

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: Мазенко Михайло

Студент групи КВ-93

Перевірив: Павловський В.І.

**Київ 2021**

**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).
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 8

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| *8* | *Btree, Gin* | *After insert, update* |

Посилання на Github: https://github.com/Qlevit8/BDIZU\_lab

**Середовище розробки**

Середовище розробки програми – Visual Studio. Мова програмування Python3.6.

Середовище розробки бази даних - PostgreSQL

Використані бібліотеки:

* sqlalchemy (для зв’язку з СУБД);
* datetime (для роботи з датою і передачею її у запити до БД);
* time (для виміру часу запиту пошуку для завдання 3);
* sys (для реалізації консольного інтерфейсу).

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом: https://github.com/Qlevit8/BDIZU\_lab/tree/main/lab3

**Завдання 1**

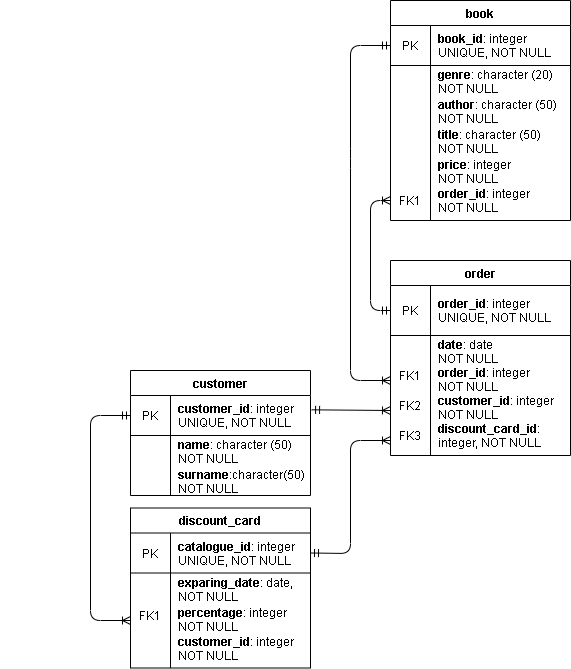


Рисунок 1 – Логічна модель БД

Класи ORM у реалізованому модулі Model

class customer(Orders):

\_\_tablename\_\_ = 'customer'

customer\_id = Column(Integer, primary\_key=True)

customer\_name = Column(String)

customer\_surname = Column(String)

def \_\_init\_\_(self, customer\_id, customer\_name, customer\_surname):

self.customer\_id = customer\_id

self.customer\_name = customer\_name

self.customer\_surname = customer\_surname

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

return 'id: ' + str(self.customer\_id) + \

' || name: ' + self.customer\_name + \

' || surname: ' + self.customer\_surname

class book(Orders):

\_\_tablename\_\_ = 'book'

book\_id = Column(Integer, primary\_key=True)

title = Column(String)

author = Column(String)

genre = Column(String)

price = Column(Integer)

def \_\_init\_\_(self, book\_id, title, author, genre, price):

self.book\_id = book\_id

self.title = title

self.author = author

self.genre = genre

self.price = price

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

return 'book\_id: ' + str(self.book\_id) + \

' || title: ' + self.title + \

' || author: ' + self.author + \

' || genre ' + self.genre + \

' || price: ' + str(self.price)

class discount\_card(Orders):

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

discount\_card\_id = Column(Integer, primary\_key=True)

date\_of\_expiring = Column(Date)

percentage = Column(Integer)

customer\_id = Column(Integer, ForeignKey('customer.customer\_id'))

def \_\_init\_\_(self, discount\_card\_id, date\_of\_expiring, percentage, customer\_id):

self.discount\_card\_id = discount\_card\_id

self.date\_of\_expiring = date\_of\_expiring

self.percentage = percentage

self.customer\_id = customer\_id

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

return 'discount\_card\_id: ' + str(self.discount\_card\_id) + \

' || date\_of\_expiring: ' + str(self.date\_of\_expiring) + \

' || percentage: ' + str(self.percentage) + \

' || customer\_id: ' + str(self.customer\_id)

class order(Orders):

\_\_tablename\_\_ = 'order'

order\_id = Column(Integer, primary\_key=True)

date = Column(Date)

customer\_id = Column(Integer, ForeignKey('customer.customer\_id'))

discount\_card\_id = Column(Integer, ForeignKey('discount\_card.discount\_card\_id'))

def \_\_init\_\_(self, order\_id, date, customer\_id, discount\_card\_id):

self.order\_id = order\_id

self.date = date

self.customer\_id = customer\_id

self.discount\_card\_id = discount\_card\_id

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

return 'order\_id: ' + str(self.order\_id) + \

' || date: ' + str(self.date) + \

' || customer\_id: ' + str(self.customer\_id) + \

' || discount\_card\_id: ' + str(self.discount\_card\_id)

class book\_order(Orders):

\_\_tablename\_\_ = 'book\_order'

book\_order\_id = Column(Integer, primary\_key=True)

book\_id = Column(Integer, ForeignKey('book.book\_id'))

order\_id = Column(Integer, ForeignKey('order.order\_id'))

def \_\_init\_\_(self, book\_order\_id, book\_id, order\_id):

self.book\_order\_id = book\_order\_id

self.book\_id = book\_id

self.order\_id = order\_id

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

return 'book\_order\_id: ' + str(self.book\_order\_id) + \

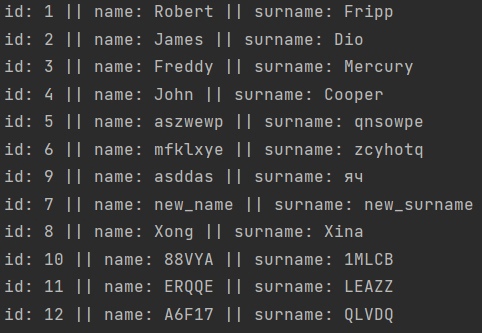
' || book\_id: ' + str(self.book\_id) + \

' || order\_id: ' + str(self.order\_id)

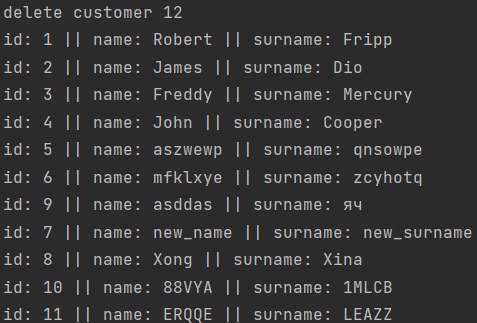
Запити у вигляді ORM

Продемонструємо вставку, вилучення, редагування даних на прикладі таблиці customer

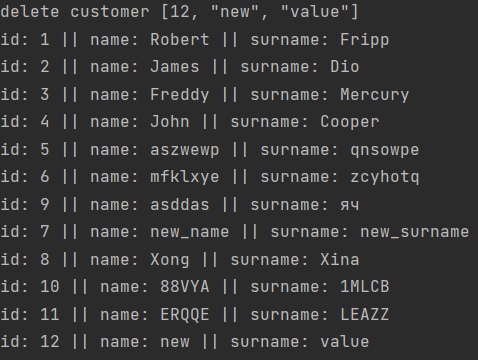
Початковий стан:



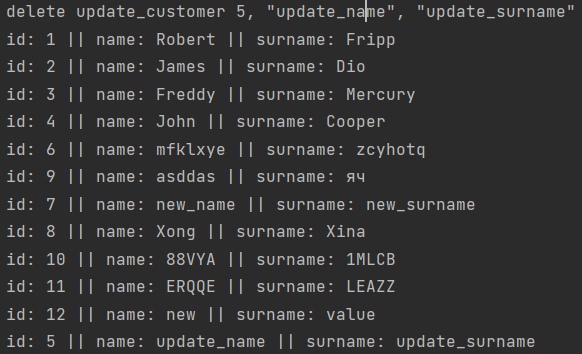
Видалення запису:



Вставка запису:



Редагування запису:



Пошук за двома атрибутами у двох таблицях:

[order\_id: 1 || date: 2019-12-20 || customer\_id: 1 || discount\_card\_id: 1, order\_id: 2 || date: 2020-03-09 || customer\_id: 2 || discount\_card\_id: 3, order\_id: 3 || date: 2020-11-30 || customer\_id: 1 || discount\_card\_id: 2, order\_id: 4 || date: 2021-06-20 || customer\_id: 3 || discount\_card\_id: 4]

**Завдання 2**

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

*GIN*

GIN 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 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. Цей індекс може взаємодіяти тільки з полем типу tsvector.

**SQL запити**

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:

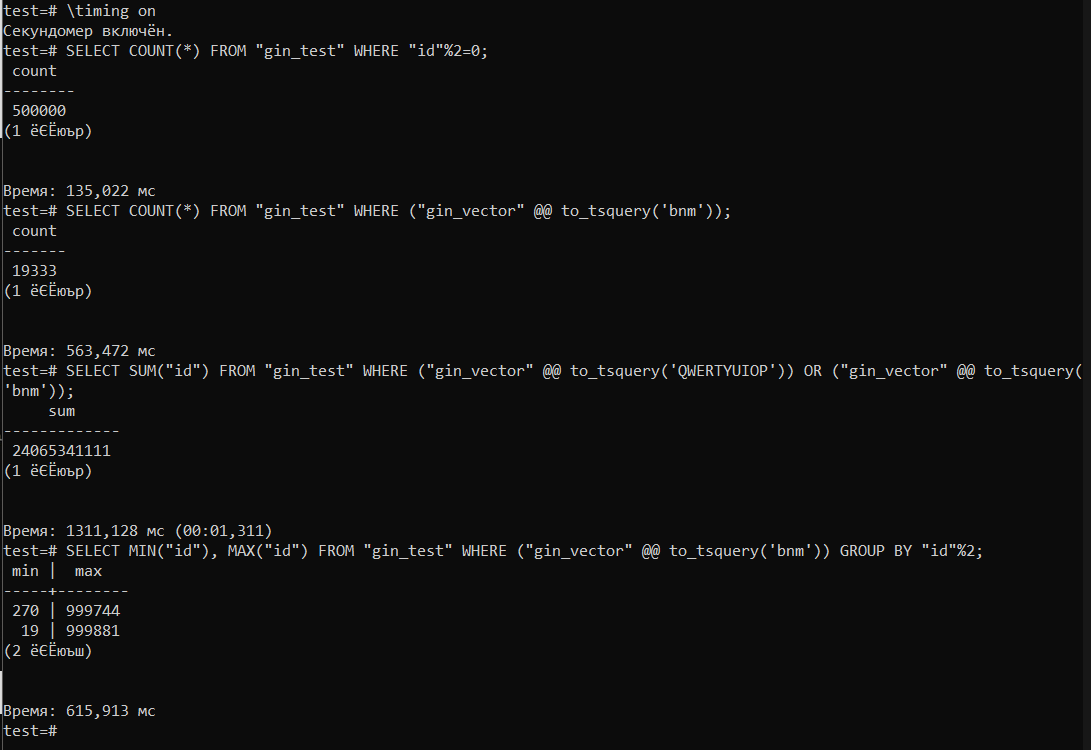


Рисунок 2 – Запити без індексування

**Час виконання запитів**

Таблиця 1 – 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 |

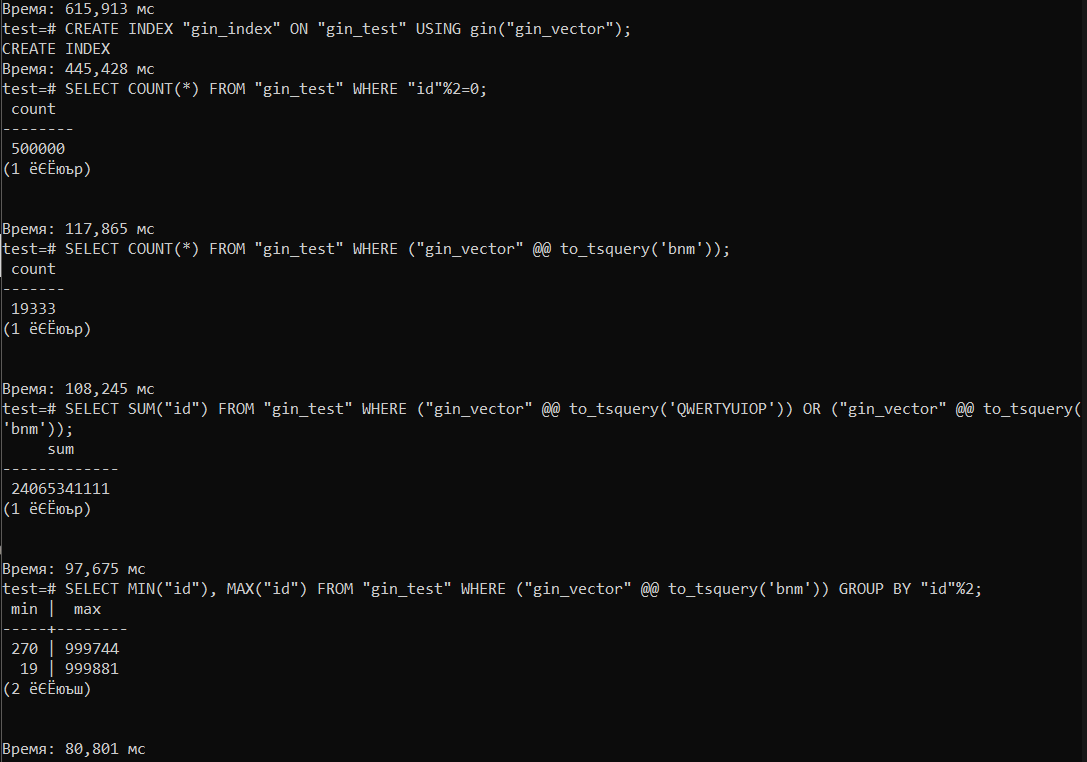


Рисунок 3 – Запити з індексуванням

**Час виконання запитів**

Таблиця 2 – 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. Іншими словами, для типу даних мають бути визначені оператори «більше», «більше або дорівнює», «менше», «менше або дорівнює» та «дорівнює». 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 до тих пір, поки не буде знайдено 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 (рис.18 та рис.21); 2 - виведення записів, у яких час більше або дорівнює 2019-10-01 (рис.19 та рис.21); 3 - виведення середнього значення ідентифікаторів записів, у яких час знаходиться в проміжку між 2019-10-01 та 2021-12-7 (рис.19 і рис.21); 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 (рис.20 і рис.21).

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:

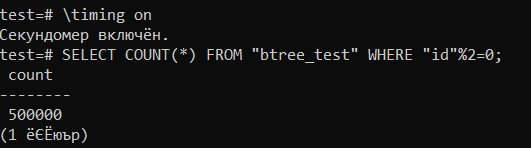


Рисунок 4 – Запит 1

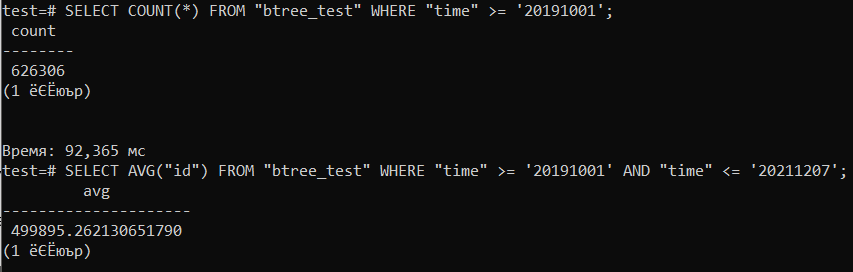


Рисунок 5 – Запит 2 і 3

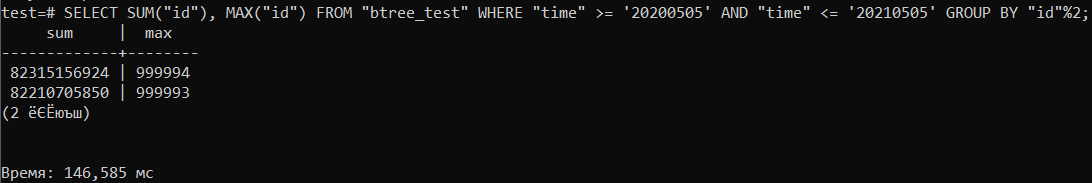


Рисунок 6 – Запит 4

**Час виконання запитів**

Таблиця 3 – 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:

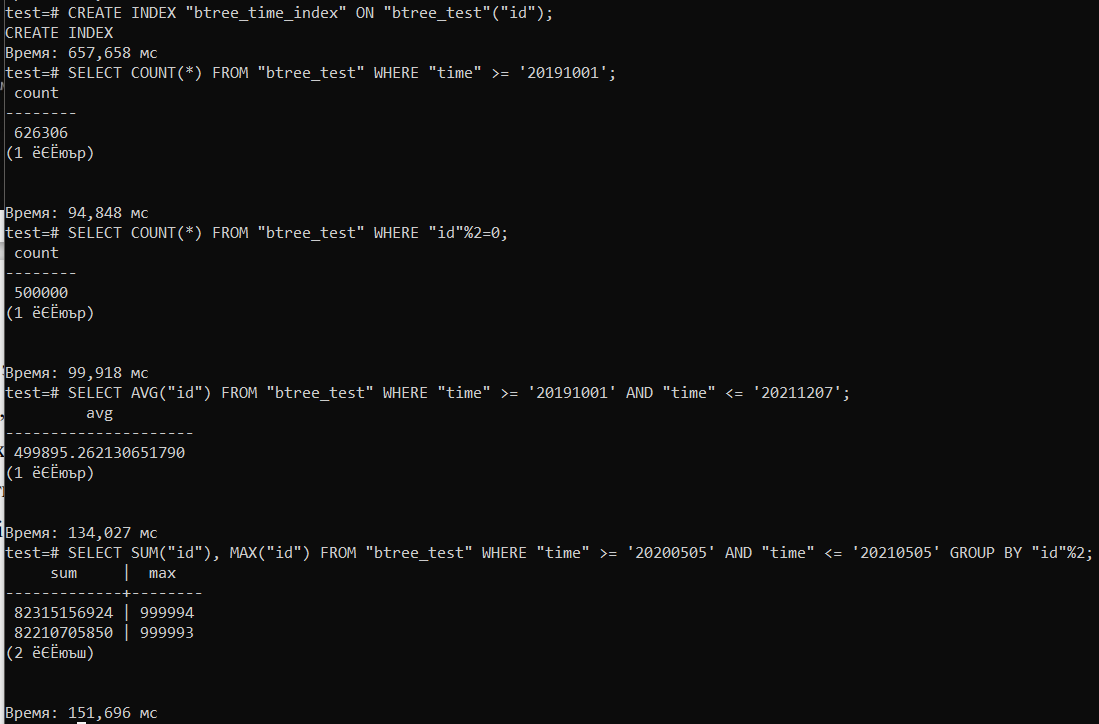


Рисунок 7 – Запити з індексуванням

**Час виконання запитів**

Таблиця 4 – 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.

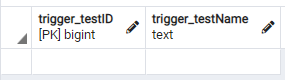


Рисунок 8 – Початковий стан таблиці trigger\_test

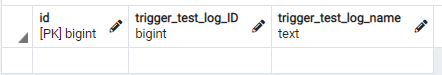


Рисунок 9 – Початковий стан таблиці trigger\_test\_log

Далі було виконано запит на вставку.

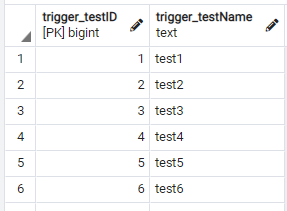
INSERT INTO trigger\_test("trigger\_testName") VALUES ('test7'), ('test8');

Рисунок 10 – Таблиця trigger\_test після виконання операції вставки

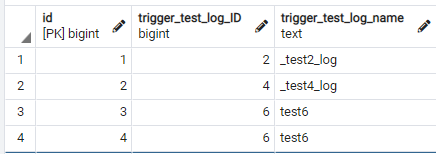


Рисунок 11 – Таблиця trigger\_test\_log після виконання операції вставки

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

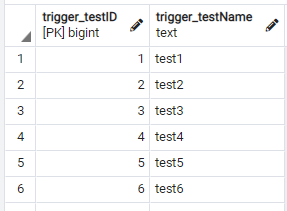


Рисунок 12 – Початковий стан таблиці trigger\_test

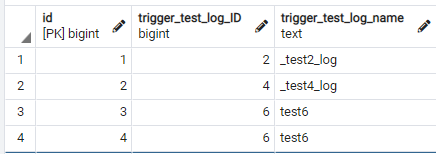


Рисунок 13 – Початковий стан таблиці trigger\_test\_log

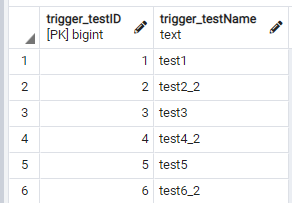


Рисунок 14 – Стан таблиці trigger\_test після виконання операції редагування

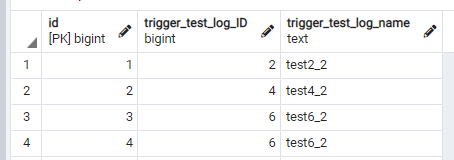


Рисунок 15 – Стан таблиці trigger\_test\_log після виконання операції редагування та спрацювання тригера

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.

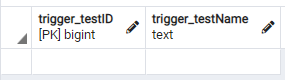


Рисунок 16 – Стан таблиці trigger\_test до почергової вставки та редагування

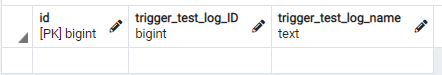


Рисунок 17 – Стан таблиці trigger\_test\_log до почергової вставки та

Редагування

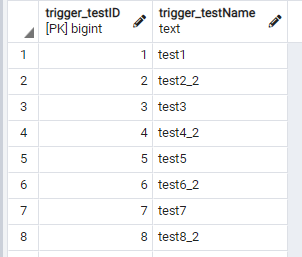


Рисунок 18 – Стан таблиці trigger\_test після вставки та редагування

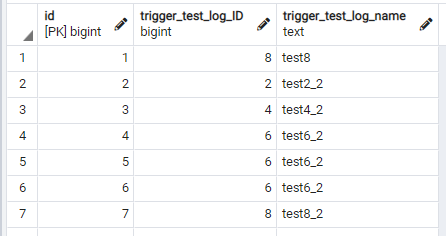


Рисунок 19 – Стан таблиці trigger\_test\_log після вставки та редагування

**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.

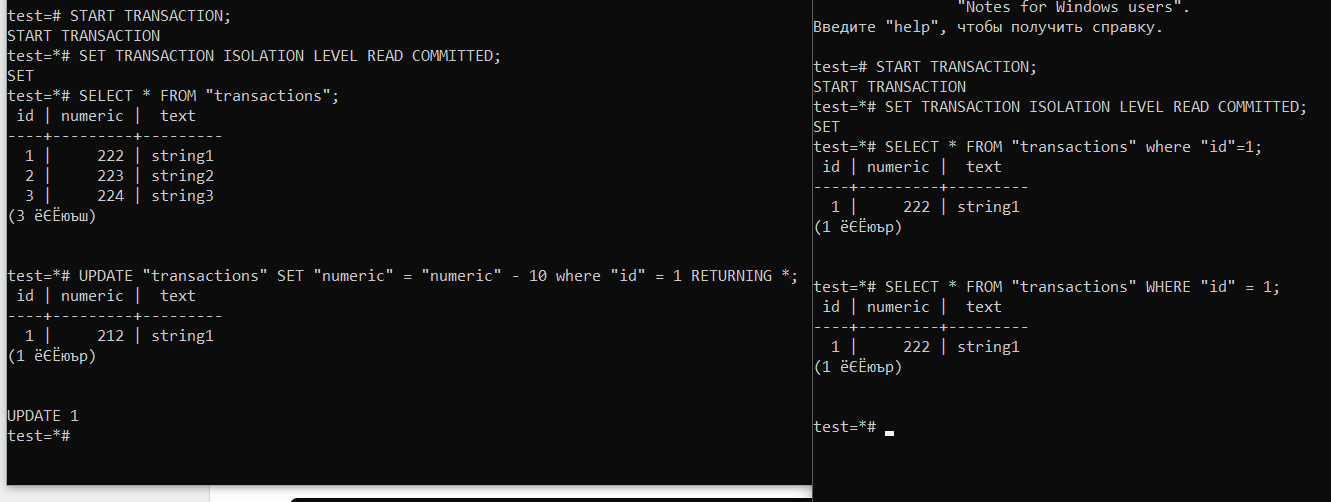


Рисунок 20 – Виконання редагування в одній з одночасних транзакцій на рівні ізоляції Rad committed

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.

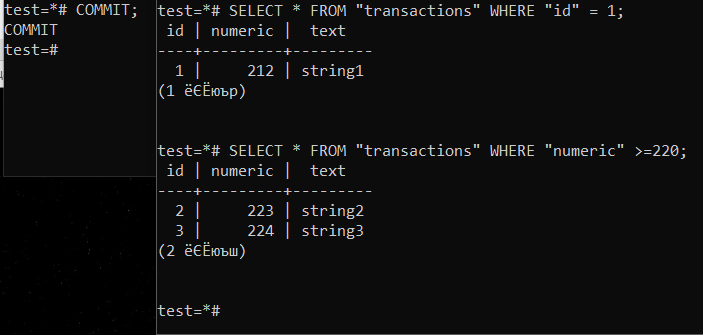


Рисунок 21 – Феномен «фантомного читання»

*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 сталося через використання рівня ізоляції repeatable read, тобто один і той самий запит має повертати той самий результат. Це призводить до того, що феномен «неповторного читання» неможливий на цьому рівні ізоляції.

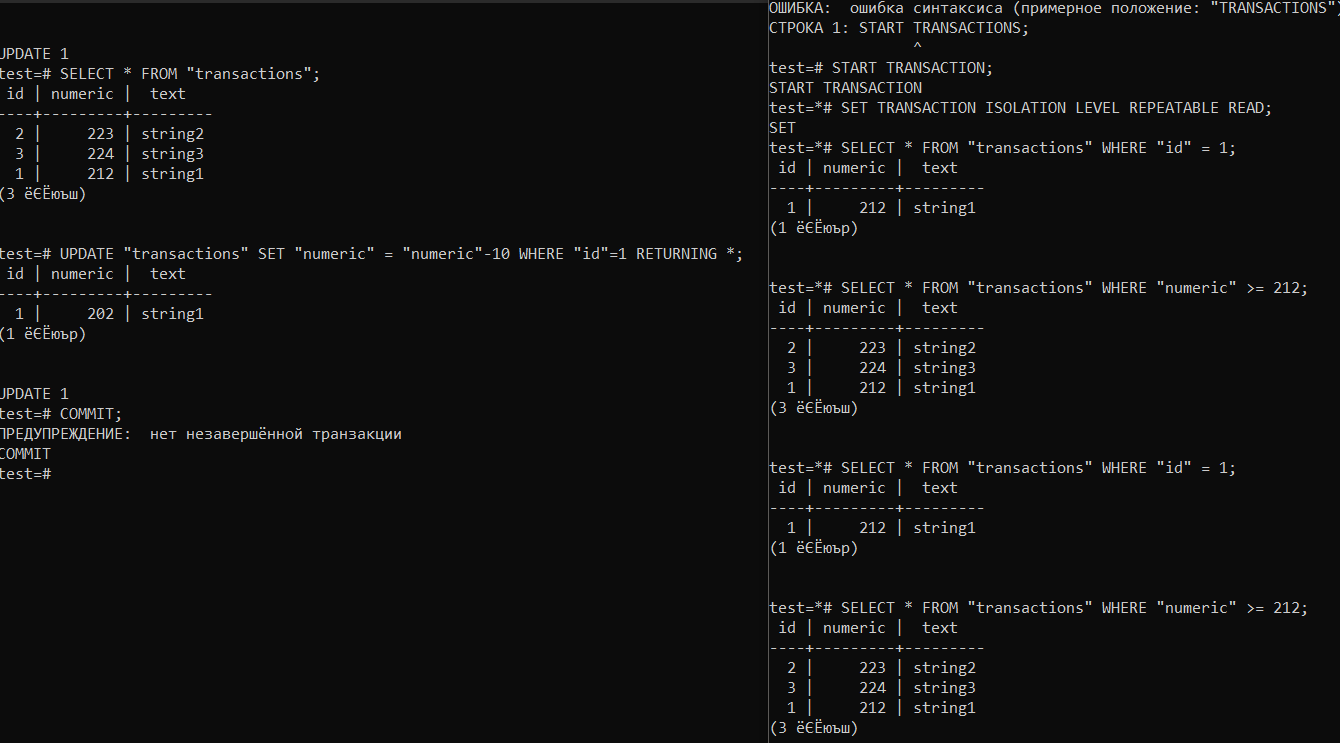


Рисунок 22 – Спроба одночасного внесення змін у таблиці на рівні ізоляції Repeatable read

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.

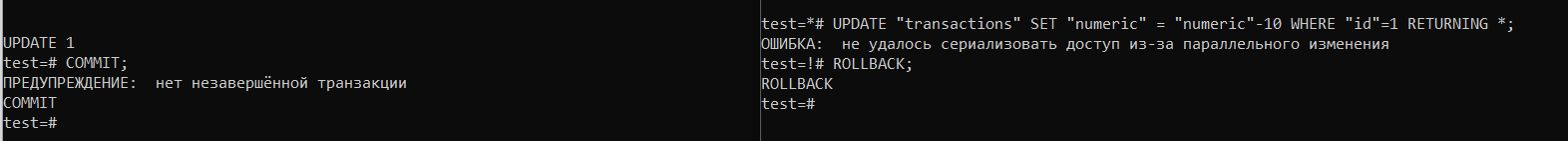


Рисунок 23 – Помилка через паралельні зміни на рівні ізоляції Repeatable read

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.

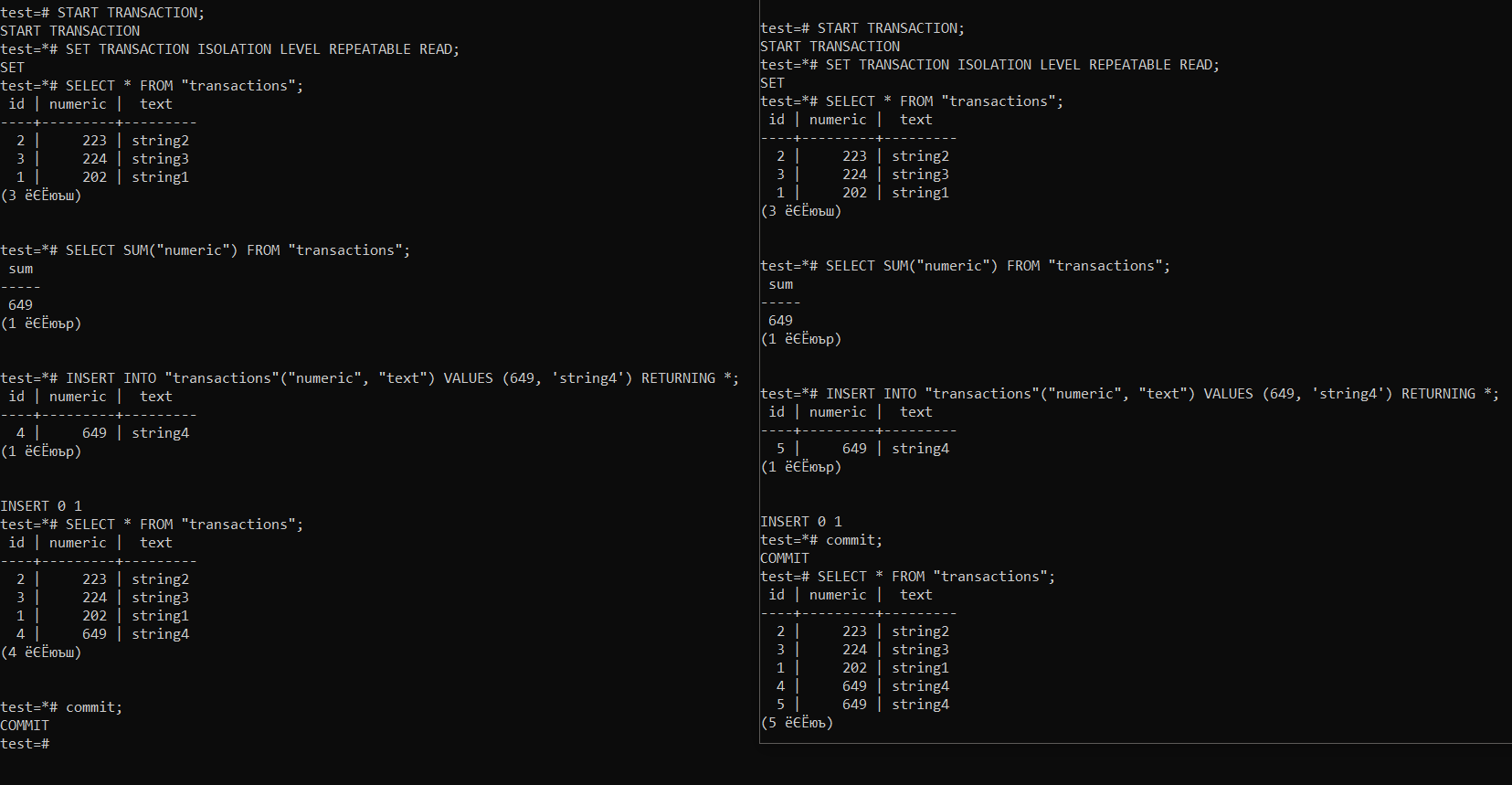


Рисунок 24 – Аномалія серіалізації

*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.

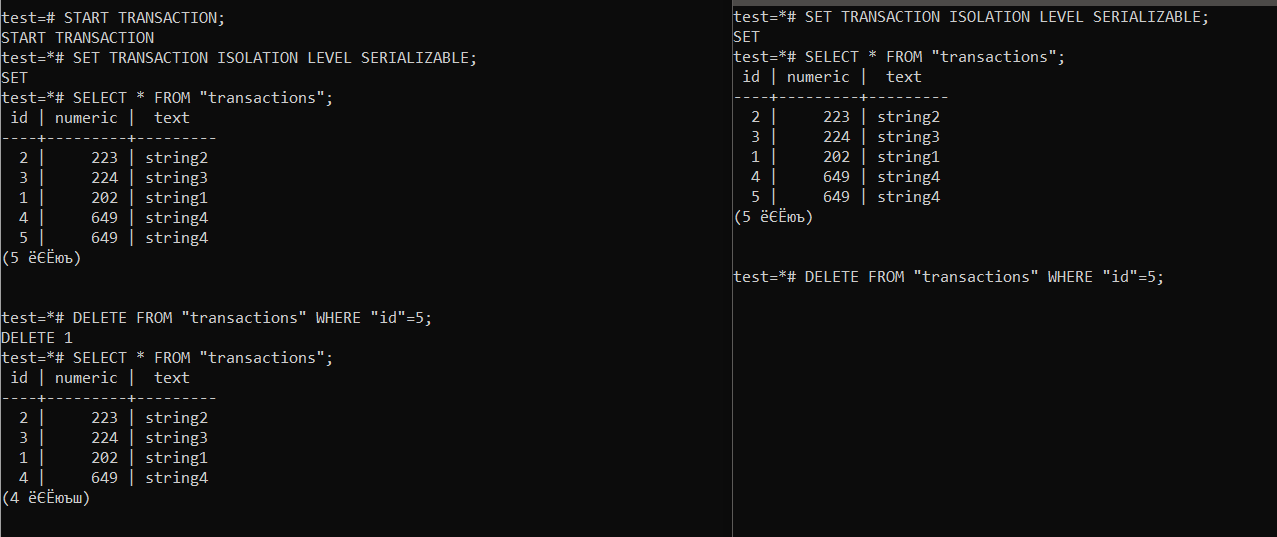


Рисунок 25 – Спроба одночасного виконання запитів на рівні ізоляції Serializable

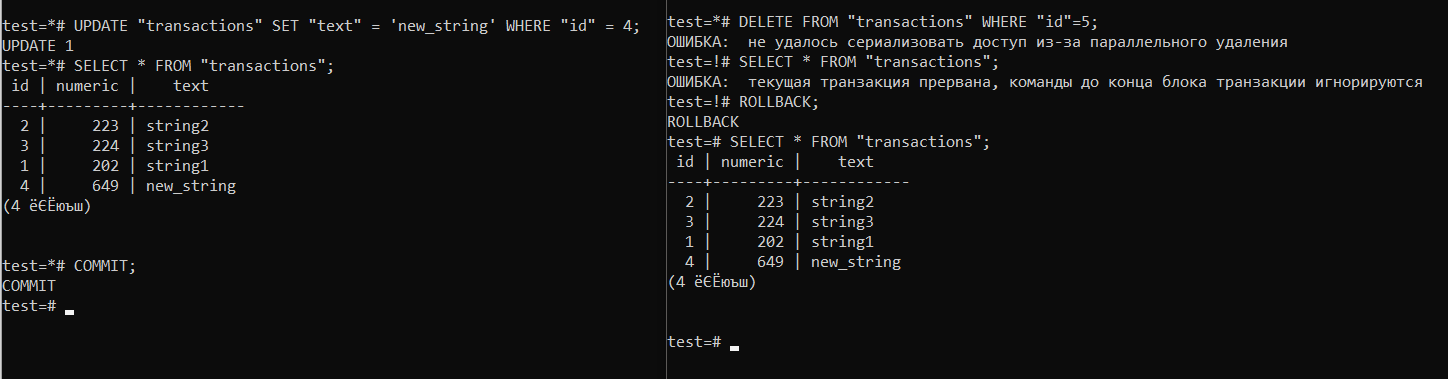


Рисунок 26 – Помилка через паралельне вилучення запису на рівні ізоляції Serializable