

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних систем**

**Лабораторна робота №3**

з дисципліни **Бази даних і засоби управління**

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

Виконав:

студент ІII курсу

групи КВ-91

Сторожук К. В.

Перевірив:

Павловський В. І.

Київ – 2021

**Лабораторна робота № 3.**

**Засоби оптимізації роботи СУБД PostgreSQL**

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

*Завдання* роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 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 22**

Індекси за варіантом: Hash, BRIN

Тригер: after delete, insert

**Логічна модель предметної області «ЄДЕБО»**

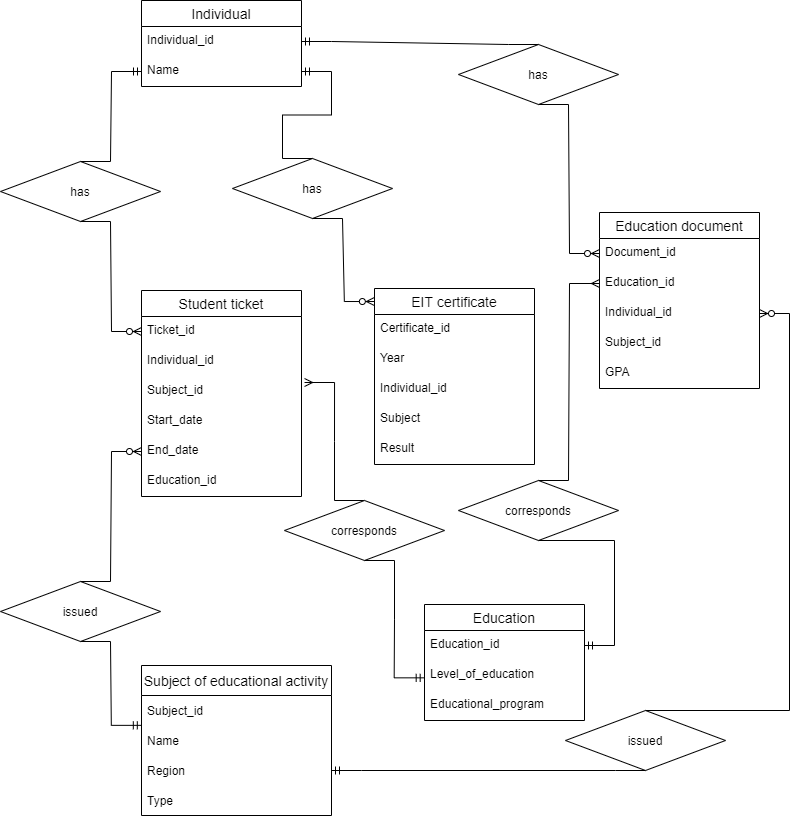


Рисунок 1 – Логічна модель предметної області «ЄДЕБО»

**Середовище та компоненти розробки**

Середовище розробки бази даних - PostgreSQL

Середовище розробки програми – Visual Studio. Мова програмування Python

Для зв’язку програми з базою даних Postgresql необхідно було встановити модуль sqlalchemy

**Фрагменти програм внесення, редагування та вилучення даних у базі даних засобами ORM**

1. Фрагмент програми внесення даних

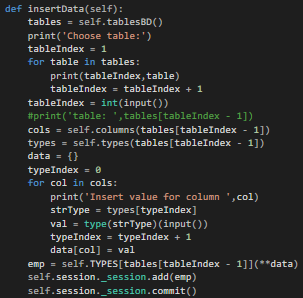


Рисунок 2 – Фрагмент програми внесення даних

1. Фрагмент програми видалення даних

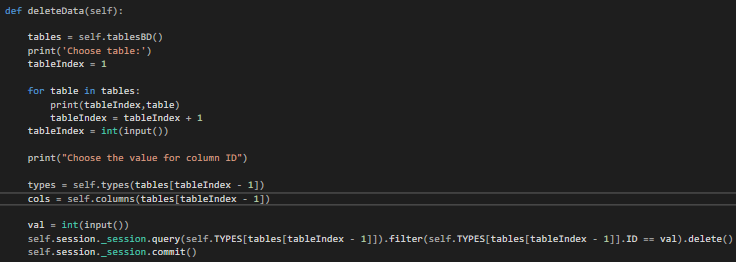


Рисунок 3 – Фрагмент програми видалення даних

1. Фрагмент програми редагування даних

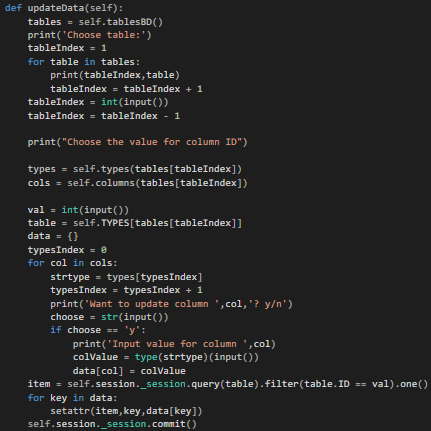


Рисунок 4 – Фрагмент програми редагування даних

1. Фрагмент програми відображення даних

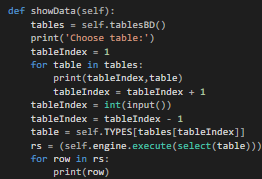


Рисунок 5 – Фрагмент програми відображення даних

**Результати виконання базових операцій з базою даних засобами ORM**

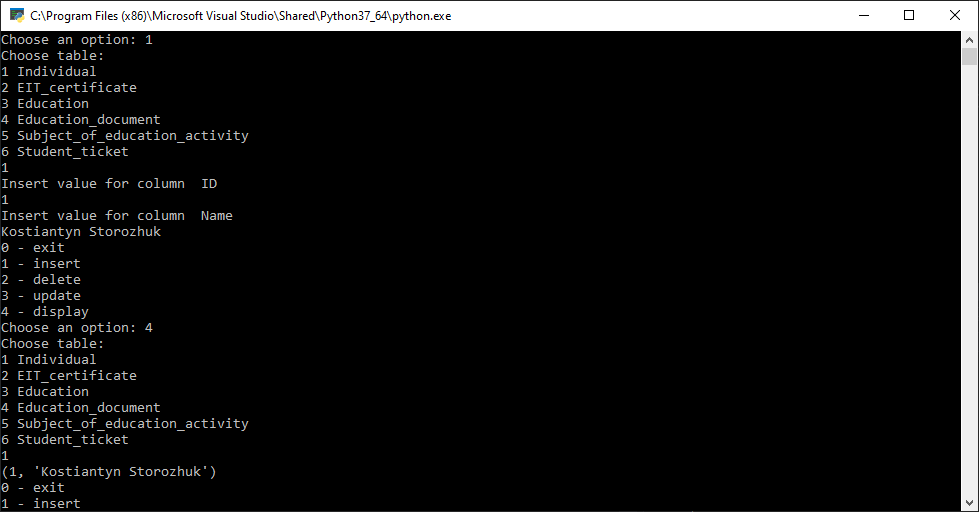


Рисунок 6 – Демонстрація виконання вставки в таблицю



Рисунок 7 – Демонстрація виконання редагування в таблиці

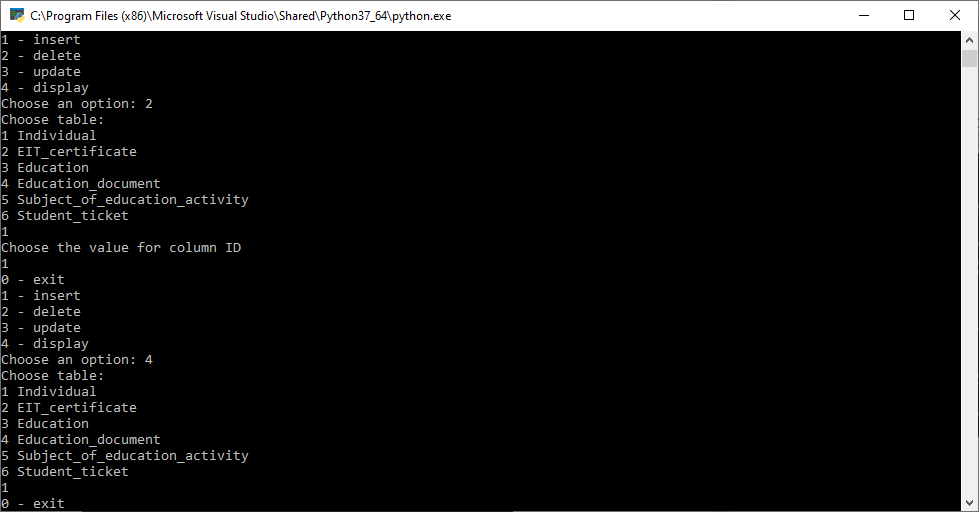


Рисунок 8 – Демонстрація виконання видалення в таблицю

**Робота з індексами**

**Hash**

Hash індекси використовуются для придшвидшення пошуку. За допомогою хеш-функції для відповідного типу данних ставиться в відповідність значення типу integer з певного діапазону значень. Це дозволяє швидко знаходити пари хеш-код – TID, оскільки хеш-коди зберігаються впорядковано та можуть бути швидко порівняні один з одним.

Для тестування хеш індексу була створена таблиця з одним полем text name, яке буде використовуватись для дослідження хеш індексу. Тип text гарно підходить для Hash індексів, оскільки рядки за замовченням мають повільну функцію порівняння.

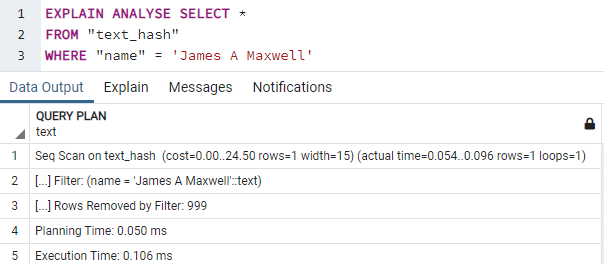


Рисунок 9 – Аналіз запиту 1 без індексу



Рисунок 10 – Аналіз запиту 1 з індексом

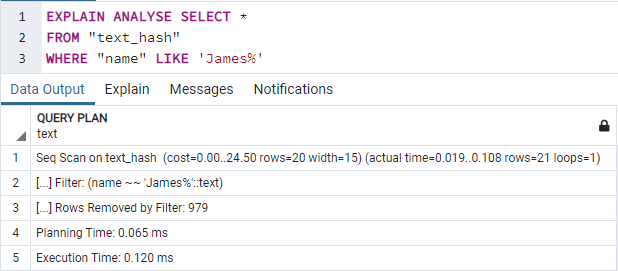


Рисунок 11 – Аналіз запиту 2 без індексу

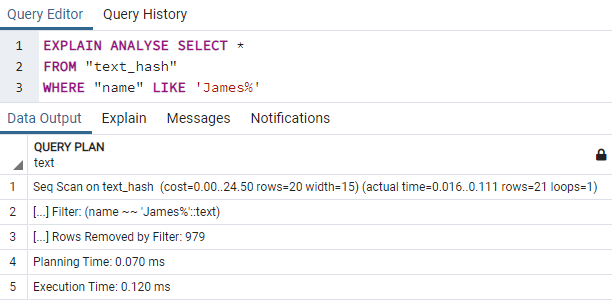


Рисунок 12 – Аналіз запиту 2 з індексом

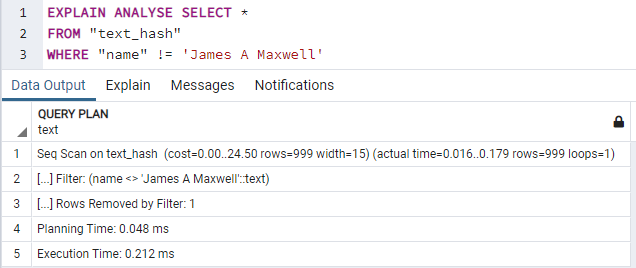


Рисунок 13 – Аналіз запиту 3 без індексу



Рисунок 14 – Аналіз запиту 3 з індексом

Пришвидшення можна помітити в 1 та 3 запитах. Це повязано з тим що в цих двох запитах за допомогою індекса ми зменшили кількість та час виконання порівнянь. В 2 запиті це не відбулось оскільки хеш-індекси можуть обробляти лише прості порівняння рівності.

**BRIN**

BRIN – це індекс блочного діапазону. Блок є базовою одиницею зберігання Postgres і за замовчуванням становить 8 КБ даних. BRIN вибирає діапазон блоків (за замовчуванням 128), зберігаючи розташування першого блоку в діапазоні, а також мінімальні та максимальні значення для всіх значень у цих блоках. Це дуже корисно для впорядкованих наборів даних, пропонуючи значну економію місця під індекс для подібної, а іноді й кращої швидкодії.

Для тестування BRIN індексу була створена таблиця з двума полями timestamp ts та text value. Дані для неє сгенеровані наступним запитом:

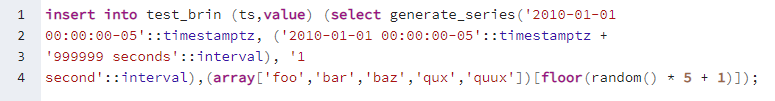


Рисунок 15 – Запит генерування даних для таблиці

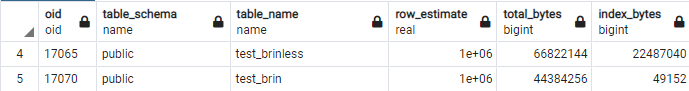


Рисунок 16 – Порівняння кількості байтів під індекс між brin та btree

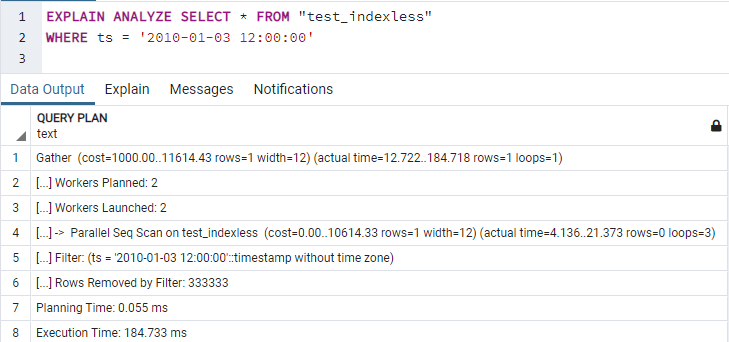


Рисунок 17 – Аналіз запиту 1 без індексу

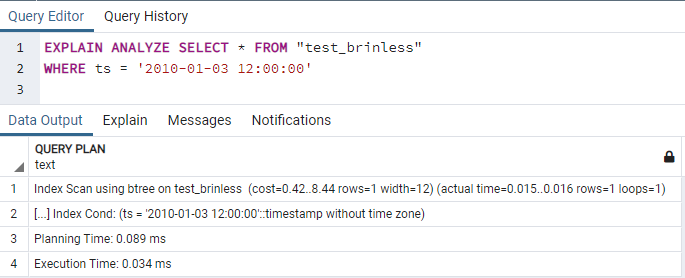


Рисунок 18 – Аналіз запиту 1 з індексом BTree

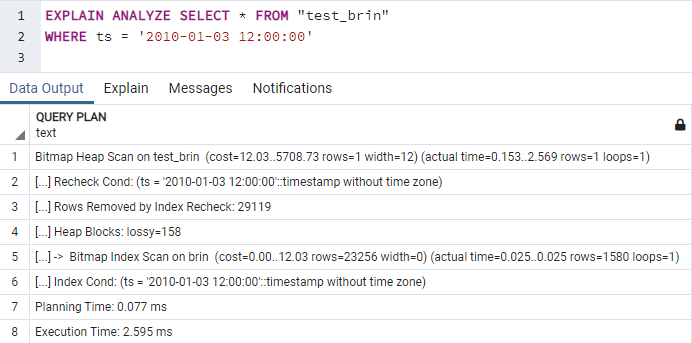


Рисунок 19 – Аналіз запиту 1 з індексом BRIN

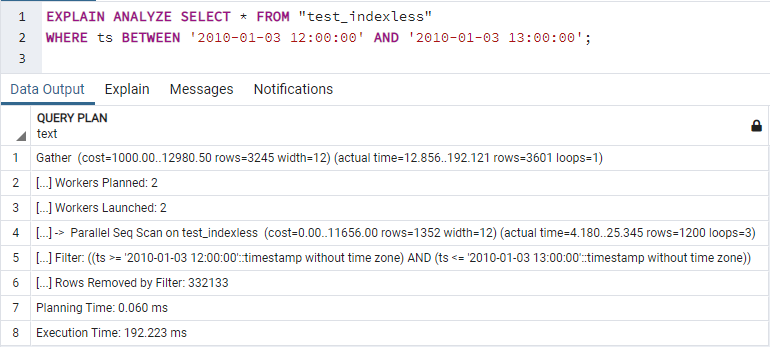


Рисунок 20 – Аналіз запиту 2 без індексу

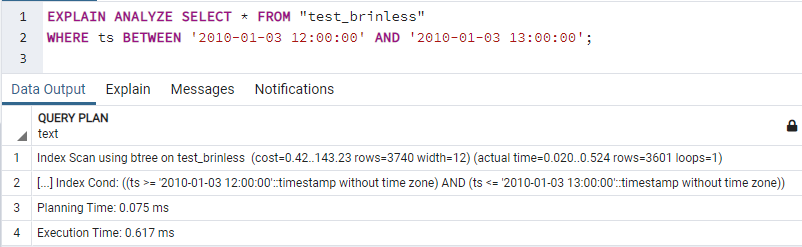


Рисунок 21 – Аналіз запиту 2 з індексом BTree

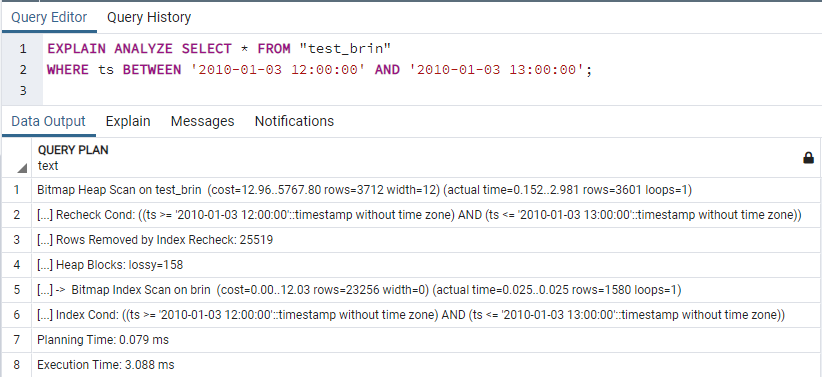


Рисунок 22 – Аналіз запиту 2 з індексом BRIN

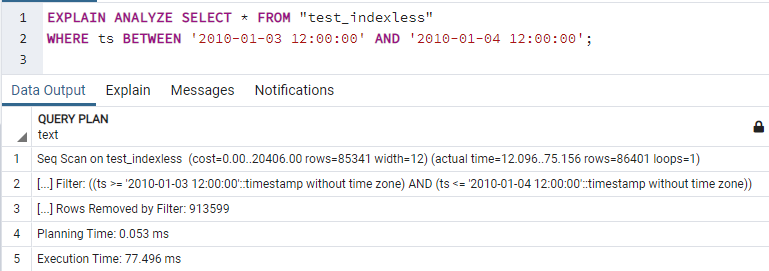


Рисунок 23 – Аналіз запиту 3 без індексу

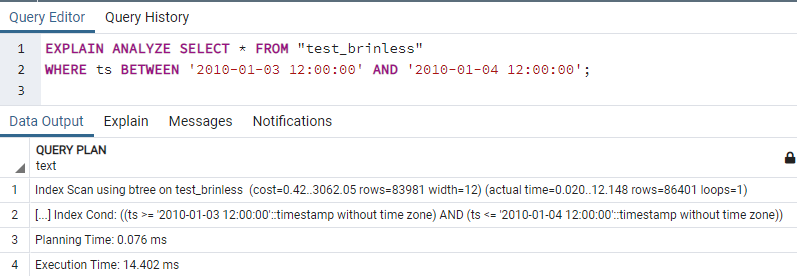


Рисунок 24 – Аналіз запиту 3 з індексом BTree

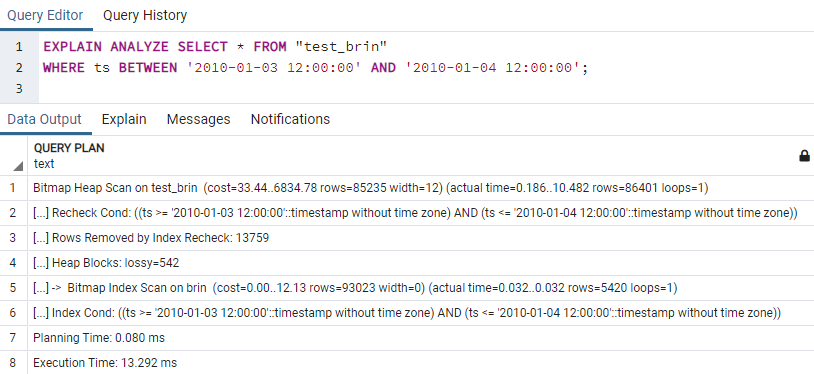


Рисунок 25 – Аналіз запиту 3 з індексом BRIN

Порівняно з індексом BTree, індекс BRIN займає дуже мало додаткового місця на диску.

При цьому чим більше записів охоплює запит, тим краще швидкодія BRIN у порівнянні з Btree.

Можна прийти до висновку до BRIN варто використовувати тоді, коли у нас є велика, впорядкована таблиця з запитами SELECT що охоплюють багато записів.

**Робота з тригерами**

Для тестування тригерів створимо таблицю складу та позик. Вони будуть містити відповідно записи про речі що містяться на складі, та речі що були взяті у позику.

Після вставки рядка в таблицю позик відповідна річ має бути видалена з складу, а після видалення рядка з таблиці позик відповідна річ повинна бути додана до складу.

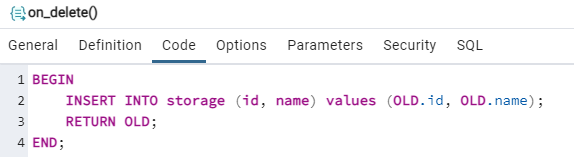


Рисунок 26 – Код тригеру що спрацьовує після видалення

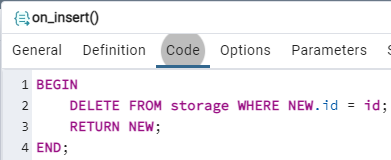


Рисунок 27 – Код тригеру що спрацьовує після додавання

Перевіримо роботу тригерів.

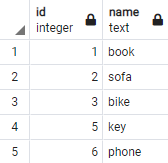


Рисунок 28 – Початковий стан таблиці складу

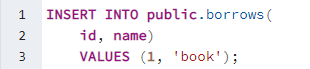


Рисунок 29 – запит для додавання в таблицю позик, що повинен викликати тригер.

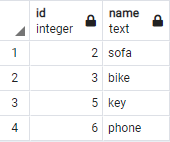


Рисунок 30 – Стан таблиці складу після спрацьовування тригера на вставку.



Рисунок 31 – Запит для видалення з таблиці позик, що повинен викликати тригер.

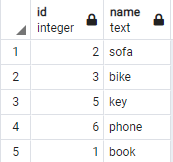


Рисунок 32 – Стан таблиці складу після спрацьовування тригера на видалення.

**Робота з транзакціями**

Будемо досліджувати рівні ізоляції транзакції на таблиці test у якого буде всього одне числове поле value. Цього буде достатньо для експериментів.

Почнемо з Readcommited, який в PostgreSQL встановлений за замовченням.

Спочатку вставимо у таблицю перший рядок.

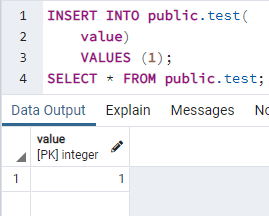


Рисунок 33 – Запит для додавання першого рядка в таблицю.

Тепер запустимо 2 транзакцій у двох вікнах та будемо послідовно виконувати команди

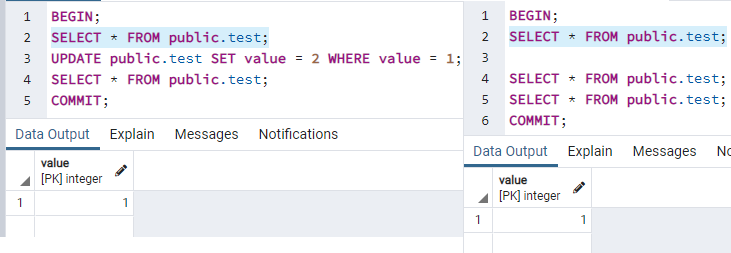


Рисунок 34 – Виконання першого SELECT у двох паралельних транзакціях

Бачимо, що дані в обох транзакціях поки однакові.

Тепер виконаємо update в першій транзакції та потім select в обох транзакціях.

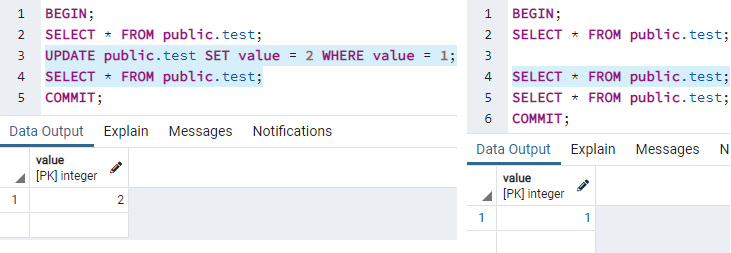


Рисунок 35 – Виконання другого SELECT у двох паралельних транзакціях.

Бачимо, що в першій транзакції дані змінилися , а в другій залишилися ті самі.

Тепер виконаємо в першій транзакції COMMIT та SELECT у другій.

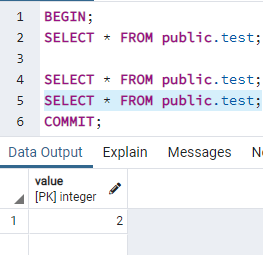


Рисунок 36 – “Unrepeatable read” в другій транзакції.

Ми маємо феномен під назвою “Unrepeatable read” в другій транзакції. Друга транзакція ще не закінчена, але ми бачимо вже зміненні дані з першої транзакції. Тому цей рівень ізоляції транзакцій має назву “READ COMMITED”, що означає, що в транзакціях будемо бачити при читанні дані, які в момент читання були в базі.

Для усунення цього ефекту треба перейти до рівня ізоляції транзакції “REPEATABLE READ”.

Запустимо знову 2 транзакції, але вже встановимо їм рівень ізоляції транзакції “REPEATABLE READ”.

Бачимо, що на початку транзакцій дані в них відображуються однакові

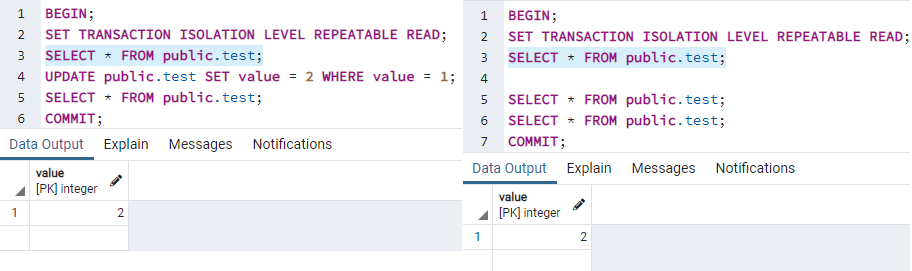


Рисунок 37 – Виконання першого SELECT у двох паралельних транзакціях з рівнем ізоляції “REPEATABLE READ”.

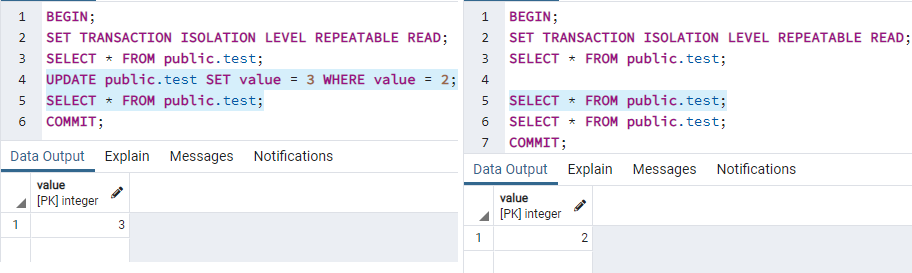


Рисунок 38 – Виконання другого SELECT у двох паралельних транзакціях з рівнем ізоляції “ REPEATABLE READ ”.

Тепер виконаємо в першій транзакції COMMIT та SELECT у другій.

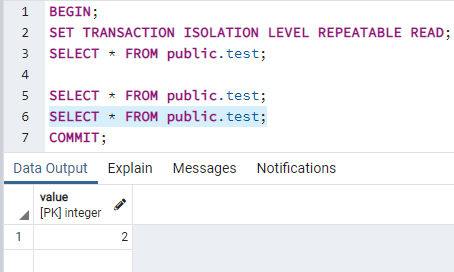


Рисунок 39 – Уникнення “Unrepeatable read” в другій транзакції.

Вдалося позбутися феномену “Unrepeatable read”. Суть рівня ізоляції транзакцій “REPEATABLE READ” в тому, що в транзакції ми можемо бачити тільки зміни зроблені в даній транзакції.

Тепер дослідимо рівень транзакції “SERIALIZABLE”.

Відкриємо 2 транзакції з рівнем ізоляції “SERIALIZABLE”, та виведемо дані таблиці test через select:

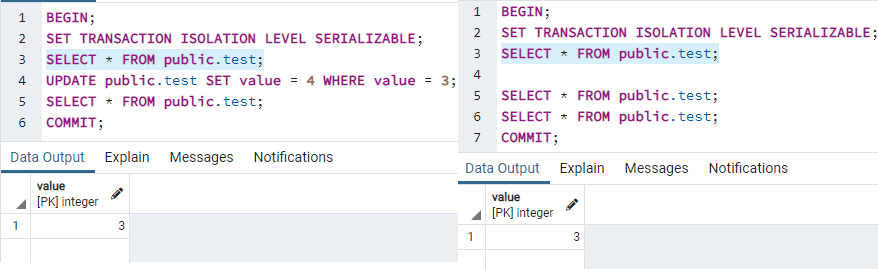


Рисунок 40 – Виконання першого SELECT у двох паралельних транзакціях з рівнем ізоляції “SERIALIZABLE”.

А тепер спробуємо зробити в обох транзакціях UPDATE. Спочатку в першій, а потім в другій

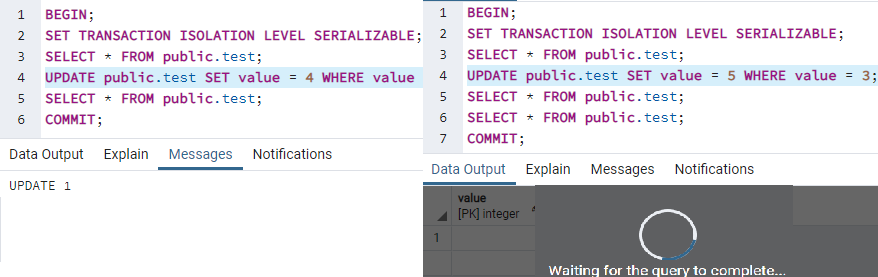


Рисунок 41 – Виконання UPDATE у двох паралельних транзакціях з рівнем ізоляції “SERIALIZABLE”.

Перша транзакція завершилась успішно, друга транзакція зависла.

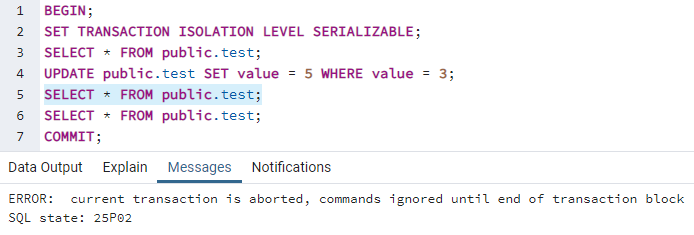


Рисунок 42 – Помилка при виконанні UPDATE та подальших запитів у другій транзакції з рівнем ізоляції “SERIALIZABLE”.

Тепер друга транзакція перервана та може бути тільки закінчена за допомогою COMMIT. Це сталося через те, що в другій транзакції була спроба модифікувати рядок, який вже було зміненено у першій транзакції, що й призвело до конфлікту, бо рівень ізоляції транзакції “SERIALIZABLE” не дозволяє цього.