

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Лабораторна робота №3**

з дисципліни

**«Бази даних і засоби управління»**

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

Виконав: студент 3 курсу

Група: КВ-03

Дунець А.М

Перевірив: Петрашенко А.В.

Київ – 2023

*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.

Репозиторій github: <https://github.com/dynetskov/dataBases.git>

|  |  |  |
| --- | --- | --- |
| *7* | *GIN, BRIN* | *before insert, delete* |

**Завдання 2**

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

1000000 записів.

**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 та внесення даних в таблицю:

**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")

**WHERE** **TRUE**;

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");

Час виконання запитів без індексування:

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 101ms | 174ms | 938ms | 168ms |

Час виконання запитів з індексуванням:

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 128ms | 53ms | 72ms | 43ms |

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.

**BRIN** Індекси ефективні, якщо впорядкування ключових значень відповідає організації блоків на рівні зберігання. У найпростішому випадку для цього може знадобитися фізичне впорядкування таблиці, яке часто є порядком створення рядків у ній, щоб відповідати порядку ключа. Ключі згенерованих порядкових номерів або створених даних є найкращими кандидатами для індексу BRIN.

SQL-запити створення та внесення даних в таблицю:

**CREATE** **TABLE** brin\_test

(

**id** **int** **NOT** **NULL** **PRIMARY** **KEY**,

**date** **TIMESTAMP** **NOT** **NULL**,

**level** **INTEGER**,

msg TEXT

);

**INSERT** **INTO** brin\_test (**id**, **date**, **level**, msg)

**SELECT** g, **CURRENT\_TIMESTAMP** + (g || 'minute') :: interval, **random**() \* 6, md5(g::text)

**FROM** generate\_series(1, 8000000) **as** g;

Запити для тестування:

**SELECT** \* **FROM** "brin\_test" **WHERE** "id" % 2 = 1;

**SELECT** **COUNT**(\*) **FROM** "brin\_test" **WHERE** msg **LIKE** 'b2%' **AND** **date** > '2023-09-10';

**SELECT** **AVG**(**id**) **FROM** "brin\_test" **WHERE** **date** > '2024-01-01' **AND** **date** < '2024-12-01';

**SELECT** **SUM**(**id**), **MIN**(**id**) **FROM** "brin\_test" **WHERE** **date** > '2023-05-10' **AND** **date** <= '2024-10-01' **GROUP** **BY** msg **LIKE** '\_c7%';

Створення індексу:

**CREATE** **INDEX** "brin\_index" **ON** brin\_test **USING** brin(**date**);

Час виконання запитів без індексування:

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 703ms | 1497ms | 1852ms | 2023ms |

Час виконання запитів з індексуванням:

|  |  |  |  |
| --- | --- | --- | --- |
| Операція 1 | Операція 2 | Операція 3 | Операція 4 |
| 681ms | 3670ms | 270ms | 556ms |

З одержаних результатів бачимо, що в першому випадку час майже однаковий, оскільки на цей запит дана індексація не впливає. У другому випадку, відбувався відбір рандомно згенерованого тексту, тому тут дана індексація зіграла негативну роль і дала значно гірший результат. А в третьому і четвертому випадках запит складався саме із відбору впорядкованої інформації, тому і результат для BRIM виявився набагато кращий.

**Завдання 3**

Для тестування тригера було творено дві додаткові таблиці,

trigger\_test(id, first\_name, age) та trigger\_log(id, test\_id, old\_first\_name,

old\_age, changed\_on, is\_delete). Тригер повинен спрацьовувати при

командах додавання та видалення даних для першої таблиці – і записувати

видалені або додані дані в другу таблицю. Стовпець is\_delete в

таблиці trigger\_log зберігає логічний стан, true – над першою таблицею було

здійсненне видалення даних, false – було здійснене додавання даних.

SQL-запити створення та внесення даних в таблицю:

**CREATE** **TABLE** trigger\_test

(

**id** **INT** **GENERATED** ALWAYS **AS** **IDENTITY**,

first\_name **VARCHAR**(20) **NOT** **NULL**,

age **INT** **NOT** **NULL**

);

**CREATE** **TABLE** trigger\_log

(

**id** **INT** **GENERATED** ALWAYS **AS** **IDENTITY**,

test\_id **INT** **NOT** **NULL**,

old\_first\_name **VARCHAR**(20) **NOT** **NULL**,

old\_age **INT** **NOT** **NULL**,

changed\_on **TIMESTAMP**(6) **NOT** **NULL**,

is\_updated **BOOLEAN** **NOT** **NULL**

);

**INSERT** **INTO** trigger\_test(first\_name, age)

**VALUES** ('Vika', 18),

('Maks', 22),

('Vlad', 17),

('Andrii', 23);

Реалізація тригера:

**CREATE** **OR** **REPLACE** **FUNCTION** **trigger**()

**RETURNS** **TRIGGER** **AS**

$$

**BEGIN**

**IF** (tg\_op = 'INSERT') **THEN**

**IF** **NEW**.first\_name = 'Maks' **THEN**

**RAISE** **EXCEPTION** 'Forbidden insert';

ELSEIF **NEW**.first\_name <> 'Maks' **THEN**

**INSERT** **INTO** trigger\_log(test\_id, old\_first\_name, old\_age, changed\_on,

is\_updated)

**VALUES** (**NEW**.**id**, **NEW**.first\_name, **NEW**.age, now(),

'FALSE'::BOOLEAN);

**RETURN** **NEW**;

**END IF**;

**END IF**;

**IF** (tg\_op = 'DELETE') **THEN**

**IF** **OLD**.first\_name = 'Maks' **THEN**

**RAISE** **EXCEPTION** 'Forbidden delete';

ELSEIF **OLD**.first\_name <> 'Maks' **THEN**

**INSERT** **INTO** trigger\_log(test\_id, old\_first\_name, old\_age, changed\_on,

is\_updated)

**VALUES** (**OLD**.**id**, **OLD**.first\_name, **OLD**.age, now(),

'TRUE'::BOOLEAN);

**END IF**;

**RETURN** **OLD**;

**END IF**;

**END**;

$$ **LANGUAGE** PLPGSQL;

**CREATE** **TRIGGER** trigger\_1

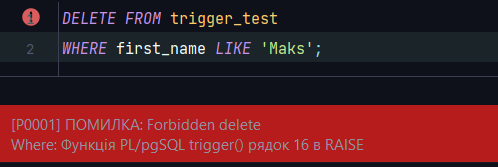
**BEFORE** **INSERT** **OR** **DELETE**

**on** trigger\_test

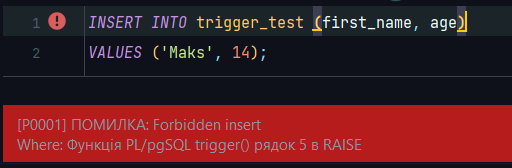
**FOR** **EACH** **ROW**

**EXECUTE** **PROCEDURE** **trigger**();

Видалення заборонених даних:



Вставка заборонених даних:

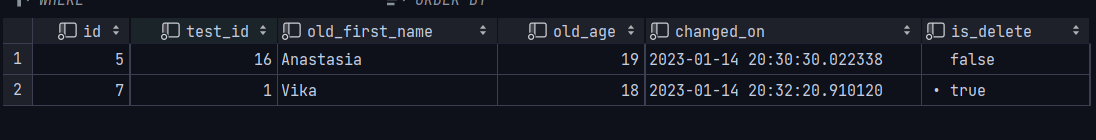


Вставка та видалення дозволених даних:

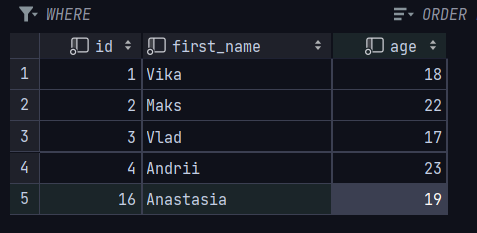
 

Вміст trigger\_log:



Вміст trigger\_test:



**Завдання 4**

**Read commited**

На цьому рівні ізоляції одна транзакція не бачить змін у базі даних,

викликаних іншою доки та не завершить своє виконання (командою

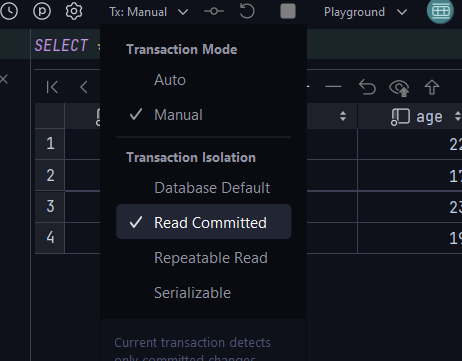
COMMIT або ROLLBACK).

Спочатку у транзакціях 1 і 2 таблиця має однаковий стан. Якщо у

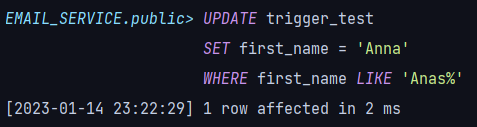
транзакції 1 виконати редагування одного рядка, то в транзакції 2 цих змін не

буде помітно, поки в першій транзакції не буде команди commit. Таким

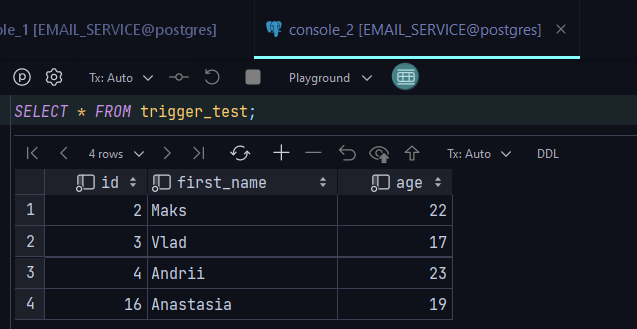
чином, феномен «брудного читання» на цьому рівні ізоляції неможливий.

Встановлення обох транзакцій в відповідний режим в JetBrains Clion

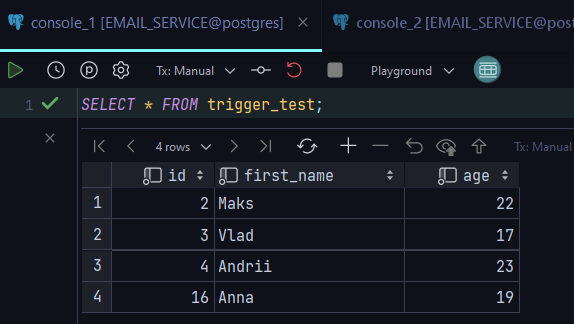
Виконуємо редагування в транзакції 1:



Вміст транзакції 2 не змінився:

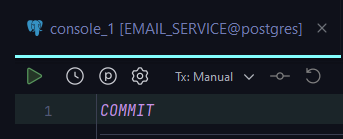


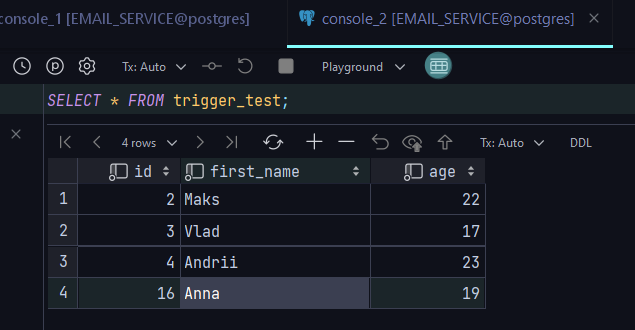
Проте транзакція 1 зберігає нові дані:



Феномен «фантомного читання»:

При виконанні операції COMMIT в транзакції 1 вміст буде змінено вміст транзакції 2:





**Repeatable read**

Почнемо дві транзакції на рівні ізоляції REPEATEBLE READ. У першій

транзакції обираємо запис з id = 3. Тепер змінемо данів в транзації 1 і

виконаємо команду COMMIT. У другій транзакції ніяких змін із цим рядком

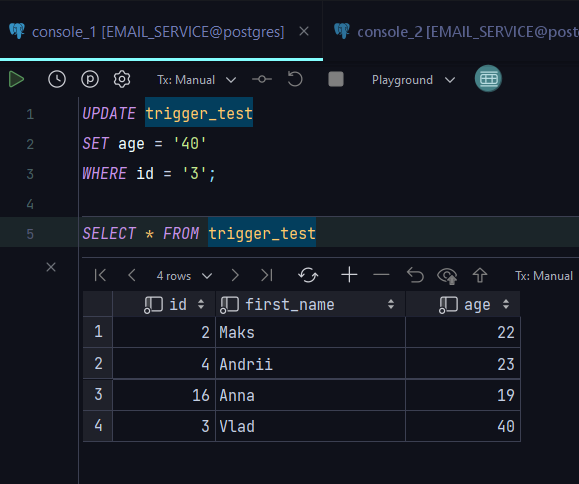
немає, хоча команда COMMIT була виконана. Це сталося через використання

рівня ізоляції REPEATEBLE READ, тобто один і той самий запит має

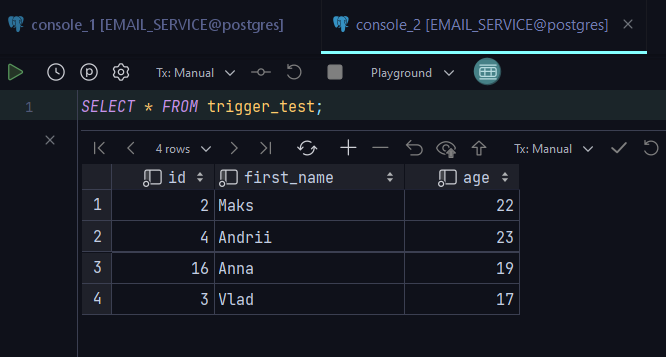
повертати той самий результат. Це призводить до того, що феномен

«неповторного читання» неможливий на цьому рівні ізоляції.

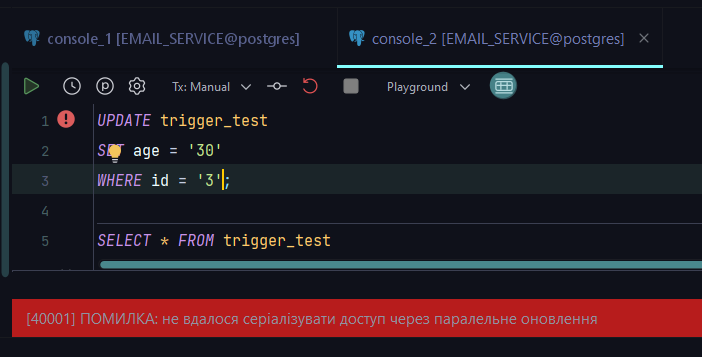
Оновлюємо дані та коммітимо їх:



Бачимо, що в транзакції два дані не змінились:



Якщо спробувати в цей час редагувати дані для другої транзакції, то отримаємо помилку:

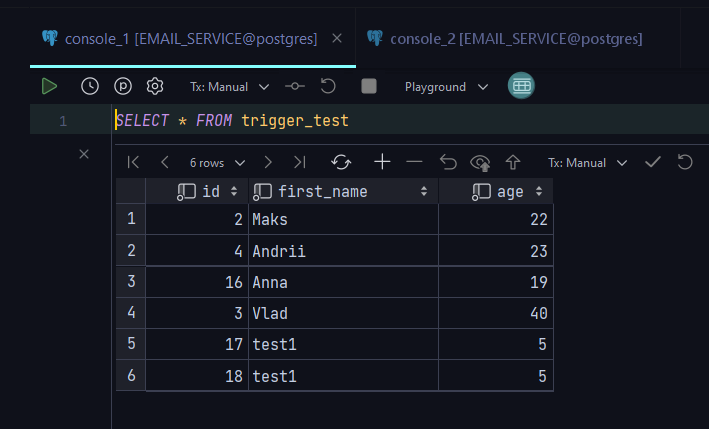


Це є перевагою рівня ізоляції REPEATABLE READ оскільки

захищає від повторного редагування даних

Тепер дослідимо аномалію серіалізації на рівні ізоляції REPEATABLE

READ. Для цього запустимо дві транзації. В першій додаємо новий рядок, після чого, робимо теж саме в другій транзакції, нам мала би висвітитись помилка, проте, цього не сталось і після коммітів, у нас буде два нових аналогічних рядки:



**SERIALIZABLE**

Запустимо дві транзакції на рівні ізоляції SERIALIZABLE. У першій

транзакції видалимо рядок. Якщо у другій транзакції спробувати

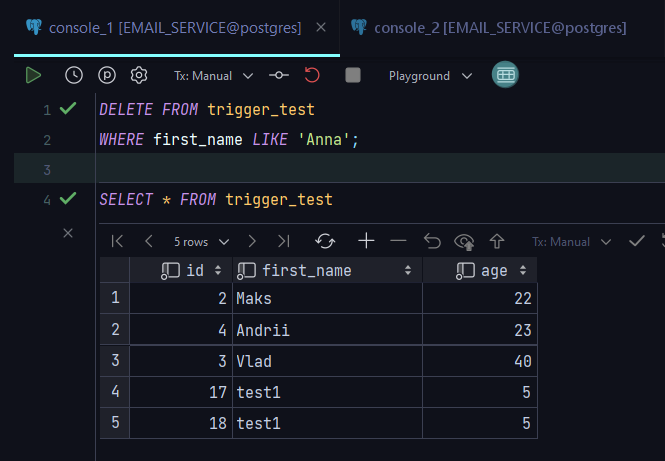
зробити ті ж операції над тими ж даними, то ми повинні будемо очікувати, доки перша транзакція не завершиться. Коли команда COMMIT у першій транзакції виконана, у другій виникає помилка через паралельне видалення. Це неможливо, оскільки якщо запис уже видалений в першій транзакції, то

видалити рядок з неіснуючим ідентифікатором неможливо. Виправити таку

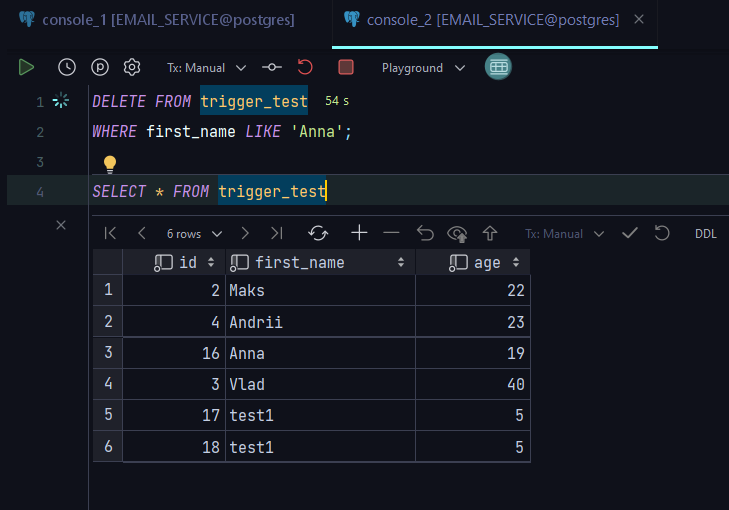
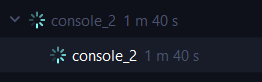
ситуацію можна з домогою команди ROLLBACK, після її виконання зміни

відбудуться і в другій транзакції.

Видалили рядок в транзакції 1:



При спробі здійснити теж саме в транзакції 2 переходимо на очікування завершення транзакції 1:

Після комміту першої транзакції, друга видасть помилку:

