manager
  – Loads the right JDBC driver

• Driver
  – Connects to the data source,
  – Transmit requests,
  – Returns results and error codes

• Data Source
  – DBMS
4 Types of Drivers

• Type I: Bridge
  – Translate SQL commands to non-native API
  – eg. JDBC-ODBC bridge. JDBC is translated to ODBC to access an ODBC compliant data source.

• Type II: Direct Translation to native API via non-Java driver
  – Translates SQL to native API of data source.
  – Needs DBMS-specific library on each client.

• Type III: Network bridge
  – SQL stmts sent to a middleware server that talks to the data source. Hence small JDBC driver at each client.

• Type IV: Direct Translation to native API via Java driver
  – Converts JDBC calls to network protocol used by DBMS.
  – Needs DBMS-specific Java driver at each client.
High Level Steps

1. Load the ODBC/JDBC driver
2. Connect to the data source
3. [optional] Prepare the SQL statements
4. Execute the SQL statements
5. Iterate over the resultset
6. Close the connection
Getting Data to/fro Host Language

• No declaration of shared variables
• Variables in host language is bound to columns of a SQL cursor
• ODBC
  – SQLBindCol – gets data from SQL environment to host variables.
  – SQLBindParameter – gets data from host variables to SQL environment
• JDBC
  – ResultSet class
  – PreparedStatement class
Prepare Statement or Not?

String sql=“SELECT * FROM books WHERE price < ?”;  
PreparedStatement pstmt = conn.prepareStatement(sql); 
Pstmt.setFloat(1, usermaxprice); 
Pstmt.executeUpdate();

• Executing without preparing statement  
  – After DBMS receives SQL statement,  
    • The SQL is compiled,  
    • An execution plan is chosen by the optimizer,  
    • The execution plan is evaluated by the DBMS engine  
    • The results are returned

• conn.prepareStatement  
  – Compiles and picks an execution plan

• pstmt.executeUpdate  
  – Evaluates the execution plan with the parameters and gets the results

cf. Static vs Dynamic SQL
ResultSet

ResultSet rs = stmt.executeQuery(sqlstr);
while( rs.next() ){
    col1val = rs.getString(1); ...
}

- Iterate over the results of a SQL statement -- cf. cursor
- Note that types of column values do not need to be known at compile time

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java Class</th>
<th>accessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Boolean</td>
<td>getBoolean</td>
</tr>
<tr>
<td>CHAR, VARCHAR</td>
<td>String</td>
<td>getString</td>
</tr>
<tr>
<td>DOUBLE, 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>
RowSet

• When inserting lots of data, calling an execute statement for each row can be inefficient
  – A message is sent for each execute

• Many APIs provide a rowset implementation
  – A set of rows is maintained in-memory on the client
  – A single execute will then insert the set of rows in a single message

• Pros: high performance

• Cons: data can be lost if client crashes.

• Analogous rowset for reads (ie. ResultSet) also available
Stored Procedures

• What?
  – A procedure that is called and executed via a single SQL statement
  – Executed in the same process space of the DBMS server
  – Can be programmed in SQL, C, java etc
  – The procedure is stored within the DBMS

• Advantages:
  – Encapsulate application logic while staying close to the data
  – Re-use of application logic by different users
  – Avoid tuple-at-a-time return of records through cursors
SQL Stored Procedures

**CREATE PROCEDURE** ShowNumReservations

```sql
SELECT S.sid, S.sname, COUNT(*)
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
GROUP BY S.sid, S.sname
```

- Parameters modes: **IN, OUT, INOUT**

**CREATE PROCEDURE** IncreaseRating (\( \text{IN } \text{sailor\_sid} \text{ INTEGER}, \text{IN } \text{increase} \text{ INTEGER} \) )

```sql
UPDATE Sailors
SET rating = rating + increase
WHERE sid = sailor\_sid
```
Java Stored Procedures

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

- SQL: `CALL IncreaseRating(101, 2);`

- Embedded SQL in C:
  ```
  EXEC SQL BEGIN DECLARE SECTION
  int sid; int rating;
  EXEC SQL END DECLARE SECTION
  EXEC SQL CALL IncreaseRating(:sid, :rating);
  ```

- JDBC
  ```
  CallableStatement cst = conn.prepareCall("{call Show Sailors}");
  ResultSet rs = cst.executeQuery();
  ```

- ODBC
  ```
  SQLCHAR *stmt = (SQLCHAR *)"CALL ShowSailors";
  cliRC = SQLPrepare(hstmt, stmt, SQL_NTS);
  cliRC = SQLExecute(hstmt);
  ```
User Defined Functions (UDFs)

• Extend and add to the support provided by SQL built-in functions

• Three types of UDFs
  – Scalar: returns a single-valued answer. Eg. Building SUBSTR()
  – Column: returns a single-valued answer from a column of values. Eg. AVG()
  – Table: returns a table. Invoked in the FROM clause.

• Programable in SQL, C, JAVA.
Scalar UDFs

- Returns the tangent of a value

```
CREATE FUNCTION TAN (X DOUBLE)
    RETURNS DOUBLE
    LANGUAGE SQL
    CONTAINS SQL
    RETURN SIN(X)/COS(X)
```

- Reverses a string

```
BEGIN ATOMIC
    DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT "";
    DECLARE LEN INT;
    IF INSTR IS NULL THEN
        RETURN NULL;
    END IF;
    SET (RESTSTR, LEN) = (INSTR, LENGTH(INSTR));
    WHILE LEN > 0 DO
        SET (REVSTR, RESTSTR, LEN) = (SUBSTR(RESTSTR, 1, 1) CONCAT REVSTR, SUBSTR(RESTSTR, 2, LEN - 1), LEN - 1);
    END WHILE;
    RETURN REVSTR;
END
```
Table UDFs

• returns the employees in a specified department number.

CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
RETURNS TABLE (  
    EMPNO CHAR(6),  
    LASTNAME VARCHAR(15),  
    FIRSTNAME VARCHAR(12))
LANGUAGE SQL
READS SQL DATA
RETURN

SELECT EMPNO, LASTNAME, FIRSTNME
FROM EMPLOYEE
WHERE EMPLOYEE.WORKDEPT
    = DEPTEMPLOYEES.DEPTNO
**Java UDFs**

- **CREATE FUNCTION** `tableUDF` (DOUBLE) RETURNS TABLE (name VARCHAR(20), job VARCHAR(20), salary DOUBLE)
- **EXTERNAL NAME** 'MYJAR1:UDFsrv!tableUDF'
- **LANGUAGE JAVA**
- **PARAMETER STYLE** DB2GENERAL
- **NOT DETERMINISTIC**
- **FENCED**
- **NO SQL**
- **NO EXTERNAL ACTION**
- **SCRATCHPAD 10**
- **FINAL CALL**
- **DISALLOW PARALLEL**
- **NO DBINFO@**

```java
import COM.ibm.db2.app.UDF;

public void tableUDF(
    double inSalaryFactor,
    String outName,
    String outJob,
    double outNewSalary
) throws Exception {
    int intRow = 0;
    ...

} // tableUDF } // UDFsrv class
```