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»**

**Тема****: «*Засоби оптимізації роботи СУБД PostgreSQL*»**

Виконав: студент IIІ курсу

ФПМ групи КВ-93

Рудницький Павло

Київ – 2021

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL

Постановка задачі

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 завдання:



Інформація про програму

Посилання на GitHub:

Мова програмування: Python 3.9

Використані бібліотеки: psycopg2 (для встановлення зв’язку з СУБД), time (для виміру часу пошуку), sys (для реалізації інтерфейсу)

**Модель «сутність-зв’язок» мережі магазинів**

Обрана предметна галузь передбачає зберігання товарів в кожному окремому філіалу мережі магазинів. Згідно цієї області для побудови бази даних було виділено наступні сутності:

1. Сутність “Shop” містить такі атрибути як ID, адреса магазину та контакти управляючого філіалом. Слугує для зберігання інформації про філіал
2. Сутність “Discount” містить такі атрибути як ID, відсоток знижки та термін її дії. Слугує зберігання інформації про знижку.
3. Сутність “Product\_discount” містить такі атрибути як ID, ідентифікатор продукту та ідентифікатор знижки. Слугує для зв’язування знижки та продукту.
4. Сутність “Product” містить такі атрибути як ID, ідентифікатор відділу, до якого належить продукт та назву продукту. Слугує для зберігання інформації про конкретний продукт.

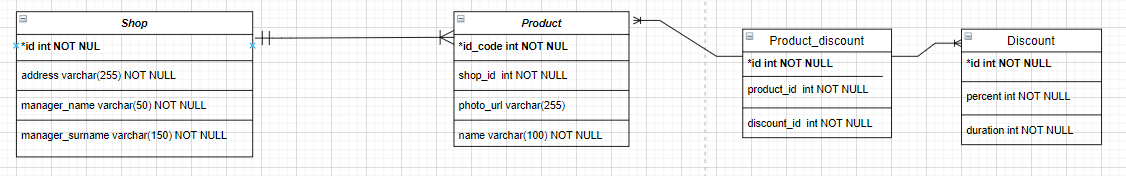
**

Рисунок 1 - Логічна модель предметної області «Мережа магазинів

***Опис структури БД «Мережа магазинів»***

|  |  |  |
| --- | --- | --- |
| Відношення | Атрибут | Розмір (тип) |
| ***Відношення “Shop”***  Вміщує інформацію про філіал магазину | **\*id** – унікальний ID філіалу магазина в БД, не допускає NULL  **address** – адреса магазину, не допускає NULL  **manager\_name** – ім’я управляючого магазином, не допускає NULL  **manager\_surname** – прізвище управляючого магазином, не допускає NULL | Числовий  Текстовий(255)  Текстовий(50)  Текстовий(150) |
| ***Відношення “Discount”***  Вміщує інформацію про знижку | **\*id** – унікальний ID знижки в БД, не допускає NULL  **percent –** відсоток знижки, не допускає NULL  **duration –** термін дії знижки, не допускає NULL | Числовий  Числовий  Числовий |
| ***Відношення “Product\_discount”***  Вміщує інформацію про продукт та знижку | **\*id** – унікальний ID таблиці в БД, не допускає NULL  **product\_id –** ідентифікатор продукту, не допускає NULL  **discount\_id –** ідентифікатор знижки, не допускає NULL | Числовий  Числовий  Числовий |
| ***Відношення “Product”***  Вміщує інформацію про конкретний продукт | **\*id** – унікальний ID продукту в БД, не допускає NULL  **photo\_url** – посилання на фото продукту  **name** – назва продукту, не допускає NULL | Числовий  Текстовий(255)  Текстовий(100) |

Таблиці бази даних у середовищі PgAdmin4

BEGIN;

CREATE TABLE IF NOT EXISTS public.discount

(

id integer NOT NULL,

percent integer NOT NULL,

duration integer NOT NULL,

PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public.product

(

id integer NOT NULL,

shop\_id integer NOT NULL,

photo\_url character varying(255) NOT NULL,

name character varying(100) NOT NULL,

PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public.product\_discount

(

id integer NOT NULL,

product\_id integer NOT NULL,

discount\_id integer NOT NULL,

PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public.shop

(

id integer NOT NULL,

address character varying(255) NOT NULL,

manager\_name character varying(50) NOT NULL,

manager\_surname character varying(150) NOT NULL,

PRIMARY KEY (id)

);

ALTER TABLE public.product

ADD FOREIGN KEY (shop\_id)

REFERENCES public.shop (id)

NOT VALID;

ALTER TABLE public.product\_discount

ADD FOREIGN KEY (discount\_id)

REFERENCES public.discount (id)

NOT VALID;

ALTER TABLE public.product\_discount

ADD FOREIGN KEY (product\_id)

REFERENCES public.product (id)

NOT VALID;

END;

Класи ORM у реалізованому модулі Model

class Shop(Orders):

\_\_tablename\_\_ = 'shop'

id = Column(Integer, primary\_key=True)

address = Column(String)

manager\_name = Column(String)

manager\_surname = Column(String)

def \_\_init\_\_(self, shop\_id, address, manager\_name, manager\_surname):

self.id = shop\_id

self.address = address

self.manager\_name = manager\_name

self.manager\_surname = manager\_surname

def \_\_repr\_\_(self):

return "{:>10}{:>35}{:>15}{:>40}" \

.format(self.id, self.address, self.manager\_name, self.manager\_surname)

class Product(Orders):

\_\_tablename\_\_ = 'product'

id = Column(Integer, primary\_key=True)

name = Column(String)

photo\_url = Column(String)

shop\_id = Column(Integer)

def \_\_init\_\_(self, product\_id, shop\_id, photo\_url, name):

self.id = product\_id

self.name = name

self.photo\_url = photo\_url

self.shop\_id = shop\_id

def \_\_repr\_\_(self):

return "{:>10}{:>10}{:>50}{:>100}" \

.format(self.id, self.shop\_id, self.name, self.photo\_url)

class Discount(Orders):

\_\_tablename\_\_ = 'discount'

id = Column(Integer, primary\_key=True)

percent = Column(Integer)

duration = Column(Integer)

def \_\_init\_\_(self, discount\_id, percent, duration):

self.id = discount\_id

self.percent = percent

self.duration = duration

def \_\_repr\_\_(self):

return "{:>10}{:>15}{:>15}" \

.format(self.id, self.percent, self.duration)

class Product\_discount(Orders):

\_\_tablename\_\_ = 'product\_discount'

id = Column(Integer, primary\_key=True)

discount\_id = Column(Integer)

product\_id = Column(Integer)

def \_\_init\_\_(self, key, product\_id, discount\_id):

self.id = key

self.discount\_id = discount\_id

self.product\_id = product\_id

def \_\_repr\_\_(self):

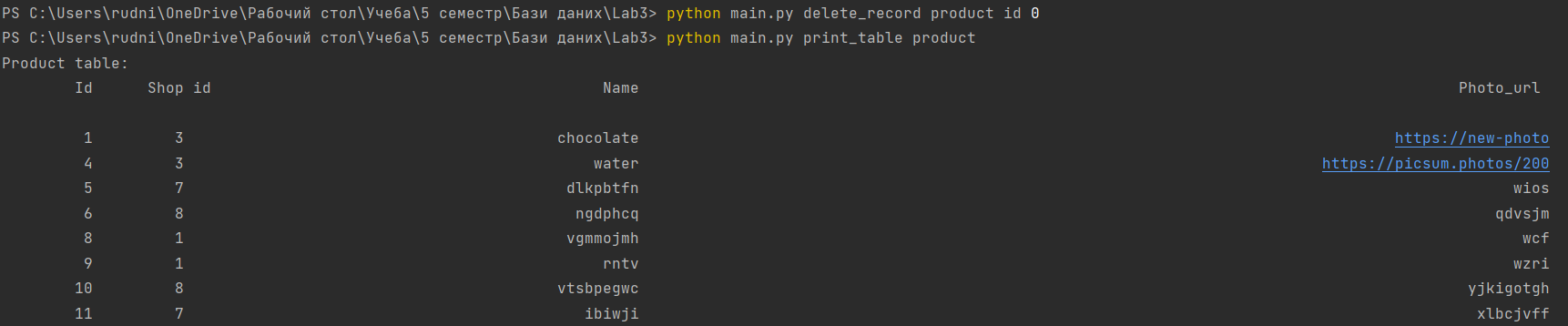
return "{:>10}{:>15}{:>15}" \

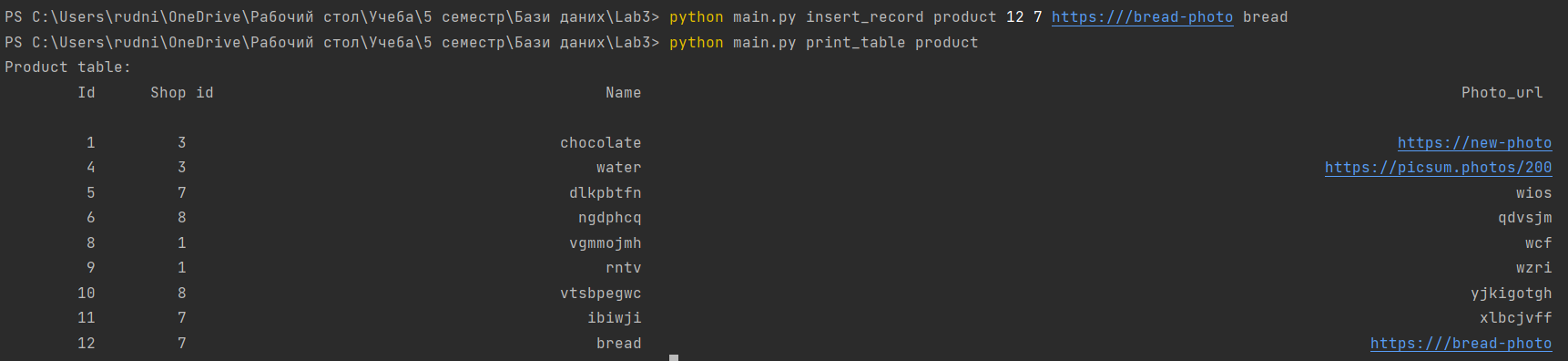
.format(self.id, self.product\_id, self.discount\_id)

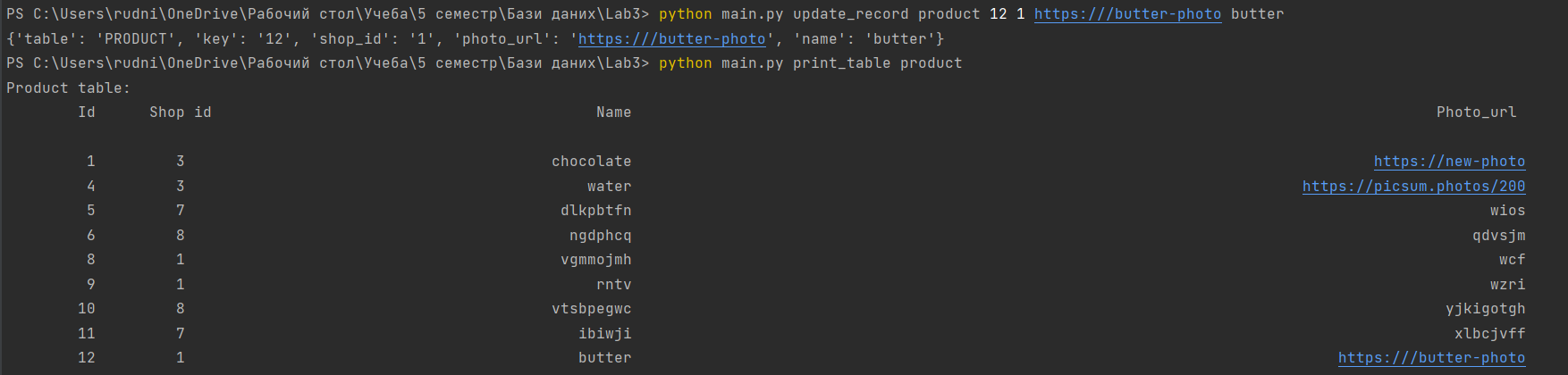
Запити у вигляді ORM

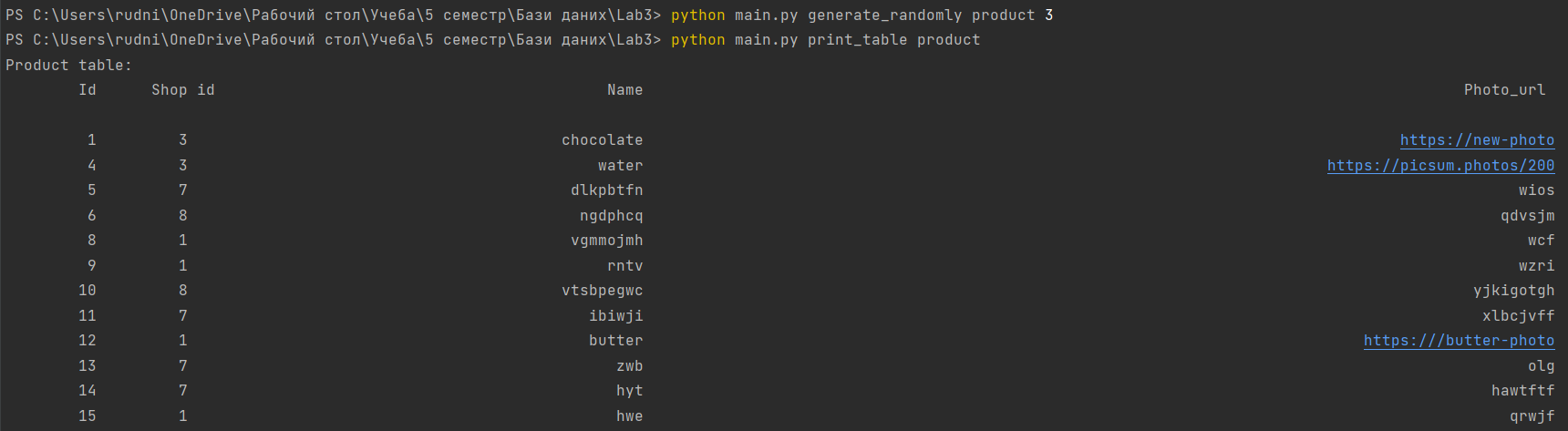
Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Product

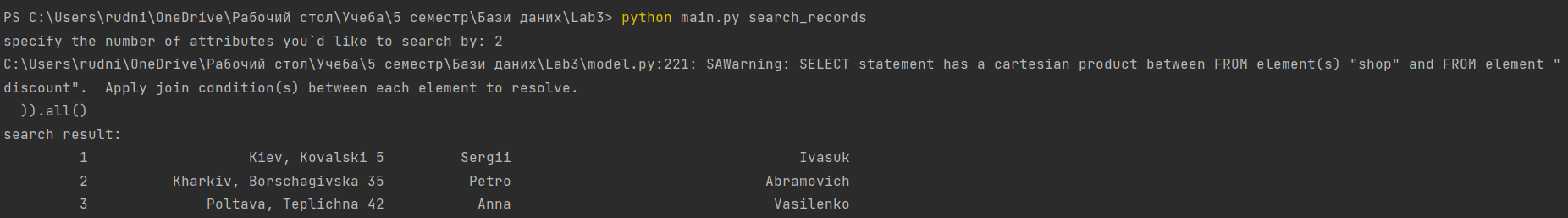
Початковий стан:

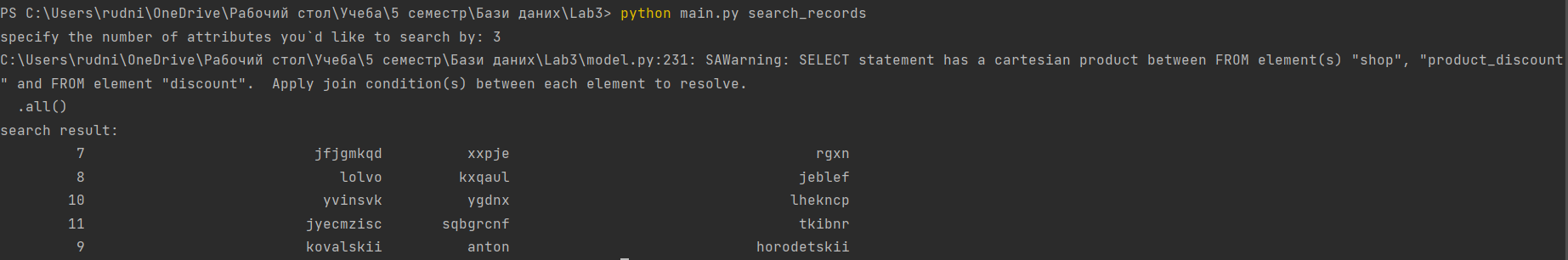
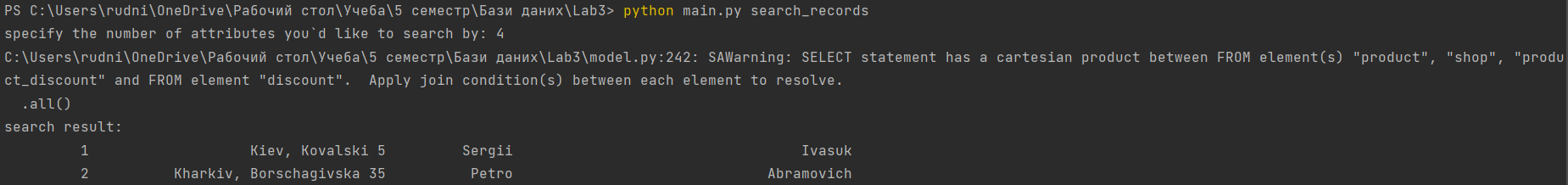
Видалення запису:

Вставка запису:

Редагування запису:

Вставка 3 рандомно згенерованих записів:

Пошук у таблицях:



# Завдання №2

Для тестування індексів було створено окремі таблиці у базі даних з 1000000 записів.

BTree

Індекс BTREE, він же B-дерево, придатний для даних, які можна відсортувати. Інакше кажучи, для типу даних мають бути визначені оператори «більше», «більше чи одно», «менше», «менше чи одно» і «рівно». Ті самі дані іноді можна сортувати різними способами, що повертає нас до концепції сімейства операторів. Як завжди, індексні записи B-дерева упаковані у сторінки. У листових сторінках ці записи містять індексовані дані (ключі) та посилання на рядки таблиці (TID-и); у внутрішніх сторінках кожен запис посилається на дочірню сторінку індексу та містить мінімальне значення ключа у цій сторінці.

****

Рисунок 2 - Аналіз виконання запиту1 без індексу

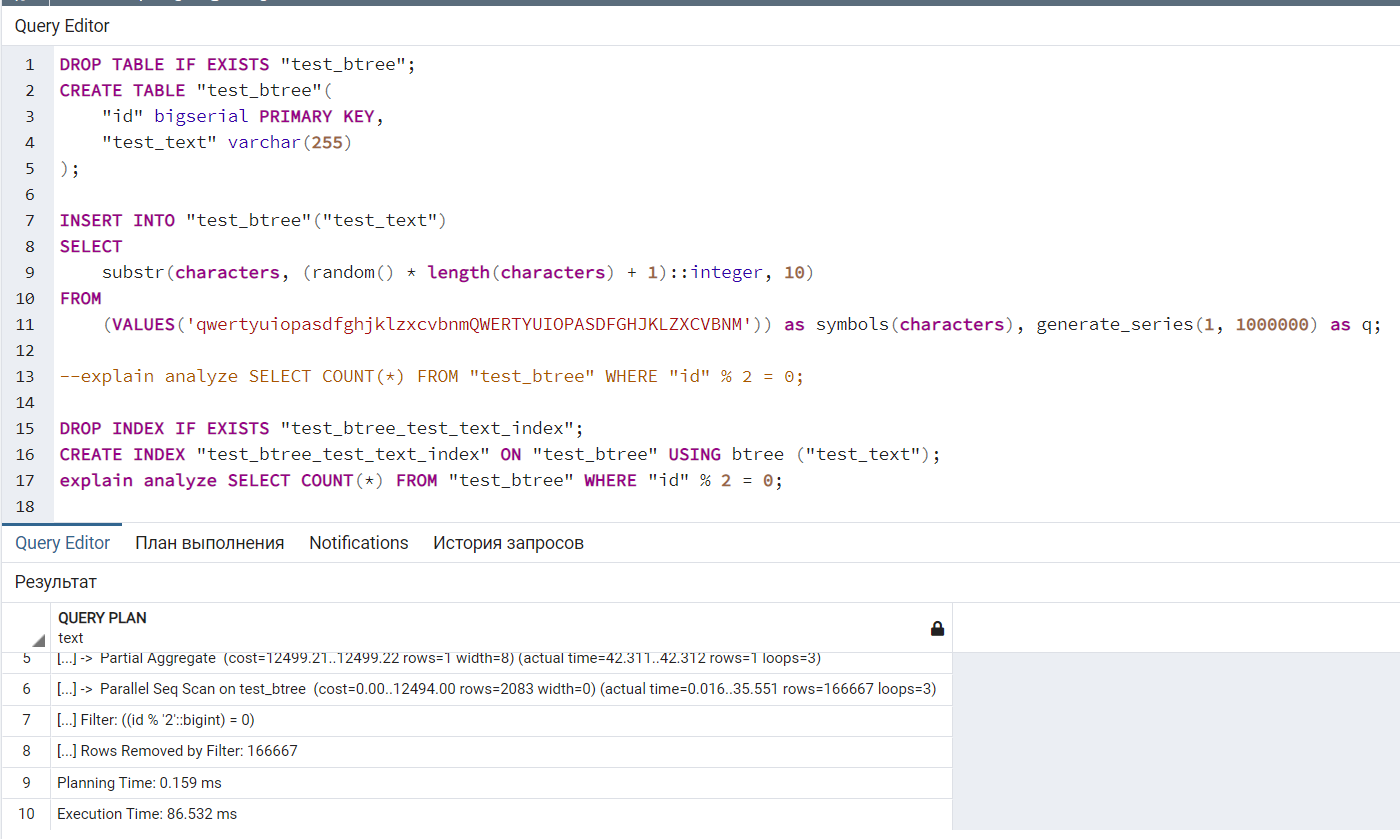
****

Рисунок 3 - Аналіз виконання запиту1 з індексом

Рисунок 4 - Аналіз виконання запиту2 без індексу

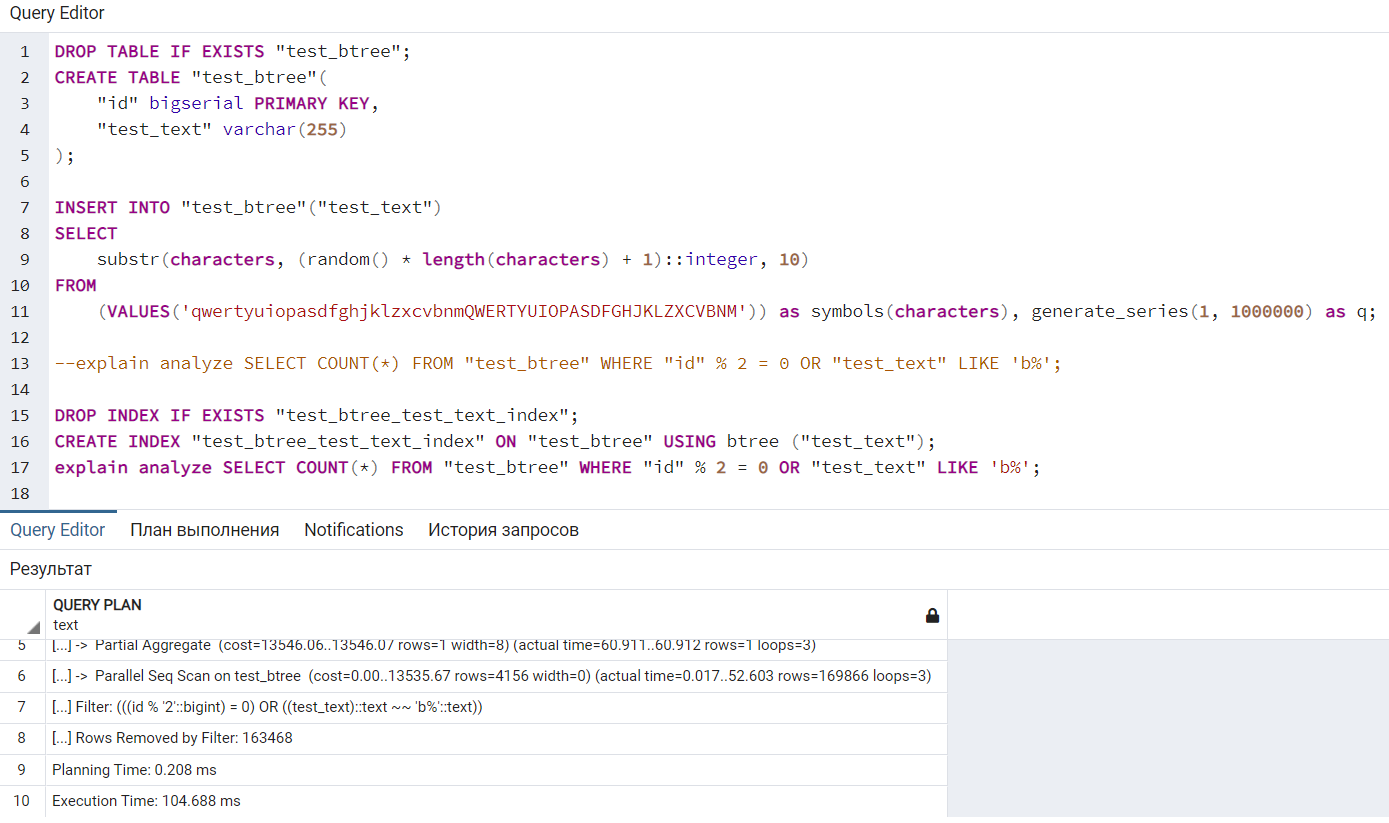
Рисунок 5 - Аналіз виконання запиту 2 з індексом

Рисунок 6 - Аналіз виконання запиту 3 без індексу

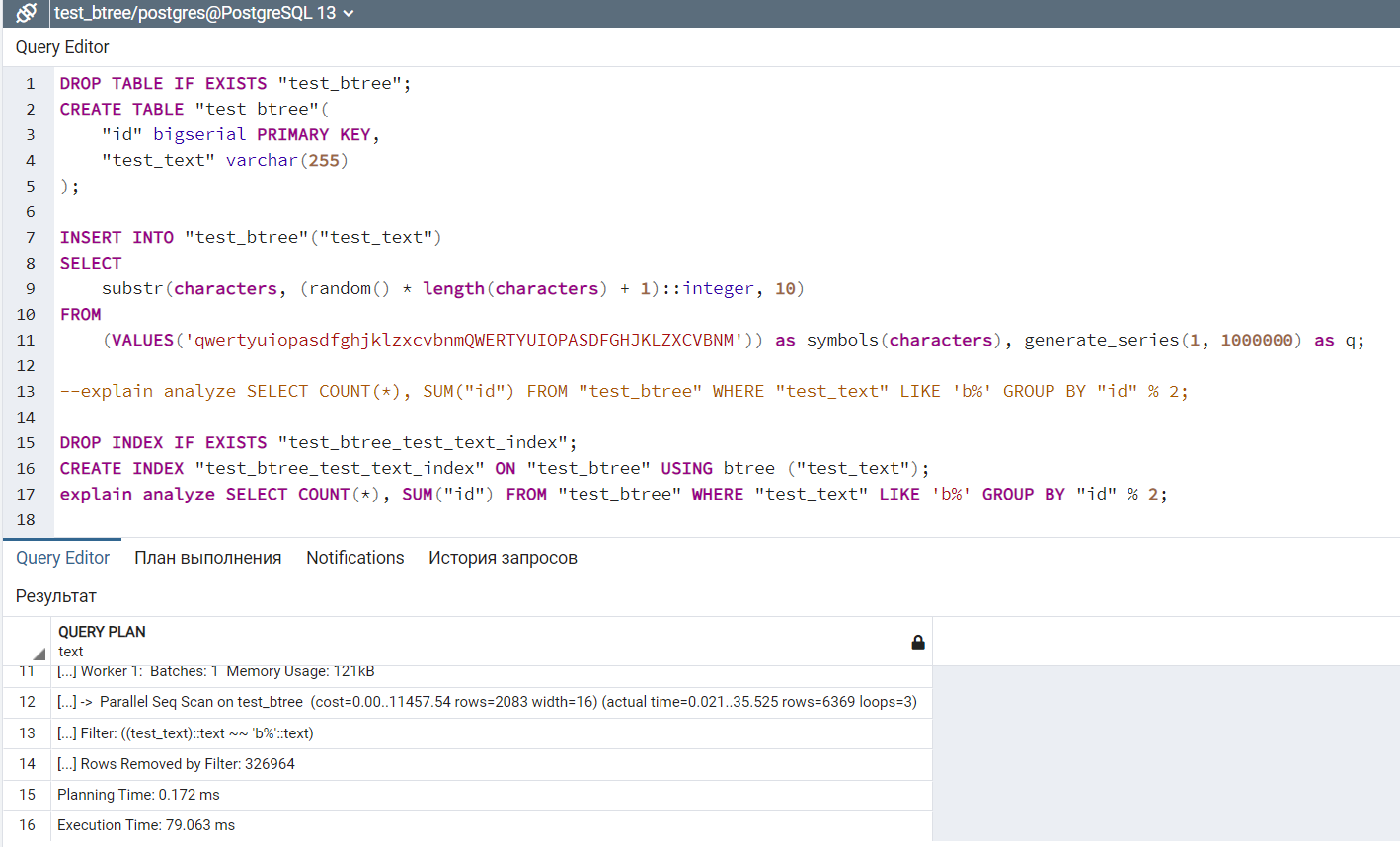
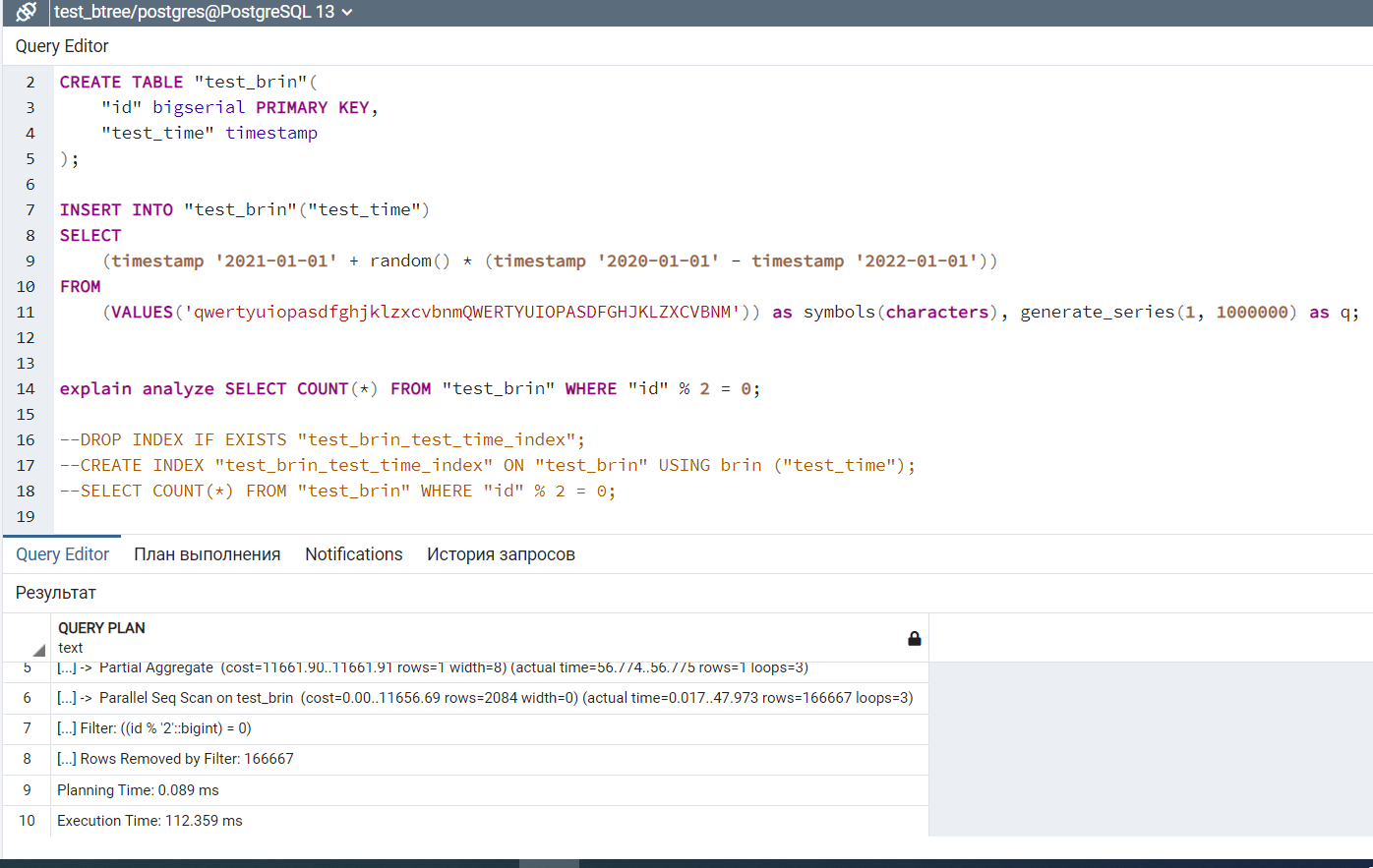


Рисунок 7 - Аналіз виконання запиту з індексом

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 таблиця, яка має дві колонки: t\_data типу timestamp without time zone (дата та час (без часового поясу)) і t\_number типу integer. Колонка t\_data проіндексована як BRIN. У таблицю занесено 1000000 записів.



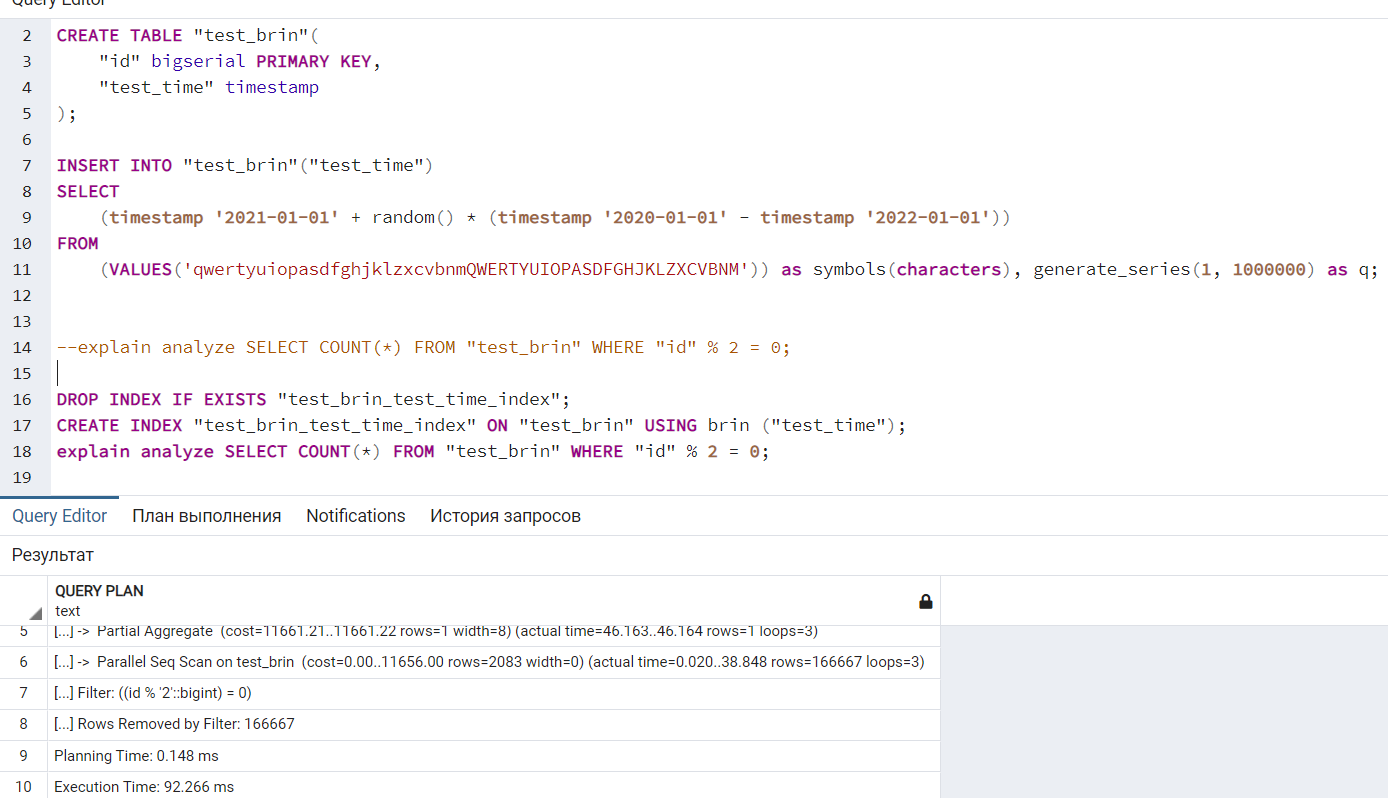
Рисунок 8 – Аналіз виконання запиту 1 без індексу

Рисунок 12 – Аналіз виконання запиту 1 з індексом

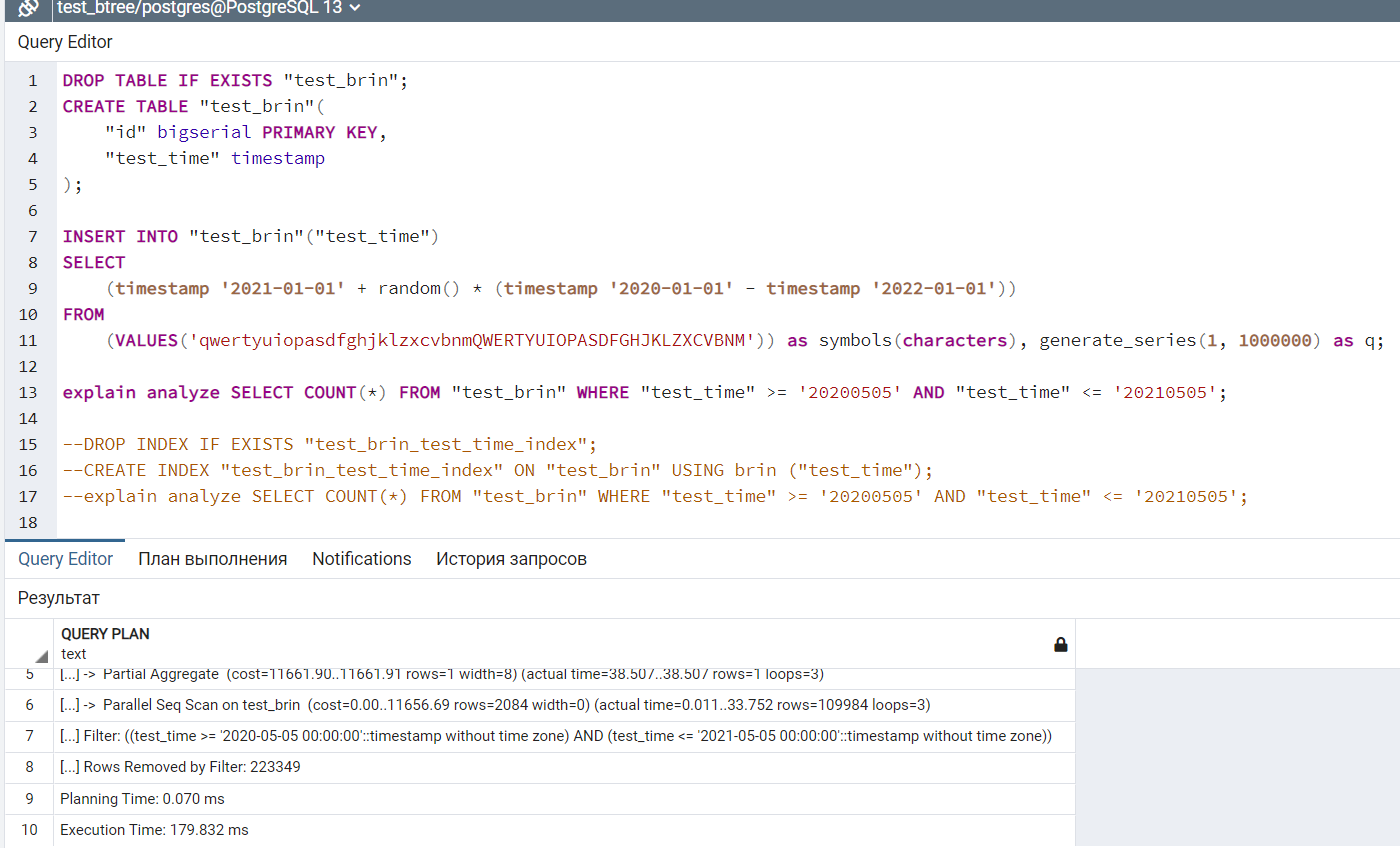
Рисунок 13 – Аналіз виконання запиту 2 без індексу

Рисунок 14 – Аналіз виконання запиту 2 з індексом

Рисунок 15 – Аналіз виконання запиту 3 без індексу 

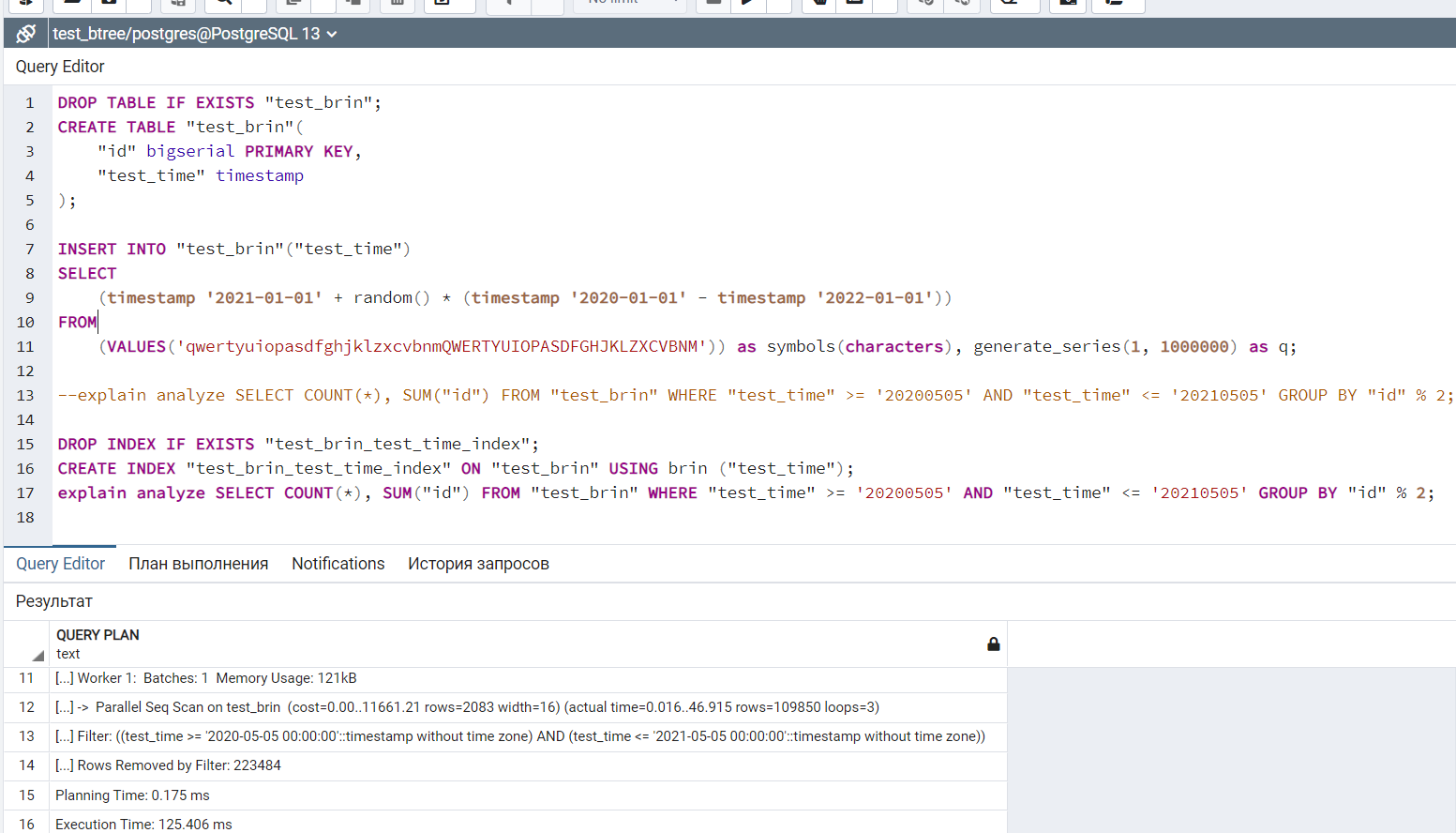


Рисунок 16 – Аналіз виконання запиту 3 з індексом

Завдання №3

Розробити тригер бази даних PostgreSQL.

Умова для тригера – before insert, delete.

**Таблиці:**

DROP TABLE IF EXISTS "reader";

CREATE TABLE "reader"(

"readerID" bigserial PRIMARY KEY, "readerName" varchar(255)

);

DROP TABLE IF EXISTS "readerLog";

CREATE TABLE "readerLog"(

"id" bigserial PRIMARY KEY, "readerLogID" bigint, "readerLogName" varchar(255)

);

**Тригер:**

CREATE OR REPLACE FUNCTION insert\_delete\_func() RETURNS TRIGGER as $$

DECLARE

CURSOR\_LOG CURSOR FOR SELECT \* FROM "readerLog";

row\_Log "readerLog"%ROWTYPE;

begin

IF old."readerID" % 2 = 0 THEN

INSERT INTO "readerLog"("readerLogID", "readerLogName") VALUES (old."readerID",

old."readerName");

UPDATE "readerLog" SET "readerLogName" = trim(BOTH 'x' FROM "readerLogName");

RETURN NEW;

ELSE

RAISE NOTICE 'readerID is odd';

FOR row\_log IN cursor\_log LOOP

UPDATE "readerLog" SET "readerLogName" = 'x' || row\_Log."readerLogName" || 'x' WHERE "id" = row\_log."id";

END LOOP;

RETURN NEW;

END IF;

END;

$$ language plpgsql;

**Ініціалізація тригера:**

create trigger test\_trigger before insert or delete on reader for each row

execute procedure insert\_delete\_func();

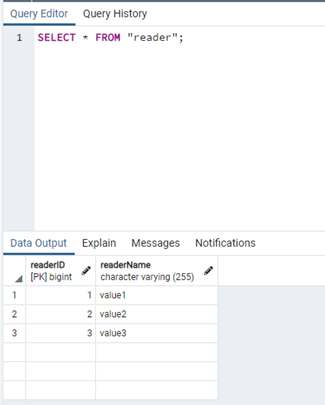
**Принцип тригера:**

Спрацьовує коли проходить оновлення чи видалення рядка з парним номером і заноситься у таблицю Logs.

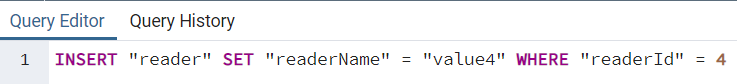
**Ініцалізуємо таблицю:**

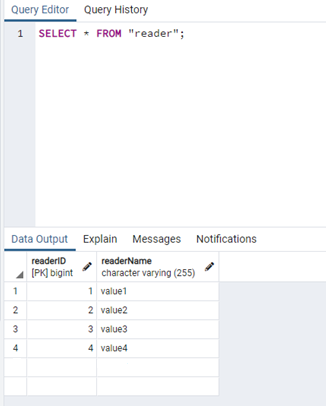
INSERT INTO "reader"("readerName")

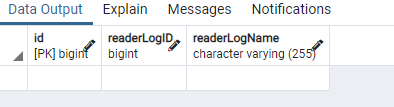
VALUES ('value1'), ('value2'), ('value3');



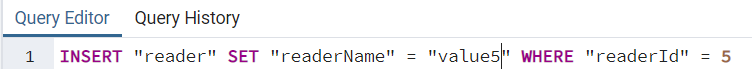
Робимо запит:

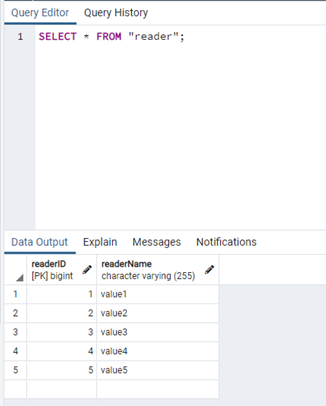


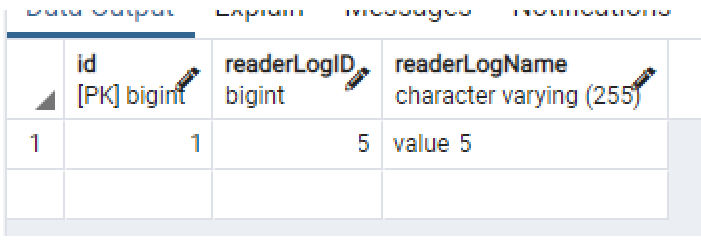
Після виконання запиту, бачимо, що рядок за номером 4 було додано та не був доданий рядок у logs.



Виконуємо такий же запит для 5 рядка:

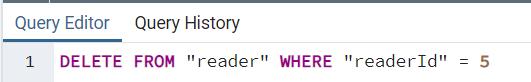


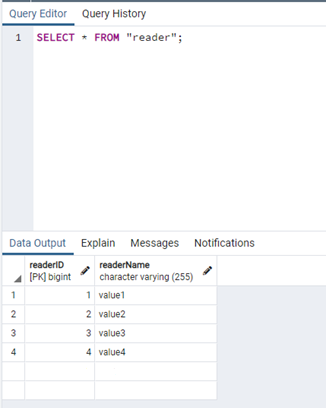


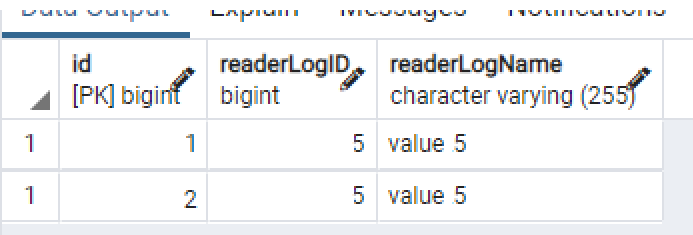


Як бачимо, тригер спрацював і зробив запис до Logs.

Робимо запит на видалення за номером рядка 5:







В цьому випадку тригер також спрацював коректно