SQL in Application Code

- Embedded SQL
- Cursors
- Dynamic SQL
- JDBC
- SQLJ
- Stored procedures
SQL in Application Code

• SQL commands can be called from within a host language (e.g., C++ or Java) program.
  • SQL statements can refer to host variables (including special variables used to return status).
  • Must include a statement to connect to the right database.

• Two main integration approaches:
  • Embed SQL in the host language (Embedded SQL, SQLJ)
  • Create special API to call SQL commands (ODBC, JDBC, ...)


Impedance mismatch:

- SQL relations are (multi-) sets of records, with no \textit{a priori} bound on the number of records. Traditional procedural programming languages such as COBOL, C, ... did not support this well.

- SQL supports a mechanism called a \texttt{cursor} to handle this.
Embedded SQL

• Approach: Embed SQL in the host language.
  • A preprocessor converts the SQL statements into special API calls.
  • Then a regular compiler is used to compile the code.

• Language constructs:
  • Connecting to a database: EXEC SQL CONNECT
  • Declaring variables: EXEC SQL BEGIN (END) DECLARE SECTION
  • Statements: EXEC SQL Statement;
Embedded SQL: Variables

EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION

- Two special “error” variables:
  - SQLCODE (long, is negative if an error has occurred)
  - SQLSTATE (char[6], predefined codes for common errors)
Cursors

• Can declare a cursor on a relation or query statement (which generates a relation).
• Can open a cursor, and repeatedly fetch a tuple then move the cursor, until all tuples have been retrieved.
  • Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
    • Fields in ORDER BY clause must also appear in SELECT clause.
    • The ORDER BY clause, which orders answer tuples, is only allowed in the context of a cursor.
• Can also modify/delete tuple pointed to by a cursor.
Cursor that gets names of sailors who’ve reserved a red boat, in alphabetical order

EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT  S.sname
    FROM    Sailors S, Boats B, Reserves R
    WHERE   S.sid=R.sid AND R.bid=B.bid AND B.color='red'
    ORDER BY S.sname

• Note that it is illegal to replace \textit{S.sname} by, say, \textit{S.sid} in the \texttt{ORDER BY} clause! (Why?)
• Can we add \textit{S.sid} to the \texttt{SELECT} clause and replace \textit{S.sname} by \textit{S.sid} in the \texttt{ORDER BY} clause?
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION

c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age FROM Sailors S
WHERE S.rating > :c_minrating
ORDER BY S.sname;
do {
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
    printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != ‘02000’);
EXEC SQL CLOSE sinfo;
Dynamic SQL

- SQL query strings up to now always known at compile time (e.g., spreadsheet, graphical DBMS frontend).
- Now construct SQL statements on-the-fly:
- Example:

```
char c_sqlstring[] =
    {"DELETE FROM Sailors WHERE raiting>5"};
EXEC SQL PREPARE readytogo FROM :
    c_sqlstring;
EXEC SQL EXECUTE readytogo;
```
Database APIs: Alternative to Embedding

• Rather than modify compiler, add library with database calls (API)
• Special standardized interface: procedures/objects
• Pass SQL strings from language, present result sets in a language-friendly way
• DBMS-neutral (supposedly)
  • a “driver” traps the calls and translates them into DBMS-specific code
  • multiple drivers => multiple DBMS’s
  • database can be across a network
Database APIs:

- Open DataBase Connectivity (ODBC)
  - first widely-used standard
- JDBC
  - Java API similar to ODBC
- Others
  - Not too widely used ...
ODBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements)
  - Driver manager (load ODBC driver)
  - Driver (connects to data source, transmits requests and returns/translations results and error codes)
  - Data source (processes SQL statements)
ODBC Components

Application

Driver Manager

Driver

Data Src

DB1

Driver

Data Src

DBk

ODBC public API
JDBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements)
  - Driver manager (load JDBC driver)
  - Driver (connects to data source, transmits requests and returns/translated results and error codes)
  - Data source (processes SQL statements)
JDBC Components

Application

Driver Manager

Driver

... Driver

Data Src

Data Src

DB1

DBk

JDBC public API
Four types of drivers:

**Type 1 - 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.

**Type 2 - Direct translation to native API, non-Java driver:**
- Translates SQL commands to native API of data source. Need OS-specific binary on each client.

**Type 3 - 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.

**Type 4 - 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.
JDBC Classes and Interfaces

Steps to interact with a database:

• Load the JDBC driver
• Connect to the data source
• Execute SQL statements
• Close the connection
JDBC Driver Management

• All drivers are managed by the java.sql.DriverManager class

• Loading a JDBC driver:
  • In Java (e.g. Servlet) code:
    • String className = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
    • Class.forName(className);
  • For a Java application:
    • -Djdbc.drivers=com.microsoft.jdbc.sqlserver.SQLServerDriver
  • For a Servlet: add driver jar files to WEB-INF/lib or a common library on server
Connections in JDBC

• A connection object identifies a logical session interacting with the database

• Establish connection using URL:
  • URL pattern (protocol info):
    • jdbc:microsoft:sqlserver://<host>:<port>
    • [:DatabaseName=<db>]
  • Complete URL includes server-specific info
    jdbc:microsoft:sqlserver://localhost:1433
    ;DatabaseName=CS330HW4DB
Example

```java
try {
    String className = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
    Class.forName(className);
} catch(ClassNotFoundException cnfe) { ... }

try {
    String dbURL = "jdbc:microsoft:sqlserver://localhost:1433" + 
                    ";DatabaseName=CS330HW4DB";
    String dbUserName = "CS330USR";
    String dbUserPassword = "cs330";
    connection = DriverManager.getConnection(
                dbURL, dbUserName, dbUserPassword);
} catch(SQLException se) { ... }
```
Connection Class Interface

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

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

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

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

- Class `java.sql.Statement`
  - Entire SQL statement is passed as an argument at runtime:

```java
try {
    String sql = "SELECT S.sid, S.name FROM sailors S"
    + "WHERE S.age < 21";
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    // process the results
    ... (talk about this in a few slides) ...
    rs.close();
    stmt.close();
} catch(SQLException se) { ... }
```
Executing SQL PreparedStatements

- Class `java.sql.PreparedStatement`
- Compile-once, execute-many:

```java
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatement pstmt=conn.prepareStatement(sql);
for( ... ) {
    pstmt.clearParameters();
    pstmt.setInt(1,sid);
    pstmt.setString(2,sname);
    pstmt.setInt(3, rating);
    pstmt.setFloat(4,age);
    int numRows = pstmt.executeUpdate();
    if( numrows != 1 ) { ... // this would be bad ... }
}
Executing SQL CallableStatements

- Invokes a DB server-side stored procedure
- We will talk about this later ...
ResultSets

- `executeUpdate` returns the number of affected records, but no results
- `executeQuery` returns data, encapsulated in a `ResultSet` object (a cursor)

```java
ResultSet rs = pstmt.executeQuery(sql);
while (rs.next()) {
    // process a row of the query result ...
    int theID = rs.getInt(1); // by column position
    String theName = rs.getString(“sname”); // by column name
}
```
ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- `next()`: moves one row forward
- `previous()`: moves one row back
- `absolute(int num)`: moves to the row with the specified number
- `relative (int num)`: moves forward or backward
- `first()` and `last()`
Life Cycle

• Ownership:
  • connection -> statement -> resultset

• At most one active resultset per statement
  • Closing the statement or executing again will implicitly close an active resultset

• Maybe more than one concurrent statement per connection
  • Not required by JDBC spec
  • Use databaseMetaData.getMaxStatements()
  • Closing connection implicitly closes owned statements (hence resultsets)
## 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>
JDBC: Exceptions and Warnings

• Most of java.sql can throw an SQLException if an error occurs.

• SQLWarning is a subclass of SQLException; not as severe (they are not thrown and their existence has to be explicitly tested)
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while (warning != null) {
        // handle SQLWarnings;
        warning = warning.getNextWarning();
    }
    con.clearWarnings();
    stmt.executeUpdate(queryString);
    warning = con.getWarnings();
    ...
} catch (SQLException SQLe) {
    // handle the exception
}
Examining Database Metadata

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

DatabaseMetaData md = conn.getMetaData();
// print information about the driver:
System.out.println("Name:" + md.getDriverName() + "version:" + md.getDriverVersion());
DatabaseMetaData md = conn.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") + ", ");
    }
}
A (Semi-)Complete Example

Connection con = // connect
    DriverManager.getConnection(url, "login", "pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try {
    // handle exceptions
    // loop through result tuples
    while (rs.next()) {
        String s = rs.getString("name");
        Int n = rs.getFloat("rating");
        System.out.println(s + "   " + n);
    }
} catch (SQLException ex) {
    System.out.println(ex.getMessage () + ex.getSQLState () + ex.getErrorCode ());
}
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 = {
        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)
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();
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**  
  - Need only variable type, and then uses FETCH .. INTO construct:  
    ```
    #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 DB 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
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 do not have to be written in 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);
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()) {
   ...
}
Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) - SQL/PSM standard is a representative. There are others: PL/SQL (Oracle), ...

<table>
<thead>
<tr>
<th>Declare a stored procedure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE PROCEDURE name(p1, p2, …, pn)</td>
</tr>
<tr>
<td>local variable declarations</td>
</tr>
<tr>
<td>procedure code;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declare a function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE FUNCTION name (p1, …, pn) RETURNS sqlDataType</td>
</tr>
<tr>
<td>local variable declarations</td>
</tr>
<tr>
<td>function code;</td>
</tr>
</tbody>
</table>
CREATE FUNCTION rate Sailor
  (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 (many) host languages and SQL.
- APIs like ODBC, JDBC introduce a layer of abstraction between application and DBMS.
• SQLJ: Static model, queries checked at compile-time.
• Stored procedures execute application logic directly at the server
• SQL/PSM standard for writing stored procedures