CS-GY 6083 A: Principles of Database Systems

Lab 5: Database Application Development

Supplementary material:
“Database Management Systems” Sec. 6.2, 6.3

Goals and challenges

Goals:

- Interact with a relational database from a host language
- Do so effectively and efficiently

Challenges:

- How do we parse code that uses both host language and SQL syntax?
  - Type matching, e.g., varchar(128) = Java.lang.String
  - Sets / bags of tuples vs. collections of instances (referred to as **impedance mismatch**)
- What can we know at compile time, and what is deferred to run time?
  - Optimization
  - Error handling

SQL Programming Models

Option 1: Embedded SQL:

- Program in a host language uses special SQL directives
- Program is interpreted by a preprocessor
- Preprocessed program is compiled, executable interacts with the DBMS

Option 2: ODBC / JDBC APIs:

- A DBMS vendor implements a standard API
  - Used in host language program, SQL statements are function calls
    - e.g., `ResultSet rs = st.executeQuery(query);` // **Java**
    - `cur.execute(sql)` // **Python**
    - can use regular (unmodified) language compiler
  - A DBMS vendor implements a **driver** that enacts the API
    - SQL statements are processed at **run time**

Embedded SQL vs. ODBC / JDBC

What are the pros and cons of each method?

- Embedded SQL is compiled, ODBC / JDBC is interpreted
  - error handling?
  - platform independence?
- APIs are implemented in the spirit of the host language, are easier to learn and use for programmers
- No need to modify the compiler to use the APIs

*** APIs are more popular today. We will focus on the **Psycopg2** in the remainder of this lab.

Database APIs: ODBC / JDBC / DB-API

- Each DBMS vendor provides a library for very popular host language
  - Microsoft developed Open DataBase Connectivity (ODBC) on Windows as a standard API to databases that implement SQL
  - Sun developed Java DataBase Connectivity (JDBC) as a set of interfaces in Java
    - The Python standard for database interfaces is the Python DB-API (PEP 249); most Python database interfaces adhere to this standard
- A legacy data source may also implement the API. In fact, this is a good idea. **Why?**

DB API (Python)

- It’s a standard, not a library
  - Most Python database interfaces adhere to this standard
- For PostgreSQL there are a few options available
  - Psycopg2 (we will use this)
    - Most common and widely supported PgSQL driver (Unix, Windows)
    - Actively maintained, last release was in 2020
  - pg8000 (Platform independent)
    - Written entirely in Python and does not rely on any external libraries
      - py-postgresql (Platform independent)
    - Pure Python with C optimizations
- More driver options: https://wiki.postgresql.org/wiki/Python

Using Psycopg2

- First: pip install psycopg2
- Note: It's already available on jedi.poly.edu
- Then: import psycopg2
- As simple as that, just use it…
- Need more information? Check the documentation.

http://initd.org/psycopg/docs/

Psycopg2: establishing a connection

```
db_info = get_config()

conn = psycopg2.connect(**db_info)


get_config()

  def get_config (filename=”database.ini”,section=”postgresql”):
parser = ConfigParser()

parser.read(filename)

return {k: v for k, v in parser.items(section)}
```

Psycopg2: establishing a connection

database.ini

```
[postgresql]
host=localhost
port=5432
dbname= <db_name>
user= <db_user_name>
```

- Write your database configuration information into a file
- Do not hard-code connection parameters

Psycopg2: executing SQL statements

- Open a cursor to perform database operations
  - `cur = conn.cursor()`
- Execute a SQL query
    - `sql = "SELECT * FROM test;"`
    - `cur.execute(sql)`

Pyscopg2: retrieving results

- We have a cursor, which we can use to get data.
- Pyscopg2 have three functions for retrieving data
  - `fetchone()`
    - Will return a python tuple (**not to be confused with** a database tuple)
    - Each element of that tuple can have a different type
  - `fetchall()`, `fetchmany(size)`
    - Will return a list of python tuples
  - E.g. `data = cur.fetchall()`
- By default you can only access fields of the tuples by id (0,1,…). If you want to access those by name you need to use Dictionary-like cursor. (http://initd.org/psycopg/docs/extras.html)
    - In Pyscopg they starts from 0. In JDBC column IDs start from 1.

Pyscopg2: type matching

PostgreSQL type	Python type
`CHAR(n), VARCHAR(n)`	`str`
`INTEGER`	`long`
`DATE`	`date`
`NUMERIC`	`decimal`

Python is a dynamically typed language. You don’t need to specify the variable type in your code, but each variables will have a specific type at the time of execution.

Pyscopg2: error handing

Psycopg2 Error

The base class of all other error exceptions. You can use this to catch all errors with one single except statement. Warnings are not considered errors and thus not use this class as base. It is a subclass of the Python StandardError.

```
StandardError
|__ Warning
|__ Error
    |__ InterfaceError
    |__ DatabaseError
        |__ DataError
        |__ OperationalError
        |   |__ psycopg2.extensions.QueryCanceledError
        |   |__ psycopg2.extensions.TransactionRollbackError
        |__ IntegrityError
        |__ InternalError
        |__ ProgrammingError
        |__ NotSupportedError
```

Pyscopg2: error handing

Example

```
>>> try:
...         cur.execute("SELECT * FROM barf;")
...     except psycopg2.Error as e:
...         pass
```

```
>>> e.pgcode
'42P01'
>>> print e.pgerror
ERROR:  relation "barf" does not exist
LINE 1: SELECT * FROM barf
```

Pyscopg2: transaction

- The PostgreSQL transactions handled by the connection object. The connection object is responsible for making changes persistent in the database or reverting it in case of transaction failure.
- The connection object is responsible for terminating its transaction. There are two ways to do that calling either the `commit()` or `rollback()` method.
- By default, the connection is in auto-commit mode. i.e., default auto-commit property is True. That means if any query executed successfully, changes are immediately committed to the database and no rollback is possible.
- To run queries inside a transaction, we need to disable auto-commit. using the conn.autocommit=False we can revert the executed queries result back to the original state in case of failure.

Pyscopg2: transaction

```
try:
    ...
    establish connection & transaction code here
    ...
     conn.commit()
     print("Transaction completed successfully ")

except (Exception, psycopg2.DatabaseError) as error :
    print ("Error in transaction. Reverting all other operations
of a transaction",   error)
    conn.rollback()

finally:
    if(conn):
        cur.close()
        conn.close()
        print("PostgreSQL connection is closed")
```

Pyscopg2: close connection

Close communication with the database

- `cur.close()`
- `conn.close()`

Opening and (not) closing connections

- Do not leave open connections around
  - An issue because of transactions
  - An open connection is a resource, particularly in a multi-user environment
- Use error handling properly: to diagnose run-time errors and to close connections

Pyscopg2 Example

```
db_info = get_config()

conn = psycopg2.connect(**db_info)

cur = conn.cursor()

sql_customer_names = “SELECT name FROM customers;”

cur.execute(sql_customer_names)

data = cur.fetchall()
```

Pyscopg2 Example

```
conn.commit()

cur.close()

conn.close()
```

```
[(1, 'Clarence', 32, 'Waterloo', 'IA', '50703'),
 (2, 'Nichole', 25, 'Colorado Springs', 'CO', '80904'),
 (3, 'Peter', 64, 'Pawpaw', 'IL', '61353'),
 (4, 'Jason', 29, 'Amarillo', 'TX', '79109'),
 (5, 'John', 41, 'Grand Rapids', 'MI', '49503'),
 (6, 'Robert', 25, 'Baltimore', 'MD', '21217'),
 (7, 'Darren', 52, 'New York', 'NY', '10013')]
```

Takeaways

1. Think carefully about your software development methodology for your project
2. Read Pyscopg2 API documentation
3. Read the demo code posted on BrightSpace

JDBC(Java) - Supplemental

- import java.sql.*
- Two Java packages: java.sql and javax.sql (JDBC Optional Package)

JDBC: load a driver

- In JDBC, data source drivers are managed by the Drivermanager class, which has registerDriver method.

- Pick a driver, write code to explicitly load it.

  The following static method shows you one way to register the driver.

Class.forName(“org.postgresql.Driver”);

Recall: we don’t need to load a driver in Pyscopg2

JDBC: establishing a connection

- A session with a data source is started through creation of a Connection object. Connections are specified through a JDBC URL, a URL that uses the jdbc protocol.

```
String url = "jdbc:postgresql://" + host + "/" + dbSID;

Connection connection = DriverManager.getConnection(url, userId,
password);
```

Recall: we need to make sure we don’t hard code the sensitive data

Supplying connection information

```
import java.util.ResourceBundle;
ResourceBundle bundle = ResourceBundle.getBundle(“PgBundle");
Connection conn = DBUtils.openDBConnection(bundle.getString(“dbUser”),
                                           bundle.getString("dbPass"),
                                           bundle.getString("dbSID"),
                                           bundle.getString("dbHost"),
                                           Integer.parseInt(bundle.getString("dbPort")));
```

- Option 1: read in on the command line
- Option 2: use Java resource bundle

JDBC: executing statements

```
Statement stmt = null;

stmt = conn.createStatement();

String sql = "select * from test";

ResultSet rs = stmt.executeQuery(sql);
```

JDBC: different statements types

- Once a connection is obtained, we can interact with the

database. The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

- Statement is most common one

Recall: we don’t have the concept of PreparedStatement and CallableStatement in Pyscopg2

JDBC: retrieving results

- The statement executeQuery returns a ResultSet object, which we can use for: forward and reverse scrolling, in-place editing and insertions. In its most basic form, the ResultSet object allows us to read one row of the output of the query at a time.

- Initially, the ResultSet is positioned before the first row, so we need to retrieve the first row with an explicit call to the next() method.

JDBC: retrieving results

```
Statement stmt = null;
stmt = conn.createStatement();
String sql = "select * from test";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()){
        //Retrieve by column name
        int id  = rs.getInt("id");
        int age = rs.getInt("age");
        String first = rs.getString("first");
        String last = rs.getString("last");

        //Display values
        System.out.print("ID: " + id);
        System.out.print(", Age: " + age);
        System.out.print(", First: " + first);
        System.out.println(", Last: " + last);
  }
```

Recall: In Pyscopg2, we can’t access the column by name directly

JDBC: type matching

PostgreSQL type	Java type
CHAR(n), VARCHAR(n)	String
INTEGER	Integer
DATE	Java.sql.Date
NUMERIC	(any number datatype)

- What will happen on a type mis-match?
- Important:
  - VARCHAR is the preferred character domain instead of CHAR. Use this in your SQL code, or String.equals() won't work
  - Use java.sql.Date rather than any other date / time class

Recall: No need to specify the variable type in python

JDBC: error handing

```
try {
    rs.getInt("name");
} catch (SQLException sale) { //handle the error }
```

- SQLException has two useful methods
  - getSQLState(): returns the same value as defined in SQL for the SQLSTATE host variable
  - getNextException(): used to access the sequence of exceptions that occurred if there were several errors

JDBC: transaction

- If your JDBC Connection is in auto-commit mode, which it is by default, then every SQL statement is committed to the database upon its completion.

- Same as in psycopg2, you can turn off the auto-commit manually.

```
try{
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "insert into cuties" + "values (1, 'brown', 'meow')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "insert into cuties" + "values (2, 'golden', 'woof')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

JDBC: closing up

```
// close ResultSet
 rs.close();
// close Statement
 stmt.close();
// close Connection
 conn.close();
```

Recall: We only need to clean up cursor and connection in psycopg2