Calling SQL from a host language (Java and Python)

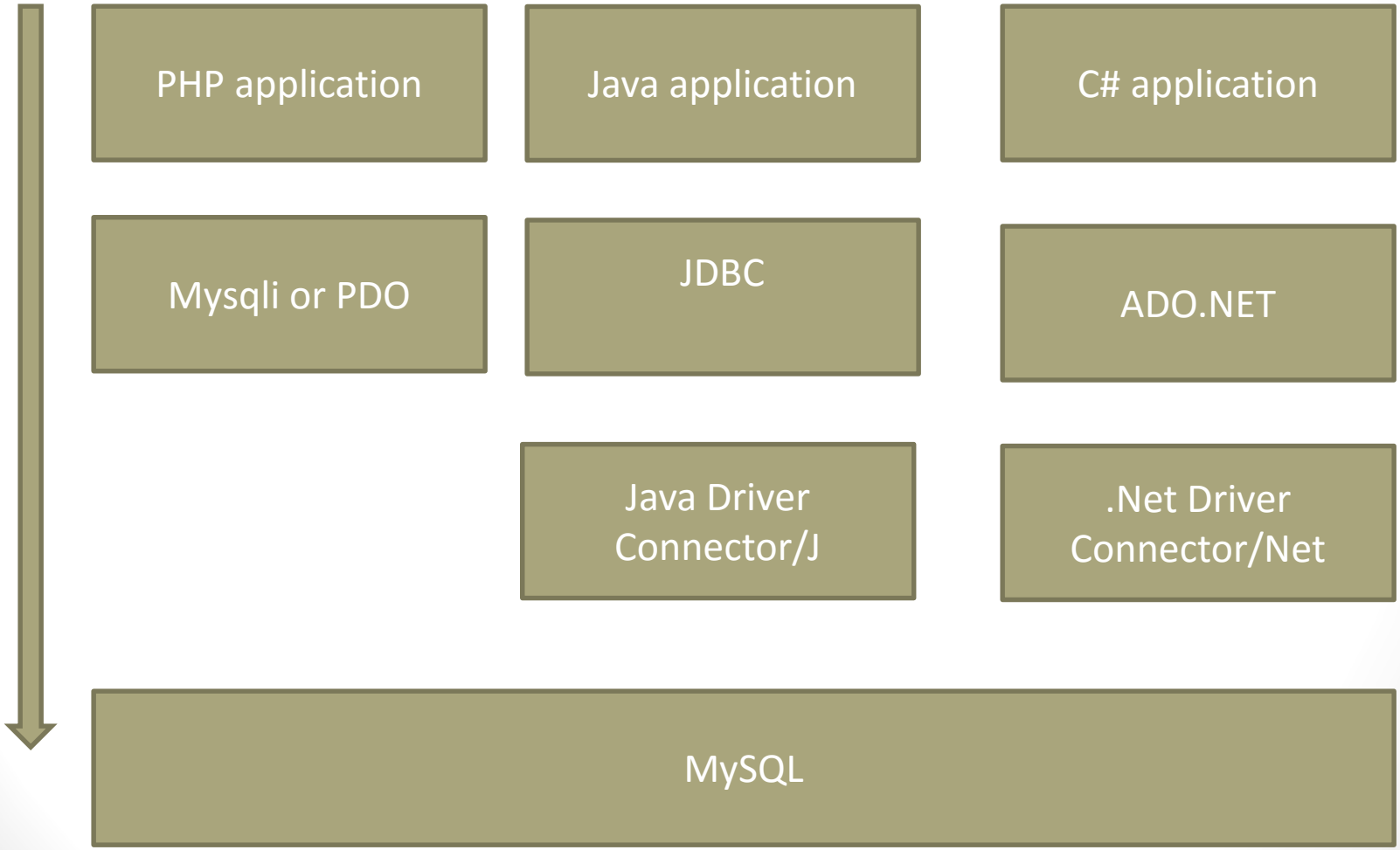
Kathleen Durant

CS 3200

SQL code in other programming languages

- 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)
    - Preprocessor converts SQL code to host language calls. The output from the preprocessor is then compiled by the host comiler
    - SQL standard but typically not used
  - Create special API to call SQL commands
    - JDBC Java Database Connectivity API http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/
    - ODBC Standard database connectivity API
    - Pep 249 – Python Database Application specification https://www.python.org/dev/peps/pep-0249/)
    - The approach we will be using in this class

Embedded SQL

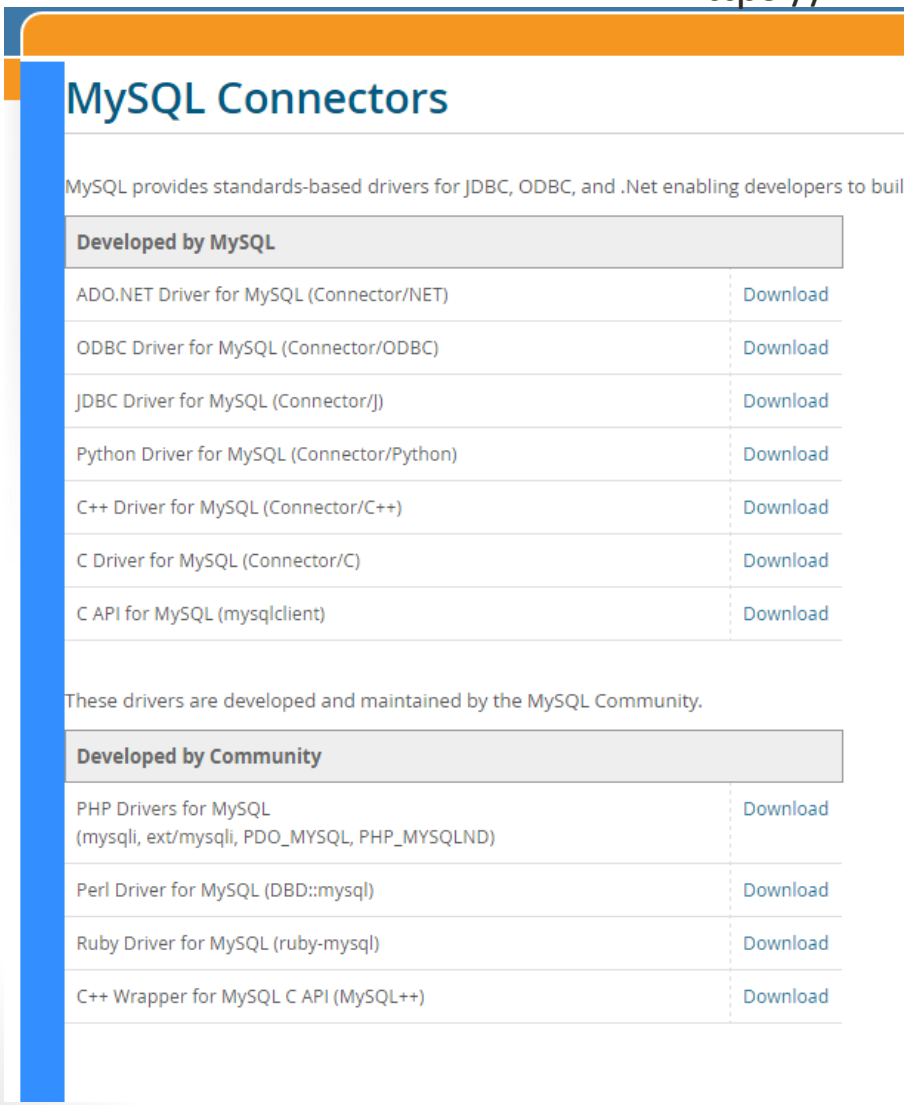


Database API's

- Add a library with database calls (API)
  - Special standardized interface: procedures/objects
  - Pass SQL strings from host language, presents result sets in a host language-friendly way
- A "driver" traps the calls and translates them into DBMS specific code (Oracle, MySQL, SQL Server etc.)
  - database can be across a network
- GOAL: applications are independent of database systems and operating systems

Download the driver you want

https://www.mysql.com/products/connector/



MySQL Connectors

MySQL provides standards-based drivers for JDBC, ODBC, and .Net enabling developers to buil

Developed by MySQL	
ADO.NET Driver for MySQL (Connector/NET)	Download
ODBC Driver for MySQL (Connector/ODBC)	Download
JDBC Driver for MySQL (Connector/J)	Download
Python Driver for MySQL (Connector/Python)	Download
C++ Driver for MySQL (Connector/C++)	Download
C Driver for MySQL (Connector/C)	Download
C API for MySQL (mysqlclient)	Download

These drivers are developed and maintained by the MySQL Community.

Developed by Community	
PHP Drivers for MySQL (mysqli, ext/mysqli, PDO_MYSQL, PHP_MYSQLND)	Download
Perl Driver for MySQL (DBD::mysql)	Download
Ruby Driver for MySQL (ruby-mysql)	Download
C++ Wrapper for MySQL C API (MySQL++)	Download

MySQL Connectors

- **Connector/ODBC** provides driver support for connecting to MySQL using the Open Database Connectivity (ODBC) API.
- **Connector/Net** enables developers to create .NET applications that connect to MySQL. Connector/Net implements a fully functional ADO.NET interface and provides support for use with ADO.NET
- **Connector/J** provides driver support for connecting to MySQL from Java applications using the standard Java Database Connectivity (JDBC) API.
- **Connector/Python** provides driver support for connecting to MySQL from Python applications using an API that is compliant with the Python DB API version 2.0.
  - http://dev.mysql.com/doc/connector-python/en/
- **Connector/C++** enables C++ applications to connect to MySQL.
- **Connector/C** is a standalone replacement for the MySQL Client Library (libmysqlclient), to be used for C applications.

JDBC Processing

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

JDBC Architecture: 4 components

- **Application** (initiates and terminates connections, submits SQL statements)
- **Driver manager** (load JDBC driver)
- **Driver** (connects to data source, transmits requests and returns/translates results and error codes)
- **Data source** (processes SQL statements)

JDBC: Driver Manager

- All drivers are managed by the DriverManager class
  - To load a JDBC driver in Java host code:
    - Class.forName(“oracle/jdbc.driver.Oracledriver”); /Oracle
    - Class.forName("com.mysql.jdbc.Driver"); /My SQL
- When starting the Java application:
  - -Djdbc.drivers=oracle/jdbc.driver
- Or provide the driver in the CLASSPATH directory

Connecting to a DB via JDBC

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

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

Example:

```
//Define URL of database server for
 // database named mysql on the localhost
 // with the default port number 3306.
 String url =
    "jdbc:mysql://localhost:3306/mysql";

//Get a connection to the database for a user named root with a root password.
// This user is the default administrator having full privileges to do anything.
Connection con = DriverManager.getConnection( url,"root", “xxxx");

//Display URL and connection information
 System.out.println("URL: " + url);
 System.out.println("Connection: " + con);
```

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 readonly
- 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 methods to execute SQL statements:
  - **Statement** (both static and dynamic SQL statements)
  - **PreparedStatement** (semi-static SQL statements)
  - **CallableStatment** (stored procedures)
- PreparedStatement class: Precompiled, parameterized SQL statements:
  - Structure of the SQL statement is fixed
  - Values of parameters are determined at run-time

PreparedStatement: Passing and defining Parameters

String sql="INSERT INTO Sailors VALUES(?,?,?,?)";

PreparedStatment pstmt=**con.prepareStatement**(sql);

pstmt.clearParameters();

pstmt.setInt(1,sid);

pstmt.setString(2,sname);

pstmt.setInt(3, rating);

pstmt.setFloat(4,age);

Parameters are positional

// No return rows use executeUpdate()

int numRows = **pstmt.executeUpdate();**

Result Sets

- **PreparedStatement.executeUpdate** only returns the number of affected records
- **PreparedStatement.executeQuery** returns data, encapsulated in a ResultSet object (a cursor)
- ResultSet rs=**pstmt.executeQuery**(sql);
- // rs is now a cursor
- While (rs.next()) {
- // process the data
- }

ResultSet: Cursor with seek functionality

- 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 or backward
  - **first**() and **last**()

Functionality not available in MySQL cursors

Java to SQL Data Types and Result methods

SQL Type	Java class	Result Set get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	Java.sql.Date	getDate()
TIME	Java.sql.Time	getTime()
TIMESTAMP	Java.sql.Timestamp	getTimestamp()

JDBC: Processing exceptions and warnings

- Most of java.sql can throw an error and set SQLException when an error occurs
- SQLWarning is a subclass of SQLException
  - Not as severe as an error
  - They are not thrown
  - Code has to explicitly test for a warning

Example of catching and error

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

Examining Metadata on the DB

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());
```

Metadata: Print out table and its columns

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

http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html

Connect, Process, Check errors

```
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 ());
}
```

Connect

Get multiset

Process with cursor

Catch Errors

Python API

- 2 main concepts for processing database queries
  - Connection object
    - Connection to the database
  - Cursor object
    - Query statement execution
      - Method to execute a statement
    - Result to the results
      - Method to retrieve row of data from the results

Python Example

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Simple MySQL database connection

import flask
import mysql.connector


def main(config):
    output = []
    cnx = mysql.connector.connect(**config)

    cur = cnx.cursor()
    cur2 = cnx.cursor()
    reb = 'rebels'
    movie_id = 1
    stmt_select = "select * from characters order by character_name"

    cur.execute(stmt_select)
    for row in cur.fetchall():
        output.append('{0:20s}  {1:15s} {2:15s}
            {3:15s}'.format(row[0], row[1], row[2], row[3]))
    cur.close()
```

Python example cont.

```
    s2 = 'SELECT * FROM movies WHERE movie_id = {}'.format(movie_id)
        cur2.execute(s2)
        for row in cur.fetchall():
            print(row)

        cur2.callproc('track_planet', args=['Endor'])

        for result in cur2.stored_results():
            print(result.fetchall())

        cur2.close()

        return output


if __name__ == '__main__':
    config = {
        'host': 'localhost',
        'port': 3306,
        'database': 'starwarsfinal',
        'user': 'root',
        'password': 'root',
        'charset': 'utf8',
        'use_unicode': True,
        'get_warnings': True,
    }
```

Summary

- APIs such as JDBC introduce a layer of abstraction between application and DBMS
- Embedded SQL allows execution of parameterized 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