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

**Лабораторна робота № 2**

з дисципліни **Бази даних і засоби управління**

*на тему: “Засоби оптимізації роботи СУБД PostgreSQL”*

Виконав:

студент ІІI курсу

групи КВ-22

Ковкін В.В.

Перевірив:

Павловский В. І.

Київ – 2024

**Мета:** здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

**Завдання за варіантом:**

|  |  |  |
| --- | --- | --- |
| *10* | *Hash, BRIN* | *after delete, insert* |

**Виконання роботи**

1. **Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM)**

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

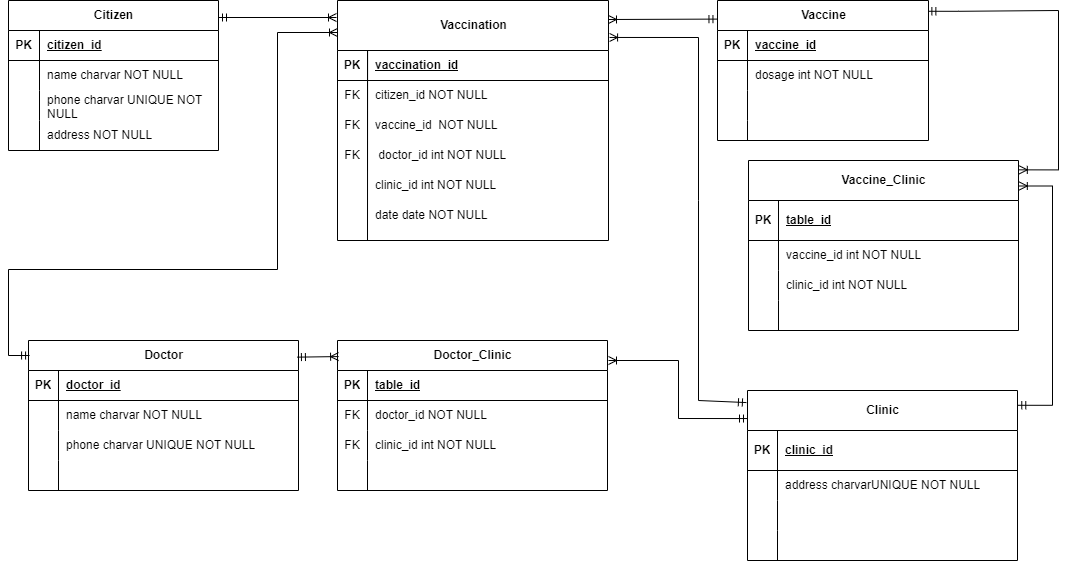


Рисунок 1 – Логічна модель

Відповідні класи ORM утворюють зв’язки що можна зобразити чином показаним на рисунку 2.

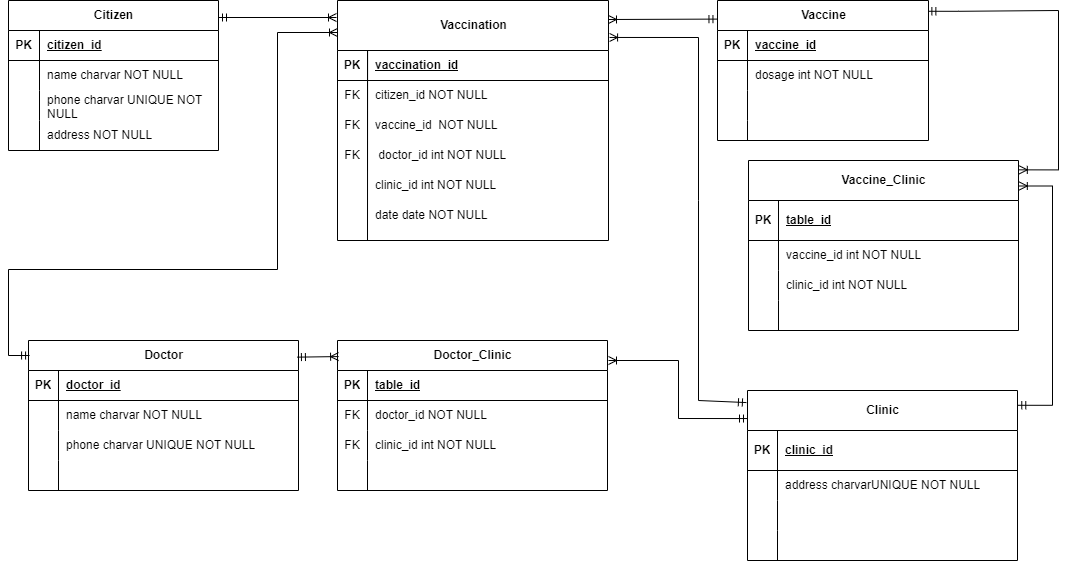


Рисунок 2 – Класи ORM

**Приклади запитів у вигляді ORM:**

Запит на вставку має наступний вигляд:

with Session() as session:  
 new\_citizen = Citizen(  
 name="John Doe",  
 address="123 Elm Street",  
 phone="123-456-7890"  
 )  
 session.add(new\_citizen)  
 session.commit()

Запит на додавання нового запису в базу даних виконується через створення нового екземпляра класу, який відповідає потрібній таблиці. Після цього об'єкт передається в сесію за допомогою методу add(). Потім викликається commit(), щоб підтвердити зміни в базі даних.

Запит на редагування:

with Session() as session:  
 citizen\_id = 1 # ID громадянина, якого потрібно змінити  
 citizen = session.query(Citizen).get(citizen\_id)  
 if citizen:  
 citizen.address = "456 Oak Avenue"  
 session.commit()

Щоб оновити дані в базі, спочатку потрібно отримати об'єкт із таблиці за допомогою методу query() або get(). Після цього редагуються потрібні атрибути об'єкта, і викликається commit(), щоб підтвердити зміни в базі даних.

Запит на вилучення:

with Session() as session:  
 citizen\_id = 1 # ID громадянина, якого потрібно видалити  
 citizen = session.query(Citizen).get(citizen\_id)  
 if citizen:  
 session.delete(citizen)  
 session.commit()

Щоб видалити запис, спочатку потрібно отримати об'єкт з таблиці за допомогою методу query() або get(). Після цього викликається метод delete() для видалення об'єкта, і commit() для підтвердження змін у базі даних.

Код зміненого модуля Model:

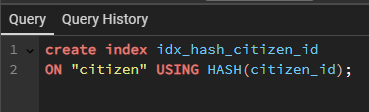
Реалізація вставки, вилучення та редагування екзамплярів класів-сутностей засобами SQLAlchemy. Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля «Модель») залишились без змін.

import psycopg2  
from time import time  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy import (  
 Column, Integer, String, ForeignKey, Date, create\_engine  
)  
  
from sqlalchemy.orm import relationship, sessionmaker  
from sqlalchemy.sql import select  
  
  
Base = declarative\_base()  
engine = create\_engine('postgresql://postgres:2534@localhost:5432/vaccination\_control')  
Session = sessionmaker(bind=engine)  
  
  
class Citizen(Base):  
 \_\_tablename\_\_ = 'citizen'  
 citizen\_id = Column(Integer, primary\_key=True)  
 name = Column(String, nullable=False)  
 address = Column(String, nullable=False)  
 phone = Column(String, unique=True, nullable=False)  
  
 vaccinations = relationship('Vaccination', back\_populates='citizen')  
  
  
class Clinic(Base):  
 \_\_tablename\_\_ = 'clinic'  
 clinic\_id = Column(Integer, primary\_key=True)  
 address = Column(String, nullable=False)  
  
 doctor\_clinics = relationship('DoctorClinic', back\_populates='clinic')  
 vaccine\_clinics = relationship('VaccineClinic', back\_populates='clinic')  
 vaccinations = relationship('Vaccination', back\_populates='clinic')  
  
  
class Doctor(Base):  
 \_\_tablename\_\_ = 'doctor'  
 doctor\_id = Column(Integer, primary\_key=True)  
 name = Column(String, nullable=False)  
 phone = Column(String, nullable=False)  
  
 doctor\_clinics = relationship('DoctorClinic', back\_populates='doctor')  
 vaccinations = relationship('Vaccination', back\_populates='doctor')  
  
class Vaccine(Base):  
 \_\_tablename\_\_ = 'vaccine'  
 vaccine\_id = Column(Integer, primary\_key=True)  
 dosage = Column(Integer, nullable=False)  
  
 # Relationships  
 vaccine\_clinics = relationship('VaccineClinic', back\_populates='vaccine')  
 vaccinations = relationship('Vaccination', back\_populates='vaccine')  
  
class DoctorClinic(Base):  
 \_\_tablename\_\_ = 'doctor\_clinic'  
 table\_id = Column(Integer, primary\_key=True)  
 doctor\_id = Column(Integer, ForeignKey('doctor.doctor\_id'), nullable=False)  
 clinic\_id = Column(Integer, ForeignKey('clinic.clinic\_id'), nullable=False)  
  
 # Relationships  
 doctor = relationship('Doctor', back\_populates='doctor\_clinics')  
 clinic = relationship('Clinic', back\_populates='doctor\_clinics')  
  
  
class VaccineClinic(Base):  
 \_\_tablename\_\_ = 'vaccine\_clinic'  
 table\_id = Column(Integer, primary\_key=True)  
 vaccine\_id = Column(Integer, ForeignKey('vaccine.vaccine\_id'), nullable=False)  
 clinic\_id = Column(Integer, ForeignKey('clinic.clinic\_id'), nullable=False)  
  
 vaccine = relationship('Vaccine', back\_populates='vaccine\_clinics')  
 clinic = relationship('Clinic', back\_populates='vaccine\_clinics')  
  
  
class Vaccination(Base):  
 \_\_tablename\_\_ = 'vaccination'  
 vaccination\_id = Column(Integer, primary\_key=True)  
 citizen\_id = Column(Integer, ForeignKey('citizen.citizen\_id'), nullable=False)  
 doctor\_id = Column(Integer, ForeignKey('doctor.doctor\_id'), nullable=False)  
 vaccine\_id = Column(Integer, ForeignKey('vaccine.vaccine\_id'), nullable=False)  
 clinic\_id = Column(Integer, ForeignKey('clinic.clinic\_id'), nullable=False)  
 date = Column(Date, nullable=False)  
  
 # Relationships  
 citizen = relationship('Citizen', back\_populates='vaccinations')  
 doctor = relationship('Doctor', back\_populates='vaccinations')  
 vaccine = relationship('Vaccine', back\_populates='vaccinations')  
 clinic = relationship('Clinic', back\_populates='vaccinations')  
  
TABLE\_NAME\_MAP = {  
 "citizen": Citizen,  
 "clinic": Clinic,  
 "doctor": Doctor,  
 "vaccine": Vaccine,  
 "doctor\_clinic": DoctorClinic,  
 "vaccine\_clinic": VaccineClinic,  
 "vaccination": Vaccination,  
}  
  
class Model:  
 def \_\_init\_\_(self):  
 self.session = Session  
 self.conn = psycopg2.connect(  
 dbname = "vaccination\_control",  
 user = "postgres",  
 password = "2534",  
 host = "localhost",  
 port = 5432  
 )  
  
 def get\_tables(self):  
 return Base.metadata.tables.keys()  
  
 def edit\_data(self, table\_name, id, columns, new\_values):  
 try:  
 table\_class = TABLE\_NAME\_MAP.get(table\_name)  
  
 if not table\_class:  
 return f"Table '{table\_name}' does not exist."  
  
 with self.session() as s:  
 record = s.query(table\_class).get(id)  
 if not record:  
 return f"Record with ID {id} not found in {table\_name}."  
  
 for column, value in zip(columns, new\_values):  
 if not hasattr(record, column):  
 return f"Column '{column}' does not exist in {table\_name}."  
 setattr(record, column, value)  
  
 s.commit()  
 return "Data updated successfully!"  
  
 except Exception as e:  
 s.rollback()  
 return f"An error occurred: {str(e)}"  
 finally:  
 s.close()  
  
 def add\_data(self, table\_name, columns, values):  
 try:  
 with self.session() as s:  
 table = TABLE\_NAME\_MAP.get(table\_name)  
 if not table:  
 return f"Table '{table\_name}' does not exist."  
 data = dict(zip(columns, values))  
 new\_record = table(\*\*data)  
 s.add(new\_record)  
 s.commit()  
 return f"Data added successfully!"  
 except Exception as e:  
 s.rollback()  
 return f"An error occurred: {str(e)}"  
 finally:  
 s.close()  
  
  
  
 def delete\_data(self, table\_name, record\_id):  
 try:  
 with self.session() as s:  
 table\_class = TABLE\_NAME\_MAP.get(table\_name)  
 if not table\_class:  
 return f"Table '{table\_name}' does not exist."  
 with self.session() as s:  
 record = s.query(table\_class).get(record\_id)  
 if not record:  
 return f"Record with ID {record\_id} not found in {table\_name}."  
  
 s.delete(record)  
 s.commit()  
 return "Record deleted successfully!"  
 except Exception as e:  
 return f"An error occurred: {str(e)}"  
 finally:  
 s.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

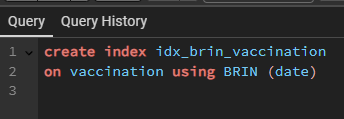
1. **Створити та проаналізувати різні типи індексів у PostgreSQL.**

**Команди створення індексів:**

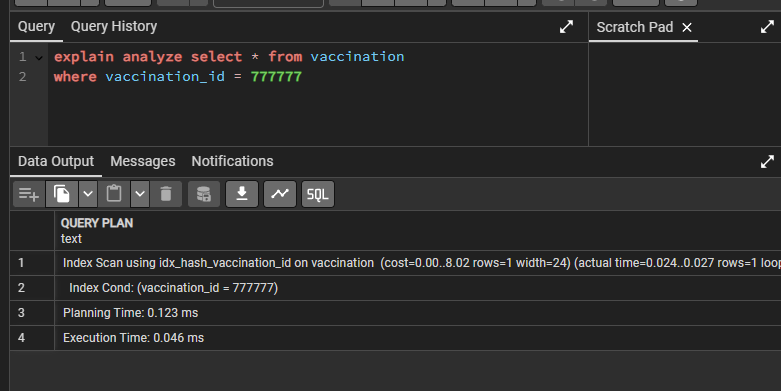
Hash Index:



Brin Index:



**Приклад 1: Запит на просту фільтрацію із виведенням результуючих даних.**

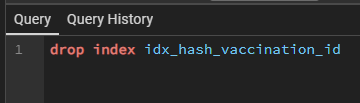


Запит відбувається з використанням створеного , що має суттєво зменьшити ресурси використані на запити пов’язані з знаходженням конкретного запису. Швидкість доступу з використанням хеш-функції є константною О(1)

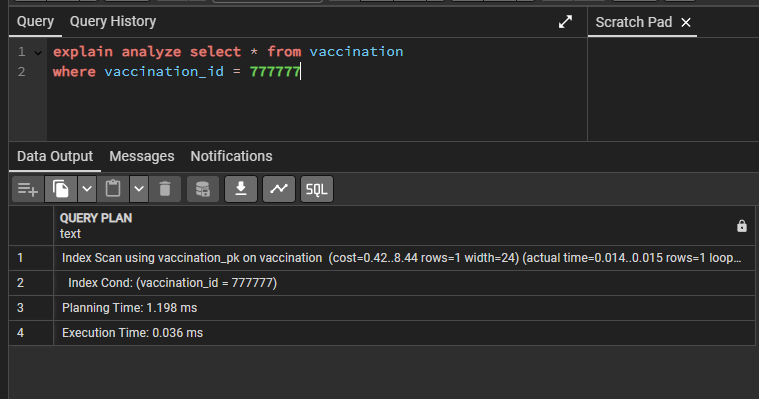
Результати:

* Час планування: 1.23 ms
* Час виконання: 0.046 ms
* Ресурси використані для підготовки: 0.00
* Ресурси використанні для виконання: 8.02

Видалення Хеш-Індексу :



Виконання того самого запиту, але без використання Хеш-індексу:

****

Результати:

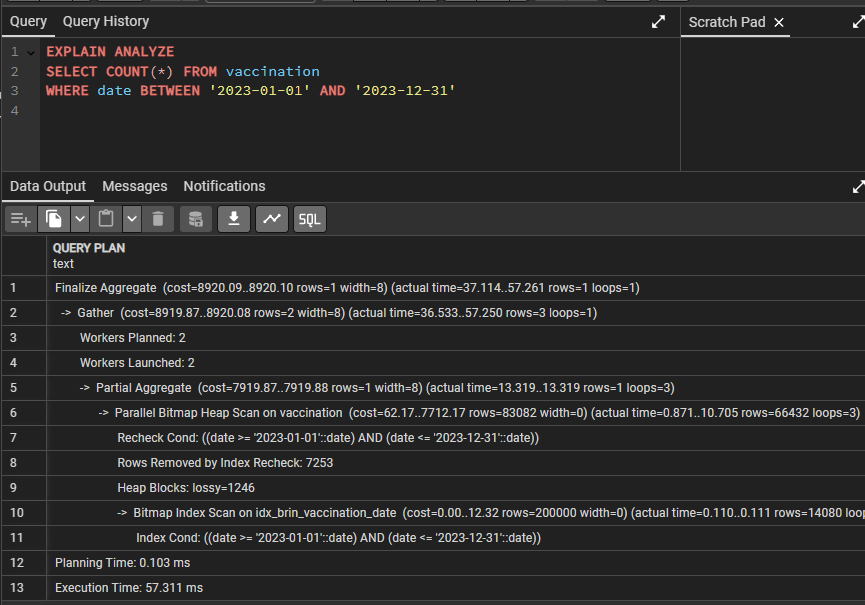
* Час планування: 1.198 ms
* Час виконання: 0.036 ms
* Ресурси використані для підготовки: 0.42
* Ресурси використанні для виконання: 8.44

Як і очікувалось, використання Хеш-індексу для запитів пошуку значень серед великого обсягу даних дозволяє отримати результат значно швидше за лінійне проходження (більше ніж в 10 разів).

**Приклад 2: Агрегатні функції**

Визначення кількості даних в заданому впорядкованому діапазоні.

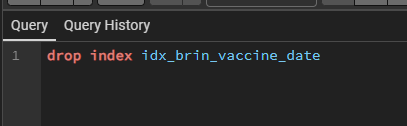
BRIN-index:



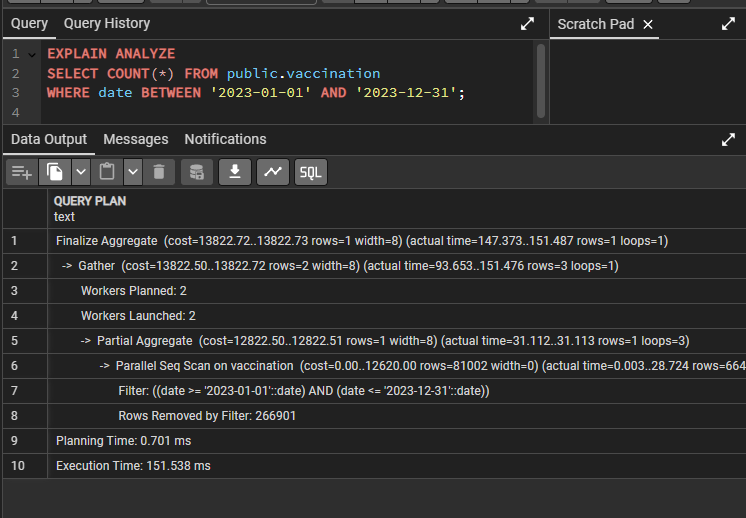
**Результат**:

* Час планування: 0.103 ms
* Час виконання 57.311 ms

Видалення індексу :



Виконання того самого запиту на неіндексованих даних:



**Результат**:

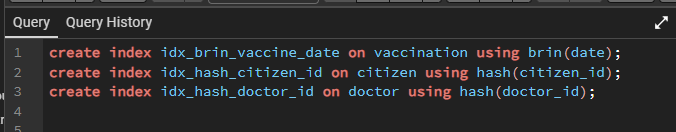
* Час планування: 0.701 ms
* Час виконання: 151.538 ms

**Пояснення**:

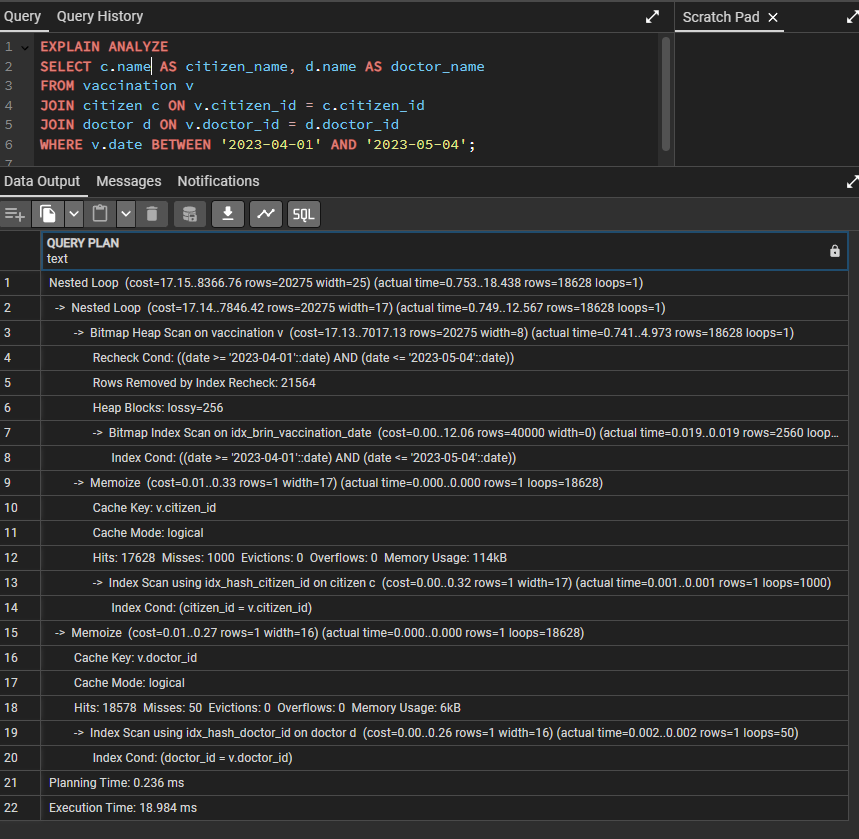
BRIN-індекс ефективний для великих таблиць з впорядкованими даними. Сканує блоки даних за мінімальними та максимальними значеннями в них, що пришвидшує пошук діапазону.

**Приклад 3: Операції з'єднання**

**Індекси таблиць які будуть з’єднуватися:**



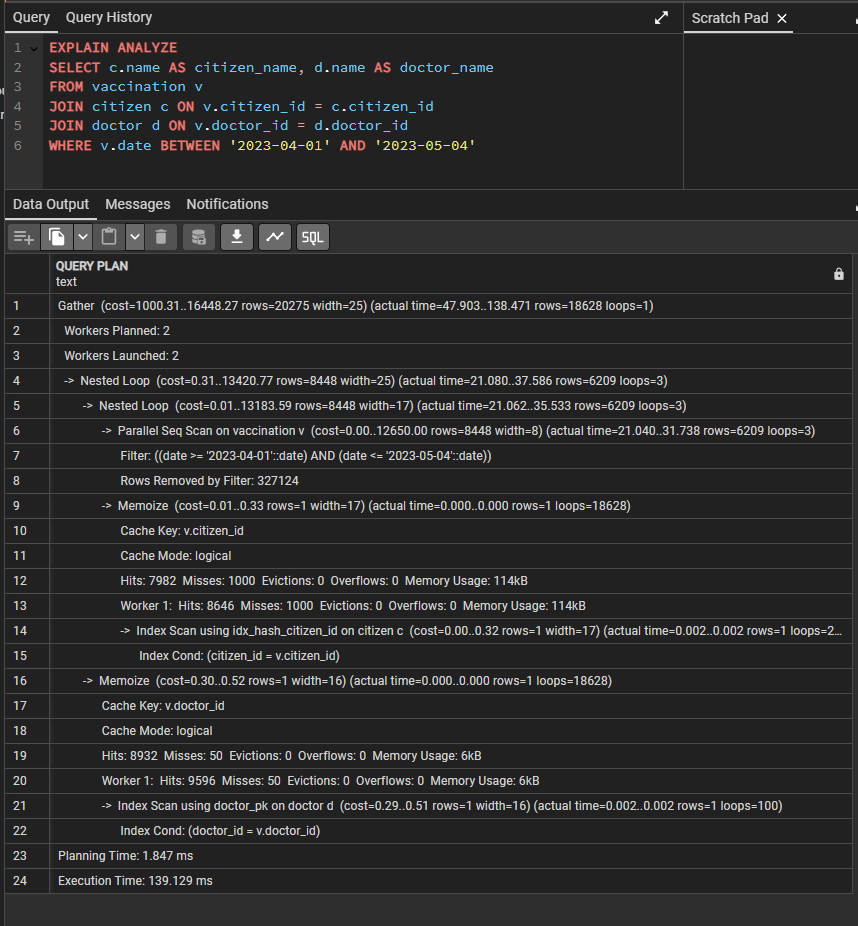
**Запит на з’єднання таблиць :**



Результат:

* Час планування: 0.236
* Час виконання: 18.984 ms

Повторний запит, з видаленими індексами:



Результат:

* Час планування: 1.847 ms
* Час виконання 139.129 ms

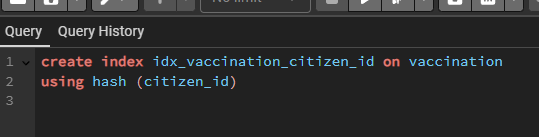
Пояснення:

У цьому запиті BRIN-індекс використовується для швидкого визначення блоків, які містять записи з датами у вказаному діапазоні. Це дозволяє уникнути повного сканування таблиці, навіть якщо дані не впорядковані ідеально. Потім PostgreSQL перевіряє відповідність кожного запису з цих блоків умовам запиту (цей етап називається "Recheck").

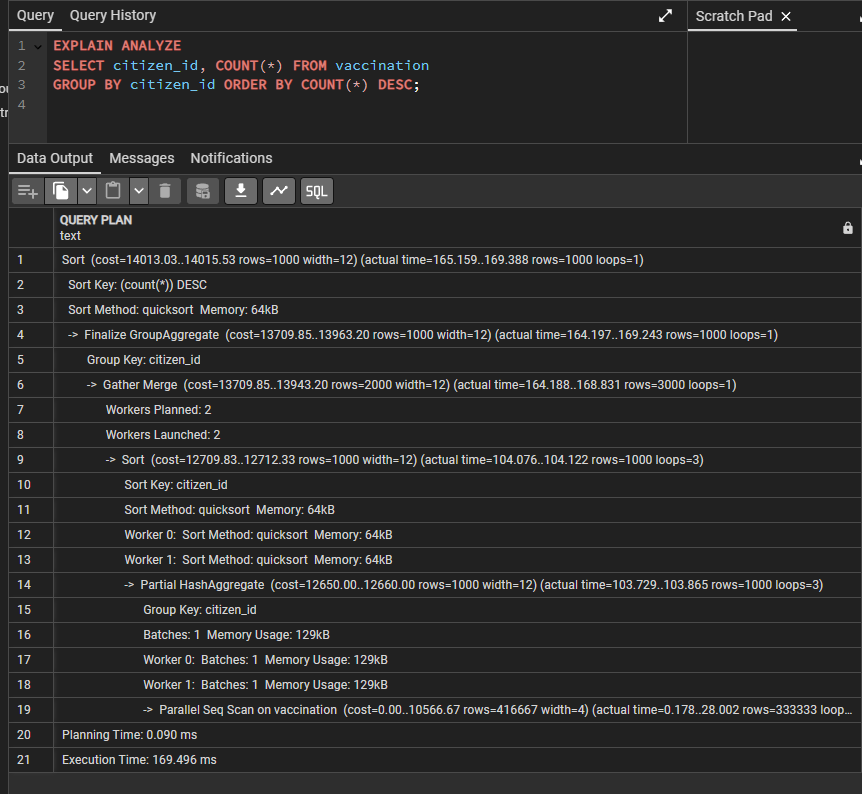
Хеш-Індекси на полях citizen\_id та doctor\_id також значно прискорюють пошук відповідних записів у таблицях citizen та doctor, завдяки чому зменшується кількість оброблюваних рядків на кожному етапі. Це все разом забезпечує високу швидкість виконання запиту.

**Приклад 4: Групування та сортування**

Індекс для таблиці vaccination по стовбцю citizen\_id



Запит який групує громадян і їх кількість вакцинацій в спадаючому порядку:

****

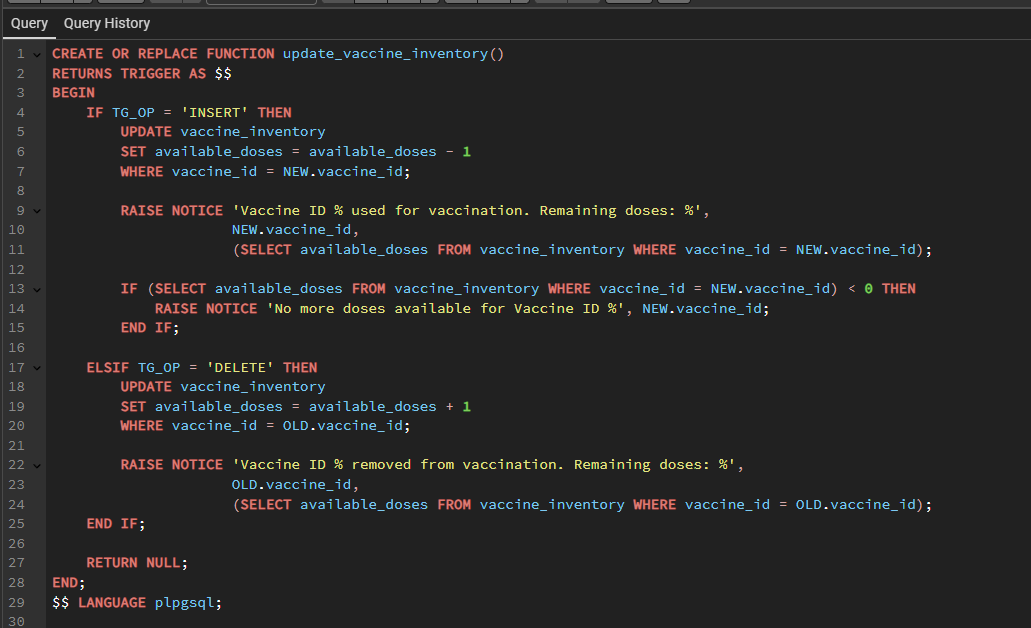
**Пояснення:**

Хеш-індексація неефективна для групування, оскільки вона створює індекс тільки для точного пошуку значень, але не зберігає дані в упорядкованому вигляді. Це унеможливлює ефективний доступ до діапазонів значень, які часто потрібні для групування. Для таких операцій краще підходять B-Tree індекси, оскільки вони забезпечують упорядкування і швидкий доступ до суміжних значень.

BRIN-індекси можуть підходити для групування, але лише за умови що дані в таблиці фізично упорядковані або мають природний кластер по значеннях, що використовуються для групування (наприклад, дати). Якщо групувати по id громадян які брали участь у вакцинації, postgresql обирає відсортувати і використувати Parallel Seq Scan.

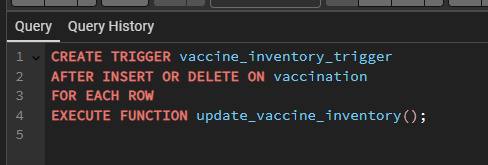
1. **Розробити тригер бази даних PostgreSQL.**

**Команди, що ініціюють виконання тригера:**

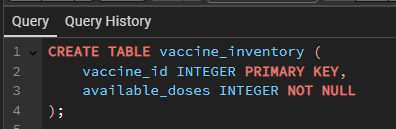
****

Функція тригера зменшує кількість вакцин в інвентарі після виклику вставлення нової вакцинації. І збільшує кількість вакцин в інвентарі після видалення запису про вакцинацію. Також піднімається повідомлення про внесені зміни.

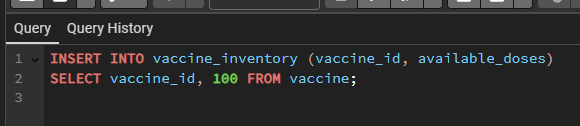
**Створення і прикріплення тригеру до відповідної таблиці:**



Створення спеціальної таблиці для аудиту кількості вакцин що залишились:

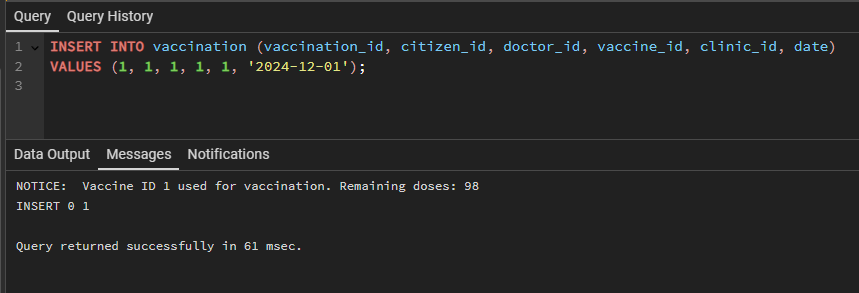


Припустимо, у кожної вакцини є 100 доз на початку



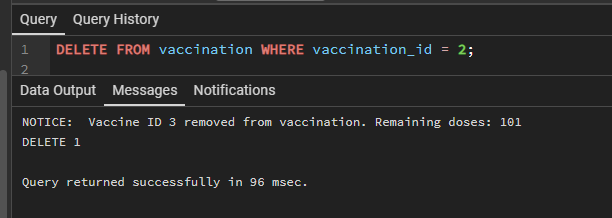
**Тестування тригеру:**

Додаємо нову вакцинацію



Бачимо повідомлення про зменшення кількості доз вакцин, що залишились.

Видалимо запис про вакцинацію



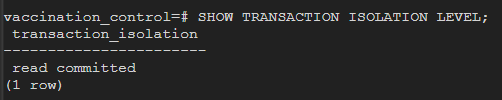
Бачимо повідомлення про збільшення кількості доз вакцин, що залишились.

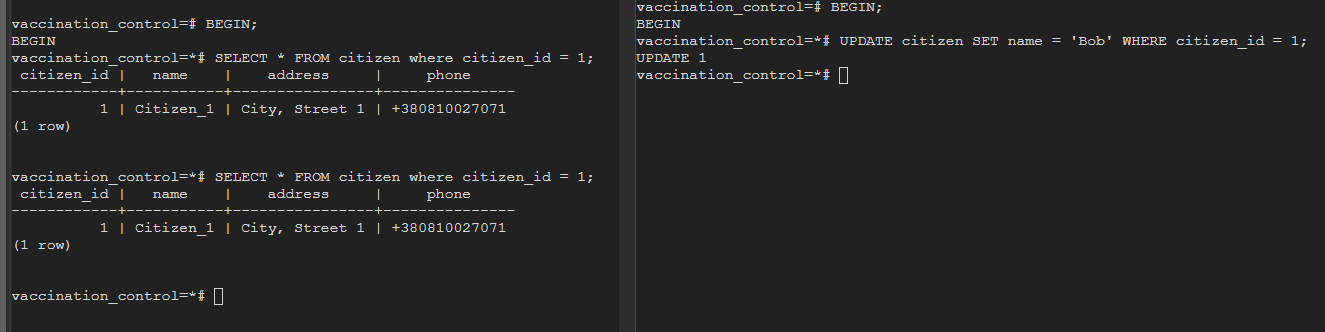
1. **Приклади та аналіз рівнів ізоляції транзакцій у PostgreSQL.**

Існує чотири рівні ізоляції транзакцій, визначені стандартом SQL, але postgres реалізує лише три з них. Перший рівень, Read uncommited, відсутній.

READ COMMITTED

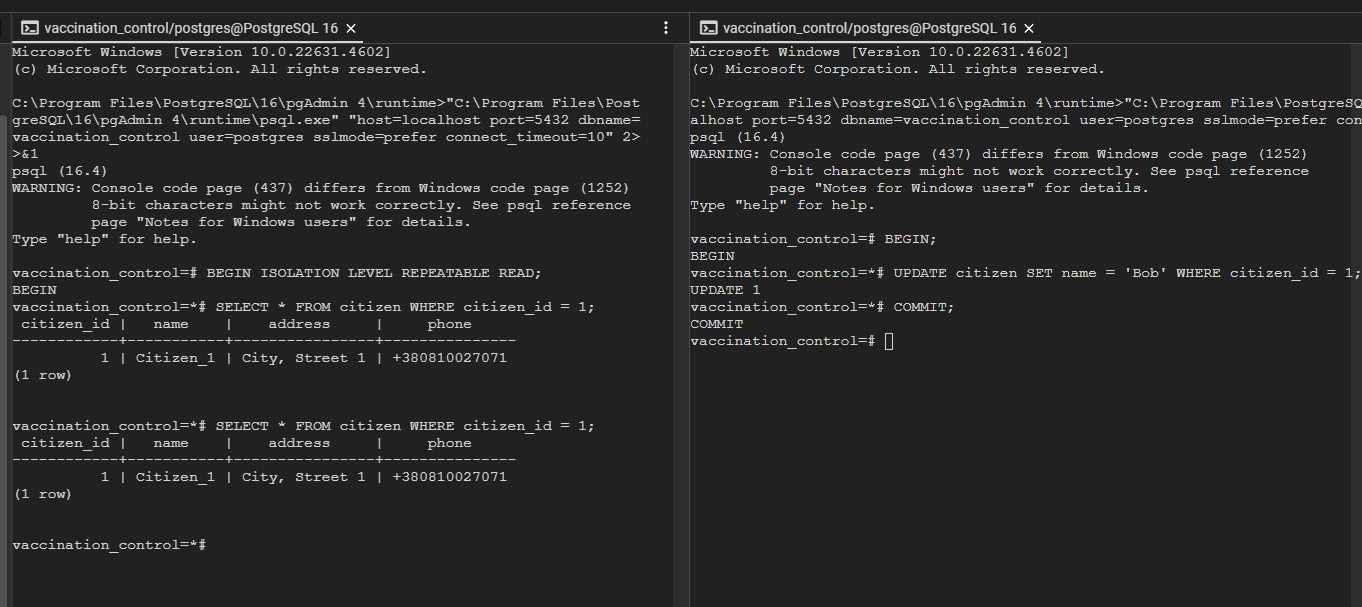
Рівень ізоляції за замовченням. Переконаємося в цьому командою show transaction isolation level:



 **1. Захист від брудного читання**

1. У Т1 (перше вікно) було виконано зчитування рядка з таблиці citizen, де citizen\_id = 1. У результаті бачимо, що атрибут name = 'Citizen\_1'.
2. У Т2 (друге вікно) було створено **непідтверджену** зміну атрибута name на 'Bob' для рядка, де citizen\_id = 1. Зміна ще не зафіксована (COMMIT не виконано).
3. У Т1 було знову зчитано рядок з таблиці citizen, де citizen\_id = 1. Результат: атрибут name залишився 'Citizen\_1'.

Це відбувається тому, що на рівні ізоляції READ COMMITTED транзакція бачить лише підтверджені зміни (дані, зафіксовані через COMMIT). Непідтверджені зміни з іншої транзакції ігноруються.

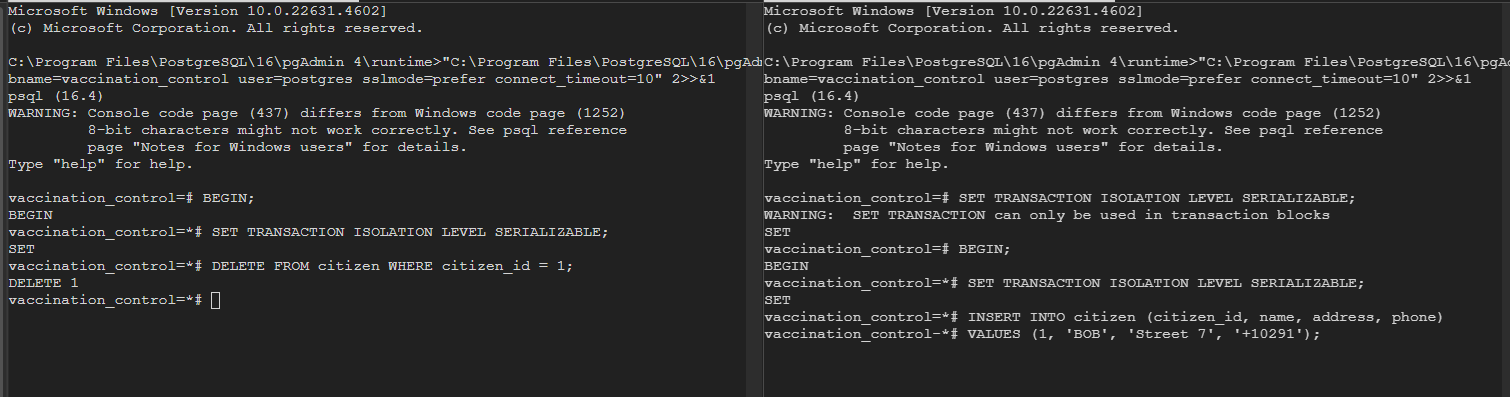
**REPEATABLE READ**

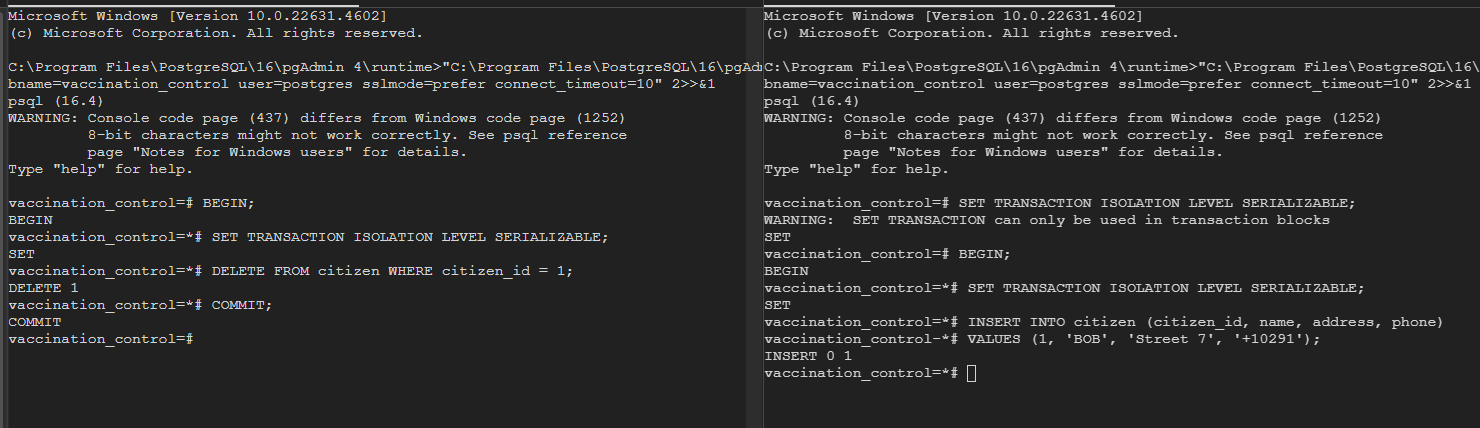
1. У Т1 (перше вікно) було встановлено режим REPEATABLE READ і виконано зчитування рядка де citizen\_id=1.
2. У Т2 (друге вікно) було створено **підтверджену** зміну атрибута name на 'Bob' для рядка, де citizen\_id = 1 (після виконання COMMIT).
3. У Т1 знову було виконано зчитування тих самих даних. Результат: набір рядків залишається таким самим, як і під час першого зчитування (атрибут name = 'Citizen\_1').

На рівні ізоляції REPEATABLE READ транзакція гарантує, що дані, прочитані на початку, залишаються незмінними до завершення цієї транзакції. **Неповторюване читання** виключено.

**SERIALIZABLE**

SERIALIZABLE рівень забезпечує найвищий ступінь ізоляції.

Приклад:

* Додавання рядка у Т2(друге вікно) не відбувається поки не підтверджена команда видалення в Т1(перше вікно)
* Підтвердження у вікні 1 (COMMIT):

Відбувається розблокування завершення формування непідтвердженної транзакції у Т2.

Це відбувається через те, що на рівні ізоляції **SERIALIZABLE** транзакції виконуються так, ніби вони відбуваються одна за одною. Кожна транзакція працює з даними, як якщо б вона була єдиною, що працює з базою, а інші транзакції блокуються до завершення поточної. Всі зміни, зроблені в іншій транзакції, не впливають на результати читання до завершення поточної транзакції. Результат виглядає так, ніби транзакції були виконані в якомусь послідовному порядку.

Посилання на GitHub: