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)  
   
 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 the schema of the database connection:

Diagram

Description automatically generated

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.

Problem 2

In some tables there are columns representing quantity (total\_...) that need to increment when some operations or transactions happen.

Here is the list of the tables and their quantity columns in my database that need to increment:

|  |  |  |
| --- | --- | --- |
| table | quantity column | when should increment |
| subcategory | total\_topics | manually |
| category | total\_subcategories | manually |
| university | total\_instructors | on instructor create |
| instructor | total\_courses | on course create |
| assistant | total\_courses | on course create |
| topic | total\_courses | on course create |
| course\* | total\_chapters\* | on chapter create\* |
| student | purchased\_courses | on purchases\_course |
| course | total\_students | on purchases\_course |
| instructor | total\_students | on purchases\_course |

\* In my case I insert courses from a csv dataset which contains total\_chapters and I insert chapters with that number

So I created procedures in order to increment the columns and ran them in the script.

Here is an overview of the procedures:

CREATE OR REPLACE PROCEDURE increment\_university(given\_university\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE university  
 SET total\_instructors = total\_instructors+1  
 WHERE university\_id = given\_university\_id;  
END; $$;  
  
CREATE OR REPLACE PROCEDURE increment\_instructor(given\_instructor\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE instructor  
 SET total\_courses = total\_courses+1  
 WHERE instructor\_id = given\_instructor\_id;  
END; $$;  
  
CREATE OR REPLACE PROCEDURE increment\_assistant(given\_assistant\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE assistant  
 SET total\_courses = total\_courses+1  
 WHERE assistant\_id = given\_assistant\_id;  
END; $$;  
  
CREATE OR REPLACE PROCEDURE increment\_topic(given\_topic\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE topic  
 SET total\_courses = total\_courses+1  
 WHERE topic\_id = given\_topic\_id;  
END; $$;

\*  
-- CREATE OR REPLACE PROCEDURE increment\_course(given\_course\_id int) LANGUAGE plpgsql AS  
-- $$ BEGIN  
-- UPDATE course  
-- SET total\_chapters = total\_chapters+1  
-- WHERE course\_id = given\_course\_id;  
-- END; $$;

CREATE OR REPLACE PROCEDURE increment\_student\_purchased(given\_student\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE student  
 SET purchased\_courses = purchased\_courses+1  
 WHERE student\_id = given\_student\_id;  
END; $$;  
  
  
CREATE OR REPLACE PROCEDURE increment\_course\_total\_students(given\_course\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE course  
 SET total\_students = total\_students+1  
 WHERE course\_id = given\_course\_id;  
END; $$;  
  
  
CREATE OR REPLACE PROCEDURE increment\_instructor\_total\_students(given\_instructor\_id int) LANGUAGE plpgsql AS  
$$ BEGIN  
 UPDATE instructor  
 SET total\_students = total\_students+1  
 WHERE instructor\_id = given\_instructor\_id;  
END; $$;

The increment\_course procedure is commented because the number is taken from the csv dataset and there is no need to increment it

Here is an example using the procedures:

def insert\_course(n):

. . . . .

. . . . .

print("Inserting courses with exams and chapters...")  
 i = 1

. . . . .

for course in courses:  
 print(str(i)+"/"+str(n))  
 i += 1  
 course\_id = connector.insert(  
 "course",  
 course\_column\_names,  
 course,  
 return\_id="course\_id",  
 add\_creation\_date=True  
 )

connector.insert(  
 "course\_rating",  
 course\_rating\_column\_names,  
 (course\_id, 0,)  
 )

connector.execute("CALL increment\_instructor(%s)" % course.instructor\_id)

connector.execute("CALL increment\_assistant(%s)" % course.assistant\_id)

connector.execute("CALL increment\_topic(%s)" % course.topic\_id)

. . . . .

Problem 3

There are creation\_date and modification\_date columns in dimension tables. There is a need of inserting current timestamp into modification\_date, when a row is updated in the table.

Here is the list of tables that contain modification\_date column:

* assistant
* category
* subcategory
* topic
* university
* instructor
* instructor\_rating
* course
* course\_rating
* student

So, I created triggers on these tables in order to set current time on modification\_date column.

Here is an overview of the triggers:

CREATE OR REPLACE FUNCTION change\_modification\_date() RETURNS TRIGGER LANGUAGE plpgsql AS  
$$ BEGIN  
 NEW.modification\_date = NOW();  
 RETURN NEW;  
END; $$;  
  
  
CREATE TRIGGER assistant\_update\_modification\_date  
 BEFORE UPDATE  
 ON assistant  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER category\_update\_modification\_date  
 BEFORE UPDATE  
 ON category  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER subcategory\_update\_modification\_date  
 BEFORE UPDATE  
 ON subcategory  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER topic\_update\_modification\_date  
 BEFORE UPDATE  
 ON topic  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER university\_update\_modification\_date  
 BEFORE UPDATE  
 ON university  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER instructor\_update\_modification\_date  
 BEFORE UPDATE  
 ON instructor  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER instructor\_rating\_update\_modification\_date  
 BEFORE UPDATE  
 ON instructor\_rating  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER course\_update\_modification\_date  
 BEFORE UPDATE  
 ON course  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER course\_rating\_update\_modification\_date  
 BEFORE UPDATE  
 ON course\_rating  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();

CREATE TRIGGER student\_update\_modification\_date  
 BEFORE UPDATE  
 ON student  
 FOR EACH ROW  
 EXECUTE PROCEDURE change\_modification\_date();