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 ІІI курсу

групи КВ-22

Ковкін В. В.

Перевірив:

Павловский В. І.

Київ – 2024

**Мета:** здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

**Виконання роботи**

Нижче наведені сутності предметної області «Відвідуваність занять студентами» та зв’язки між ними.

**Сутності предметної області**

Для побудови бази даних обраної області, були виділені такі сутності:

1. **Громадянин (Citizen)**  
   Атрибути: ідентифікатор громадянина, ім’я, номер телефону, адреса.  
   Призначення: збереження даних щодо громадян, які потребують вакцинації.
2. **Вакцинація (Vaccination)**  
   Атрибути: ідентифікатор вакцинації, ідентифікатор громадянина, ідентифікатор лікаря, ідентифікатор вакцини, ідентифікатор медичного закладу, дата вакцинації.  
   Призначення: збереження даних про кожну здійснену вакцинацію.
3. **Вакцина (Vaccine)**  
   Атрибути: ідентифікатор вакцини, кількість необхідних доз.  
   Призначення: збереження інформації про вакцини.
4. **Лікар (Doctor)**  
   Атрибути: ідентифікатор лікаря, ім’я, номер телефону.  
   Призначення: збереження даних щодо лікарів, які здійснюють вакцинацію.
5. **Медичний заклад (Clinic)**  
   Атрибути: ідентифікатор медичного закладу, адреса.  
   Призначення: збереження даних про медичні заклади, в яких працюють лікарі.
6. **Лікарня-Лікар (Doctor\_Clinic)**  
   Атрибути: ідентифікатор відповідності, ідентифікатор лікаря, ідентифікатор медичного закладу.  
   Призначення: збереження інформації про відповідність лікаря та медичного закладу.
7. **Вакцина-Медичний заклад (Vaccine\_Clinic)**  
   Атрибути: ідентифікатор таблиці, ідентифікатор вакцини, ідентифікатор медичного закладу.  
   Призначення: збереження інформації про наявність вакцин у медичних закладах.

**Зв’язки між сутностями предметної області**

Один медичний заклад може мати багато лікарів і один лікар може працювати в різних медичних закладах. Тому між сутностями Медичний заклад і Лікар існує зв’язок M:N.

В одному медичному закладі може бути доступно багато вакцин і також вакцини можуть зберігатися у різних медичних закладах. Тому між сутностями Медичний заклад і Вакцина існує зв'язок M:N.

Для вакцинації кожен раз використовується один громадянин, одна вакцина, один лікар і один медичний заклад. Для цього зв’язку додана сутність **Вакцинація**. **Вакцинація** це сутність-зв’язок 4-х сутностей **Громадянин – Вакцина – Лікар – Лікарня**.

Графічне подання концептуальної моделі зображено на рисунку 1.

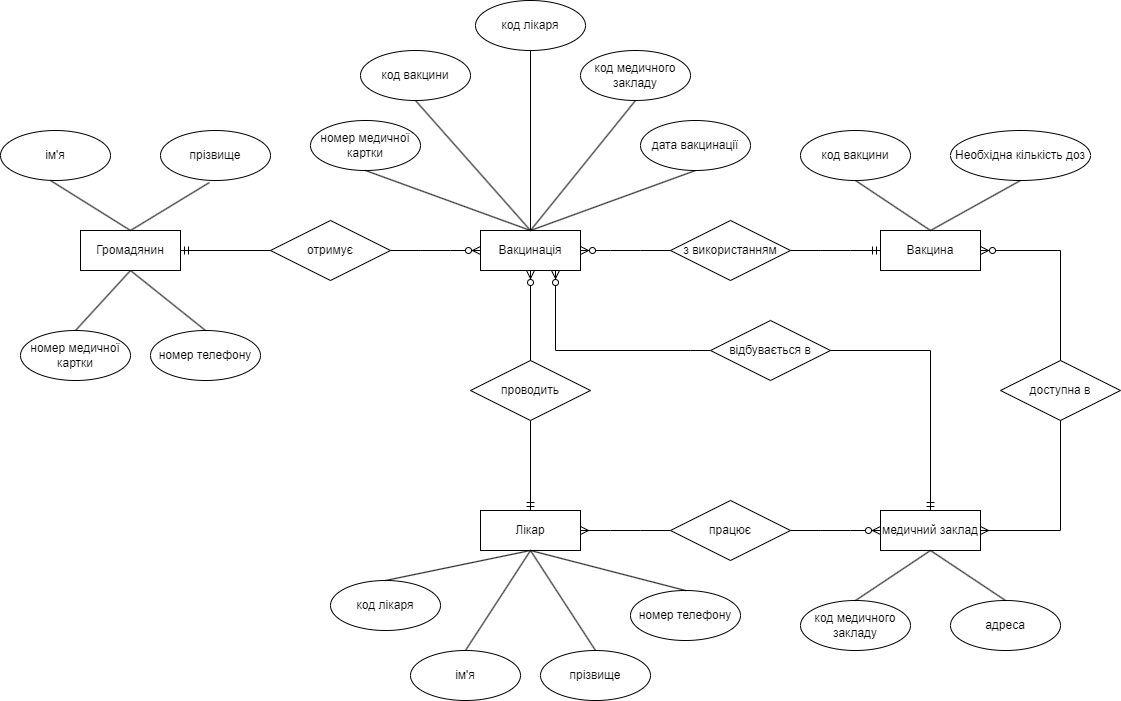


Рисунок 1 – ER-діаграма, побудована за нотацією Чена

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

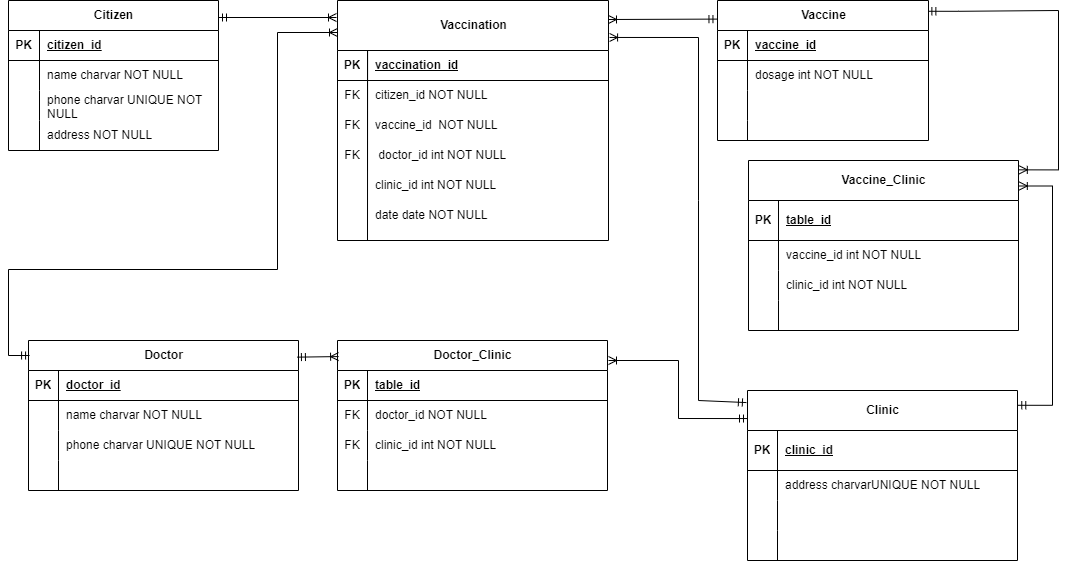


Рисунок 2 – Логічна модель

**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**

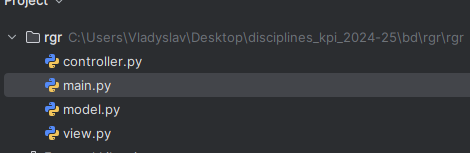
****

Рисунок 3 – Структура програми

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

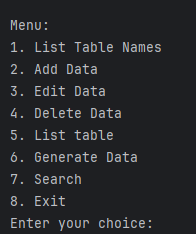
У файлі model.py описаний клас моделі, який відповідає за управління підключенням до бази даних і виконанням низькорівневих запитів до неї.

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.

**Схему меню користувача з описом функціональності кожного пункту**

****

**Основне меню:**

**List Table Names**  
**Опис:** Виводить список усіх таблиць, доступних у базі даних. Цей пункт дозволяє переглядати назви таблиць, що містяться у базі.

**Add Data**  
**Опис:** Дозволяє користувачу додавати нові записи в таблицю. Користувач вводить назву таблиці, стовпці та значення, і програма виконує вставку.

**Edit Data**  
**Опис:** Дозволяє редагувати наявні записи в таблиці. Користувач вказує таблицю, ідентифікатор рядка, стовпці, які потрібно змінити, і нові значення.

**Delete Data**  
**Опис:** Дає можливість видалити записи з таблиці. Користувач вводить назву таблиці та ідентифікатор рядка, який потрібно видалити.

**List Table**  
**Опис:** Виводить кілька перших рядків з обраної таблиці. Користувач вводить назву таблиці та кількість рядків, які потрібно показати.

**Generate Data**  
**Опис:** Генерує випадкові дані для вибраної таблиці. Користувач вводить кількість рядків для генерації, і програма автоматично додає ці рядки у таблицю.

**Search**  
**Опис:** Дозволяє виконувати пошук у базі даних за різними критеріями. Пошук можна здійснювати за датами, іменами, діапазонами чисел і т.д.

**Exit**  
**Опис:** Завершує роботу програми.

**Фрагмент коду (файл сontroller.py), в якому наведено головний цикл роботи програми**

def run(self):  
 while True:  
 choice = self.show\_menu()  
 if choice == '1':  
 self.list\_tables\_names()  
 elif choice == '2':  
 self.add\_data()  
 elif choice == '3':  
 self.edit\_data()  
 elif choice == '4':  
 self.delete\_data()  
 elif choice == '5':  
 self.list\_table()  
 elif choice == '6':  
 self.generate\_data()  
 elif choice == '7':  
 self.search\_menu()  
 elif choice == '8':  
 break

**Фрагмент коду (файл model.h), в якому наведено функції перегляду, внесення, редагування, вилучення та генерації у базі даних:**

Функція перегляду даних:

def get\_listed\_table(self, table\_name, n\_rows):  
 try:  
 query = f"SELECT \* FROM {table\_name} LIMIT %s"  
 cursor = self.conn.cursor()  
 cursor.execute(query, (n\_rows,))  
  
 column\_names = [desc[0] for desc in cursor.description]  
 rows = cursor.fetchall()  
  
 result = [column\_names] + rows  
 return result, 0  
 except psycopg2.Error as e:  
 return f"Error: {e.pgerror}", 1  
 finally:  
 cursor.close()

Функція get\_listed\_table отримує список перших n\_rows записів з таблиці. Вона виконує SQL-запит SELECT \* і обмежує кількість рядків за допомогою LIMIT. Функція повертає назви стовпців і вибрані рядки. У разі помилки повертається повідомлення про помилку.

**Параметри:**

* **table\_name** (str): Назва таблиці.
* **n\_rows** (int): Кількість рядків, що потрібно вибрати з таблиці.

Функція внесення даних:

def add\_data(self, table, columns, values):  
 try:  
 cursor = self.conn.cursor()  
 columns\_str = ", ".join(columns)  
 placeholders = ", ".join(["%s"] \* len(values))  
 query = f"INSERT INTO {table} ({columns\_str}) VALUES ({placeholders})"  
 cursor.execute(query, values)  
 self.conn.commit()  
 return "Data added successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()

Функція add\_data додає новий запис до таблиці в базі даних. Вона приймає назву таблиці, список стовпців і відповідні значення для вставки. Функція генерує SQL-запит для вставки даних і виконує його. Після успішної вставки даних змінення зберігаються в базі даних. У разі помилки транзакція відкочується, і повертається повідомлення про помилку.

**Параметри:**

* **table** (str): Назва таблиці.
* **columns** (list of str): Список стовпців.
* **values** (list): Список значень для вставки.

Функція оновлення даних:

def edit\_data(self, table, id, columns, new\_values):  
 try:  
 cursor = self.conn.cursor()  
 set\_clause = ", ".join([f"{col} = %s" for col in columns])  
 primary\_keys = self.get\_primary\_key\_columns(table)  
 query = f"UPDATE {table} SET {set\_clause} WHERE {primary\_keys[0]} = %s"  
 values = new\_values + [id]  
 cursor.execute(query, values)  
 self.conn.commit()  
 return "Data updated successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()

Функція edit\_data оновлює існуючі дані в таблиці бази даних на основі заданих стовпців та значень. Вона приймає назву таблиці, ідентифікатор запису, список стовпців та нові значення, після чого формує SQL-запит для оновлення та виконує його. Якщо запит успішно виконано, зміни зберігаються в базі даних. У разі помилки транзакція відкочується, і повертається повідомлення про помилку.

**Параметри:**

* **table** (str): Назва таблиці.
* **id** (int): Ідентифікатор запису, що потребує редагування.
* **columns** (list of str): Список стовпців для оновлення.
* **new\_values** (list): Список нових значень для стовпців.

Функція видалення значень:

def delete\_data(self, table, id):  
 try:  
 cursor = self.conn.cursor()  
 primary\_keys = self.get\_primary\_key\_columns(table)  
 query = f"DELETE FROM {table} WHERE {primary\_keys[0]} = %s"  
 cursor.execute(query, (id,))  
 self.conn.commit()  
 return "Data deleted successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()

Функція delete\_data видаляє запис із зазначеної таблиці на основі його ідентифікатора. Вона формує SQL-запит для видалення даних та виконує його. Якщо операція видалення успішна, зміни фіксуються в базі даних. У разі виникнення помилки транзакція відкочується, і повертається повідомлення про помилку.

**Параметри:**

* **table** (str): Назва таблиці.
* **id** (int): Ідентифікатор запису, який потрібно видалити.

Функції генерування даних:

def generate\_data(self, table\_name, n\_rows):  
 if table\_name not in self.get\_tables():  
 return f"Error: Table '{table\_name}' does not exist! {self.get\_tables()}"  
 try:  
 primary\_key = self.get\_primary\_key\_columns(table\_name)[0]  
 with self.conn.cursor() as cursor:  
 cursor.execute(f"SELECT COALESCE(MAX({primary\_key}), 0) FROM {table\_name}")  
 last\_key = cursor.fetchone()[0]  
  
 table\_generators = {  
 "vaccine": self.generate\_vaccine\_data,  
 "doctor": self.generate\_doctor\_data,  
 "clinic": self.generate\_clinic\_data,  
 "vaccination": self.generate\_vaccination\_data,

Функція generate\_data автоматично генерує дані для таблиці, вибираючи відповідний генератор для кожної таблиці. Вона також перевіряє наявність таблиці в базі даних і генерує нові записи, починаючи з максимального значення поточного первинного ключа.

**Параметри:**

* **table\_name** (str): Назва таблиці для генерації даних.
* **n\_rows** (int): Кількість рядків для вставки.

def generate\_vaccine\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccine ({primary\_key}, dosage)  
 SELECT i, floor(random() \* 10 + 1)::integer  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccine records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccine data: {e.pgerror}"

Функція generate\_vaccine\_data генерує дані для таблиці vaccine, включаючи випадкові значення доз вакцин.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_doctor\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO doctor ({primary\_key}, name, phone)  
 SELECT i, 'Doctor\_' || i,  
 '+380' || LPAD((floor(random() \* 1000000000)::bigint)::text, 9, '0')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} doctor records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating doctor data: {e.pgerror}"

Функція generate\_doctor\_data генерує дані для таблиці doctor, створюючи випадкові імена лікарів і їхні телефони.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO clinic ({primary\_key}, address)  
 SELECT i, 'Clinic Address ' || i  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating clinic data: {e.pgerror}"

Функція generate\_clinic\_data генерує дані для таблиці clinic, створюючи випадкові адреси клінік.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_citizen\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO citizen ({primary\_key}, name, address, phone)  
 SELECT i, 'Citizen\_' || i, 'City, Street ' || i,  
 '+380' || LPAD((floor(random() \* 1000000000)::bigint)::text, 9, '0')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} citizen records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating citizen data: {e.pgerror}"

Функція generate\_citizen\_data генерує дані для таблиці citizen, створюючи випадкові імена громадян, адреси та телефони.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_doctor\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO doctor\_clinic ({primary\_key}, doctor\_id, clinic\_id)  
 SELECT i,  
 (SELECT doctor\_id FROM doctor ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1)  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} doctor\_clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating doctor\_clinic data: {e.pgerror}"

Функція generate\_doctor\_clinic\_data генерує зв'язки між лікарями та клініками для таблиці doctor\_clinic.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_vaccination\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccination ({primary\_key}, citizen\_id, doctor\_id, vaccine\_id, clinic\_id, date)  
 SELECT i,  
 (SELECT citizen\_id FROM citizen ORDER BY random() LIMIT 1),  
 (SELECT doctor\_id FROM doctor ORDER BY random() LIMIT 1),  
 (SELECT vaccine\_id FROM vaccine ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1),  
 NOW() - (floor(random() \* 365)::int \* INTERVAL '1 day')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccination records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccination data: {e.pgerror}"

Функція generate\_vaccination\_data генерує дані для таблиці vaccination, зв'язуючи громадян, лікарів, вакцини та клініки.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

def generate\_vaccine\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccine\_clinic ({primary\_key}, vaccine\_id, clinic\_id)  
 SELECT i,  
 (SELECT vaccine\_id FROM vaccine ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1)  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccine\_clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccine\_clinic data: {e.pgerror}"

Функція generate\_vaccine\_clinic\_data генерує зв'язки між вакцинами та клініками для таблиці vaccine\_clinic.

**Параметри:**

* **last\_key** (int): Останнє значення первинного ключа в таблиці.
* **n\_rows** (int): Кількість рядків для генерації.
* **primary\_key** (str): Назва первинного ключа для таблиці.

**Повний код програми**

**Файл main.py:**

from controller import Controller  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 controller = Controller()  
 controller.run()

**Файл model.py:**

import psycopg2  
from time import time  
  
class Model:  
 def \_\_init\_\_(self):  
 self.conn = psycopg2.connect(  
 dbname = "vaccination\_control",  
 user = "postgres",  
 password = "2534",  
 host = "localhost",  
 port = 5432  
 )  
  
 def get\_tables(self):  
 try:  
 cursor = self.conn.cursor()  
 cursor.execute("""SELECT table\_name FROM information\_schema.tables  
 WHERE table\_schema = 'public'""")  
 return [table[0] for table in cursor.fetchall()]  
 except psycopg2.Error as e:  
 return f"Error fetching tables: {e.pgerror}"  
  
 def edit\_data(self, table, id, columns, new\_values):  
 try:  
 cursor = self.conn.cursor()  
 set\_clause = ", ".join([f"{col} = %s" for col in columns])  
 primary\_keys = self.get\_primary\_key\_columns(table)  
 query = f"UPDATE {table} SET {set\_clause} WHERE {primary\_keys[0]} = %s"  
 values = new\_values + [id]  
 cursor.execute(query, values)  
 self.conn.commit()  
 return "Data updated successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()  
  
 def add\_data(self, table, columns, values):  
 try:  
 cursor = self.conn.cursor()  
 columns\_str = ", ".join(columns)  
 placeholders = ", ".join(["%s"] \* len(values))  
 query = f"INSERT INTO {table} ({columns\_str}) VALUES ({placeholders})"  
 cursor.execute(query, values)  
 self.conn.commit()  
 return "Data added successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()  
  
  
 def delete\_data(self, table, id):  
 try:  
 cursor = self.conn.cursor()  
 primary\_keys = self.get\_primary\_key\_columns(table)  
 query = f"DELETE FROM {table} WHERE {primary\_keys[0]} = %s"  
 cursor.execute(query, (id,))  
 self.conn.commit()  
 return "Data deleted successfully!"  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return e.pgerror  
 finally:  
 cursor.close()  
  
 def get\_columns(self, table\_name):  
 query = """  
 SELECT column\_name  
 FROM information\_schema.columns  
 WHERE table\_schema = 'public' AND table\_name = %s;  
 """  
 try:  
 with self.conn.cursor() as cursor:  
 cursor.execute(query, (table\_name,))  
 columns = [row[0] for row in cursor.fetchall()]  
 return columns  
 except Exception as e:  
 self.conn.rollback()  
 print(f"Error retrieving columns for table {table\_name}: {e}")  
 return []  
 finally:  
 cursor.close()  
  
 def get\_primary\_key\_columns(self, table\_name):  
 try:  
 query = """  
 SELECT a.attname  
 FROM pg\_index i  
 JOIN pg\_attribute a ON a.attrelid = i.indrelid AND a.attnum = ANY(i.indkey)  
 WHERE i.indrelid = %s::regclass AND i.indisprimary;  
 """  
 cursor = self.conn.cursor()  
 cursor.execute(query, (table\_name,))  
 result = cursor.fetchall()  
 return [row[0] for row in result]  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error: {e.pgerror}"  
 finally:  
 cursor.close()  
  
  
 def get\_listed\_table(self, table\_name, n\_rows):  
 try:  
 query = f"SELECT \* FROM {table\_name} LIMIT %s"  
 cursor = self.conn.cursor()  
 cursor.execute(query, (n\_rows,))  
  
 column\_names = [desc[0] for desc in cursor.description]  
 rows = cursor.fetchall()  
  
 result = [column\_names] + rows  
 return result, 0  
 except psycopg2.Error as e:  
 return f"Error: {e.pgerror}", 1  
 finally:  
 cursor.close()  
  
 def generate\_data(self, table\_name, n\_rows):  
 if table\_name not in self.get\_tables():  
 return f"Error: Table '{table\_name}' does not exist! {self.get\_tables()}"  
 try:  
 primary\_key = self.get\_primary\_key\_columns(table\_name)[0]  
 with self.conn.cursor() as cursor:  
 cursor.execute(f"SELECT COALESCE(MAX({primary\_key}), 0) FROM {table\_name}")  
 last\_key = cursor.fetchone()[0]  
  
 table\_generators = {  
 "vaccine": self.generate\_vaccine\_data,  
 "doctor": self.generate\_doctor\_data,  
 "clinic": self.generate\_clinic\_data,  
 "vaccination": self.generate\_vaccination\_data,  
 "doctor\_clinic": self.generate\_doctor\_clinic\_data,  
 "citizen": self.generate\_citizen\_data  
 }  
  
 if table\_name not in table\_generators:  
 return f"Error: No data generator defined for table '{table\_name}'"  
  
 return table\_generators[table\_name](int(last\_key), int(n\_rows), primary\_key)  
  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Database error: {e.pgerror}"  
  
 except Exception as e:  
 return f"Unexpected error: {str(e)}"  
  
 def generate\_vaccine\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccine ({primary\_key}, dosage)  
 SELECT i, floor(random() \* 10 + 1)::integer  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccine records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccine data: {e.pgerror}"  
  
 def generate\_doctor\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO doctor ({primary\_key}, name, phone)  
 SELECT i, 'Doctor\_' || i,  
 '+380' || LPAD((floor(random() \* 1000000000)::bigint)::text, 9, '0')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} doctor records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating doctor data: {e.pgerror}"  
  
 def generate\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO clinic ({primary\_key}, address)  
 SELECT i, 'Clinic Address ' || i  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating clinic data: {e.pgerror}"  
  
 def generate\_citizen\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO citizen ({primary\_key}, name, address, phone)  
 SELECT i, 'Citizen\_' || i, 'City, Street ' || i,  
 '+380' || LPAD((floor(random() \* 1000000000)::bigint)::text, 9, '0')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} citizen records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating citizen data: {e.pgerror}"  
  
 def generate\_doctor\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO doctor\_clinic ({primary\_key}, doctor\_id, clinic\_id)  
 SELECT i,  
 (SELECT doctor\_id FROM doctor ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1)  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} doctor\_clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating doctor\_clinic data: {e.pgerror}"  
  
 def generate\_vaccination\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccination ({primary\_key}, citizen\_id, doctor\_id, vaccine\_id, clinic\_id, date)  
 SELECT i,  
 (SELECT citizen\_id FROM citizen ORDER BY random() LIMIT 1),  
 (SELECT doctor\_id FROM doctor ORDER BY random() LIMIT 1),  
 (SELECT vaccine\_id FROM vaccine ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1),  
 NOW() - (floor(random() \* 365)::int \* INTERVAL '1 day')  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccination records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccination data: {e.pgerror}"  
  
 def generate\_vaccine\_clinic\_data(self, last\_key, n\_rows, primary\_key):  
 try:  
 with self.conn.cursor() as cursor:  
 query = f"""  
 INSERT INTO vaccine\_clinic ({primary\_key}, vaccine\_id, clinic\_id)  
 SELECT i,  
 (SELECT vaccine\_id FROM vaccine ORDER BY random() LIMIT 1),  
 (SELECT clinic\_id FROM clinic ORDER BY random() LIMIT 1)  
 FROM generate\_series(%s, %s) AS i  
 """  
 cursor.execute(query, (last\_key + 1, last\_key + n\_rows))  
 self.conn.commit()  
 return f"Inserted {n\_rows} vaccine\_clinic records starting from ID {last\_key + 1}."  
 except psycopg2.Error as e:  
 self.conn.rollback()  
 return f"Error generating vaccine\_clinic data: {e.pgerror}"  
  
 def search\_query\_1(self, date\_start, date\_end, citizen\_name\_pattern):  
 *"""  
 Пошук вакцинацій у вказаному діапазоні дат, імені громадянина за шаблоном.  
 """* query = """  
 SELECT vaccination.vaccination\_id, citizen.name, doctor.name, vaccination.date  
 FROM vaccination  
 JOIN citizen ON vaccination.citizen\_id = citizen.citizen\_id  
 JOIN doctor ON vaccination.doctor\_id = doctor.doctor\_id  
 WHERE vaccination.date BETWEEN %s AND %s  
 AND citizen.name LIKE %s  
 GROUP BY vaccination.vaccination\_id, citizen.name, doctor.name, vaccination.date  
 ORDER BY vaccination.date;  
 """  
 try:  
 with self.conn.cursor() as cursor:  
 start\_time = time()  
 cursor.execute(query, (date\_start, date\_end, f"%{citizen\_name\_pattern}%"))  
 column\_names = [desc[0] for desc in cursor.description]  
 rows = cursor.fetchall()  
 results = [column\_names] + rows  
 execution\_time = (time() - start\_time) \* 1000  
 return results, execution\_time  
 except psycopg2.Error as e:  
 return f"Database error: {e.pgerror}", 0  
  
 def search\_query\_2(self, dosage\_range, clinic\_address\_pattern):  
 *"""  
 Пошук клінік з вакциною у зазначеному діапазоні дозувань та адресою за шаблоном.  
 """* query = """  
 SELECT clinic.address, vaccine.dosage, COUNT(vaccine\_clinic.vaccine\_id)  
 FROM clinic  
 JOIN vaccine\_clinic ON clinic.clinic\_id = vaccine\_clinic.clinic\_id  
 JOIN vaccine ON vaccine\_clinic.vaccine\_id = vaccine.vaccine\_id  
 WHERE vaccine.dosage BETWEEN %s AND %s  
 AND clinic.address LIKE %s  
 GROUP BY clinic.address, vaccine.dosage  
 ORDER BY COUNT(vaccine\_clinic.vaccine\_id) DESC;  
 """  
 try:  
 with self.conn.cursor() as cursor:  
 start\_time = time()  
 cursor.execute(query, (dosage\_range[0], dosage\_range[1], f"%{clinic\_address\_pattern}%"))  
 column\_names = [desc[0] for desc in cursor.description]  
 rows = cursor.fetchall()  
 results = [column\_names] + rows  
 execution\_time = (time() - start\_time) \* 1000  
 return results, execution\_time  
 except psycopg2.Error as e:  
 return f"Database error: {e.pgerror}", 0  
  
 def search\_query\_3(self, doctor\_name\_pattern, max\_vaccines):  
 *"""  
 Пошук лікарів, які проводили менше зазначеної кількості вакцинацій, і їх клінік.  
 """* query = """  
 SELECT doctor.name, clinic.address, COUNT(vaccination.vaccination\_id)  
 FROM doctor  
 JOIN doctor\_clinic ON doctor.doctor\_id = doctor\_clinic.doctor\_id  
 JOIN clinic ON doctor\_clinic.clinic\_id = clinic.clinic\_id  
 LEFT JOIN vaccination ON doctor.doctor\_id = vaccination.doctor\_id  
 WHERE doctor.name LIKE %s  
 GROUP BY doctor.name, clinic.address  
 HAVING COUNT(vaccination.vaccination\_id) < %s  
 ORDER BY COUNT(vaccination.vaccination\_id) ASC;  
 """  
 try:  
 with self.conn.cursor() as cursor:  
 start\_time = time()  
 cursor.execute(query, (f"%{doctor\_name\_pattern}%", max\_vaccines))  
 column\_names = [desc[0] for desc in cursor.description]  
 rows = cursor.fetchall()  
 results = [column\_names] + rows  
 execution\_time = (time() - start\_time) \* 1000  
 return results, execution\_time  
 except psycopg2.Error as e:  
 return f"Database error: {e.pgerror}", 0

**view.py**

class View:  
 def show\_message(self, message):  
 print(message)  
  
 def list\_names(self, tables):  
 print("Tables names:")  
 for table in tables:  
 print(table)  
  
 def get\_list\_table\_input(self):  
 table\_name = input("Enter table name: ")  
 n\_rows = int(input("Enter number of rows: "))  
 return table\_name, n\_rows  
  
 def get\_data\_input(self):  
 try:  
 table = input("Enter table name: ")  
 columns = input("Enter column names separated by space: ").split()  
 val = input("Enter values separated by space: ").split()  
 if len(columns) != len(val):  
 raise ValueError("The number of columns should be equal to the number of values")  
 except ValueError as e:  
 print((f"ERROR: {e}"))  
 return table, columns, val  
  
  
 def get\_update\_input(self):  
 while True:  
 try:  
 table = input("Enter Table Name: ")  
 id = int(input("Enter ID of row that needs to be edited: "))  
 columns = input("Enter column names separated by space: ").split()  
 new\_values = input("Input new values separated by space: ").split()  
 return table, id, columns, new\_values  
 except ValueError as e:  
 print(f"ERROR: {e}")  
  
 def get\_delete\_input(self):  
 try:  
 table = input("Enter Table Name: ")  
 id = int(input("Enter ID of row that needs to be deleted: "))  
 return table, id  
 except ValueError as e:  
 print(f"ERROR: {e}")  
  
 def get\_generate\_data\_input(self):  
 try:  
 table = input("Enter table name: ")  
 n\_rows = input("Enter number of generated rows: ")  
 return table, n\_rows  
 except ValueError as e:  
 print(f"ERROR: {e}")  
  
 def get\_search\_query\_1\_input(self):  
 print("Search Query 1: Find vaccinations within a specific date range and filter by citizen name.")  
 range\_start = input("Enter the start date (YYYY-MM-DD): ")  
 range\_end = input("Enter the end date (YYYY-MM-DD): ")  
 citizen\_name = input("Enter part of the citizen's name (for pattern matching): ")  
 return range\_start, range\_end, citizen\_name  
  
 def get\_search\_query\_2\_input(self):  
 print("Search Query 2: Find clinics with vaccines in a specific dosage range and filter by clinic address.")  
 dosage\_start = input("Enter the minimum dosage: ")  
 dosage\_end = input("Enter the maximum dosage: ")  
 clinic\_address = input("Enter part of the clinic's address (for pattern matching): ")  
 return (int(dosage\_start), int(dosage\_end)), clinic\_address  
  
 def get\_search\_query\_3\_input(self):  
 print("Search Query 3: Find doctors with fewer vaccinations than a given limit and filter by doctor name.")  
 doctor\_name = input("Enter part of the doctor's name (for pattern matching): ")  
 max\_vaccinations = input("Enter the maximum number of vaccinations: ")  
 return doctor\_name, int(max\_vaccinations)  
  
 def show\_results(self, results):  
 if not results:  
 print("No results found.")  
 return  
  
 if isinstance(results[0], (tuple, list)):  
 column\_count = len(results[0])  
 column\_headers = [f"Column {i + 1}" for i in range(column\_count)]  
 else:  
 column\_headers = ["Result"]  
 results = [[row] for row in results]  
 header\_row = " | ".join(column\_headers)  
 print(header\_row)  
 print("-" \* len(header\_row))  
 for row in results:  
 print(" | ".join(map(str, row)))

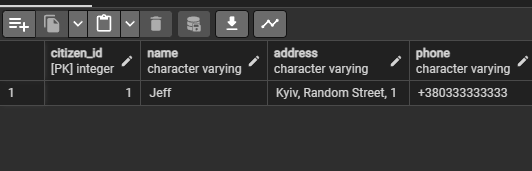
**controller.py**

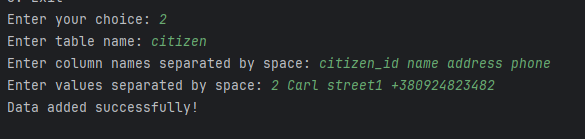
from model import Model  
from view import View  
  
import sys  
  
class Controller:  
 def \_\_init\_\_(self):  
 self.view = View()  
 try:  
 self.model = Model()  
 self.view.show\_message("Connected successfully")  
 except Exception as e:  
 self.view.show\_message(f"Error occurred: {e}")  
 sys.exit(1)  
 def run(self):  
 while True:  
 choice = self.show\_menu()  
 if choice == '1':  
 self.list\_tables\_names()  
 elif choice == '2':  
 self.add\_data()  
 elif choice == '3':  
 self.edit\_data()  
 elif choice == '4':  
 self.delete\_data()  
 elif choice == '5':  
 self.list\_table()  
 elif choice == '6':  
 self.generate\_data()  
 elif choice == '7':  
 self.search\_menu()  
 elif choice == '8':  
 break  
  
 def show\_menu(self):  
 self.view.show\_message("\nMenu:")  
 self.view.show\_message("1. List Table Names")  
 self.view.show\_message("2. Add Data")  
 self.view.show\_message("3. Edit Data")  
 self.view.show\_message("4. Delete Data")  
 self.view.show\_message("5. List table")  
 self.view.show\_message("6. Generate Data")  
 self.view.show\_message("7. Search")  
 self.view.show\_message("8. Exit")  
 return input("Enter your choice: ")  
  
  
  
  
 def list\_tables\_names(self):  
 tables = self.model.get\_tables()  
 self.view.list\_names(tables)  
  
 def add\_data(self):  
 table, columns, val = self.view.get\_data\_input()  
 self.view.show\_message(self.model.add\_data(table, columns, val))  
  
 def edit\_data(self):  
 table, id, columns, new\_value = self.view.get\_update\_input()  
 self.view.show\_message(self.model.edit\_data(table, id, columns, new\_value))  
  
 def delete\_data(self):  
 table, id = self.view.get\_delete\_input()  
 self.view.show\_message(self.model.delete\_data(table, id))  
  
 def list\_table(self):  
 table\_name, n\_rows = self.view.get\_list\_table\_input()  
 listed\_table, error = self.model.get\_listed\_table(table\_name, n\_rows)  
 if error == 1:  
 self.view.show\_message(listed\_table)  
 else:  
 for row in listed\_table:  
 self.view.show\_message(row)  
  
 def generate\_data(self):  
 self.view.show\_message("\nGenerate Data Menu:")  
 table\_name = input("Enter Table Name:")  
 n\_rows = input("Enter number of generated rows:")  
 self.view.show\_message(self.model.generate\_data(table\_name, n\_rows))  
  
 def search\_menu(self):  
 self.view.show\_message("Search Menu:")  
 self.view.show\_message("1: Query 1 - Find vaccinations by date range and citizen name.")  
 self.view.show\_message("2: Query 2 - Find clinics by dosage range and address.")  
 self.view.show\_message("3: Query 3 - Find doctors by name and vaccination count.")  
  
 choice = input("Enter your choice (1-3): ")  
 choice\_query = {  
 '1': (self.view.get\_search\_query\_1\_input, self.model.search\_query\_1),  
 '2': (self.view.get\_search\_query\_2\_input, self.model.search\_query\_2),  
 '3': (self.view.get\_search\_query\_3\_input, self.model.search\_query\_3),  
 }  
 if choice not in choice\_query:  
 self.view.show\_message("Invalid choice. Please enter a valid option.")  
 return  
 input\_func, query\_func = choice\_query[choice]  
 inputs = input\_func()  
 results, execution\_time = query\_func(\*inputs)  
  
 self.view.show\_results(results)  
 self.view.show\_message(f"Query executed in {execution\_time:.2f} ms.")

**Результати виконання програми**

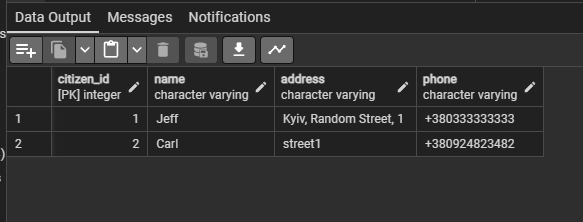
Вставка даних:

До вставки даних:

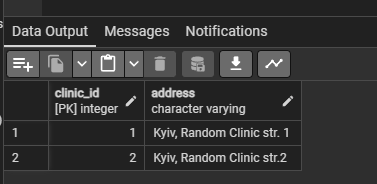


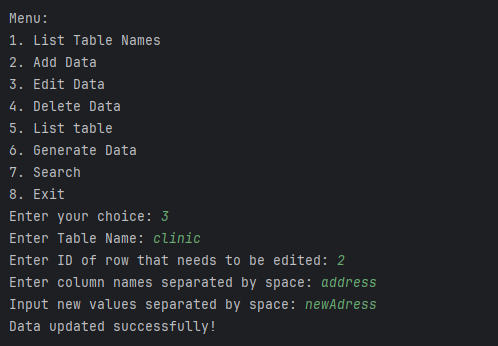


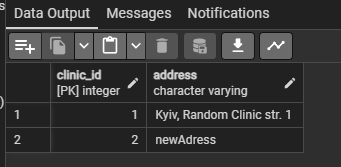
Після вставки даних



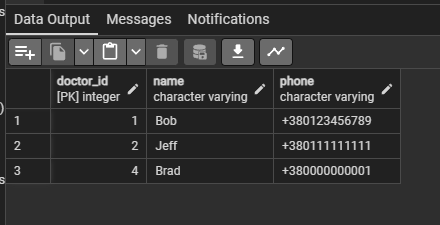
Редагування даних:

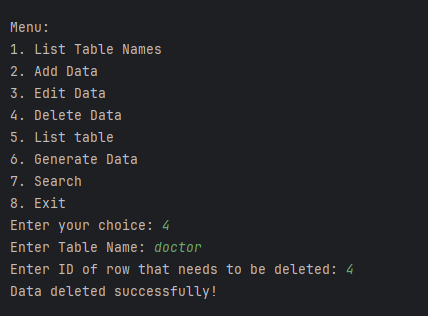


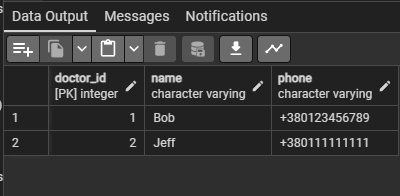




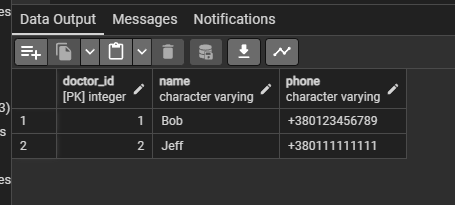
Видалення даних:

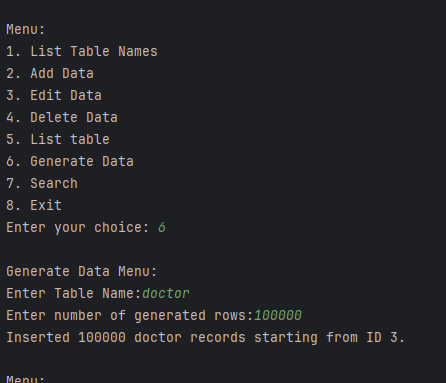


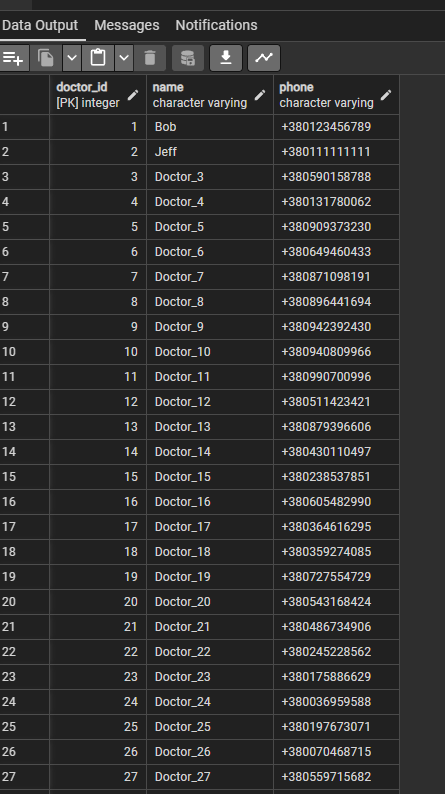


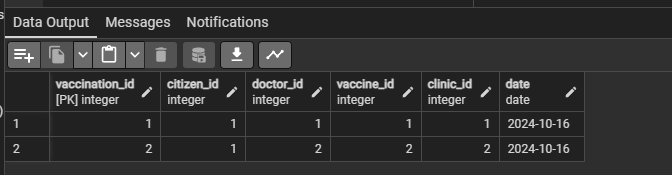


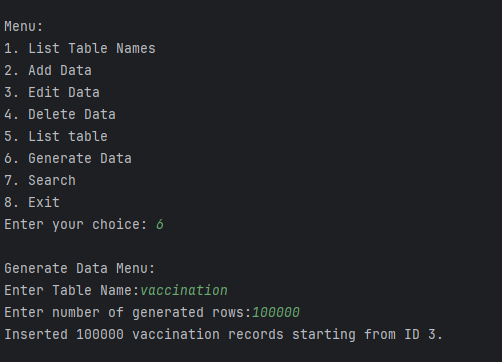
Генерування даних:

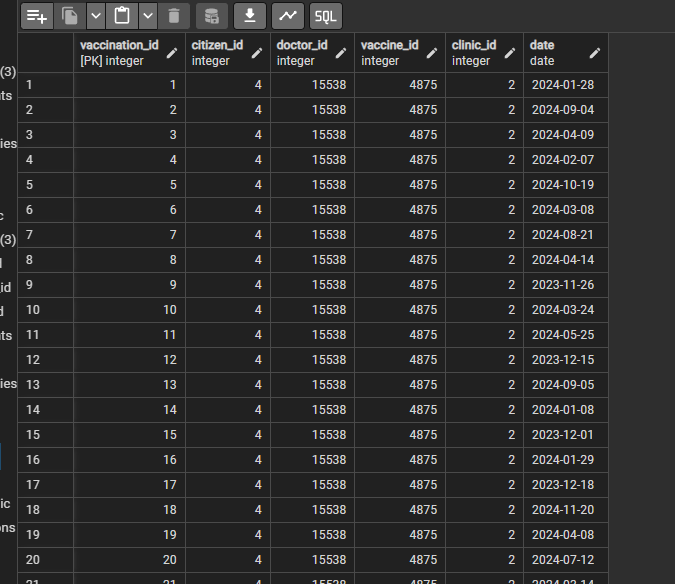




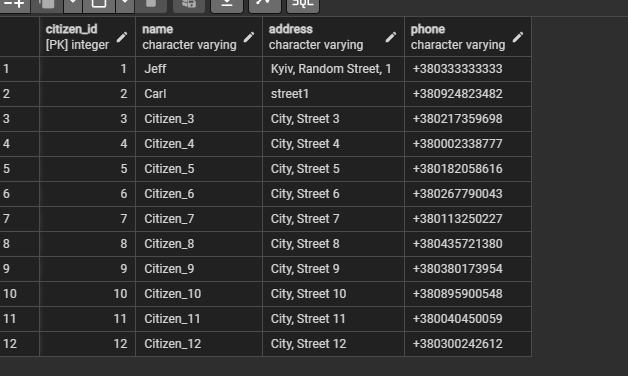






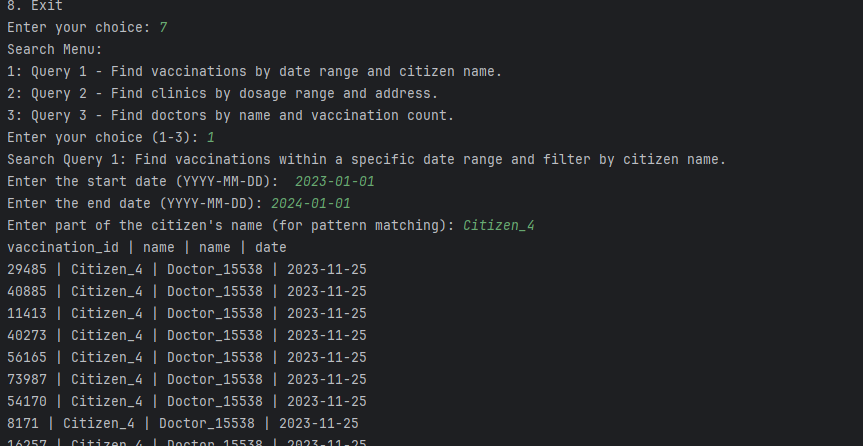


Перегляд даних:

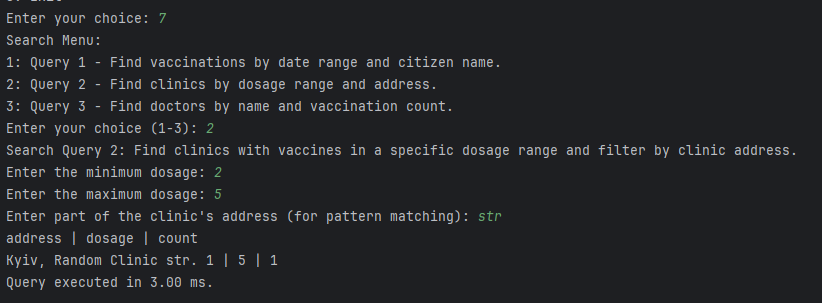




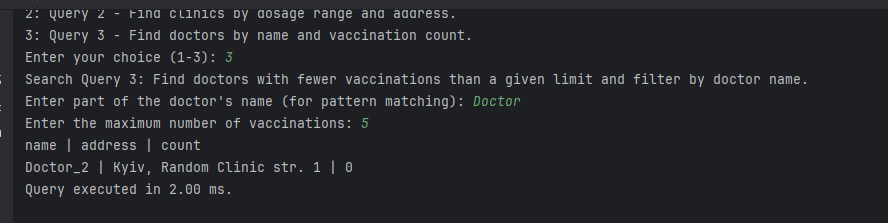
Пошуковий запит 1:



Пошуковий запит 2:



Пошуковий запит 3:



Репозиторій Github: <https://github.com/kovkinvladyslav/bd_labs/tree/main/rgr>