

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

Батура В. В.

Перевірив:

Київ – 2021

# Постановка задачі

*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.

|  |  |  |
| --- | --- | --- |
| № варіанта | Види індексів | Умови для тригера |
| *1* | *Btree, Hash* | *before insert, delete* |

Посилання на репозиторій у GitHub з та звітом:

https://github.com/BaturaVetal/database

# Завдання №1

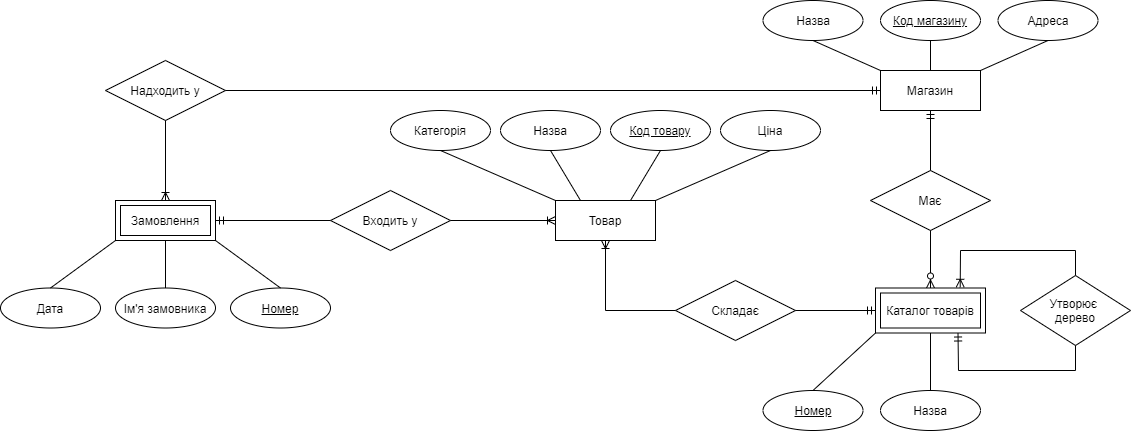
Обрана предметна галузь передбачає отримання і обробку замовлень з різних інтернет-магазинів.

Рисунок 1. ER-діаграма, побудована за нотацією Чена

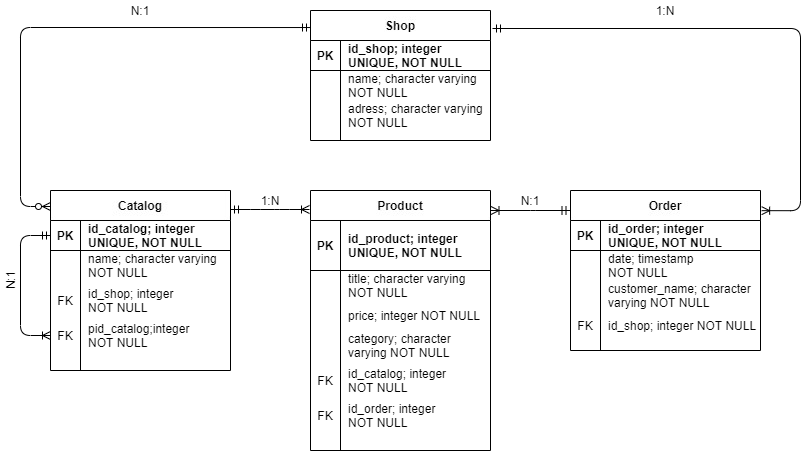


Рисунок 2. Схема бази даних

Таблиці бази даних у середовищі PgAdmin4

-- Table: public.Product

-- DROP TABLE public."Product";

CREATE TABLE IF NOT EXISTS public."Product" (

id\_product integer NOT NULL,

title character varying COLLATE pg\_catalog."default" NOT NULL,

price integer NOT NULL,

category character varying COLLATE pg\_catalog."default" NOT NULL,

id\_catalog integer NOT NULL,

id\_order integer NOT NULL,

CONSTRAINT "Product\_pkey" PRIMARY KEY (id\_product),

CONSTRAINT "Product\_id\_catalog\_fkey" FOREIGN KEY (id\_catalog)

REFERENCES public."Catalog" (id\_catalog) MATCH SIMPLE ON UPDATE NO ACTION

ON DELETE NO ACTION NOT VALID,

CONSTRAINT "Product\_id\_order\_fkey" FOREIGN KEY (id\_order) REFERENCES public."Order" (id\_order) MATCH SIMPLE

ON UPDATE NO ACTION ON DELETE NO ACTION NOT VALID

)

TABLESPACE pg\_default;

ALTER TABLE public."Product" OWNER to postgres;

COMMENT ON COLUMN public."Product".id\_order IS '(UNIQUE)';

-- Table: public.Order

-- DROP TABLE public."Order";

CREATE TABLE IF NOT EXISTS public."Order" (

id\_order integer NOT NULL,

customer\_name character varying COLLATE pg\_catalog."default" NOT NULL,

id\_shop integer NOT NULL,

date timestamp without time zone NOT NULL, CONSTRAINT "Order\_pkey" PRIMARY KEY (id\_order),

CONSTRAINT "Order\_id\_shop\_fkey" FOREIGN KEY (id\_shop) REFERENCES public."Shop" (id\_shop) MATCH SIMPLE ON UPDATE NO ACTION

ON DELETE NO ACTION NOT VALID

)

TABLESPACE pg\_default;

ALTER TABLE public."Order"

OWNER to postgres;

-- Table: public.Catalog

-- DROP TABLE public."Catalog";

CREATE TABLE IF NOT EXISTS public."Catalog" (

id\_catalog integer NOT NULL,

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

id\_shop integer NOT NULL,

pid\_catalog integer NOT NULL,

CONSTRAINT "Catalog\_pkey" PRIMARY KEY (id\_catalog), CONSTRAINT "Catalog\_id\_shop\_fkey" FOREIGN KEY (id\_shop)

REFERENCES public."Shop" (id\_shop) MATCH SIMPLE ON UPDATE NO ACTION

ON DELETE NO ACTION NOT VALID,

CONSTRAINT pid\_catalog\_fkey FOREIGN KEY (pid\_catalog) REFERENCES public."Catalog" (id\_catalog) MATCH SIMPLE ON UPDATE NO ACTION

ON DELETE NO ACTION NOT VALID

)

TABLESPACE pg\_default;

ALTER TABLE public."Catalog"

OWNER to postgres;

-- Table: public.Shop

-- DROP TABLE public."Shop";

CREATE TABLE IF NOT EXISTS public."Shop" (

id\_shop integer NOT NULL,

address character varying COLLATE pg\_catalog."default" NOT NULL,

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

CONSTRAINT "Shop\_pkey" PRIMARY KEY (id\_shop)

)

TABLESPACE pg\_default;

ALTER TABLE public."Shop"

OWNER to postgres;

Класи ORM у реалізованому модулі Model

class Shop(Orders):

tablename\_\_= 'Shop'

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

address = Column(String)

name = Column(String)

catalogs = relationship("Catalog")

orders = relationship("Order")

def init (self, id\_shop, address, name):

self.id\_shop = id\_shop

self.address = address

self.name = name

def repr (self):

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

.format(self.id\_shop, self.address, self.name)

class Catalog(Orders):

tablename\_\_= 'Catalog'

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

name = Column(String)

id\_shop = Column(Integer, ForeignKey('Shop.id\_shop'))

pid\_catalog = Column(Integer, ForeignKey('Catalog.id\_catalog'))

products = relationship("Product")

parent\_catalogs = relationship("Catalog")

def init (self, id\_catalog, name, id\_shop, pid\_catalog):

self.id\_catalog = id\_catalog

self.name = name

self.id\_shop = id\_shop

self.pid\_catalog = pid\_catalog

def repr (self):

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

.format(self.id\_catalog, self.name, self.id\_shop, self.pid\_catalog)

class Order(Orders):

tablename\_\_= 'Order'

id\_order = Column(Integer, primary\_key=True) customer\_name = Column(String)

id\_shop = Column(Integer, ForeignKey('Shop.id\_shop'))

date = Column(Date)

products = relationship("Product")

def init (self, id\_order, customer\_name, id\_shop, date):

self.id\_order = id\_order

self.customer\_name = customer\_name

self.id\_shop = id\_shop

self.date = date

def repr (self):

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

.format(self.id\_order, self.customer\_name, self.id\_shop, self.date)

class Product(Orders):

tablename\_\_= 'Product'

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

title = Column(String)

price = Column(Float)

category = Column(String)

id\_catalog = Column(Integer, ForeignKey('Catalog.id\_catalog'))

id\_order = Column(Integer, ForeignKey('Order.id\_order'))

def init (self, id\_product, title, price, category, id\_catalog, id\_order):

self.id\_product = id\_product

self.title = title

self.price = price

self.category = category

self.id\_catalog = id\_catalog

self.id\_order = id\_order

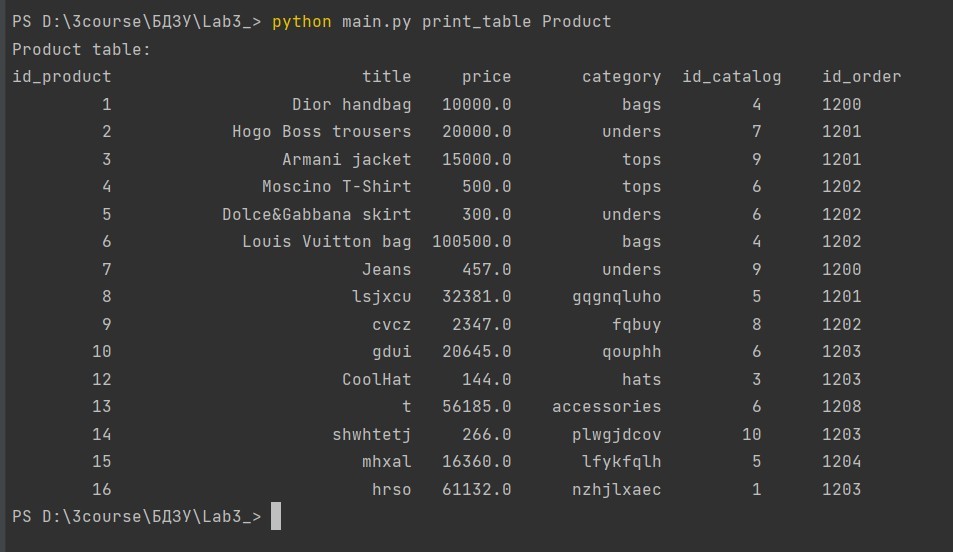
def repr (self):

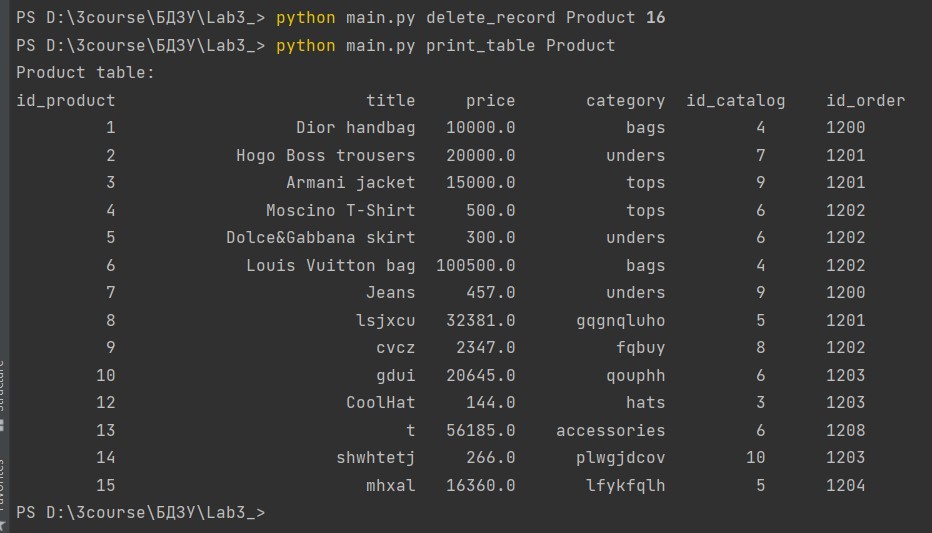
return "{:>10}{:>30}{:>10}{:>15}{:>10}{:>10}" \

.format(self.id\_product, self.title, self.price, self.category, self.id\_catalog, self.id\_order)

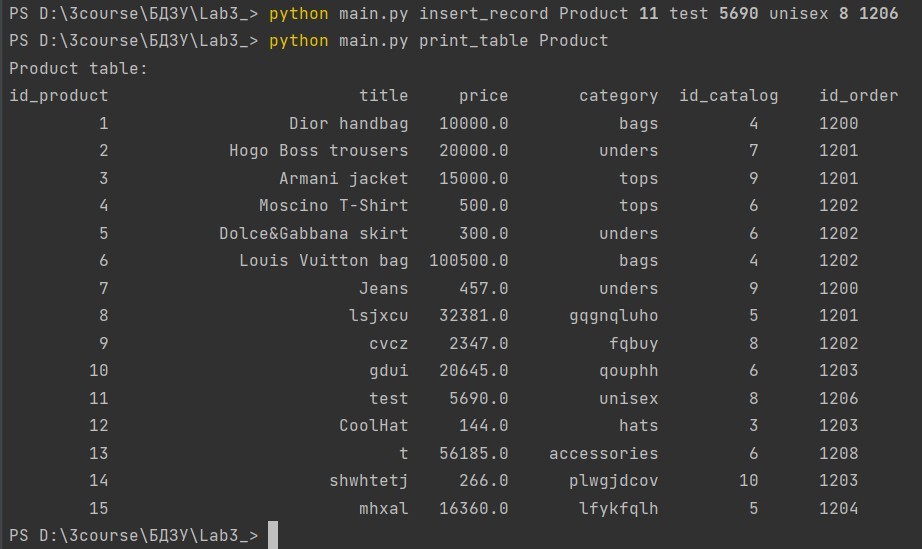
Запити у вигляді ORM

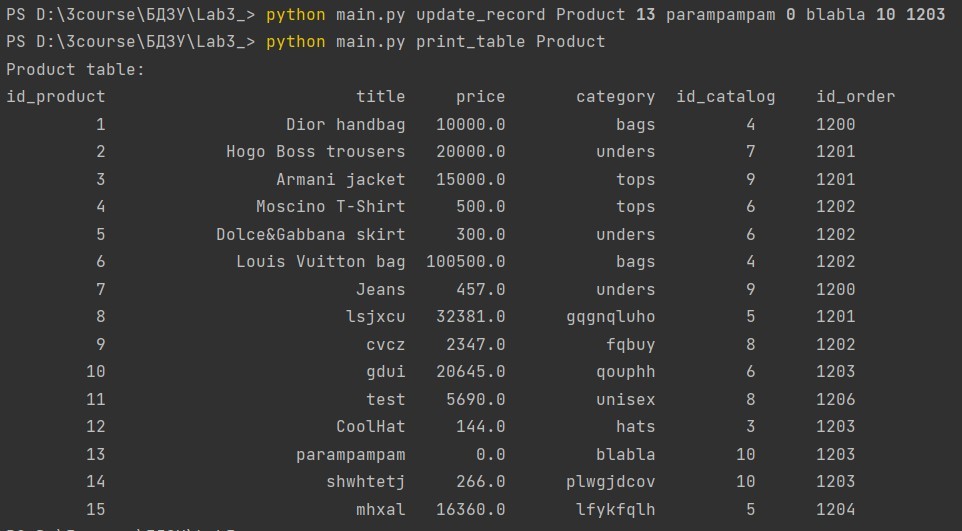
Продемонструємо вставку, вилучення, редагування даних на прикладі таблиці Product

Початковий стан:

Видалення запису:

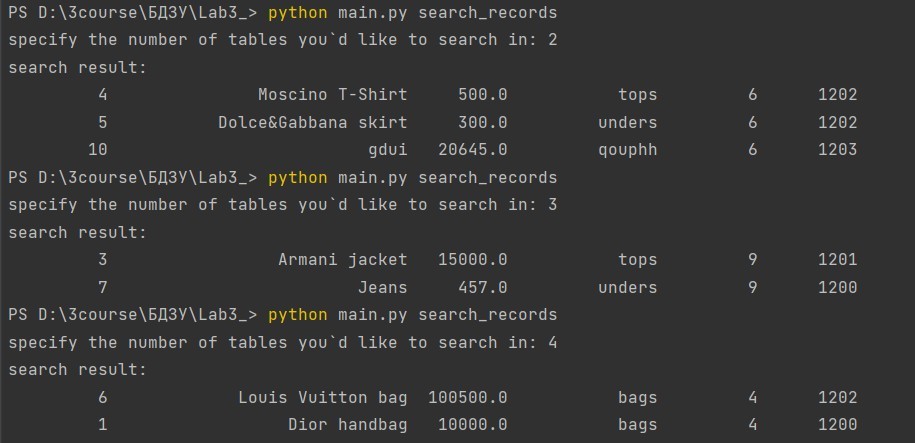
Вставка запису:



Редагування запису:

Запити пошуку та генерації рандомізованих даних також було реалізовано, логіку пошуку було змінено у порівнянні з лабораторною роботою №2 (усі дані для пошуку передвизначено, тепер вони не вводяться з клавіатури).

Пошук за трьома атрибутами у двох таблицях, за трьома атрибутами у трьох таблицях, за чотирма атрибутами у чотирьох таблицях (виводяться відповідні записи з таблиці Product):



# Завдання №2

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

Btree

Btree призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями (наприклад - реченнями), а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах (повторювані частини слів або речень). Індекс Btree зберігає набір пар (ключ, список появи ключа), 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.

Створення таблиці БД:

DROP TABLE IF EXISTS "btr\_test";

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

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

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

Запити для тестування:

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

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

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

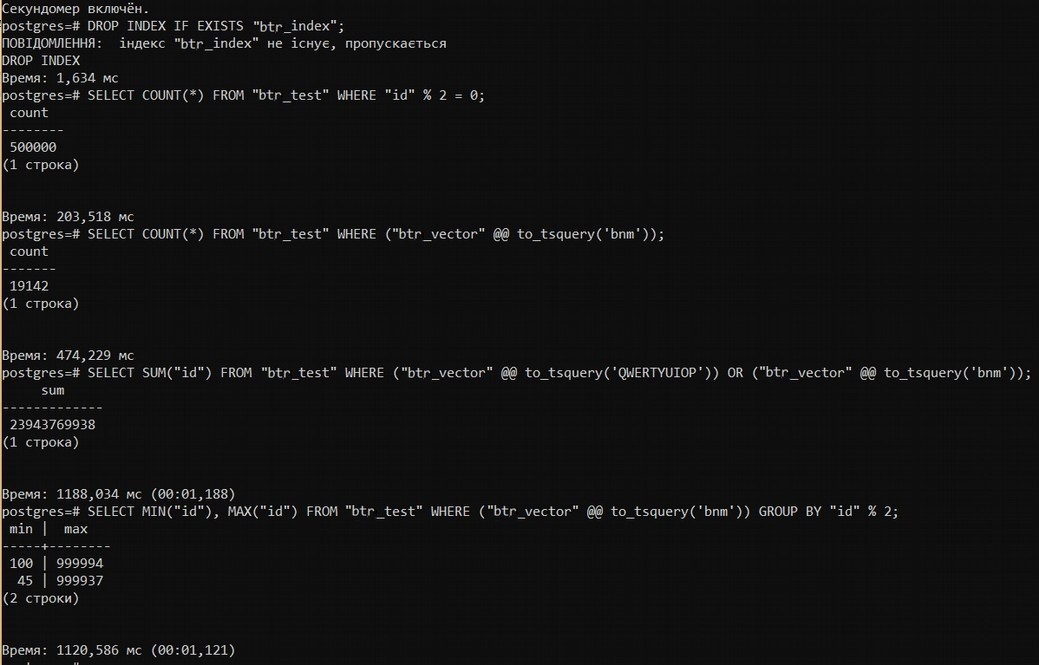
SELECT MIN("id"), MAX("id") FROM "btr\_test" WHERE ("btr\_vector" @@ to\_tsquery('bnm')) GROUP BY "id" % 2;

Створення індексу:

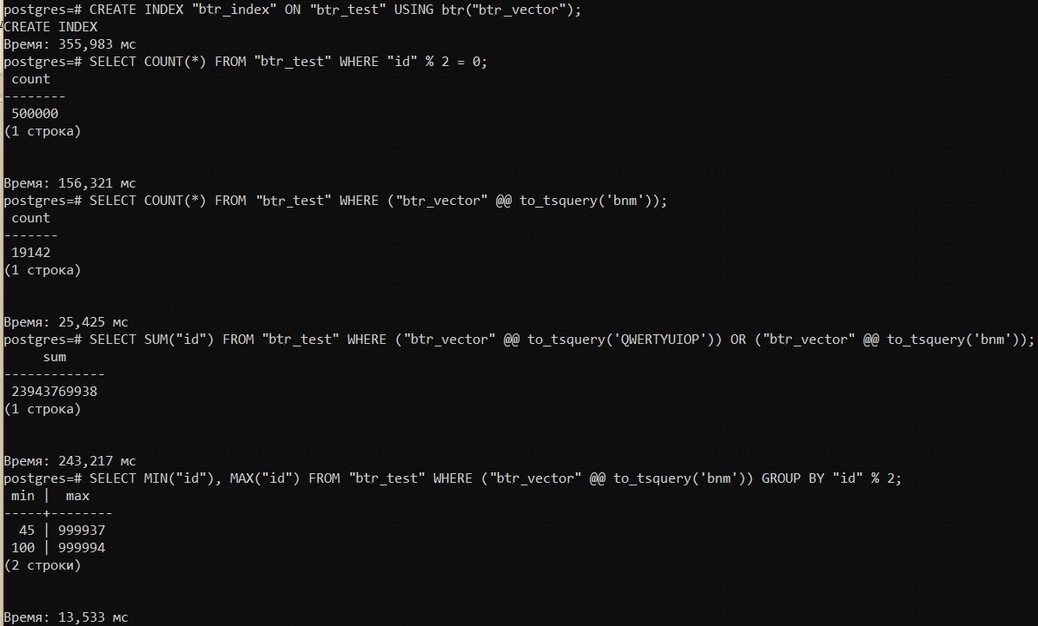
DROP INDEX IF EXISTS "btr\_index";

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

Результати і час виконання на скріншотах з psql.exe Запити без індексування:



Запити з індексуванням:



З отриманих результатів бачимо, що в усіх заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). 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.

Створення 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;

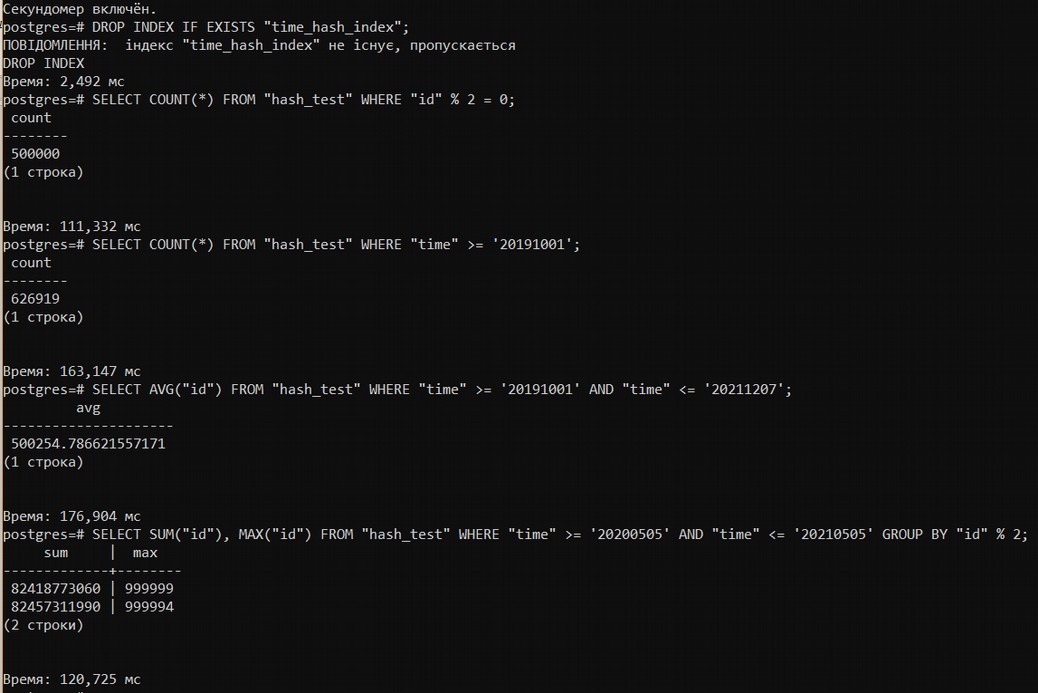
Створення індексу:

DROP TABLE IF EXISTS "hash\_test";

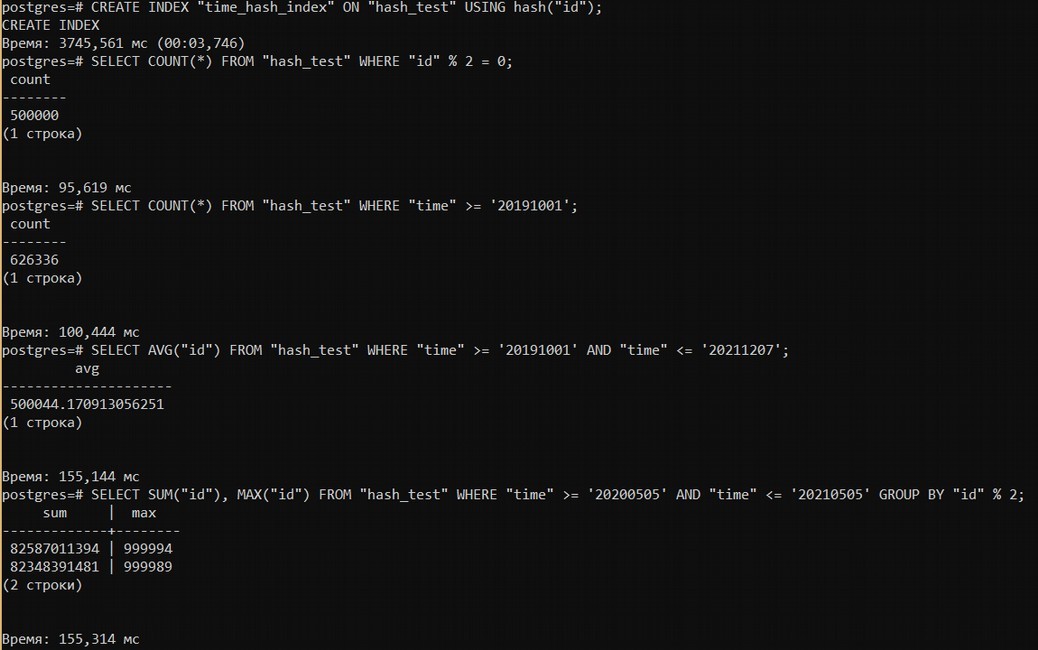
CREATE INDEX "time\_hash\_index" ON "hash\_test" USING hash("id");

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 індексування. 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. 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";

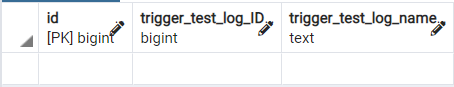
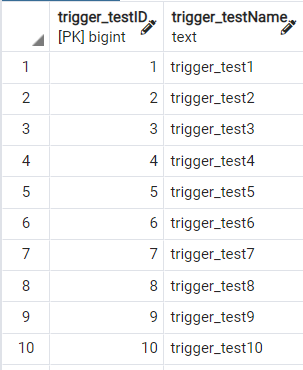
END LOOP; RETURN OLD;

END IF; END;

$trigger$ LANGUAGE plpgsql;

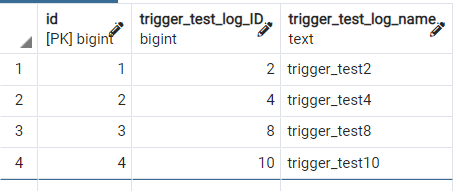
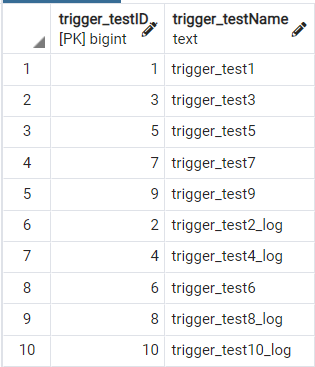
Скріншоти зі змінами у таблицях бази даних

Початковий стан

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

Після виконання запиту на оновлення

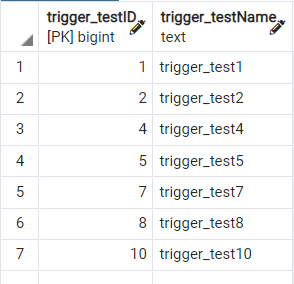
UPDATE "trigger\_test" SET "trigger\_testName" = "trigger\_testName" || '\_log' WHERE "trigger\_testID" % 2 = 0;



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

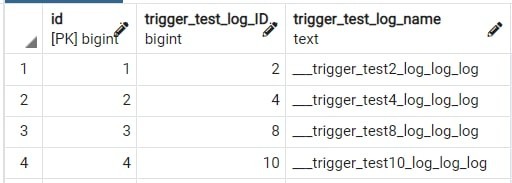
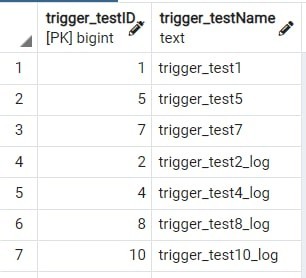
DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;



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).

# Завдання №4

Для цього завдання також створювалась окрема таблиця з деякими початковими даними:

DROP TABLE IF EXISTS "transactions"; CREATE TABLE "transactions"(

"id" bigserial PRIMARY KEY, "numeric" bigint,

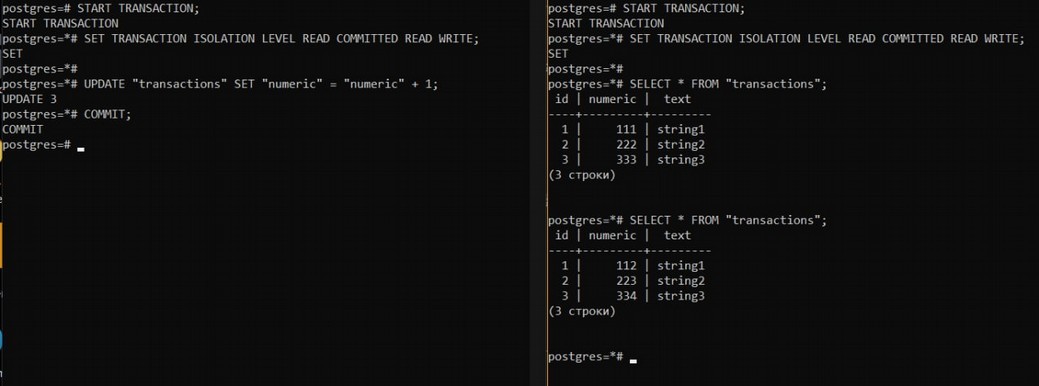
"text" text

);

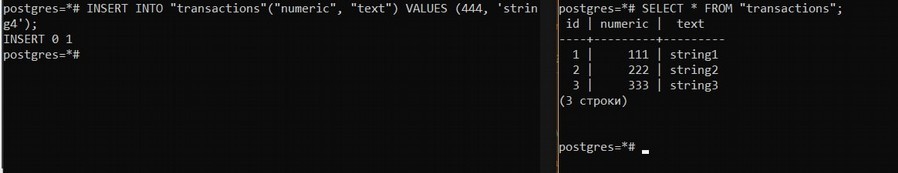
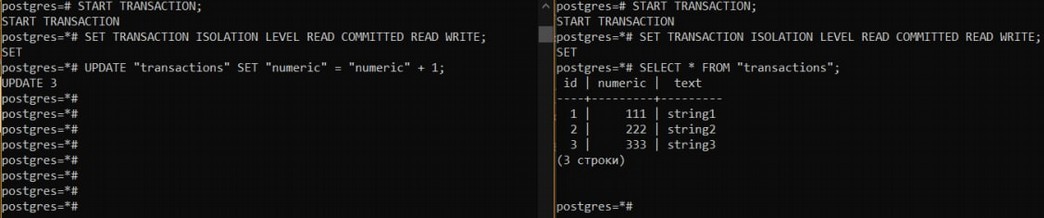
INSERT INTO "transactions"("numeric", "text") VALUES (111, 'string1'), (222, 'string2'), (333, 'string3');

READ COMMITTED

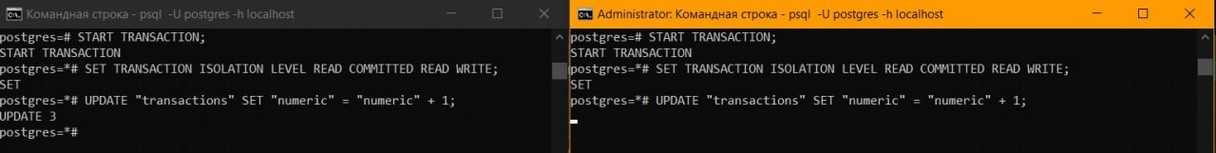
На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).



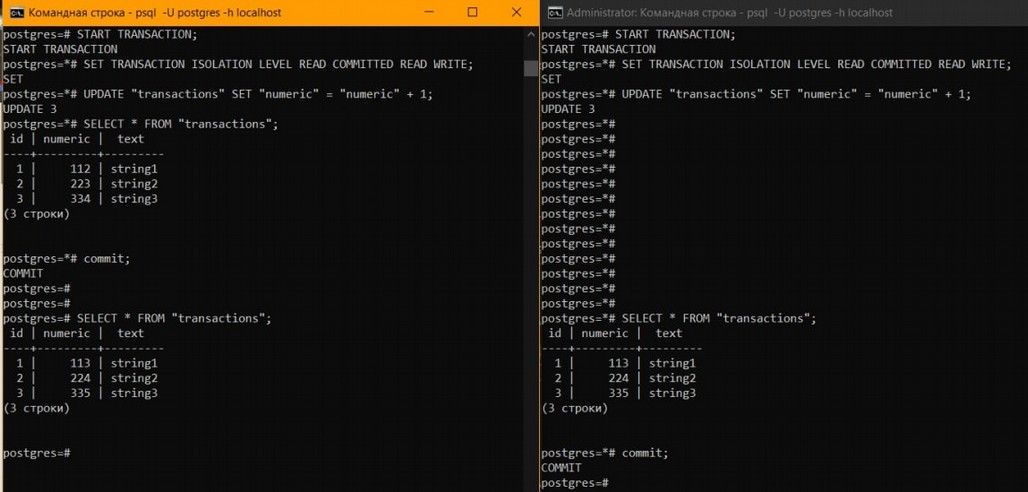
Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.



На цьому знімку також бачимо, що друга транзакція (справа) не може внести дану базу, доки не завершилась попередня.



А тут бачимо, що після завершення першої, друга транзакція виконала запит, змінивши вже ті дані, що були закомічені першою транзакцією.

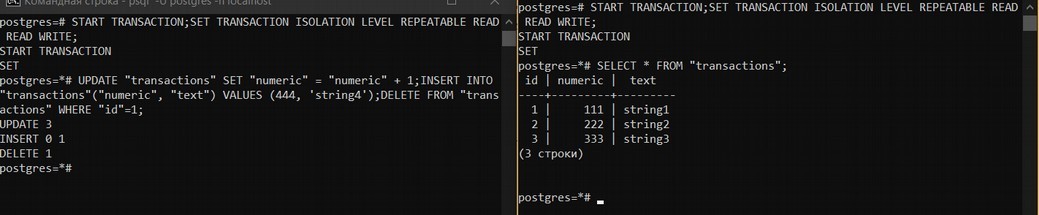


Коли Т2 бачить дані Т1 запитів UPDATE, DELETE виникає феномен повторного читання, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

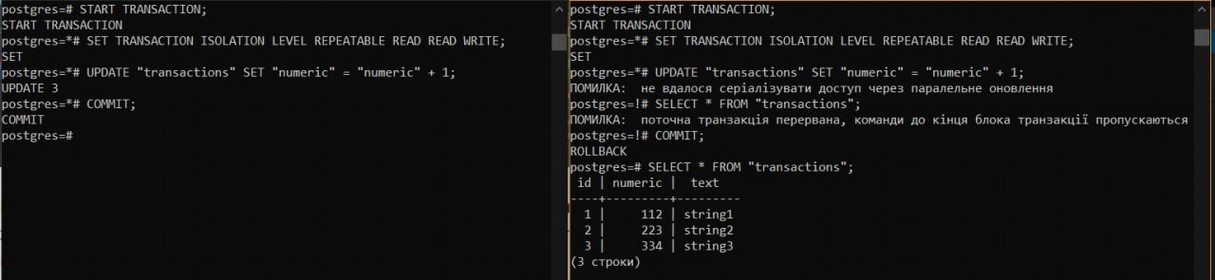
REPEATABLE READ

На цьому рівні ізоляції Т2 не бачитиме змінені дані транзакцією Т1, але також не зможе отримати доступ до тих самих даних.

Тут видно, що друга не бачить змін з першої:



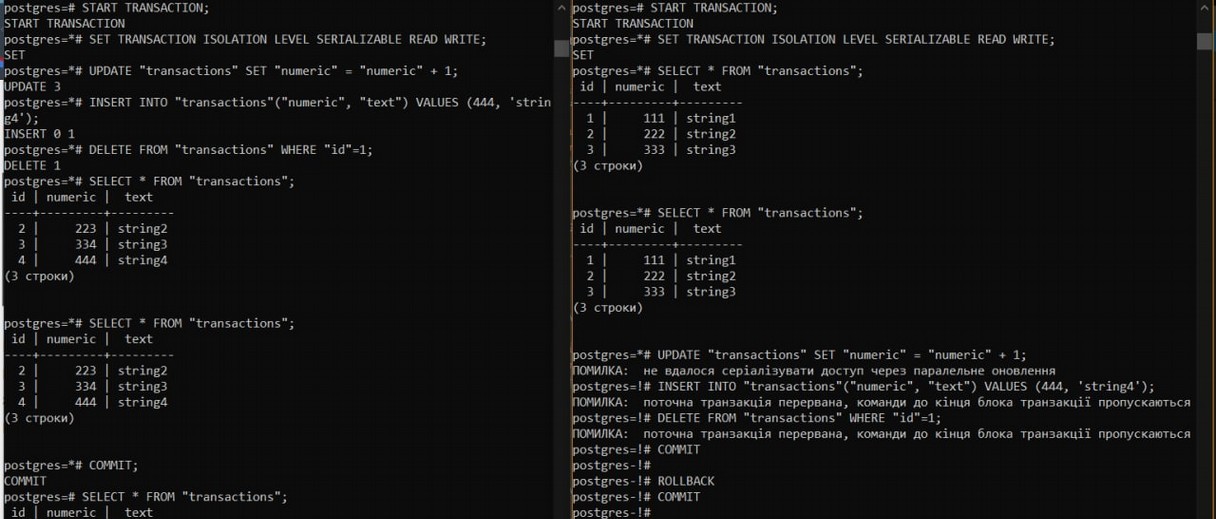
А тут, що отримуємо помилку при спробі доступу до тих самих даних:



Бачимо, що не виникає читання фантомів та повторного читання, а також заборонено одночасний доступ до незбережених даних. Хоча класично цей рівень ізоляції призначений для попередження повторного читання.

SERIALIZABLE

На цьому рівні транзакції поводять себе так, ніби вони не знають одна про одну. Вони не можуть вплинути одна на одну і одночасний доступ строго заборонений.



У попередньому випадку вдалось відкотити другу транзакцію і це не вплинуло на подальшу можливість роботи в терміналі. На цьому ж рівні навіть після завершення першої не вдалося зробити ні COMMIT ні ROLLBACK для другої транзакції. Взагалі, в класичному представленні цей рівень призначений для недопущення явища читання фантомів. На цьому рівні ізоляції ми отримуємо максимальну узгодженість даних і можемо бути впевнені, що зайві дані не будуть зафіксовані.