Database Management Systems INFO 210

SQL Part III – Accessing DBs from programs Lecture 10

Franz Wotawa
TU Graz, Institut for Software Technologie
Inffeldgasse 16b/2
wotawa@ist.tugraz.at

Objective

- **Accessing DBs from programs**
  - Programs written in Java und JDBC
  - (Programs written in Python)

- **Important do note**:
  - For almost all databases there are interface libraries available
  - In our JDBC example, there is a specific library for PostgreSQL necessary!
  - Libraries must be added to the path of compilers (e.g., Java compilers)

Database Access from a Programming Language

Two Approaches:

1. Embedding SQL into programming language

   e.g., "Embedded SQL" for C and C++

2. DB access via API (= "application programming interface")

   e.g., JDBC, ODBC

Embedded SQL

- SQL code occurs in program, separated by markers:

```
EXEC SQL SELECT ranking INTO :r
FROM sailors
WHERE sailors.sid = 15765;
r++;
EXEC SQL UPDATE sailors S
SET ranking = :r
WHERE sailors.sid = 15765;
```

- Transfer of values between PL and SQL:
  - use of host language variables in SQL **(prefixed with ":")**
- Compilation in two steps:
  1. Preprocessor translates SQL fragments into function calls of SQL run time library **(= pure C/C++ code, depends on DBMS)**
  2. Regular compiler for C/C++ produces executable

APIs

An **application programming interface** (API) is an **interface or communication protocol** between a client and a server intended to simplify the building of client-side software. It has been described as a "**contract**" **between the client and the server**, such that if the client makes a **request** in a specific format, it will always get a **response** in a specific format or initiate a defined action.[1]

[1] Braunstein, Mark L. (26 July 2018). Health Informatics on FHIR: How HL7's New API is Transforming Healthcare. Springer. p. 9. ISBN 978-3-319-93414-3

ODBC - Accessing DBs from OO programs

- **Objectives**:
  - Provide an interface between DBs and application programs
  - Independence of database systems and operating systems
- **Implementation**:
  - *ODBC driver* is a translation layer between the application and the DB
  - Application uses ODBC functions through an ODBC driver manager with which it is linked
  - The driver passes the query to the DBMS.
    - Like a print driver!

ACCESSING RELATIONAL DATABASES IN JAVA USING JDBC

JDBC functionality

- JDBS implements methods allowing to access databases using SQL statements directly
- For Java we have to import the libraries, i.e.:

```
import java.sql.*;
```

JDBC Introduction using examples

- 1. Step: Open the database!

```
con = DriverManager.getConnection(
    "jdbc:postgresql:
//localhost:5432/lecturedatabase",
    username,
    password);
```

Name of database

Database server address & port

JDBC Introduction using examples

- 2. Step: Issue SQL `SELECT` statements

```
Statement stmt = con.createStatement();
rs = stmt.executeQuery("SELECT id FROM
type");
while (rs.next()) {
     String s = rs.getString("id");
}
rs.close();
stmt.close();
```

JDBC Introduction using examples

- 2. Step (alternative): Issue a SQL `SELECT` statement

```
Statement stmt = con.prepareStatement(
   "SELECT id FROM type WHERE type = ?");
stmt.setInt(1,’VU’);
rs = stmt.executeQuery();
while (rs.next()) {
      String s = rs.getString("id");
}
rs.close()
stmt.close();
```

JDBC Introduction using examples

- 3. Step: Issue SQL `UPDATE`, `INSERT` and `DELETE` statements

```
String sql = "INSERT INTO type VALUES ('S')";
Statement stmt = con.createStatement();
int res = stmt.executeUpdate(sql);
stmt.close();
```

JDBC Introduction using examples

- 3. Step: Issue SQL `UPDATE`, `INSERT` and `DELETE` statements

```
String sql = "UPDATE type SET id = 'SE' WHERE
id = 'S'";
Statement stmt = con.createStatement();
int res = stmt.executeUpdate(sql);
stmt.close();
```

JDBC Introduction using examples

- 3. Step: Issue SQL `UPDATE`, `INSERT` and `DELETE` statements

```
String sql = "DELETE FROM type WHERE id =
'SE'";
Statement stmt = con.createStatement();
int res = stmt.executeUpdate(sql);
stmt.close();
```

JDBC Introduction using examples

- 4. Step: Issue SQL `CREATE` and `DROP` statements

```
Statement stmt = con.createStatement();
stmt.execute("CREATE TABLE test (id
   char(2))");
stmt.close();
```

JDBC Introduction using examples

- 4. Step: Issue SQL `CREATE` and `DROP` statements

```
Statement stmt = con.createStatement();
stmt.execute("DROP TABLE test");
stmt.close();
```

JDBC Introduction using examples

- At the end – Do not forget to CLOSE the connection to the database server!

```
con.close();
```

JDBC Summary

- Using JDBC enables us to establish connections to relational databases using SQL.
- In case of a fault in the SQL statement a `SQLException` is raised! This exception must be catched! Use meaningful treatments of exceptions.

```
try {
    <JDBC command>
} catch(SQLException e) {
    <fault procedure> }
```

ACCESSING RELATIONAL DATABASES IN PYTHON

Accessing relational databases using Python programs

- Similar to JDBC using libraries and library functions
- Python Database API Specification v2.0
- Various implementations, e.g., Psycopg (see http://initd.org/psycopg/docs/)

Main entry points of Psycopg

- The function `connect()` creates a new database session and returns a new `connection` instance.
- The class connection encapsulates a database session. It allows to:
  - create new cursor instances using the `cursor()` method to execute database commands and queries,
  - terminate transactions using the methods `commit()` or `rollback()`.
- The class `cursor` allows interaction with the database:
  - send commands to the database using methods such as `execute()` and `executemany()`,
  - retrieve data from the database by iteration or using methods such as `fetchone()`, `fetchmany()`, `fetchall()`.

Accessing DB using Python

```
import psycopg2

# Connect to an existing database
conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
cur = conn.cursor()

INTERACTION WITH DATABASE COMES HERE

# Make the changes to the database persistent
conn.commit()

# Close communication with the database
cur.close()
conn.close()
```

Create statement

```
# Execute a command: this creates a
# new table

cur.execute("CREATE TABLE test (id
serial PRIMARY KEY, num integer, data
varchar);")
```

Insert statement

```
# Pass data to fill a query
# placeholders and let Psycopg perform
# the correct conversion (no more SQL
# injections!)

cur.execute("INSERT INTO test (num,
data) VALUES (%s, %s)", (100,
"abc'def"))
```

```
#!/usr/bin/python
import psycopg2

# Open database connection
db = psycopg2.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = """INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
    # Execute the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()

# disconnect from server
 db.close()
```

Select statement

```
# Query the database and obtain data
# as Python objects

cur.execute("SELECT * FROM test;")
cur.fetchone()
```

```
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # Execute the SQL command
   cursor.execute(sql)

   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()

   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
      (fname, lname, age, sex, income )
except:
   print "Error: unable to fetch data"
```

Performing Transactions

- Transactions are a mechanism that ensures data consistency.
- Transactions have the following four properties:
  - **Atomicity** – Either a transaction completes or nothing happens at all.
  - **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
  - **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
  - **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.
- The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Performing Transactions

- **COMMIT Operation:**
  - Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.
- **ROLLBACK Operation**
  - If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use **rollback()** method.

Performing Transactions

- **Disconnecting Database:**
  - To disconnect Database connection, use `close()` method.
  - If the connection to a database is closed by the user with the `close()` method, any outstanding transactions are rolled back by the DB.

    However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

Conclusions

- DBs can be accessed from programs directly
- Make use of:
  - Embedded SQL
  - Libraries or APIs
- Use pattern:

  Open DB connection, interact with DB, close connection

Conclusions

- Example applications:
  - JDBC (for Java)
  - Psycopg (for Python)
- In practice:
  - Have a look at available interfaces between the programming language and the DB used
  - Select the one, which offers maintenance

Next Class

Conceptual Modeling