

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**»

Виконала: Костюченко А. В.

Студентка групи КВ-03

**Київ 2023**

**Лабораторна робота №3**

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). (не реалізовано)
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 12

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| 12 | *Btree, Gin* | *After insert, update* |

Посилання на Github: <https://github.com/reginakasteen/data-bases/tree/main/data_bases>

***Завдання 2***

Для виконання індексів було створено окремі таблиці у базі даних test з 1000 000 записів.

***GIN***

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. Наприклад, такою задачею може бути пошук повторюваних частин в реченні. Індекс GIN 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. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з’являється багато разів.

**SQL запити**

*Запит для створення 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:*

Було протестовано 4 запити: 1 - виведення записів, у яких ідентифікатор кратний числу 2; 2 - виведення записів, у яких наявне сполучення букв bnm; 3 - виведення суми ідентифікаторів записів, у яких наявні сполучення букв QWERTYUIO або bnm; 4 - виведення мінімального ідентифікатора та максимального ідентифікатора записів, де є сполучення букв bnm, сортування за кратними 2 ідентифікаторами.

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:*

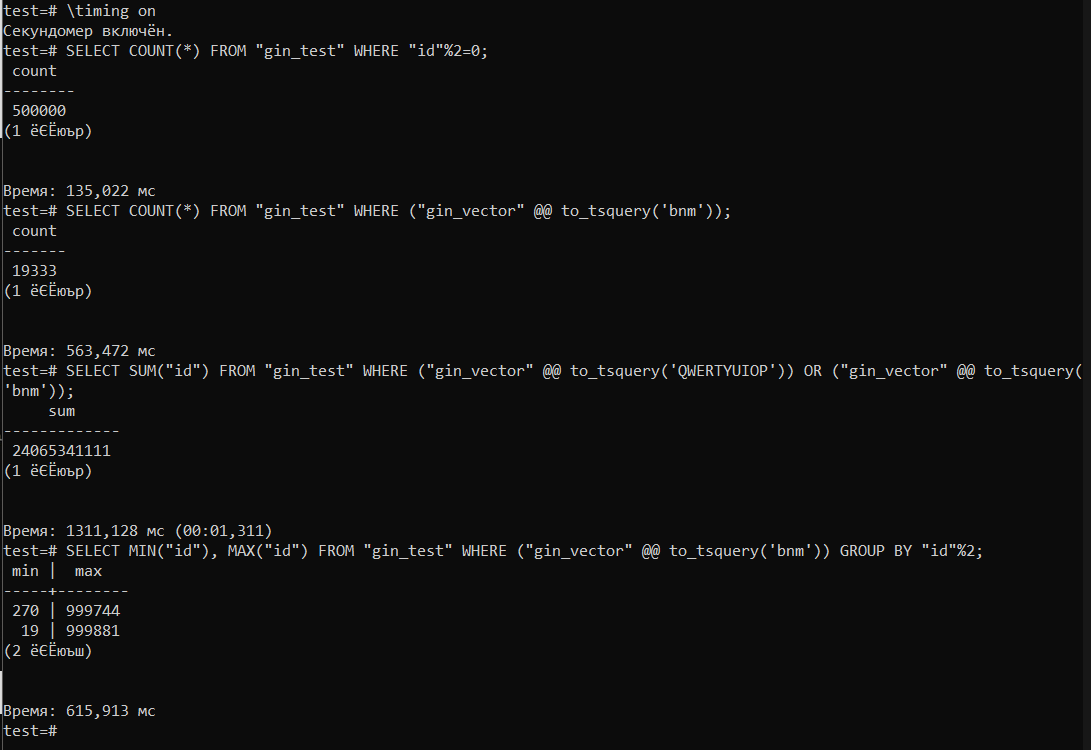
DROP INDEX IF EXISTS "gin\_index";

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

**Результати виконання запитів**

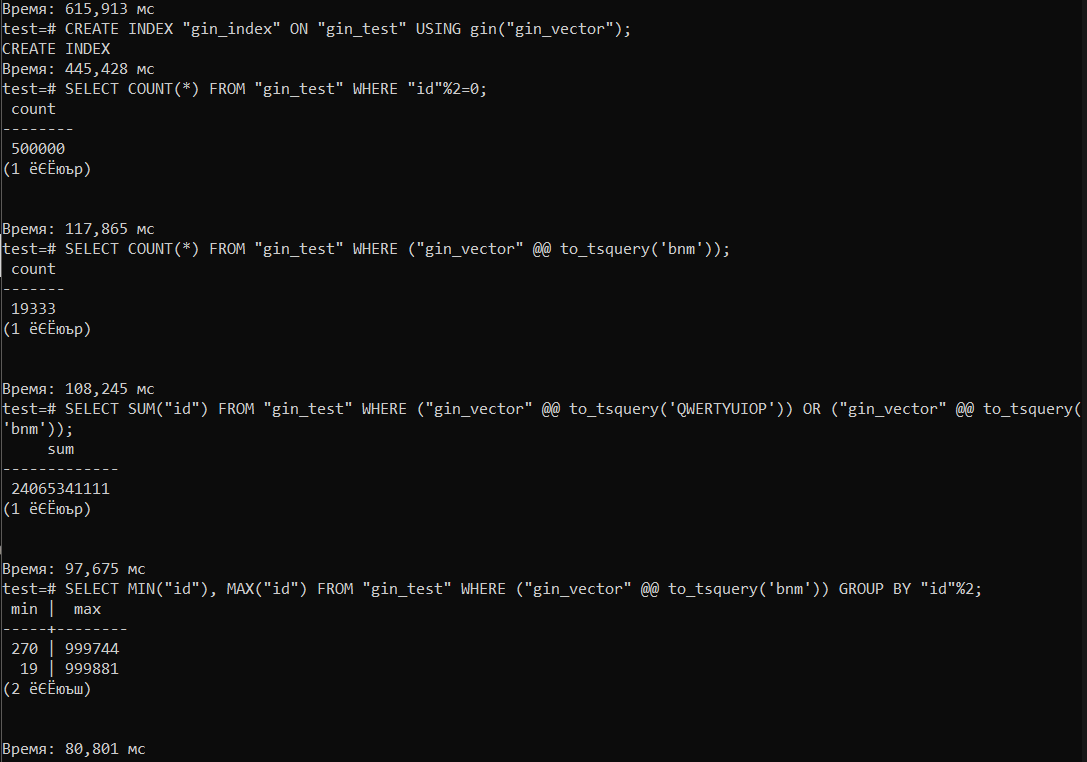
Запити вводились у psql.exe.

Запити без індексування:



**Час виконання запитів** **без індексу GIN**

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 135,022 мс | 563,472 мс | 1311,128 мс | 615,913 мс |



**Час виконання операції з індексом GIN**

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 117,865 мс | 108,245 мс | 97,675 мс | 80,801 мс |

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). 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.

***BTree***

Індекс BTree призначений для даних, які можна відсортувати, наприклад, дати чи просто числові дані. Пошук починається з кореня вузла, і потрібно визначити, по якому з дочірніх вузлів спускатися. Знаючи ключи в корені, можна зрозуміти діапазони значень в дочірніх вузлах. Процедура повторюється доки не знайдемо вузол, з якого можна отримати необхідні дані.

**SQL запити**

*Створення таблиці БД і внесення 1000000 записів:*

DROP TABLE IF EXISTS "btree\_test";

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

INSERT INTO "btree\_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:*

Було протестовано 4 запити: 1 – виведення записів, ідентифікатор яких кратний 2; 2 - виведення записів, у яких час більше або дорівнює 2019-10-01; 3 - виведення середнього значення ідентифікаторів записів, у яких час знаходиться в проміжку між 2019-10-01 та 2021-12-7; 4 - виведення суми ідентифікаторів, а також максимального ідентифікатора записів, у яких час знаходиться в промяжку між 2020-05-05 та 2021-05-05, сортування за кратними 2 ідентифікаторами.

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

SELECT COUNT(\*) FROM "btree\_test" WHERE “time” >= ‘20191001’;

SELECT AVG(“id”) FROM “btree\_test” WHERE “time” >= ‘20191001’ AND “time” <= ‘20211207’;

SELECT SUM(“id”), MAX(“id”) FROM “btree\_test” WHERE “time” >= ‘20200505’ AND “time” <= ‘20210505’ GROUP BY “id”%2;

*Створення індексу:*

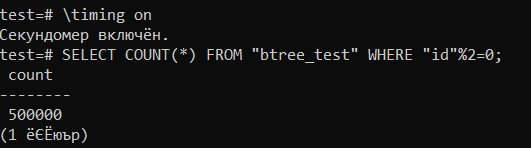
DROP INDEX IF EXISTS "btree\_index";

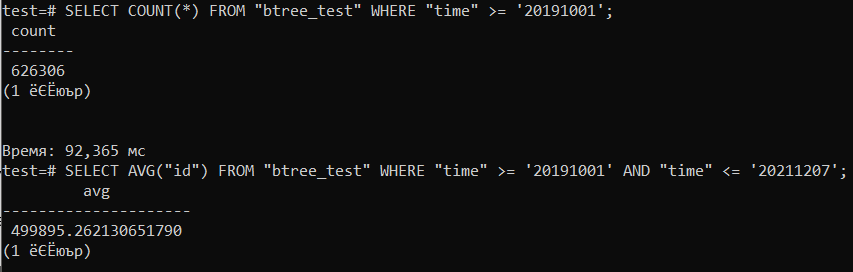
CREATE INDEX "btree\_time\_index" ON "btree\_test" ("id");

**Результати виконання запитів**

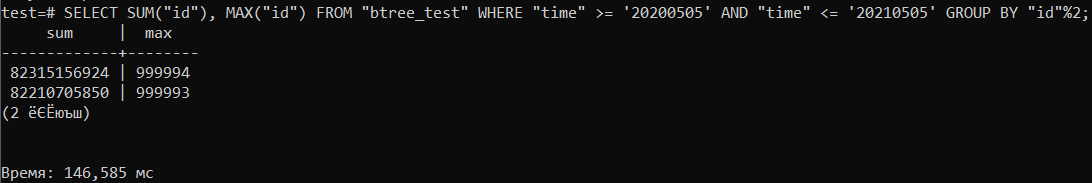
Запити вводились у psql.exe.

Запити без індексування:



Перший запит

Запити 2 і 3

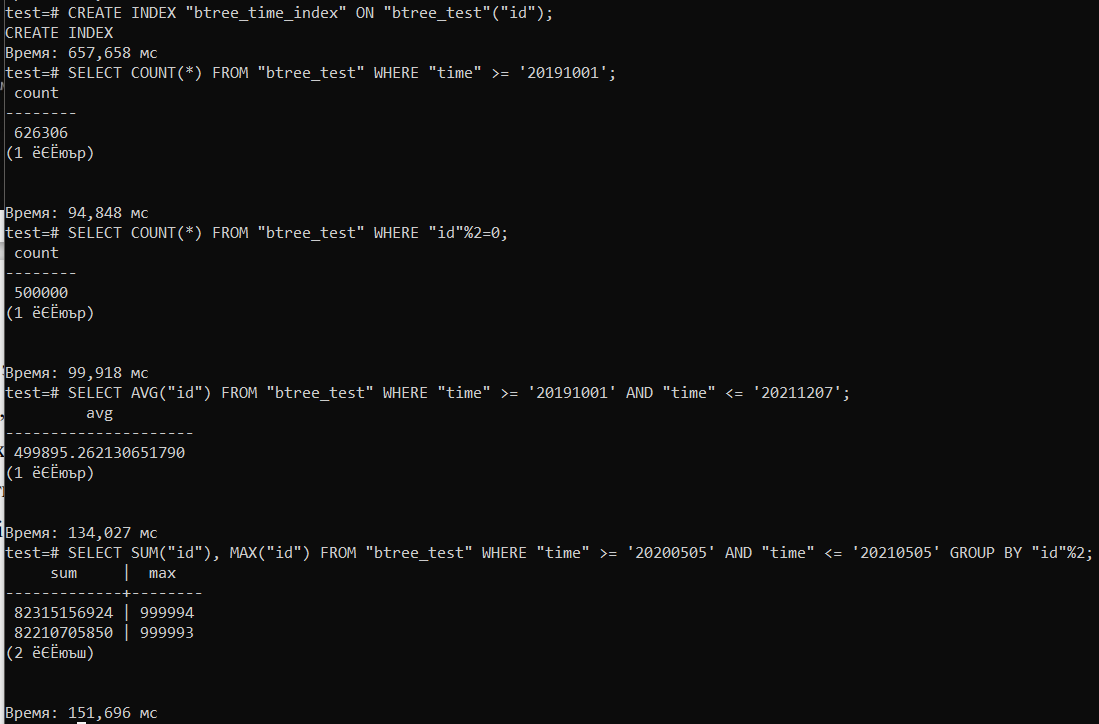


Четвертий запит

**Час виконання запитів без індексу BTree**

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 111,491 мс | 92,365 мс | 154,643 | 146,584 мс |

Запити з індексуванням:



**Час виконання запитів з індексом BTree**

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 94,848 мс | 99,918 мс | 134,027 | 151,696 мс |

Як бачимо з результатів, за допомогою використання індексу BTree виконання операцій дещо пришвидшилося. Це пов’язано з тим, що дерево утворює багато гілок, і через це В-дерево виходить неглибоким навіть для дуже великих таблиць. Цей індекс також рекомендовано використовувати саме для операцій пошуку с порівнянням (нерівностями), що і було продемонстровано в запитах.

**Завдання 3**

Для тестування тригера було створено дві таблиці в базі даних test: таблиця trigger\_test з атрибутами trigger\_testID (ідентифікатор) trigger\_textName (ім’я), trigger\_test\_log з атрибутами id (ідентифікатор), trigger\_test\_log\_ID (зовнішній ключ для зв’язку з таблицею trigger\_test), trigger\_test\_log\_name (ім’я).

Тригер спрацьовує після операції вставки (after insert) та під час операції редагування (update). Серед усіх записів таблиці trigger\_test у курсорному циклі обираються ті, що мають ідентифікатори кратні 2. Якщо цей ідентифікатор також кратний 3, то висвічується повідомлення, що число ділиться на 2 і 3. Також якщо ідентифікатор кратний 2 і 3, то в таблицю trigger\_test\_log вставляються рядки з цими ідентифікаторами та відповідними іменами. В іншому випадку (якщо число не ділиться на 3, але ділиться на 2), викликається повідомлення - «Число парне» і в таблицю trigger\_test\_log вставляються рядки з цими ідентифікаторами та відповідними іменами. Далі з атрибуту trigger\_test\_log\_name видаляються набори символів ‘log’. Якщо число не ділиться на 2, то висвічується повідомлення «Число непарне» і виконується редагування в курсорному циклі: для всіх записів таблиці trigger\_test\_log, що мають в назві сполучення букв ‘\_id’ потрібно замінити ім’я на ‘\_’ та trigger\_test\_log\_name та ‘\_log’.

Тригер спрацьовує, якщо викликати операцію вставки (insert) або редагування (update). Нижче на знімках екрану продемонстровано коректну роботу тригера.

Запити для створення таблиць:

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

);

Команди, що ініціюють виконання тригера:

CREATE TRIGGER "after\_insert\_update\_trigger"

AFTER INSERT OR UPDATE ON "trigger\_test"

FOR EACH ROW

EXECUTE procedure after\_insert\_func();

Початковий вміст таблиці trigger\_test було задано запитом:

INSERT INTO trigger\_test("trigger\_testName") VALUES ('test1'), ('test2'), ('test3'), ('test4'), ('test5'), ('test6');

Текст тригера:

CREATE OR REPLACE FUNCTION after\_insert\_func() RETURNS TRIGGER AS $trigger$

DECLARE

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

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

BEGIN

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

IF NEW."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";

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

END LOOP;

RETURN NEW;

ELSE

RAISE NOTICE 'trigger\_testID is even';

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

END IF;

END;

$trigger$ LANGUAGE plpgsql;

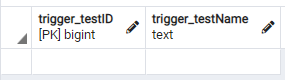
CREATE TRIGGER after\_insert\_test

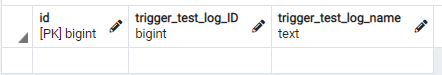
AFTER INSERT OR UPDATE ON "trigger\_test"

FOR EACH ROW EXECUTE PROCEDURE after\_insert\_func();

**Результат виконання**

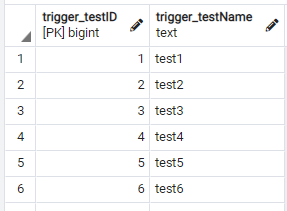
Приклад виконання наведено для таблиць trigger\_test та trigger\_test\_log.



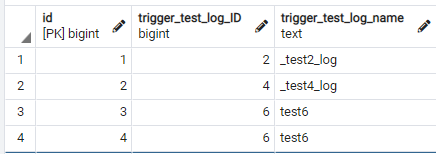


Вище приведені скріншоти початкового стану. Далі було виконано запит на вставку:

INSERT INTO trigger\_test("trigger\_testName") VALUES ('test7'), ('test8');



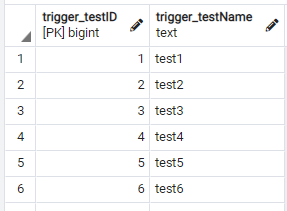
Таблиця trigger\_test після виконання операції вставки



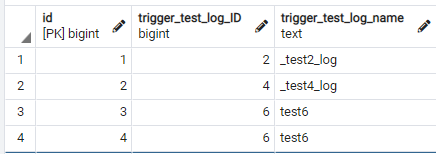
Таблиця trigger\_test\_log після виконання операції вставки

Запит на редагування (додає у всі записи із парними ідентифікаторами в імені сполучення символів ‘\_2’):

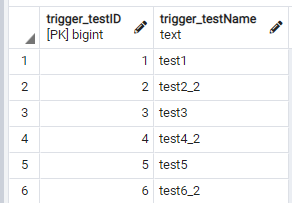
UPDATE "trigger\_test" SET "trigger\_testName" = "trigger\_testName" || '\_2' WHERE "trigger\_testID"%2=0



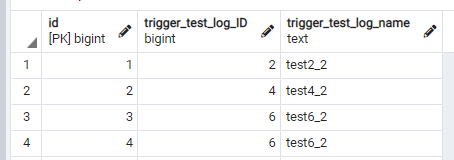
Початковий стан таблиці trigger\_test



Початковий стан таблиці trigger\_test\_log



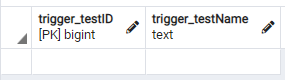
Таблиця trigger\_test після виконання операції редагування



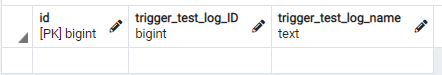
Таблиця trigger\_test\_log після виконання операції редагування та спрацювання тригера

Як видно на знімках, записи із парними ідентифікаторами записалися в таблицю trigger\_test\_log, аналогічно до першого випадку із запитом вставки. Алгоритмічно нічого не змінилося, оскільки всі дії виконуються в залежності від значення ідентифікатора, який не змінювався.

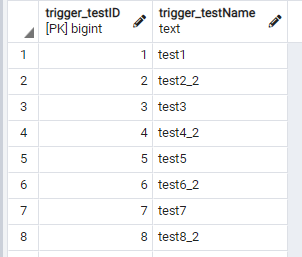
Якщо виконати запити вставки та редагування по черзі, то ситуація дещо зміниться. Спочатку у таблицю trigger\_test вставляються записи з ідентифікаторами від 1 до 8. Далі для парних ідентифікаторів записи копіюються в trigger\_test\_log, а запис з ідентифікатором 6 копіюється в таблицю тричі, оскільки він ділиться і на 2, і на 3.



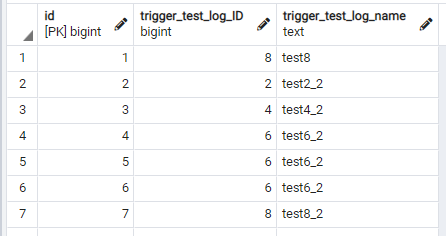
Стан таблиці trigger\_test до почергової вставки та редагування



trigger\_test\_log до почергової вставки та редагування



Стан таблиці trigger\_test після вставки та редагування



Стан таблиці trigger\_test\_log після вставки та редагування

**Завдання 4**

Для цього завдання знадобилася окрема таблиця “transactions” з атрибутами id (ідентифікатор), numeric (число), text (текст). Також у таблицю було додано три записи за допомогою запиту вставки insert.

Запит на створення таблиці та вставку:

DROP TABLE IF EXISTS "transactions";

CREATE TABLE "transactions"(

"id" bigserial PRIMARY KEY,

"numeric" bigint,

"text" text

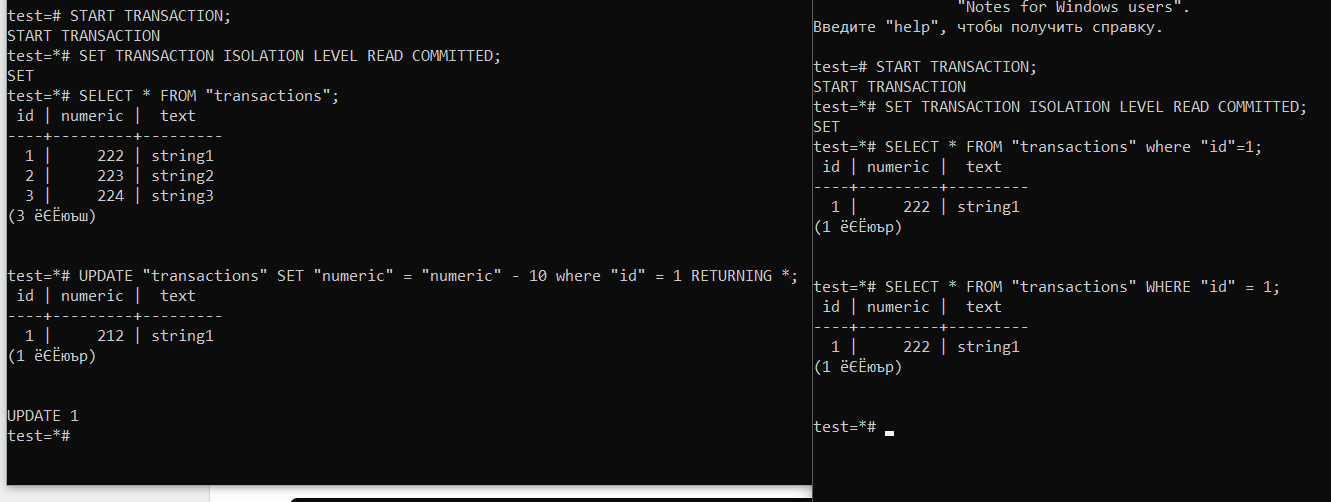
);

INSERT INTO "transactions"("numeric", "text") VALUES (222, 'string1'), (223, 'string2'), (224, 'string3');

*READ COMMINTTED*

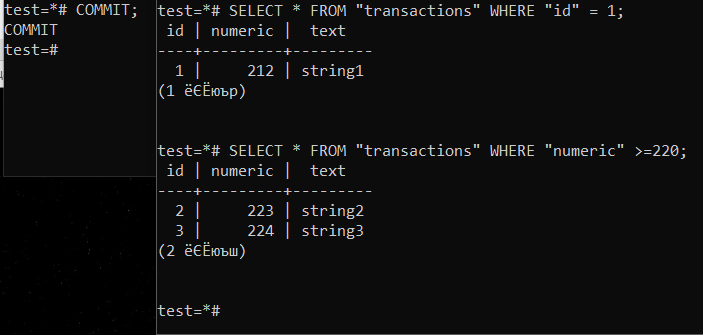
На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).

Спочатку у транзакціях 1 і 2 таблиця має однаковий стан. Якщо у транзакції 1 виконати редагування одного рядка, то в транзакції 2 цих змін не буде помітно, поки в першій транзакції не буде команди commit. Таким чином, феномен «брудного читання» на цьому рівні ізоляції неможливий.



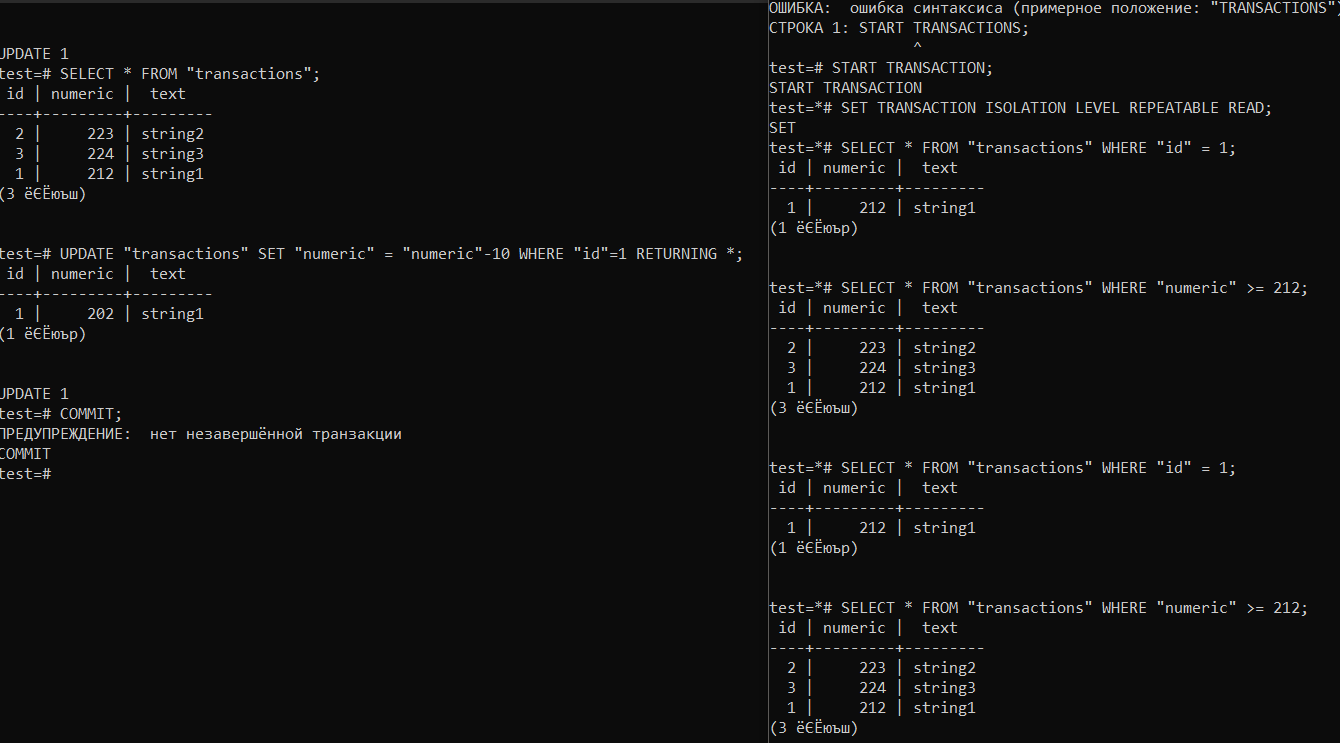
Виконання редагування в одній з одночасних транзакцій на рівні ізоляції Rad committed

Тепер дослідимо феномен «фантомного читання». Після команди commit у першій транзакції у другій ми побачимо, що умові numeric >= 220 відповідають лише 2 рядки, а не 3, як раніше, оскільки зміни були внесені і збережені в обох транзакціях.

«Фантомне читання»

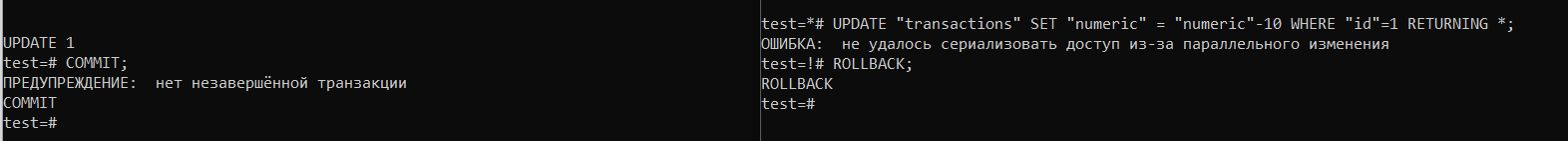
*REPEATABLE READ*

Почнемо дві транзакції на рівні ізоляції repeatable read. У другій транзакції обираємо запис з id = 1 і виводимо записи з numeric >=212. Тепер віднімемо від numeric у рядку з ідентифікатором 1 першої транзакції 10. Викликаємо команду commit. У другій транзакції ніяких змін із цим рядком немає, хоча commit був виконаний. Це призводить до того, що феномен «неповторного читання» неможливий на цьому рівні ізоляції.



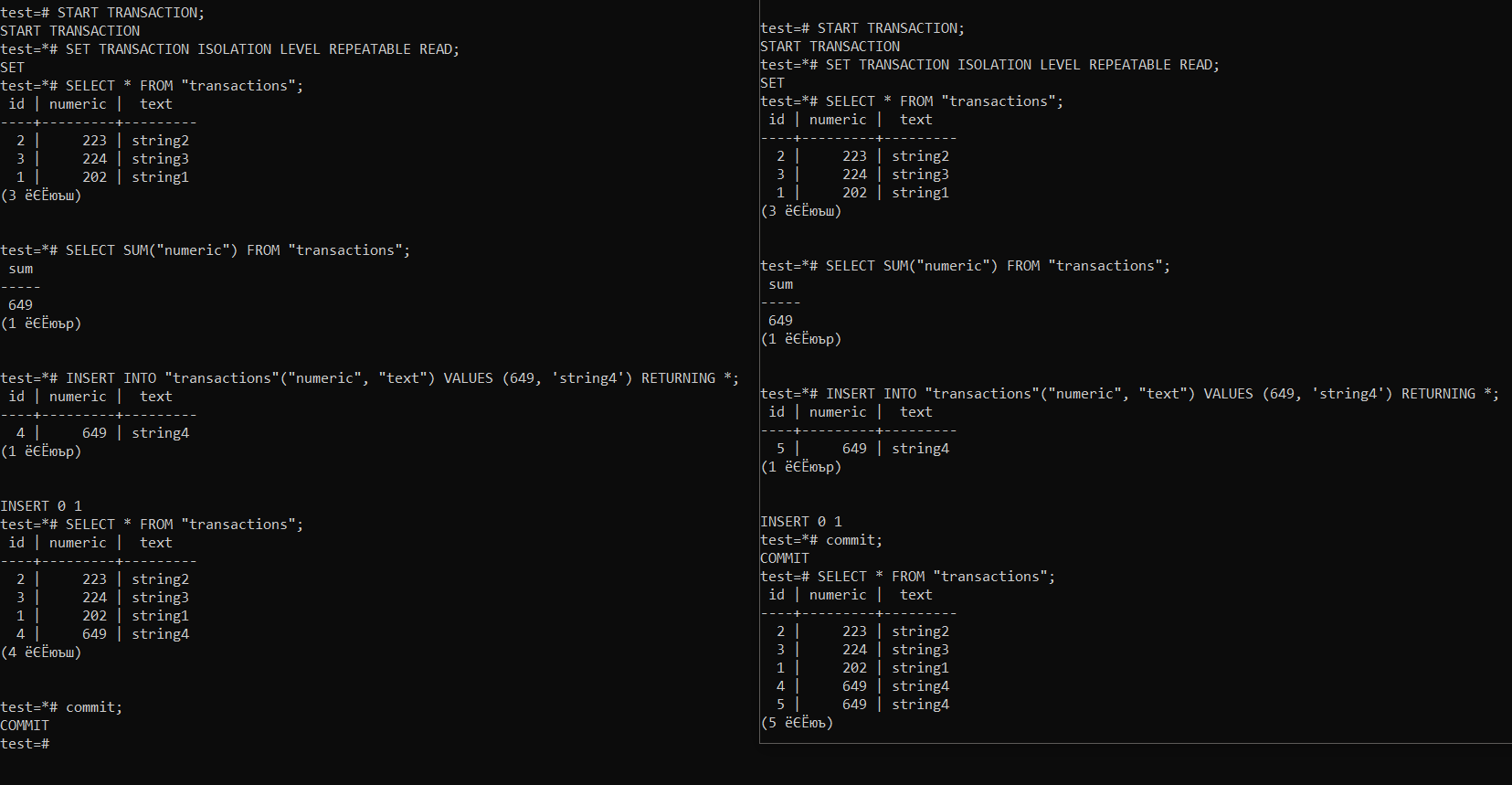
Спроба одночасного внесення змін у таблиці на рівні ізоляції Repeatable read

Якщо спробувати в другій транзакції виконати запит редагування того самого рядка і відняти від numeric 10, то нам висвітиться помилка через паралельні зміни в транзакціях. Це є перевагою repeatable read, оскільки тоді ми можемо уникнути ситуацій, коли при відніманні 10 від 212 ми отримаємо 192 замість 202. Щоб відмінити транзакцію, було викликано команду rollback.



Помилка через паралельні зміни на рівні ізоляції Repeatable read

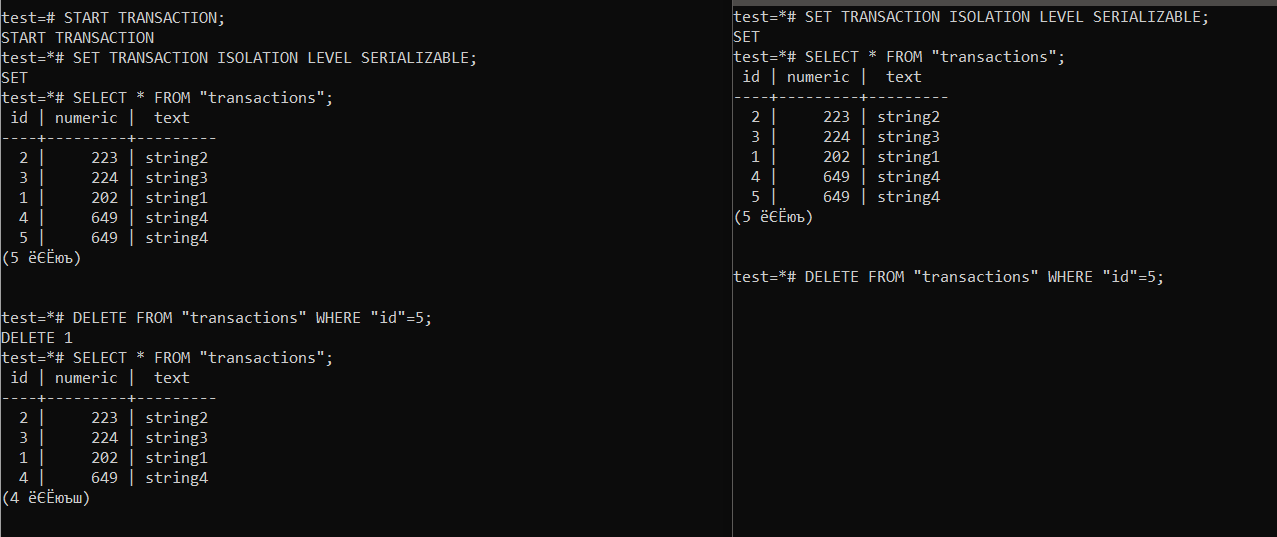
Дослідимо аномалію серіалізації. На рівні ізоляції repeatable read запустимо дві транзакції. У першій виведемо всі рядки і порахуємо суму стовпчика numeric у всіх записах. Додаємо запис із цим значенням в таблицю. Якщо у другій транзакції повторити ті ж самі операції, то стан таблиці на початку ще не змінений, сума буде такою ж, як у першій транзакції. Таким чином, ми додамо до таблиці такий самий рядок, як і першій транзакції. Виконуючи commit в обох транзакціях, ми побачимо два однакових записи в таблиці. Це і є феномен «серіалізації», що пояснюється серійним виконанням двох транзакцій однієї за одною, причому порядок виконання транзакції неважливий.



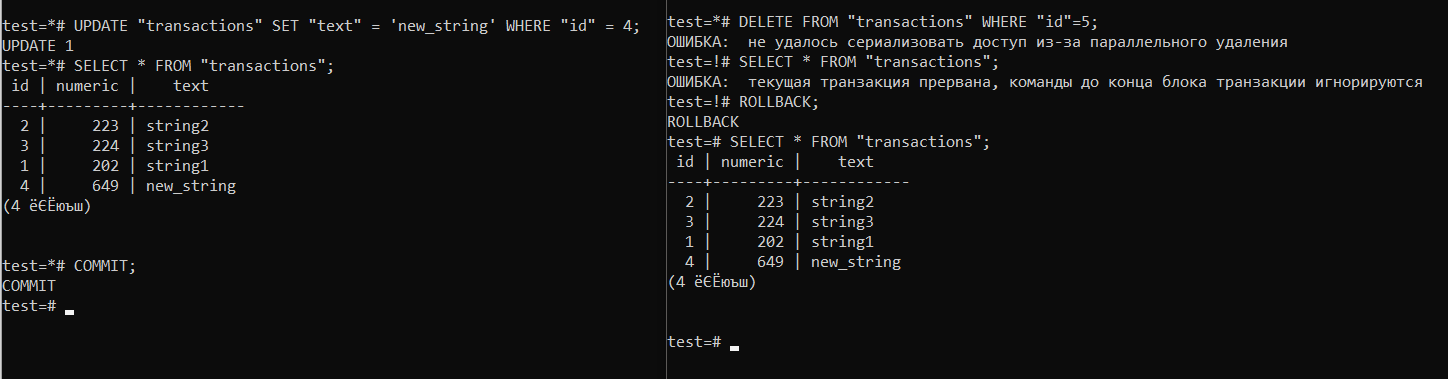
Аномалія серіалізації

*SERIALIZABLE*

Запустимо дві транзакції на рівні Serializable. Спочатку стан таблиці однаковий. У першій транзакції видалимо рядок з id = 5 та виконаємо редагування рядку з id = 4. Якщо у другій транзакції спробувати зробити ті ж операції, то ми повинні будемо очікувати, доки перша транзакція не завершиться. Коли команда commit у першій транзакції виконана, у другій виникає помилка через паралельне видалення. Це неможливо, оскільки якщо запис уже видалений в першій транзакції, то видалити рядок з неіснуючим ідентифікатором неможливо. Ситуацію рятує команда rollback, і після цього бачимо, що зміни внесені і в другу транзакцію.



Спроба одночасного виконання запитів на рівні ізоляції Serializable



Помилка через паралельне вилучення запису на рівні ізоляції Serializable