SQL in Application Code

- Most often DBMS access is performed from applications
  - Command-line interface only suitable for advanced users, admins
- SQL commands can be called from within a host language
  - C, Java, etc.
  - Application connects to the database
- Two main approaches:
  - Embed SQL in the host language (Embedded SQL, SQLJ)
  - Create special API to call SQL commands (JDBC)

Important Aspects

- Mapping to language variables and data structures
  - SQL statements “linked” to host variables (+ status variables)
  - Alignment of SQL and host language “native” types
- The “impedance mismatch” problem
  - SQL relations are multisets of records (of unknown cardinality)
  - Some host languages may not support such data types
    - Modern ones do (STL in C++, Collections in Java)
- SQL supports a mechanism called a cursor
  - Bridges the gap between SQL sets and native variables

Outline

- Embedded SQL
- Dynamic SQL
- JDBC (API) - important
- SQLJ (Embedded) – not in use, skip
- LINQ-to-SQL: for C#, newer, since ’07, in use with .NET, more than embedded: supported in C# compiler
- Stored procedures, PL/SQL for Oracle
- We’ll survey embedded SQL/C, LINQ, dig into JDBC and PL/SQL

Overview of Embedded SQL

```
C code + Embedded SQL
Preprocessor
C code + function calls
SQL Library
Executable
```

Oracle implementation:

```
$gcc prog.c -L .../lib -l clntshcore -l clntsh
```

```
$proc MODE=ANSI prog.pc
```
Overview of Constructs

- Connect to DBMS on local machine:
  ```sql
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  ```
- Connect to DBMS on another machine (i.e. our case):
  ```sql
  EXEC SQL CONNECT :username IDENTIFIED BY :password USING :service;
  ```
  Works from topcat or pe07, with :service string "dbs3.cs.umb.edu/dbs3".

- Declaring variables (shared with SQL):
  ```sql
  EXEC SQL BEGIN DECLARE SECTION;
  ... 
  EXEC SQL END DECLARE SECTION;
  ```

Executing Statements
```sql
EXEC SQL SQL_Statement;
``` 
Tested on topcat using dbs3 Oracle

- Need to override bad defaults (supplied proc assumes Red Hat Linux)
  ```bash
  topcat3 proc sys_include=$ORACLE_HOME/precomp/public/sys_include
  sys_include=/usr/include/sys_include
  sys_include=/usr/lib/gcc/x86_64-linux-gnu/include/sys_include
  procdemoPC
  topcat3 proc -I $ORACLE_HOME/precomp/public/procdemo.c
  -L $ORACLE_HOME/lib/ -l clntshcore -l clntsh
  topcat3 ./a.out
  ```

The company’s salespeople are:

<table>
<thead>
<tr>
<th>Salesperson</th>
<th>Salary</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>300.00</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>500.00</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>1400.00</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

GOOD-BYE!!

- This displays data from table `emp`, user `scott` on dbs3

Variables

```sql
EXEC SQL BEGIN DECLARE SECTION;
char c_sname[20]; /*convention is c_ prefix*/
long c_sid;
int c_rating;
float c_age;
EXEC SQL END DECLARE SECTION;
``` 

- In EXEC SQL lines, variables are prefixed by "c;"
- Two special error-handling variables:
  - `SQLCODE (long, is negative if an error has occurred)
  - `SQLSTATE (char[6], codes for common errors)

<table>
<thead>
<tr>
<th>SQLCODE</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>'00000'</td>
<td>no error</td>
</tr>
<tr>
<td>'02000'</td>
<td>no data</td>
</tr>
</tbody>
</table>

(recall that in C, that is 5 chars + 0 terminator)

Type casting

- How are various SQL types mapped to native types?
  ```sql
  Type must match in SQL statement assignments!
  Note: number is not a SQL data type, should be integer
  Number(p,s) should be decimal(p,s)
  ```

Executing Queries

- Insertion (no need to return value in host program)
  ```sql
  EXEC SQL INSERT INTO Sailor(sname) VALUES ('Lubber');
  ```

- If only one single tuple is retrieved
  ```sql
  EXEC SQL SELECT sname INTO :c_sname, FROM Sailors where sid = :c_sid;
  ```

  Will fail at runtime if more than one tuple retrieved!

Cursors

- Mechanism that helps traversing data
  ```sql
  Can declare a cursor on a relation or query statement
  Inspect tuple, or even modify/delete tuples (when allowed)
  ```

  Operations:
  ```sql
  open or close a cursor
  fetch a tuple and move cursor to next tuple in result
  move the cursor
  ```

- Can use `ORDER BY` to control tuple order
  ```sql
  ORDER BY fields must also appear in SELECT clause
  ```
Cursor Example

```sql
EXEC SQL DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND S.rating > :c_minrating
ORDER BY S.sname;

Variable :c_minrating evaluated at the time OPEN is executed!
```

```
EXEC SQL OPEN sinfo;
EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
printf("%s is %d years old\n", c_sname, c_age);
EXEC SQL CLOSE sinfo;
```

Error Handling

- Define action to execute when event/error occurs
  - `EXEC SQL WHENEVER condition action;`
- Condition can be
  - `SQLWARNING`
  - `SQLERROR` (SQLSTATE will indicate error code)
  - `NOT FOUND` (SQLSTATE='02000', means no data retrieved)
- Action
  - `stop`, `goto` label, `continue`, `do f` (function call)
- Examples:
  - `EXEC SQL WHENEVER sqlerror goto report_error;`
  - `EXEC SQL WHENEVER not found goto notfound;`

Looping through rows, closing down

```sql
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;) {
    EXEC SQL FETCH salespeople INTO :emp_rec_ptr;
    printf("%s %9.2f %12.2f\n", emp_rec_ptr->emp_name,
           emp_rec_ptr->salary, emp_rec_ptr->commission);
}
EXEC SQL CLOSE salespeople;
EXEC SQL COMMIT WORK RELEASE;
```

procdemo.pc (partial)

```c
/* Connect to ORACLE. */
strcpy(username, "scott");
strcpy(password, "tiger1");
strcpy(service, "dbs3.cs.umb.edu/dbs3");
EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--");
EXEC SQL CONNECT :username IDENTIFIED BY :password USING :service;
EXEC SQL DECLARE salespeople CURSOR FOR
SELECT ENAME, SAL, COMM
FROM EMP
WHERE JOB LIKE 'SALES%';

/* Open the cursor. */
EXEC SQL OPEN salespeople;

for (;;){
    EXEC SQL FETCH salespeople INTO :emp_rec_ptr;
    printf("%s %9.2f %12.2f\n", emp_rec_ptr->emp_name,
           emp_rec_ptr->salary, emp_rec_ptr->commission);
}
EXEC SQL CLOSE salespeople;
EXEC SQL COMMIT WORK RELEASE;
```

APIs: Alternative to Embedding

- Use library that implements API of DBMS calls
  - No need to modify compilation process
  - Much easier debugging (no tangling with "extra" code)
  - API standardized interface with objects and procedures
- Pass SQL strings from the programming language
  - API returns result sets in language-friendly form
- DBMS API for Java is Java's JDBC
  - DBMS-neutral
  - Each DBMS vendor implements its own "JDBC driver"
  - JDBC driver accepts calls, translates them into DBMS-specific code
  - Packages `java.sql.*`, `javax.sql.*`
    - Collection of classes and interfaces
  - DBMS API for C? Yes, mysql has this. See examples
    - Oracle has OCI under its embedded C, can be programmed directly

 JDBC
JDBC: Architecture

- **Application** initiates terminates connections; submits SQL statements
- **Driver Manager** loads JDBC driver; connects to data source; transmits requests and returns/translations results and error codes
- **JDBC Driver (Oracle)** processes SQL statements
- **JDBC Driver (MySQL)**

Driver Types

- **Bridge (not in common use today)**: Translates JDBC commands into some other API
  - Example: JDBC-ODBC bridge
  - Rarely needed today; all DBs provide direct JDBC driver
- **Direct 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
  - Our drivers are in “jar files” in `/cs630/jdbc`
    - `ojdbc6.jar` for Oracle (for Java6+)
    - `mysql-connector-java-5.1.43.jar` for MySQL

Using JDBC

- 3 steps to submit a database query:
  1. Load the JDBC driver (if necessary)
     - Recent JDBC is smarter; can find the driver itself, if it’s on the path
  2. Connect to the data source
  3. Execute SQL statements

JDBC Driver Management

- All drivers are managed by the **DriverManager** class
- **Loading a JDBC driver (if necessary)**
  - From inside the Java code:
    - If using JDBC version 4.0 or later: it is automatic, no line of code needed
      - This is not covered in R&G, since JDBC v4 is newer than the book
    - If using earlier version JDBC: R&G pg. 198, use Java dynamic loading:
      ```java
      Class.forName("oracle/jdbc.driver.Oracledriver");
      ```
    - Or, pre-version-4, when starting the Java VM (R&G pg. 198)
      ```java
      -Djdbc.drivers=oracle/jdbc.driver
      ```

JDBC classes and their relationships

- JDBC exposes a series of object interfaces through which drivers, connections, SQL statements and results are expressed
  - **DriverManager** itself is a singleton object

Connections in JDBC

- Interaction with data source through sessions
  - A connection identifies a logical session
  - **JDBC URL**: `jdbc:<protocol>:<otherParameters>`
  - Example:
    ```java
    String url="jdbc:oracle:www.bookstore.com:3083";
    Connection conn;
    try{
      conn = DriverManager.getConnection(url, "user", "passwd");
    } catch(SQLException e {...}
    ```
    - For our programs:
      ```
      url="jdbc:oracle:thin:@dbs3.cs.umb.edu:1521:dbs3"
      url="jdbc:mysql://pe07.cs.umb.edu:3306/joedb"
      ```
Executing SQL Statements

- Statement interface
  - 2 subinterfaces:
    - PreparedStatement (for SQL statements)
    - CallableStatement (stored procedures)

- PreparedStatement interface:
  - Precompiled, parametrized SQL statements
  - Structure is fixed
  - Values of parameters are determined at run-time

Example

```java
/* local variables */
int sid = 10;
String sname = "Yuppy";
int rating = 5;
float age = 40.0;

/* creating the statement object */
String sql = "INSERT INTO Sailors VALUES(?,?,?,?)
PreparedStatement pstmt = conn.prepareStatement(sql);
```

Example (contd.)

```java
/* initialize parameters */
pstmt.clearParameters();
pstmt.setInt(1, sid);
pstmt.setString(2, sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4, age);

/* no rows will be returned, use executeUpdate() method */
int numRows = pstmt.executeUpdate();
```

executeUpdate() returns the number of affected records

Retrieving Data: ResultSet class

- Statement.executeQuery returns data
- PreparedStatement can also be used for this purpose
- Retrieval by attribute name or position
  ```java
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT sname FROM Sailors WHERE rating = 10");
  // rs lets us access the sequence of rows returned
  while (rs.next()) {
    String name = rs.getString("sname");
  }
  ```

Note: Similar code in Figure 6.4 on pg. 202 needs fixing

ResultSet

- ResultSet can be a very powerful cursor:
  - next(), previous(), first(), last()
  - absolute(int num): moves to the row with the specified number
  - relative(int num): moves forward or backward
  - However, from the JDK docs:
    A default ResultSet object is not updatable and has a cursor that moves forward only.
    So normally the only movement methods are next() and last(), and last() is rarely used.

CREATE TABLE product(
  product_id INT NOT NULL,
  product_code VARCHAR(10) NOT NULL,
  product_description VARCHAR(100) NOT NULL,
  product_price FLOAT NOT NULL, -- better to use decimal(10,2)
  PRIMARY KEY (product_id)
)

How to create a result set from product with multiple columns and rows

```java
Statement statement = connection.createStatement();
ResultSet products = statement.executeQuery("SELECT * FROM Product");
```

How to loop through a result set

```java
while (products.next()) {
  // statements that process each row
}
```
** Methods of a ResultSet object that return data **

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next()</td>
<td>Moves the cursor to the next row in the result set.</td>
</tr>
<tr>
<td>last()</td>
<td>Moves the cursor to the last row in the result set.</td>
</tr>
<tr>
<td>close()</td>
<td>Releases the result set’s resources.</td>
</tr>
<tr>
<td>getRow()</td>
<td>Returns an int value that identifies the current row of the result set.</td>
</tr>
</tbody>
</table>

** How to return a result set and move the cursor through it **

- To return a result set, use the createStatement method of a Connection object to create a Statement object. Use the executeQuery method of the Statement object to execute a SELECT statement that returns a ResultSet object.
- By default, the createStatement method creates a forward-only, read-only result set. You can only move the cursor through it from the first row to the last and you can’t update it. This is appropriate for most web applications.
- When a result set is created, the cursor is positioned before the first row.
- Use the methods of the ResultSet object to move the cursor.
- To move the cursor to the next row, call the next method. If the row is valid, this method moves the cursor to the next row and returns a true value.

** The Product class**

```java
public class Product implements Serializable {
    private String code;
    private String description;
    private double price;

    public Product() {
        code = "";
        description = "";
        price = 0;
    }
    public void setCode(String code) {
        this.code = code;
    }
    public String getCode() {
        return code;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getDescription() {
        return description;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public double getPrice() {
        return price;
    }
    public String getPriceCurrencyFormat() {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        return currency.format(price);
    }
}
```

** Code that uses indexes to return columns **

- `String code = products.getString(1);`
- `String description = products.getString(2);`
- `double price = products.getDouble(3);`

** Code that uses names to return columns (usually a better way) **

- `String code = products.getString("ProductCode");`
- `String description = products.getString("ProductDescription");`
- `double price = products.getDouble("ProductPrice");`

** Code that creates a Product object (see next slide) from the products result set **

```java
Product product = new Product(products.getString(1),
                               products.getString(2),
                               products.getDouble(3));
```

---

** ResultSet methods for forward-only, read-only result sets **

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getXXX(int columnIndex)</td>
<td>Returns data from the specified column number.</td>
</tr>
<tr>
<td>getXXX(String columnName)</td>
<td>Returns data from the specified column name.</td>
</tr>
</tbody>
</table>
The getXXX methods

- The getXXX methods can be used to return all eight primitive types. For example, the getInt method returns the int type and the getLong method returns the long type.
- The getXXX methods can also be used to return strings, dates, and times. For example, the getString method returns any object of the String class, and the getDate, getTime, and getTimestamp methods return objects of the Date, Time, and Timestamp classes of the java.sql package.

Matching Java and SQL Data Types

Note: there is no standard “Double” SQL type. It should be DOUBLE PRECISION.

How to use the executeUpdate method to...

Add a row

String query = "INSERT INTO Product (ProductCode, ProductDescription, ProductPrice) VALUES ('" + product.getCode() + ", ", ' + product.getDescription() + ", ", ' + product.getPrice());
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);

Update a row

String query = "UPDATE Product SET ProductCode = '" + product.getCode() + ", ", ' + product.getDescription() + ", ", ' + product.getPrice() + ";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);

Delete a row

String query = "DELETE FROM Product WHERE ProductCode = " + productCode;
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);

JDBC: Exceptions and Warnings

- Most of java.sql methods throw SQLException
- SQLWarning is a subclass of SQLException
- But is rarely used in application code (may be logged)
- Not as severe (and their existence has to be explicitly tested)

try {
    String query = "SELECT * FROM table_name;"
    Statement statement = connection.createStatement();
    ResultSet rs = statement.executeQuery();
}

Examining Database Metadata

- DatabaseMetaData object gives catalog information

DatabaseMetaData md = conn.getMetaData();
ResultSet rs = md.getColumns(null, null, null, null);
while (rs.next()) {
    System.out.println("Table: " + rs.getString("TABLE_NAME"));
    System.out.println("Column: " + rs.getString("COLUMN_NAME"));
}