Lab 5: SQpy

February 28, 2023

The goal of this lab is to develop an interpreter for a subset of the SQL programming language.

1 Get the code

The base code used in the labs is in the directory */home/TDDE55/Labs/Lab5/src*. To get the code and start working on it, in your home directory:

```
cp -r /home/TDDE55/Labs/Lab5/src $HOME/Lab7
```

This will copy the skeleton for the *Lab7* assignments, you can now find them in the directory *$HOME/Lab7*. In the rest of the document and in other lab, we will refer to this directory as *dir_to_lab5*.

2 SQL AST and Database

2.1 database class

The main interface to your SQL interpreter is the *database* class. The most important function is the *exec* function that can be used to execute a SQL query on the given database.

```

class database(object):
  def __init__(self):
    '''Create a new database'''
    pass
  def exec(self, sql_ast):
    '''Execute a query'''
  ...
```

Within that database, you will store a list of table, which are made out of a list of row. For this lab, you should store data in your table using a python *namedtuple*, like in *lecture 10*.

In python3:

```
from collections import namedtuple

# This create a new class definition with member 'x', 'y'
Point = namedtuple('Point', ['x', 'y'])

# This create a new object, with x=2 and y=4
```

```python
pt = Point(2, 4)

# Access members of point
print(pt.x)
print(pt.y)
```

2.2 SQL AST

You can find the class representing the AST for the SQL language in *dir_to_lab5/SQpy/sql_ast_node.py*, it contains two classes:

1. *token* which is an enum class contains the list of SQL tokens
2. *ast* which represent a node in the AST

```python
from enum import Enum

class token(Enum):
  select = 0,
  create_table = 1,
  ...

class ast(object):
  def __init__(self, token, **kwargs):
    self.token = token
    for name,value in kwargs:
      self.__setattr__(name,value)
  ...
```

3 CREATE TABLE query

The first step in creating your SQL interpreter and database engine is to be able to create a table. When creating a table in SQL, the user define a set of fields. To simplify our database engine, we will assume that fields do not have types in our implementation of SQL, and a query to create a table would look like:

```sql
CREATE TABLE cities (name, population, longitude,
                     latitude, country, comment)
```

Which translate to the following AST:

```python
query = ast.create_table('cities', ['name', 'population',
                                    'longitude', 'latitude',
                                    'country', 'comment'])
```

Then the query can be executed in the database with:

```python
db = database()
db.execute(query)
```

SQL has no standard way of accessing the list of tables or the list of field of a specific table. In your implementation you should provide two functions:

- *tables* which returns the list of tables
- *fields* which returns the list of fields of a given table You can get the list of fields of an object by accessing its `klass` definition:

```
        namedtuple('Point', ['x', 'y']).klass._fields == ['x', 'y']
```

The following code:

```
print(db.tables())
print(db.fields('cities'))
```

should output:

```
['cities']
['name', 'population', 'latitude', 'longitude', 'country', 'comment']
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 create_table
```

4 INSERT query

Now that you can create tables, you will need to be able to fill the table with data, in SQL the following queries could be used to add a city:

```
INSERT INTO cities VALUES 'Linkoping', 152966, 58.410833, 15.621389,
                          'Sweden', 'My home town';
INSERT INTO cities (name, population, longitude, latitude, country)
                    VALUES 'Paris', 11836970, 48.85, 2.35, 'France';
```

Which translate to the following AST:

```
query1 = ast.insert_into(
              'cities', values = ['Linkoping', 152966, 58.410833,
                                  15.621389, 'Sweden', 'My home town'])
query2 = ast.insert_into(
              'cities', columns = ['name', 'population', 'longitude',
                                   'latitude', 'country'],
              values = ['Paris', 11836970, 48.85, 2.35, 'France'])
```

Note that query2 does not specify the comment field.

At this point, you have not implemented the *SELECT* query yet, to be able to test that your implementation of the insert function works, you will need to implement a *dump_table* function in the database class that take the name of a table and return its content. The following code should insert the two cities and printout the content of the table:

```
db.execute(query1)
db.execute(query2)
print(db.dump_table('cities'))
```

This should output something like:

```
[cities_row(name='Linkoping', population=152966, longitude=58.410833,
             latitude=15.621389, country='Sweden',
             comment='My home town'),
 cities_row(name='Paris', population=11836970, longitude=48.85,
             latitude=2.35, country='France', comment=None)]
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 insert
```

5 DELETE query

Now you can insert data in a table, but your user might want to be able to remove some of that data. This is done with *DELETE FROM* SQL query:

```
DELETE FROM cities WHERE country = 'France' AND latitude < 3;
```

Which is translate to the following AST:

```
query = ast.delete_from(
                'cities', where=ast.op_and(
                    ast.op_eq(
                          ast.identifier('country'), 'France'),
                    ast.op_inferior(
                          ast.identifier('latitude'), 3)))
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 delete
```

6 UPDATE query

Now you can insert and remove data in a table, but your user might be interested in changing the values of a row, this is done with the *UPDATE* SQL query:

```
UPDATE cities SET comment = 'My birth town' WHERE name = 'Paris'
```

Which translate to the AST:

```
query = ast.update(
              'cities',
              set=[('comment', 'My birth town')]
              where=ast.op_eq(
                        ast.identifier('name'), 'Paris'))

```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 update
```

7 SELECT ALL query

Now that you can modify the content of tables, your users are going to be interested in getting data out of your database engine, and they will want to be able to execute *SELECT* SQL queries. First they will want to extract all rows of the tables, with either all columns or only a subset:

```
SELECT * FROM cities
SELECT name,population FROM cities
```

Which translate to the AST:

```
query1 = ast.select(ast.star(), from_table='cities')
query2 = ast.select(['name', 'population'], from_table='cities')
```

The execution function should now return a list of *namedtuple* containing the results. To execute the query:

```
print(db.execute(query1))
print(db.execute(query2))
```

This should output:

```
  [Row(name='Linkoping', population=152966, longitude=58.410833,
    latitude=15.621389, country='Sweden', comment='My home town'),
   Row(name='Paris', population=11836970, longitude=48.85,
    latitude=2.35, country='France', comment=None)]
  [Row(name='Linkoping', population=152966),
   Row(name='Paris', population=11836970)]
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab select_all
```

8 SELECT column expression query

Database users are also interested in filtering the result of their query, they would use the SQL *WHERE* clause for that purpose:

```
SELECT name, population / 1000000 AS population_proportion FROM cities
```

Which translate to the following AST:

```
query = ast.select(['name', (ast.op_divide(ast.identifier('population'), 1000000), 'po
```

And after printing the execution result:

```
  [{'name': 'Paris', 'population': 11836970}]
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 select_exrpression
```

9 SELECT WHERE query

Database users are also interested in filtering the result of their query, they would use the SQL *WHERE* clause for that purpose:

```
SELECT name,population FROM cities WHERE population > 1000000
```

Which translate to the following AST:

```
query = ast.select(['name', 'population'], from_table = 'cities',
          where=ast.op_superior(
                        ast.identifier('population'), 1000000))
```

And after printing the execution result:

```
  [{'name': 'Paris', 'population': 11836970}]
```

To test your code you should use the command:

```
tdde55_lab5_tests dir_to_lab5 select_where
```

10 SELECT aggregation query

Database user are also interested in getting aggregated results, such as the number of cities above a certain population:

```
SELECT count(name) AS city_count FROM cities WHERE population > 1000000
```

Which translate to the following AST:

```
query = ast.select([(ast.count([ast.identifier('name')]) , 'city_count')],
                    from_table = 'cities', where=ast.op_superior(
                        ast.identifier('population'), 1000000))
```

And after printing the execution result:

```
[{'city_count': 1}]
```

```
tdde55_lab5_tests dir_to_lab5 select_aggregation
```

11 SELECT JOIN query

Finally, most databases contains several tables that needs to be connected for some query. We assume that we have a *countries* table with the following information:

```
CREATE TABLE countries (name, population)
INSERT INTO countries VALUES 'Sweden', 9858794;
INSERT INTO countries VALUES 'France', 64513000;
```

If we want to know the proportion of people leaving in each city for each country:

```
SELECT name, cities.population / countries.population AS proportion
        FROM cities INNER JOIN countries ON cities.country = countries.name
```

Which translate to the following AST:

```
query = ast.select(
              ['name',
                (ast.op_divide(
                  ast.identifier('cities', 'population'),
                      ast.identifier('countries', 'population')),
                 'proportion')],
              from_table = 'cities',
              joins=[ast.inner_join('countries',
                    on=ast.op_equal(
                      ast.identifier('cities', 'country'),
                        ast.identifier('countries', 'name')))])
```

Which when executed should output:

```
[{'name': 'Linkoping', 'proportion':0,015515691},
 {'name': 'Paris', 'proportion': 0,18348193}]
```

```
tdde55_lab5_tests dir_to_lab5 select_join
```