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-21

Недобойко Д. О.

Перевірив:

**Київ – 2024**

**Репозиторій GitHub:** <https://github.com/yobananes/DB/tree/main/lab2>

**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 №13**

****

У другому завданні проаналізувати індекси BTree, BRIN.

Умова для тригера – before insert, delete

**Опис предметної області**

Предметна область платформа для замовлення та доставки продуктів харчування.. Вона включає комплекс процесів і засобів, призначених для обробки замовлень клієнтів, управління товарною інформацією та організації доставки. Система пропонує клієнтам зручний інтерфейс для створення замовлення і водночас надає інструменти для контролю всіх етапів — від оформлення замовлення до його доставки.

**Опис сутностей**

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

1. Клієнт(User)

Атрибути: ідентифікатор клієнта, ім’я, електронна скринька, номер телефону.

Призначення: збереження даних щодо клієнтів.

1. Замовлення(Order)

Атрибути: ідентифікатор замовлення, дата замовлення .

Призначення: збереження даних щодо замовлень.

1. Позиція їжі (Food)

Атрибути: ідентифікатор їжі, назва їжі, ціна

Призначення: збереження даних щодо товарів.

1. Доставка(Delivery)

Атрибути: ідентифікатор доставки, адреса доставк и, статус доставки.

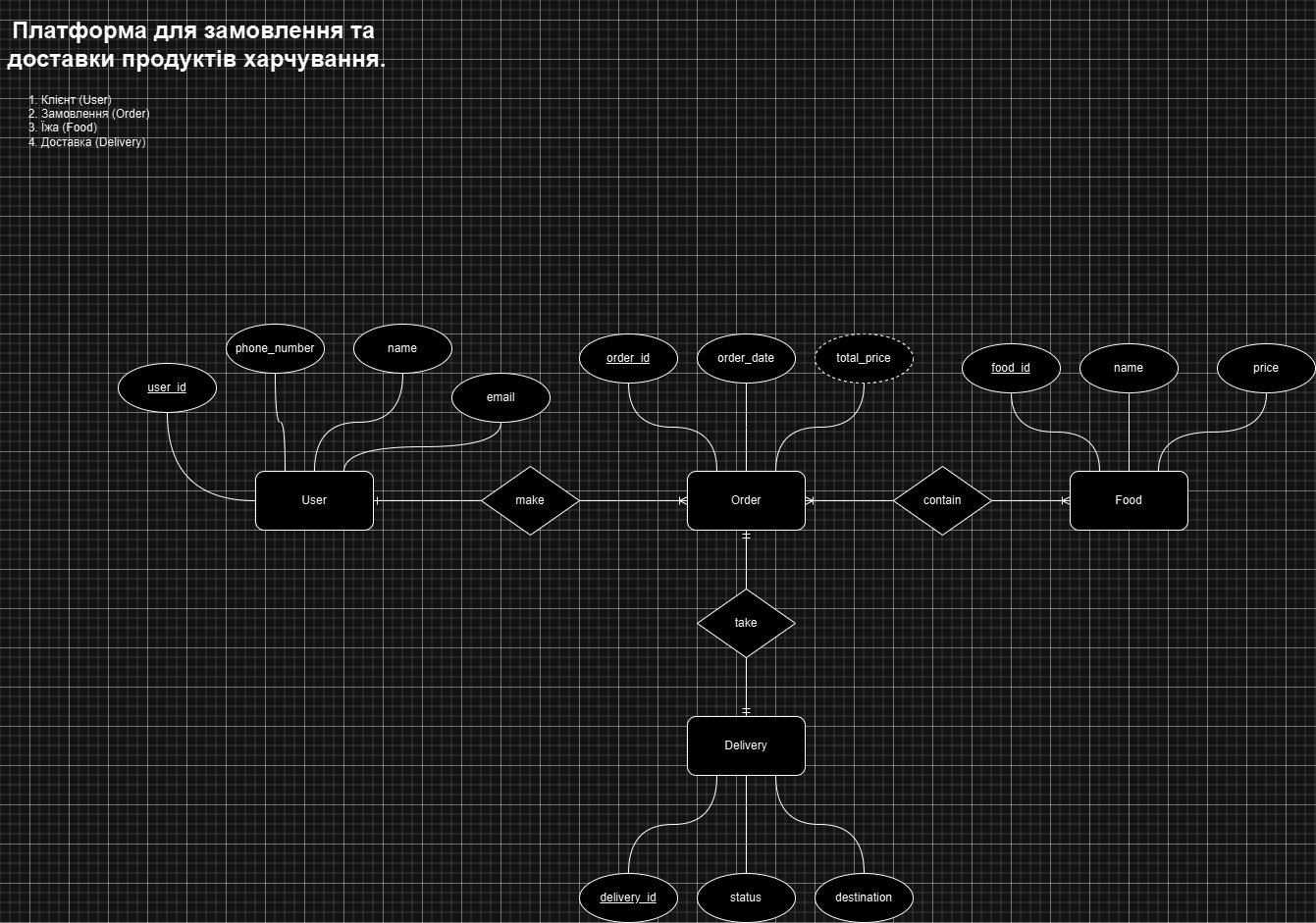
Призначення: збереження даних щодо доставок.

**Опис зв’язків між сутностями**

Зв’язок «Клієнт» - «Замовлення» є зв’язком 1:N. Один клієнт може створити багато замовлень, але кожне замовлення прив’язано лише до одного клієнта.

Зв’язок «Замовлення» - «Їжа» є зв’язком N:M. Одне замовлення може містити багато позицій їжі, і одна позиція може входити до складу багатьох замовлень.

Зв’язок «Замовлення» - «Доставка» є зв’язком 1:1. Кожне замовлення може мати тільки одну доставку, і кожна доставка бере лише по одному замовленню.

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

*Рисунок 1 – ER-діаграма, побудована за нотацією Пташиної лапки (Crow’s foot)*

**Перетворення концептуальної моделі у логічну модель та схему бази даних**

Сутність User перетворено в таблицю User з первинним ключем user\_id та атрибутами name, email, phone.

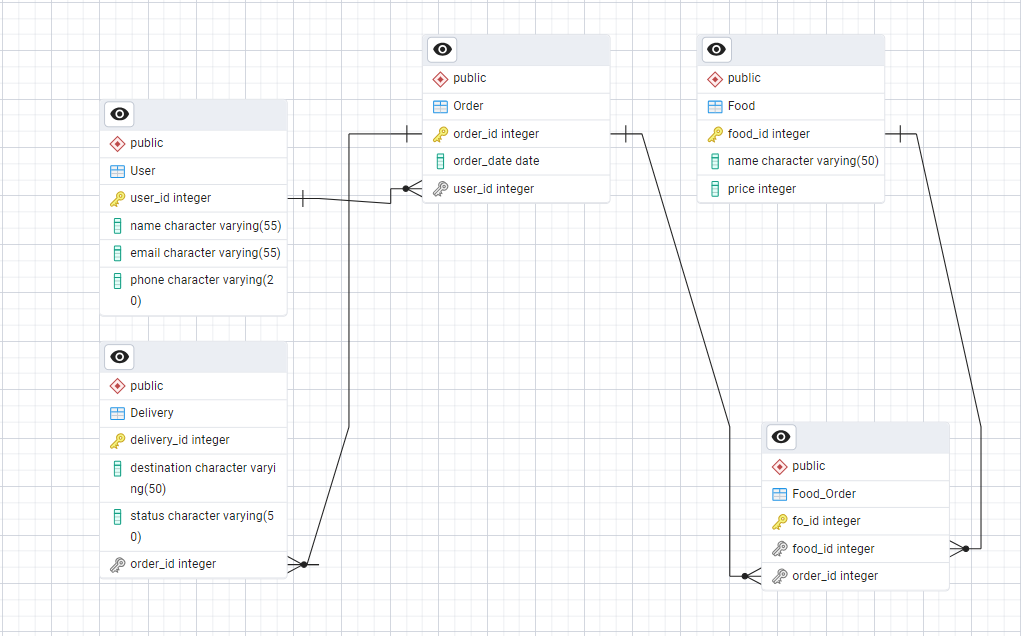
Сутність Order перетворено в таблицю Order з первинним ключем order\_id та атрибутом order\_ date та зовнішнім ключем user\_id.

Сутність Delivery перетворено в таблицю Delivery з первинним ключем delivery\_id та атрибутами destination, status, та зовнішнім ключем order\_id.

Сутність Food перетворено в таблицю Food з первинним ключем food\_id та атрибутами name, price.

Оскільки в логічній моделі безпосередній зв’язок N:M є неможливим, а в концептуальній моделі він існує між сутностями Food i Order, то для його реалізації було створено таблицю Food\_Order, з первинним ключем fo\_id, та зовнішніми ключами food\_id i order\_id.

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



*Рисунок 2 – Схема бази даних*

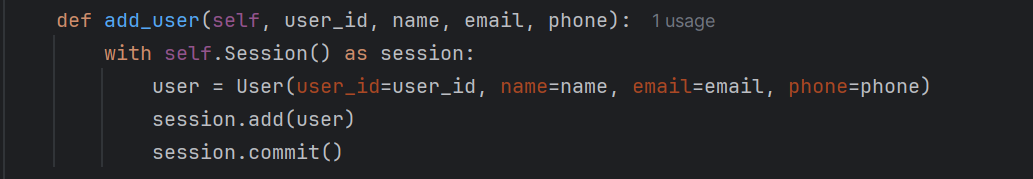
**Завдання №1**

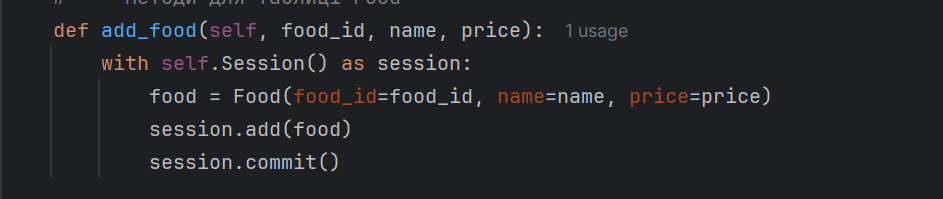
Для перетворення модулів програми, створених в розрахунковій роботі, у вигляд об'єктно-реляційної моделі було використано:

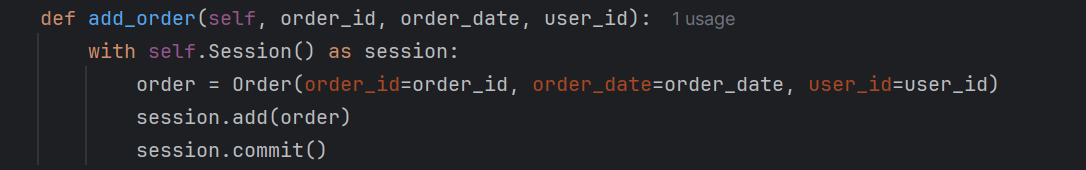
1. Бібліотека для реалізації ORM - SQLAlchemy для Python.
2. Середовище для відлагодження SQL-запитів до бази даних – pgAdmin 4.
3. СУБД - PostgreSQL 16.

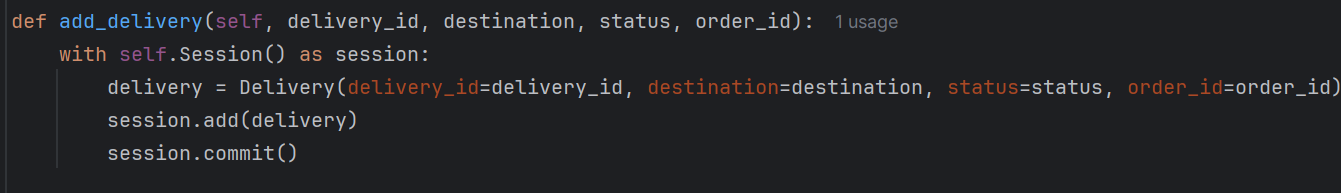
Також код буде надано в репозиторії **GitHub**.

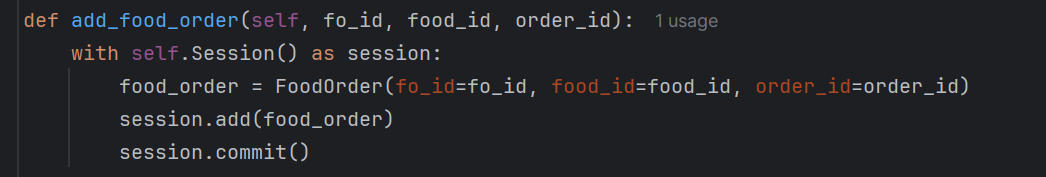
**Додавання**

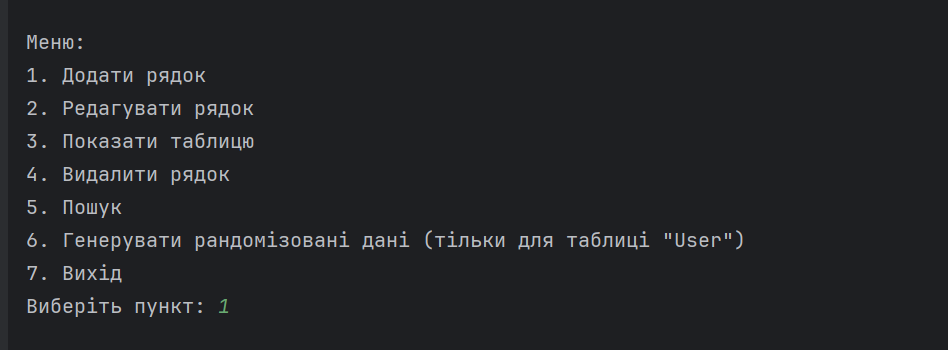


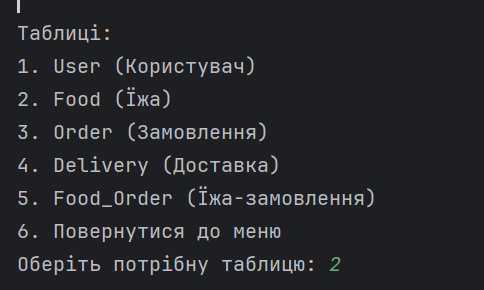


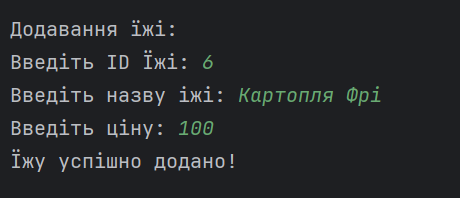


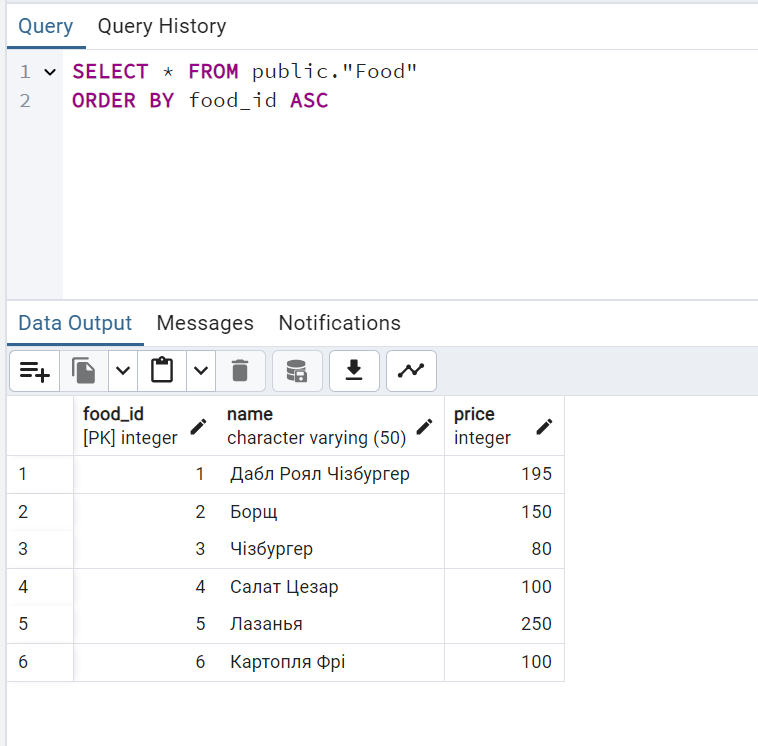




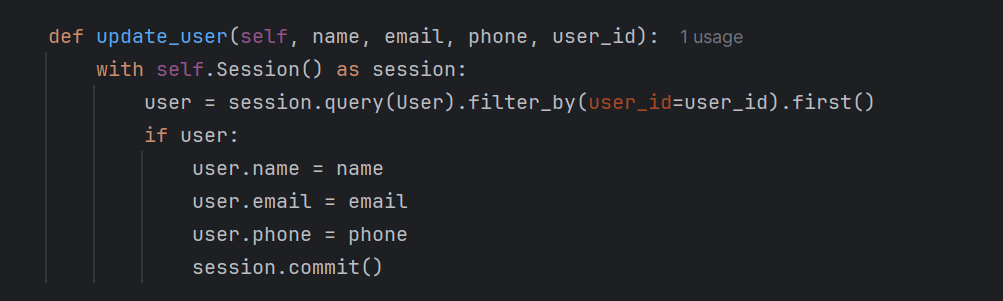


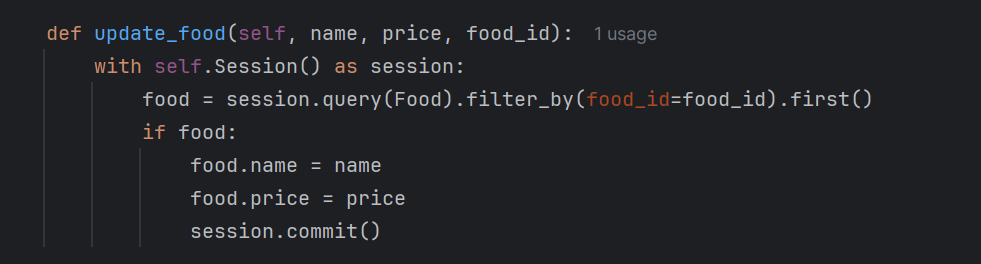


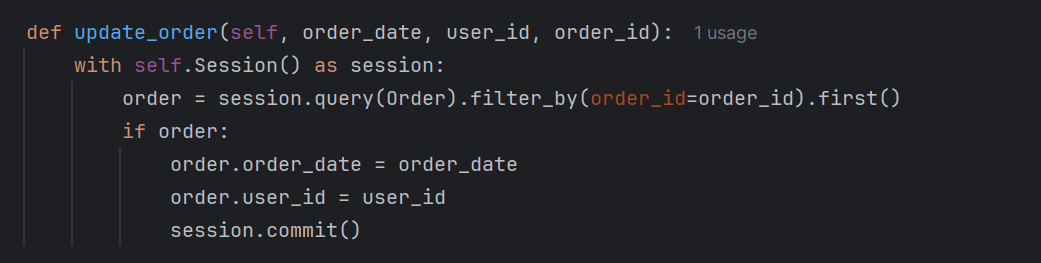


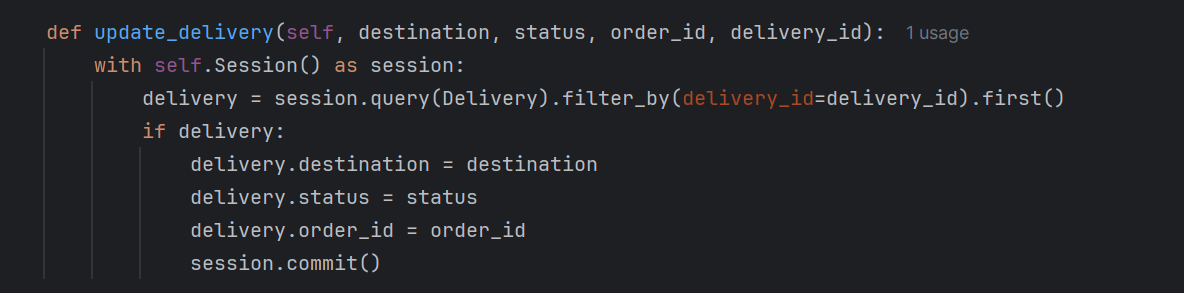


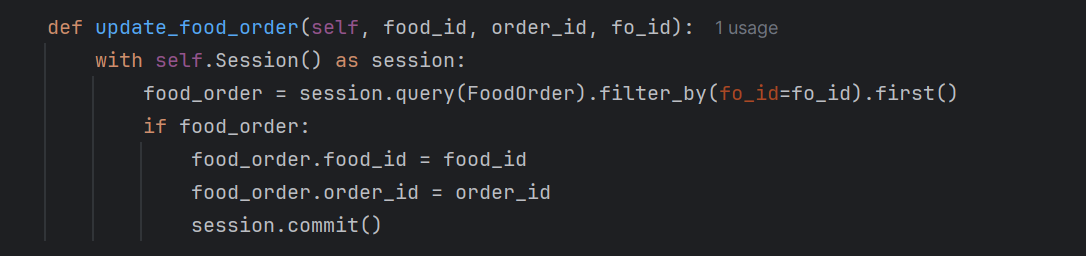
**Редагування:**



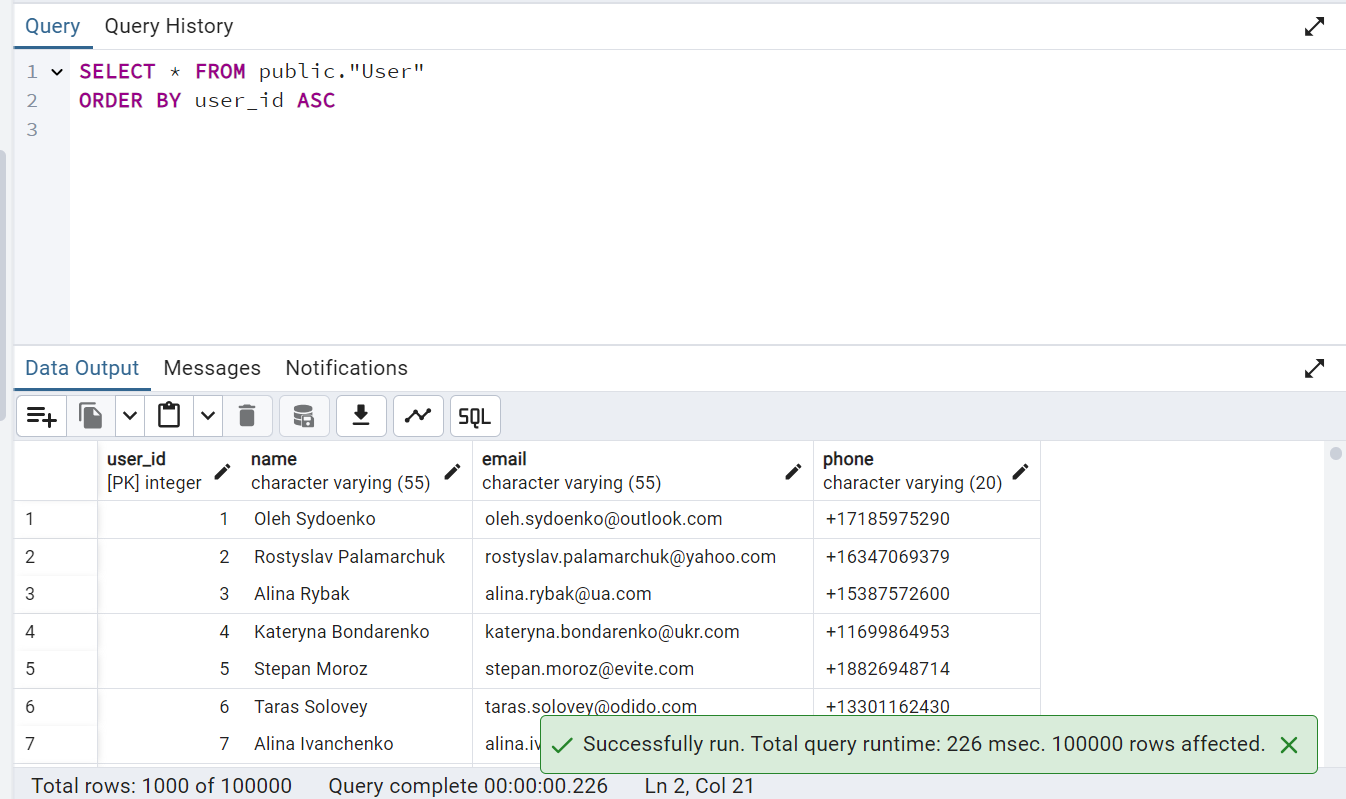




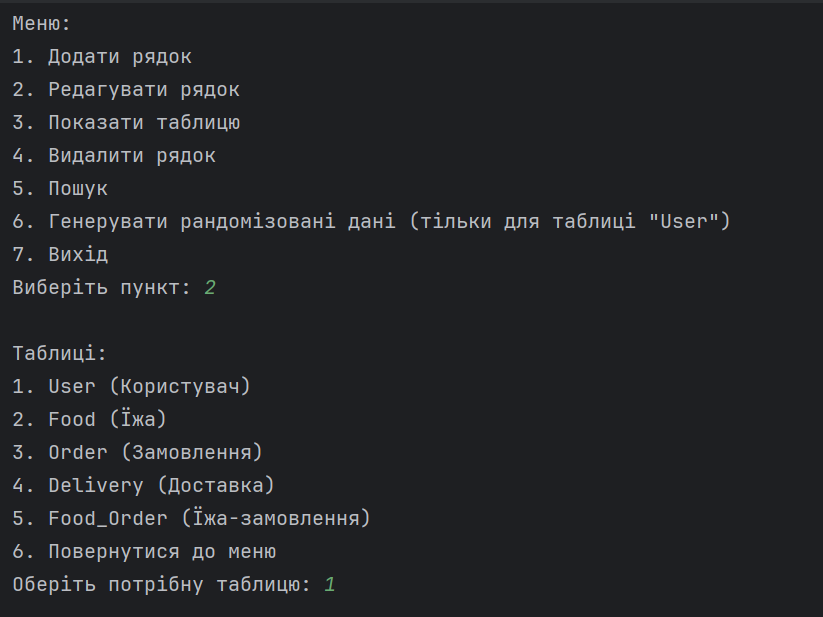


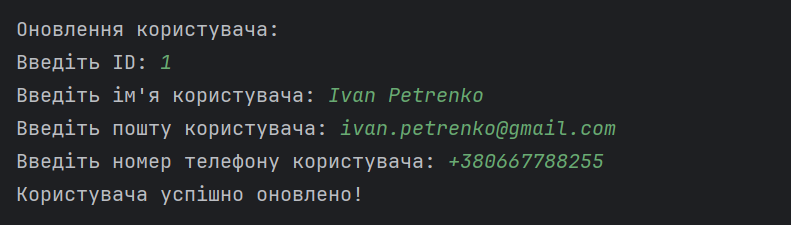


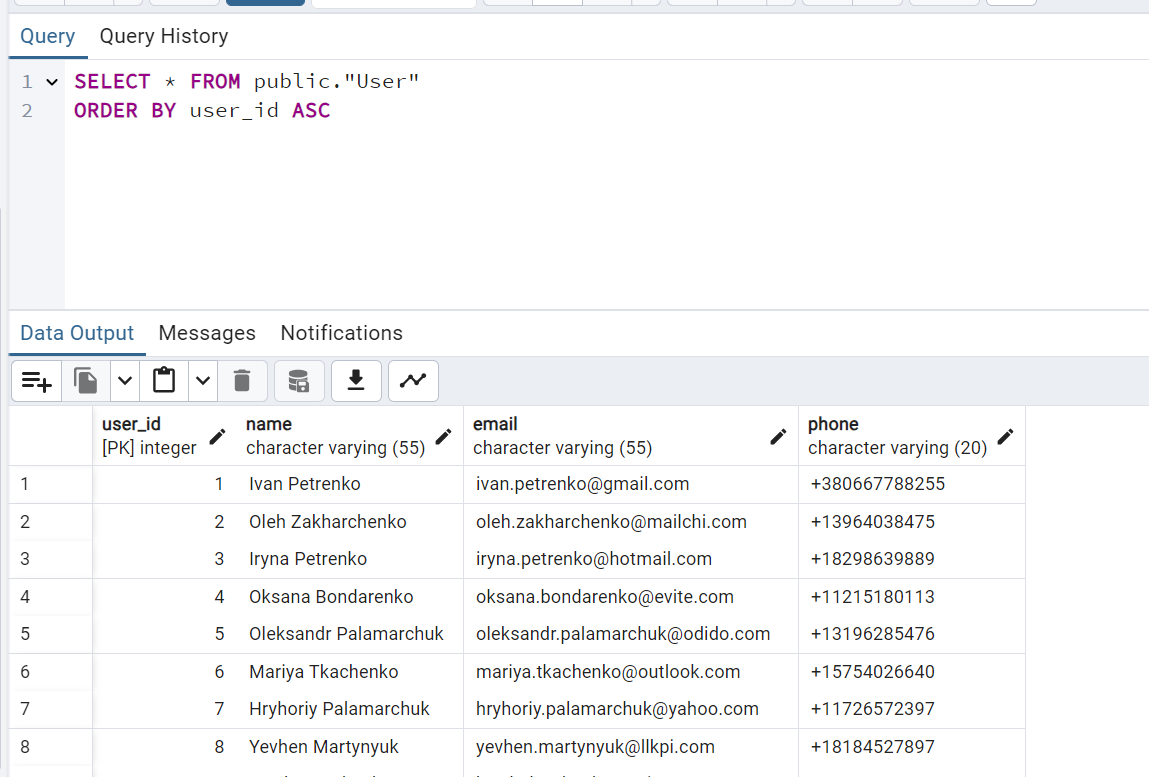
Було:



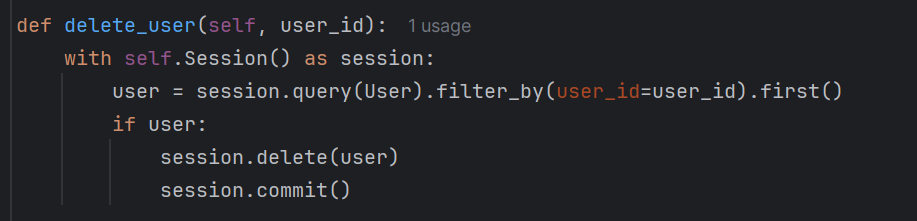
Стало:

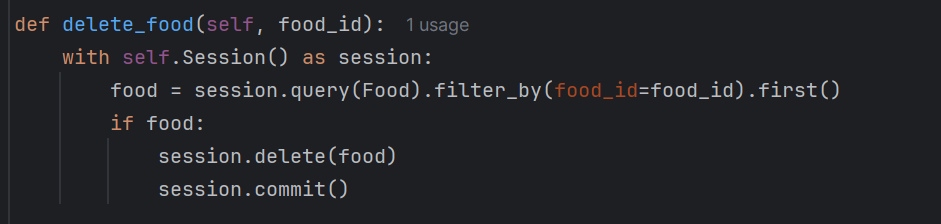


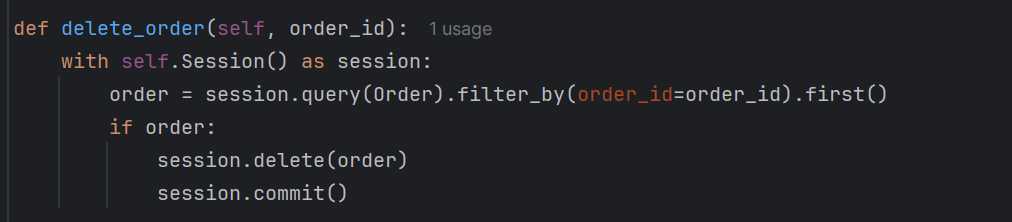




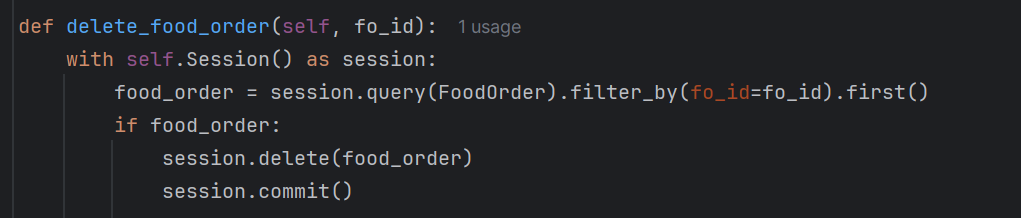
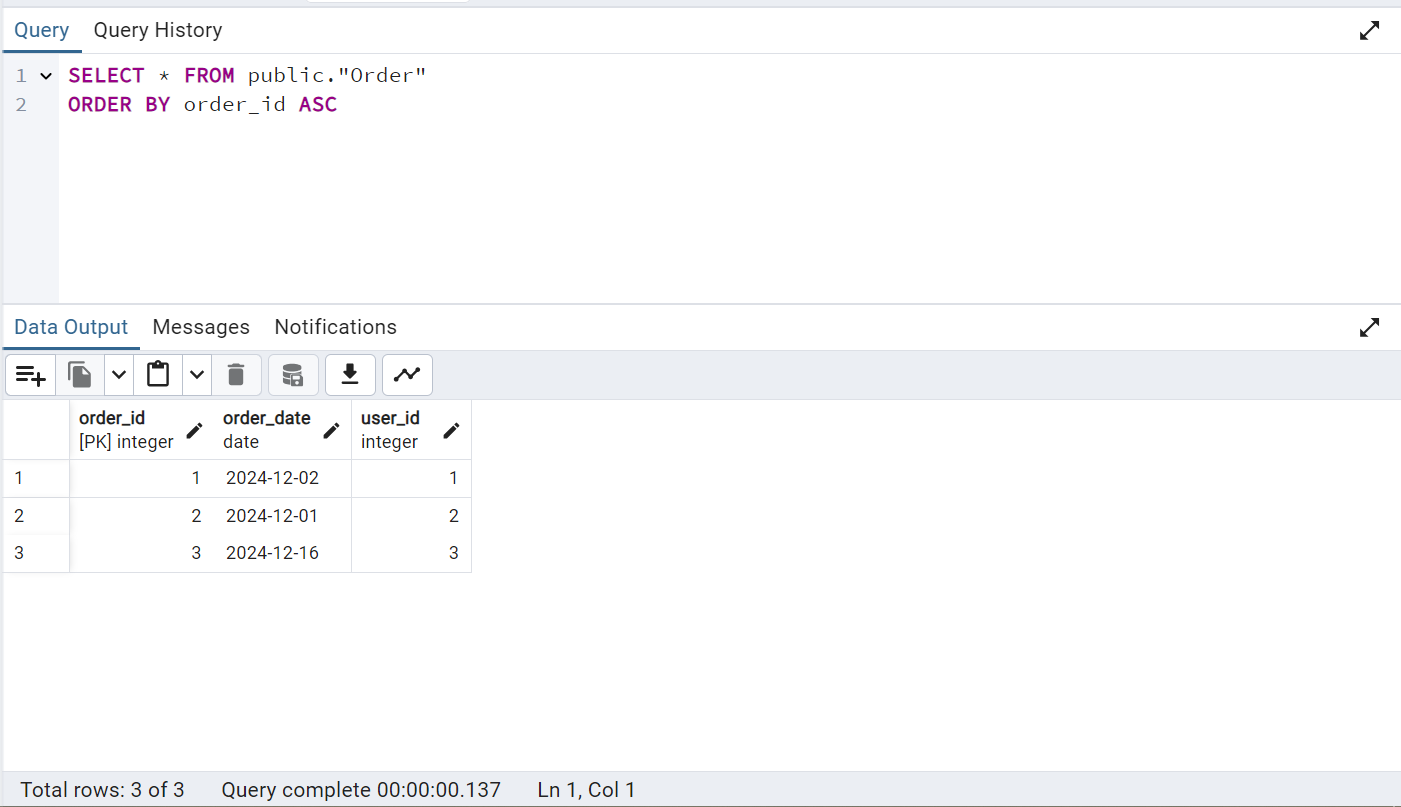
**Видалення:**

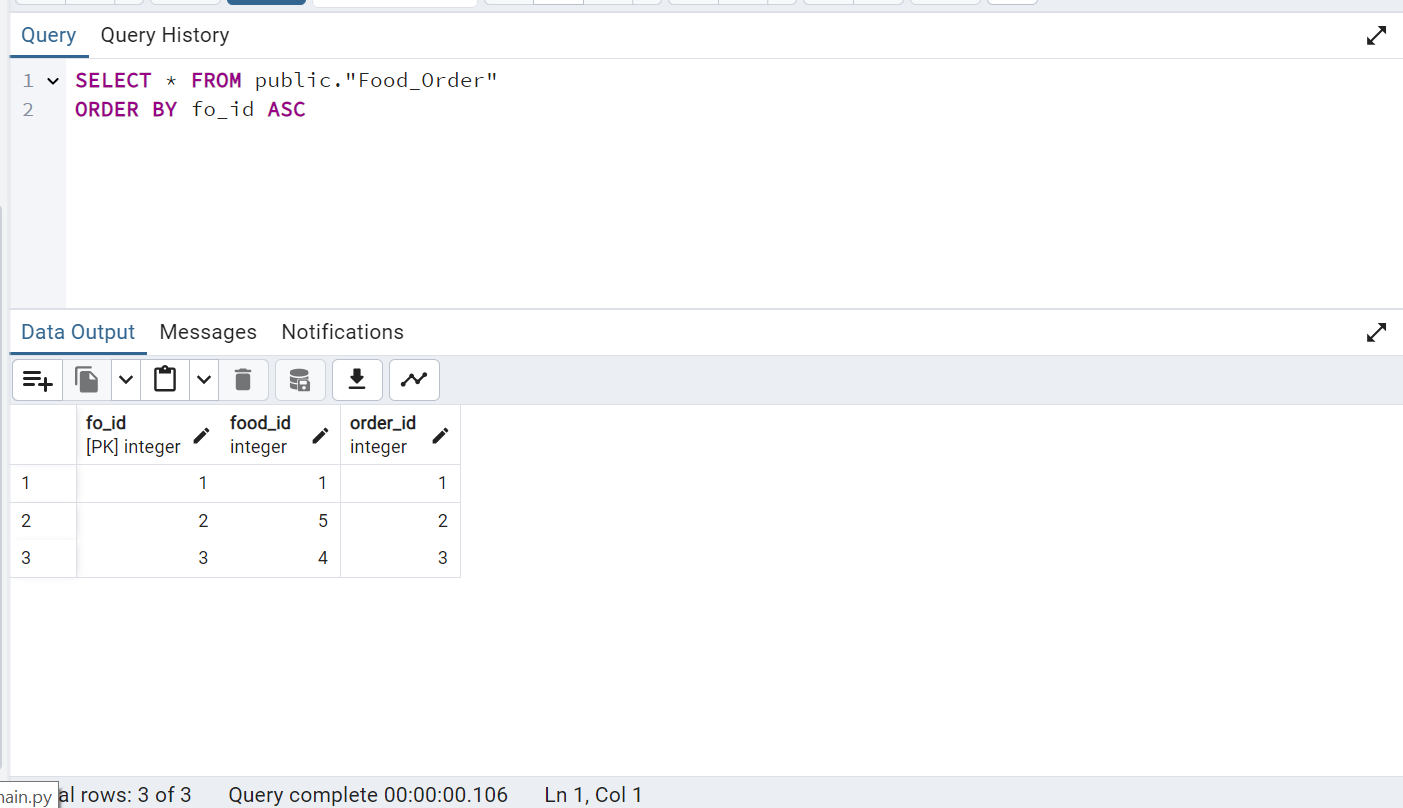


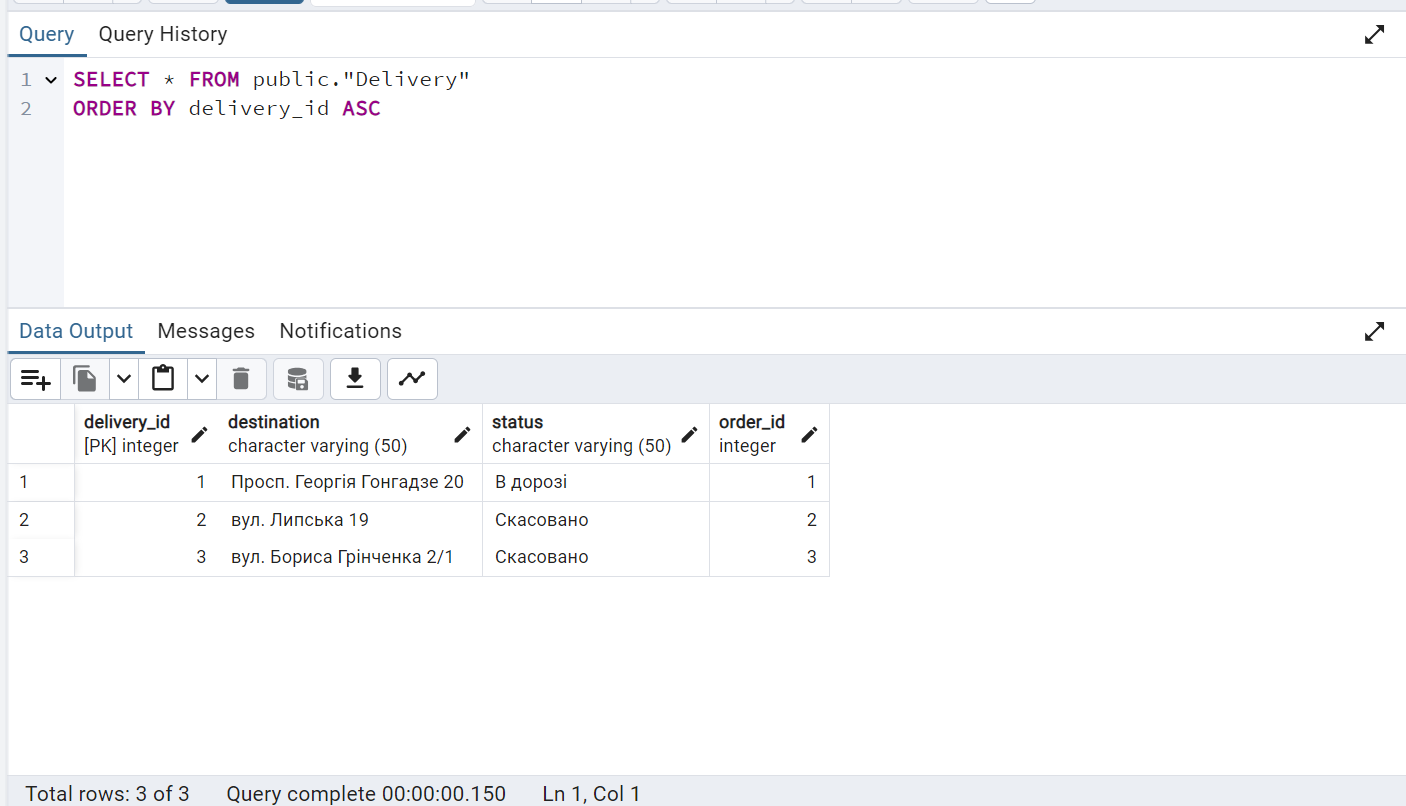




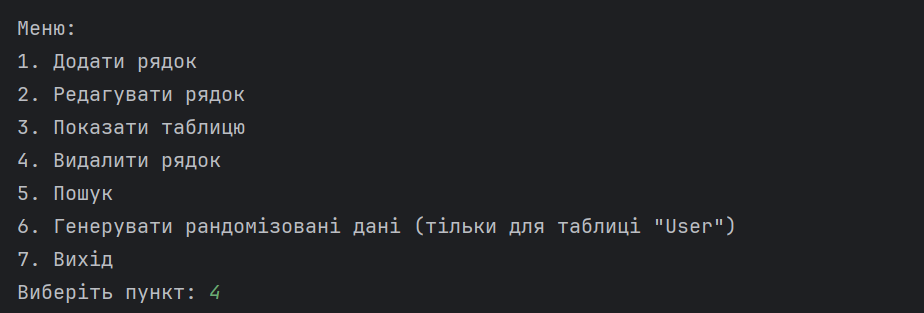


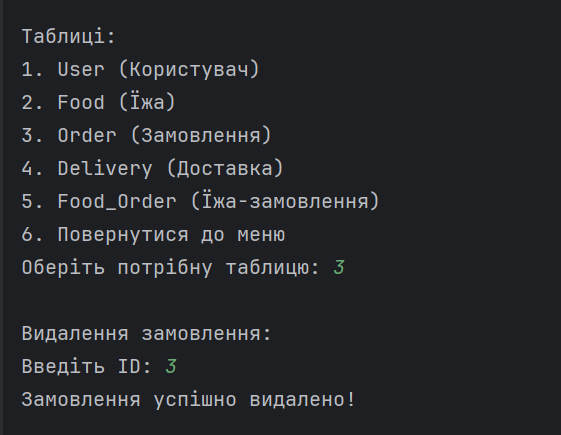
  
Було:  


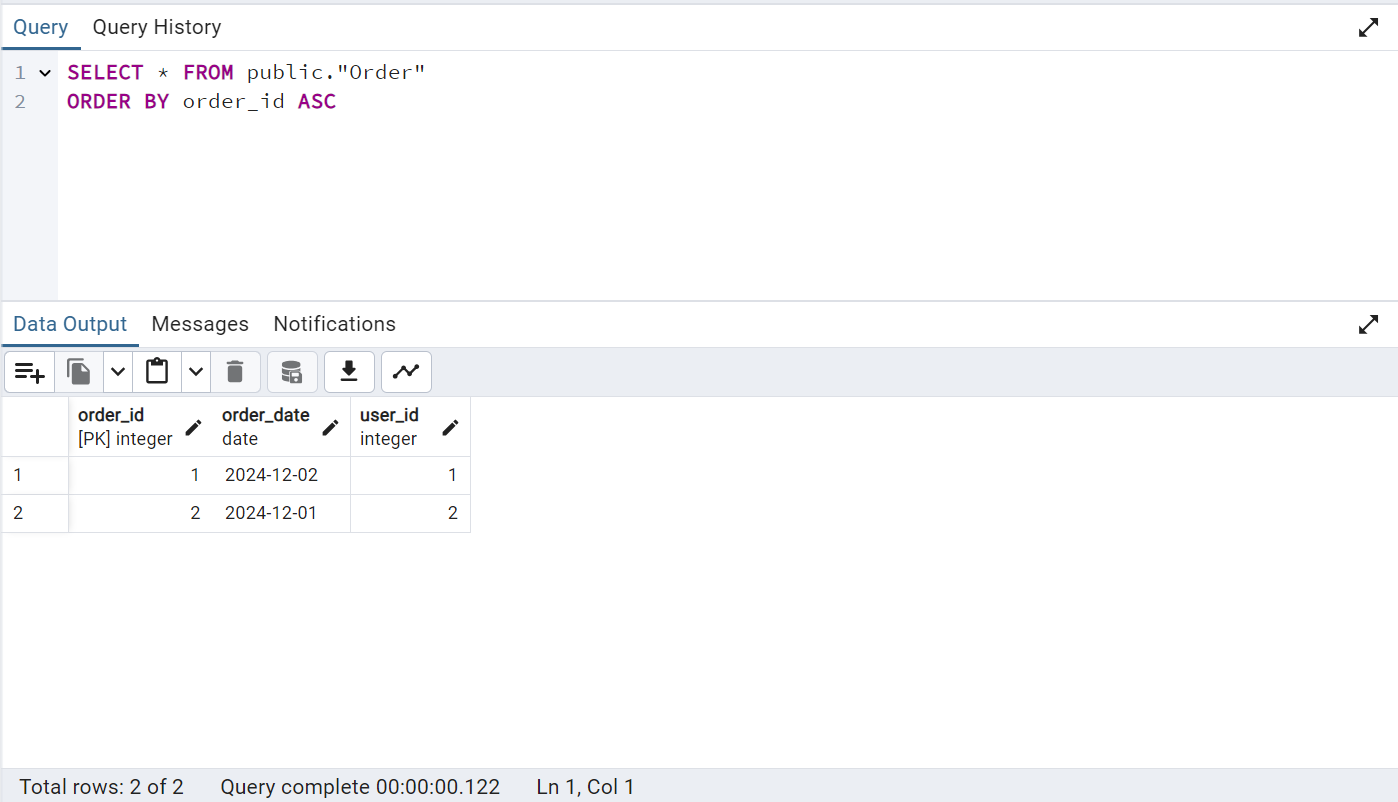


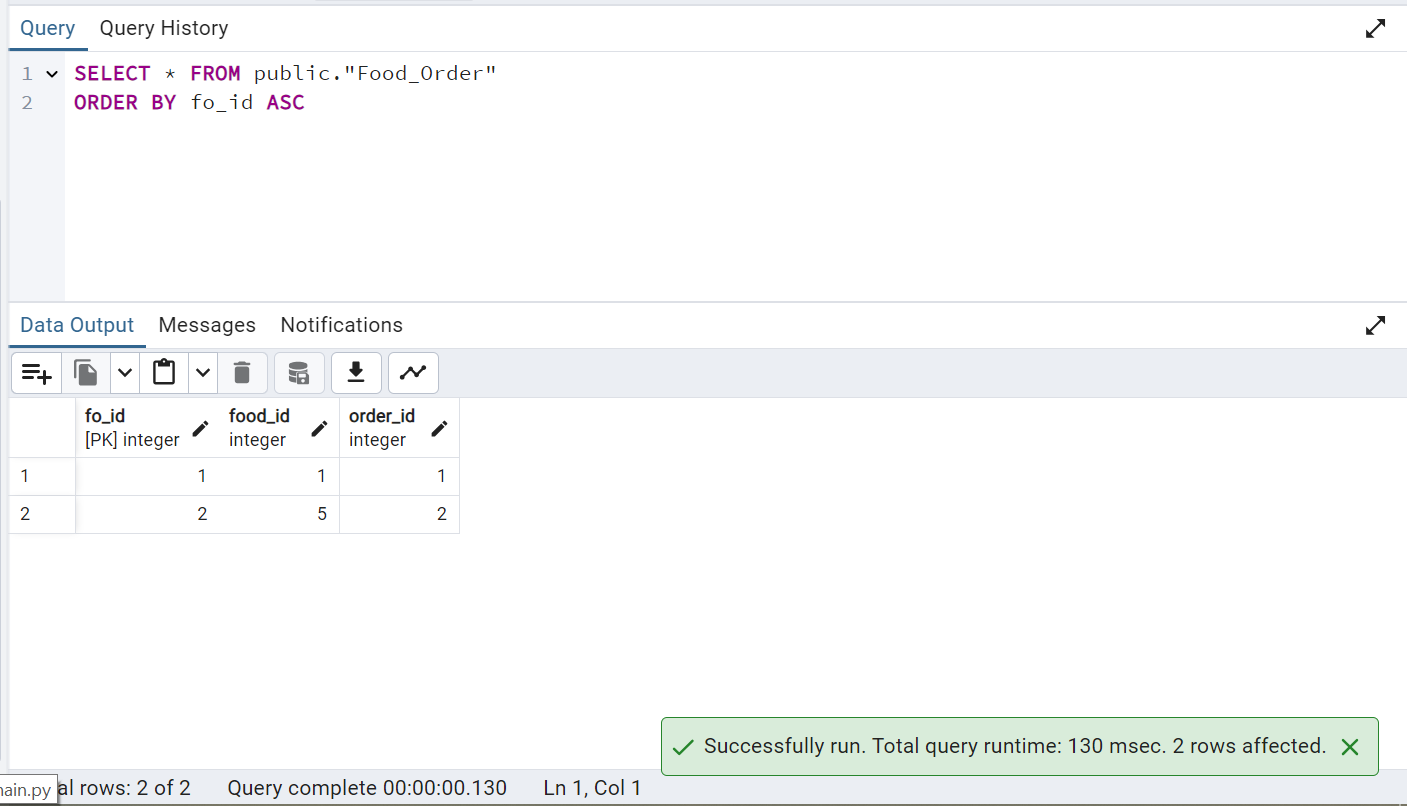


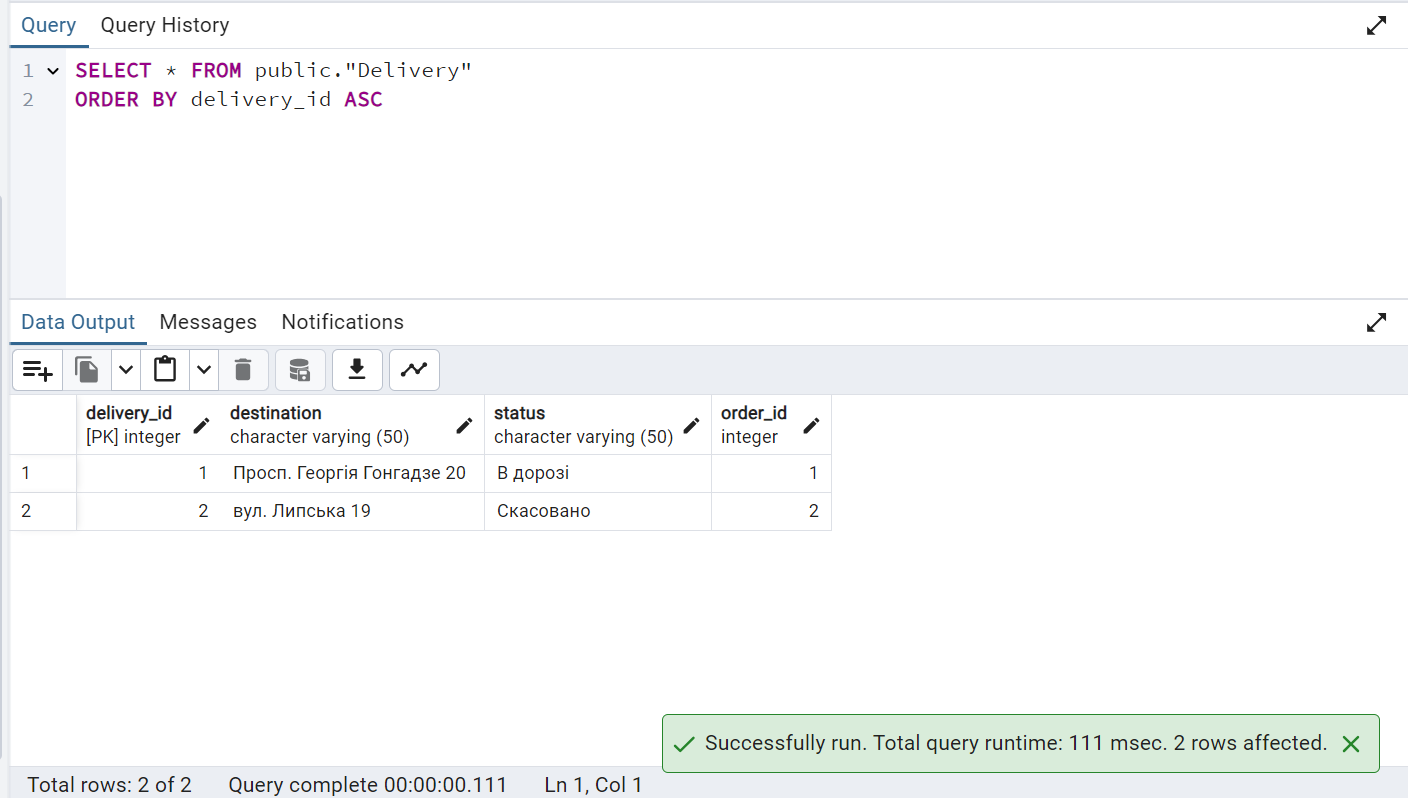
Стало(видалилось каскадно):









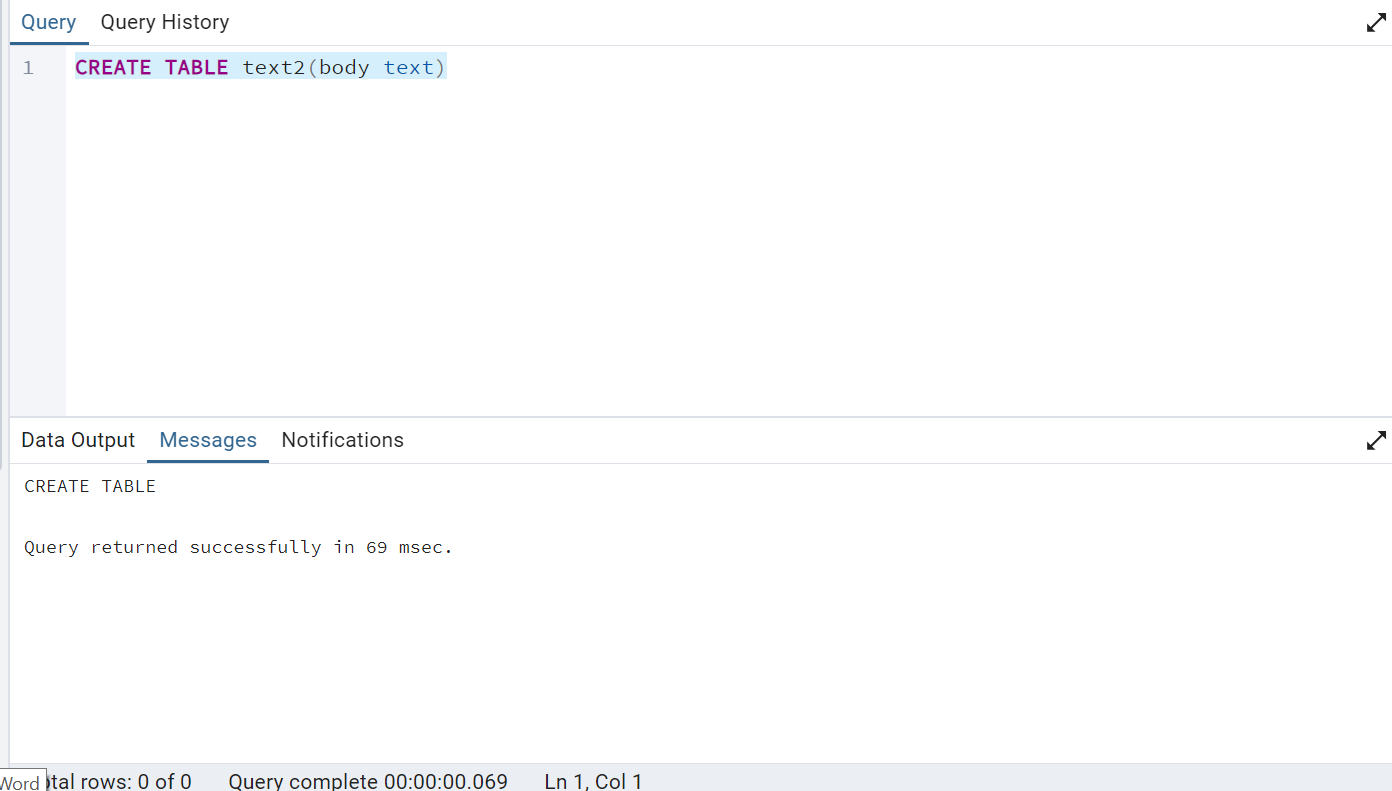


**Завдання №2**

**BTree**

Створення таблиці для індексу Btree:

CREATE TABLE text2(body text)



Вставка та генерація даних в таблиці:

INSERT INTO text2

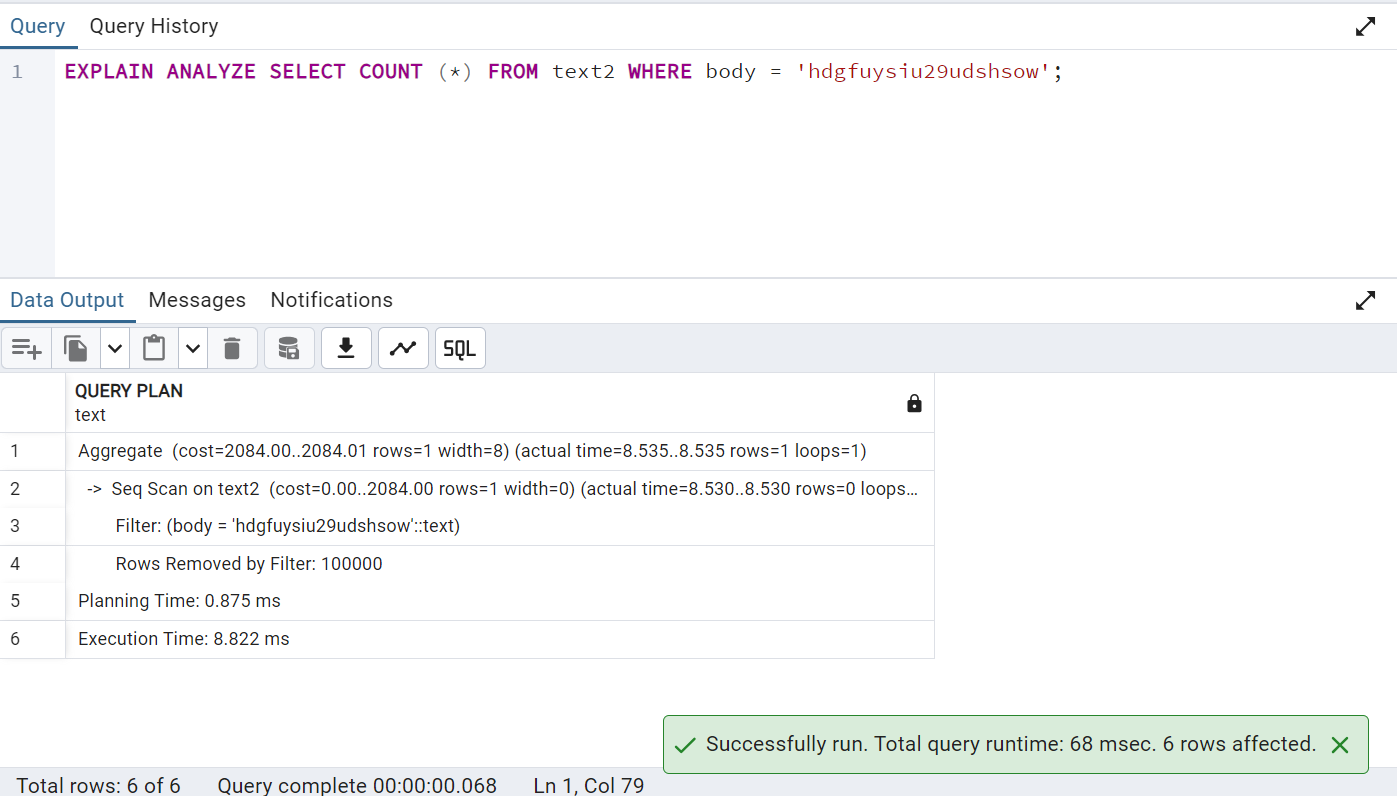
SELECT

md5(random()::text)

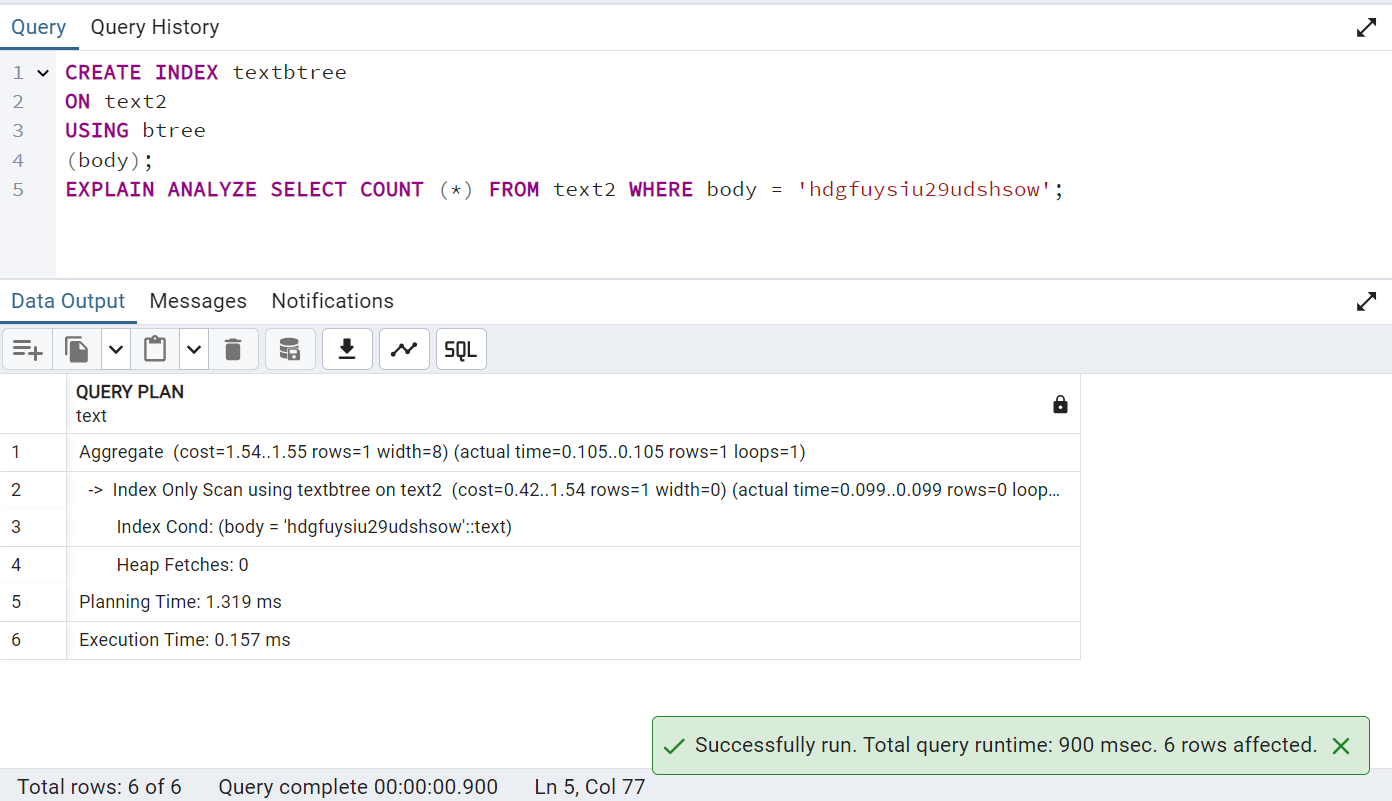
FROM(SELECT \* FROM generate\_series(1,100000)AS id)AS x;



Час виконання без індексу:



Час виконання з використанням індексу:



Індекс B-дерева (BTree) — це структура даних, яка застосовується в базах даних для швидкого пошуку, вставки та видалення записів. B-дерева є ефективними для роботи з великими обсягами невпорядкованих даних, зокрема у сховищах типу ключ-значення, базах даних і файлових системах.

Основні властивості B-дерева:

* **Балансованість**: B-дерево завжди залишається збалансованим, тобто шлях від кореня до будь-якого листового вузла має приблизно однакову довжину. Це досягається завдяки перерозподілу ключів під час вставки або видалення.
* **Ефективність пошуку**: Завдяки своїй структурі та балансованості, час пошуку становить O(log n), де n — кількість ключів у дереві.
* **Підтримка діапазонних запитів**: B-дерево дозволяє ефективно виконувати діапазонні запити, оскільки ключі зручно відстежуються у порівнянні з бінарними деревами пошуку (BST).
* **Підтримка вставки та видалення**: Операції вставки та видалення виконуються ефективно завдяки адаптивності дерева до змін.

BTree широко застосовуються в базах даних для індексації даних, що дозволяє прискорити доступ до записів, знизити кількість блокувань і підвищити загальну продуктивність системи.

**BRIN**

Створення таблиці для Brin індексу:

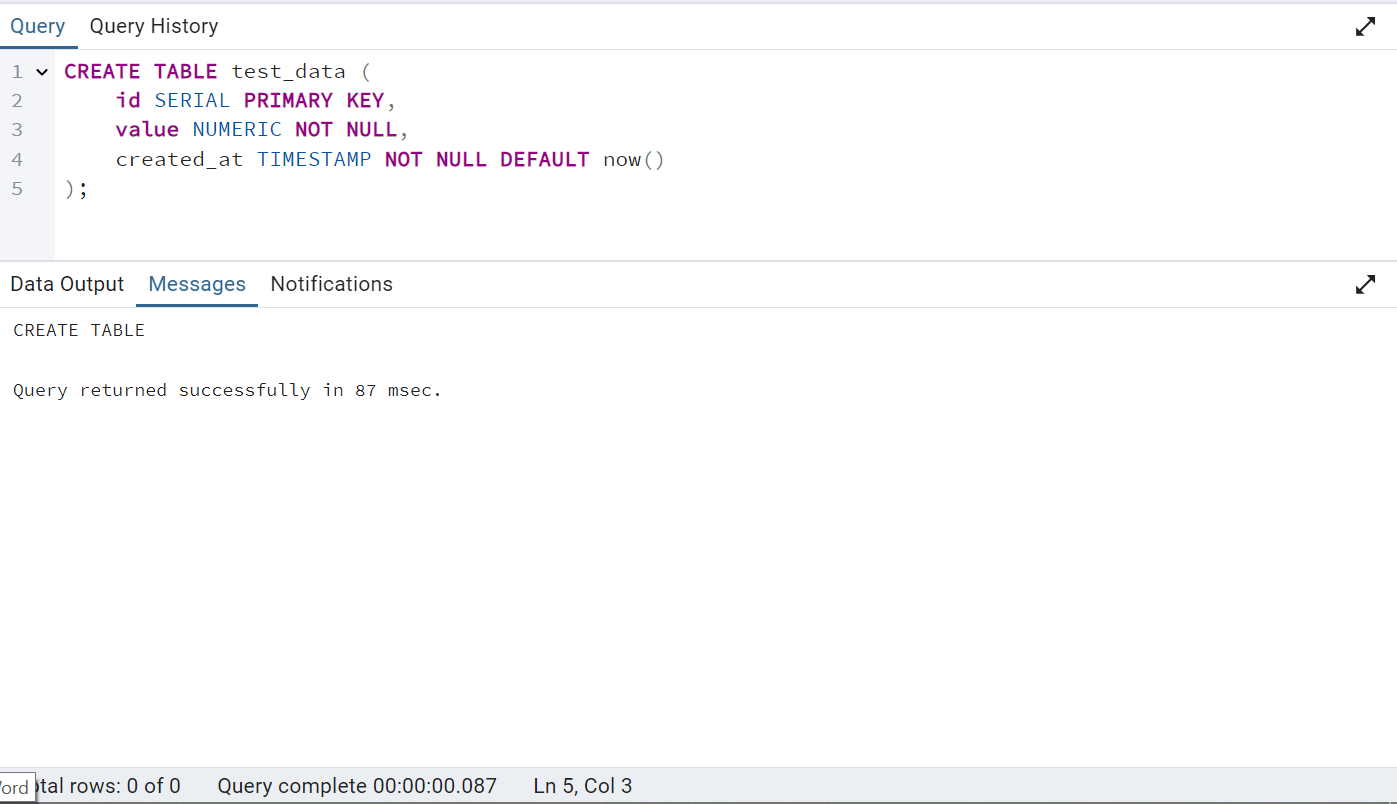
CREATE TABLE test\_data (

id SERIAL PRIMARY KEY,

value NUMERIC NOT NULL,

created\_at TIMESTAMP NOT NULL DEFAULT now()

);



Вставка та генерація даних в таблиці:

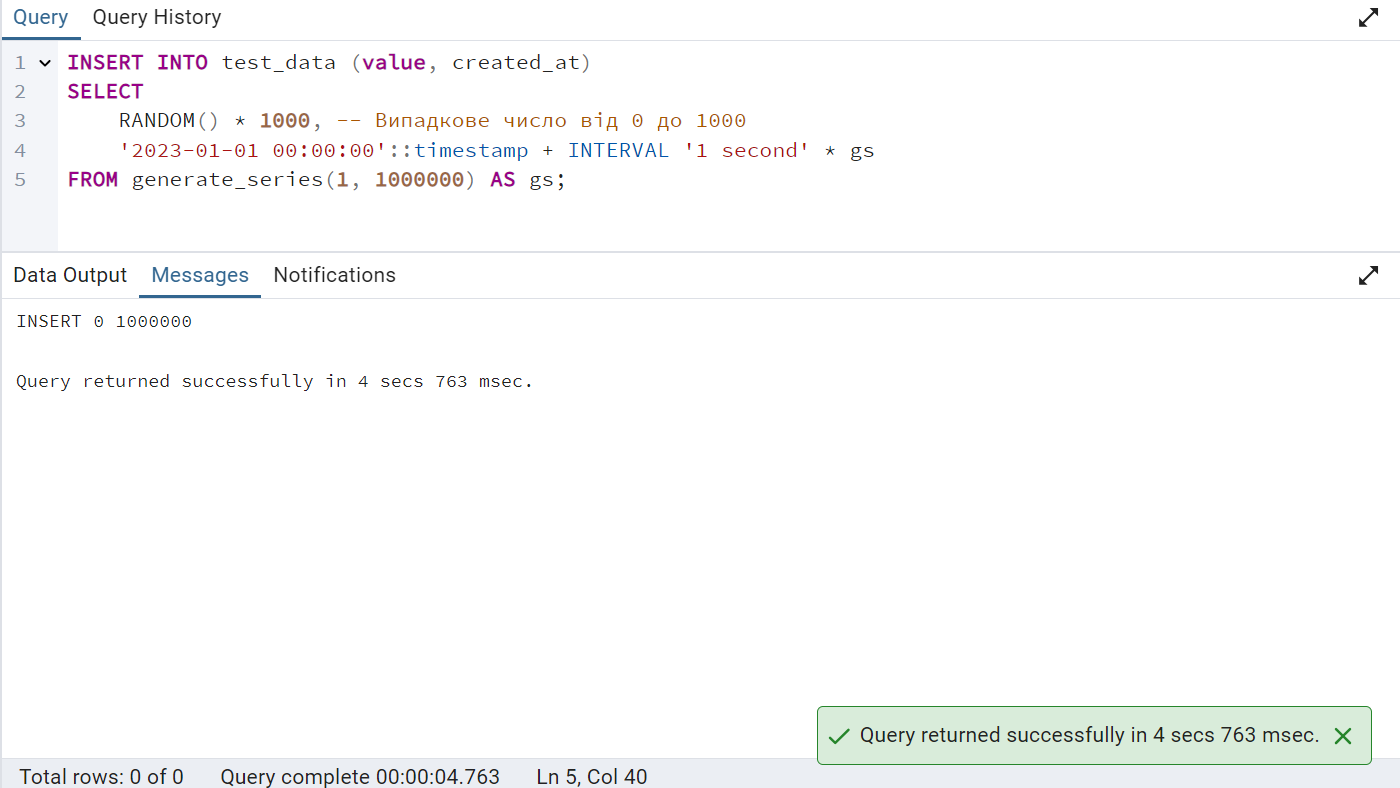
INSERT INTO test\_data (value, created\_at)

SELECT

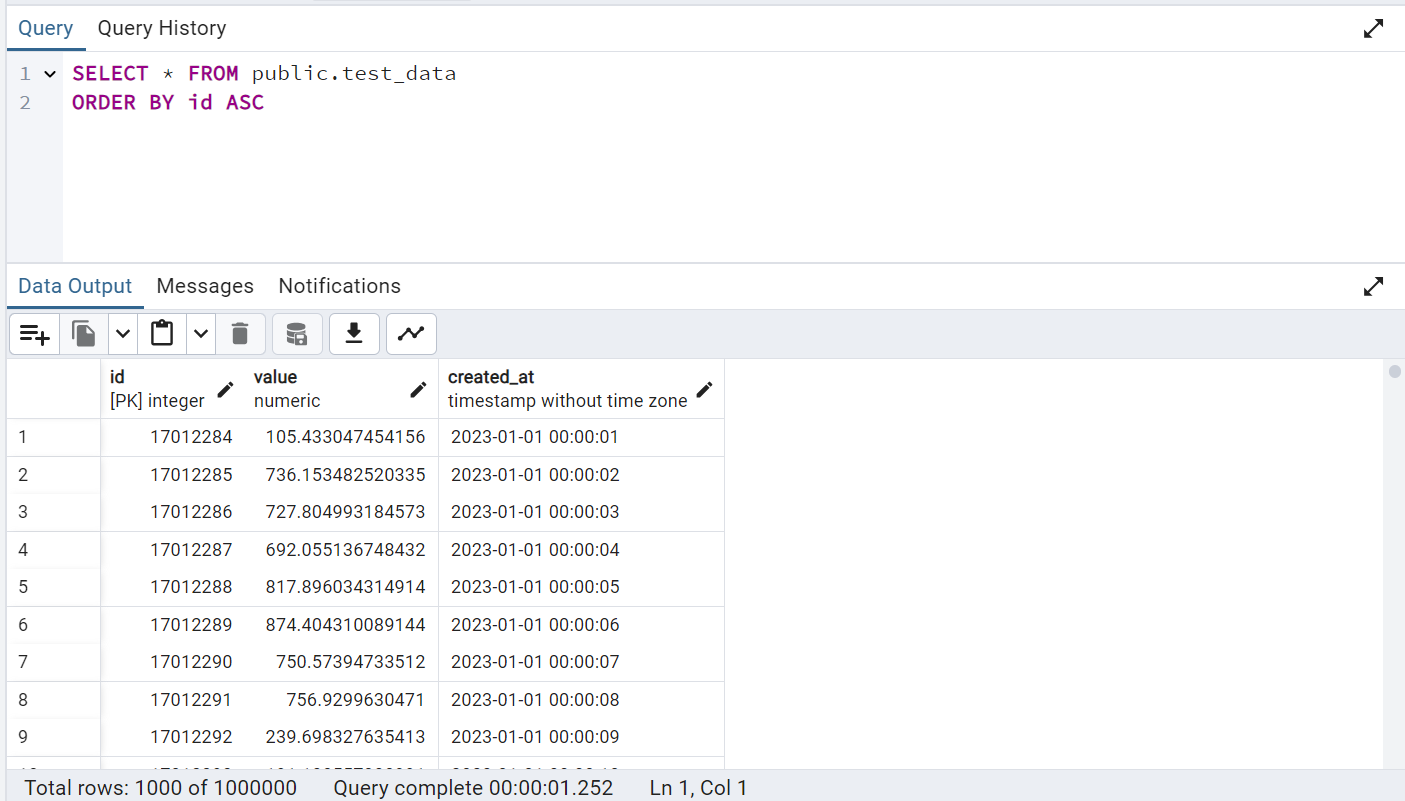
RANDOM() \* 1000, -- Випадкове число від 0 до 1000

'2023-01-01 00:00:00'::timestamp + INTERVAL '1 second' \* gs

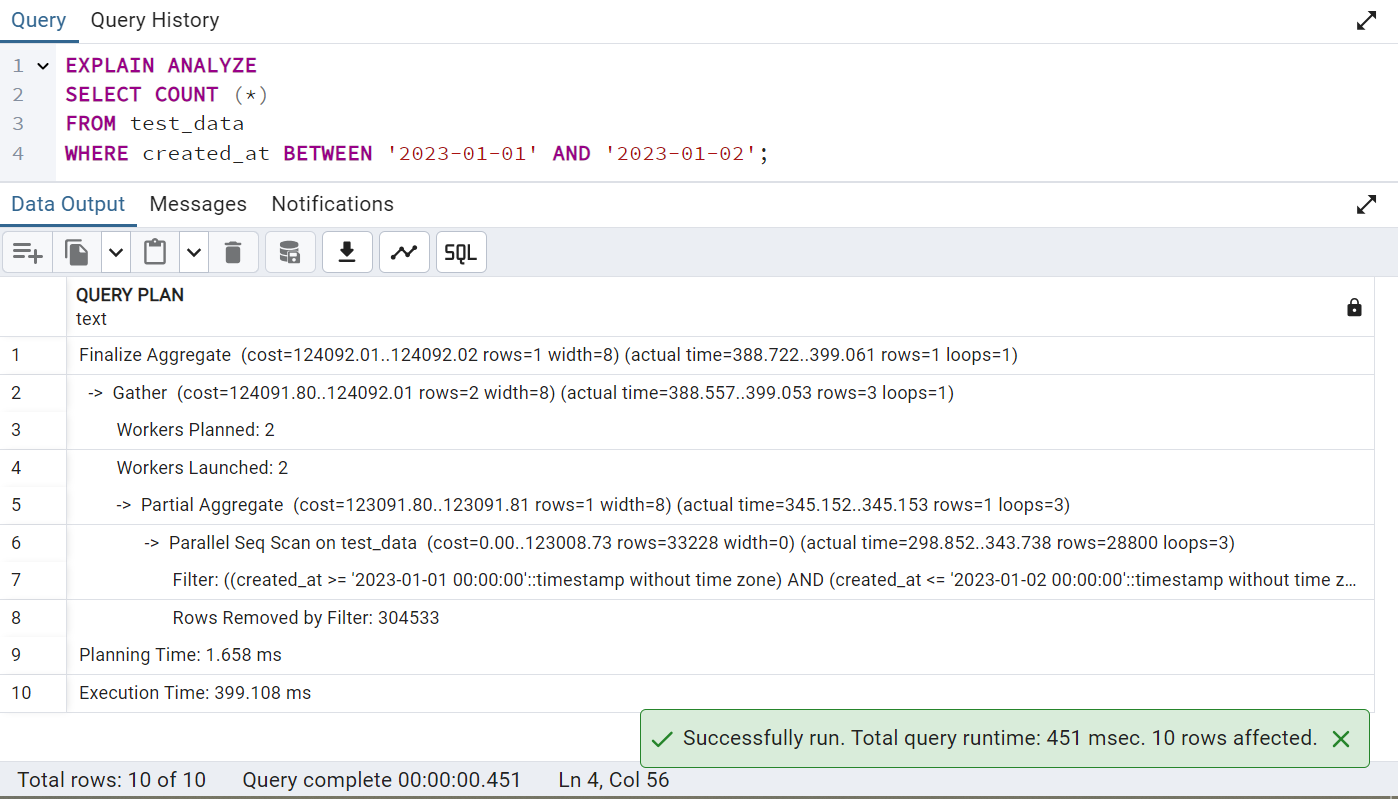
FROM generate\_series(1, 1000000) AS gs;



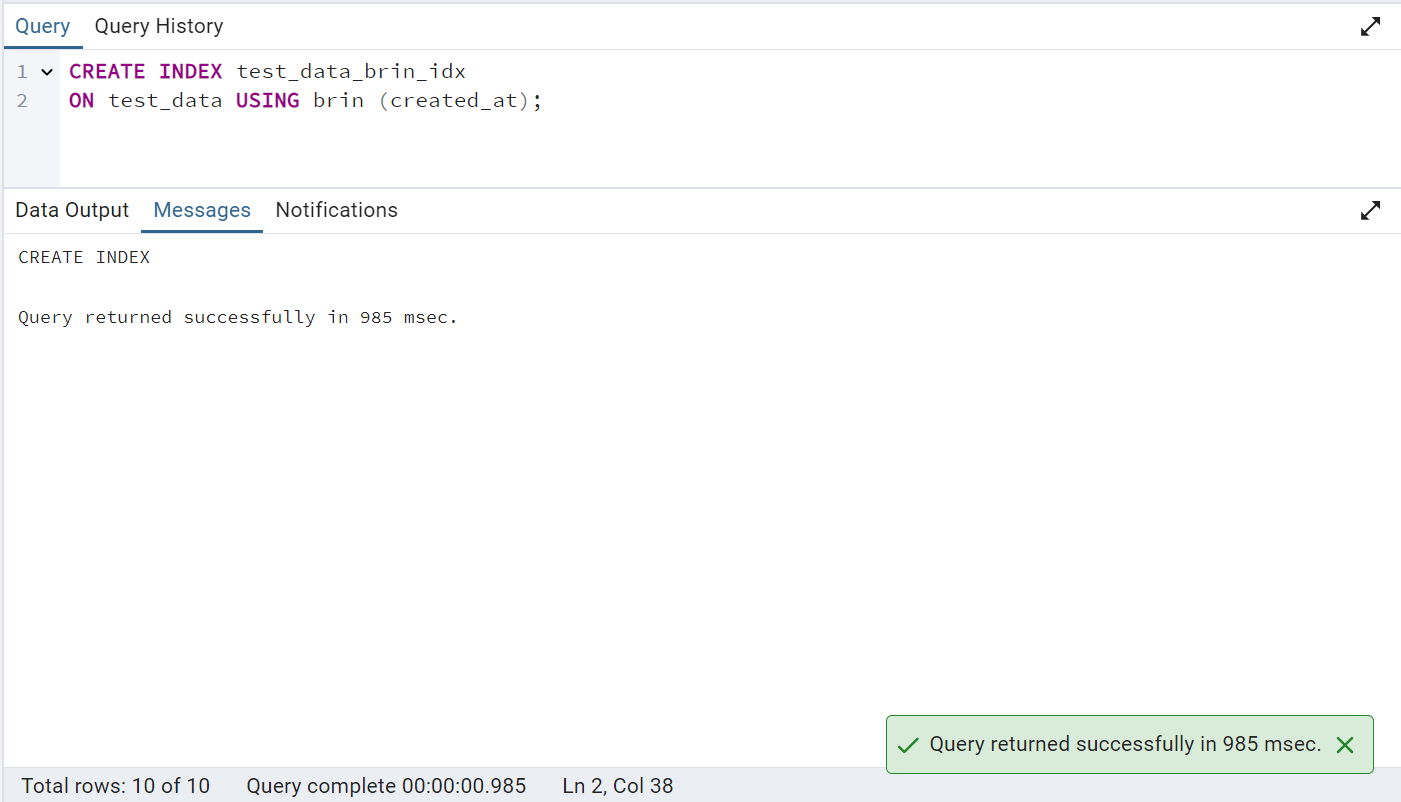
Вигляд з таблиці з впорядкованим create\_at:

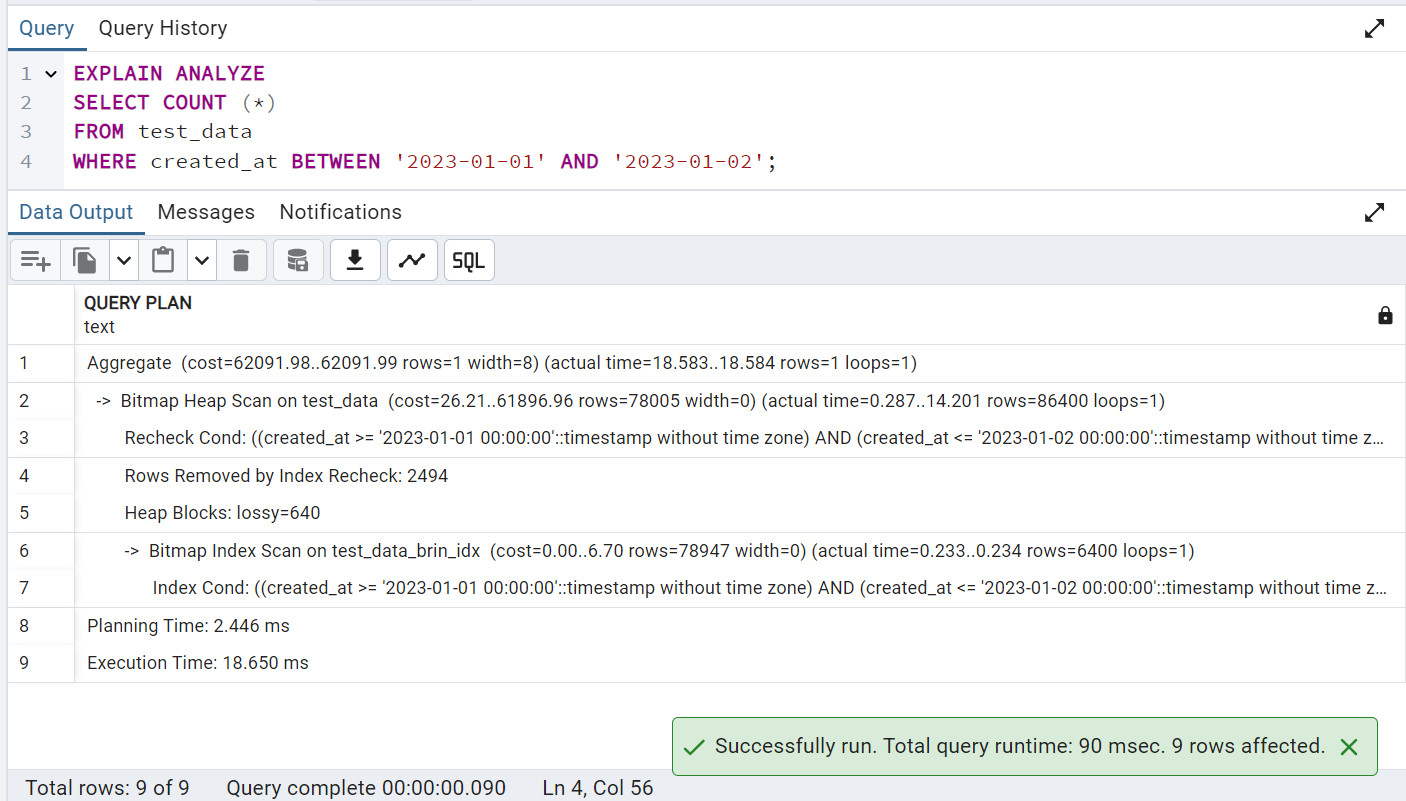


Час виконання без індексу:



Час виконання з використанням індексу:





Індекс BRIN (Block Range INdex) — це структура даних, яка використовується у базах даних, зокрема в PostgreSQL, для оптимізації пошуку в дуже великих таблицях. BRIN індекси є ефективним рішенням для ситуацій, коли дані мають природний порядок або коли виконання точних індексів (наприклад, B-дерев) надто ресурсозатратне.

Основні властивості BRIN-індексу:

1. **Економія місця**: BRIN-індекси займають набагато менше простору, ніж інші типи індексів (наприклад, B-tree). Це досягається завдяки групуванню даних у **блоки** та збереженню лише мінімального і максимального значення для кожного діапазону блоків.
2. **Підходить для впорядкованих даних**: BRIN-індекс є ефективним, коли дані у таблиці мають **природний порядок** або впорядковані за певним критерієм (наприклад, дати, послідовні ідентифікатори).
3. **Діапазон блоків**: Таблиця ділиться на діапазони сторінок (блоків), і для кожного діапазону зберігається мінімальне та максимальне значення індексованого стовпця. При виконанні запиту система швидко визначає, які блоки можуть містити потрібні дані.
4. **Низька вартість створення**: Створення BRIN-індексу є швидким та маловитратним у порівнянні з B-tree або іншими індексами, оскільки він зберігає агреговані значення лише для діапазонів сторінок.
5. **Обмежена точність**: BRIN не гарантує точного попадання в запис — він лише скорочує кількість блоків, що потрібно перевірити, при виконанні запиту.

BRIN-індекси є ефективним інструментом для великих таблиць з впорядкованими даними. Вони забезпечують значну економію ресурсів порівняно з B-tree індексами, особливо коли повна точність не є критично важливою.

**Завдання №3**

**Before insert**

Тригер автоматично перевіряє значення поля price для кожного нового запису в таблиці Food перед його додаванням. Це дозволяє гарантувати, що ціна продукту завжди більша за 0, запобігаючи внесенню некоректних даних у таблицю. Завдяки цьому забезпечується дотримання бізнес-логіки та підтримка цілісності даних без необхідності ручних перевірок або дублювання перевірок у зовнішніх додатках.

Код тригеру:  
CREATE OR REPLACE FUNCTION check\_food\_price()

RETURNS TRIGGER AS $$

BEGIN

IF NEW.price <= 0 THEN

RAISE EXCEPTION 'Price for food must be greater than 0. Provided: %', NEW.price;

END IF;

RETURN NEW;

END;

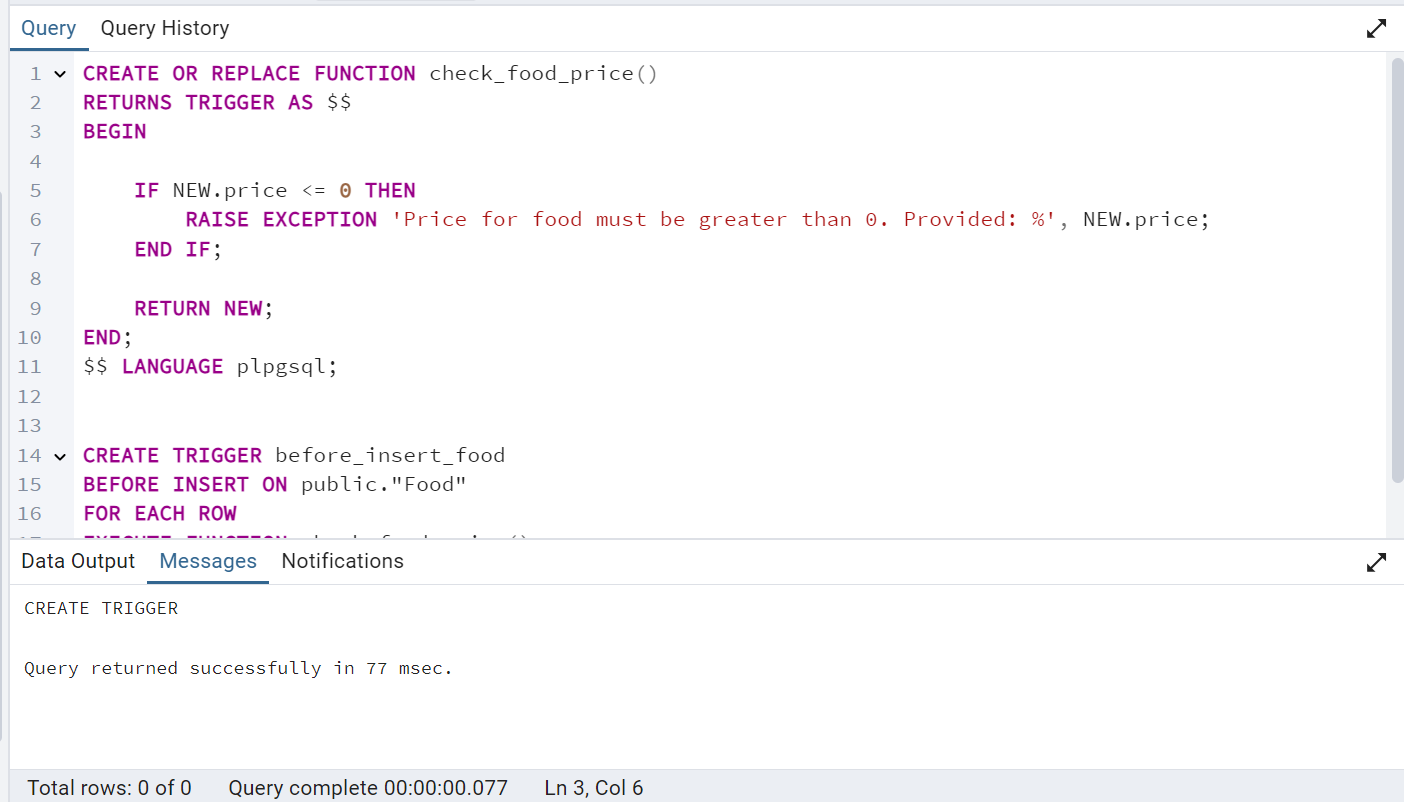
$$ LANGUAGE plpgsql;

CREATE TRIGGER before\_insert\_food

BEFORE INSERT ON public."Food"

FOR EACH ROW

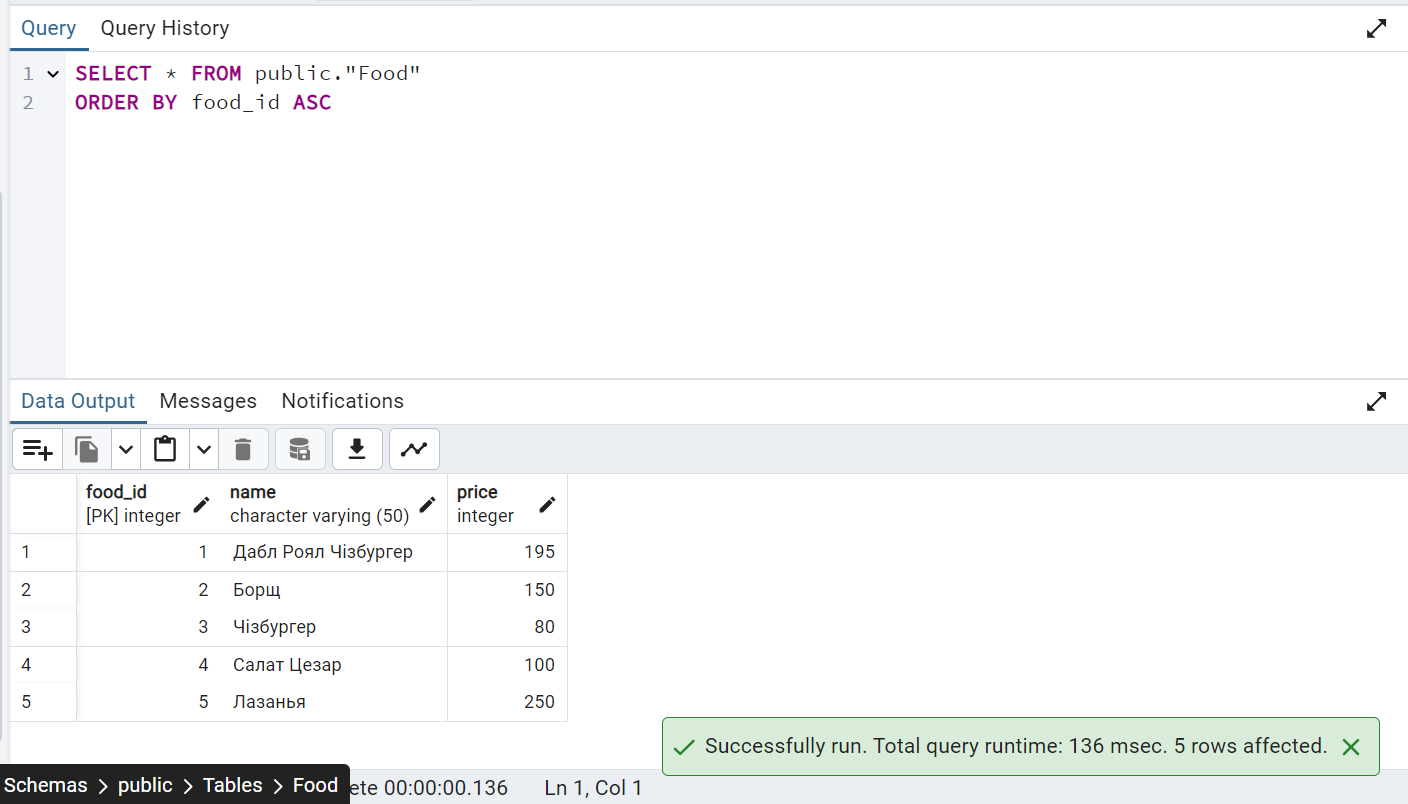
EXECUTE FUNCTION check\_food\_price();



Тестування роботи тригеру:

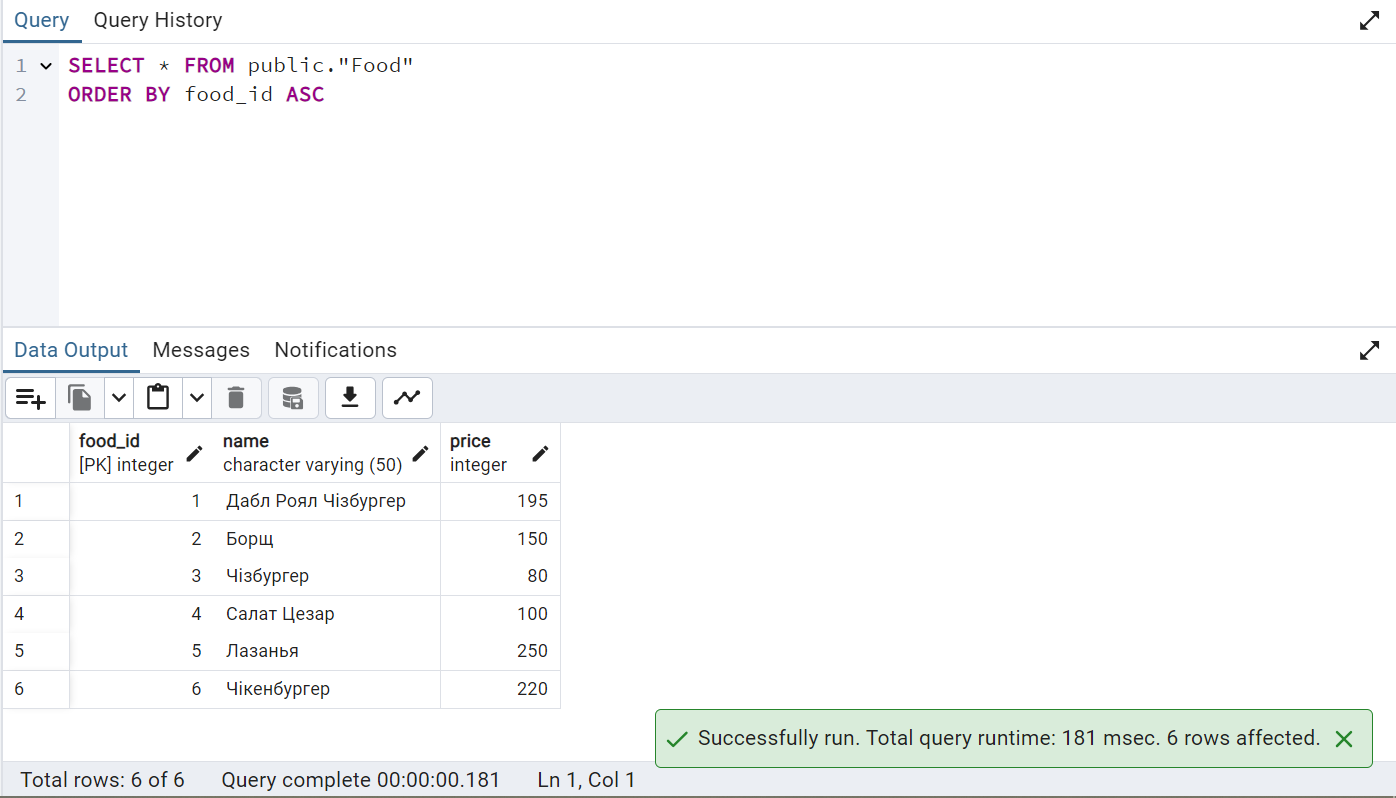


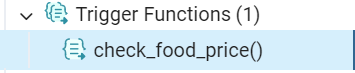
Запис не додався, що свідчить про правильність роботи тригеру.



Але, якщо ціна буде більше 0,то запис внесеться в таблицю:







**Before delete**

Тригер автоматично видаляє всі записи, пов’язані із замовленням у таблиці Food\_Order, перед видаленням відповідного запису з таблиці Order. Це дозволяє підтримувати цілісність даних між таблицями, запобігаючи залишенню "осиротілих" записів у таблиці Food\_Order. Крім того, тригер забезпечує обробку можливих помилок при видаленні, залишаючи відповідне повідомлення про проблему, що допомагає з діагностикою потенційних помилок.

Код тригеру:

CREATE OR REPLACE FUNCTION delete\_related\_food\_orders()

RETURNS TRIGGER AS $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN

SELECT fo\_id FROM public."Food\_Order" WHERE order\_id = OLD.order\_id

LOOP

BEGIN

DELETE FROM public."Food\_Order" WHERE fo\_id = rec.fo\_id;

EXCEPTION WHEN OTHERS THEN

RAISE WARNING 'Failed to delete Food\_Order with ID: %. Error: %', rec.fo\_id, SQLERRM;

END;

END LOOP;

RAISE NOTICE 'Order with ID % and its related records have been successfully deleted.', OLD.order\_id;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

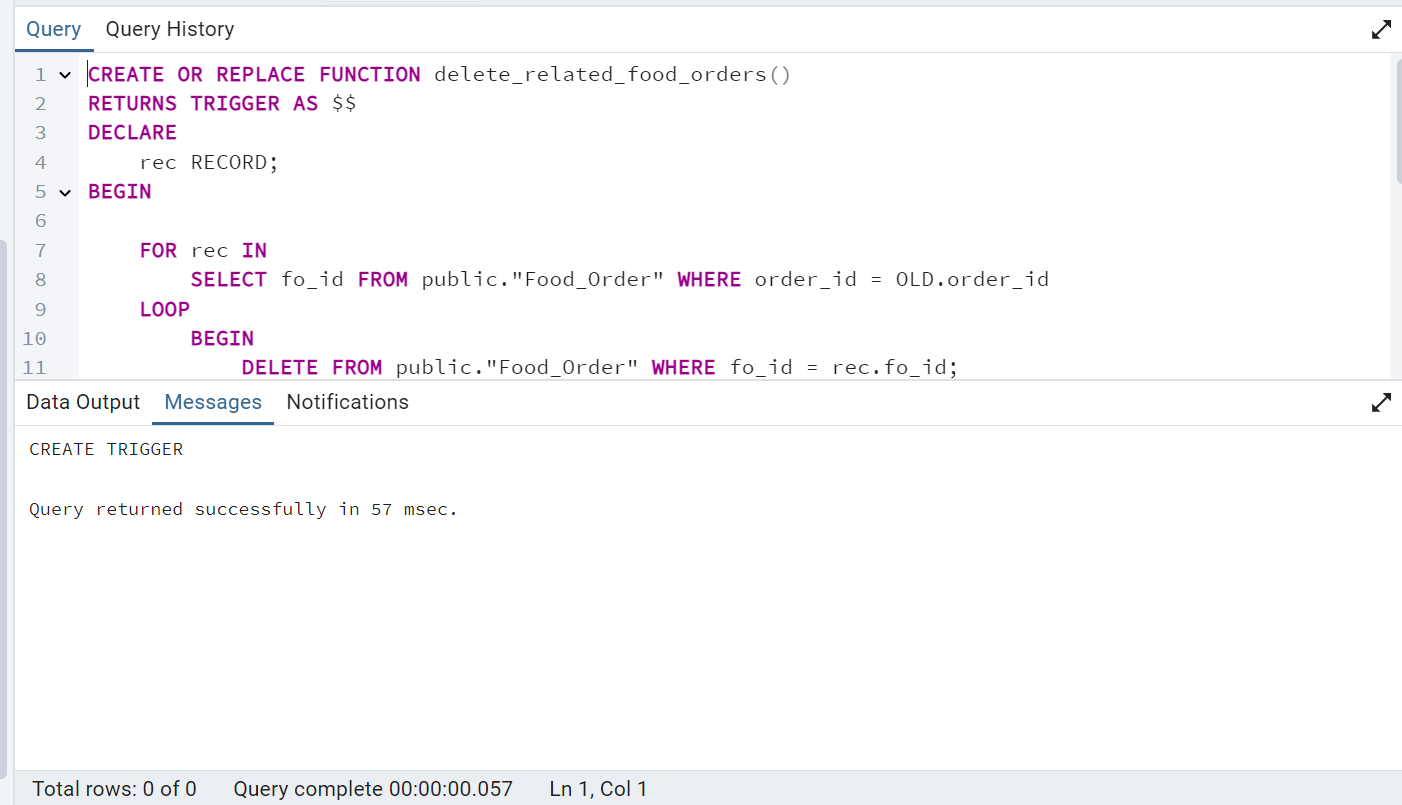
CREATE TRIGGER before\_delete\_order

BEFORE DELETE ON public."Order"

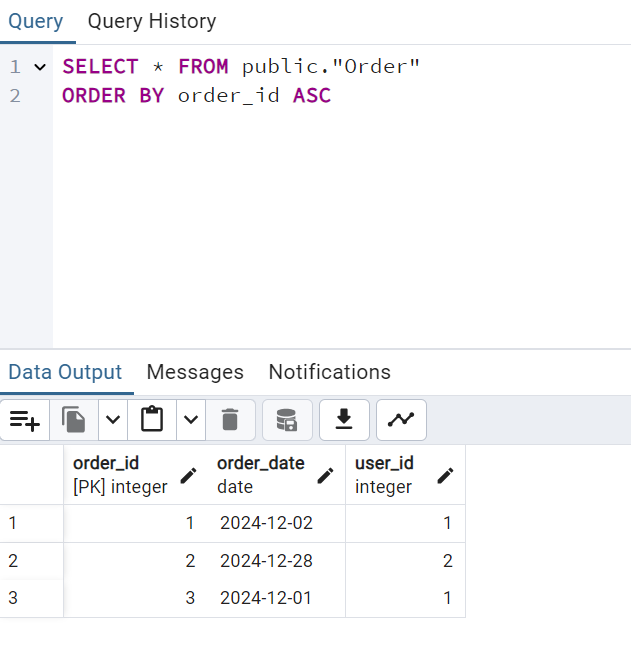
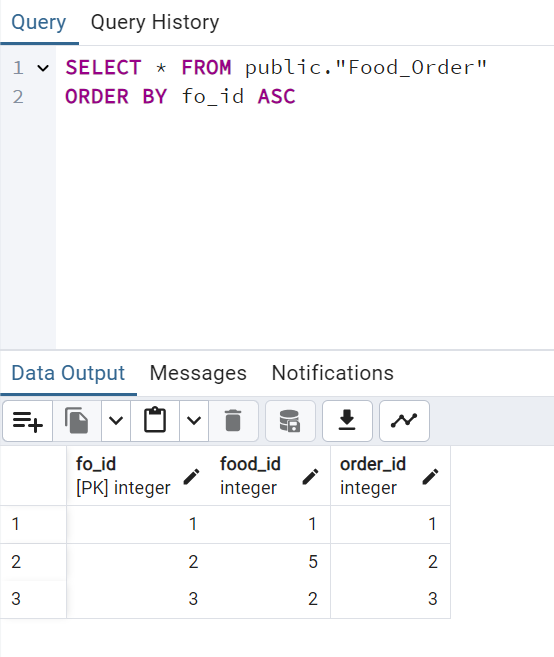
FOR EACH ROW

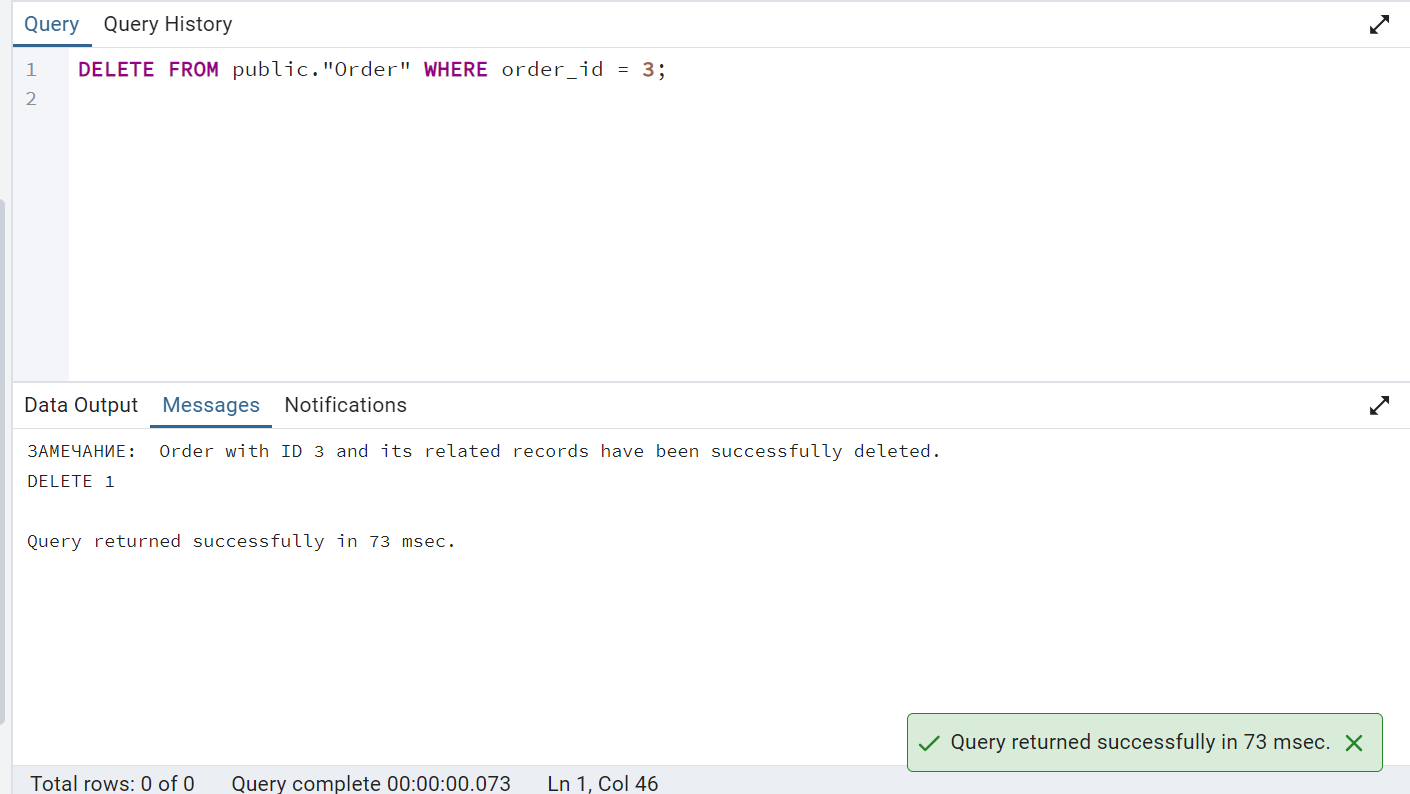
EXECUTE FUNCTION delete\_related\_food\_orders();

Тестування роботи тригеру:

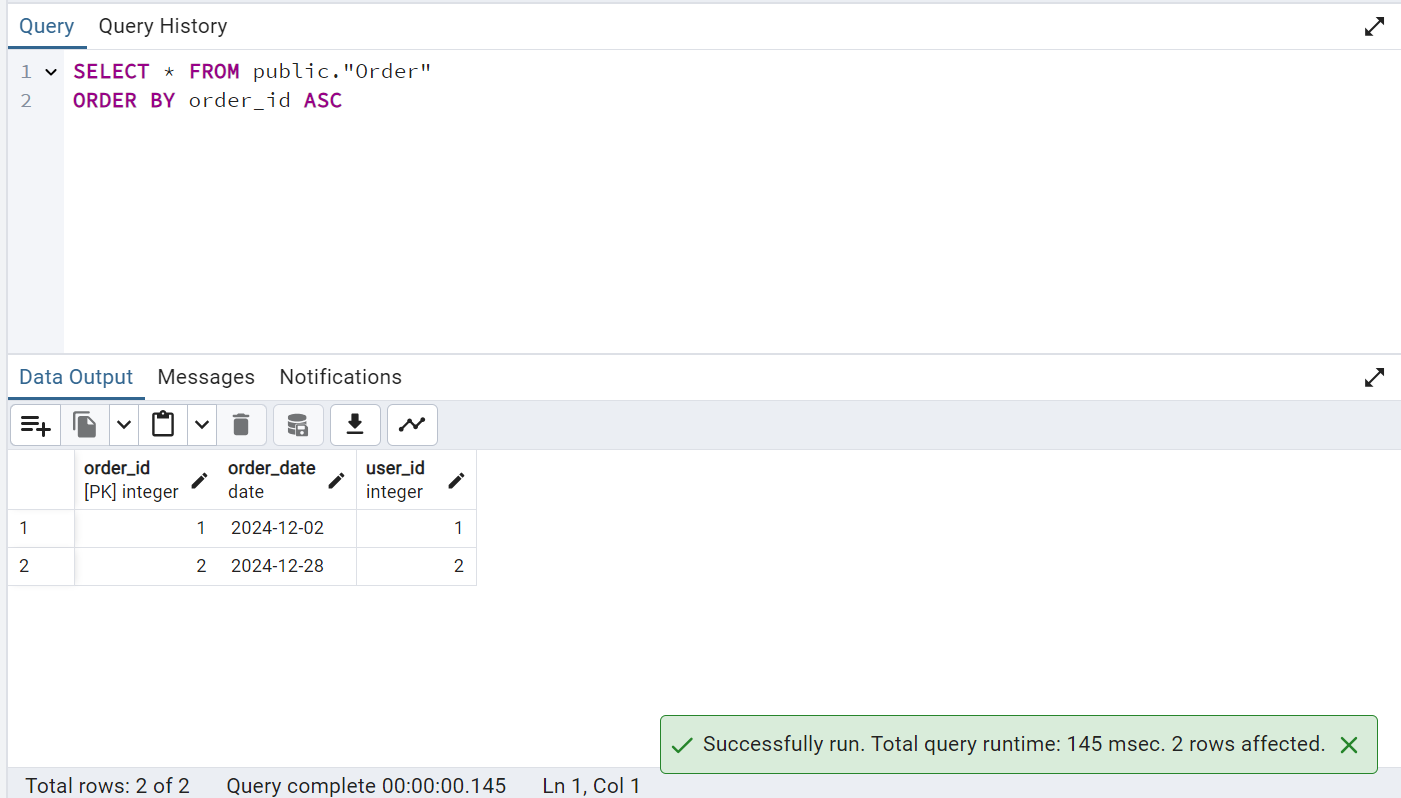
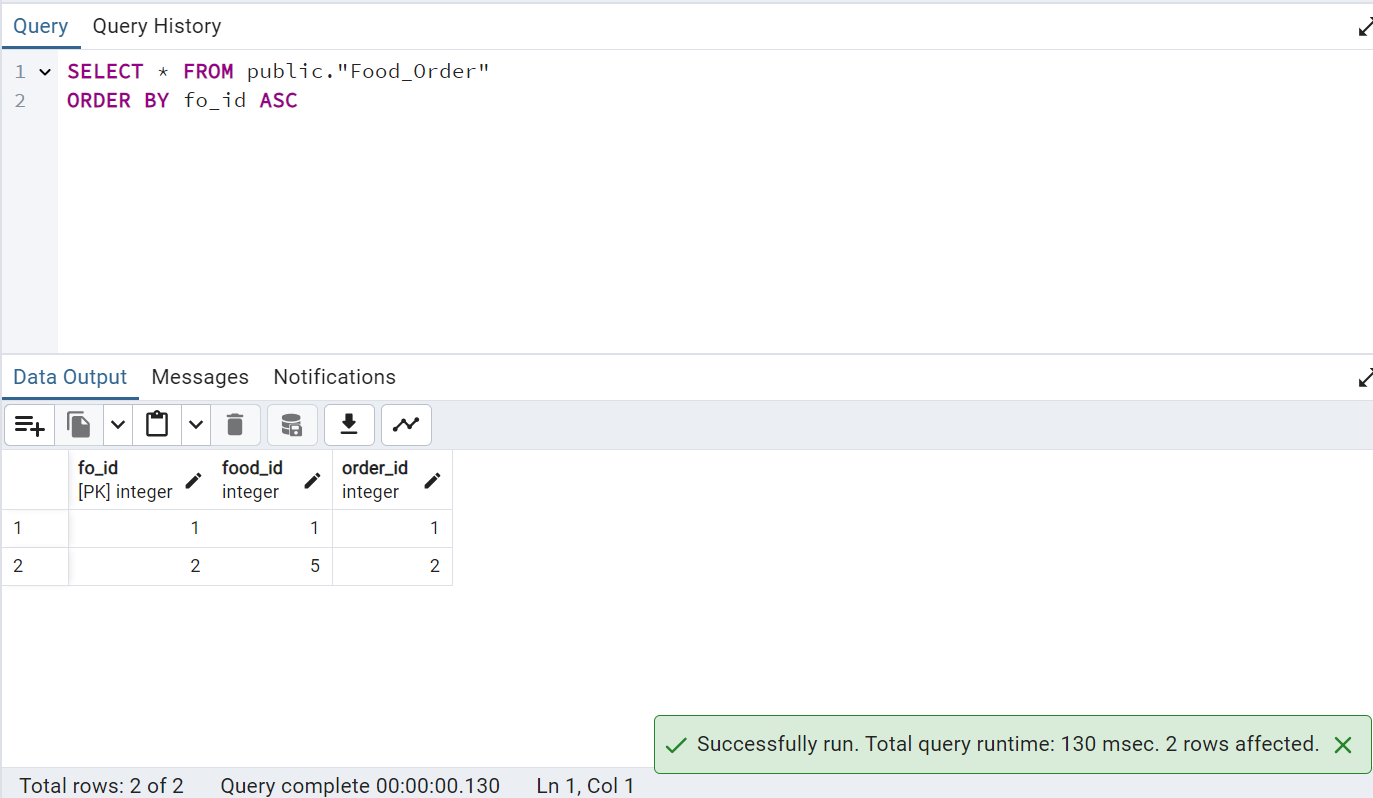


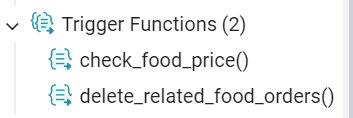
Додаємо нові записи до таблиць:  



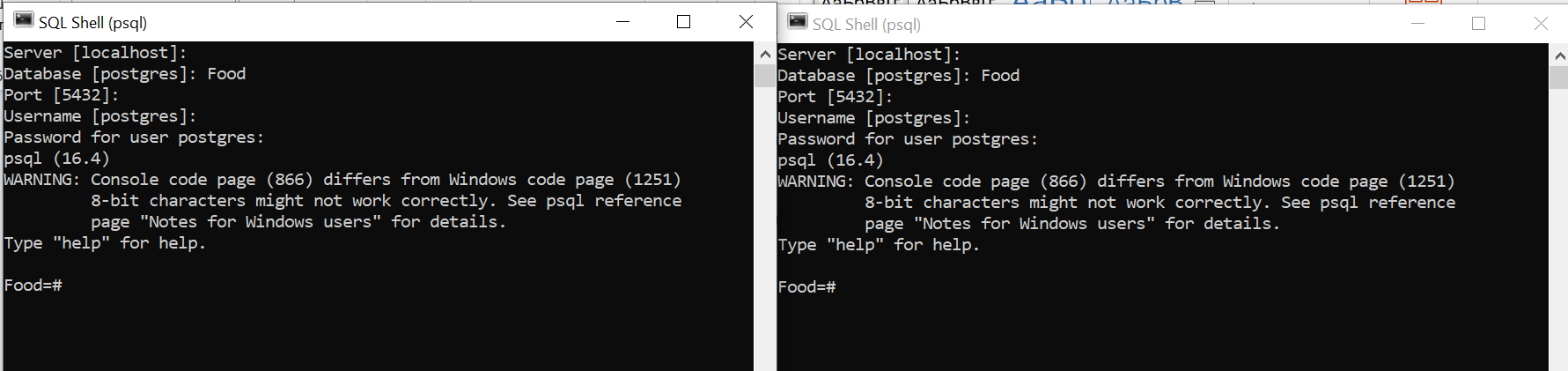
Вивелося повідомлення, тому тригер працює справно.





**Завдання №4**

Спочатку необхідно під’єднатись до БД:



**READ COMMITTED**

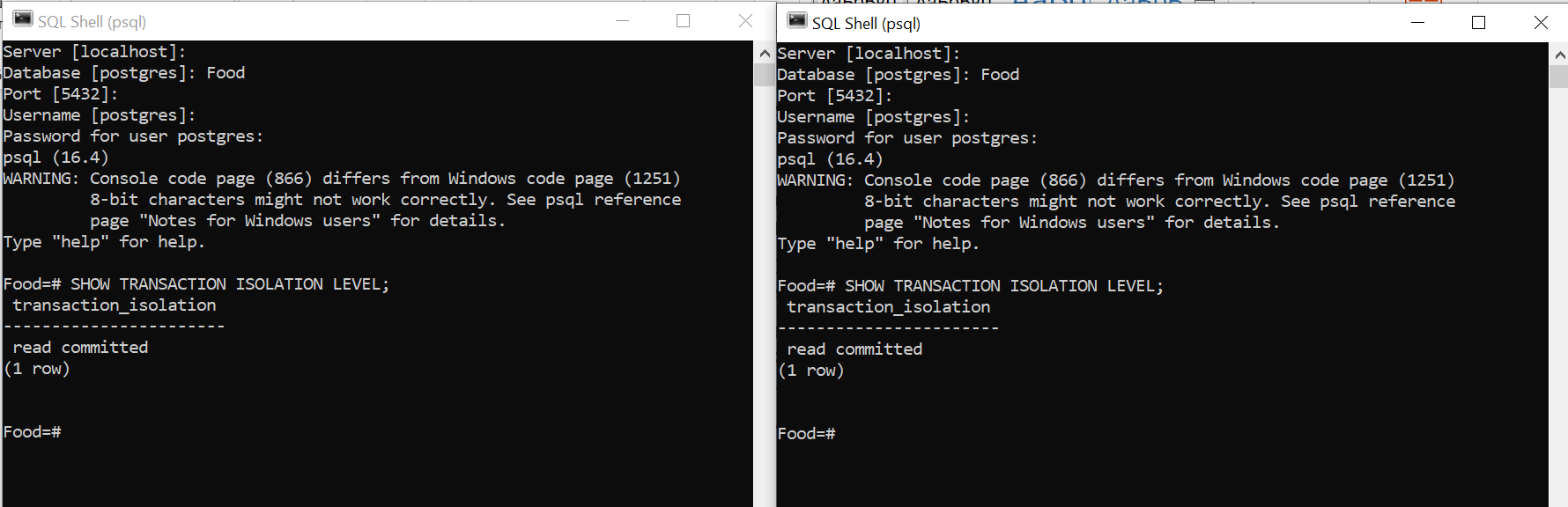
**READ COMMITTED** — це рівень ізоляції транзакцій у системах управління базами даних (СУБД), який пропонує баланс між продуктивністю та ізоляцією. У порівнянні з рівнем SERIALIZABLE, він забезпечує меншу ізоляцію, але підвищує ефективність і знижує ймовірність блокування ресурсів.

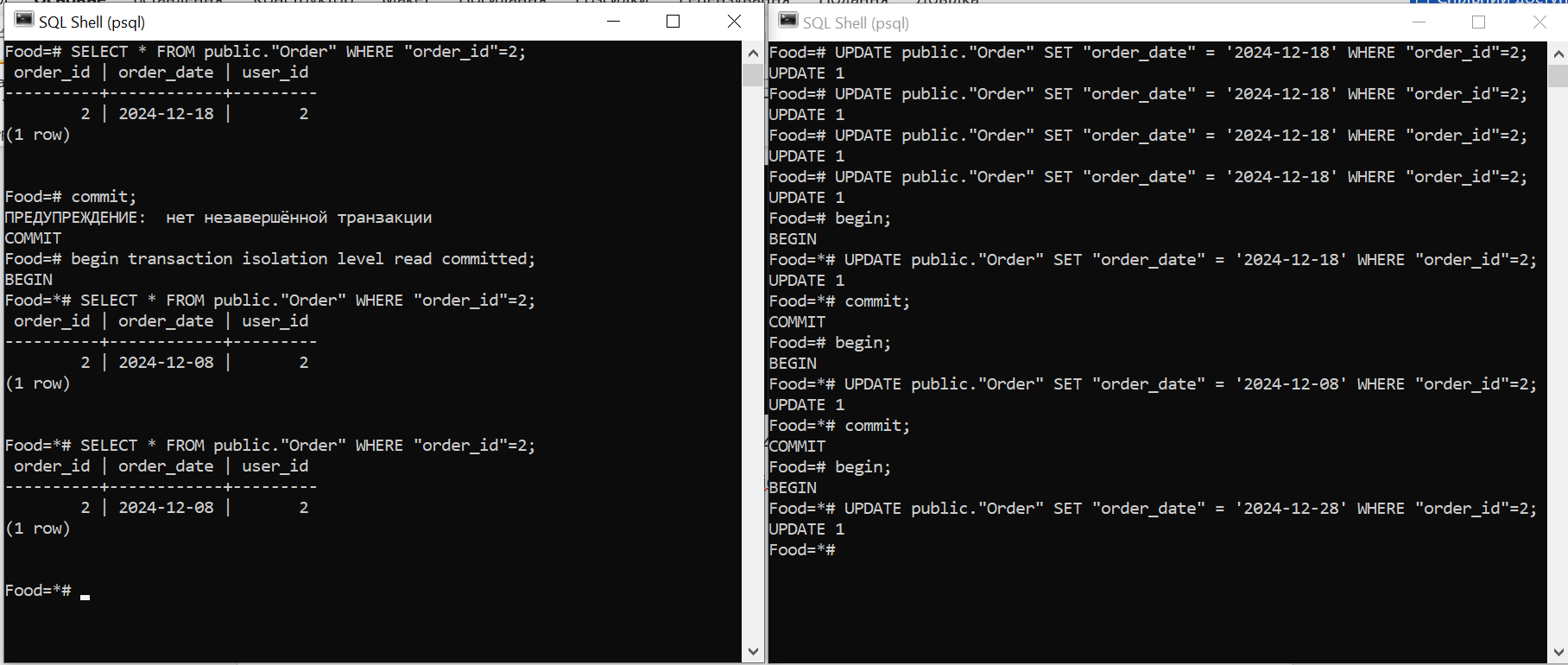
**Основні особливості рівня READ COMMITTED:**

1. **Блокування при записі, але не при читанні**: Транзакція блокує дані для запису під час їх зміни, однак інші транзакції можуть читати ці дані, навіть якщо вони ще не зафіксовані.
2. **Відсутність "брудних читань"**: На рівні READ COMMITTED транзакція не може читати незбережені зміни інших транзакцій. Це забезпечує узгодженість даних під час читання.
3. **Допущення "неповторюваних читань" і "фантомних вставок"**:
   * "Неповторювані читання" означають, що один і той самий рядок може бути прочитаний з різними значеннями, якщо інша транзакція змінила його після початку поточної транзакції.
   * "Фантомні вставки" або "фантомні оновлення" можливі, коли в поточну транзакцію додаються або змінюються нові рядки.
4. **Читання лише зафіксованих даних**: Транзакція може читати тільки ті зміни, які були зафіксовані іншими транзакціями. Незбережені зміни недоступні для читання.

**READ COMMITTED** є популярним рівнем ізоляції у багатьох СУБД завдяки своєму компромісу між ізоляцією транзакцій і продуктивністю. Проте, при розробці додатків слід враховувати особливості цього рівня, особливо якщо необхідно запобігти "брудним читанням".

Рівень ізоляції за замовчуванням у postgres — це READ COMMITTED. Щоб підтвердити, виконаємо цю команду в psql:





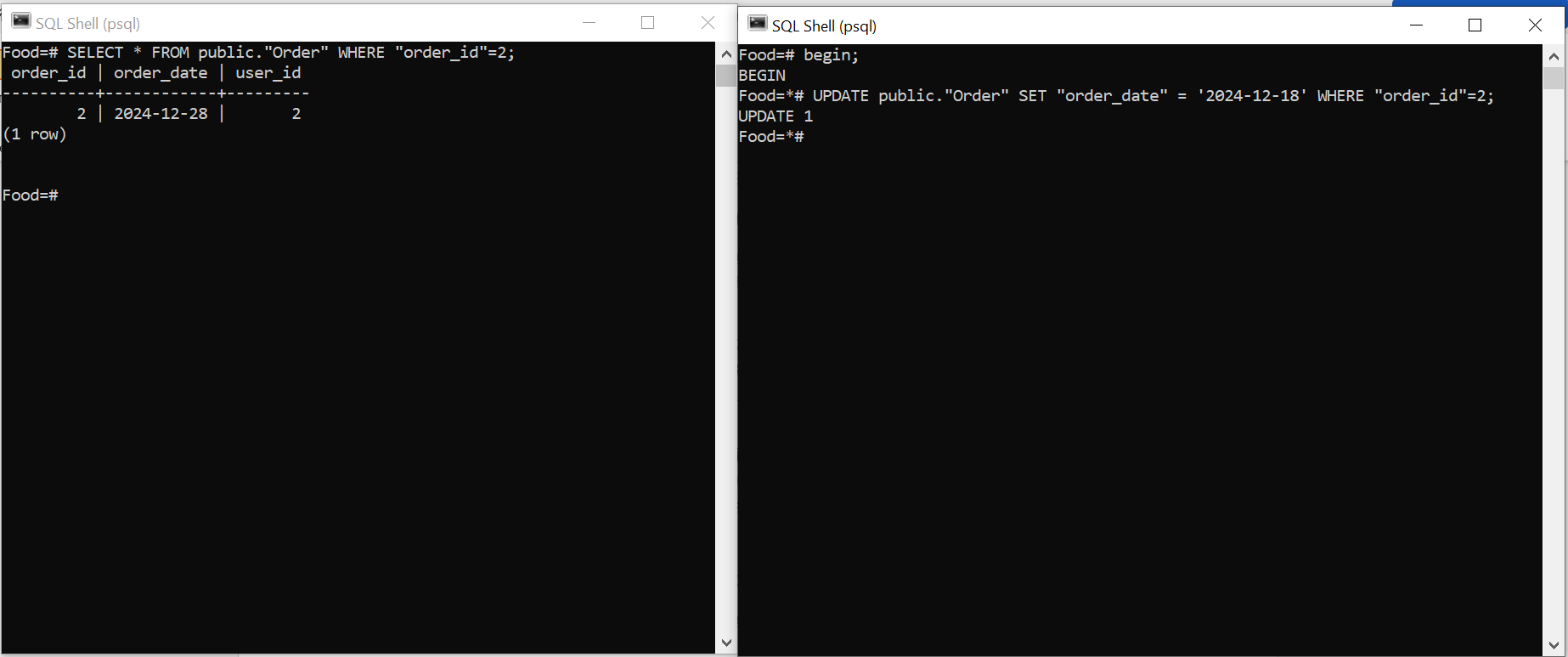
**REPEATABLE READ**

**REPEATABLE READ** — це рівень ізоляції транзакцій, що забезпечує високий рівень консистентності даних під час читання, але допускає можливість *"фантомних оновлень"*.

**Основні особливості рівня REPEATABLE READ:**

1. **Стабільність даних протягом життя транзакції**: Усі дані, доступні для транзакції, залишаються консистентними протягом усього її виконання.
2. **Блокування при читанні та записі**:
   * Транзакція блокує рядки даних, які вона читає, а також ті, що використовуються у виразах WHERE у запитах SELECT.
   * Це дозволяє уникнути *"брудних читань"* та *"неповторюваних читань"*.
3. **Запобігання неповторним читанням**: Якщо транзакція прочитала певний рядок, значення цього рядка не зміниться до завершення транзакції, навіть якщо інші транзакції намагаються його модифікувати.
4. **Фантомні оновлення**: Попри блокування вже прочитаних рядків, нові рядки можуть з’являтися в результаті виконання SELECT-запиту протягом часу життя транзакції. Це означає, що при повторному виконанні того ж самого запиту транзакція може побачити нові дані.

**REPEATABLE READ** є ефективним компромісом між продуктивністю та високим рівнем ізоляції. Цей рівень підходить для ситуацій, де критично важливо уникнути *"брудних читань"* і *"неповторюваних читань"*, але можливість *"фантомних оновлень"* є прийнятною.



**SERIALIZABLE**

**SERIALIZABLE** — це найвищий рівень ізоляції транзакцій у системах баз даних. Він забезпечує повну ізоляцію між транзакціями, не дозволяючи жодній транзакції бачити зміни, внесені іншими транзакціями, поки вони не завершаться. Цей рівень усуває всі можливі аномалії, включно з "брудним читанням", "неповторюваним читанням", "фантомними читаннями", "брудними оновленнями" та "фантомними оновленнями".

**Основні характеристики рівня SERIALIZABLE:**

1. **Повне блокування читання і запису**: Транзакція блокує всі рядки, з якими вона взаємодіє, як для читання, так і для запису. Це забезпечує повну ізоляцію та запобігає конфліктам між транзакціями.
2. **Запобігання фантомним читанням і вставкам**: Транзакції на рівні SERIALIZABLE не бачать нових рядків, вставлених іншими транзакціями після їхнього початкового читання. Це також унеможливлює додавання нових рядків у той самий набір даних під час виконання транзакції.
3. **Уникнення фантомних оновлень**: SERIALIZABLE гарантує, що жодна транзакція не зможе оновити дані, якщо вони були змінені іншою транзакцією після початкового читання.
4. **Стабільність результатів читання**: Результати запитів SELECT залишаються незмінними в межах однієї транзакції, забезпечуючи стабільний порядок даних.

**SERIALIZABLE** надає максимальний рівень ізоляції, але це може призвести до значного блокування ресурсів та зниження продуктивності системи. Тому цей рівень слід використовувати лише тоді, коли повна ізоляція є критично важливою, і ви готові пожертвувати частиною продуктивності задля забезпечення цілісності даних.

