

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»

Github: https://github.com/imaihs/bd\_lab/tree/main/lab3

Виконав: студент ІІI курсу ФПМ групи КВ-03

Черненький А.О.

Перевірив:

Київ – 2022

*Метою роботи* є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

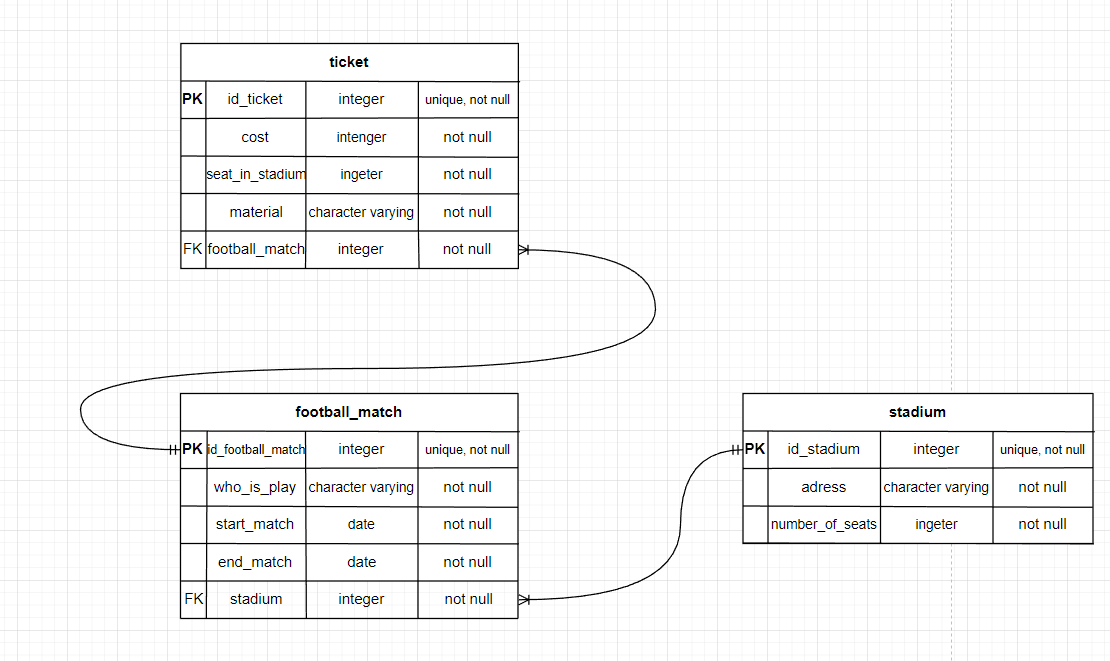
*Завдання* роботи полягає у наступному:

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. 

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом:

Git

# *Завдання 1*



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Сутність | Атрибут | Опис Атрибуту | Тип | Обмеженя |
| ticket | id | unique identifier | integer | not null  unique |
| cost | ціна в ₴ | integer | not null |
| seat\_in\_the\_stadium | Місце на стадіонні для вболівальник | integer | not null |
| material | Матеріал з якого виготовлено квиток | character varying | not null |
| football\_match | посилання на характеристику | integer | not null |
| football\_match | id | unique identifier | integer | not null  unique |
| who\_is\_play | які команди грають? | character  varying | not null |
| start\_match | час початку матчу | date | not null |
| end\_match | час кінця матчу | date | not null |
| stadium | посилання на характеристику | integer | not null |
| stadium | id | unique identifier | integer | not null  unique |
| adress | адреса стадіону | character varying | not null |
| number\_of\_seats | кількість місць | integer | not null |

Опис функціоналу меню:

Update – оновлення данних в певній колонці,

Add – додання нової колонки,

Delete – видалення нової колонки

Random - рандомна генерація нових значень для колонок на n раз

Search – пошук по базовим данним якоїсь колонки

Info about tables – повна інформація по бд

**Класи сутностей в ORM**

class Stadium(Base):  
 \_\_tablename\_\_ = 'stadium'  
 id\_stadium = Column('id\_stadium', Integer, primary\_key=True, autoincrement=True)  
 adress = Column('adress', String(99))  
 number\_of\_seats = Column('number\_of\_seats', Integer)  
 stadium\_football\_match = relationship('Football\_match', cascade='delete')  
  
 def \_\_repr\_\_(self):  
 return "<Stadium(id\_stadium='{}', adress='{}', number\_of\_seats='{}')>".format(self.id\_stadium,self.adress,  
 self.number\_of\_seats)  
  
  
class Football\_match(Base):  
 \_\_tablename\_\_ = 'football\_match'  
 id\_football\_match = Column('id\_football\_match', Integer, primary\_key=True, autoincrement=True)  
 who\_is\_play = Column('who\_is\_play', String(50))  
 start\_match = Column('start\_match', Date)  
 end\_match = Column('end\_match', Date)  
 stadium = Column('stadium', Integer, ForeignKey('Stadium.id\_stadium', onupdate='cascade'), primary\_key=True)  
 football\_match\_ticket = relationship('Ticket', cascade='delete')  
  
 def \_\_repr\_\_(self):  
 return "<Football\_match(id\_football\_match='{}', who\_is\_play='{}', start\_match='{}', end\_match='{}')>".format(  
 self.id\_football\_match,  
 self.who\_is\_play,  
 self.start\_match,  
 self.end\_match)  
  
  
class Ticket(Base):  
 \_\_tablename\_\_ = 'ticket'  
 id\_ticket = Column('id\_ticket', Integer, primary\_key=True, autoincrement=True)  
 cost = Column('cost', Integer)  
 seat\_in\_the\_train = Column('seat\_in\_the\_train', Integer)  
 material = Column('material', String(50))  
 football\_match = Column('football\_match', Integer, ForeignKey('Football\_match.id\_football\_match', onupdate='cascade'), primary\_key=True)  
  
 def \_\_repr\_\_(self):  
 return "Ticket(id\_ticket='{}', cost='{}', seat\_in\_the\_train='{}', material='{}'".format(self.id\_ticket,  
 self.cost,  
 self.seat\_in\_the\_train,  
 self.material)

# *Завдання 2*

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

*GIN*

GIN – так званий обернений індекс. Він працює з типами даних, значення яких не є атомарними, а складаються з елементів. При цьому індексуються не самі значення, а окремі елементи; кожен елемент посилається на ті значення, у яких він зустрічається. Індекс GIN зберігає набір пар виду: ключ, список появи ключа – де список появи — набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з’являється багато разів.

DROP TABLE IF EXISTS "GIN\_test";

*Запити мовою SQL*

CREATE TABLE "GIN\_test"("id" bigserial PRIMARY KEY, "doc" text, "doc\_tsv" tsvector); INSERT INTO "GIN\_test"("doc") SELECT chr(trunc(65 + random()\*25)::int)||chr(trunc(65 +

random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int)||chr(trunc(65 + random()\*25)::int) FROM generate\_series(1, 1000000) as q;

UPDATE "GIN\_test" SET "doc\_tsv" = to\_tsvector("doc");

- SELECT COUNT(\*) FROM "GIN\_test" where "id" % 11 = 0;

SELECT COUNT(\*) FROM "GIN\_test" WHERE ("doc\_tsv" @@ to\_tsquery('RQFWRFPGUJYIF'));

SELECT AVG("id") from "GIN\_test" where ("doc\_tsv" @@ to\_tsquery('WBHWIWGEFCYBP')) or ("doc\_tsv" @@ to\_tsquery('RQFWRFPGUJYIF'));

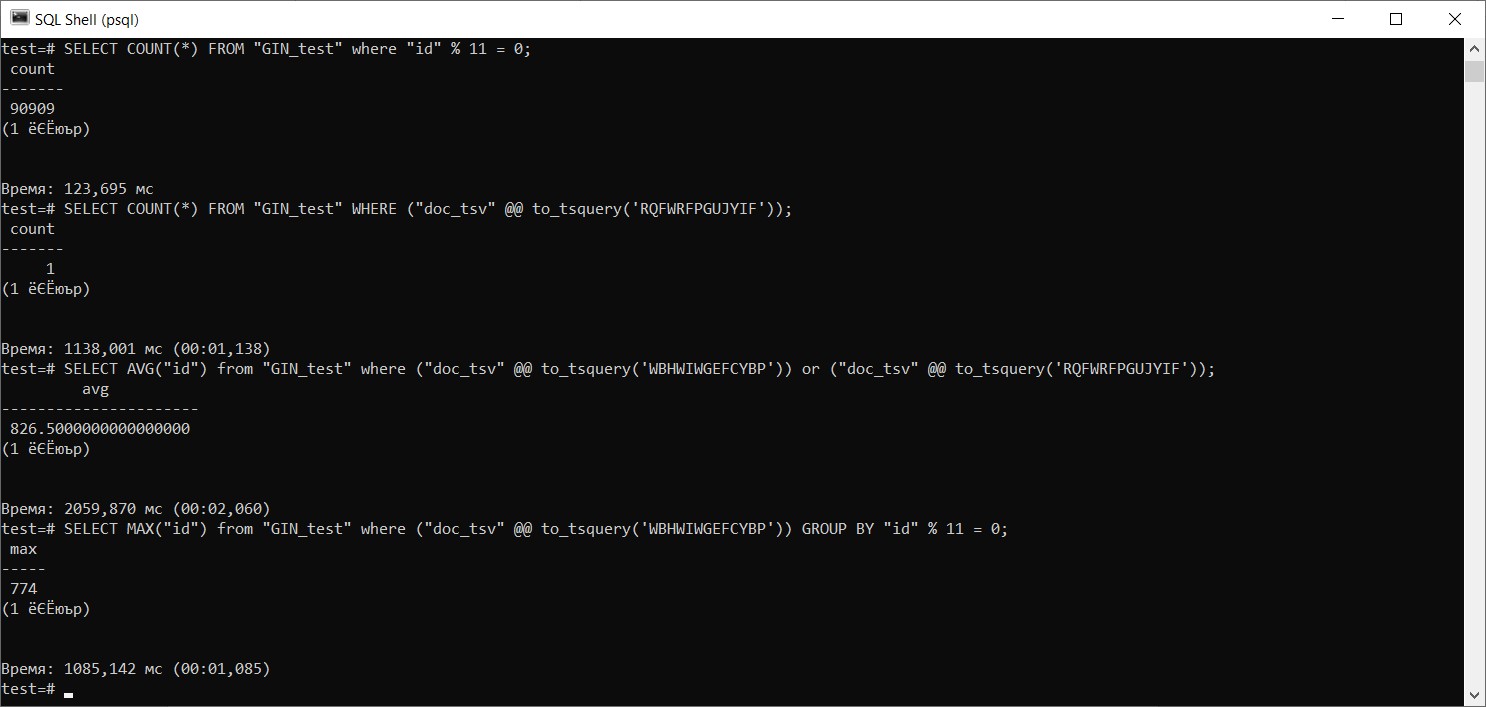
SELECT MAX("id") from "GIN\_test" where ("doc\_tsv" @@ to\_tsquery('WBHWIWGEFCYBP')) GROUP BY "id" % 11 = 0;

DROP INDEX IF EXISTS "GIN\_time\_index";

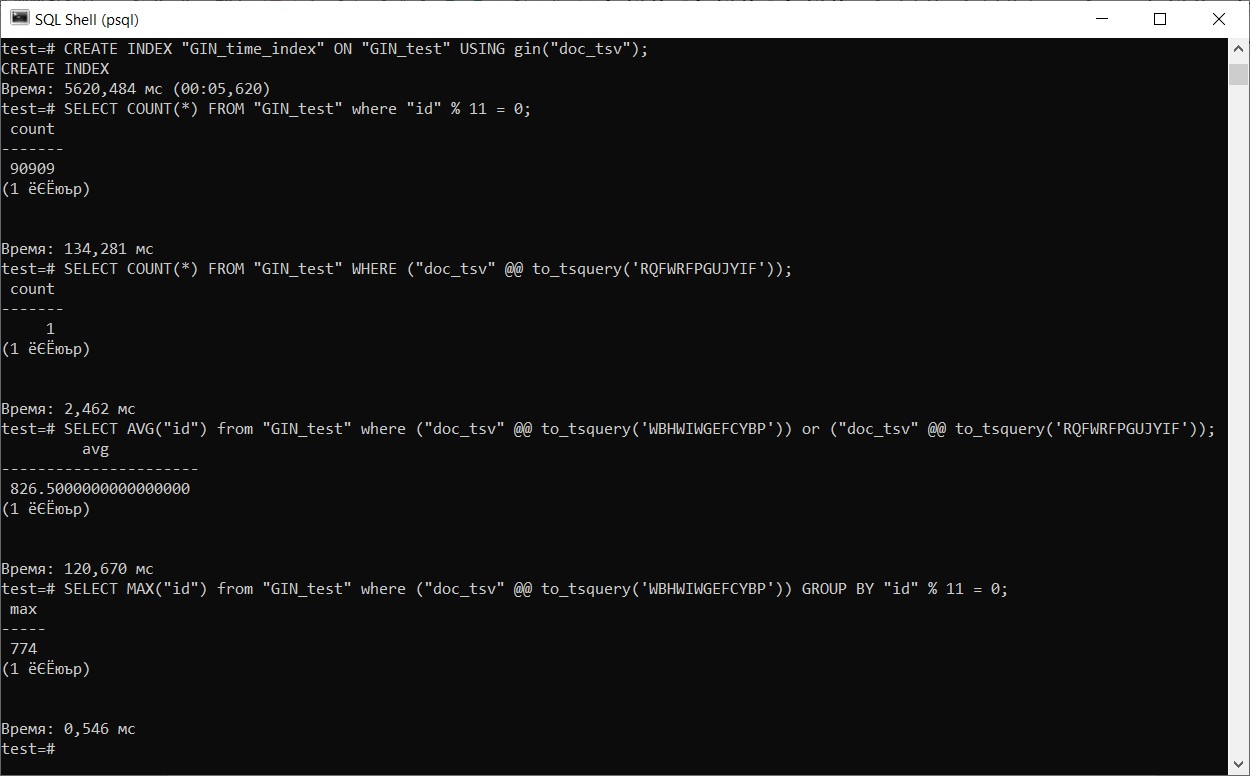
CREATE INDEX "GIN\_time\_index" ON "GIN\_test" USING gin("doc\_tsv");

*Перевірка результатів*

До індексування:



Після індексування:



З результатів бачимо, що використаня індексування GIN значно підвищило швидкість пошуку даних (окрім першого запиту, тому що на даний виклик індексування не впливає). В цілому, така поведінка є очікуваною, тому що основна ідея індексування GIN це те, що кожне значення шуканого ключа зберігається лише один раз і запит йде лише по тим даним, що містяться у списку появи цього ключа.

*BRIN*

BRIN – техніка індексації даних, призначена для обробки великих таблиць, в яких значення індексованого стовпця має деяку природну кореляцію з фізичним положенням рядка в таблиці. Вони мають такі якості партиціонованих таблиць, як швидка вставка рядка, швидке сворення індексу, без необхідності явного оголошення партицій. Спрощено кажучи, BRIN добре працює для тих стовпців, значення яких корелюють з їх фізичним розташуванням утаблиці.

Працює це наступним чином. Таблиця розбивається на зони (range) розміром кілька сторінок (або блоків, що те саме) — звідси й назва: Block Range Index, BRIN. Для кожної зони в індексі зберігаєтьсінформація про дані в цій зоні. Як правило, це мінімальне та максимальне значення, але буває інакше. Якщо при виконанні запиту, що містить умову на стовпець, шукані значення не потрапляють у діапазон, всю зону можна сміливо пропускати; якщо ж потрапляють - усі рядки у всіх блоках зони доведеться переглянути та вибрати серед них підходящі.

*Запити мовою SQL*

DROP TABLE IF EXISTS "BRIN\_test";

CREATE TABLE "BRIN\_test"("id" int PRIMARY KEY, "date" timestamp NOT NULL, "level" integer, "msg" text);

INSERT INTO "BRIN\_test"("id", "date", "level", "msg") SELECT q, CURRENT\_TIMESTAMP + ( q || 'minute' ) :: interval, random() \* 6, md5(q::text) FROM generate\_series(1,1000000) as q;

- SELECT COUNT(\*) FROM "BRIN\_test" where "date" between '2021-12-26 18:35:10.318196' and

'2021-12-27 02:03:10.318196';

SELECT MAX("date") FROM "BRIN\_test" where "date" between '2021-12-27 00:33:10.318196' and '2021-12-27 02:45:10.318196';

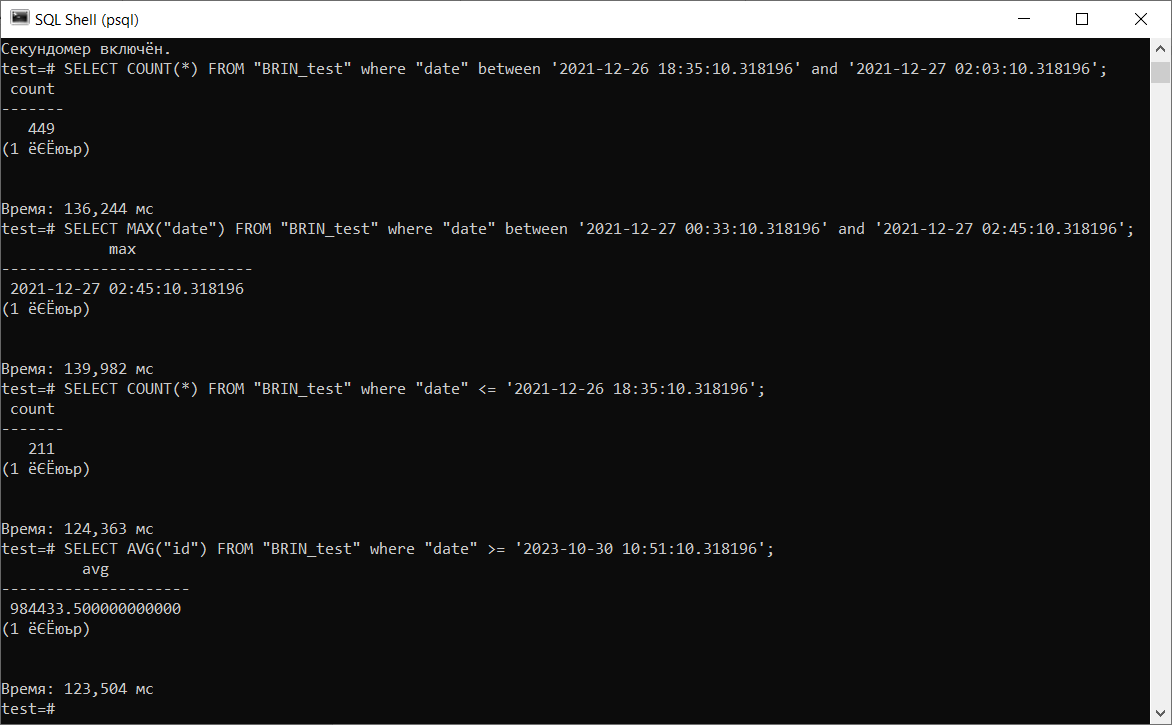
SELECT COUNT(\*) FROM "BRIN\_test" where "date" <= '2021-12-26 18:35:10.318196'; SELECT AVG("id") FROM "BRIN\_test" where "date" >= '2023-10-30 10:51:10.318196';

- DROP INDEX IF EXISTS "BRIN\_test\_index";

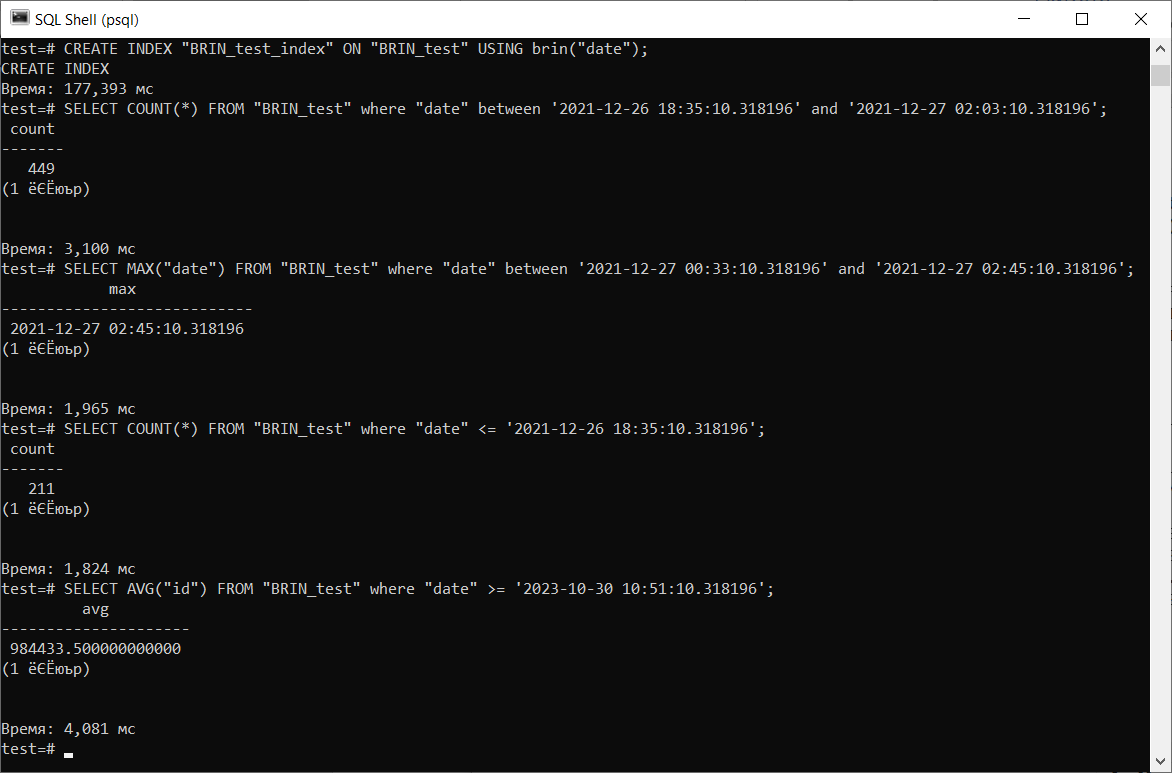
CREATE INDEX "BRIN\_test\_index" ON "BRIN\_test" USING brin("date");

*Перевірка результатів*

До індексування:



Після індексування:



З результатів, отриманих до і після використання індексування BRIN бачимо, що швидкість пошуку необхідних даних значно збільшилася. Знову ж таки, така поведінка є очікуваною, тому що індекси створені для пришвидшення пошуку необхідної інформації. Індекси BRIN ефективні, якщо впорядкування значень ключів відповідає організації блоків на рівні зберігання. У найпростішому впадку це може вимагати фізичного впорядкування таблиці, яке часто є порядком створення рядків у ній, щоб відповідати порядку ключа. Ключі до згенерованих порядкових номерів або створених даних є найкращими кандидатами для індексу BRIN.

# *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 "after\_update\_insert\_trigger" BEFORE DELETE OR UPDATE ON "trigger\_test" FOR EACH ROW

EXECUTE procedure after\_update\_insert\_func();

Текст тригера:

CREATE OR REPLACE FUNCTION after\_update\_insert\_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";

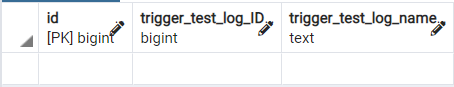
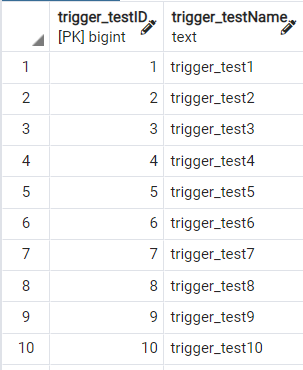
END LOOP; RETURN OLD;

END IF; END;

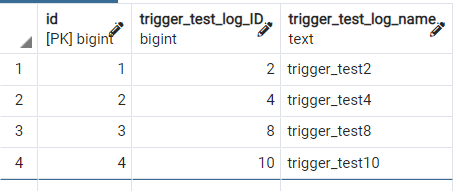
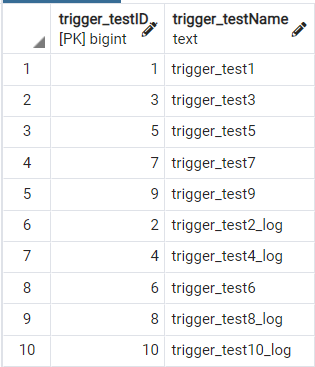
$trigger$ LANGUAGE plpgsql;

Скріншоти зі змінами у таблицях бази даних

Початковий стан

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

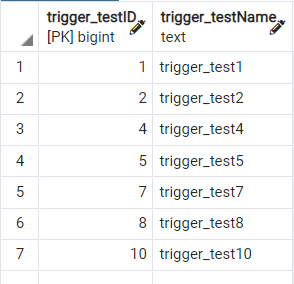
Після виконання запиту на оновлення

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 ядка він також виконавс, проте пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на видалення потрібн повертати новий стан, а при запиті на оновлення старий.

Після виконання запиту на видалення

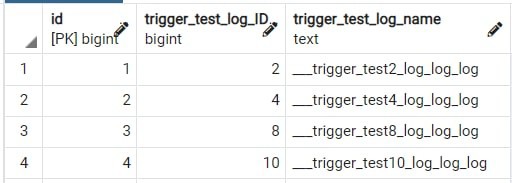
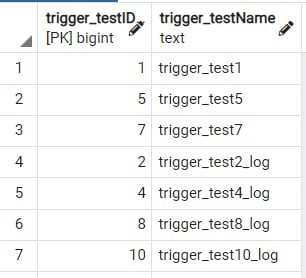
DELETE FROM "trigger\_test" WHERE "trigger\_testID" % 3 = 0;



Якщо виконувати ці запити окремо одне від одного, то у таблиці trigger\_test видаляються кратні трьом рядки, але таблиця trigger\_test\_log виявлєтья пустою. Так відбувається тому, що у гілці алгоритму для чисел кратних трьом у trigger\_test\_log лише модифуються існуючі записи, але нові не додаються.

Оскільки до цього не було виконан 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 завдки викоанню запит update для всіх паних рядків. А інші два "\_log" та два символи "\_" на початку додались тому, що запит на видалння для записів 3 та 9 виконались за тією самою гілкою алгоритму (кратні трьом), а запит на видалення зпису 6 виконався за іншою гілкою (кратність 2 та 3)

*Завдання 4*

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Самі транзакції особливих пояснень не вимагають, транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. Ізольованість транзакції показує те, наскільки сильно вони вливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у виборі між високою узгодженістю даних між транзакціями та швидкістю виконання цих транзакцій.

Варто зазначити, що найвищу швидкість виконання та найнижчу узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та найвищу узгодженість — serializable.

При паралельному виконанні транзкцій можливі виникненя таких проблем:

1. *Втрачене оновлення* Ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.

2. *«Брудне» читання*Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

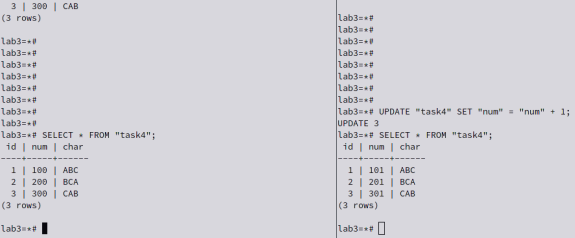
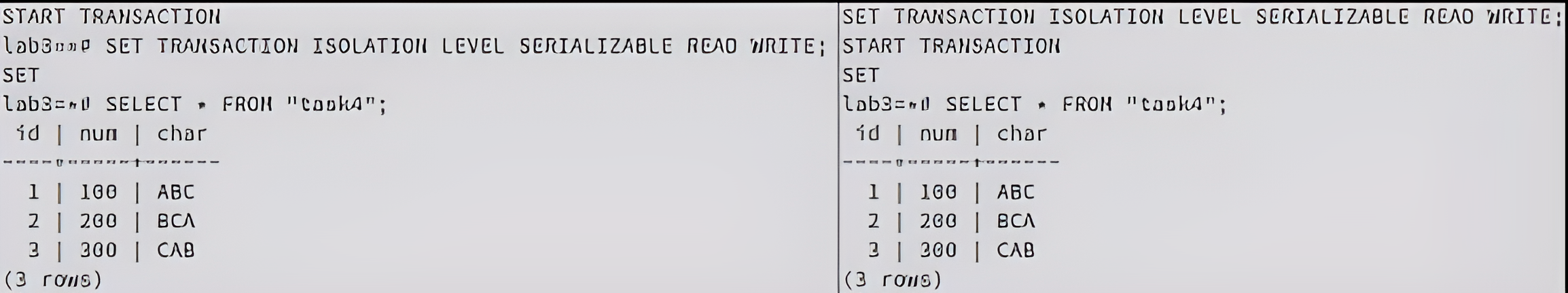
3. *Неповторюване читання* Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляюься зміненими.

4. *Фантомне читання* Ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

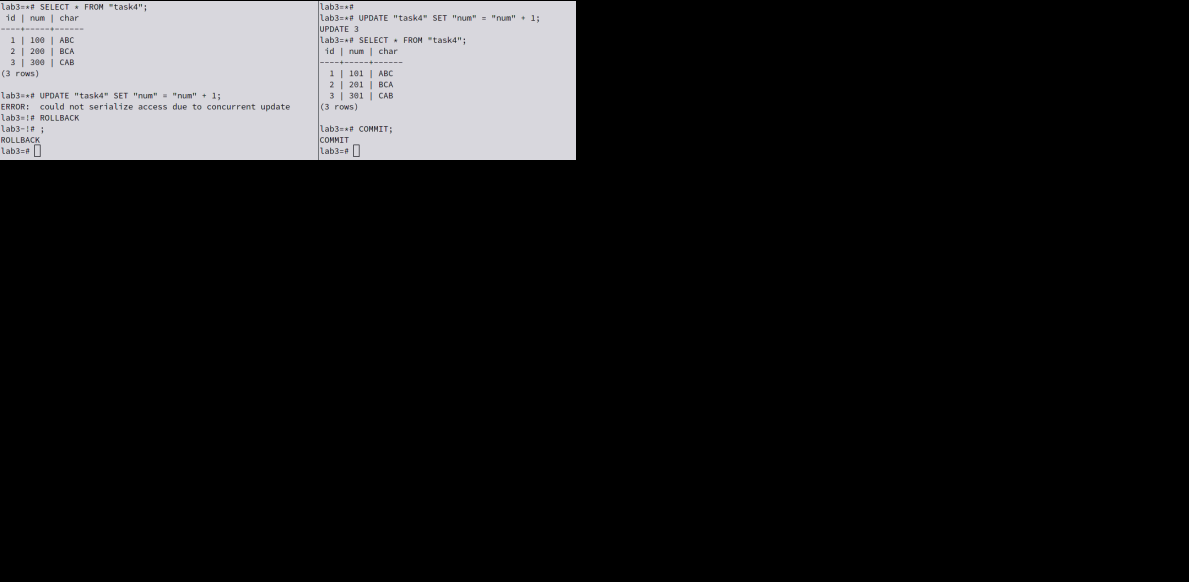
Стандарт SQL-92 визначає наступні рівні ізоляції:

1. *Serializable (впорядкованість)*

Найбільш високий рівень ізольованості; транзакції повністю ізолюються одна від одної. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку). Як бачимо, дані у транзаціях ізольовано.

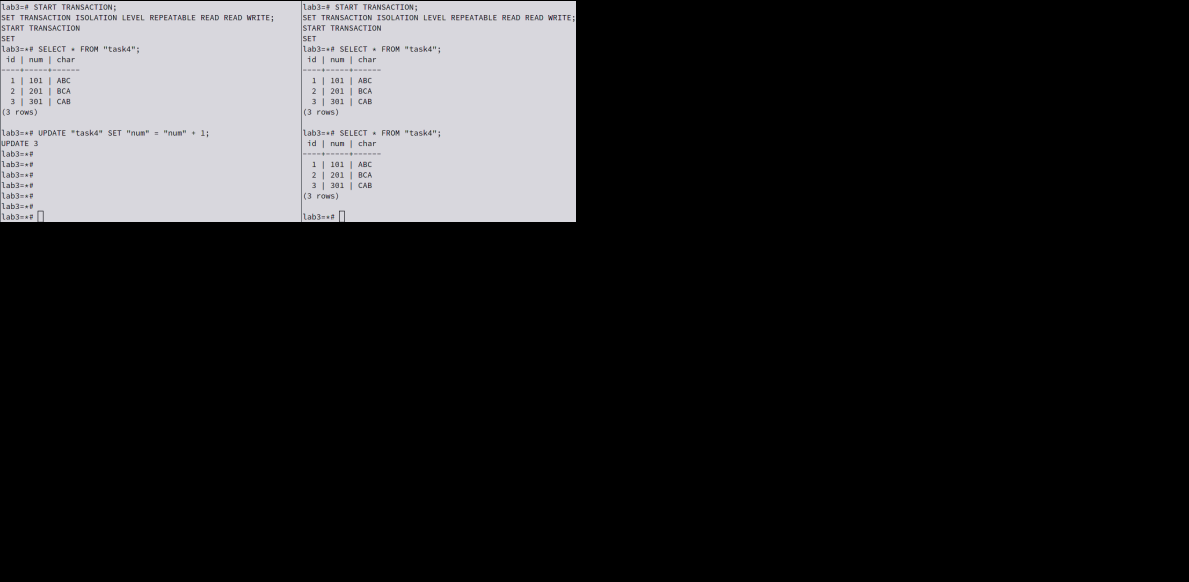


Тепер при оновленіданих в T2(частина фото зправа) бачимо, що T2 блокується поки T1 не не зафіксує зміни або не відмінить їх.

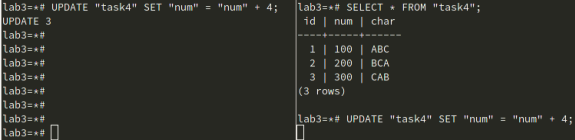


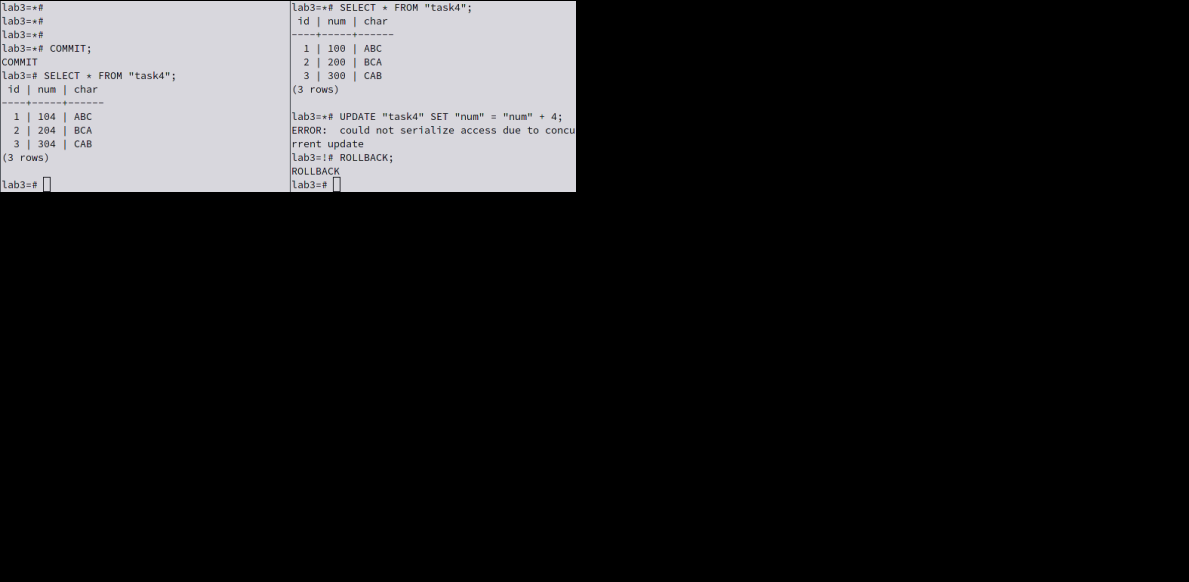
1. *Repeatable read (повторюваність читання)*

Рівень, при якому читання дного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані)



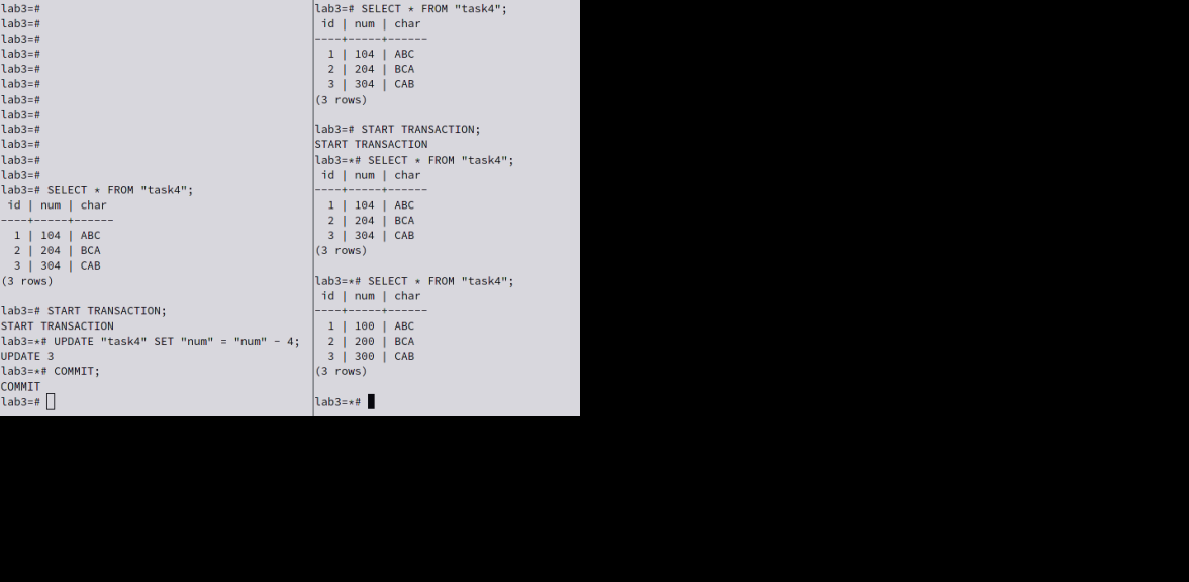
Тепер транзакція T2(зправа) буде чекати поки T1 не не зафіксує зміни або не відмінить їх.





Як бачимо, Repeatable read не дозволяє виконувати операції зміни даних, якщо дані вже було модифіковано у іншій незавершеній транзакції. Тому використання Repeatable read рекомендоване тільки для режиму читаня.

1. *Read committed (читання фіксованих даних)* Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша мое бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання.



*4.Read uncommitted*

(читання незафіксованих даних) Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той жерядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.