НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних систем**

**Лабораторна робота №2**

з дисципліни **Бази даних і засоби управління**

*на тему: “Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL”*

Виконав:

студент ІII курсу

групи КВ-01

Щикалюк Г. І.

Перевірив:

Київ – 2022

*Метою роботи* є здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

*Загальне завдання* роботи полягає у наступному:

1. Реалізувати функції перегляду, внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.
2. Передбачити автоматичне пакетне генерування «рандомізованих» даних у базі.
3. Забезпечити реалізацію пошуку за декількома атрибутами з двох та більше сутностей одночасно: для числових атрибутів – у рамках діапазону, для рядкових – як шаблон функції LIKE оператора SELECT SQL, для логічного типу – значення True/False, для дат – у рамках діапазону дат.
4. Програмний код виконати згідно шаблону MVC (модель-подання-контролер).

*Деталізоване завдання*:

1. Забезпечити можливість уведення/редагування/вилучення даних у таблицях бази даних з можливістю контролю відповідності типів даних атрибутів таблиць (рядків, чисел, дати/часу). Для контролю пропонується два варіанти: контроль при введенні (валідація даних) та перехоплення помилок (try..except) від сервера PostgreSQL при виконанні відповідної команди SQL. Особливу увагу варто звернути на дані таблиць, що мають зв’язок 1:N. При цьому з боку батьківської таблиці необхідно контролювати **вилучення** рядків за умови наявності даних у підлеглій таблиці. З точки зору підлеглої таблиці варто контролювати наявність відповідного рядка у батьківській таблиці при виконанні **внесення** нових даних. Унеможливити виведення програмою системних помилок на екрані шляхом їх перехоплення і адекватної обробки. Внесення даних виконується користувачем у консольному вікні програми.
2. Забезпечити можливість автоматичної генерації великої кількості даних у таблицях за допомогою вбудованих у PostgreSQL функцій роботи з псевдовипадковими числами. Дані мають бути згенерованими **не мовою програмування, а відповідним SQL-запитом**!
3. Для реалізації пошуку необхідно підготувати 3 запити, що включають дані з декількох таблиць і фільтрують рядки за 3-4 атрибутами цих таблиць. Забезпечити можливість уведення конкретних значень констант для фільтрації з клавіатури користувачем. Крім того, після виведення даних необхідно вивести час виконання запиту у мілісекундах. Перевірити швидкодію роботи запитів на попередньо згенерованих даних.
4. Програмний код організувати згідно шаблону Model-View-Controller(MVC). При цьому модель, подання та контролер мають бути реалізовані у окремих файлах. Для доступу до бази даних використовувати **лише мову SQL** (без ORM).

**Логічна модель бази даних**

Нижче (Рисунок 1) наведено логічну модель бази даних:

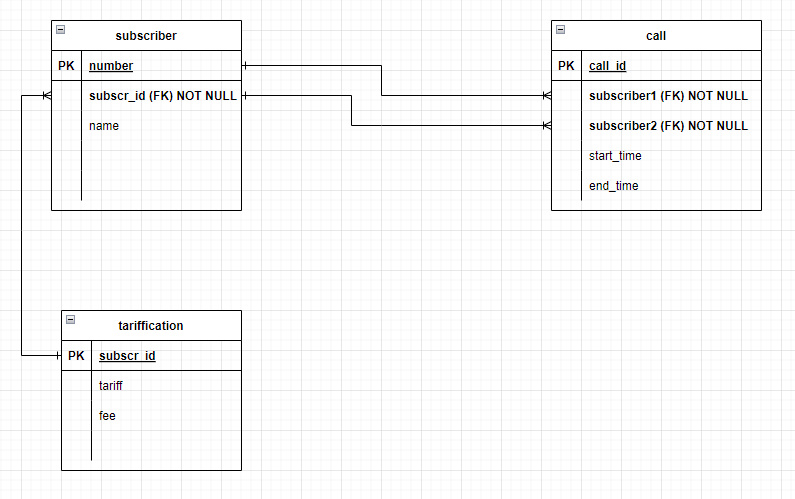


Рисунок 1 – Логічна модель бази даних

**Середовише розробки та налаштування підключення до бази даних**

Застосунок був розроблений мовою програмування Python, середовище розробки – Microsoft Visual Studio 2022. Для розробки програми також була використана стороння бібліотека – psycopg2, яка надає API для доступу до баз даних на основі PostgreSQL

**Шаблон проектування**

У програмі був використаний шаблон проектування MVC

Model - клас, що описує так звану бізнес-логіку програмного забезпечення. В розробленій програмі йому відповідає клас db\_model з файлу Model.py.

View – клас що відповідає за візуалізацію данних а також елементи інтерфейсу. В розробленій програмі йому відповідає клас db\_view з файлу View.py.

Controller – клас, що забезпечує зв’язок між користувачем та системою. Він отримує данні що вводяться користувачем, використовує клас Model для їх обробки та клас View для подання результатів. В розробленій програмі йому відповідає клас db\_controller з файлу Controller.py.

**Структура програми та її опис**

Структура програми зображена на рисунку 2:

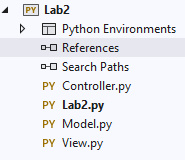


Рисунок 2 – Структура програми

Програма складається з чотирьох файлів: Lab2.py, Model.py, View.py, Controller.py.

У файлі Model.py описаний клас, який реалізує методи управління базою данних.

У файлі View.py описаний клас, що надає методи для виведення меню а також подання результатів запитів у консолі.

У файлі Controller.py описаний клас, що забезпечує взаємодію з користувачем. У своїй роботі він використовує об’єкти класів що описані в файлах Model.py та View.py

**Структура меню програми**

Нижче (Рисунок 3) наведено стартове меню програми:

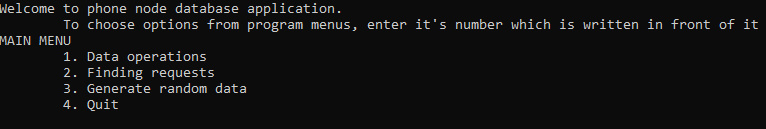


Рисунок 3 – Початкове меню програми

Перший пункт відкриває доступ до меню операцій (Рисунок4).

Другий пункт надає доступ до пошукових запитів. Користувачу необхідно вибрати один з трьох запитів, після чого ввести значення атрибутів, за якими необхідно виконувати пошук.

Третій пункт пропонує генерацію випадкових данних. Користувач повинен вибрати таблицю, в якій будуть згенеровані данні, а потім ввести кількість записів, які потрібно згенерувати.

Четвертий пункт дозволяє виконати вихід з програми.

Далі наведено зображення меню з операціями а також його опис.

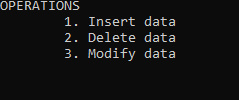


Рисунок 4 – Меню операцій з даними таблиць

Перший пункт пропонує внести дані. Користувач має вибрати таблицю до якої будуть внесені дані, після чого, ввести їх.

Другий пункт пропонує видалення даних. Користувач повинен вибрати таблицю з якої будуть видалені дані, після чого, ввести номер рядку.

Третій пункт пропонує змінити існуючі данні. Користувач повинен обрати таблицю для редагування, ввести номер рядку, після чого користувач зможе ввести нові данні.

**Лістинги**

Лістинг фрагменту програми для внесення даних

**def** **insert\_data**(self, table\_name, values, columns):

columns\_line = '('

values\_line= '('

**for** key **in** range(len(values)):

**if** values[key]:

columns\_line += columns[key] + ','

values\_line += f"'{values[key]}'" + ','

values\_line = values\_line[:-**1**] + ')'

columns\_line = columns\_line[:-**1**] + ')'

query = sql.SQL('INSERT INTO {table} {columns} VALUES {values}').format(

table = sql.Identifier(table\_name),

columns = sql.SQL(columns\_line),

values = sql.SQL(values\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** ins\_ex:

print(ins\_ex)

Лістинг фрагменту програми для видалення даних

**def** **delete\_data**(self, table\_name, attr, cond):

condition\_line = f"{attr} = {cond}"

query = sql.SQL('DELETE FROM {table} WHERE {condition}').format(

table = sql.Identifier(table\_name),

condition = sql.SQL(condition\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** del\_ex:

print(del\_ex)

Лістинг фрагменту програми для зміни даних

**def** **change\_data**(self, table\_name, new\_values, columns, attr, cond):

condition\_line = f"{attr} = '{cond}'"

new\_values\_line = ''

**for** key **in** range(len(new\_values)):

**if** new\_values[key]:

new\_values\_line += f"{columns[key]} = '{new\_values[key]}'" + ','

new\_values\_line = new\_values\_line[:-**1**] + ''

query = sql.SQL('UPDATE {table} SET {new\_values} WHERE {condition}').format(

table = sql.Identifier(table\_name),

new\_values = sql.SQL(new\_values\_line),

condition = sql.SQL(condition\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** chng\_ex:

print(chng\_ex)

Лістинг фрагменту програми для пошуку даних у таблицях

**def** **join\_general**(self, main\_query, condition=""):

new\_cond = condition

**if** condition:

new\_cond = "WHERE " + condition

t1 = time.time()

**try**:

self.\_\_cursor.execute(main\_query.format(new\_cond))

**except** **Exception** **as** find\_ex:

print(find\_ex)

t2 = time.time()

**return** ((t2 - t1) \* **1000**, self.\_\_cursor.fetchall())

**def** **join\_subscr\_id\_fee\_tariff**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, fee\_rng\_start, fee\_rng\_end, tariff):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "t.fee BETWEEN " + str(fee\_rng\_start) + " AND " + str(fee\_rng\_end)

sub\_cond\_3 = "t.tariff LIKE " + "'" + str(tariff) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN tarification as t ON s.subscr\_id = t.subscr\_id {} ORDER BY s.subscr\_id ASC

""", condition)

**def** **join\_subscr\_id\_subscriber1\_start\_time**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, subscriber1, start\_time\_rng\_start, start\_time\_rng\_end):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "c.subscriber1 LIKE " + "'" + str(subscriber1) + "'"

sub\_cond\_3 = "c.start\_time BETWEEN " + "'" + str(start\_time\_rng\_start) + "'" + " AND " + "'" + str(start\_time\_rng\_end) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN call as c ON s.subscr\_id = c.subscriber2 {} ORDER BY s.subscr\_id ASC

""", condition)

**def** **join\_subscr\_id\_subscriber1\_end\_time**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, subscriber1, end\_time\_rng\_start, end\_time\_rng\_end):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "c.subscriber1 LIKE " + "'" + str(subscriber1) + "'"

sub\_cond\_3 = "c.end\_time BETWEEN " + "'" + str(end\_time\_rng\_start) + "'" + " AND " + "'" + str(end\_time\_rng\_end) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN call as c ON s.subscr\_id = c.subscriber2 {} ORDER BY s.subscr\_id ASC

""", condition)

Лістинг фрагменту програми для генерації випадкових даних у таблицях

**def** **generate\_data**(self, table\_name, count):

types = self.get\_column\_types(table\_name)

fk\_array = self.get\_foreign\_key\_info(table\_name)

select\_subquery = ""

insert\_query = "INSERT INTO " + table\_name + " ("

from\_query = "generate\_series(1," + str(count) + ") as ser"

limit\_query = " LIMIT " + str(count)

pk = self.get\_table\_data(table\_name)[**0**]

pk = str(pk[**0**])

**for** i **in** range(**1** **if** re.match(r'.\*\_id', pk) **else** **0**, len(types)):

t = types[i]

name = t[**0**]

type = t[**1**]

fk = [x **for** x **in** fk\_array **if** x[**0**] == name]

**if** fk:

**if** self.test\_is\_referenced(table\_name, str(fk[**0**][**0**])):

select\_subquery += str(fk[**0**][**2**])

from\_query += ", " + str(fk[**0**][**1**])

**else**:

select\_subquery += ('(SELECT {} FROM {} ORDER BY RANDOM(), ser LIMIT 1)'.format(fk[**0**][**2**], fk[**0**][**1**]))

**elif** type == 'integer':

select\_subquery += 'trunc(random()\*100)::INT'

**elif** type == 'text':

select\_subquery += 'chr (trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT)'

**elif** type == 'time without time zone':

select\_subquery += "time '00:00:00' + DATE\_TRUNC('second',RANDOM() \* time '24:00:00')"

**else**:

**continue**

insert\_query += name

**if** i != len(types) - **1**:

select\_subquery += ','

insert\_query += ','

**else**:

insert\_query += ') '

self.\_\_cursor.execute(

insert\_query + "SELECT " + select\_subquery +

"FROM " + from\_query + limit\_query)

self.\_\_context.commit()

Для генерування випадкових даних, для кожного атрибуту в обраній таблиці за спеціальним шаблоном генерується у вигляді рядка частина SQL-запиту, що буде генерувати набір данних певного. Після чого, ці частини поєднуються, після чого отримується готовий запит для генерації данних.

Результат пошуку формується та подається як рядки з’єднання таблиць, атрибути яких використовуються для пошуку, що відповідають умовам пошуку. Запит на виконання операції з’єднання формується та запускається на виконання функцією join\_general, яка приймає в якості аргументів запит та умову пошуку. В свою чергу, кожна з інших трьох функцій формує відповідну умову пошуку, використовуючи обмеження задані користувачем, після чого викликає функцію join\_general, передаючи в якості аргументів текст основного запиту за який відповідальна ця функція, а також умову пошуку, після чого – повертає результат її виконання.

**Результати роботи програми**

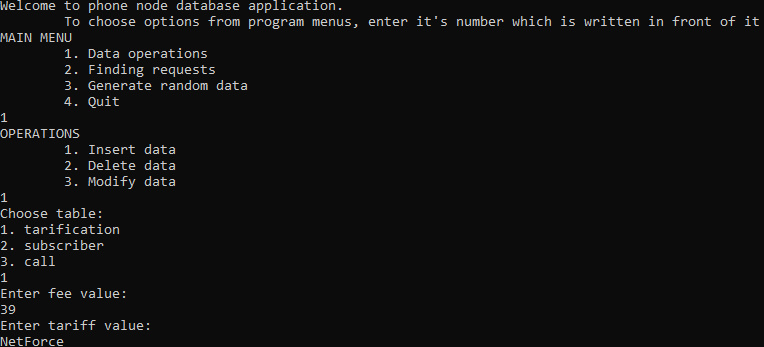


Рисунок 5 – Введення даних

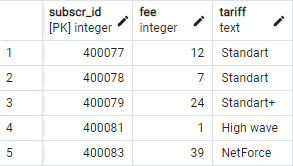


Рисунок 6 – Таблиця ‘tarification’ після внесення до неї даних

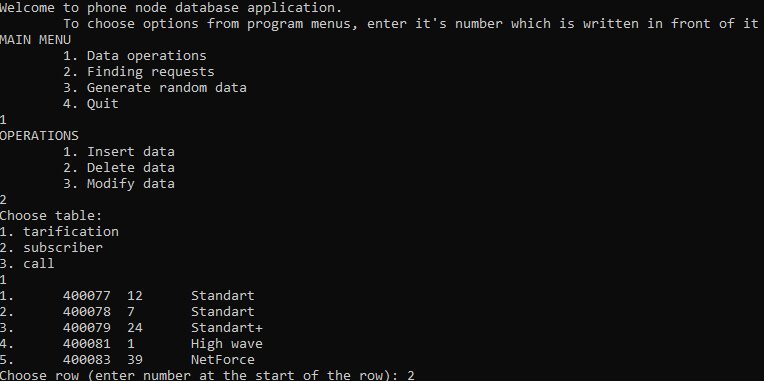


Рисунок 7 – Видалення данних

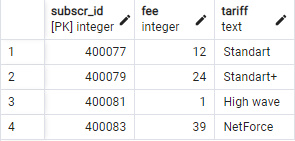


Рисунок 8 – Таблиця ‘tarification’ після видалення даних

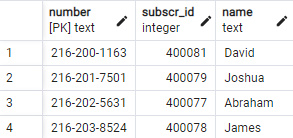
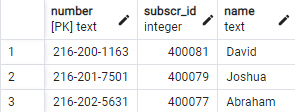


Рисунок 9 – Таблиця ‘subscriber’ до та після вилучення даних у таблиці ‘tarification’

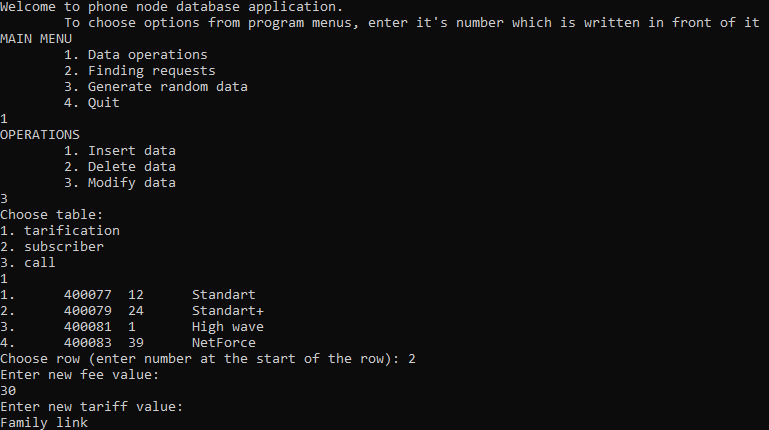


Рисунок 10 – Зміна даних

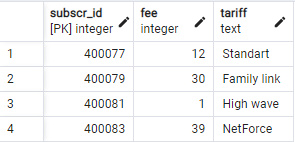


Рисунок 11 – Таблиця ‘tarification’ після зміни даних

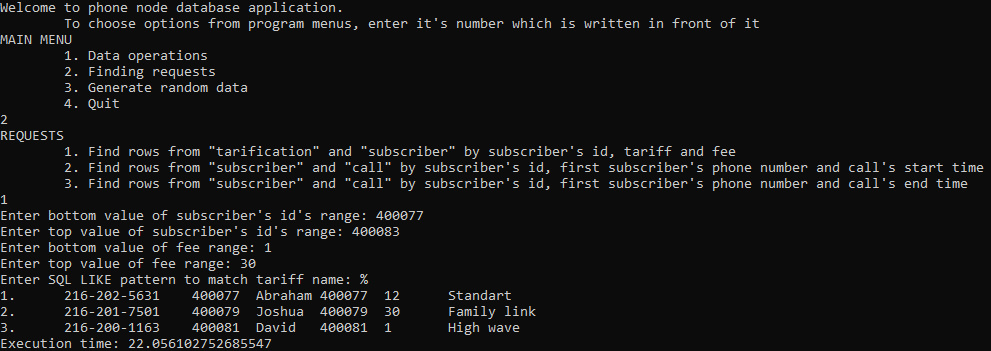


Рисунок 12 – Пошук даних

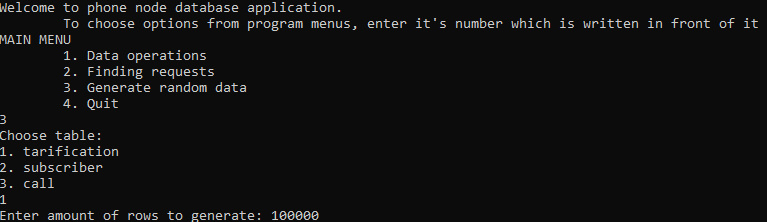


Рисунок 13 – генерація випадкових значень

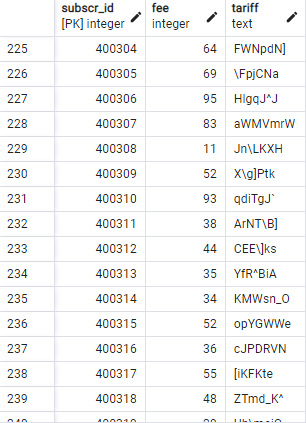


Рисунок 14 – таблиця ‘tarification’ після генерації випадкових значень

**Текст програми**

**Lab2.py**

**from** **Model** **import** db\_model

**from** **View** **import** db\_view

**from** **Controller** **import** db\_controller

m = db\_model("postgres", "postgres", "anonimus", "127.0.0.1")

v = db\_view()

c = db\_controller(m, v)

**Model.py**

**import** **psycopg2**

**from** **psycopg2** **import** sql

**import** **time**

**import** **re**

**class** **db\_model**():

**def** **\_\_init\_\_**(self, dbname, user\_name, password, host):

**try**:

self.\_\_context = psycopg2.connect(dbname=dbname,

user=user\_name,

password=password,

host=host)

self.\_\_cursor = self.\_\_context.cursor()

self.\_\_table\_names = **None**

**except** **Exception** **as** init\_ex:

print("Initialisation error: ", init\_ex)

**return** **None**

**def** **\_\_del\_\_**(self):

self.\_\_cursor.close()

self.\_\_context.close()

**def** **get\_table\_names**(self):

**if** self.\_\_table\_names **is** **None**:

self.\_\_cursor.execute("""

SELECT table\_name

FROM information\_schema.tables

WHERE table\_schema = 'public'

""")

self.\_\_table\_names = [table[**0**] **for** table **in** self.\_\_cursor]

**return** self.\_\_table\_names

**def** **get\_column\_types**(self, table\_name):

self.\_\_cursor.execute("""

SELECT column\_name, data\_type

FROM information\_schema.columns

WHERE table\_schema = 'public' AND table\_name = %s

ORDER BY table\_schema, table\_name

""", (table\_name,))

**return** self.\_\_cursor.fetchall()

**def** **get\_column\_names**(self, table\_name):

self.\_\_cursor.execute("""

SELECT column\_name FROM information\_schema.columns

WHERE table\_schema = 'public' AND table\_name = %s

ORDER BY table\_schema, table\_name

""", (table\_name,))

**return** [x[**0**] **for** x **in** self.\_\_cursor.fetchall()]

**def** **get\_foreign\_key\_info**(self, table\_name):

self.\_\_cursor.execute("""

SELECT kcu.column\_name, ccu.table\_name AS

foreign\_table\_name,

ccu.column\_name AS foreign\_column\_name

FROM information\_schema.table\_constraints AS tc

JOIN information\_schema.key\_column\_usage AS kcu

ON tc.constraint\_name = kcu.constraint\_name

AND tc.table\_schema = kcu.table\_schema

JOIN information\_schema.constraint\_column\_usage AS ccu

ON ccu.constraint\_name = tc.constraint\_name

AND ccu.table\_schema = tc.table\_schema

WHERE tc.constraint\_type = 'FOREIGN KEY' AND

tc.table\_name=%s;

""", (table\_name,))

**return** self.\_\_cursor.fetchall()

**def** **get\_table\_data**(self, table\_name):

id\_column = self.get\_column\_types(table\_name)[**0**][**0**]

cursor = self.\_\_cursor

**try**:

cursor.execute(

sql.SQL('SELECT \* FROM {} ORDER BY {} ASC'

).format(sql.Identifier(table\_name), sql.SQL(id\_column)))

**except** **Exception** **as** e:

**return** str(e)

**return** ([col.name **for** col **in** cursor.description], cursor.fetchall())

**def** **test\_is\_referenced**(self, table\_name, column\_name):

**for** t **in** self.get\_table\_names():

**for** k **in** self.get\_foreign\_key\_info(t):

**if** k[**1**] == table\_name **and** k[**2**] == column\_name:

**return** **True**

**return** **False**

**def** **insert\_data**(self, table\_name, values, columns):

columns\_line = '('

values\_line= '('

**for** key **in** range(len(values)):

**if** values[key]:

columns\_line += columns[key] + ','

values\_line += f"'{values[key]}'" + ','

values\_line = values\_line[:-**1**] + ')'

columns\_line = columns\_line[:-**1**] + ')'

query = sql.SQL('INSERT INTO {table} {columns} VALUES {values}').format(

table = sql.Identifier(table\_name),

columns = sql.SQL(columns\_line),

values = sql.SQL(values\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** ins\_ex:

print(ins\_ex)

**def** **change\_data**(self, table\_name, new\_values, columns, attr, cond):

condition\_line = f"{attr} = '{cond}'"

new\_values\_line = ''

**for** key **in** range(len(new\_values)):

**if** new\_values[key]:

new\_values\_line += f"{columns[key]} = '{new\_values[key]}'" + ','

new\_values\_line = new\_values\_line[:-**1**] + ''

query = sql.SQL('UPDATE {table} SET {new\_values} WHERE {condition}').format(

table = sql.Identifier(table\_name),

new\_values = sql.SQL(new\_values\_line),

condition = sql.SQL(condition\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** chng\_ex:

print(chng\_ex)

**def** **delete\_data**(self, table\_name, attr, cond):

condition\_line = f"{attr} = {cond}"

query = sql.SQL('DELETE FROM {table} WHERE {condition}').format(

table = sql.Identifier(table\_name),

condition = sql.SQL(condition\_line))

**try**:

self.\_\_cursor.execute(query)

self.\_\_context.commit()

**except** **Exception** **as** del\_ex:

print(del\_ex)

**def** **generate\_data**(self, table\_name, count):

types = self.get\_column\_types(table\_name)

fk\_array = self.get\_foreign\_key\_info(table\_name)

select\_subquery = ""

insert\_query = "INSERT INTO " + table\_name + " ("

from\_query = "generate\_series(1," + str(count) + ") as ser"

limit\_query = " LIMIT " + str(count)

pk = self.get\_table\_data(table\_name)[**0**]

pk = str(pk[**0**])

**for** i **in** range(**1** **if** re.match(r'.\*\_id', pk) **else** **0**, len(types)):

t = types[i]

name = t[**0**]

type = t[**1**]

fk = [x **for** x **in** fk\_array **if** x[**0**] == name]

**if** fk:

**if** self.test\_is\_referenced(table\_name, str(fk[**0**][**0**])):

select\_subquery += str(fk[**0**][**2**])

from\_query += ", " + str(fk[**0**][**1**])

**else**:

select\_subquery += ('(SELECT {} FROM {} ORDER BY RANDOM(), ser LIMIT 1)'.format(fk[**0**][**2**], fk[**0**][**1**]))

**elif** type == 'integer':

select\_subquery += 'trunc(random()\*100)::INT'

**elif** type == 'text':

select\_subquery += 'chr (trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT) || chr(trunc(65 + random()\*52)::INT)'

**elif** type == 'time without time zone':

select\_subquery += "time '00:00:00' + DATE\_TRUNC('second',RANDOM() \* time '24:00:00')"

**else**:

**continue**

insert\_query += name

**if** i != len(types) - **1**:

select\_subquery += ','

insert\_query += ','

**else**:

insert\_query += ') '

self.\_\_cursor.execute(

insert\_query + "SELECT " + select\_subquery +

"FROM " + from\_query + limit\_query)

self.\_\_context.commit()

**def** **join\_general**(self, main\_query, condition=""):

new\_cond = condition

**if** condition:

new\_cond = "WHERE " + condition

t1 = time.time()

**try**:

self.\_\_cursor.execute(main\_query.format(new\_cond))

**except** **Exception** **as** find\_ex:

print(find\_ex)

t2 = time.time()

**return** ((t2 - t1) \* **1000**, self.\_\_cursor.fetchall())

**def** **join\_subscr\_id\_fee\_tariff**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, fee\_rng\_start, fee\_rng\_end, tariff):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "t.fee BETWEEN " + str(fee\_rng\_start) + " AND " + str(fee\_rng\_end)

sub\_cond\_3 = "t.tariff LIKE " + "'" + str(tariff) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN tarification as t ON s.subscr\_id = t.subscr\_id {} ORDER BY s.subscr\_id ASC

""", condition)

**def** **join\_subscr\_id\_subscriber1\_start\_time**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, subscriber1, start\_time\_rng\_start, start\_time\_rng\_end):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "c.subscriber1 LIKE " + "'" + str(subscriber1) + "'"

sub\_cond\_3 = "c.start\_time BETWEEN " + "'" + str(start\_time\_rng\_start) + "'" + " AND " + "'" + str(start\_time\_rng\_end) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN call as c ON s.subscr\_id = c.subscriber2 {} ORDER BY s.subscr\_id ASC

""", condition)

**def** **join\_subscr\_id\_subscriber1\_end\_time**(self, sub\_id\_rng\_start, sub\_id\_rng\_end, subscriber1, end\_time\_rng\_start, end\_time\_rng\_end):

sub\_cond\_1 = "s.subscr\_id BETWEEN " + str(sub\_id\_rng\_start) + " AND " + str(sub\_id\_rng\_end)

sub\_cond\_2 = "c.subscriber1 LIKE " + "'" + str(subscriber1) + "'"

sub\_cond\_3 = "c.end\_time BETWEEN " + "'" + str(end\_time\_rng\_start) + "'" + " AND " + "'" + str(end\_time\_rng\_end) + "'"

condition = sub\_cond\_1 + " AND " + sub\_cond\_2 + " AND " + sub\_cond\_3

**return** self.join\_general(

"""SELECT \* FROM subscriber as s JOIN call as c ON s.subscr\_id = c.subscriber2 {} ORDER BY s.subscr\_id ASC

""", condition)

**View.py**

**class** **db\_view**():

**def** **starting\_dialog**(self):

print("""Welcome to phone node database application.

To choose options from program menus, enter it's number which is written in front of it""")

**def** **main\_menu**(self):

print("""MAIN MENU

1. Data operations

2. Finding requests

3. Generate random data

4. Quit""")

**def** **operations\_menu**(self):

print("""OPERATIONS

1. Insert data

2. Delete data

3. Modify data""")

**def** **finding\_requests\_menu**(self):

print("""REQUESTS

1. Find rows from "tarification" and "subscriber" by subscriber's id, tariff and fee

2. Find rows from "subscriber" and "call" by subscriber's id, first subscriber's phone number and call's start time

3. Find rows from "subscriber" and "call" by subscriber's id, first subscriber's phone number and call's end time""")

**def** **table\_menu**(self, tables):

print("""Choose table:""")

**for** i **in** range(len(tables)):

print(f"{i+1}.", tables[i])

**def** **show\_rows**(self, rows):

number = **1**

**for** line **in** rows:

string = f"{number}.**\t**"

**for** elem **in** range(len(line)):

string += str(line[elem]) + "**\t**"

print(string)

number += **1**

**Controller.py**

**import** **re**

**class** **db\_controller**():

**def** **\_\_init\_\_**(self, model, view):

self.\_\_model = model

self.\_\_view = view

self.main\_process()

**def** **insert**(self):

table\_names = self.\_\_model.get\_table\_names()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

table\_data = self.\_\_model.get\_table\_data(table)

**if** re.match(r'.\*\_id', table\_data[**0**][**0**]):

table\_data[**0**].pop(**0**)

columns = table\_data[**0**]

values = []

**for** i **in** range(len(columns)):

print("Enter " + columns[i] + " value:")

values.append(input())

self.\_\_model.insert\_data(table, tuple(values), tuple(columns))

**def** **delete**(self):

table\_names = self.\_\_model.get\_table\_names()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

table\_data = self.\_\_model.get\_table\_data(table)

self.\_\_view.show\_rows(table\_data[**1**])

self.\_\_model.delete\_data(table, table\_data[**0**][**0**], table\_data[**1**][int(input('Choose row (enter number at the start of the row): '))-**1**][**0**])

**def** **modify**(self):

table\_names = self.\_\_model.get\_table\_names()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

table\_data = self.\_\_model.get\_table\_data(table)

self.\_\_view.show\_rows(table\_data[**1**])

attribute = table\_data[**0**][**0**]

row\_number = table\_data[**1**][int(input('Choose row (enter number at the start of the row): '))-**1**][**0**]

**if** re.match(r'.\*\_id', table\_data[**0**][**0**]):

table\_data[**0**].pop(**0**)

values = []

**for** i **in** range(len(table\_data)):

print("Enter new " + table\_data[**0**][i] + " value:")

values.append(input())

self.\_\_model.change\_data(table, values, table\_data[**0**], attribute, row\_number)

**def** **rand\_gen**(self):

table\_names = self.\_\_model.get\_table\_names()

self.\_\_view.table\_menu(table\_names)

table = int(input())

table = table\_names[table - **1**]

self.\_\_model.generate\_data(table, int(input("Enter amount of rows to generate: ")))

**def** **find\_sub\_id\_fee\_tariff**(self):

sub\_id\_bottom = input("Enter bottom value of subscriber's id's range: ")

sub\_id\_top = input("Enter top value of subscriber's id's range: ")

fee\_bottom = input("Enter bottom value of fee range: ")

fee\_top = input("Enter top value of fee range: ")

tariff\_pattern = input("Enter SQL LIKE pattern to match tariff name: ")

result = self.\_\_model.join\_subscr\_id\_fee\_tariff(sub\_id\_bottom, sub\_id\_top, fee\_bottom, fee\_top, tariff\_pattern)

self.\_\_view.show\_rows(result[**1**])

print("Execution time: " + str(result[**0**]))

**def** **find\_sub\_id\_sub1\_start\_time**(self):

sub\_id\_bottom = input("Enter bottom value of subscriber's id's range: ")

sub\_id\_top = input("Enter top value of subscriber's id's range: ")

sub1 = input("Enter SQL LIKE pattern to match first subscriber's phone number: ")

start\_time\_bottom = input("Enter bottom value of calls starting times range: ")

start\_time\_top = input("Enter top value of calls starting times range: ")

result = self.\_\_model.join\_subscr\_id\_fee\_tariff(sub\_id\_bottom, sub\_id\_top, sub1, start\_time\_bottom, start\_time\_top)

self.\_\_view.show\_rows(result[**1**])

print("Execution time: " + str(result[**0**]))

**def** **find\_sub\_id\_sub1\_end\_time**(self):

sub\_id\_bottom = input("Enter bottom value of subscriber's id's range: ")

sub\_id\_top = input("Enter top value of subscriber's id's range: ")

sub1 = input("Enter SQL LIKE pattern to match first subscriber's phone number: ")

end\_time\_bottom = input("Enter bottom value of calls ending times range: ")

end\_time\_top = input("Enter top value of calls ending times range: ")

result = self.\_\_model.join\_subscr\_id\_fee\_tariff(sub\_id\_bottom, sub\_id\_top, sub1, end\_time\_bottom, end\_time\_top)

self.\_\_view.show\_rows(result[**1**])

print("Execution time: " + str(result[**0**]))

**def** **main\_process**(self):

self.\_\_view.starting\_dialog()

**while** **1**:

self.\_\_view.main\_menu()

action = int(input())

**if** action == **4**:

**break**

**elif** action == **1**:

self.\_\_view.operations\_menu()

operation = int(input())

**if** operation == **1**:

self.insert()

**elif** operation == **2**:

self.delete()

**elif** operation == **3**:

self.modify()

**elif** action == **2**:

self.\_\_view.finding\_requests\_menu()

request = int(input())

**if** request == **1**:

self.find\_sub\_id\_fee\_tariff()

**elif** request == **2**:

self.find\_sub\_id\_sub1\_start\_time()

**elif** request == **3**:

self.find\_sub\_id\_sub1\_end\_time()

**elif** action == **3**:

self.rand\_gen()