

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 групи КВ-23

Перетятько Богдан

Телеграм: <https://t.me/akk3rm4n>

Перевірив: Петрашенко А.В.

Київ – 2024

<https://github.com/4k3rm4n/DB-LAB2.git>

*Метою роботи* є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

*Завдання* роботи полягає у наступному:

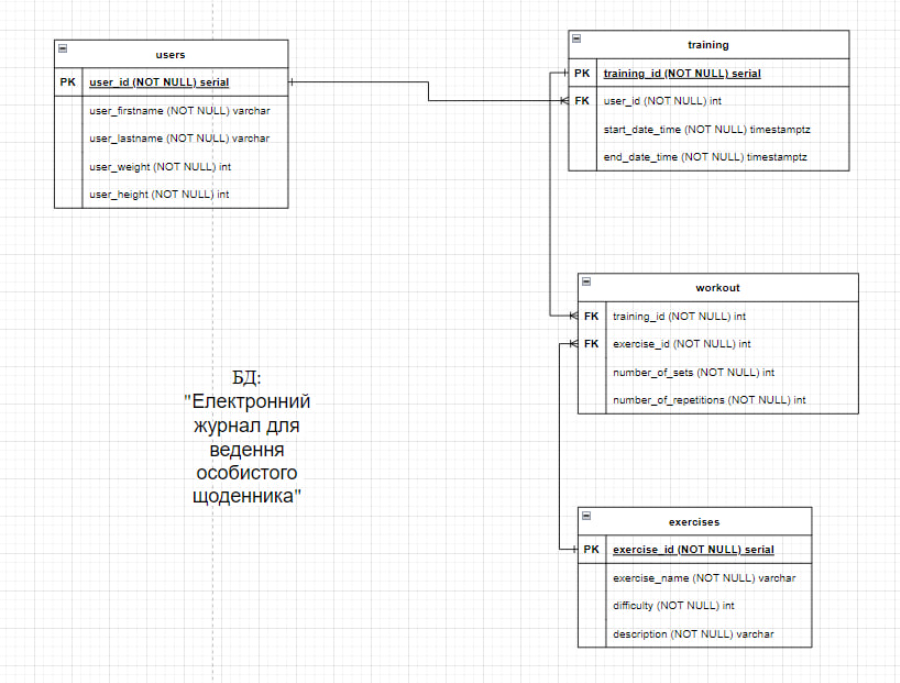
1. Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

**Варіант 21**

****

**Виконання роботи**

**Логічна модель (схема) “Система управління та аналізу даних в галузі робототехніки”**



**Класи ORM**

class User(Base):  
 \_\_tablename\_\_ = 'users'  
 user\_id = Column(Integer, primary\_key=True)  
 user\_firstname = Column(String, nullable=False)  
 user\_lastname = Column(String, nullable=False)  
 user\_weight = Column(Integer, nullable=False)  
 user\_height = Column(Integer, nullable=False)  
  
  
  
class Exercise(Base):  
 \_\_tablename\_\_ = 'exercises'  
 exercise\_id = Column(Integer, primary\_key=True)  
 exercise\_name = Column(String, nullable=False)  
 difficulty = Column(Integer, nullable=False)  
 description = Column(String, nullable=False)  
  
  
class Training(Base):  
 \_\_tablename\_\_ = 'training'  
 training\_id = Column(Integer, primary\_key=True)  
 start\_date\_time = Column(DateTime, nullable=False)  
 end\_date\_time = Column(DateTime, nullable=False)  
 user\_id = Column(Integer, ForeignKey('users.user\_id'))  
  
  
class Workout(Base):  
 \_\_tablename\_\_ = 'workout'  
 workout\_id = Column(Integer, primary\_key=True)  
 training\_id = Column(Integer, ForeignKey('training.training\_id'), nullable=False)  
 exercise\_id = Column(Integer, ForeignKey('exercises.exercise\_id'), nullable=False)  
 number\_of\_sets = Column(Integer, nullable=False)  
 number\_of\_repetitions = Column(Integer, nullable=False)

**Приклади запитів у вигляді ORM**

Фрагмент програми для введення даних в таблицю:

def add\_data(table\_name, data\_dict):  
 model = get\_model(table\_name)  
 new\_record = model(\*\*data\_dict)  
 session.add(new\_record)  
 session.commit()  
 return new\_record

Фрагмент програми для видалення даних з таблиці:

def delete\_data(table\_name, record\_id):  
 model = get\_model(table\_name)  
  
 try:  
 record = session.query(model).get(record\_id)  
  
 if record is None:  
 raise ValueError(f"Запис із ID {record\_id} не знайдено у таблиці {table\_name}.")  
  
 session.delete(record)  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise e

Фрагмент програми для оновлення даних в таблиці:

def update\_data(table\_name, record\_id, updates):  
 model = get\_model(table\_name)  
  
 try:  
 record = session.query(model).get(record\_id)  
  
 if record is None:  
 raise ValueError(f"Запис із ID {record\_id} не знайдено у таблиці {table\_name}.")  
  
 for column, value in updates.items():  
 if not hasattr(record, column):  
 raise ValueError(f"Колонка '{column}' не існує в таблиці {table\_name}.")  
 setattr(record, column, value)  
  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise e

Фрагмент програми для генерування даних в таблиці:

def generate\_data(table, rows\_count): # генерує і додає дані в таблицю  
 data\_dict = {}  
 model = get\_model(table)  
 try:  
 column\_types = get\_table\_columns\_and\_types(table, engine)  
 for i in range(rows\_count):  
 for key, value in column\_types.items():  
 reference\_table\_name = get\_referred\_table\_by\_column(key, table, engine)  
 if reference\_table\_name is not None:  
 data\_dict[key] = generate\_random\_value(value, reference\_table\_name, key)  
 else:  
 data\_dict[key] = generate\_random\_value(value)  
 new\_record = model(\*\*data\_dict)  
 session.add(new\_record)  
 session.commit()  
 except Exception as e:  
 session.rollback()  
 raise e

Фрагмент програми для отримання імен та типів стовпчиків таблиці:

def get\_table\_columns\_and\_types(table\_name, engine):  
 inspector = inspect(engine)  
  
 columns = inspector.get\_columns(table\_name)  
 pk\_columns = inspector.get\_pk\_constraint(table\_name)['constrained\_columns']  
  
 column\_types = {column['name']: str(column['type']) for column in columns if column['name'] not in pk\_columns}  
 return column\_types

Фрагмент програми для отримання випадкового занчення

def generate\_random\_value(column\_type, reference\_table\_name=None, column\_name=None):  
 column\_type = column\_type.lower()  
  
 if column\_name:  
 return get\_random\_foreign\_key\_value(reference\_table\_name, column\_name)  
  
 if column\_type == 'text':  
 return ''.join(random.choices(string.ascii\_letters, k=10))  
  
 elif column\_type == 'integer':  
 return random.randint(1, 100)  
  
 elif column\_type == 'timestamp':  
 now = datetime.now()  
 random\_days = random.randint(0, 30)  
 random\_seconds = random.randint(0, 86400)  
 return now - timedelta(days=random\_days, seconds=random\_seconds)  
  
 else:  
 raise ValueError(f"Невідомий тип колонки: {column\_type}")

Фрагмент для отримання випадкового значення з таблиці на яку посилається зовнішній ключ

def get\_random\_foreign\_key\_value(table\_name, column\_name):  
 result = session.execute(text(f"SELECT {column\_name} FROM {table\_name}"))  
 values = [row[0] for row in result.fetchall()]  
  
 if not values:  
 raise ValueError(f"Немає доступних значень у таблиці {table\_name} для колонки {column\_name}.")  
  
 # Повертаємо випадкове значення  
 return random.choice(values)

Фрагмент для отримання назви таблиці на яку посилається зовнішній ключ  
def get\_referred\_table\_by\_column(column\_name, table\_name, engine):  
 inspector = inspect(engine)  
  
 foreign\_keys = inspector.get\_foreign\_keys(table\_name)  
  
 for fk in foreign\_keys:  
  
 if column\_name in fk['constrained\_columns']:  
 referred\_table = fk['referred\_table']  
 return referred\_table  
  
 return None

**Створення індексів**

**BTree**

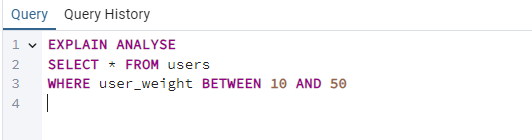
****

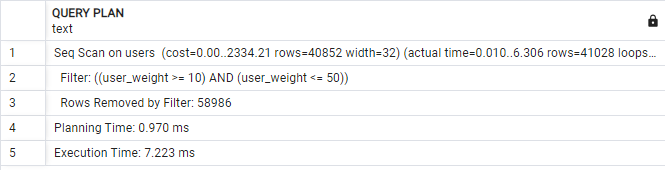
**Hash**

****

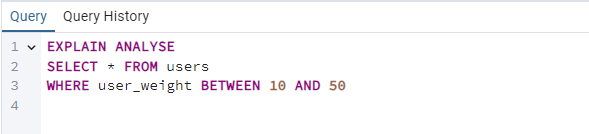
**Приклад 1: Просте фільтрування**

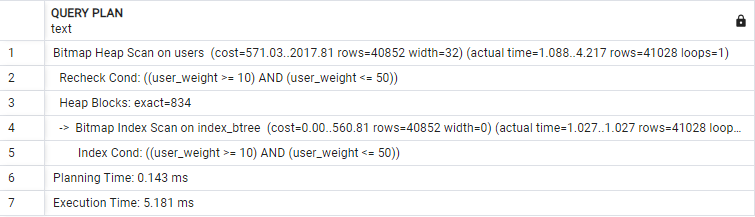
Без індексів:



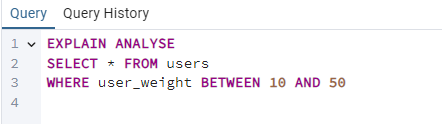


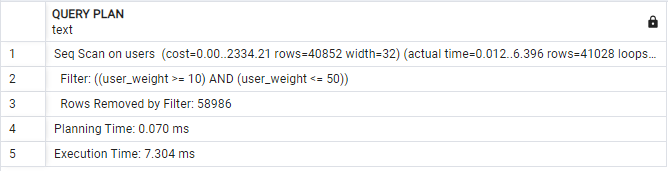
З індексом BTree:





З індексом Hash:

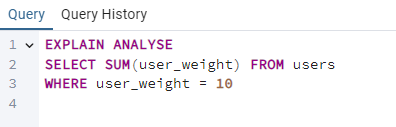


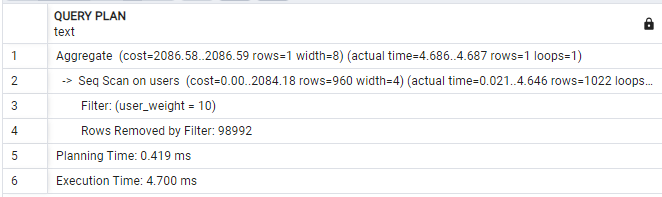


Як бачимо індекс у цьому запиті не використовувався тому що Хеш-коди не мають порядку, тому неможливо визначити діапазон значень.

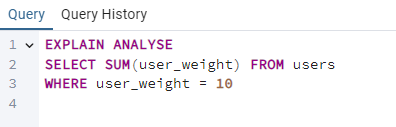
**Приклад 2: Агрегатні функції**

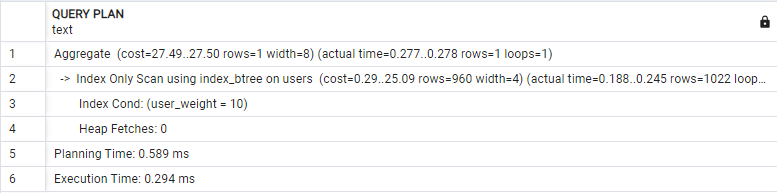
Без індексів:



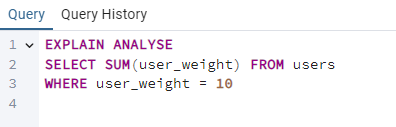


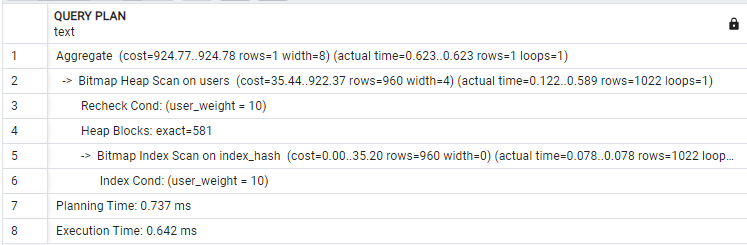
З індексом BTree:





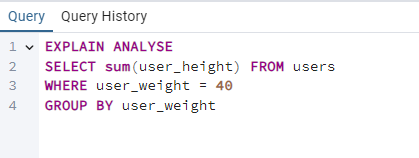
З індексом Hash:

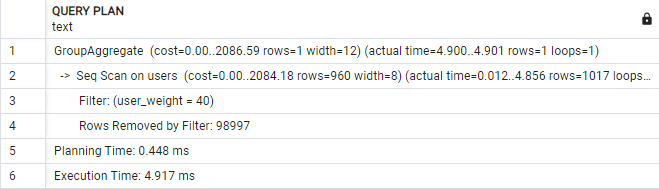




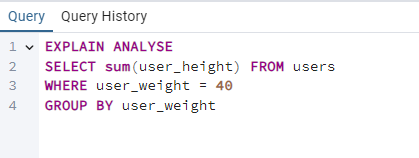
**Приклад 3: Групування**

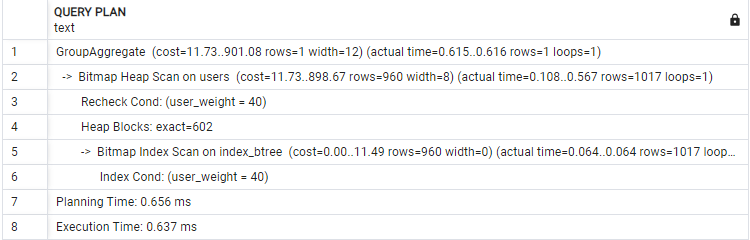
Без індексів:



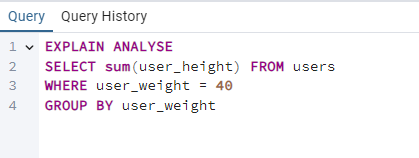


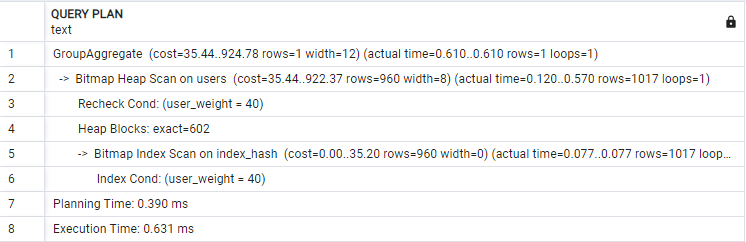
З індексом BTree:





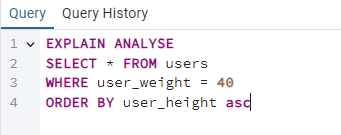
З індексом Hash:

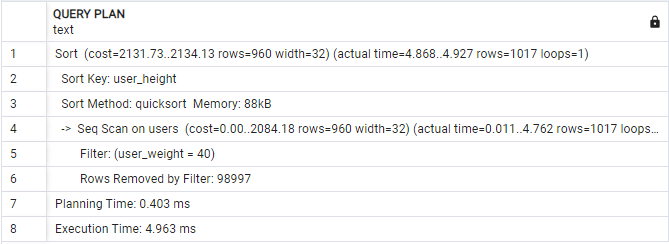




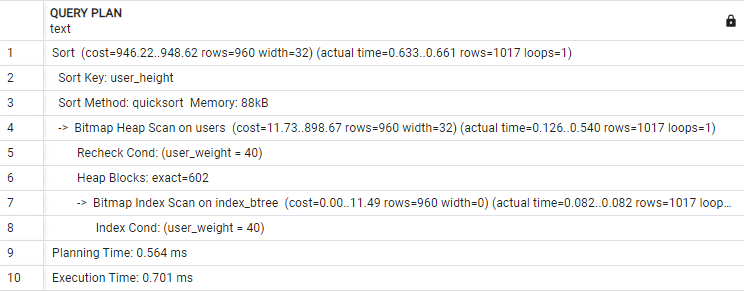
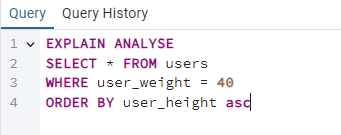
**Приклад 4: Сортування**

Без індексів:

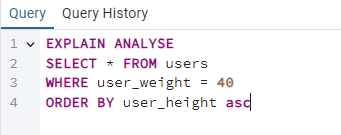


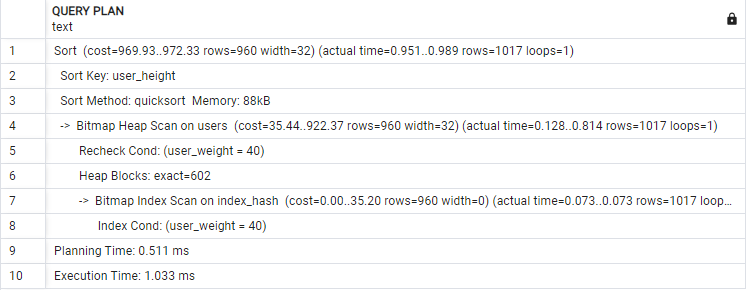


З індексом BTree:



З індексом Hash:





Результати:

Як бачимо на прикладах вище майже завжди найшвидшим варіантом був **BTree** індекс. По ідеї найшвидшим у всіх випадках крім першого (вибірка з діапазону) мав бути **Hash** index. Але через те що в мене було замало рекордів в таблиці найшвидше запити виконував саме **Btree** індекс. Точно можна сказати що для мого розміру таблиці навіть індекс **Hash** виконує запити набагато швидше ніж якщо виконання проходить не використовуючи індекси.

**Btree-індекс (Balanced Tree)**

Гарно працює з:

1.Діапазонні запити

2.Сортування

3.Точний пошук

4.Комбіновані запити

**Hash-індекс**

Гарно працює з:

1.Точний пошук за ключем

2.Запити з оператором рівності

3.Швидкий доступ до окремих записів

**Розробка тригерів**

Створення функції для тригера:

CREATE OR REPLACE FUNCTION delete\_update\_trigger()

RETURNS TRIGGER AS $$

BEGIN

IF TG\_OP = 'DELETE' THEN

RAISE NOTICE 'deleted user with ID: %', OLD.user\_id;

RETURN OLD;

ELSIF TG\_OP = 'UPDATE' THEN

RAISE NOTICE 'Updated user with ID: %', NEW.user\_id;

RETURN NEW;

END IF;

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'An error occurred in delete\_update\_trigger trigger: %s', SQLERRM;

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

Створення тригера для таблиці фабрика:

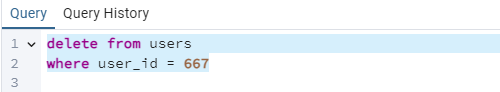
CREATE TRIGGER delete\_update\_trigger\_t

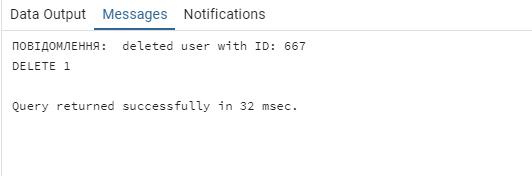
BEFORE DELETE OR UPDATE ON users

FOR EACH ROW

EXECUTE FUNCTION delete\_update\_trigger();

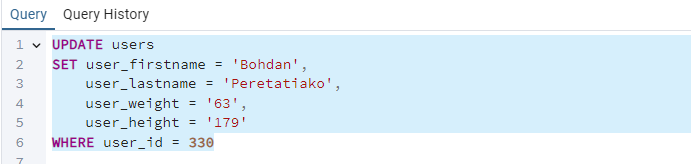
Видалення рядка з таблиці users:

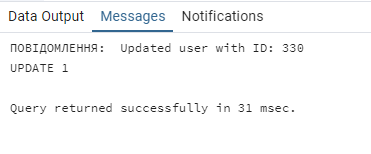




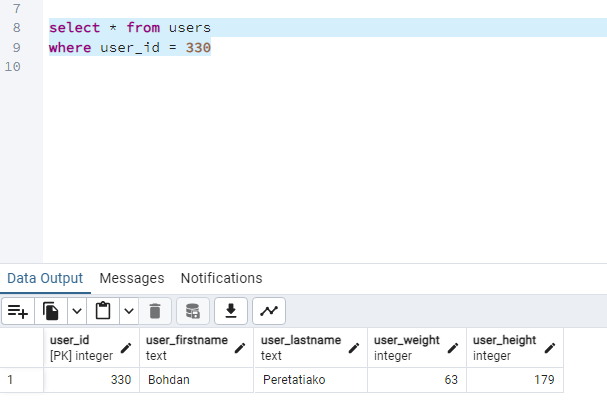
Звідси видно, що тригер спрацював, оскільки вивелося повідомлення про видадення.

Зміна рядка в таблиці users:



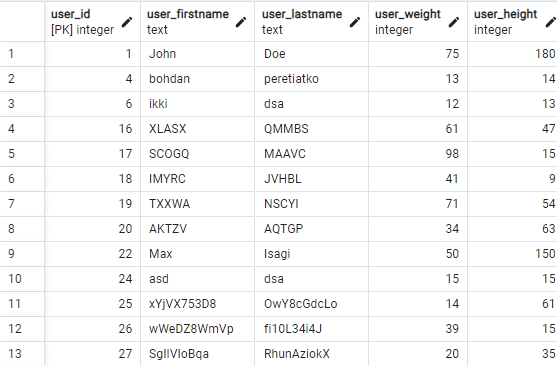


Звідси видно, що тригер спрацював, оскільки вивелося повідомлення про зміну.

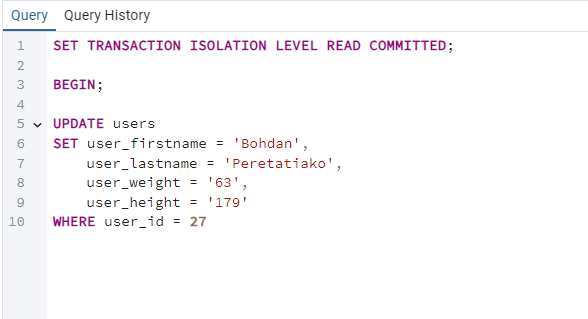


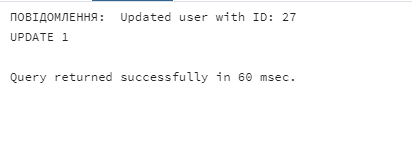
**Використання рівнів ізоляції**

**READ COMMITTED**

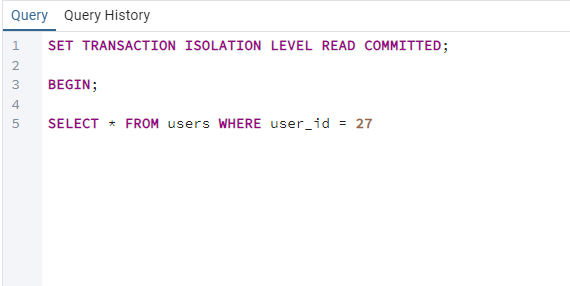
****

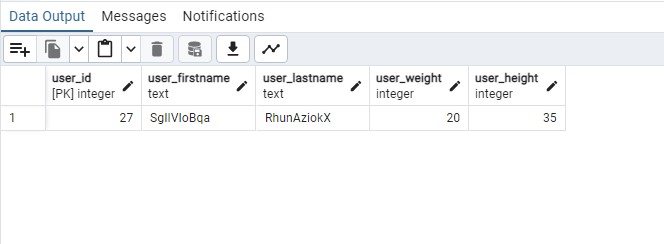
Вікно 1:

****

****

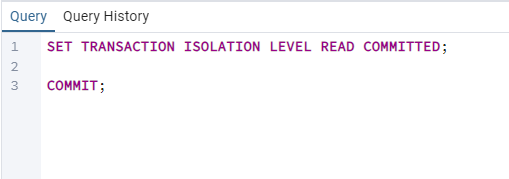
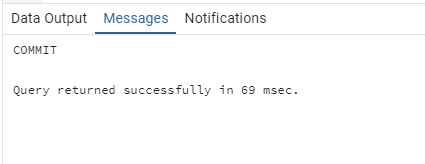
Вікно 2:



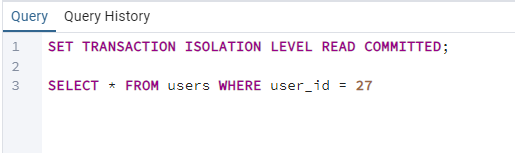


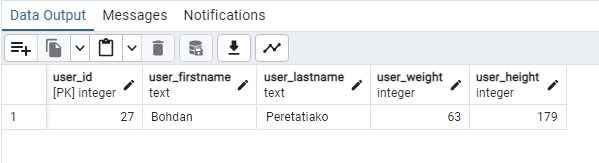
Звідси видно, що без фіксації змін у першому вікні, у другому вікні немає змін і запис залишився таким самим.

Вікно 1:

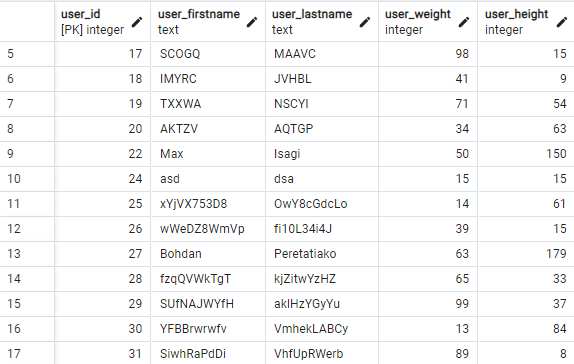
Вікно 2:



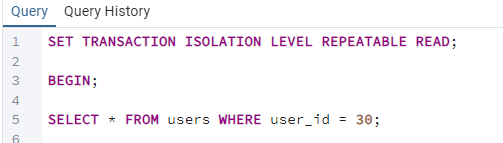


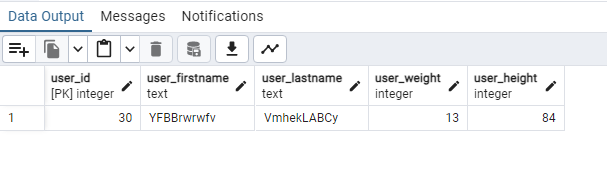
Тепер видно, що після фіксації у першому вікні, у другому вікні відображаються зміни, запис оновлено.

**REPEATABLE READ**

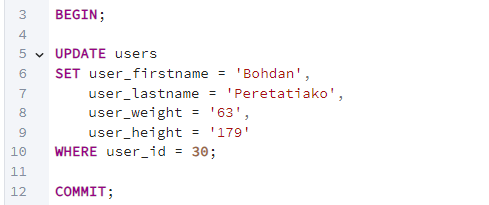
****

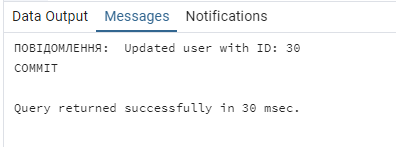
Вікно 1:

****

****

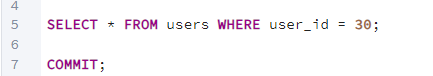
Вікно 2:

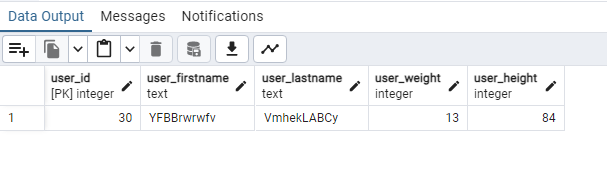




Змінюємо таблицю з фіксацією

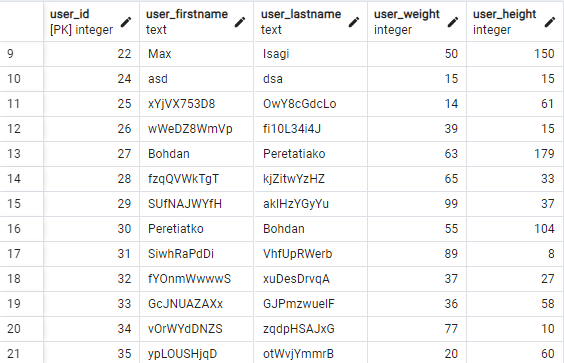
Вікно 1:



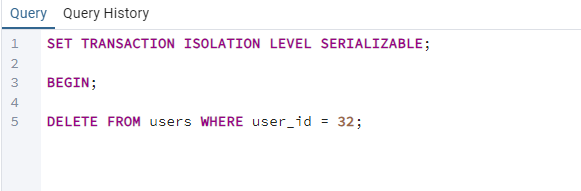
****

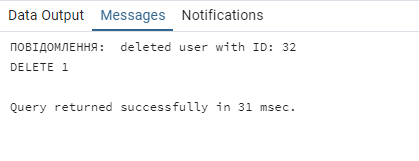
Видно, що навіть після фіксації змін у другому вікні, у першому вікні не відображаються зміни.

**SERIALIZABLE**

****

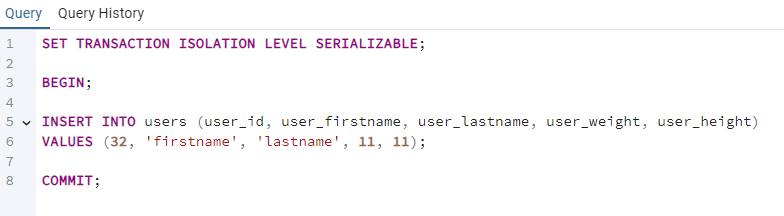
Вікно 1:

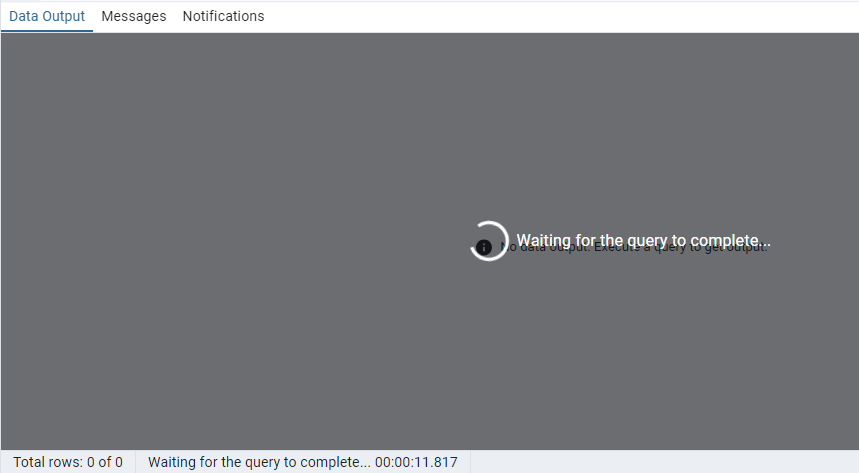
****

****

Видаляємо рядок без фіксації зміни.

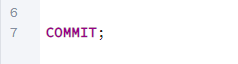
Вікно 2:

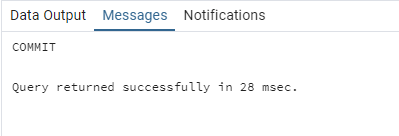




Як видно без фіксації зміни у першому вікні, у другому вікні запит не може виконатися.

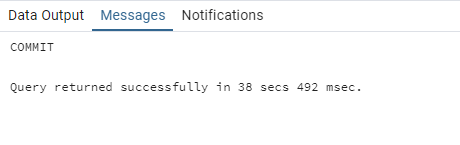
Вікно 1:

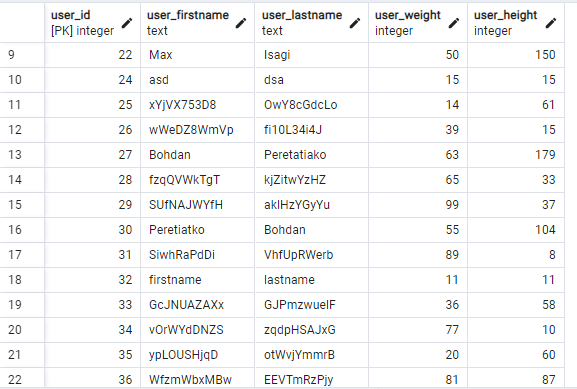




Робимо фіксацію зміни.

Вікно 2:





Тепер у другому вікні запит зміг виконатися, оскільки відбулася фіксація змін у першому вікні.