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»**

Виконала: студентка IIІ курсу

ФПМ групи КВ-93

Коломієць Анна

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;
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Інформація про програму

Посилання на GitHub: <https://github.com/AnnaKolomiietss/DB_lab2>

|  |  |  |
| --- | --- | --- |
| № Варіанта | Види індексів | Умови для тригера |
| 11 | Gin, Hash | Before update, delete |

**Завдання 1**

Відомості про предметну галузь «Аптека» з лабораторної роботи №1

Опис структури БД «Аптека»

Предметна галузь «Аптека» описує схему замовлень лікарських засобів. Згідно цієї області для побудови бази даних було виділено наступні сутності:

1. Category: вміщує атрибути ID, name та description. Слугує для зберігання інформації про певну категорію ліків;

2. Pill: вміщує атрибути ID, name, price, manufacturer\_id. Слугує для зберігання інформації про конкретний лікарський засіб;

3. Category\_pill: вміщує ID препарату та ID категорії. Слугує для зв’язування лікарського засобу та її категорії;

4. Manufacturer: вміщує атрибути ID, name, country, email. Слугує для зберігання інформації про виробника.

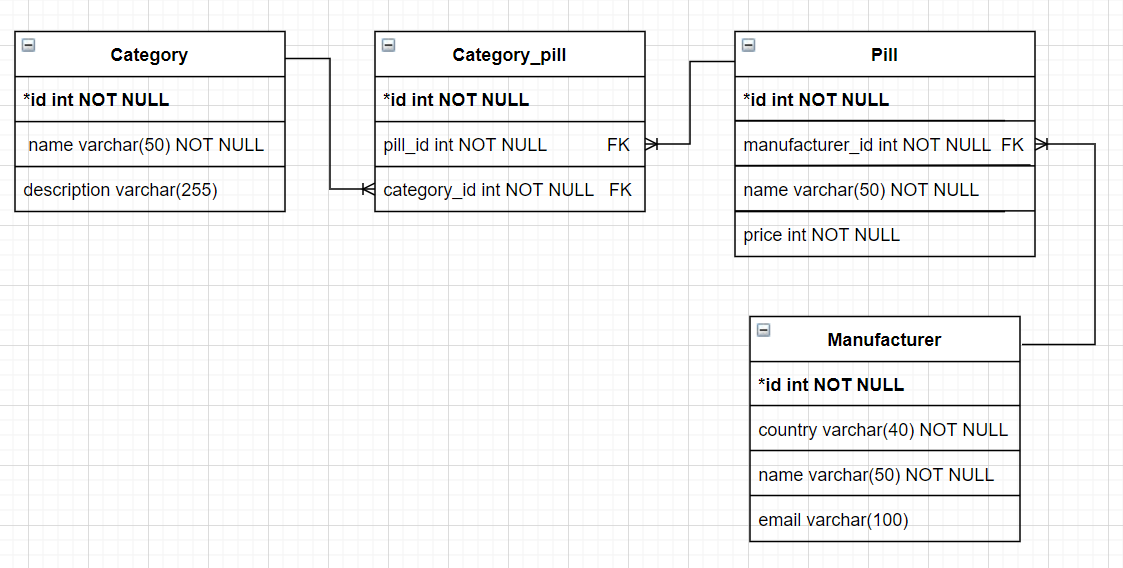


Рисунок 1 - Логічна модель предметної області «Аптека»

**Таблиці бази даних у середовищі PgAdmin4**

BEGIN;

CREATE TABLE IF NOT EXISTS public."Category"

(

id integer NOT NULL,

name character varying(50) COLLATE pg\_catalog."default" NOT NULL,

description character varying(255) COLLATE pg\_catalog."default",

CONSTRAINT "Category \_pkey" PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Category\_pill"

(

id integer NOT NULL,

pill\_id integer NOT NULL,

category\_id integer NOT NULL,

CONSTRAINT "Category\_pill\_pkey" PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Manufacturer"

(

id integer NOT NULL,

name character varying(50) COLLATE pg\_catalog."default" NOT NULL,

country character varying(40) COLLATE pg\_catalog."default" NOT NULL,

email character varying(100) COLLATE pg\_catalog."default",

CONSTRAINT "Manyfacturer \_pkey" PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Pill"

(

id integer NOT NULL,

manufacturer\_id integer NOT NULL,

name character varying(60) COLLATE pg\_catalog."default" NOT NULL,

price integer NOT NULL,

CONSTRAINT "Pill\_pkey" PRIMARY KEY (id)

);

ALTER TABLE IF EXISTS public."Category\_pill"

ADD CONSTRAINT fk\_category\_pill\_category FOREIGN KEY (category\_id)

REFERENCES public."Category" (id) MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION

NOT VALID;

ALTER TABLE IF EXISTS public."Category\_pill"

ADD CONSTRAINT fk\_category\_pill\_pill FOREIGN KEY (pill\_id)

REFERENCES public."Pill" (id) MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION

NOT VALID;

ALTER TABLE IF EXISTS public."Pill"

ADD CONSTRAINT fk\_pill\_manufacturer FOREIGN KEY (manufacturer\_id)

REFERENCES public."Manufacturer" (id) MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION

NOT VALID;

END;

**Класи ORM у реалізованому модулі Model**

class Category(Orders):

\_\_tablename\_\_ = 'Category'

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

name = Column(String)

description = Column(String)

def \_\_init\_\_(self, key, name, description):

self.id = key

self.name = name

self.description = description

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

return "{:>10}{:>35}{:>80}" \

.format(self.id, self.name, self.description)

class Category\_pill(Orders):

\_\_tablename\_\_ = 'Category\_pill'

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

pill\_id = Column(Integer, ForeignKey('Pill.id'))

category\_id = Column(Integer, ForeignKey('Category.id'))

pills = relationship('Pill')

categories = relationship('Category')

def \_\_init\_\_(self, key, pill\_id, category\_id):

self.id = key

self.pill\_id = pill\_id

self.category\_id = category\_id

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

return "{:>10}{:>20}{:>20}" \

.format(self.id, self.pill\_id, self.category\_id)

class Pill(Orders):

\_\_tablename\_\_ = 'Pill'

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

manufacturer\_id = Column(Integer, ForeignKey('Manufacturer.id'))

name = Column(String)

price = Column(Integer)

manufacturers = relationship('Manufacturer')

def \_\_init\_\_(self, key, manufacturer\_id, name, price):

self.id = key

self.manufacturer\_id = manufacturer\_id

self.name = name

self.price = price

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

return "{:>10}{:>20}{:>25}{:>10}" \

.format(self.id, self.manufacturer\_id, self.name, self.price)

class Manufacturer(Orders):

\_\_tablename\_\_ = 'Manufacturer'

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

country = Column(String)

name = Column(String)

email = Column(String)

def \_\_init\_\_(self, key, country, name, email):

self.id = key

self.country = country

self.name = name

self.email = email

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

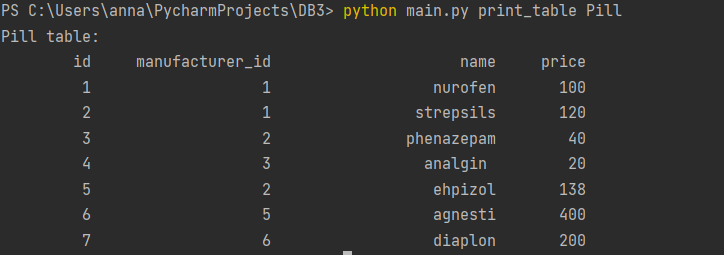
return "{:>10}{:>20}{:>20}{:>40}" \

.format(self.id, self.country, self.name, self.email)

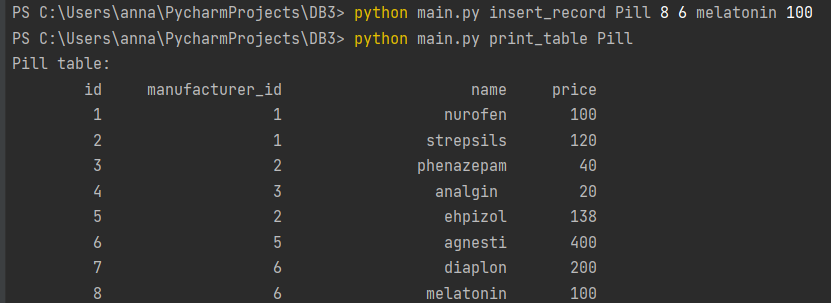
**Запити у вигляді ORM**

Продемонструємо вставку, редагування, виучення даних на прикладі таблиці Pill

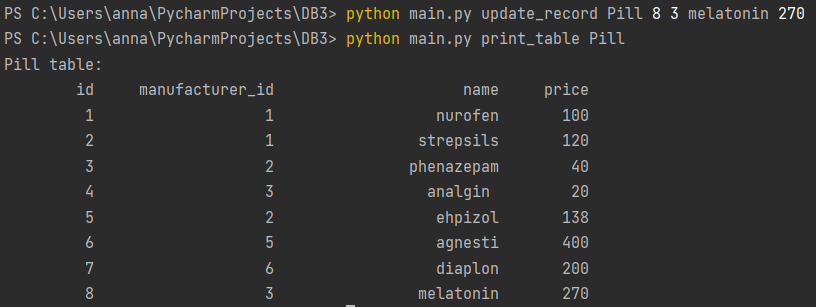
Початковий стан:



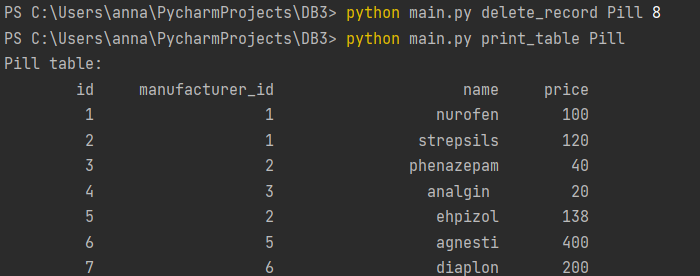
Вставка запису з id 8 у таблицю Pill:



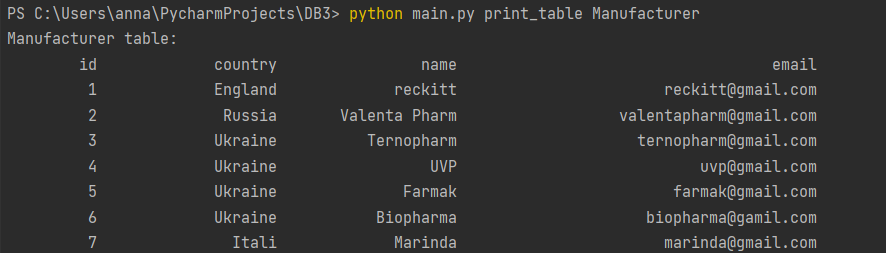
Редагування запису у таблиці Pill з id 8:



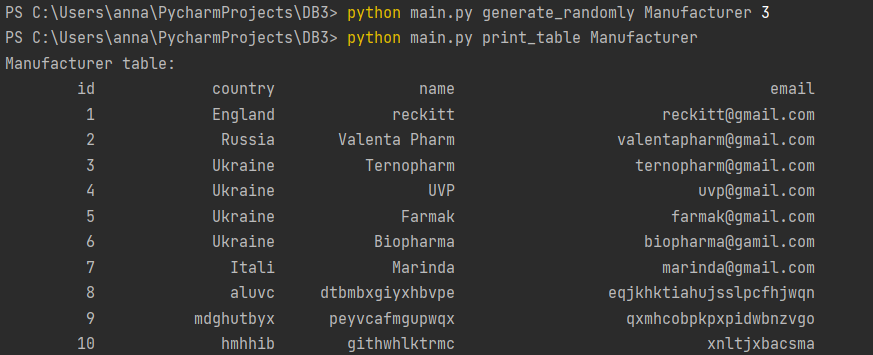
Видалення запису з таблиці Pill з id 8:



Запит на генерацію даних продемонструємо на прикладі таблиці Manufacturer  
Початковий стан:

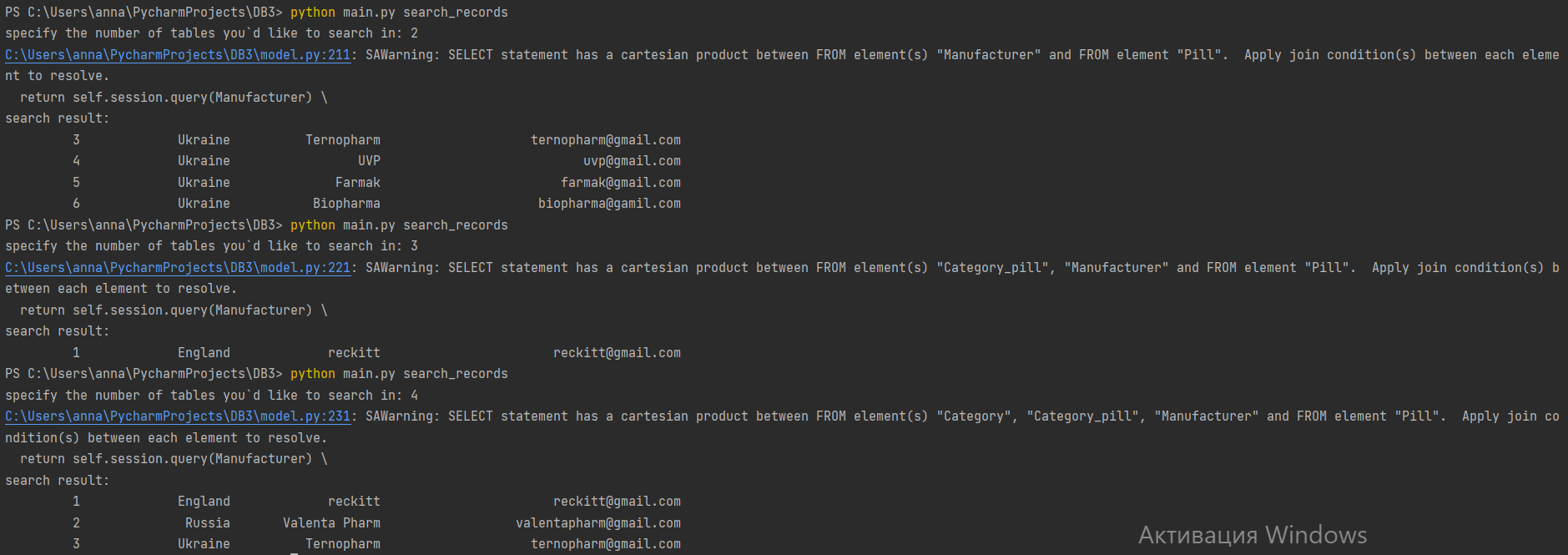


Вставка 3-х випадково згенерованих записів до таблиці Manufacturer:



Пошук за трьома атрибутами у двох таблицях, за трьома атрибутами у трьох

таблицях, за чотирма атрибутами у чотирьох таблицях (виводяться відповідні

записи з таблиці Manufacturer):

**Завдання №2**

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

GIN

Основна сфера застосування методу GIN - прискорення повнотекстового пошуку. GIN призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями, а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах. Він працює з типами даних, значення яких не є атомарними, а складаються з елементів. У ньому індексуються не самі значення, а окремі елементи; кожен елемент посилається ті значення, у яких зустрічається. Індекс GIN зберігає набір пар (ключ), де список появи — це набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з’являється багато разів. Пошук за такою структурою є набагато ефективнішим, проте процес додавання нового документа повільніший, т.к. зміни вносяться у велику кількість записів індексу. Цей індекс може взаємодіяти тільки з полем типу tsvector.

Стверення таблиці БД:

DROP TABLE IF EXISTS "gin\_test";

CREATE TABLE "gin\_test"("id" bigserial PRIMARY KEY, "string" text, "gin\_vector" tsvector);

INSERT INTO "gin\_test"("string") SELECT substr(characters, (random() \* length(characters) + 1)::integer, 10) FROM (VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters), generate\_series(1, 1000000) as q;

UPDATE "gin\_test" set "gin\_vector" = to\_tsvector("string");

Запити для тестування:

SELECT COUNT(\*) FROM "gin\_test" WHERE "id" % 2 = 0;

SELECT COUNT(\*) FROM "gin\_test" WHERE ("gin\_vector" @@ to\_tsquery('bnm'));

SELECT SUM("id") FROM "gin\_test" WHERE ("gin\_vector" @@ to\_tsquery('QWERTYUIOP')) OR ("gin\_vector" @@ to\_tsquery('bnm'));

SELECT MIN("id"), MAX("id") FROM "gin\_test" WHERE ("gin\_vector" @@ to\_tsquery('bnm')) GROUP BY "id" % 2;

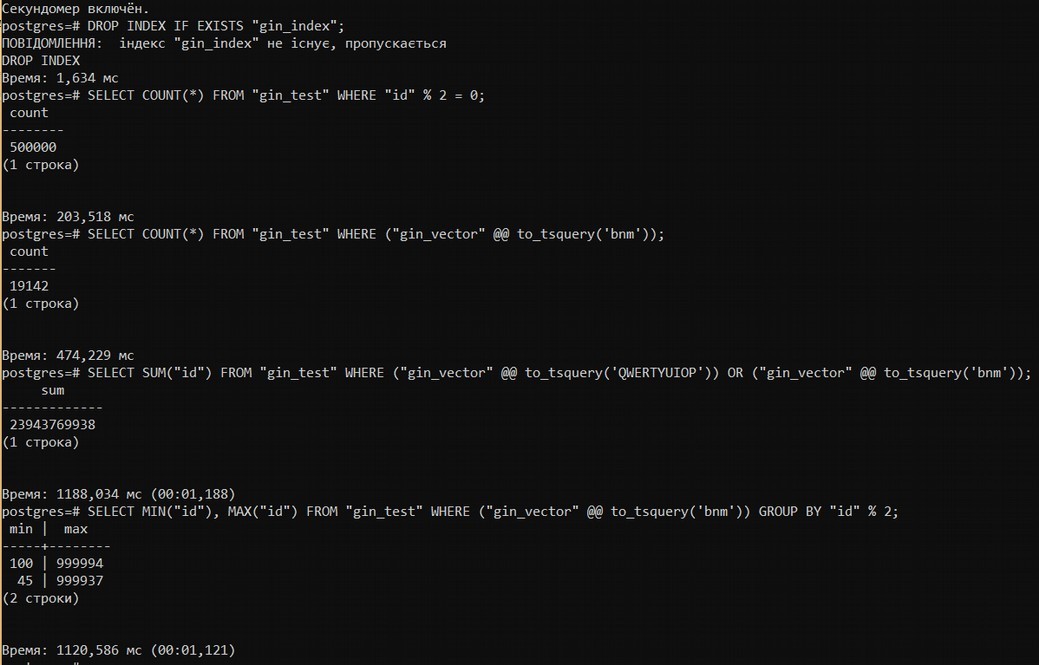
Створення індексу:

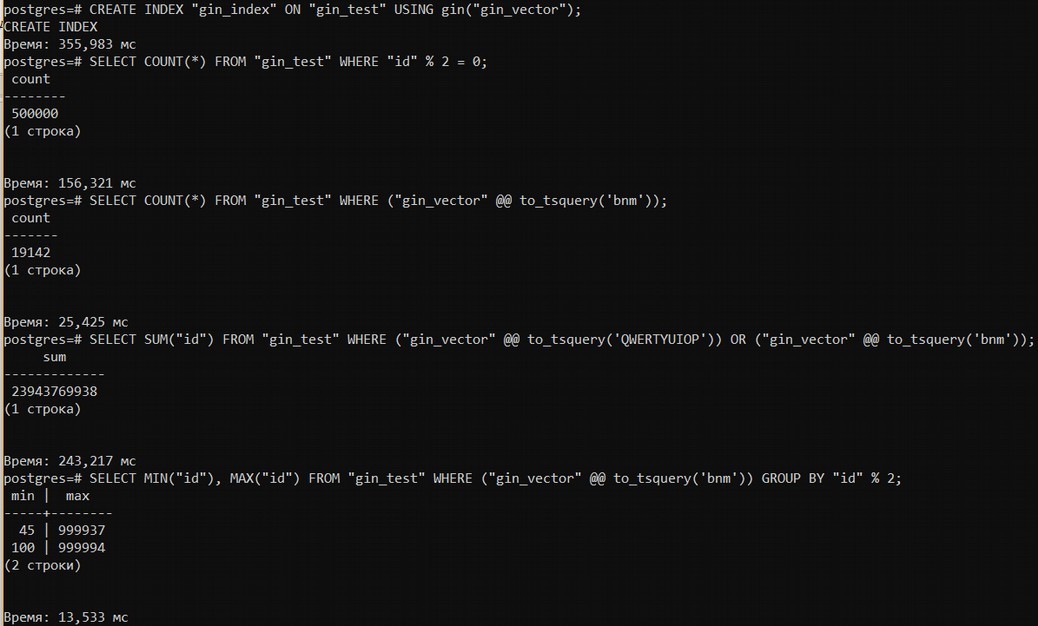
DROP INDEX IF EXISTS "gin\_index";

CREATE INDEX "gin\_index" ON "gin\_test" USING gin("gin\_vector");

Результати і час виконання на скріншотах з psql.exe

Запити без індексування:



Запити з індексуванням:

У заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). Це відбувається завдяки головній особливості індексування GIN: кожне значення шуканого ключа зберігається один раз і запит іде не по всій таблиці, а лише по тим даним, що містяться у списку появи цього ключа. Для даних типу numeric даний тип індексування використовувати недоцільно і неможливо.

Hash

Хеш-індекси в PostgreSQL використовують форму структури даних хеш-таблиці (використовують хеш-функцію). Хеш-коди поділені на обмежену кількість комірок. Коли до індексу додається нове значення, PostgreSQL застосовує хеш- функцію до значення і поміщає хеш-код і вказівник на кортеж у відповідну комірку. Коли відбувається запит за допомогою індексу хешування, PostgreSQL бере значення індексу і застосовує хеш-функцію, щоб визначити, яка комірка може містити потрібні дані.

Стверення таблиці БД:

DROP TABLE IF EXISTS "hash\_test";

CREATE TABLE "hash\_test"("id" bigserial PRIMARY KEY, "time" timestamp);

INSERT INTO "hash\_test"("time") SELECT (timestamp '2021-01-01' + random() \* (timestamp '2020-01-01' - timestamp '2022-01-01')) FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))as symbols(characters), generate\_series(1, 1000000) as q;

Запити для тестування:

SELECT COUNT(\*) FROM "hash\_test" WHERE "id" % 2 = 0;

SELECT COUNT(\*) FROM "hash\_test" WHERE "time" >= '20191001';

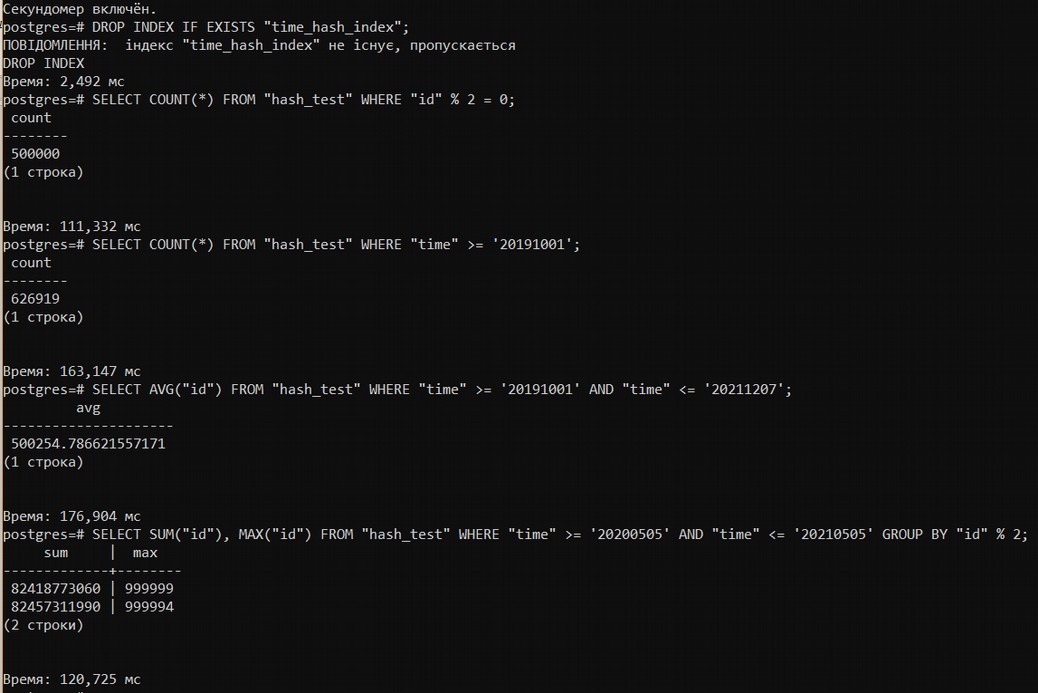
SELECT AVG("id") FROM "hash\_test" WHERE "time" >= '20191001' AND "time" <= '20211207'; SELECT SUM("id"), MAX("id") FROM "hash\_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;

Створення індексу:

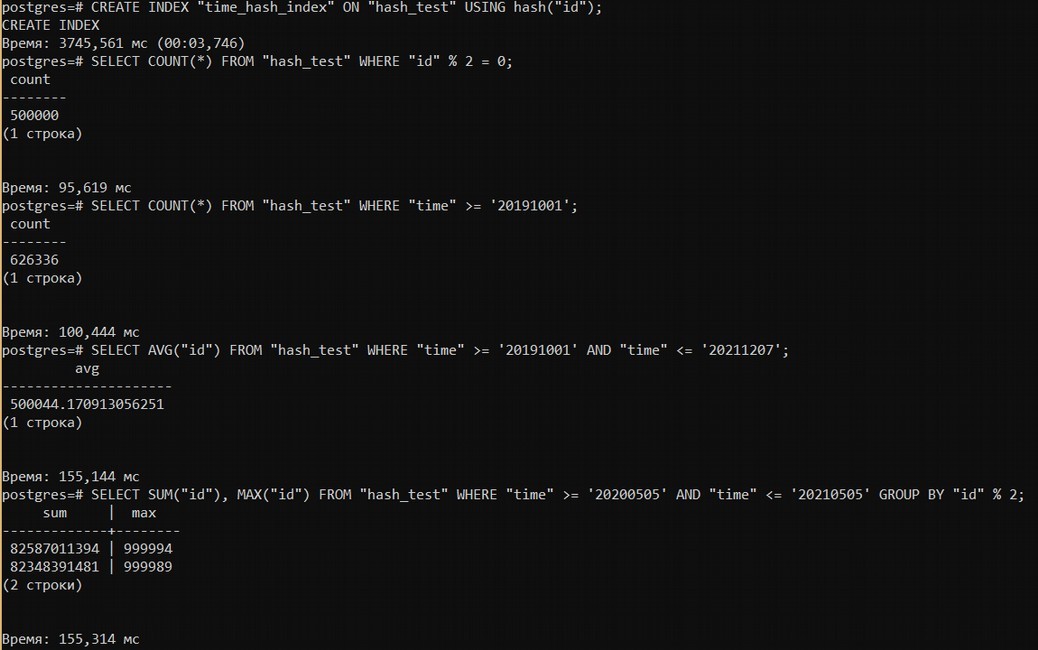
DROP TABLE IF EXISTS "hash\_test";

CREATE INDEX "time\_hash\_index" ON "hash\_test" USING hash("id");

Результати і час виконання на скріншотах з psql.exe

Запити без індексування:

Запити з індексуванням:



Очевидно, що індексування за допомогою hash не значно пришвидшує пошук даних у таблиці, а іноді навіть показує гірші результати, ніж запити без ідексування. Це випливає з того, що це один із найпримітивніших методів індексування і для пошуку потрібних даних алгоритм все одно проходить через усі записи у таблиці (на відміну від GIN). Він ефективний при застосуванні до поля числового типу.

**Завдання 3**

Для тестування тригера було створено дві таблиці:

DROP TABLE IF EXISTS "trigger\_test";

CREATE TABLE "trigger\_test"(

"trigger\_testID" bigserial PRIMARY KEY,

"trigger\_testName" text

);

DROP TABLE IF EXISTS "trigger\_test\_log";

CREATE TABLE "trigger\_test\_log"(

"id" bigserial PRIMARY KEY,

"trigger\_test\_log\_ID" bigint,

"trigger\_test\_log\_name" text

);

Початкові дані у таблицях:

INSERT INTO "trigger\_test"("trigger\_testName")

VALUES ('trigger\_test1'), ('trigger\_test2'), ('trigger\_test3'), ('trigger\_test4'), ('trigger\_test5'), ('trigger\_test6'), ('trigger\_test7'), ('trigger\_test8'), ('trigger\_test9'), ('trigger\_test10');

Команди, що ініціюють виконання тригера:

CREATE TRIGGER "before\_update\_delete\_trigger"

BEFORE UPDATE OR DELETE ON "trigger\_test"

FOR EACH ROW

EXECUTE procedure before\_update\_delete\_func();

Текст тригера:

CREATE OR REPLACE FUNCTION before\_update\_delete\_func() RETURNS TRIGGER as $trigger$ DECLARE

CURSOR\_LOG CURSOR FOR SELECT \* FROM "trigger\_test\_log";

row\_ "trigger\_test\_log"%ROWTYPE;

BEGIN

IF old."trigger\_testID" % 2 = 0 THEN

IF old."trigger\_testID" % 3 = 0 THEN

RAISE NOTICE 'trigger\_testID is multiple of 2 and 3';

FOR row\_ IN CURSOR\_LOG LOOP

UPDATE "trigger\_test\_log" SET "trigger\_test\_log\_name" = '\_' || row\_."trigger\_test\_log\_name" || '\_log' WHERE "id" = row\_."id";

END LOOP;

RETURN OLD;

ELSE

RAISE NOTICE 'trigger\_testID is even';

INSERT INTO "trigger\_test\_log"("trigger\_test\_log\_ID","trigger\_test\_log\_name") VALUES (old."trigger\_testID", old."trigger\_testName");

UPDATE "trigger\_test\_log" SET "trigger\_test\_log\_name" = trim(BOTH '\_log' FROM "trigger\_test\_log\_name");

RETURN NEW;

END IF;

ELSE

RAISE NOTICE 'trigger\_testID is odd';

FOR row\_ IN CURSOR\_LOG LOOP

UPDATE "trigger\_test\_log" SET "trigger\_test\_log\_name" = '\_' || row\_."trigger\_test\_log\_name" || '\_log' WHERE "id" = row\_."id";

END LOOP;

RETURN OLD;

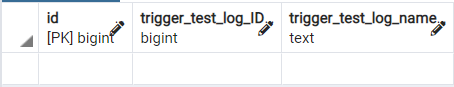
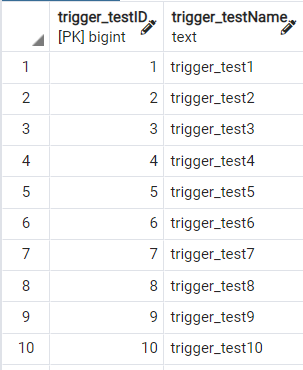
END IF;

END;

$trigger$ LANGUAGE plpgsql;

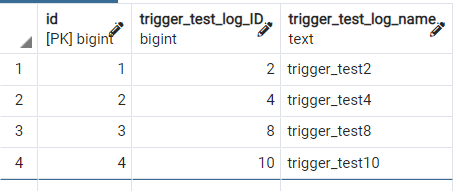
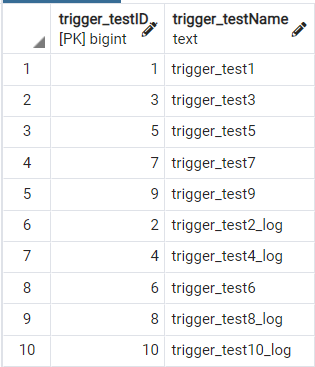
Зміни у таблицях бази даних

Початковий стан:

SELECT \* FROM "trigger\_test"; SELECT \* FROM "trigger\_test\_log";

Після виконання запиту на оновлення:

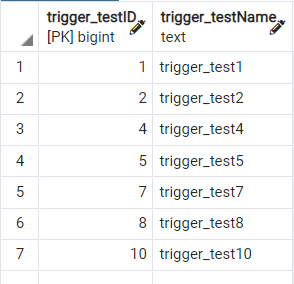
UPDATE "trigger\_test" SET "trigger\_testName" = "trigger\_testName" || '\_log' WHERE "trigger\_testID" % 2 = 0;



Наочно можемо переконатись, що виконалась та гілка алгоритму тригера, що відповідає за парні рядки (оскільки є умова для парних), а для 6 рядка він також виконався, але пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на оновлення потрібно повертати новий стан, а при запиті а видалення старий.

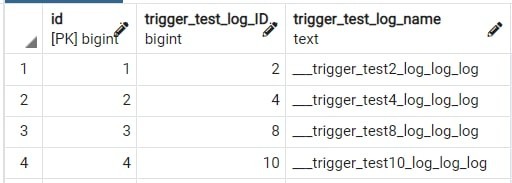
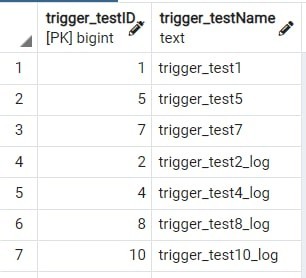
Після виконання запиту на видалення:

DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;



Якщо виконувати ці запити окремо одне від одного, то у таблиці trigger\_test видаляються кратні трьом рядки, але таблиця trigger\_test\_log виявляється пустою. Так відбувається тому, що у гілці алгоритму для чисел кратних трьом у trigger\_test\_log лише модифікуються існуючі записи, але нові не додаються. Оскільки до цього не було виконано оновлення, ця таблиця пуста і модифікувати нема чого.

Якщо зробити вищезгадані запити підряд побачимо наступне:



Бачимо, що записи кратні трьом видалились з trigger\_test, а до текстових полів цих записів у кінці додалось "\_log".

До текстових полів trigger\_test\_log на початку додались два вимволи "\_", а в кінці три "\_log". Один "\_log" в кінці додався завдяки виконанню запиту update для всіх парних рядків. А інші два "\_log" та два символи "\_" на початку додались тому, що запит на видалення для записів 3 та 9 виконались за тією самою гілкою алгоритму (кратні трьом), а запит на видалення запису 6 виконався за іншою гілкою (кратність 2 та 3).