

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

**Лабораторна робота № 3**

з дисципліни «Бази даних і засоби управління»

«**Засоби оптимізації роботи СУБД PostgreSQL**»

Виконав: Дріф Руслан

Студент групи КВ-91

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.

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

**Варіант 10**

Індекси за варіантом: Hash, BRIN

Тригер: after delete, insert

**Логічна модель учбової предметної області «IT-company»**

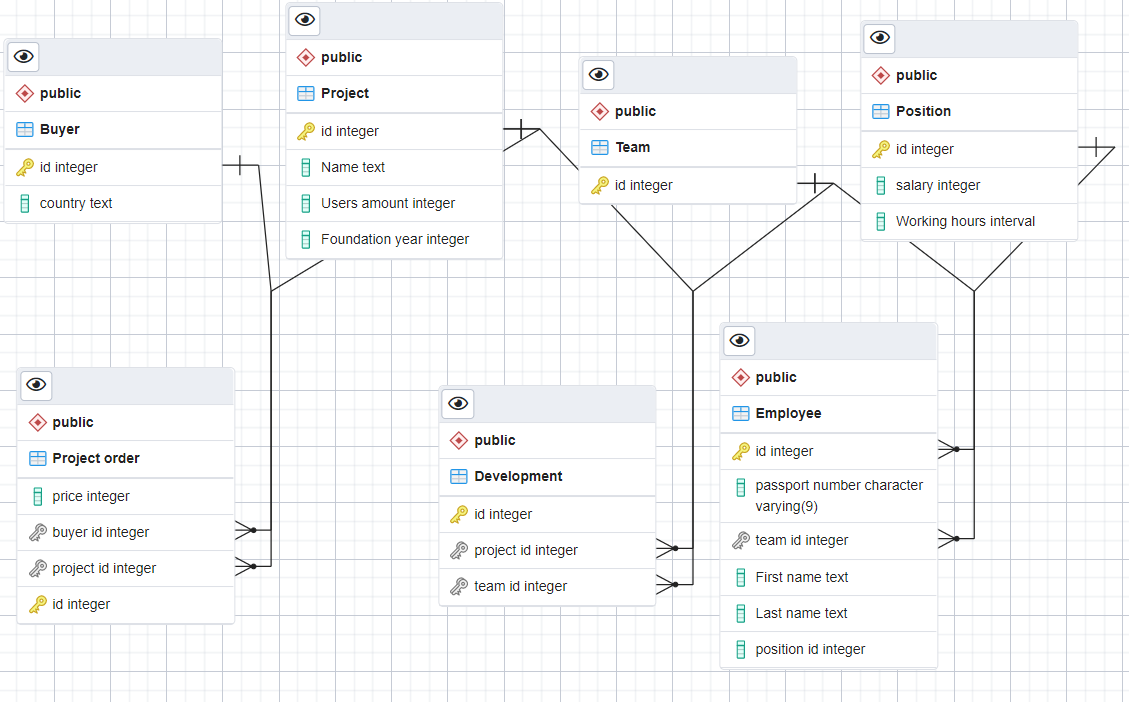


рисунок 1 – логічна модель предметної області «IT-компанія»

**Середовище 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

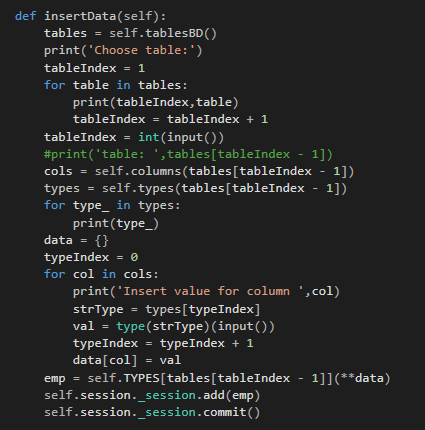


Рисунок 2 – фрагмент програми внесення даних

1. Фрагмент програми вилучення даних

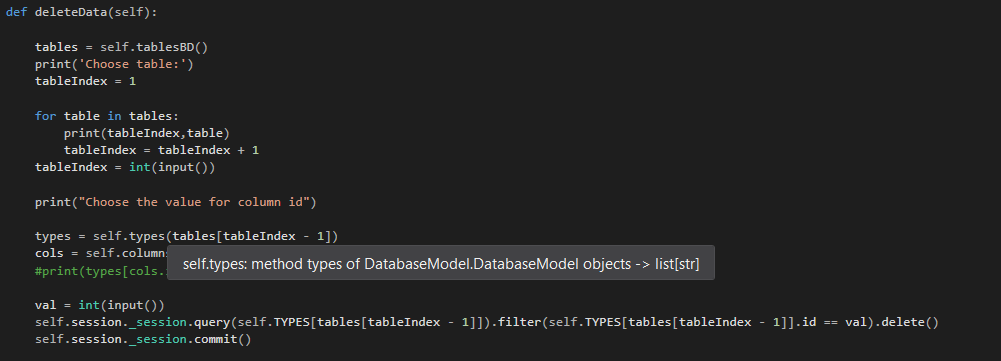


Рисунок 3 – фрагмент програми вилучення даних

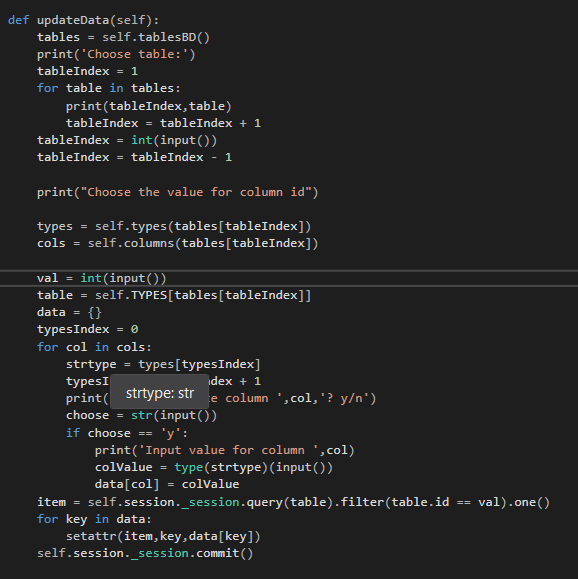
1. Фрагмент програми редагування даних

Рисунок 4 – фрагмент програми редагування даних

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

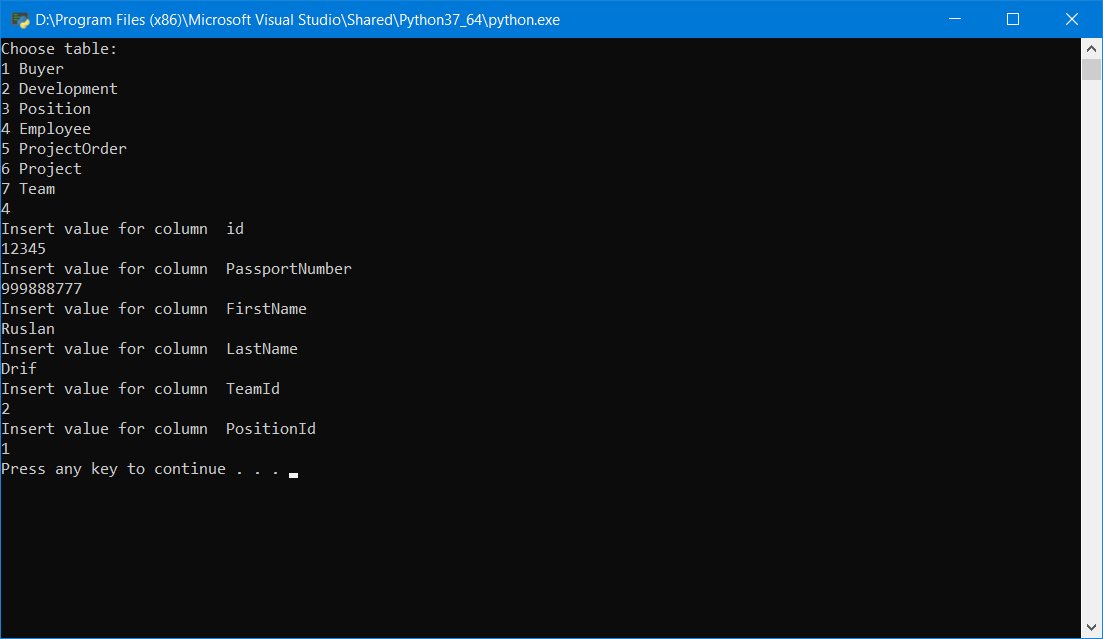


рисунок 5 – демонстрація виконання вставки в таблицю

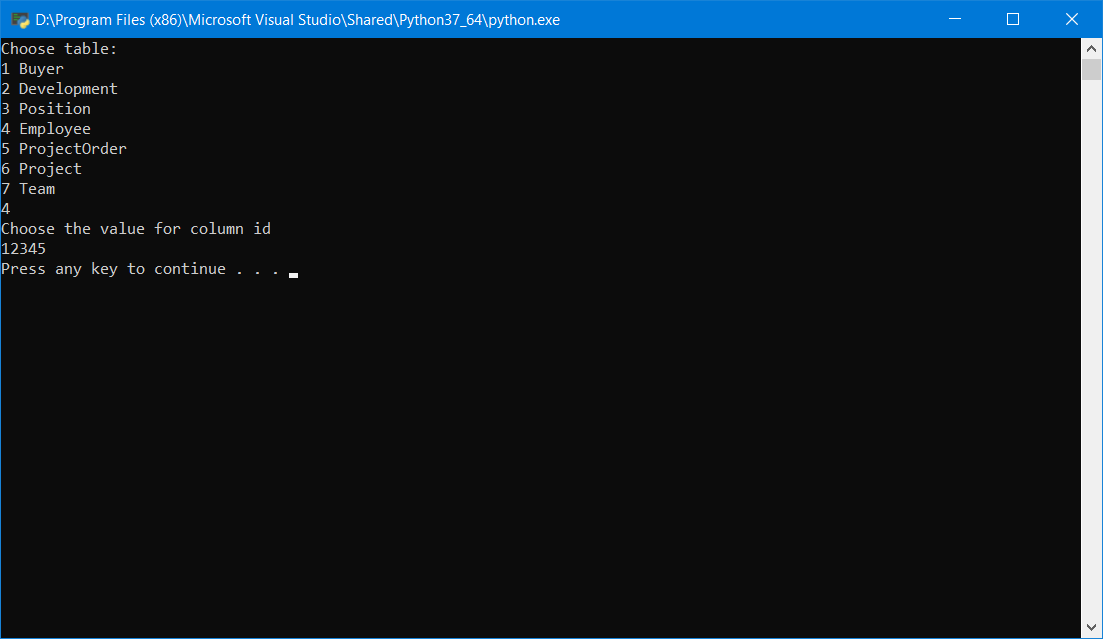


рисунок 6 – демонстрація видалення запису з таблиці

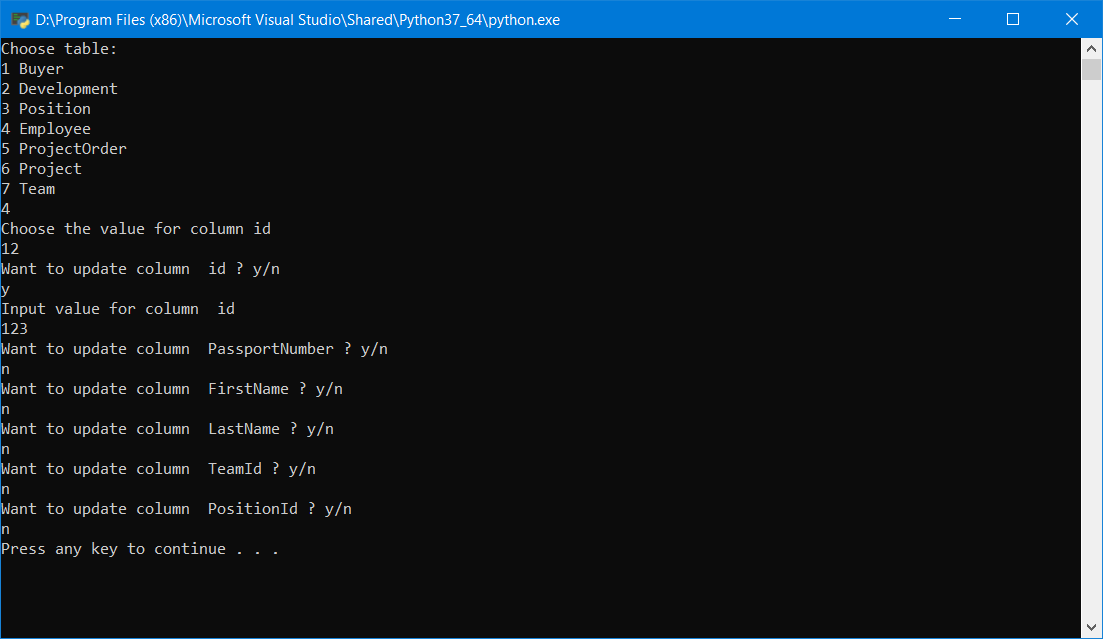


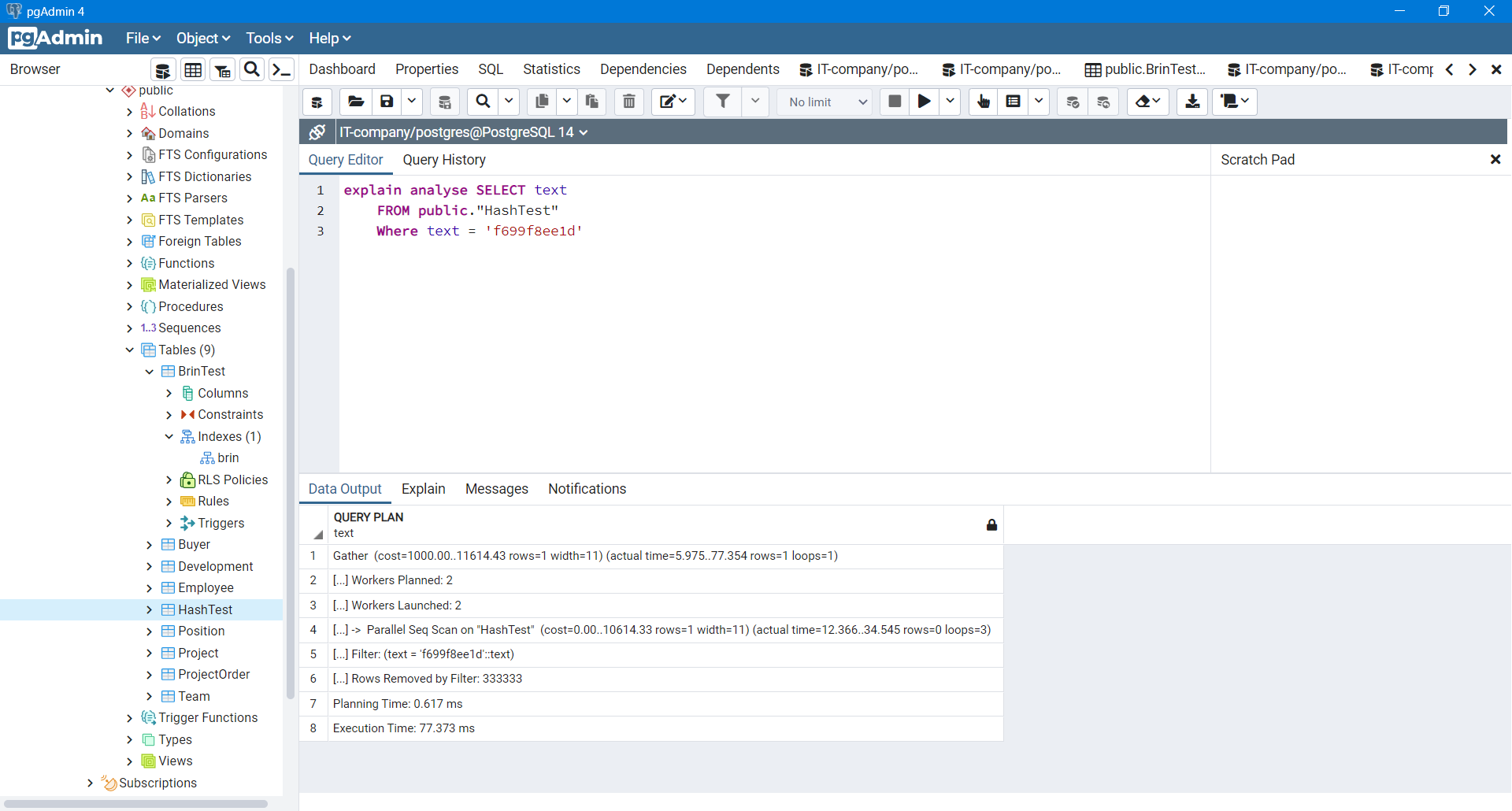
рисунок 7 – демонстрація редагування запису в таблиці

**Робота з індексами**

**Hash**

Hash індекс насамперед використовується для пришвидшення запитів на знаходження конкретного елементу(ів). Кожному значенню ставиться у відповідність ціле число, що отримується з хеш-функції, і таким чином пришвидшується пошук.

Для тестування хеш індексу була створена таблиця HashTest з одним полем text TEXT, яке буде використовуватись для дослідження хеш індексу.

Рисунок5 - Аналіз виконання запиту1 без індексу

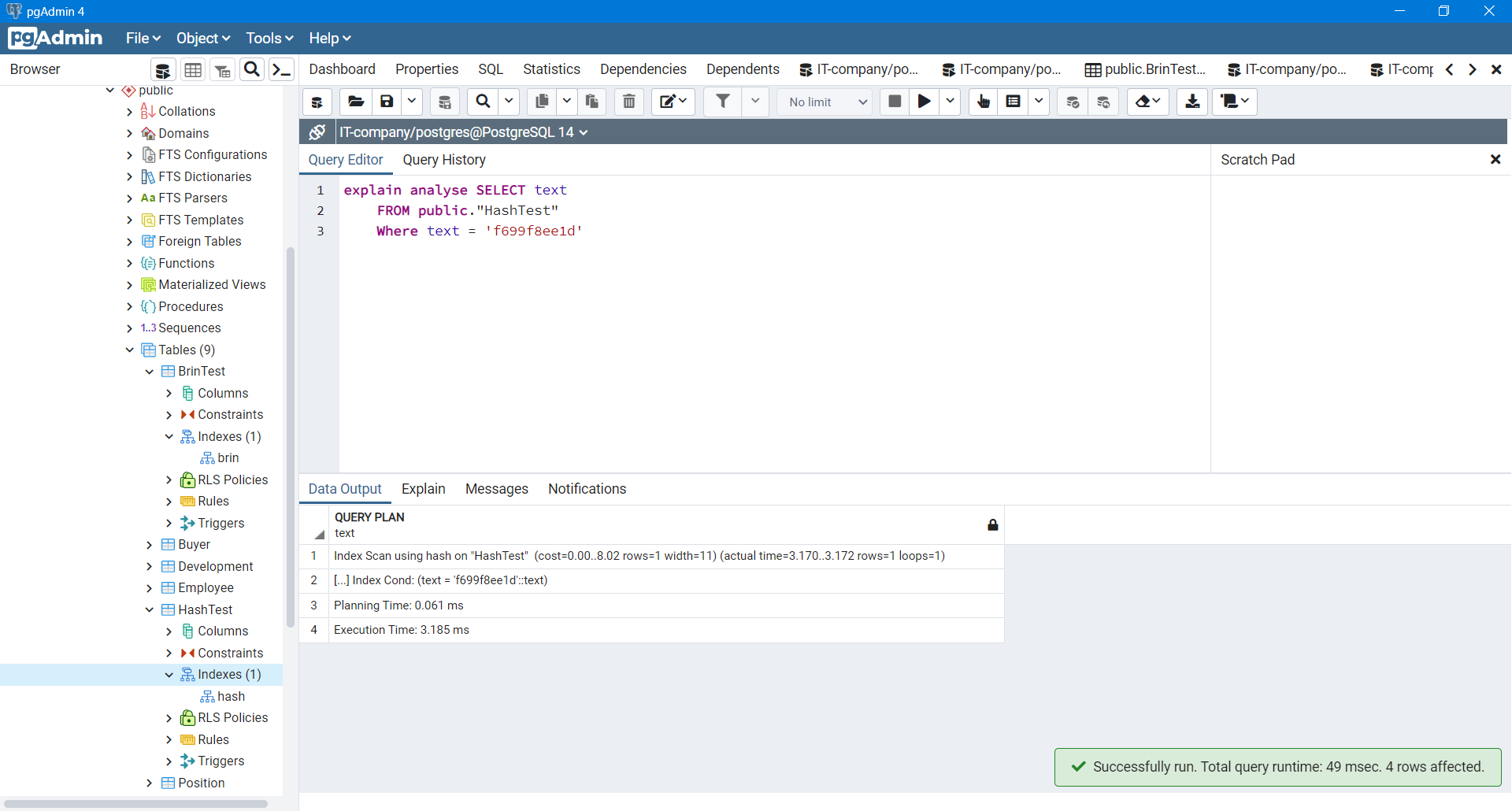


Рисунок6 - Аналіз виконання запиту1 з індексом

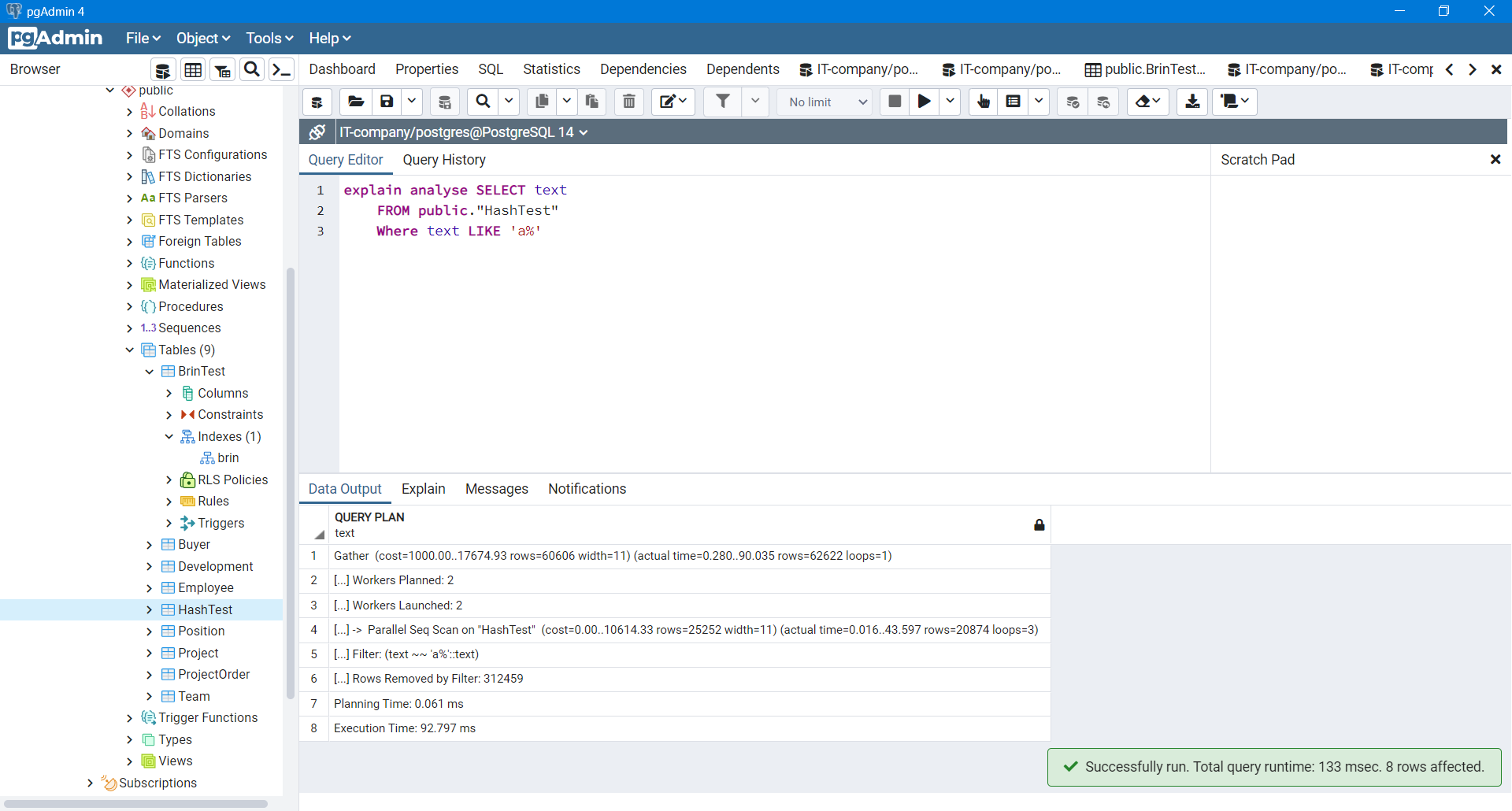


Рисунок7 - Аналіз виконання запиту2 без індексу

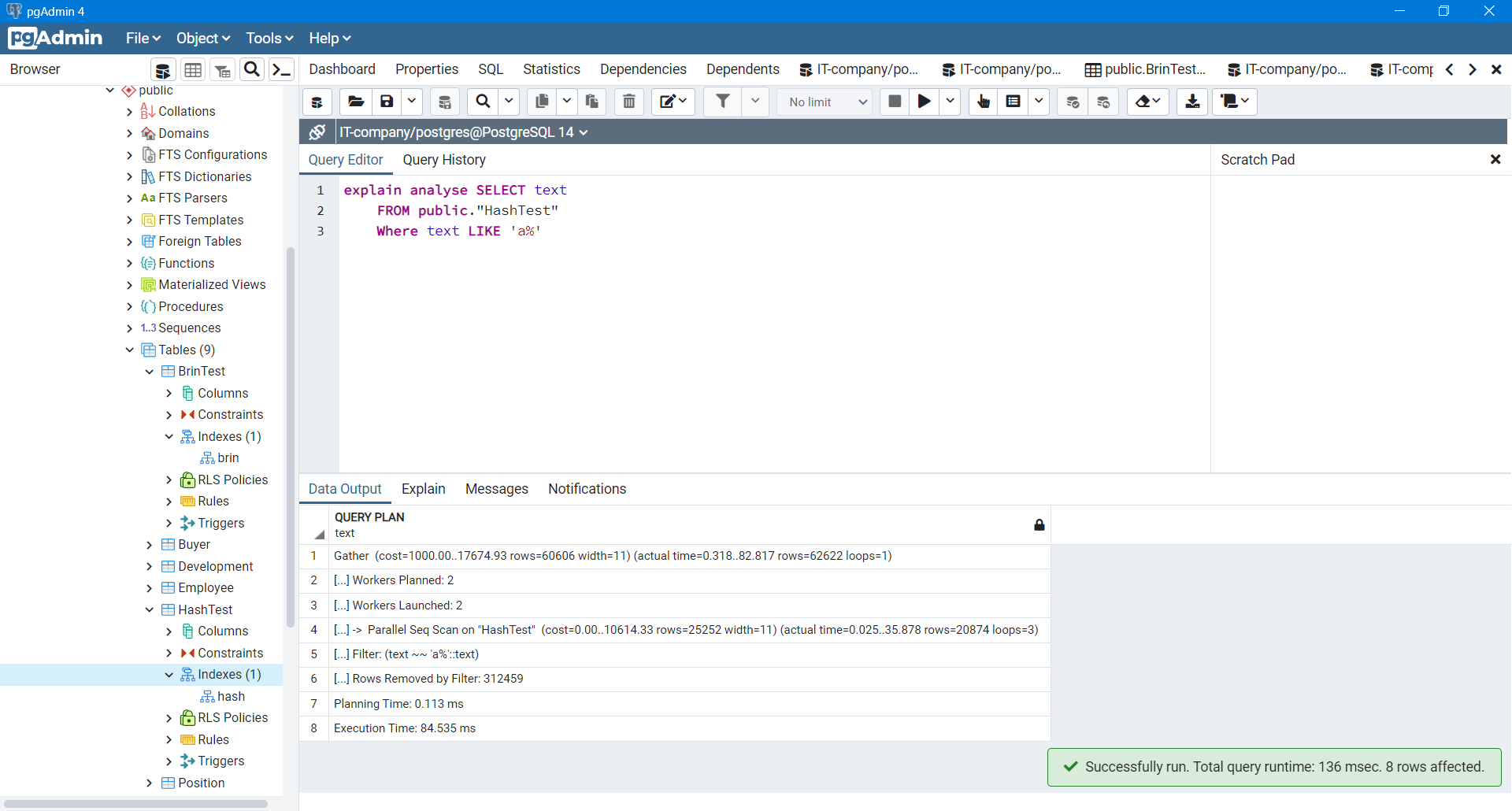


Рисунок8 - Аналіз виконання запиту2 з індексом

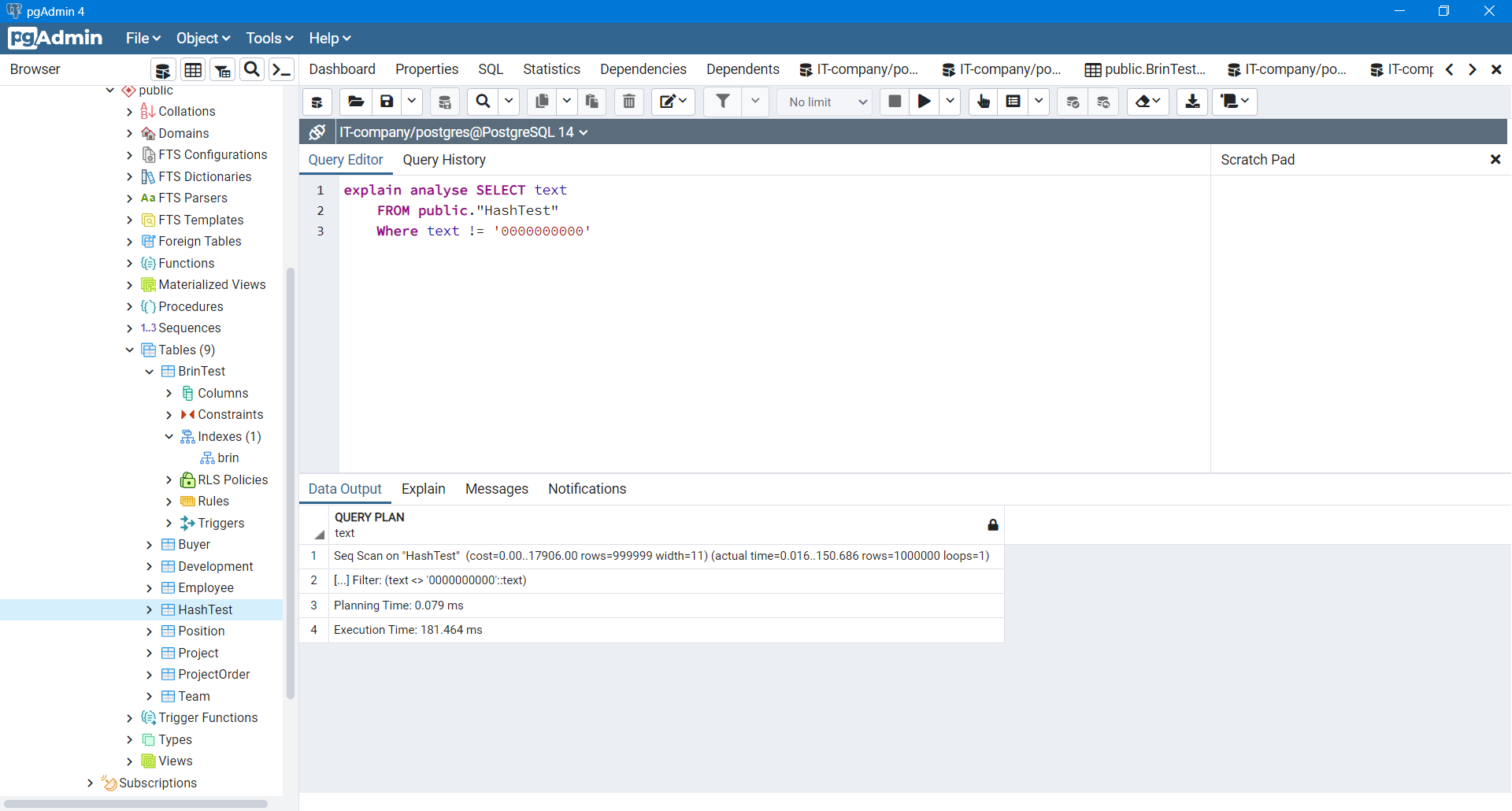


Рисунок9 - Аналіз виконання запиту3 без індексу

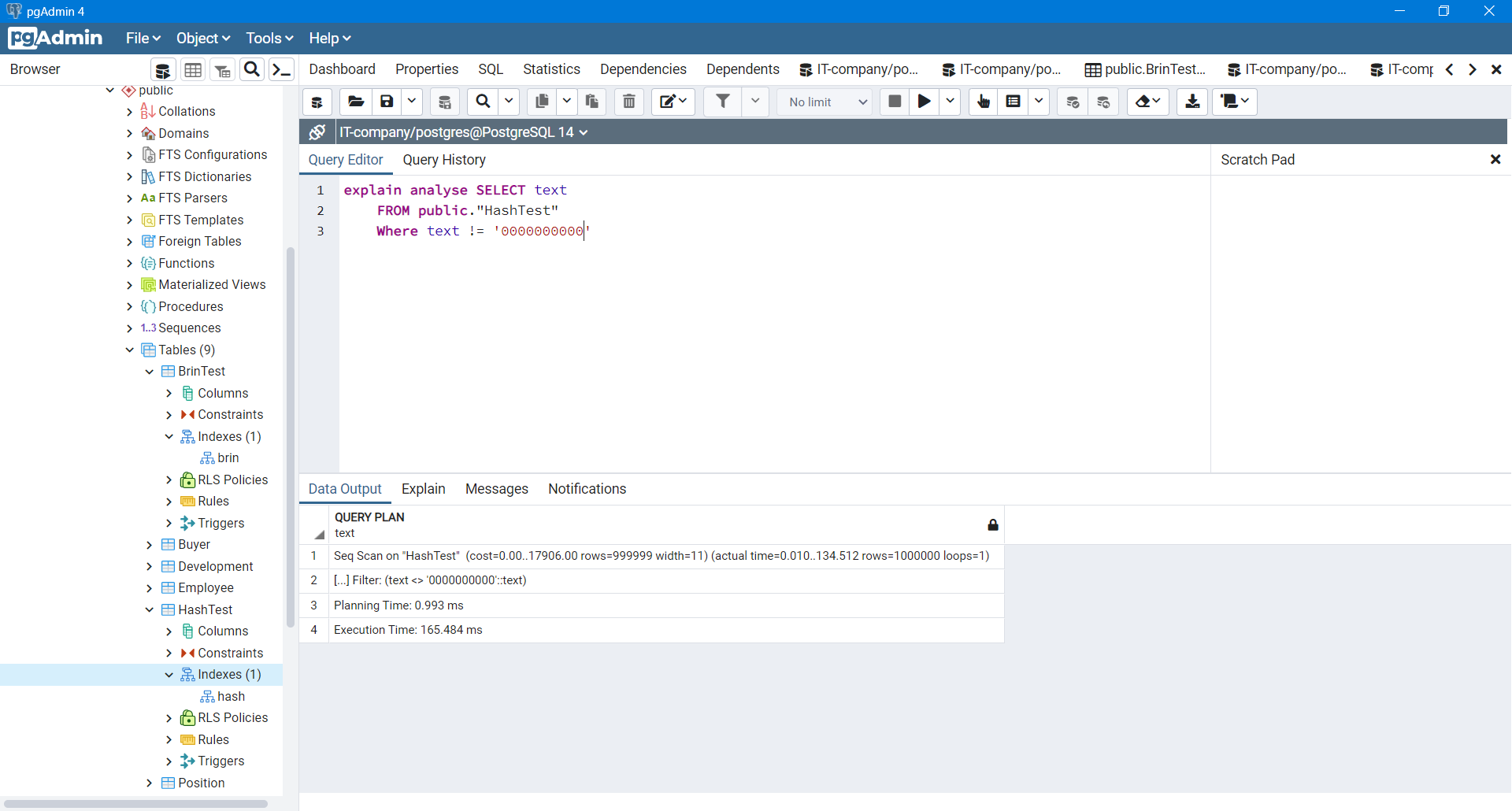


Рисунок10 - Аналіз виконання запиту з індексом

Як видно, найбільше підвищення швидкодії спостерігалося при виконанні запиту з порівнянням по значенню, що пояснюється як раз будовою хеш індексу

**Brin**

BRIN – це Block Range Index, головна концепція якого не знаходження необхідного значення, а уникнення перегляду свідомо непотрібних.

Він працює добре для тих стовпчиків, де значення корелюють із їх фізичним положенням в таблиці. Тобто, якщо запит без ORDER BY видає значення стовпчика практично в порядку зростання чи спадання.

Для дослідження індексу була створена тестова таблиця, яка має дві колонки: number INTEGER та name CHAR

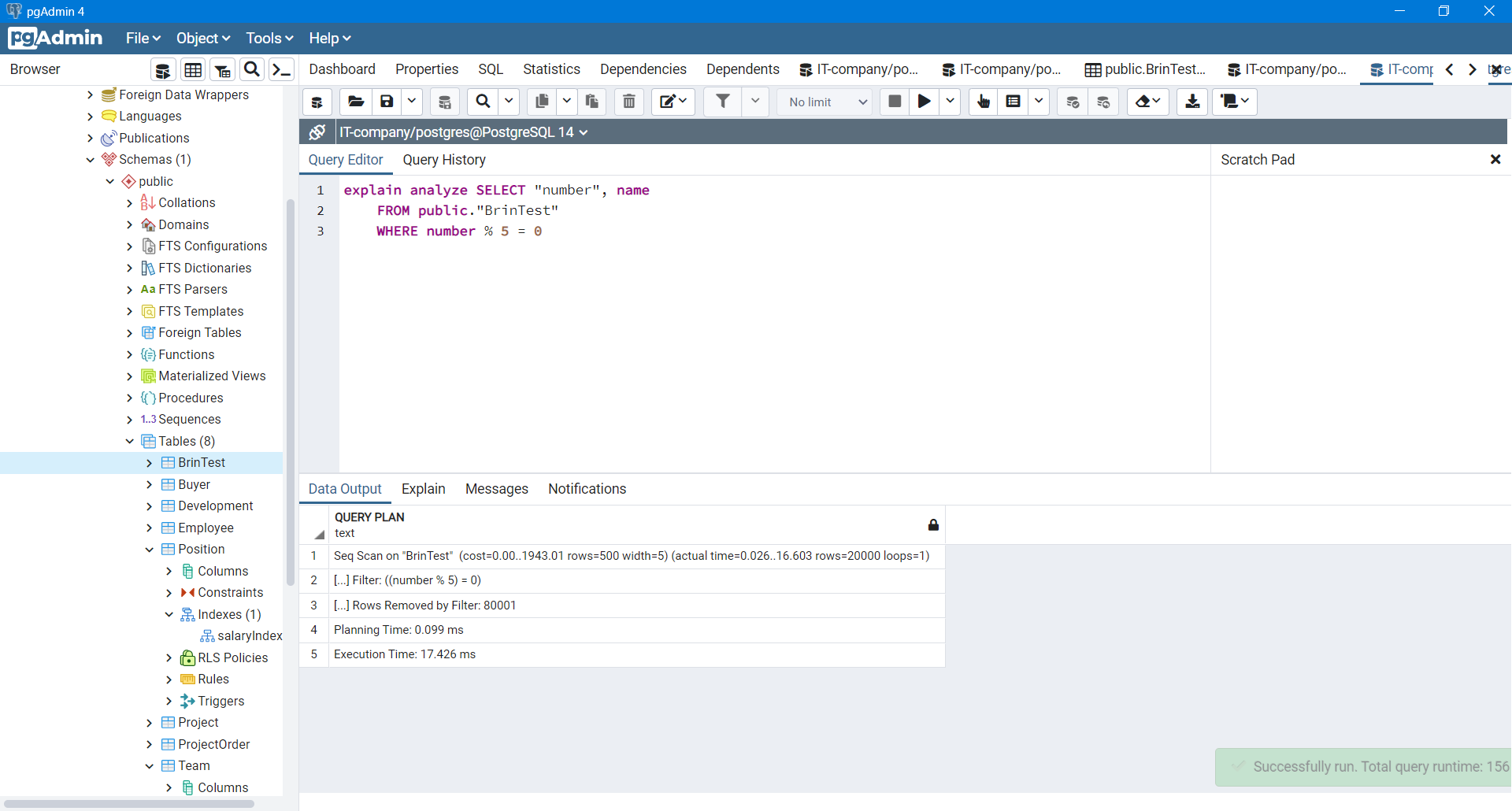


Рисунок8 – Аналіз виконання запиту1 без індексу

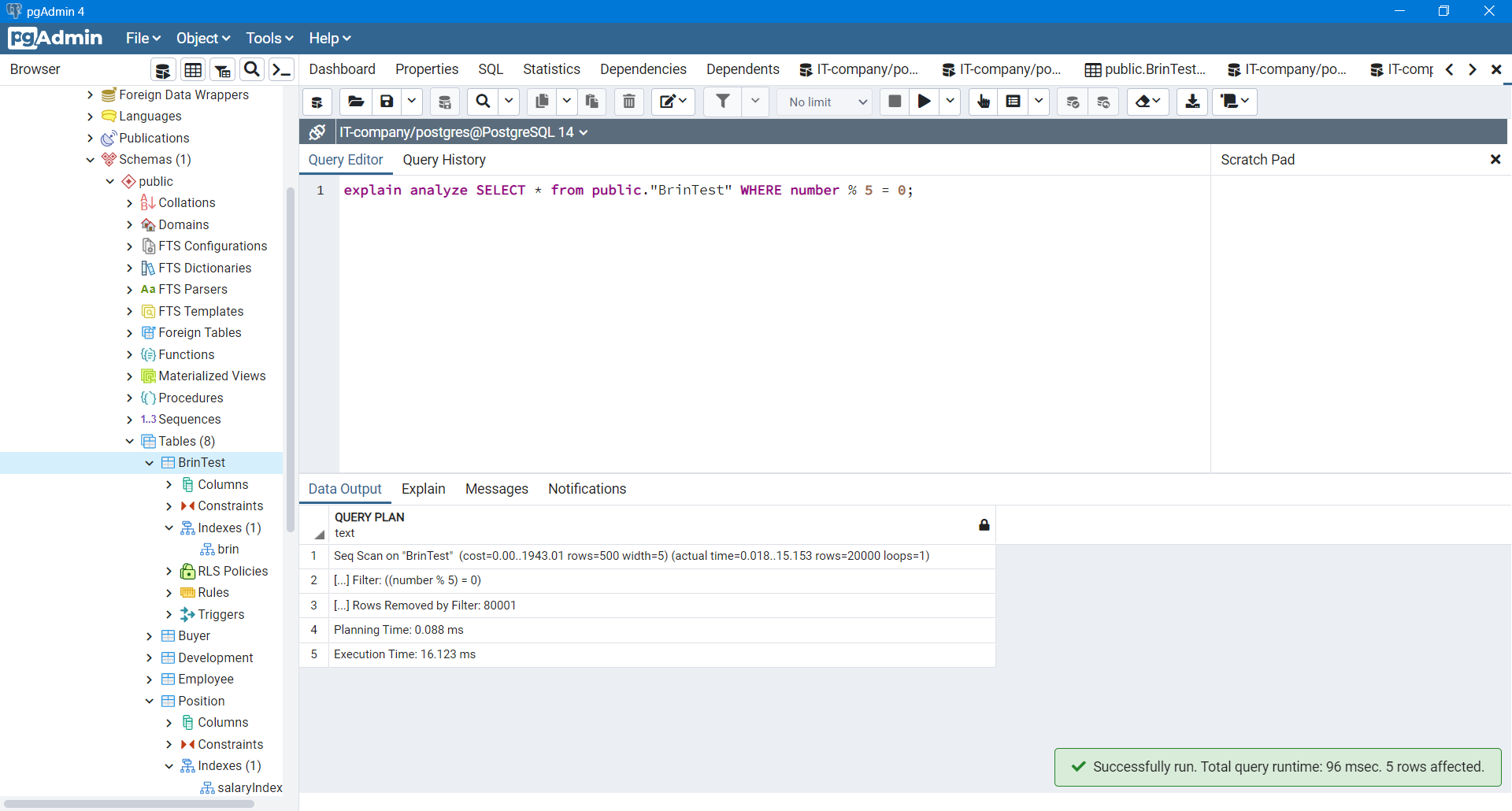


Рисунок9 – Аналіз виконання запиту1 з індексом

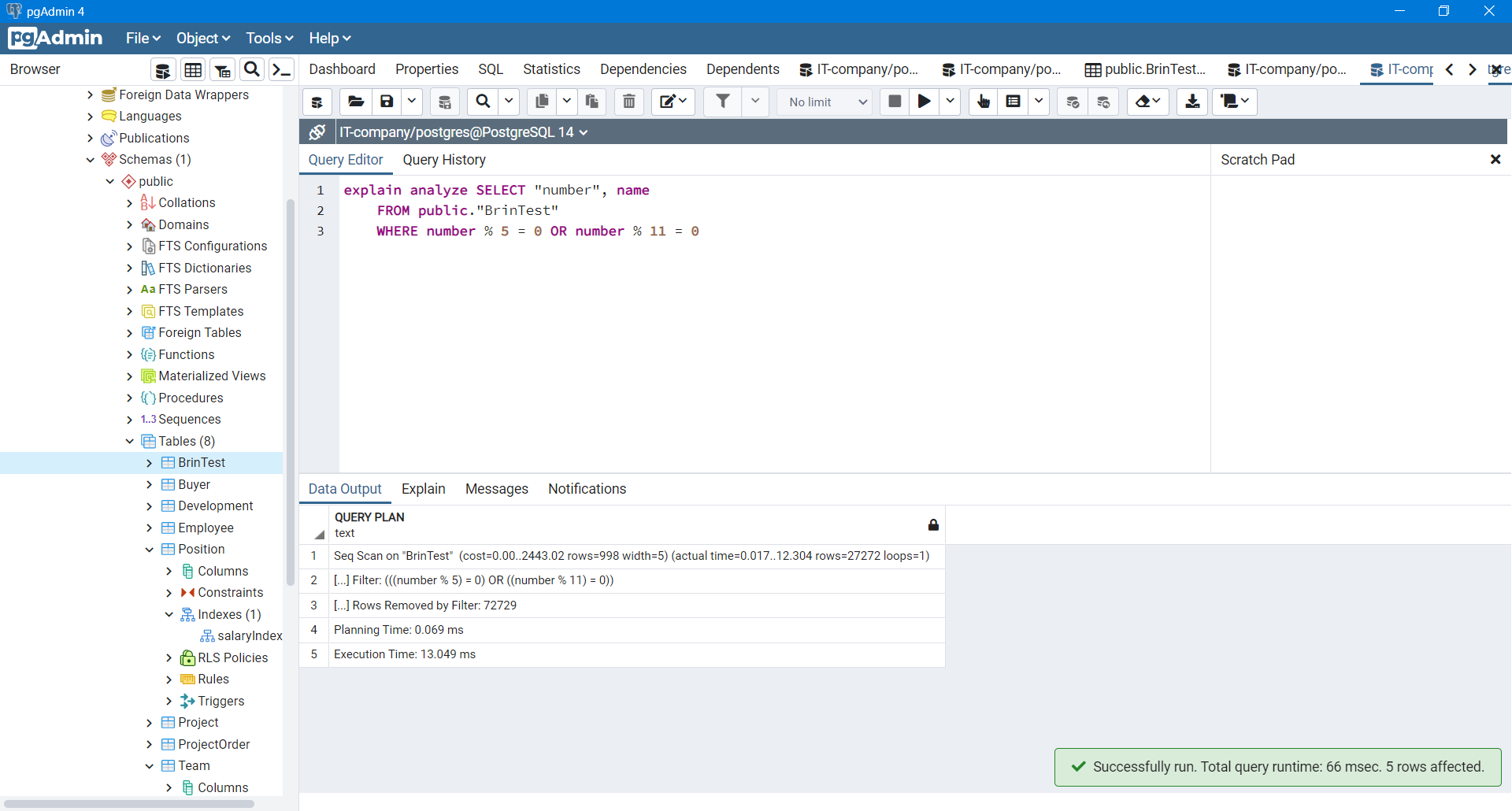


Рисунок10 – Аналіз виконання запиту2 без індексу

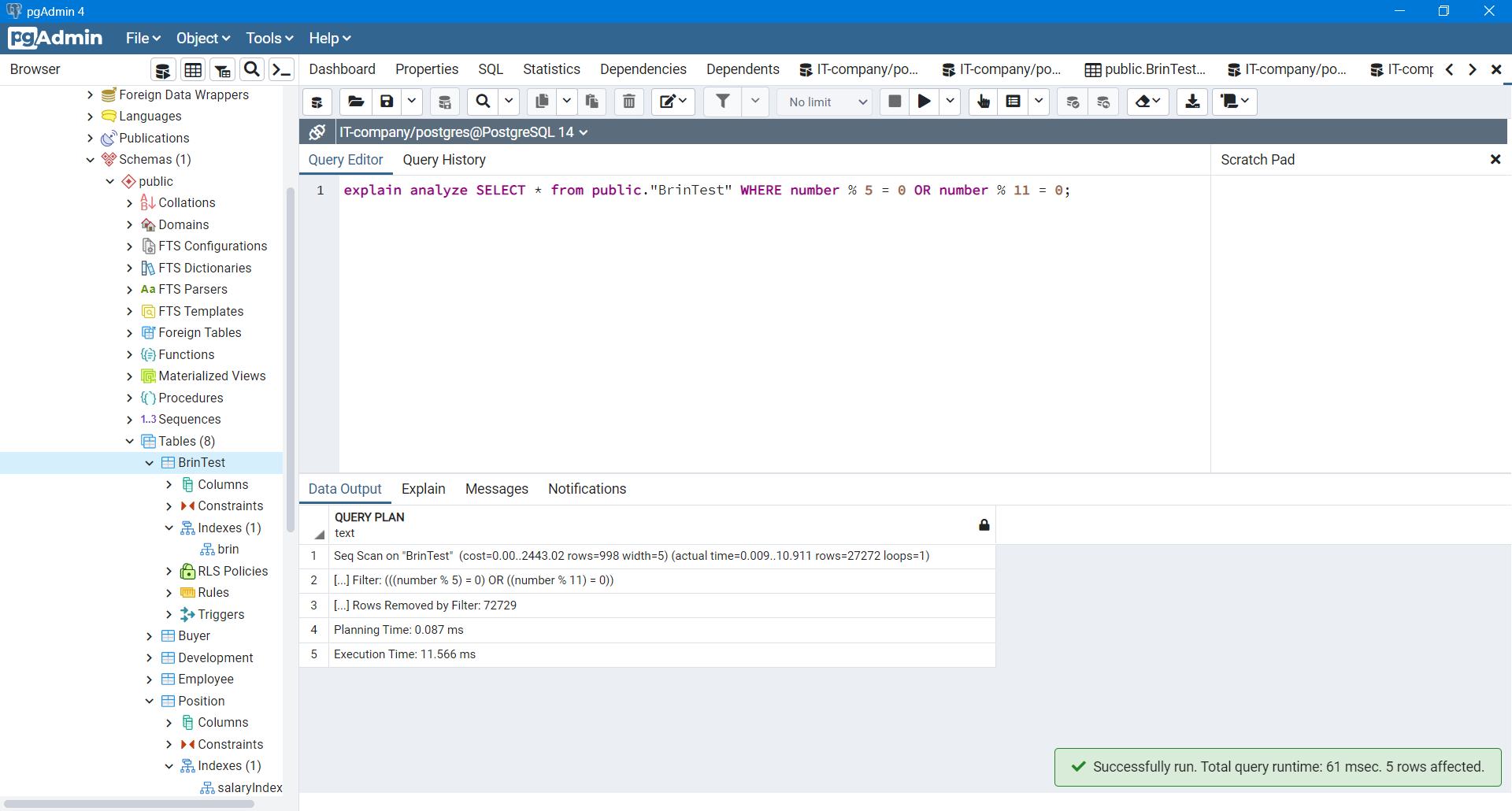


Рисунок11 – Аналіз виконання запиту2 з індексом

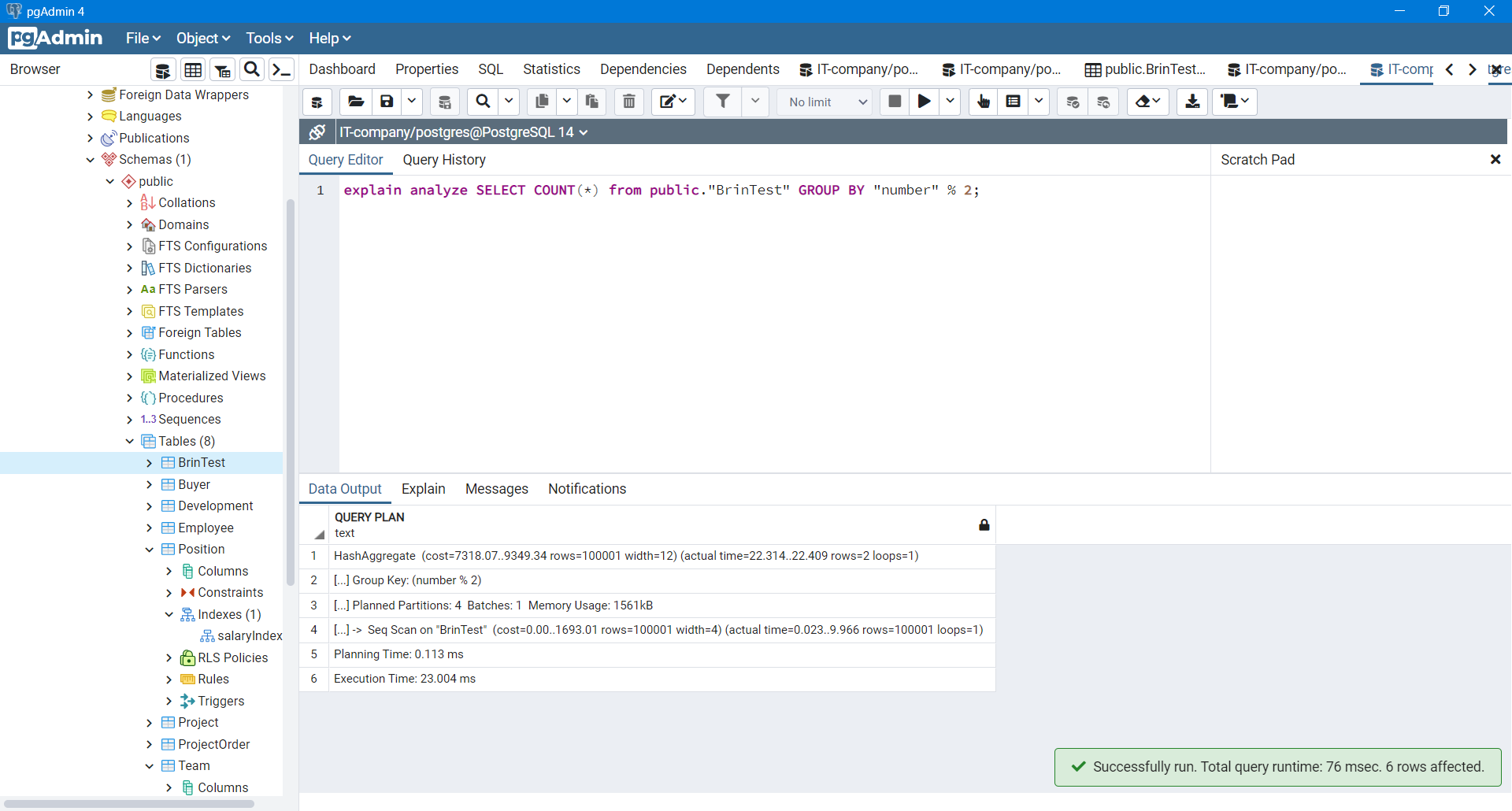


Рисунок12 – Аналіз виконання запиту3 без індексу

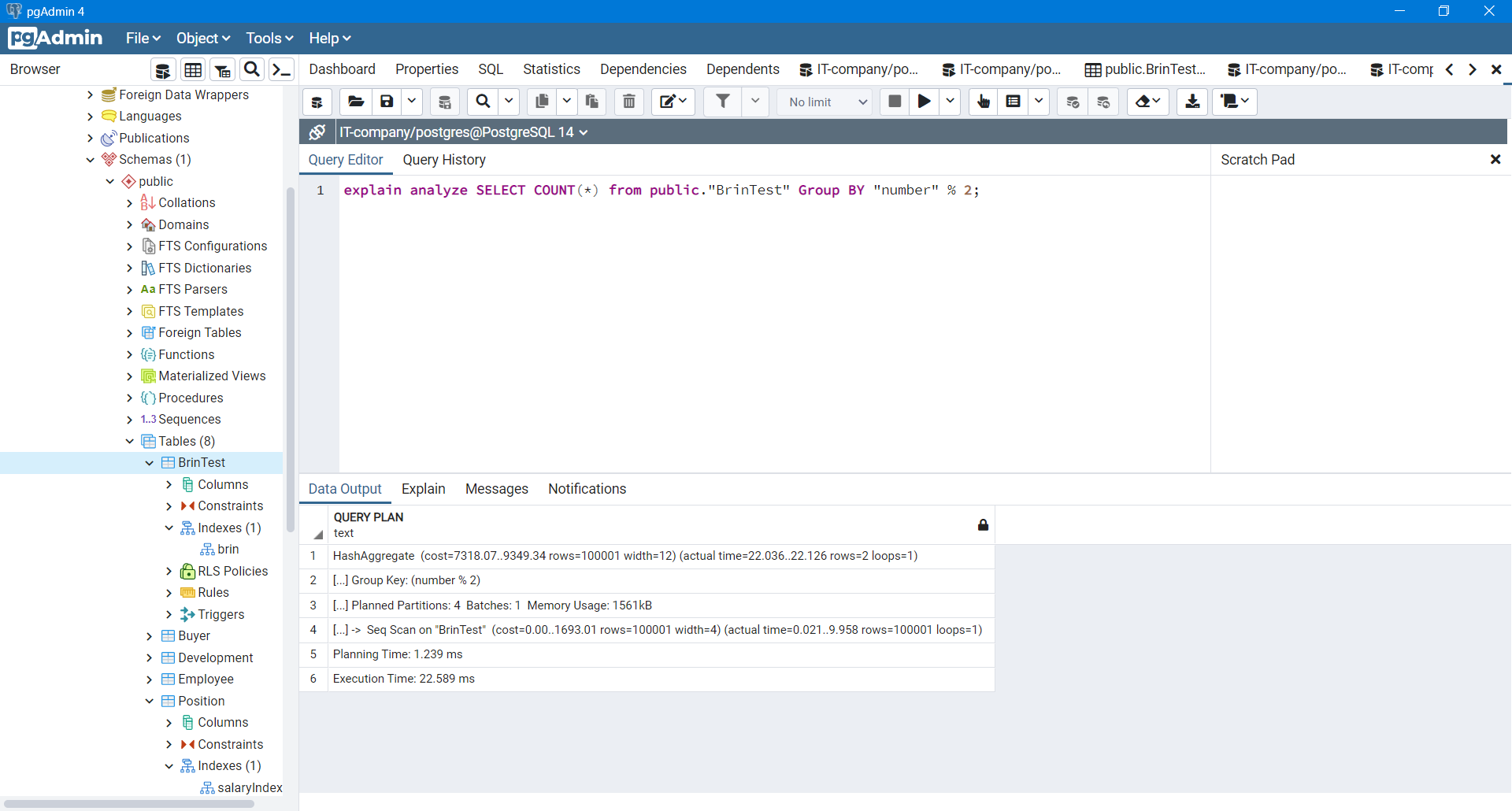


Рисунок13 – Аналіз виконання запиту3 з індексом

**Робота з тригерами**

Для тестування тригерів створимо таблицю робітника, в якій буде два поля – зарплата та ідентифікатор. При додаванні, або видаленні співробітника, заробітна плата всіх працівників буде розподілятися порівну враховуючи бюджет компанії – 5000$.

Після вставки або видалення рядка з таблиці в функції відбувається прохід по всім рядкам таблиці та встановлення заробітної плати порівну кожному співробітнику.

Виконаємо вставку запису зі значенням ЗП 7000, після виконання тригерної функції значення повинно стати рівним 5000.

INSERT INTO public."Emp"(salary) VALUES (7000);

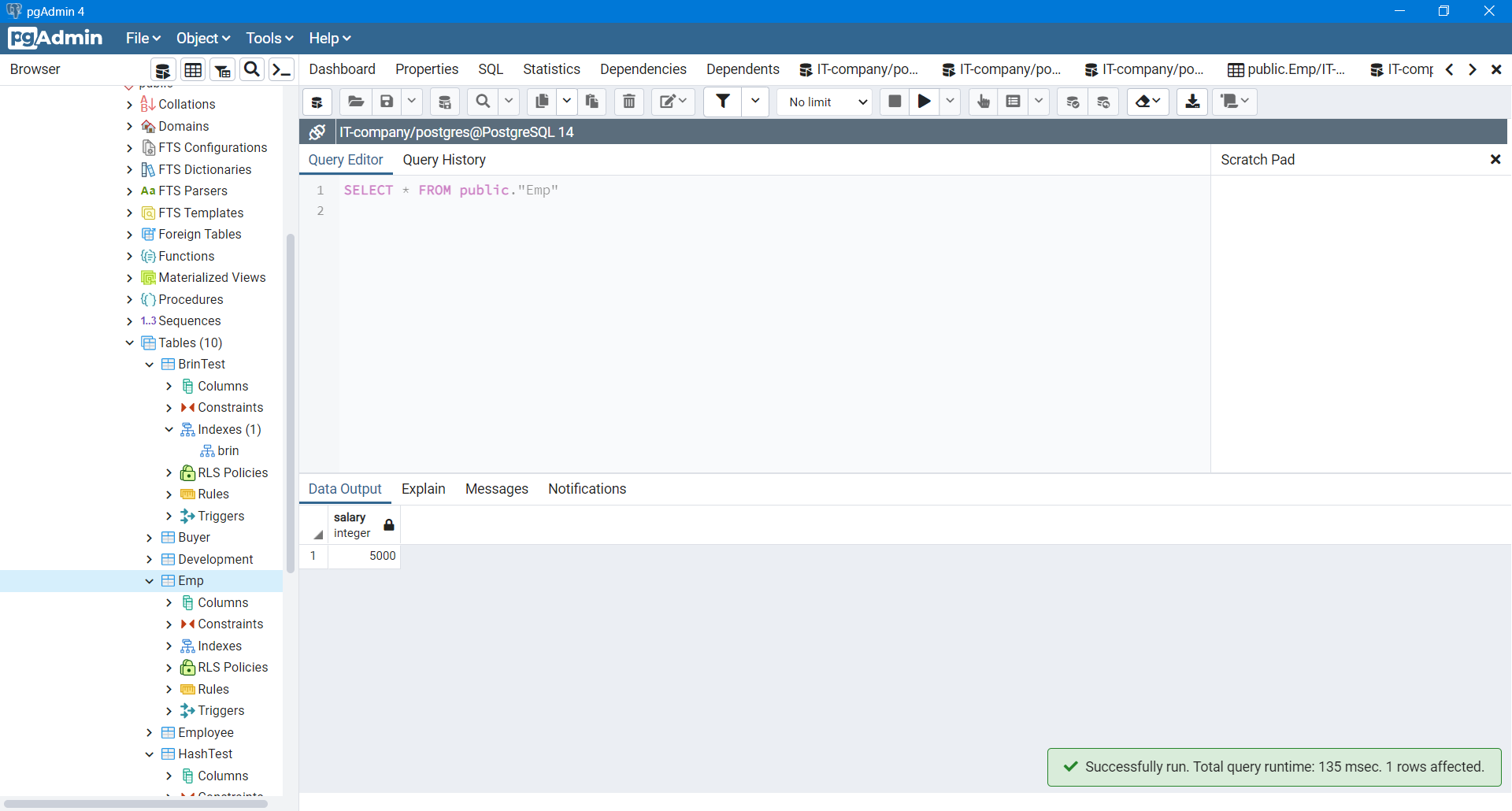


Рисунок14 – Результат виконання вставки в таблицю

Виконаємо вставку ще одного запису зі значенням ЗП 6000, після після чого видалимо запис з зп 7000, значення ЗП залишеного співробітника має дорівнювати 5000.

INSERT INTO public."Emp"(salary) VALUES (6000);

DELETE FROM public."Emp" WHERE id = 1;

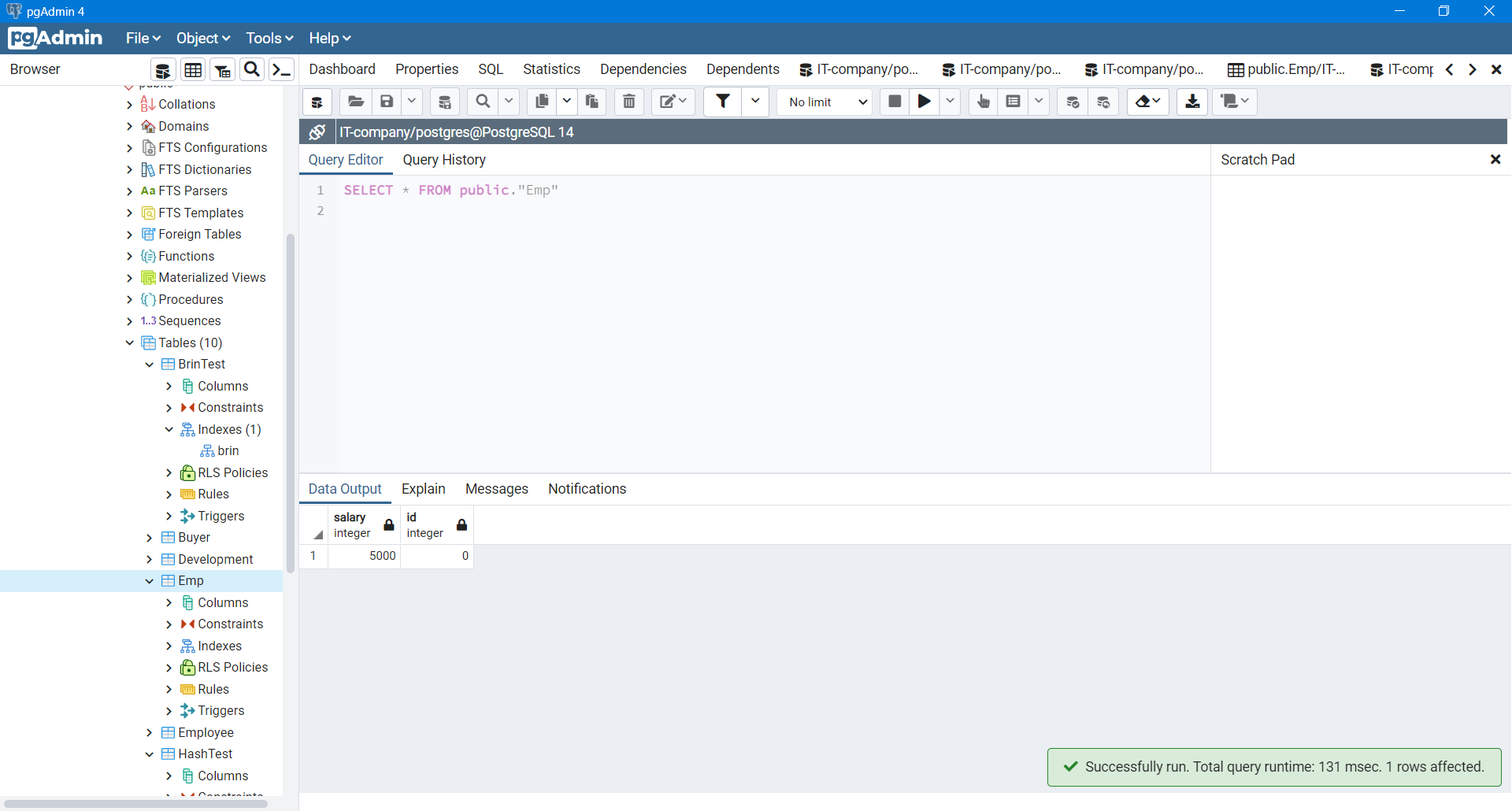


Рисунок15 – Результат видалення запису з таблиці

UPDATE э операцією, на яку не діє тригер. Оновимо значення зп нашого співробітника до значення 4500.

UPDATE public."Emp" SET salary=4500 WHERE id = 0

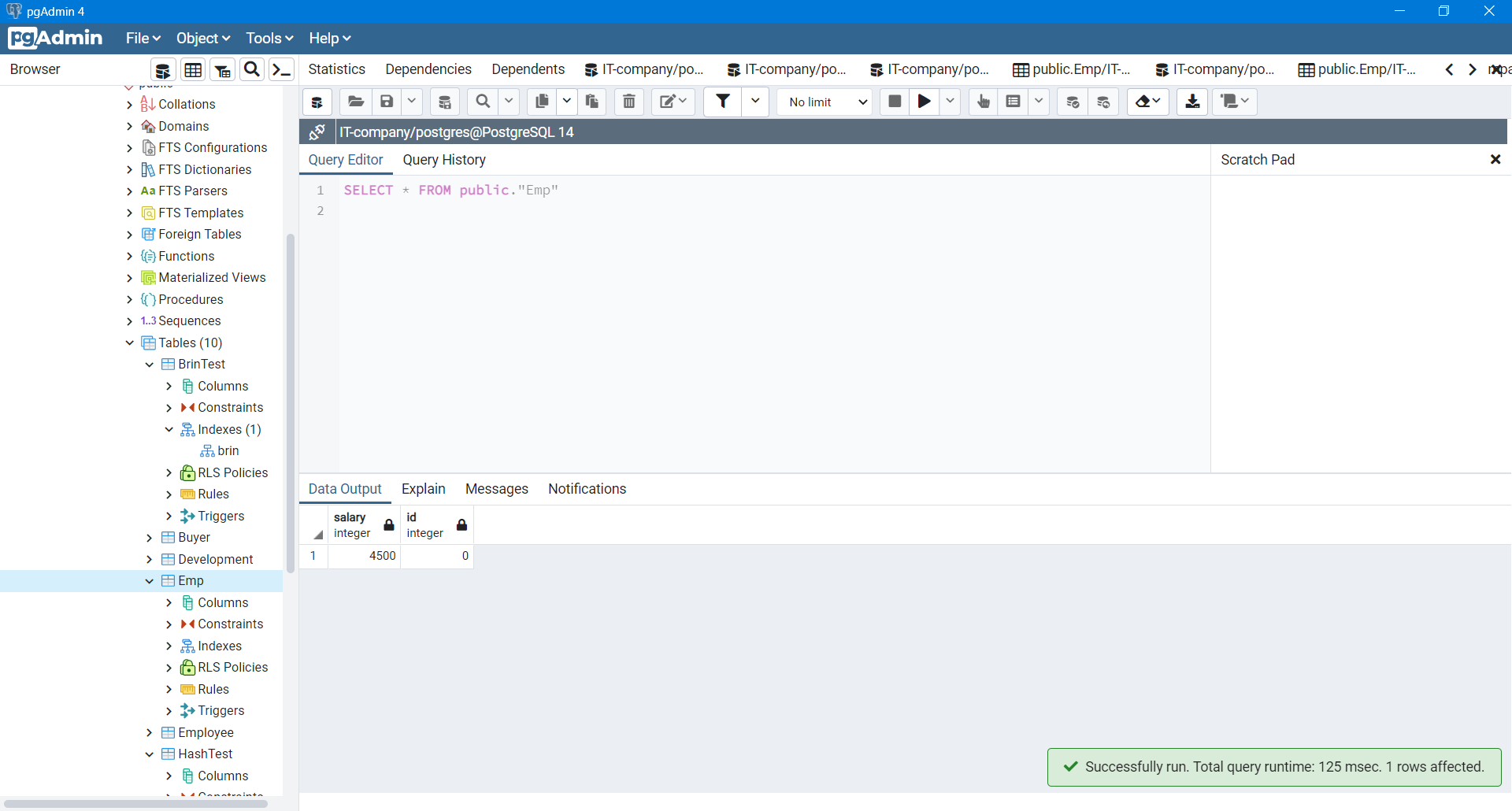


Рисунок16 – Результат виконання не тригерної операції

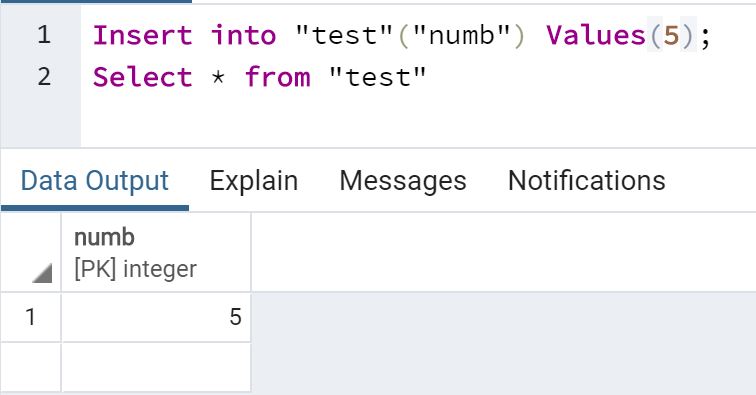
**Робота з транзакціями**

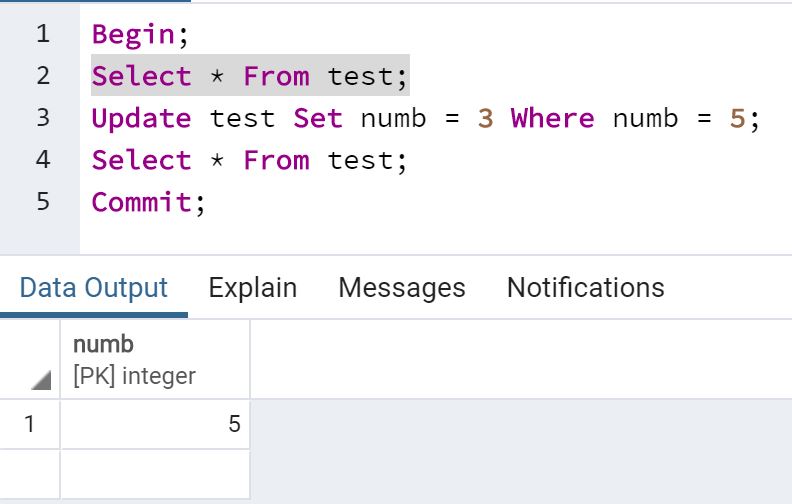
**Робота з транзакціями**

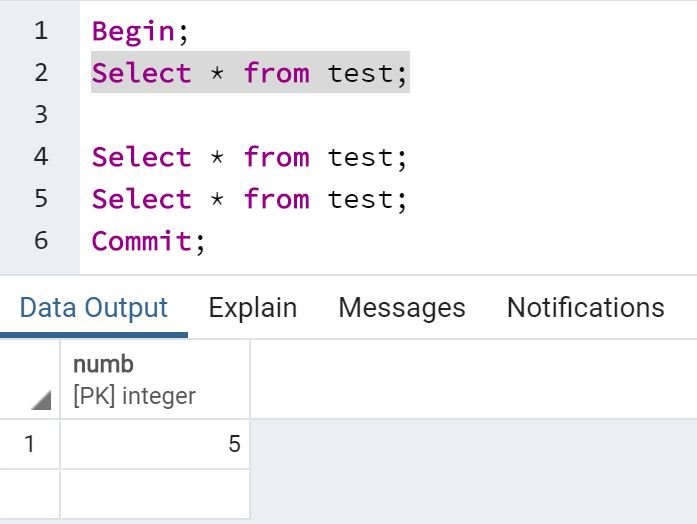
Будемо досліджувати рівні ізоляції транзакції на таблиці test у якого буде всього одне числове поле numb. Цього буде достатньо для експериментів.

Почнемо з Readcommited, який в PostgreSQL встановлений за замовченням.

Спочатку вставимо у таблицю перший рядок.

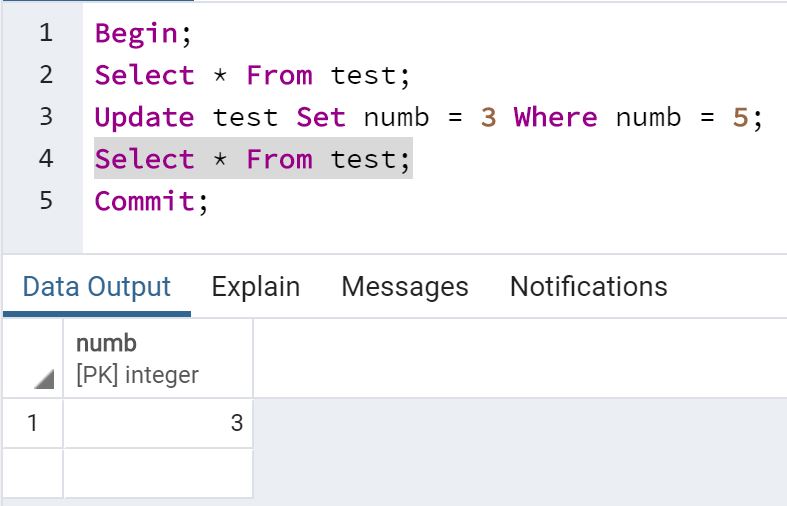
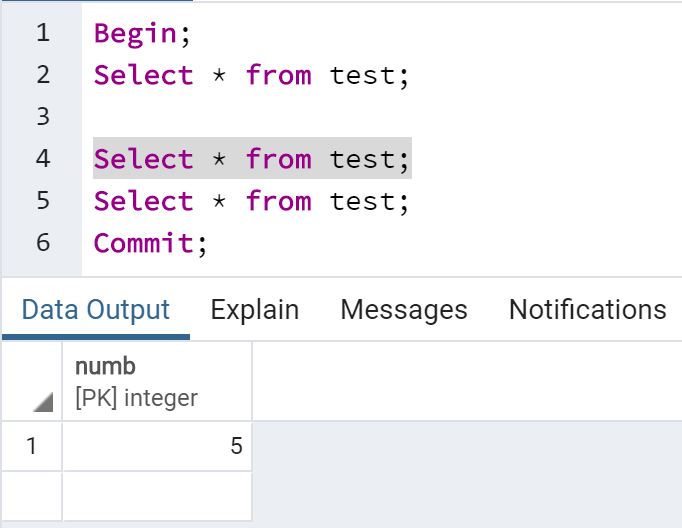


Тепер запустимо 2 транзакцій у двох вікнах та будемо послідовно виконувати команди



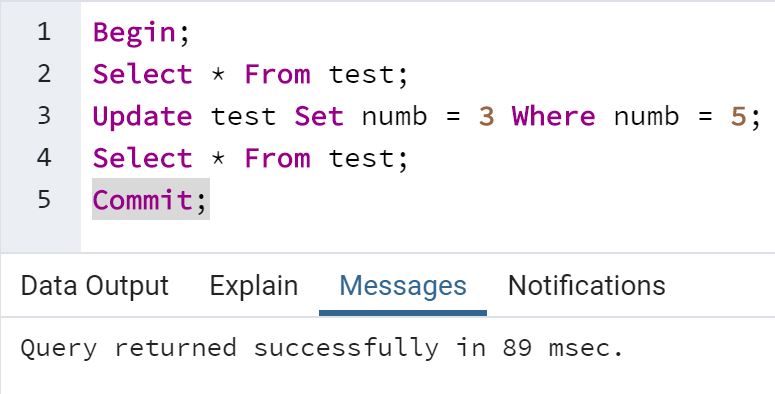
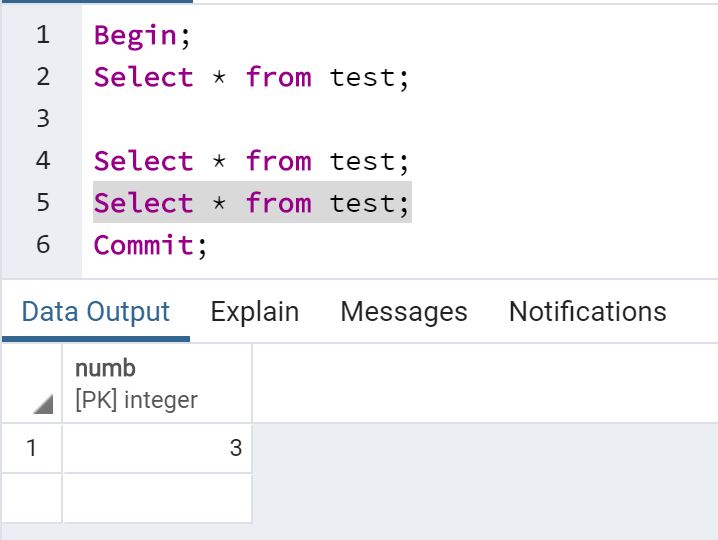
Бачимо, що дані в обох транзакціях поки однакові.

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

****

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

Тепер виконаємо в першій транзакції commit та select у другій.

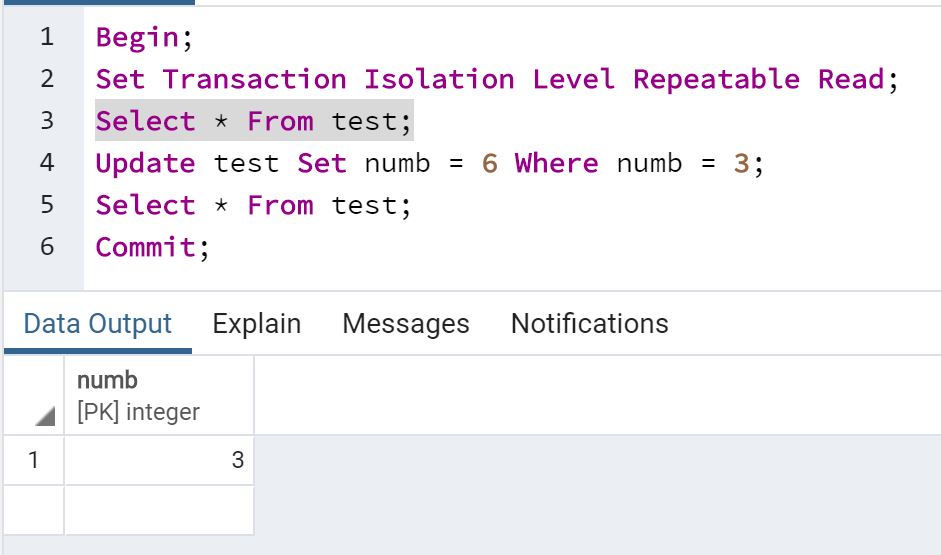
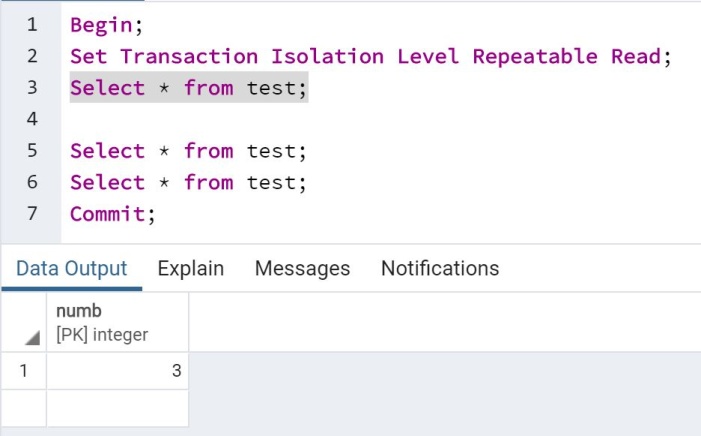
****

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

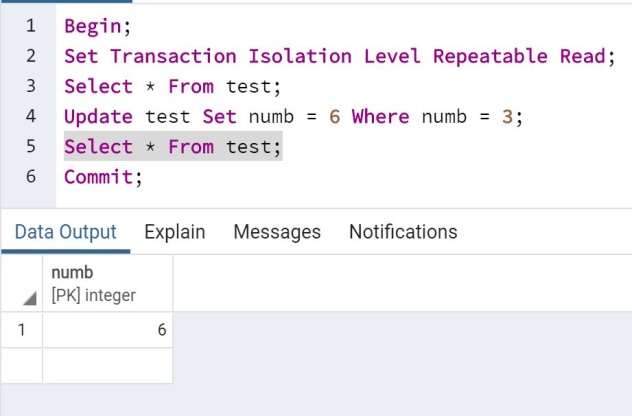
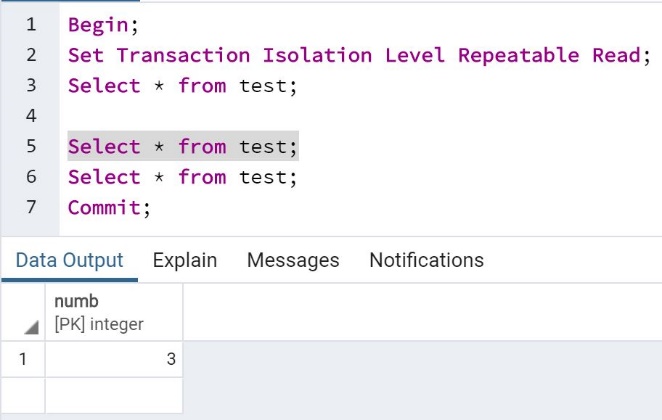
Для усунення цього ефекту треба перейти до рівня ізоляції транзакції “Repeatable read”.

Запустимо знову 2 транзакції, але вже встановимо їм рівень ізоляції транзакції “Repeatable read”.

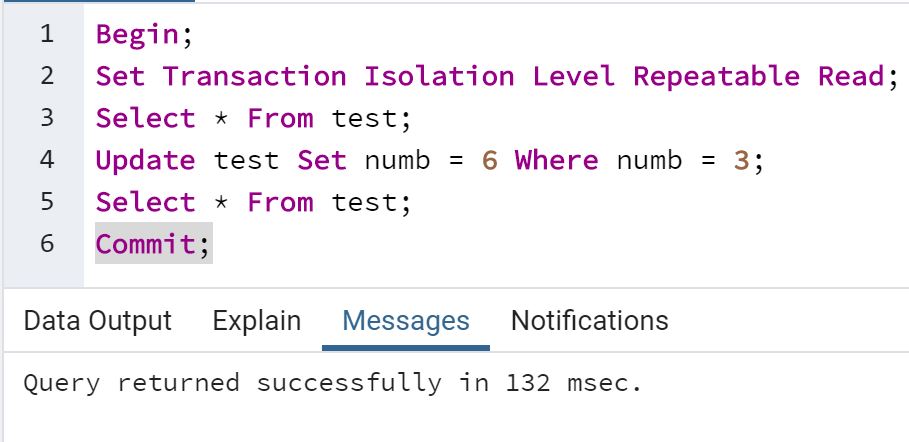
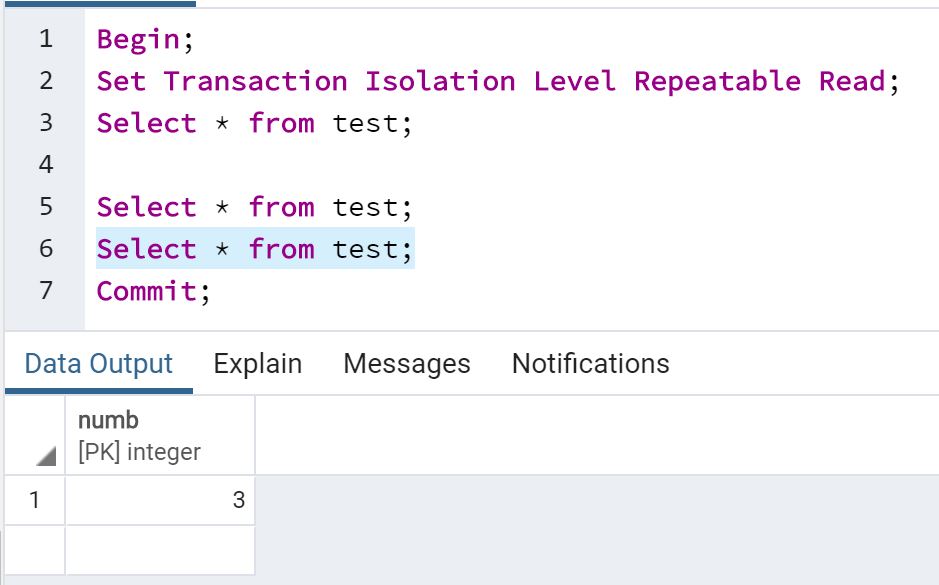
Бачимо, що на початку транзакцій дані в них відображуються однакові

****

Виконаємо в першій транзакції update та потім у обох транзакціях select.

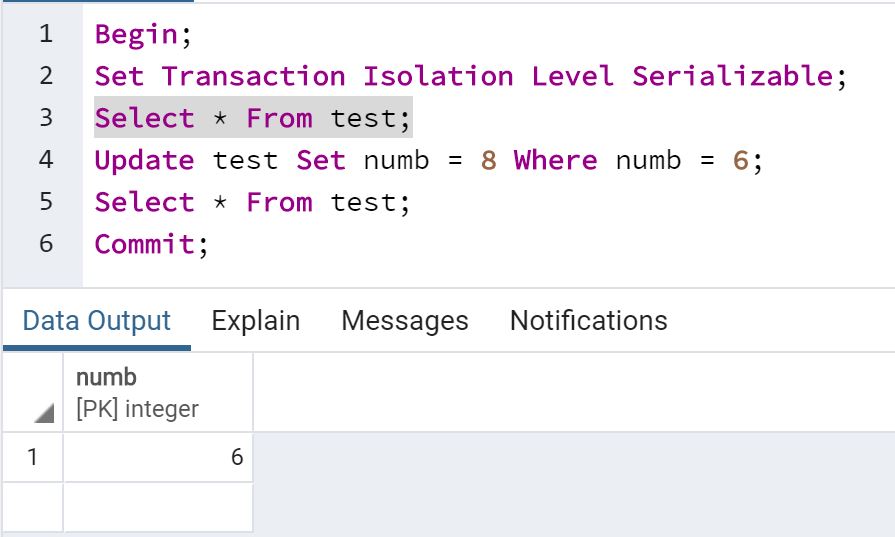
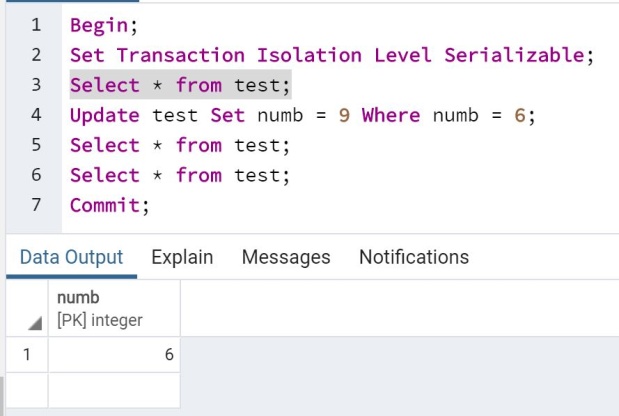
****

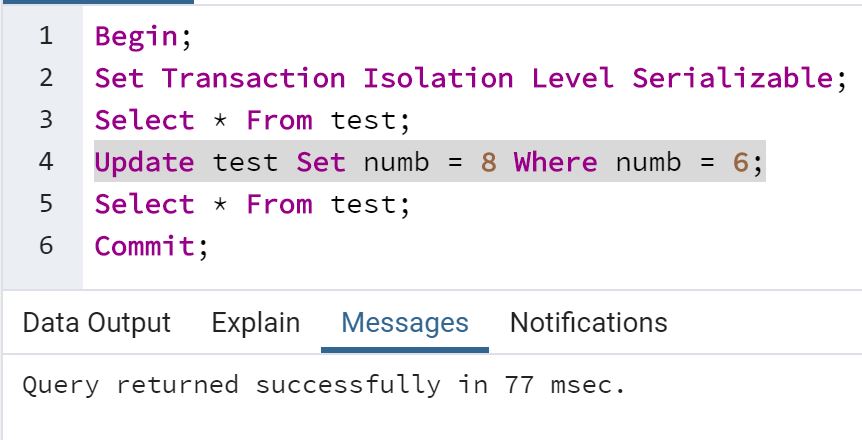
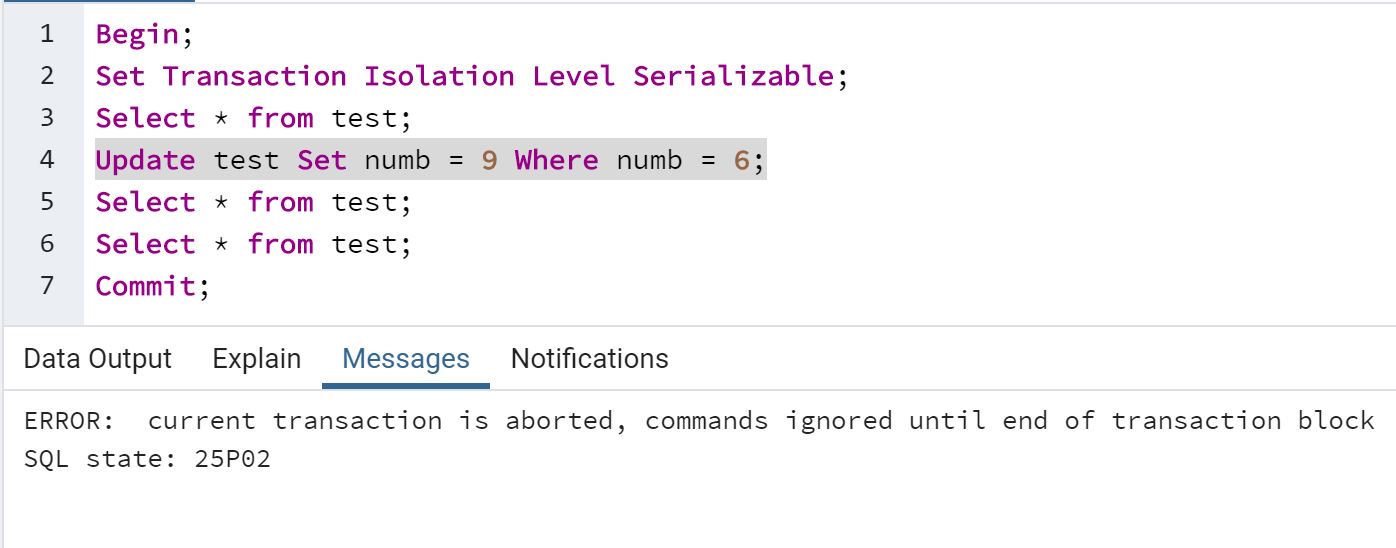
У першій транзакції бачимо оновлені дані , а в другій ще старі. Тепер зробимо commit першої транзакції та select в другій.

****

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

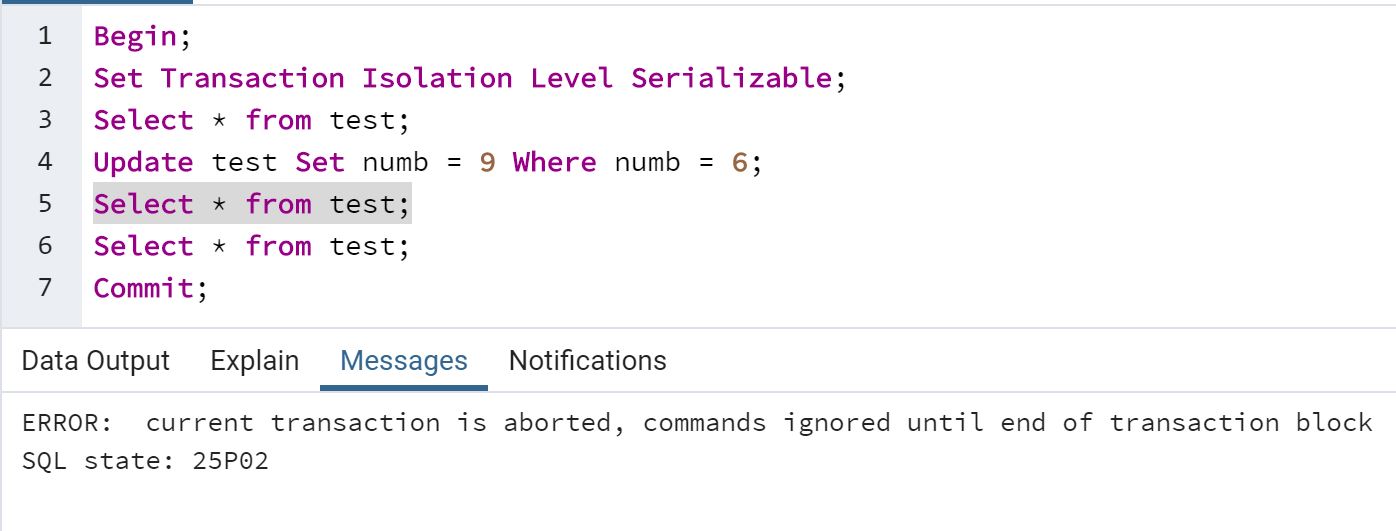
Тепер дослідимо рівень транзакції “Serializable”

Відкриємо 2 транзакції з рівнем ізоляції «Serializable», та виведемо дані таблиці test через select :

А тепер спробуємо зробити в обох транзакція update. Спочатку в першій, а потім в другій та побачимо, що буде.

Перша транзакція завершилась успішно, друга транзакція зависла до того моменту як її не завершили примусово, потім було видано повідомлення про помилку.

Після спроби виконати інші команду із транзакції також отримуємо дане повідомлення.



По-факту, тепер можна зробити тільки commit. Це сталося по причині того, що в другій транзакції була спроба модифікувати рядок, який вже був змінений у першій транзакції, що й призвело до конфлікту, бо рівень ізоляції транзакції “Serializable” не дозволяє цього.