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:

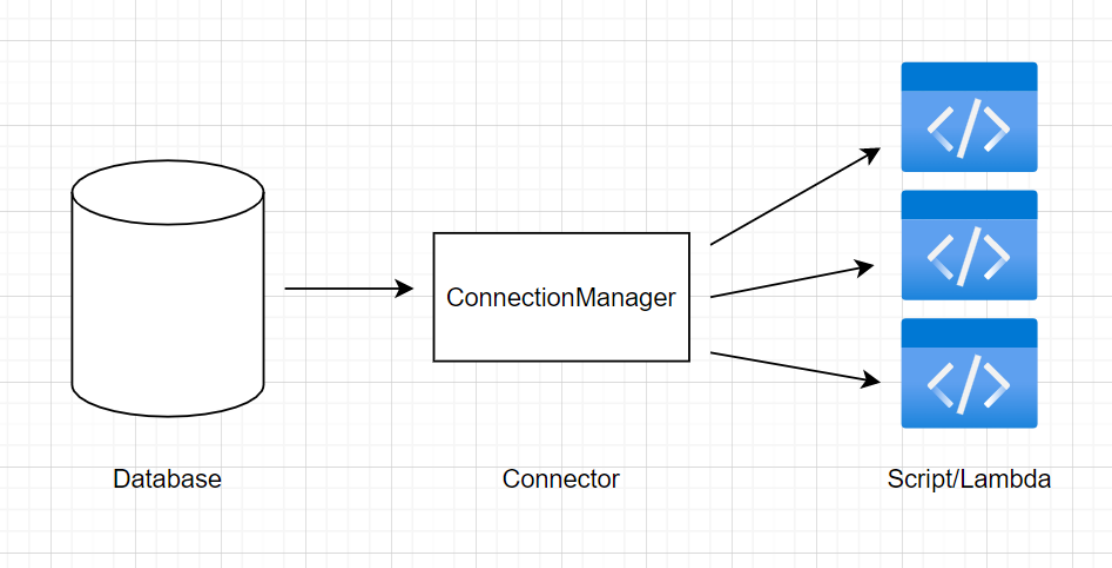
```
import psycopg2
from config.config import configure
from datetime import datetime


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:



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()
```

2 3 4 5 1 6

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();
```