

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**”

Виконала: студентка 3 курсу

групи КВ-92

Невінчаний А. В.

Перевірив: Павловський В.І.

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 |
| 26 | *BTree, BRIN* | *before update, delete* |

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

<https://github.com/nutasanchik/DB_Lab3>

Завдання №1

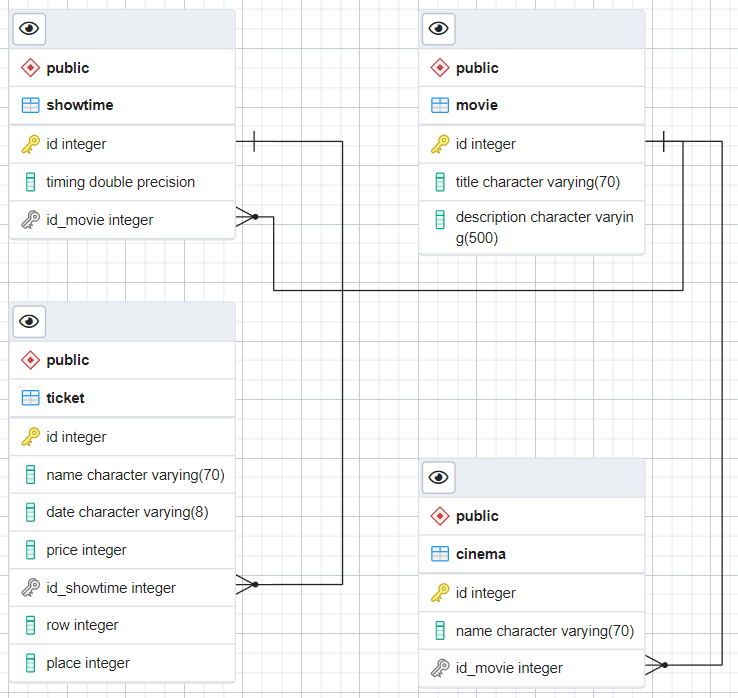


Рисунок 1 - Схема бази даних у pgAdmin 4 Таблиці бази даних у середовищі PgAdmin4

BEGIN;

CREATE TABLE IF NOT EXISTS public."Cinema" (

id integer NOT NULL,

name character varying(70) NOT NULL, id\_movie integer NOT NULL,

PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Movie" (

id integer NOT NULL,

title character varying(70) NOT NULL, description character varying(500) NOT NULL, PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Showtime" (

id integer NOT NULL, timing integer NOT NULL, id\_movie integer NOT NULL, PRIMARY KEY (id)

);

CREATE TABLE IF NOT EXISTS public."Ticket" (

id integer NOT NULL,

name character varying(70) NOT NULL, date character varying(8) NOT NULL, price integer NOT NULL,

id\_showtime integer NOT NULL, "row" integer NOT NULL,

place integer NOT NULL, PRIMARY KEY (id)

);

ALTER TABLE public."Cinema" ADD FOREIGN KEY (id\_movie)

REFERENCES public."Movie" (id) NOT VALID;

ALTER TABLE public."Showtime" ADD FOREIGN KEY (id\_movie)

REFERENCES public."Movie" (id) NOT VALID;

ALTER TABLE public."Ticket"

ADD FOREIGN KEY (id\_showtime) REFERENCES public."Showtime" (id) NOT VALID;

END;

Класи ORM у реалізованому модулі Model

class Cinema(Orders):

tablename = 'Cinema'

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

name = Column(String)

id\_movie = Column(Integer, ForeignKey('Movie.id')) movies = relationship("Movie")

def init (self, key, name, id\_movie): self.id = key

self.name = name self.id\_movie = id\_movie

def repr (self):

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

.format(self.id, self.name, self.id\_movie)

class Movie(Orders):

tablename = 'Movie'

id = Column(Integer, primary\_key=True) title = Column(String)

description = Column(String)

def init (self, key, title, description): self.id = key

self.title = title self.description = description

def repr (self):

return "{:>10}{:>15}{:>50}" \

.format(self.id, self.title, self.description)

class Showtime(Orders):

tablename = 'Showtime'

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

timing = Column(Float)

id\_movie = Column(Integer, ForeignKey('Movie.id')) movies = relationship("Movie")

def init (self, key, timing, id\_movie): self.id = key

self.timing = timing self.id\_movie = id\_movie

def repr (self):

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

.format(self.id, self.timing, self.id\_movie)

class Ticket(Orders):

tablename = 'Ticket'

id = Column(Integer, primary\_key=True) name = Column(String)

date = Column(String) price = Column(Integer)

id\_showtime = Column(Integer, ForeignKey('Showtime.id')) showtimes = relationship("Showtime")

row = Column(Integer) place = Column(Integer)

def init (self, key, name, date, price, id\_showtime, row, place):

self.id = key self.name = name self.date = date self.price = price

self.id\_showtime = id\_showtime self.row = row

self.place = place

def repr (self):

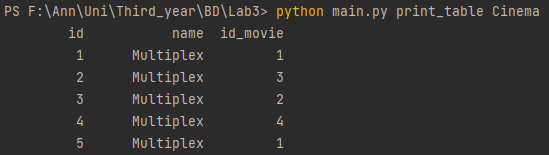
return "{:>10}{:>15}{:>15}{:>10}{:>10}{:>10}{:>10}" \

.format(self.id, self.name, self.date, self.price, self.id\_showtime, self.row, self.place)

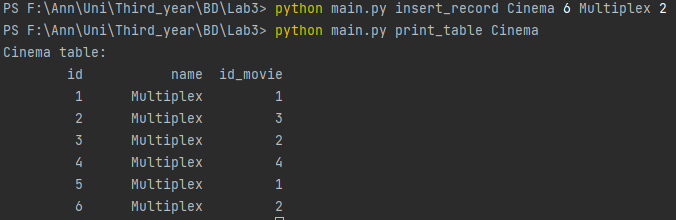
Запити у вигляді ORM

Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Cinema.

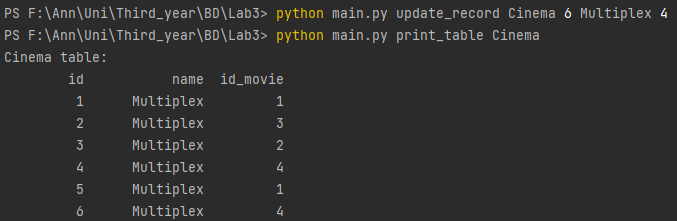
Початковий стан:



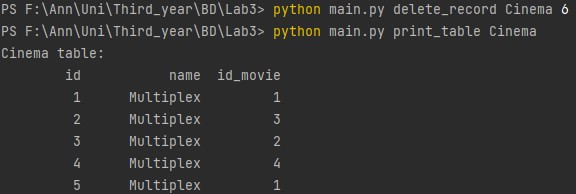
Вставка запису:



Редагування запису:



Видалення запису:

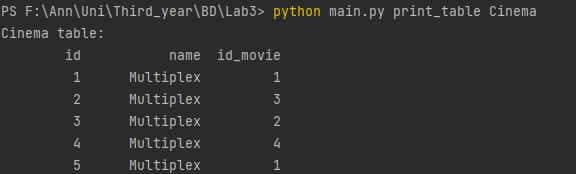


Запити пошуку та генерації рандомізованих даних також було реалізовано, логіку пошуку було змінено у порівнянні з лабораторною

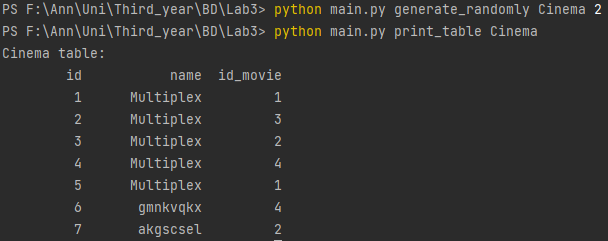
роботою №2 (усі дані для пошуку передвизначено, тепер вони не вводяться з клавіатури). Запити на пошук ті самі, що і л.р. №2.

Запит на генерацію даних продемонструємо на прикладі таблиці Cinema.

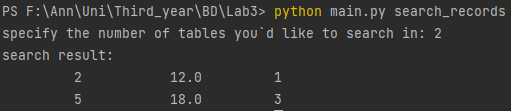
Початковий стан:

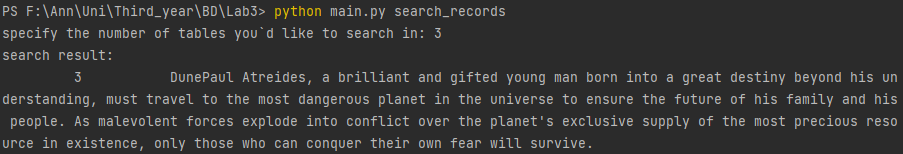


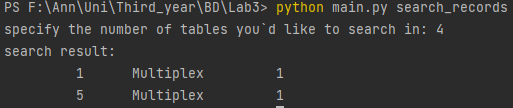
Вставка 2-х випадково згенерованих записів:



Пошук за трьома атрибутами у двох таблицях, за трьома атрибутами у трьох таблицях, за чотирма атрибутами у чотирьох таблицях (виводяться відповідні записи з таблиці Showtime, Movie, Cinema віповідно):







Завдання 2 BTree

Для дослідження індексу була створена таблиця, яка має дві колонки:

числову та текстову. Вони проіндексовані як BTree. У таблицю було занесено 1000000 записів.

Створення таблиці та її заповнення:

DROP TABLE IF EXISTS "test\_btree"; CREATE TABLE "test\_btree"(

"id" bigserial PRIMARY KEY, "test\_text" varchar(255)

);

INSERT INTO "test\_btree"("test\_text") SELECT

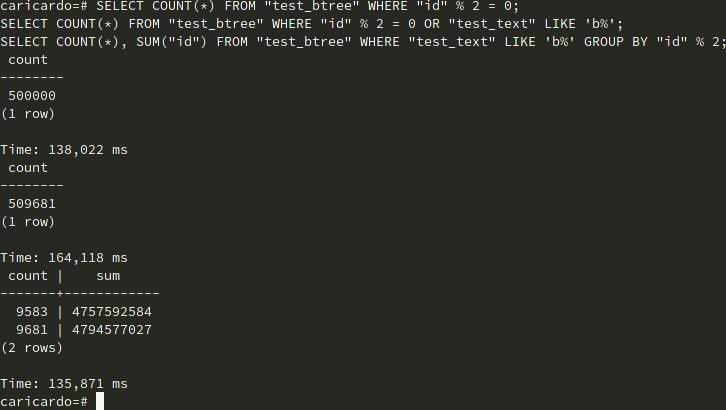
substr(characters, (random() \* length(characters) + 1)::integer, 10)

FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as

symbols(characters), generate\_series(1, 1000000) as q;

Вибір даних без індексу:

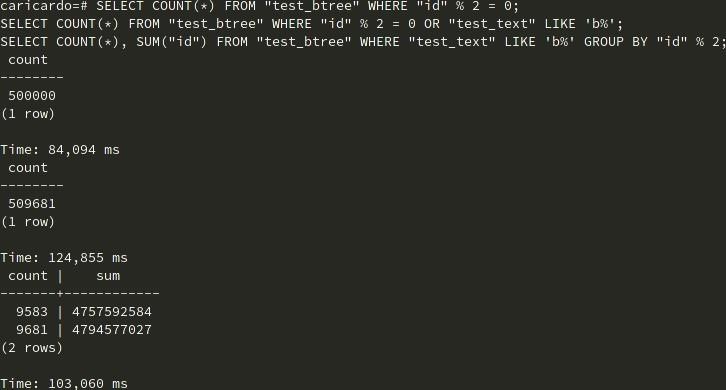


Створюємо індекс:

DROP INDEX IF EXISTS "test\_btree\_test\_text\_index";

CREATE INDEX "test\_btree\_test\_text\_index" ON "test\_btree" USING btree ("test\_text");

Вибір даних з створеним індексом:



BRIN

Для дослідження індексу була створена таблиця, яка має дві колонки: t\_date типу timestamp without time zone (дата та час (без часового поясу)) і t\_number типу integer. Колонка t\_data проіндексована як BRIN. У таблицю занесено 1000000 записів.

Створення таблиці та її заповнення:

DROP TABLE IF EXISTS "test\_brin"; CREATE TABLE "test\_brin"(

"id" bigserial PRIMARY KEY, "test\_time" timestamp

);

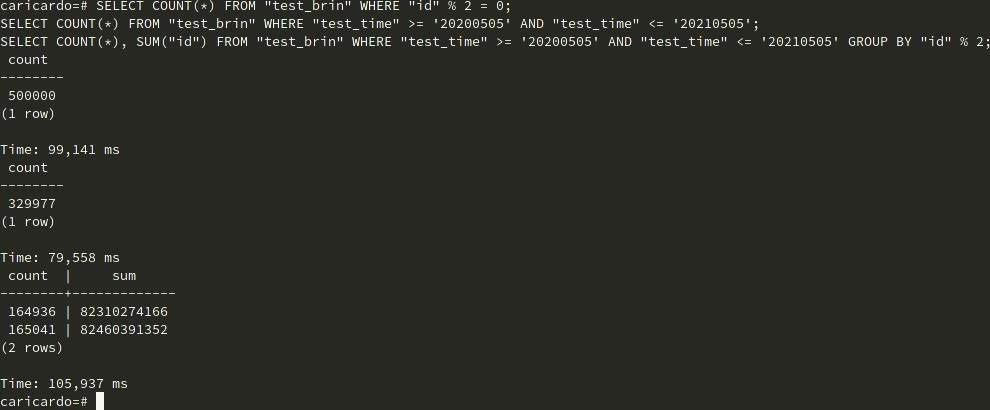
INSERT INTO "test\_brin"("test\_time") SELECT

(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;

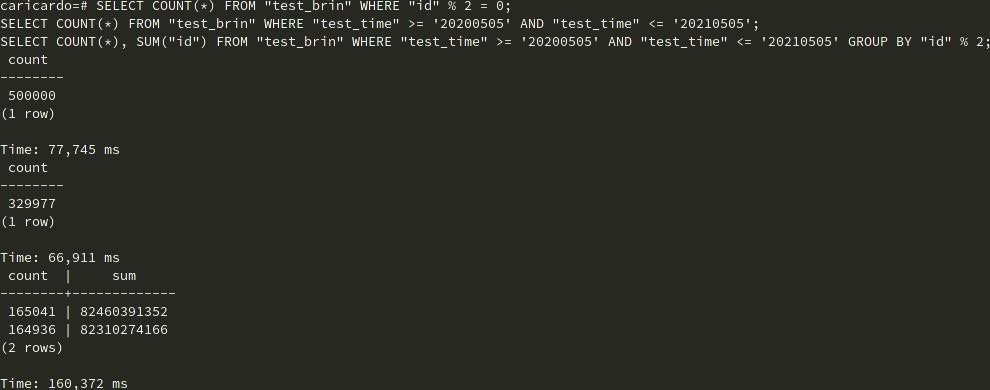
Вибір даних з створеним індексом:



Створюємо індекс:

DROP INDEX IF EXISTS "test\_brin\_test\_time\_index";

CREATE INDEX "test\_brin\_test\_time\_index" ON "test\_brin" USING brin ("test\_time");

Вибір даних з створеним індексом:

Завдання 3 Розробити тригер бази даних PostgreSQ Умова для тригера – before update, delete. Таблиці:

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

"readerID" bigserial PRIMARY KEY, "readerName" varchar(255)

);

DROP TABLE IF EXISTS "readerLog";

CREATE TABLE "readerLog"(

"id" bigserial PRIMARY KEY, "readerLogID" bigint, "readerLogName" varchar(255)

);

Тригер:

CREATE OR REPLACE FUNCTION update\_delete\_func() RETURNS TRIGGER as $$

DECLARE

CURSOR\_LOG CURSOR FOR SELECT \* FROM "readerLog";

row\_Log "readerLog"%ROWTYPE;

begin

IF old."readerID" % 2 = 0 THEN

INSERT INTO "readerLog"("readerLogID", "readerLogName") VALUES

(old."readerID", old."readerName");

UPDATE "readerLog" SET "readerLogName" = trim(BOTH 'x' FROM "readerLogName");

RETURN NEW;

ELSE

RAISE NOTICE 'readerID is odd'; FOR row\_log IN cursor\_log LOOP

UPDATE "readerLog" SET "readerLogName" = 'x' ||

row\_Log."readerLogName" || 'x' WHERE "id" = row\_log."id"; END LOOP;

RETURN NEW; END IF;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER "test\_trigger" BEFORE UPDATE OR DELETE ON "reader" FOR EACH ROW

EXECUTE procedure update\_delete\_func();

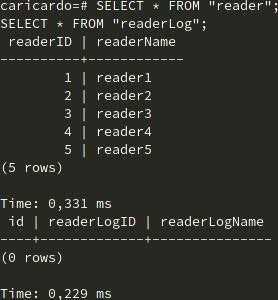
Принцип роботи:

Тригер спрацьовує після видалення з таблиці чи при оновлені у таблиці reader. Якщо значення ідентифікатора запису, який видаляється або оновлюється, парне, то цей запис заноситься у додаткову таблицю readerLog.

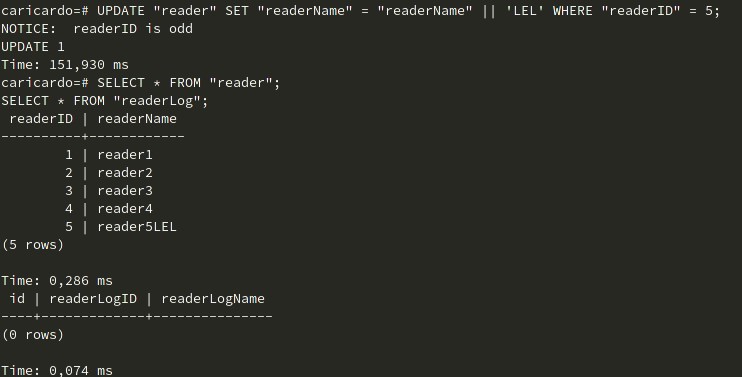
Також, з кожного значення «readerName» видаляються символи «х» на початку і кінці. Якщо значення ідентифікатора непарне, то до кожного значення «readerLogName» у таблиці readerLog додається “х” на початку і кінці.

Занесемо тестові дані до таблиці:

INSERT INTO "reader"("readerName")

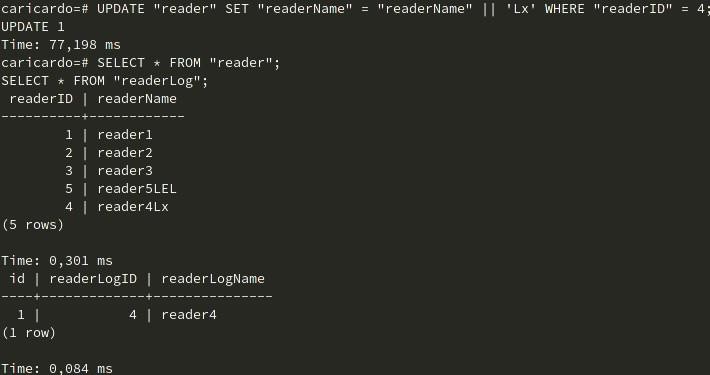
VALUES ('reader1'), ('reader2'), ('reader3'), ('reader4'), ('reader5');

Оновимо дані в одному з рядків:



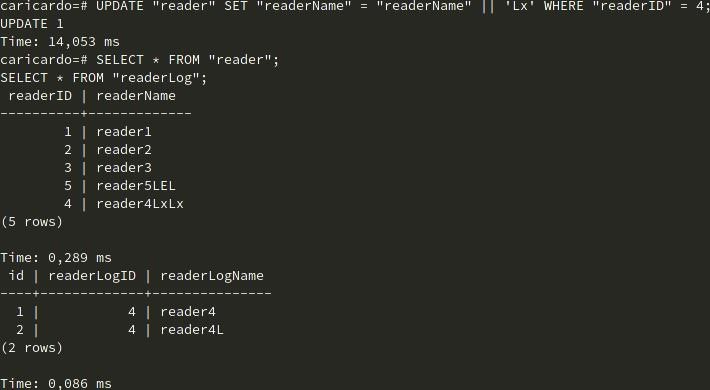
Оскільки id рядку який було оновлено є непарним числом, та оскільки таблиця "readerLog" є пустою, то отримано просте оновлення запису.

Змінимо значення парного рядка:



Як бачимо, перед оновленням рядка його значення буде занесено у таблицю "readerLog".

Виконаємо такий самий запит ще раз:



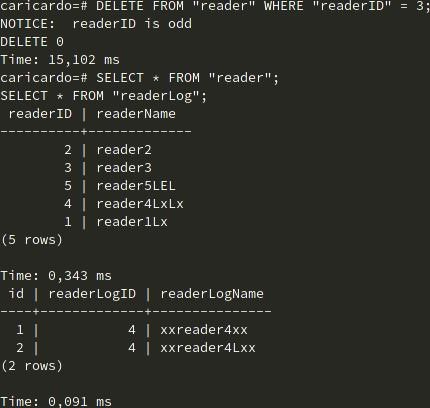
Як бачимо, перед оновленням рядка його значення з видаленими з початку та кінця символами «х» буде занесено у таблицю "readerLog".

Оновимо дані в одному з непарних рядків:



Як бачимо, значення непарних рядків не заносятся до таблиці "readerLog", проте, зміна або видалення непарного рядка призводить до того, що до всіх значень ’’readerLog"."readcrLogNamc" у початок та в кінець

додаються символи «х». Видалення рядку:



Як бачимо, операція видалення також призводить до спрацьовування тригера. Також можна зазначити, що тригер може відміняти операцію

видалення для рядка, в залежності від того яке значення повертається: «old» для видалення, «new» для відміни видалення.

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 впливають 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 транзакцій можливі виникнення таких проблем:

1. Втрачене оновлення

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

1. «Брудне» читання

Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

1. Неповторюване читання

Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

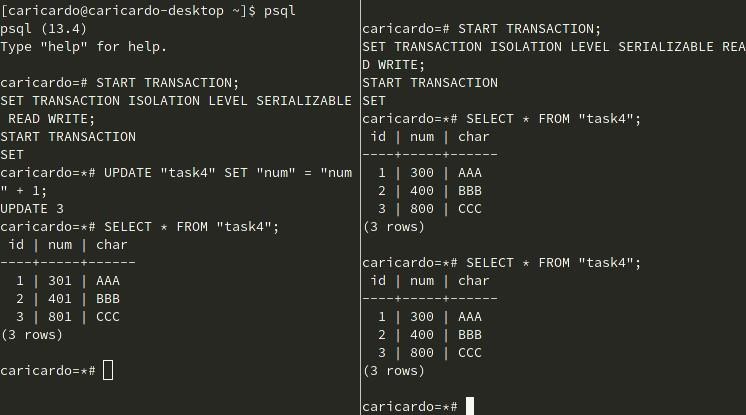
1. Фантомне читання

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

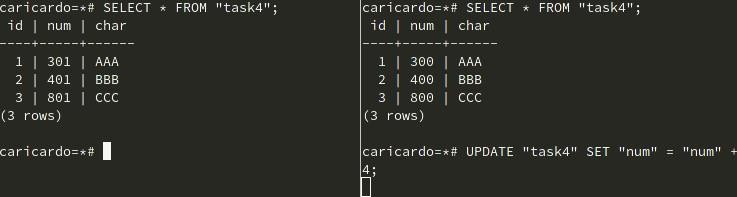
Стандарт SQL-92 визначає наступні рівні ізоляції:

1. Serializable (впорядкованість)

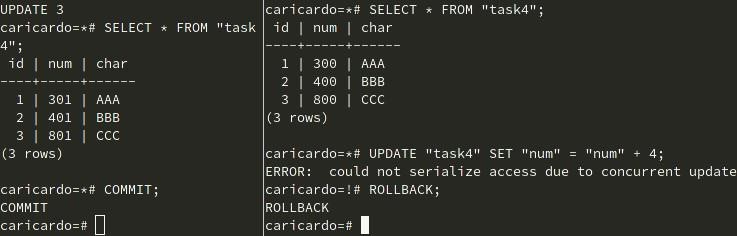
Найбільш високий рівень ізольованості; транзакції повністю ізолюються одна від одної. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими, що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку)



Як бачимо, дані у транзакціях ізольовано.



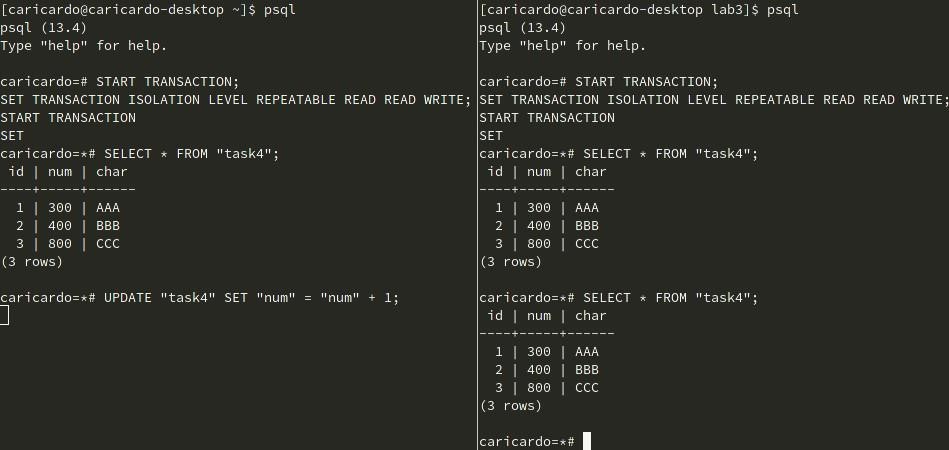
Тепер при оновлені даних в T2 бачимо, що Т2 блокується поки Т1 не зафіксує зміни або не відмінить їх.



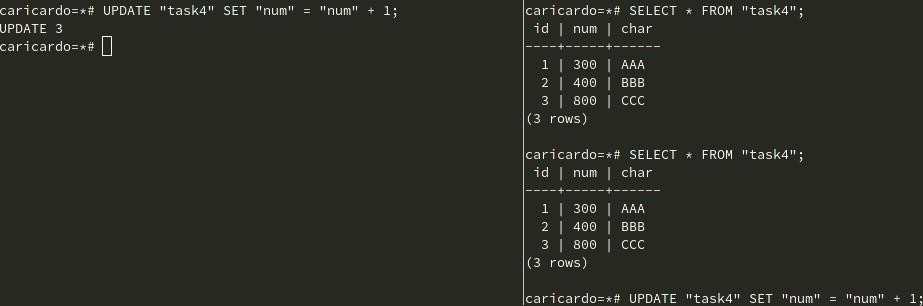
1. Repeatable read (повторюваність читання)

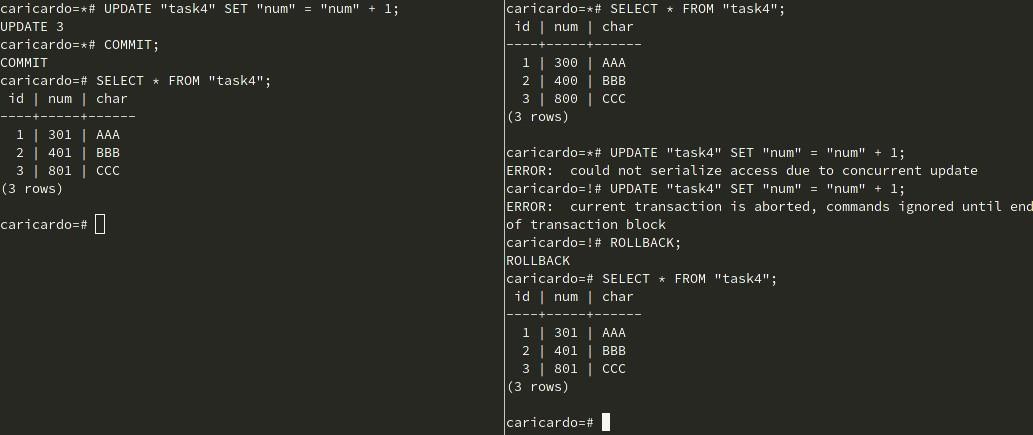
Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші

транзакції не можуть змінити ці дані).



Тепер транзакція Т2 буде чекати поки Т1 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 читання.

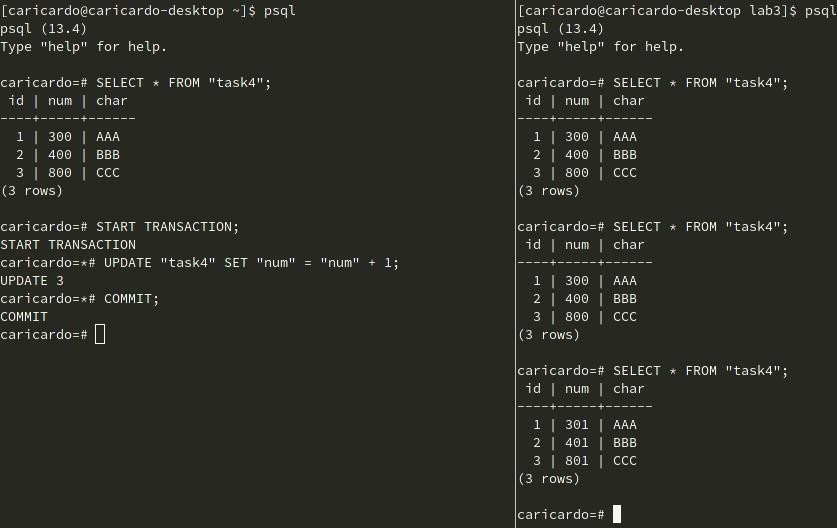
1. Read committed (читання фіксованих даних)

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 може бути

успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема

неповторюваного читання.



1. 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 же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцию. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.