Problem 1

It is important not to overload the database when performing workloads, so the number of connections opened with the database can be critical. That’s way one of the essential actions in applications that work with database is keeping as little connections as possible.

So, in order to work with connections more optimally, I created a connector.py file with the connector class (ConnectionManager), that keeps one connection for one session of operations.

Here is an overview of the class:

import psycopg2  
from config.config import configure  
from datetime import datetime

class ConnectionManager:  
 def \_\_init\_\_(self, configpath="../config/database.ini"):  
 print('Connecting to the PostgreSQL database...')  
 try:  
 self.params = configure(filename=configpath)  
 self.conn = psycopg2.connect(\*\*self.params)  
 self.cur = self.conn.cursor()  
 print("Successful connect")  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
  
 def close(self):  
 print("Closing connection...")  
 if self.conn is not None:  
 self.conn.close()  
 print("Have a nice day :)")  
  
 def db\_info(self):  
 try:  
 print('PostgreSQL database version:')  
 self.cur.execute('SELECT version()')  
 db\_version = self.cur.fetchone()  
 print(db\_version)  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
  
 def execute(self, sql):  
 try:  
 self.cur.execute(sql)  
 self.conn.commit()  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
  
 def insert(self, table, columns, values, return\_id=False, add\_creation\_date=False):  
 if add\_creation\_date:  
 values = values + (datetime.now().replace(microsecond=0),)  
 attributes = ("%s,"\*len(columns))[:-1]  
 columns = ", ".join(columns)  
 sql = "INSERT INTO "+table+"("+columns+") VALUES("+attributes+")"  
 if return\_id:  
 sql += " RETURNING "+return\_id+";"  
 try:  
 self.cur.execute(sql, values)  
 self.conn.commit()  
 if return\_id:  
 return self.cur.fetchone()[0]  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
 # if type(error).\_\_name\_\_ == "UniqueViolation":  
 # raise Exception  
  
 def insert\_list(self, table, columns, values):  
 attributes = ("%s," \* len(columns))[:-1]  
 columns = ", ".join(columns)  
 sql = "INSERT INTO " + table + "(" + columns + ") VALUES(" + attributes + ")"  
 try:  
 self.cur.executemany(sql, values)  
 self.conn.commit()  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
  
 def get\_column\_names(self, table):  
 try:  
 self.cur.execute("Select \* FROM "+table+" LIMIT 0")  
 colnames = [desc[0] for desc in self.cur.description]  
 return colnames  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)  
  
 def get\_columns(self, needed\_cols, table, condition=False):  
 needed\_cols = ", ".join(needed\_cols)  
 sql = "SELECT " + needed\_cols + " FROM "+table  
 if condition:  
 sql += " " +condition  
 try:  
 self.cur.execute(sql)  
 received\_columns = self.cur.fetchall()  
 return received\_columns  
 except (Exception, psycopg2.DatabaseError) as error:  
 print(error)

It has all the necessary functions to perform the specific operations in my case.

For connecting to the database, ConnectionManager gets credentials stored in database.ini file using configure() function in config file.

Here is an example using the class:

import datetime  
import random  
from connect import ConnectionManager  
from randomtimestamp import randomtimestamp  
  
  
def generate\_views\_course(n):  
 print("Generating views...")  
 views\_course\_column\_names = connector.get\_column\_names("views\_course")[1:]  
 students\_list = connector.get\_columns(["student\_id"], "student")  
 courses\_list = connector.get\_columns(["course\_id, release\_date"], "course")  
 for i in range(n):  
 print(str(i + 1) + "/" + str(n))  
 student\_id = random.choice(students\_list)[0]  
 course\_id, release\_date = random.choice(courses\_list)  
 release\_date = datetime.datetime(release\_date.year, release\_date.month, release\_date.day)  
 date = randomtimestamp(start=release\_date)  
 connector.insert(  
 "views\_course",  
 views\_course\_column\_names,  
 (student\_id, course\_id, date,)  
 )  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 connector = ConnectionManager()  
 generate\_views\_course(200)  
 connector.close()

5

4

3

2

6

1

This code inserts random view\_courses into the transactional table.

1. Creating a ConnectionManager object and connecting to the database

2. Getting column names for specific table

3. 4. Getting specified columns from the table

5. Inserting data into table

6. Closing connection

In order to use the class for lambdas, I zipped the connector.py file with config.py and database.ini files in /python directory and deployed it as a layer.

In the layer, the codes will be placed into /opt/python directory. From there lambda runtimes include paths to ensure that the function code has access to the libraries that are included in layers.