

# Database Application Programming

## Chapter 6

Michael Emmerich, LIACS

March 24, 2010

# Database Application Programming

## Chapter 6

Michael Emmerich, LIACS

March 24, 2010

# Table of Contents

Introduction

Embedded SQL

Dynamic SQL

JDBC

Metadata

Stored Procedures

## Challenges in application programming

- ▶ SQL commands are often used from within a host language (e.g., C++ or Java)
- ▶ SQL statements can refer to host variables (including special variables used to return status)
- ▶ We must include a statement to connect to the right database.
- ▶ Special data-structured for relation handling: *cursors*
- ▶ Execute small procedural programs on data-base level: *stored procedures*

## Two main integration approaches

- ▶ Embed SQL in the host language
  - ▶ Syntax checked at compile time
  - ▶ Embedded SQL in C++, SQLJ in Java
- ▶ Create special application program interface (API) to call SQL commands (Dynamic SQL, JDBC)
  - ▶ Syntax checked at run time
  - ▶ Dynamic SQL in C++, JDBC in Java

## Impedance Mismatch and Cursors

- ▶ Impedance mismatch: Object-oriented languages and relational model does not fit together perfectly
- ▶ SQL relations are (multi-) sets of records, with no a priori bound on the number of records
- ▶ Standard data structure for relations or multi-sets does not exist traditionally in procedural programming languages such as C++ (though now STL seems to get a standard)
- ▶ SQL supports a mechanism called a cursor to handle this

# Embedded SQL

- ▶ Approach: Embed SQL in the host language.
- ▶ A preprocessor converts the SQL statements into special Application Program Interface (API) calls.
- ▶ Then a regular compiler is used to compile the code.  
Language constructs

## Basic Elements in Embedded SQL

- ▶ Connecting to a database:

```
EXEC SQL CONNECT
```

- ▶ Declaring variables:

```
EXEC SQL BEGIN (END) DECLARE SECTION
```

- ▶ Statements:

```
EXEC SQL Statement
```

## Variables in Embedded SQL


- ▶ Declaration of SQL variables

```
EXEC SQL BEGIN DECLARE SECTION
char csname[10];
long csid;
short crating;
float cage;
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), but also status O.K. has code (020000 when reading files)

## Cursors

- ▶ How can we access result records from host language?
- ▶ Declare a cursor on a relation or query statement (which generates a relation).
- ▶ *Open* the cursor, and repeatedly *fetch* a tuple and *move* the cursor, until all tuples have been retrieved.
- ▶ We can also modify/delete tuple pointed to by a cursor.



MedicineID	Price	Producer	website
Aspirin	20 EURO	BAYER	www.bayer.de
Spalt	20 EURO	Merck	www.merck.com
ProIjzer	20 EURO	Sunwell	www.sunwell.com

## Declaring a Cursor

```
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
```

Some remarks:

- ▶ `ORDER BY` statement can be used to sort based on *output attributes*.
- ▶ The declaration of a cursor is done inside an `EXEC` section and not in the `DECLARE` section

## A complete example

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char ccsname[20]; short ccminrating; float ccage;
EXEC SQL END DECLARE SECTION
ccminrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname, S.age FROM Sailors S
    WHERE S.rating > :ccminrating
    ORDER BY S.sname;
do {
    EXEC SQL FETCH sinfo INTO :ccsname, :ccage;
    printf("%s is %d years old", ccsname, ccage);
} while (SQLSTATE != 02000);
EXEC SQL CLOSE sinfo;
```

# Dynamic SQL

- ▶ In Embedded SQL, query strings are always known at compile time
- ▶ Dynamic SQL allows generation of queries at run time
  - ▶ Allow construction of SQL statements on-the-fly
  - ▶ for instance useful when programming search engines, graphical DBMS frontends, etc.
  - ▶ queries checked the syntactical correctness is now the responsibility of the programmer

## Code fragment in Dynamic SQL

```
char ccsqlstring[]={"DELETE FROM Sailors WHERE rating  
> 5" };  
EXEC SQL PREPARE readytogo FROM :ccsqlstring;  
EXEC SQL EXECUTE readytogo;
```

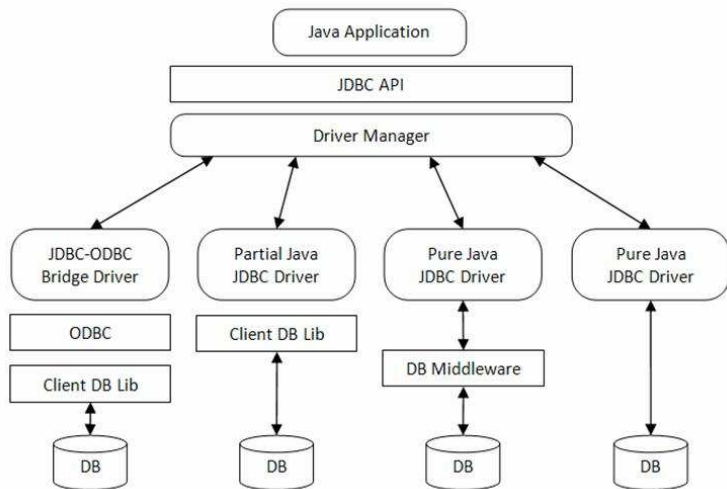
## API instead of embedding

- ▶ Rather than modify compiler, add library with database calls: Application Program Interface (API)
- ▶ Special standardized interface: procedures and objects
- ▶ Pass SQL strings from language, presents result sets in a language-friendly way
- ▶ Example: Sun's JDBC: Java API; Supposedly DBMS-neutral
- ▶ A *driver* traps the calls and translates them into DBMS-specific code database can be across a network

# The four JDBC Components

1. Application (initiates and terminates connections, submits SQL statements)
2. Driver manager (load JDBC driver), Java Object
3. Driver (connects to data source, transmits requests and returns/translates results and error codes)
4. Data source (processes SQL statements)

# JDBC Architecture



## Steps to submit a SQL Query

1. *Load* the JDBC driver
2. *Connect* to the data source
3. *Execute* SQL statements

## Loading a driver

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

## Interaction through sessions

- ▶ We interact with the *data source* through *sessions*.
- ▶ Each connection identifies a logical session.

JDBC URL: `jdbc:<subprotocol>:<otherParameters>`

Example:

```
String url=jdbc:oracle:www.bookstore.com:3083;
Connection con;
try{
    con =
    DriverManager.getConnection(url,usedId,password); }
catch SQLException myexcpt {... }
```

## Control your connections via ...

- ▶ `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

- ▶ question marks can be used as placeholders, for filling in variable settings

```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatement pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,"Michael");
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);
/* we know that no rows are returned, thus we use
executeUpdate() */
int numRows = pstmt.executeUpdate();
```

## ResultSet objects

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

Example:

```
ResultSet rs=pstmt.executeQuery(sql);  
/* rs is now a cursor */  
While (rs.next()) {  
    // process the data  
}
```

## The power of ResultSet cursors

- ▶ `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()`
- ▶ `getRow()`: Gets row number of current row (if ResultSet is scrollable)

How to obtain the number of rows in a result set?

## A complete example

```
/* Load the vendor specific driver: */
Class.forName("oracle.jdbc.driver.OracleDriver")
/* Make the connection */
connection con =
DriverManager.getConnection(jdbc:oracle:thin:oracle:1521:ONW,
username, password);
/* Reading the data:*/
String bar, cola; Float price;
PreparedStatement pstmt;
ResultSet rs = stmt.executeQuery("SELECT * FROM Sells
ORDER BY price");
While (rs.next())
{ bar = rs.getString("bar");
  cola = rs.getString("cola");
  price = rs.getFloat("price");
  System.out.println(bar+ "sells" + cola + "for" + price +
"Euro"); }
```

## Matching SQL and Java Datatypes

SQL type	Java Class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInteger()
DATA	java.sql.date	getDate()
TIME	java.sql.time	getTime()
TIMESTAMP	java.sql.timestamp	getTimeStamp()

## 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)
- ▶ Warnings should at least be checked in debug phase

## Exception in java.sql

```
try {
    stmt=con.createStatement();
    warning=con.getWarnings();
    while(warning != null) {
        /* handle SQLWarnings; */
        warning = warning.getNextWarning();
    }
    con.clearWarnings();
    stmt.executeUpdate(queryString);
    warning = con.getWarnings();
    ...
}
/textttt /* end try */
catch( SQLException SQLe) {
    /* handle the exception */ }
```

## Complete Example with Exceptions

```
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.getInt("rating");
        System.out.println(s + "," + n);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage ()
+ ex.getSQLState () + ex.getErrorCode ());
}
```

## Reading Metadata

- ▶ DatabaseMetaData object gives information about the database system and the catalog.
- ▶ Simple example: print information about the driver:

```
DatabaseMetaData md = con.getMetaData();  
System.out.println("Name:" + md.getDriverName()  
    +" version:  " + md.getDriverVersion());  
>>
```

## Reading a catalog

```
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" + ",
");
    }
}
```

# Stored Procedures

- ▶ Stored procedures allow to execute procedures on database level. Why?:
  - ▶ Security reasons: making only certain information available
  - ▶ Efficiency reasons: Communication overhead is avoided
  - ▶ Transparency reasons: The application programs look more 'readable'

## Application side

- ▶ In JDBC:

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

- ▶ In Embedded SQL:

```
EXEC SQL BEGIN DECLARE SECTION  Int sid;  Int rating;  
EXEC SQL END DECLARE SECTION  
Now call procedure: EXEC CALL IncreaseRating(:sid,  
:rating);
```

## Database Side

- ▶ Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL)
- ▶ SQL/PSM standard is a representative; ORACLE has its own standard
- ▶ 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;*/
```

## Conclusions

- ▶ Embedded SQL allows execution of parametrized static queries within a host language
- ▶ Dynamic SQL and JDBC allow 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
- ▶ Exception Handling and reading metadata are important concepts in JDBC