A black and white drawing of a building

Description automatically generated

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського» ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

# Кафедра системного програмування та спеціалізованих комп’ютерних систем

**Розрахунково-графічна робота**

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-21

Юдін Дмитро

Перевірив:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Київ – 2024

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

**Виконання роботи**

Нижче наведені сутності предметної області «Electronic database for hostel clients» та зв’язки між ними.

GitHub репозиторій: https://github.com/montrealkiss/rgr

**Сутності предметної області**

Предметна галузь «Electronic database for hostel clients» включає в себе 3 сутності, кожна сутність містить по 3 атрибути:

1. Client (client\_id, name, phone\_number).
2. Booking (booking\_id, booking\_status, date, client\_id, room\_id).
3. Room (room\_id, type, room\_status).

Сутність Client описує клієнта хостелу. Кожен клієнт має свій ідентифікатор, ім’я та номер мобільного телефону для зв’язку.

Сутність Booking описує бронювання номеру. Кожне бронювання має ідентифікатор бронювання, статус бронювання та час до якого заброньоване перебування в хостелі.

Сутність Room описує кімнату в хостелі. Кожна кімната має свій ідентифікатор, тип кімнати (кількість кімнат), та статус кімнати (вільна, зайнята, потребує прибирання).

**Зв’язки між сутностями предметної області**

Зв’язок між Client та Booking:

Один клієнт може мати кілька бронювань. Це означає, що клієнт може робити різні бронювання в різний час. Одне бронювання належить лише одному клієнту. Зв’язок 1:N.

Зв’язок між Booking та Room:

Одне бронювання стосується конкретного номера. Один номер може бути пов'язаний з багатьма бронюваннями, але кожне бронювання відбувається на різні дати. Зв’язок 1:N.

Графічне подання концептуальної моделі «Сутність-зв’язок» зображено на рисунку 1.

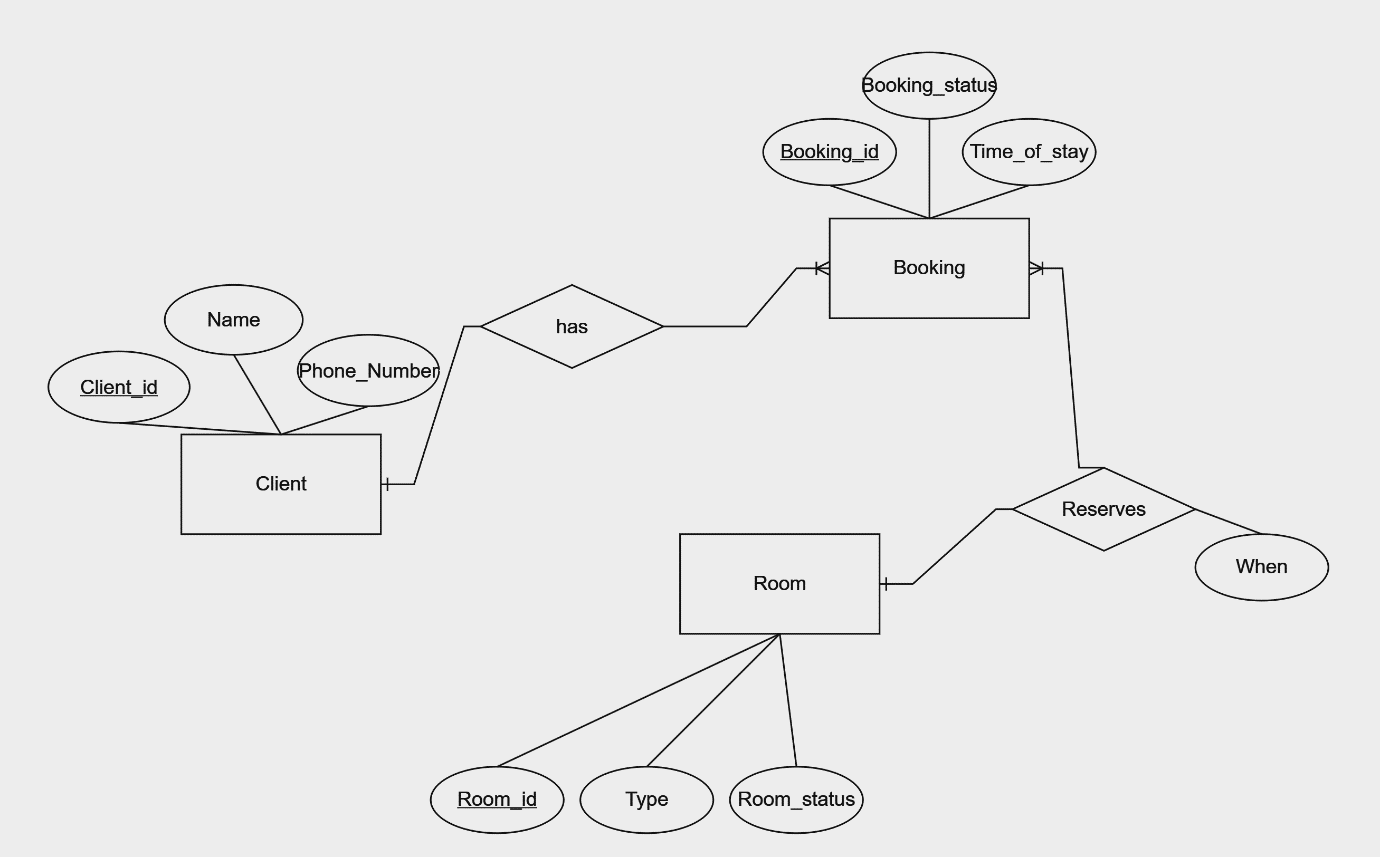


Рисунок 1. ER-діаграма побудована за нотацією Чена

Графічне подання логічної моделі «Сутність-зв’язок» зображено на рисунку 2:

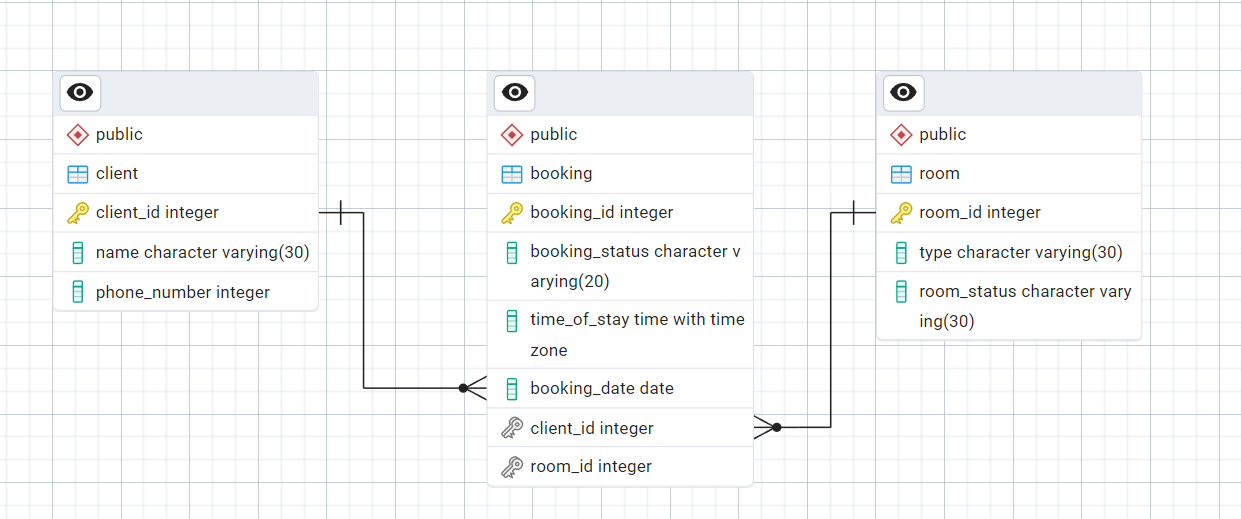
****

Рисунок 2. Схема бази даних у pgAdmin4

**Середовище та компоненти розробки**

У процесі розробки була використана мова програмування Python, інтегроване середовище розробки PyCharm, а також була використана бібліотека psycopg2, яка надає API для взаємодії з базою даних PostgreSQL.

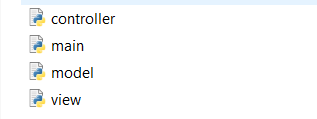
**Шаблон проектування**

Модель-представлення-контролер (MVC) – це шаблон проектування, що використовується у програмі. Кожен компонент відповідає за певну функціональну частину:

1. Модель (Model) – це клас, що відображає логіку роботи з даними, обробляє всі операції з даними, такі як додавання, оновлення, вилучення.
2. Представлення (View) – це клас, через який користувач взаємодіє з програмою. У даному випадку, консольний інтерфейс, який відображає дані для користувача та зчитує їх з екрану.
3. Контролер (Controller) – це клас, який відповідає за зв’язок між користувачем і системою. Він приймає введені користувачем дані та обробляє їх. В залежності від результатів, викликає відповідні дії з Model або View.

Даний підхід дозволяє розділити логіку програми на логічні компоненти, що полегшує розробку, тестування і підтримку продукту.

**Структура програми та її опис**

****

З файлу main.py відбувається виклик контролера та передача йому управління.

У файлі model.py описаний клас моделі, який відповідає за управління підключенням до бази даних і виконанням низькорівневих запитів до неї.

У файлі controller.py реалізовано інтерфейс взаємодії з користувачем, включаючи обробку запитів користувача, виконання пошуку, а також інші дії, необхідні для взаємодії з моделлю та представленням.

У файлі view.py описаний клас, який відображає результати виконання різних дій користувача на екрані консолі. Цей компонент відповідає за представлення даних користувачу в зручному для сприйняття вигляді.

Отже, структура програми відповідає патерну MVC.

**Структура меню програми**

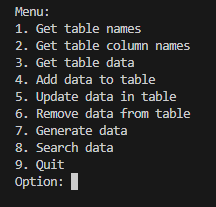


Рисунок 3 – Структура меню користувача

**Опис функціональності пунктів меню користувача**

**1 Get table names**

Отримання списку всіх таблиць у базі даних.  
Користувач може переглянути доступні таблиці в базі для подальших операцій.

**2 Get table column names**

Перегляд списку назв колонок конкретної таблиці.  
Ця функція дозволяє користувачу дізнатися структуру вибраної таблиці.

**3 Get table data**

Отримання та перегляд даних з таблиці.  
Користувач може вивести на екран всі або вибрані записи з обраної таблиці.

**4 Add data to table**

Додавання нових записів у таблицю.  
Забезпечує внесення даних, перевіряючи їх відповідність типам колонок таблиці.

**5 Update data in table**

Оновлення наявних записів у таблиці.  
Дозволяє редагувати дані в конкретних рядках таблиці на основі заданих умов.

**6 Remove data from table**

Видалення записів із таблиці.  
Користувач може видаляти рядки таблиці за заданими умовами, з перевіркою на зв’язки з іншими таблицями.

**7 Generate data**

Автоматичне генерування випадкових даних у таблиці.  
Генерує великий обсяг тестових даних із використанням функцій PostgreSQL, таких як generate\_series і random().

1. **Search data**

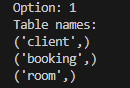
Пошук по значенню поля таблиці з використанням GROUP BY і ORDER BY для бажаного користувачем сортування даних

1. **Quit**

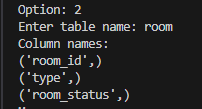
Вихід із програми.  
Завершує роботу програми та закриває з’єднання з базою даних.

**Тестування роботи функцій програми в різних ситуаціях**

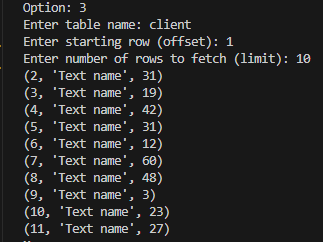
Перегляд назв таблиць (опція 1):



Перегляд назв колонок таблиці (опція 2):

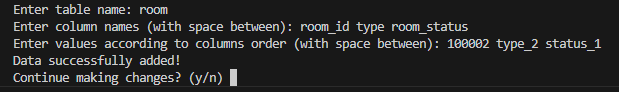


Перегляд даних таблиці (опція 3):

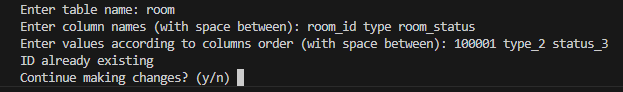


Додавання даних до таблиці (опція 4):

* Успішне додавання:

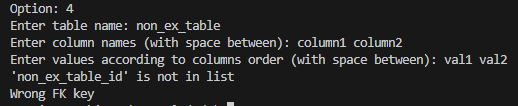


* Спроба додати з уже існуючим унік. ідент.:



(Виведено повідомлення про те, що рядок з таким ідент. уже існує, програма запитує користувача чи хоче він продовжити вносити зміни)

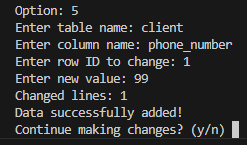
* Спроба додати дані вказуючи неіснуючі назви (таблиць, колонок)



(Виведено повідомлення про те, не існує такої таблиці / колонок, програма продовжує роботу)

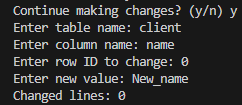
Оновлення даних (опція 5)

* Успішне оновлення:



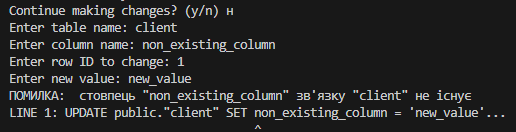
Повідомлення про успішне оновленя

* Оновлення рядка з неіснуючим ідент.:



Виведено повідомлення про те, що не було змінено жодного рядка

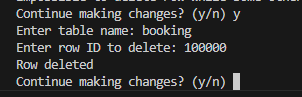
* Оновлення значення неіснуючого поля:



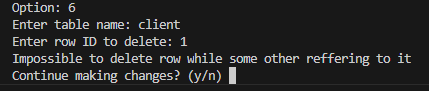
Виведено повідомлення про помилку, програма продовжує роботу запитуючи користувача, чи бажає він продовжувати вносити зміни за цим пунктом меню.

Видалення даних (опція 6):

* Успішне видалення:

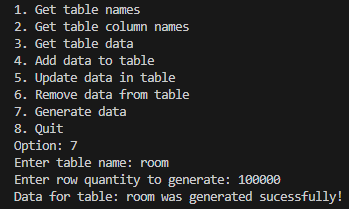


* Видалення рядка, на який посилаються інші рядки, інших таблиць:

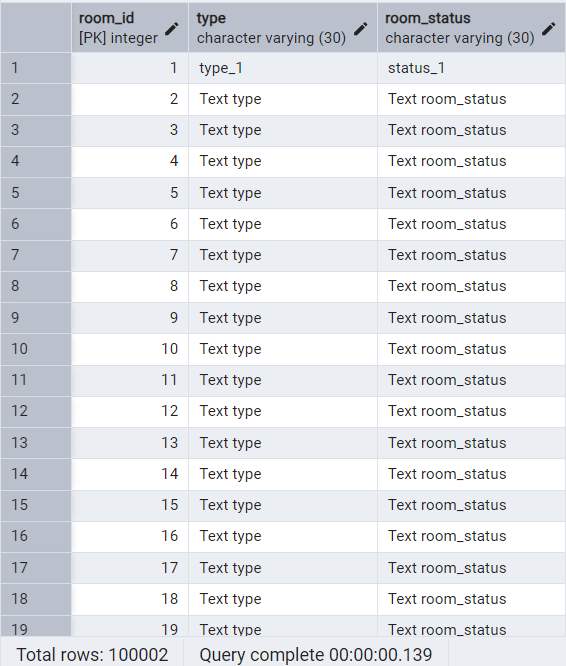


(Виведено повідомлення про те, що видалення даного рядка неможливе, поки на нього посилаються інші рядки інших таблиць)

Генерування даних:

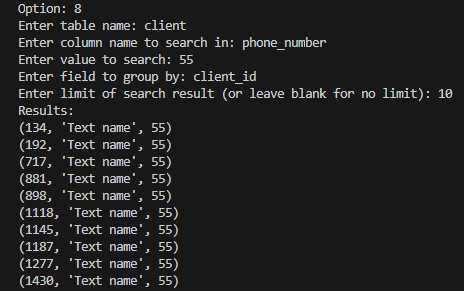


Виведено повідомлення про успішну генерацію даних



Перевіряючи чи були згенеровані рядки у PgAdmin4 бачимо, що їх кількість тпепр 100.002 (2 записи були в таблиці до генерації)

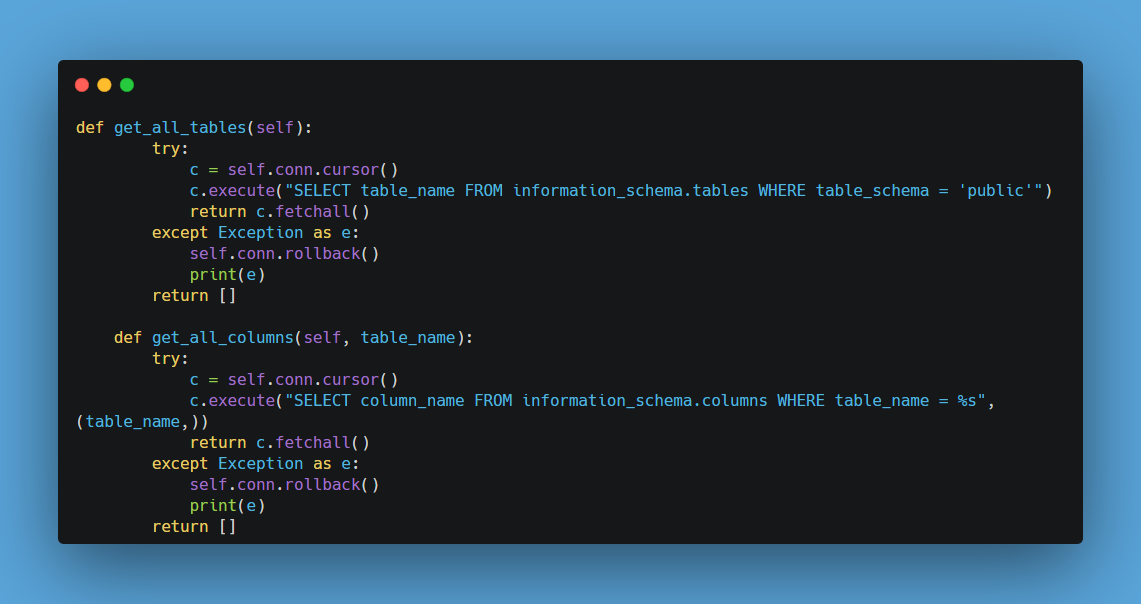
Пошук по даним:



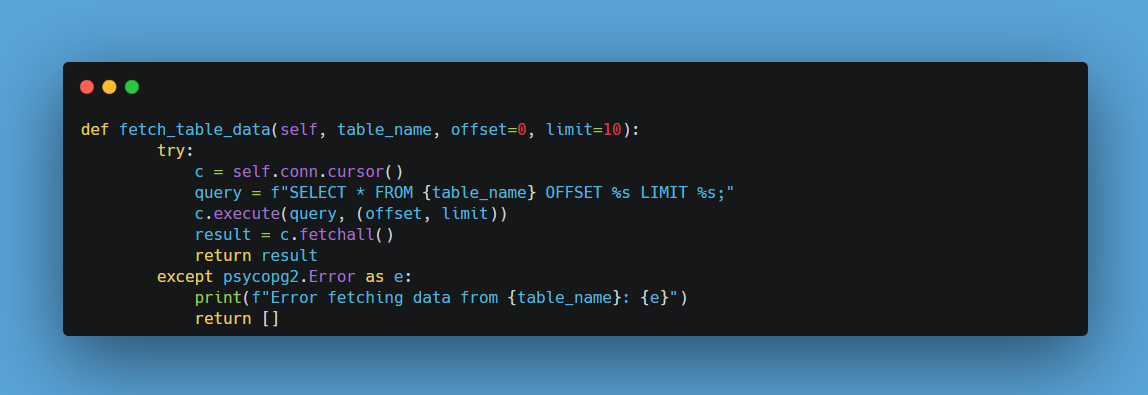
**Фрагменти коду:**

Нижче наведені деякі приклади що ілюструють фукнціональні можливості додатку.

Отримання назв таблиць і колонок:



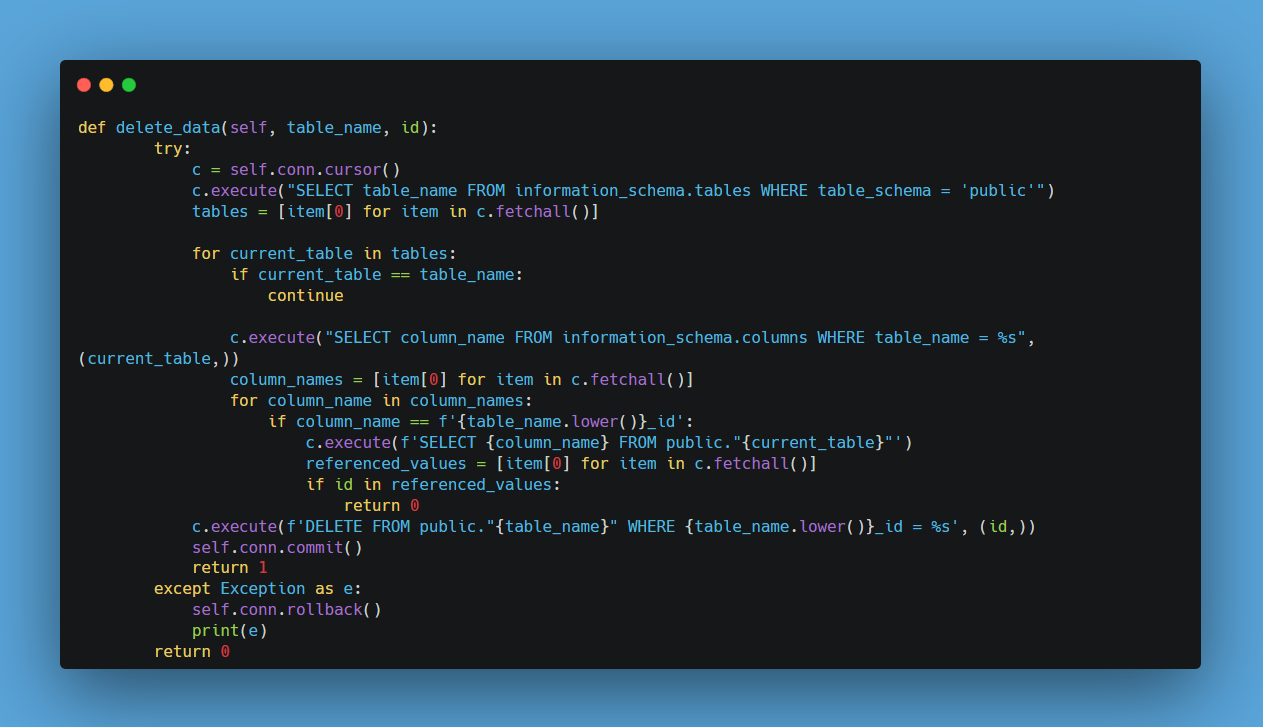
Отримання даних з таблиці



Додавання даних до таблиць:



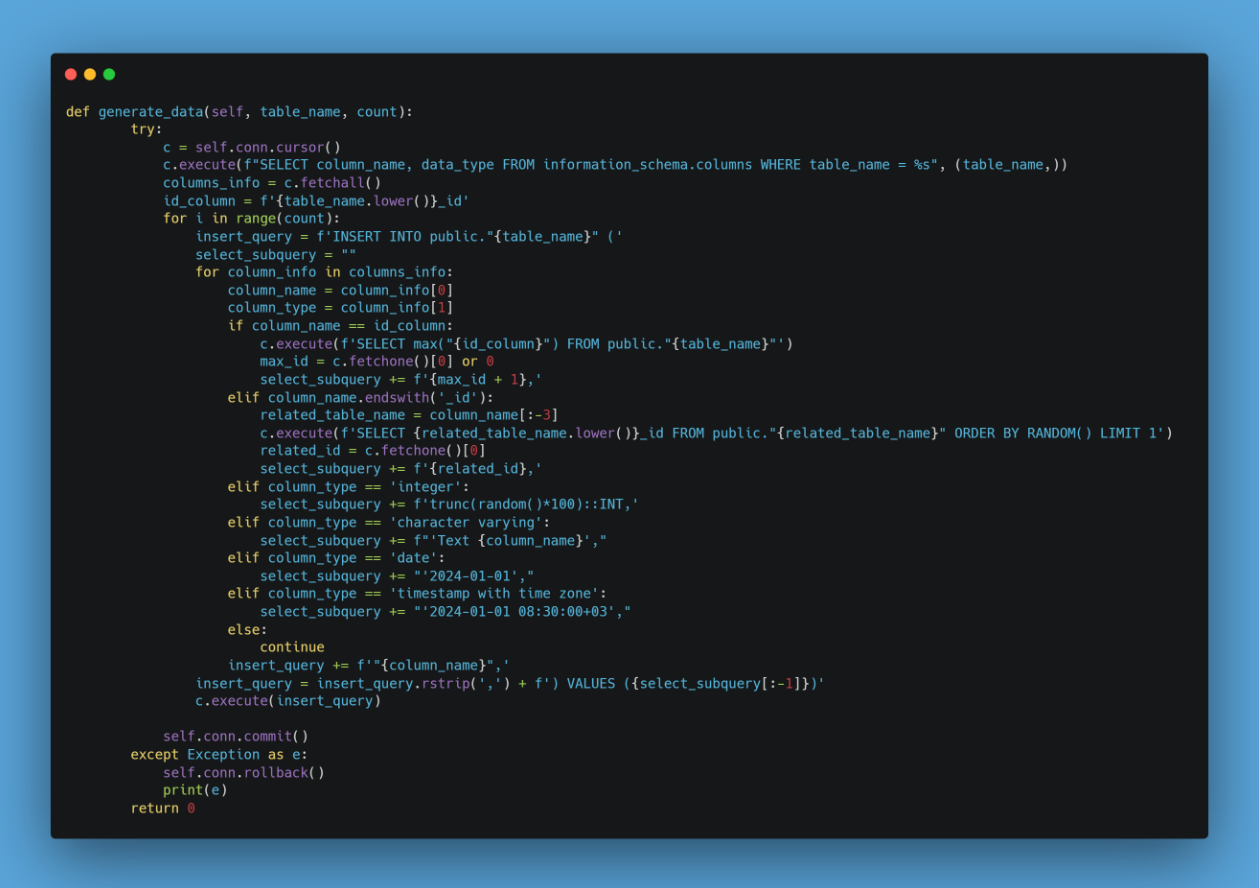
Видалення рядків з таблиць:



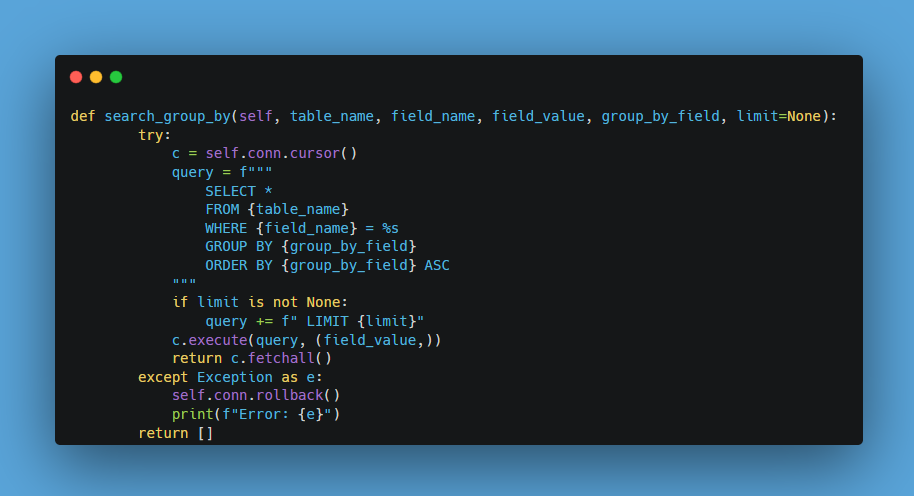
Оновлення даних:



Генерування рядків:



Пошук

****

**Код програми**

**main.py**

from controller import Controller

if \_\_name\_\_ == "\_\_main\_\_":

controller = Controller()

controller.run()

**controller.py**

import sys

from model import Model

from view import View

class Controller:

def \_\_init\_\_(self):

self.view = View()

try:

self.model = Model()

self.view.show\_message("Connecting to DB...")

except Exception as e:

self.view.show\_message(f"Error druing initializtion {e}")

sys.exit(1)

def run(self):

while True:

choice = self.view.show\_menu()

if choice == '1':

self.view\_tables()

elif choice == '2':

self.view\_columns()

elif choice == '3':

self.view\_table\_data()

elif choice == '4':

self.add\_data()

elif choice == '5':

self.update\_data()

elif choice == '6':

self.delete\_data()

elif choice == '7':

self.generate\_data()

elif choice == '8':

break

def view\_tables(self):

tables = self.model.get\_all\_tables()

self.view.show\_tables(tables)

def view\_columns(self):

table\_name = self.view.ask\_table()

columns = self.model.get\_all\_columns(table\_name)

self.view.show\_columns(columns)

def view\_table\_data(self):

table\_name = self.view.ask\_table()

offset = int(self.view.inp("Enter starting row (offset): "))

limit = int(self.view.inp("Enter number of rows to fetch (limit): "))

data = self.model.fetch\_table\_data(table\_name, offset, limit)

self.view.show\_table\_data(data)

def get\_search\_group\_by\_input(self):

try:

table\_name = self.view.inp("Enter table name: ")

field\_name = self.view.inp("Enter column name to search in: ")

field\_value = self.view.inp("Enter value to search: ")

group\_by\_field = self.view.inp("Enter field to group by: ")

limit = input("Enter limit of search result (or leave blank for no limit): ").strip()

limit = int(limit) if limit == '' else None

results = self.model.search\_group\_by(table\_name, field\_name, field\_value, group\_by\_field)

if results:

self.view.show\_message("Results:")

self.view.show\_table\_data(results)

else:

self.view.show\_message("Not found")

except Exception as e:

self.view.show\_message(f"Error: {e}")

def add\_data(self):

while True:

table, columns, val = self.view.insert()

error = self.model.add\_data(table, columns, val)

if int(error) == 1:

self.view.show\_message("Data successfully added!")

agree = self.view.ask\_continue()

if agree == 'n':

break

elif int(error) == 2:

self.view.show\_message("ID already existing")

agree = self.view.ask\_continue()

if agree == 'n':

break

else:

self.view.show\_message("Wrong FK key")

agree = self.view.ask\_continue()

if agree == 'n':

break

def update\_data(self):

while True:

table, column, id, new\_value = self.view.update()

error = self.model.update\_data(table, column, id, new\_value)

if int(error) == 1:

self.view.show\_message("Data successfully added!")

agree = self.view.ask\_continue()

if agree == 'n':

break

elif int(error) == 2:

self.view.show\_message(f"ID {new\_value} already existing")

agree = self.view.ask\_continue()

if agree == 'n':

break

else:

self.view.show\_message(f"Wrong FK key: {new\_value} in column: {column}")

agree = self.view.ask\_continue()

if agree == 'n':

break

def delete\_data(self):

while True:

table, id = self.view.delete()

error = self.model.delete\_data(table, id)

if int(error) == 1:

self.view.show\_message("Row deleted")

agree = self.view.ask\_continue()

if agree == 'n':

break

else:

self.view.show\_message("Impossible to delete row while some other reffering to it")

agree = self.view.ask\_continue()

if agree == 'n':

break

def generate\_data(self):

table\_name, num\_rows = self.view.generate\_data\_input()

self.model.generate\_data(table\_name, num\_rows)

self.view.show\_message(f"Data for table: {table\_name} was generated sucessfully!")

**view.py**

import time

class View:

def show\_menu(self):

while True:

print("Menu:")

print("1. Get table names")

print("2. Get table column names ")

print("3. Get table data")

print("4. Add data to table")

print("5. Update data in table")

print("6. Remove data from table")

print("7. Generate data")

print("8. Search data")

print("9. Quit")

choice = input("Option: ")

if choice in ('1', '2', '3', '4', '5', '6', '7', '8', '9'):

return choice

else:

print("Please enter correct choice in range (from 1 to 8)")

time.sleep(2)

def show\_message(self, message):

print(message)

time.sleep(2)

def ask\_continue(self):

agree = input("Continue making changes? (y/n) ")

return agree

def show\_tables(self, tables):

print("Table names:")

for table in tables:

print(table)

time.sleep(2)

def ask\_table(self):

table\_name = input("Enter table name: ")

return table\_name

def inp(self, message):

inp = input(message)

return inp

def show\_table\_data(self, data):

if not data:

print("No data found or an error occurred.")

else:

for row in data:

print(row)

def show\_columns(self, columns):

print("Column names:")

for column in columns:

print(column)

time.sleep(2)

def insert(self):

while True:

try:

table = input("Enter table name: ")

columns = input("Enter column names (with space between): ").split()

val = input("Enter values according to columns order (with space between): ").split()

if len(columns) != len(val):

raise ValueError("Columns and values aren't matching")

return table, columns, val

except ValueError as e:

print(f"Error: {e}")

def update(self):

while True:

try:

table = input("Enter table name: ")

column = input("Enter column name: ")

id = int(input("Enter row ID to change: "))

new\_value = input("Enter new value: ")

return table, column, id, new\_value

except ValueError as e:

print(f"Error: {e}")

def delete(self):

while True:

try:

table = input("Enter table name: ")

id = int(input("Enter row ID to delete: "))

return table, id

except ValueError as e:

print(f"Error: {e}")

def generate\_data\_input(self):

while True:

try:

table\_name = input("Enter table name: ")

num\_rows = int(input("Enter row quantity to generate: "))

return table\_name, num\_rows

except ValueError as e:

print(f"Error: {e}")

**model.py**

import psycopg2

class Model:

def \_\_init\_\_(self):

self.conn = psycopg2.connect(

dbname='hr',

user='postgres',

password='postgres',

host='localhost',

port=5432

)

def get\_all\_tables(self):

try:

c = self.conn.cursor()

c.execute("SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'public'")

return c.fetchall()

except Exception as e:

self.conn.rollback()

print(e)

return []

def get\_all\_columns(self, table\_name):

try:

c = self.conn.cursor()

c.execute("SELECT column\_name FROM information\_schema.columns WHERE table\_name = %s", (table\_name,))

return c.fetchall()

except Exception as e:

self.conn.rollback()

print(e)

return []

def fetch\_table\_data(self, table\_name, offset=0, limit=10):

try:

c = self.conn.cursor()

query = f"SELECT \* FROM {table\_name} OFFSET %s LIMIT %s;"

c.execute(query, (offset, limit))

result = c.fetchall()

return result

except psycopg2.Error as e:

print(f"Error fetching data from {table\_name}: {e}")

return []

def add\_data(self, table\_name, columns, val):

try:

c = self.conn.cursor()

columns\_str = ', '.join(columns)

placeholders = ', '.join(['%s'] \* len(val))

identifier\_column = f'{table\_name.lower()}\_id'

identifier\_index = columns.index(identifier\_column)

val[identifier\_index] = int(val[identifier\_index])

c.execute(f'SELECT {identifier\_column} FROM public."{table\_name}"')

existing\_identifiers = [item[0] for item in c.fetchall()]

if val[identifier\_index] in existing\_identifiers:

return 2

external\_keys = [column for column in columns if column.endswith('\_id') and column != identifier\_column]

for key\_column in external\_keys:

referenced\_table = key\_column[:-3].capitalize()

c.execute(f'SELECT {key\_column} FROM public."{referenced\_table}"')

referenced\_values = [item[0] for item in c.fetchall()]

if int(val[columns.index(key\_column)]) not in referenced\_values:

return 3

c.execute(f'INSERT INTO public."{table\_name}" ({columns\_str}) VALUES ({placeholders})', val)

self.conn.commit()

return 1

except Exception as e:

self.conn.rollback()

print(e)

return 0

def search\_group\_by(self, table\_name, field\_name, field\_value, group\_by\_field, limit=None)

try:

c = self.conn.cursor()

query = f"""

SELECT \*

FROM {table\_name}

WHERE {field\_name} = %s

GROUP BY {group\_by\_field}

ORDER BY {group\_by\_field} ASC

"""

if limit is not None:

query += f" LIMIT {limit}"

c.execute(query, (field\_value,))

return c.fetchall()

except Exception as e:

self.conn.rollback()

print(f"Error: {e}")

return []

def update\_data(self, table\_name, column, id, new\_value):

try:

c = self.conn.cursor()

identifier\_column = f'{table\_name.lower()}\_id'

is\_unique\_identifier = identifier\_column == column

if is\_unique\_identifier:

c.execute(f'SELECT {identifier\_column} FROM public."{table\_name}"')

existing\_identifiers = [item[0] for item in c.fetchall()]

if int(new\_value) in existing\_identifiers:

return 2

elif column.endswith('\_id'):

referenced\_table = column[:-3].capitalize()

c.execute(f'SELECT {column} FROM public."{referenced\_table}"')

referenced\_values = [item[0] for item in c.fetchall()]

if int(new\_value) not in referenced\_values:

return 3

c.execute(f'UPDATE public."{table\_name}" SET {column} = %s WHERE {identifier\_column} = %s',

(new\_value, id))

self.conn.commit()

print(f"Changed lines: {c.rowcount}")

return 1

except Exception as e:

self.conn.rollback()

print(e)

return 0

def delete\_data(self, table\_name, id):

try:

c = self.conn.cursor()

c.execute("SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'public'")

tables = [item[0] for item in c.fetchall()]

for current\_table in tables:

if current\_table == table\_name:

continue

c.execute("SELECT column\_name FROM information\_schema.columns WHERE table\_name = %s", (current\_table,))

column\_names = [item[0] for item in c.fetchall()]

for column\_name in column\_names:

if column\_name == f'{table\_name.lower()}\_id':

c.execute(f'SELECT {column\_name} FROM public."{current\_table}"')

referenced\_values = [item[0] for item in c.fetchall()]

if id in referenced\_values:

return 0

c.execute(f'DELETE FROM public."{table\_name}" WHERE {table\_name.lower()}\_id = %s', (id,))

self.conn.commit()

return 1

except Exception as e:

self.conn.rollback()

print(e)

return 0

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

try:

c = self.conn.cursor()

c.execute(f"SELECT column\_name, data\_type FROM information\_schema.columns WHERE table\_name = %s", (table\_name,))

columns\_info = c.fetchall()

id\_column = f'{table\_name.lower()}\_id'

for i in range(count):

insert\_query = f'INSERT INTO public."{table\_name}" ('

select\_subquery = ""

for column\_info in columns\_info:

column\_name = column\_info[0]

column\_type = column\_info[1]

if column\_name == id\_column:

c.execute(f'SELECT max("{id\_column}") FROM public."{table\_name}"')

max\_id = c.fetchone()[0] or 0

select\_subquery += f'{max\_id + 1},'

elif column\_name.endswith('\_id'):

related\_table\_name = column\_name[:-3]

c.execute(f'SELECT {related\_table\_name.lower()}\_id FROM public."{related\_table\_name}" ORDER BY RANDOM() LIMIT 1')

related\_id = c.fetchone()[0]

select\_subquery += f'{related\_id},'

elif column\_type == 'integer':

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

elif column\_type == 'character varying':

select\_subquery += f"'Text {column\_name}',"

elif column\_type == 'date':

select\_subquery += "'2024-01-01',"

elif column\_type == 'timestamp with time zone':

select\_subquery += "'2024-01-01 08:30:00+03',"

else:

continue

insert\_query += f'"{column\_name}",'

insert\_query = insert\_query.rstrip(',') + f') VALUES ({select\_subquery[:-1]})'

c.execute(insert\_query)

self.conn.commit()

except Exception as e:

self.conn.rollback()

print(e)

return 0