

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”*

Виконав:

студент ІII курсу

групи КВ-01

Гнатенко Я. М.

Перевірив: Павловський В. І.

Київ – 2022

**Варіант 4**

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.

**Логічна модель бази даних**

Нижче (Рисунок 1) наведено логічну модель бази даних:

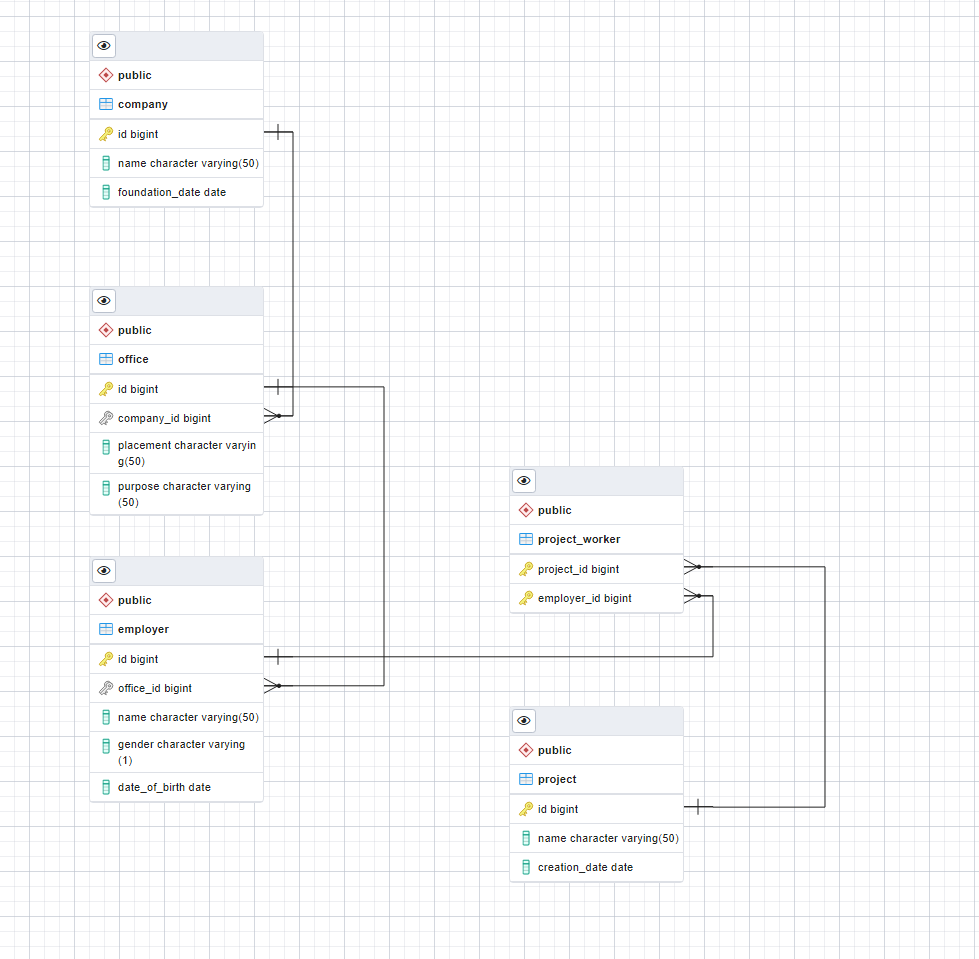


Рисунок 1 – Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Класи ORM**

class company(Base):

    \_\_tablename\_\_ = "company"

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

    name = Column(String)

    foundation\_date = Column(Date)

class office(Base):

    \_\_tablename\_\_ = "office"

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

    company\_id = Column(Integer, ForeignKey("company.cid"))

    placement = Column(String)

    purpose = Column(String)

class employer(Base):

    \_\_tablename\_\_ = "employer"

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

    office\_id = Column(Integer, ForeignKey("office.oid"))

    name = Column(String)

    gender = Column(CHAR)

    date\_of\_birth = Column(Date)

class project(Base):

    \_\_tablename\_\_ = "project"

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

    name = Column(String)

    creation\_date = Column(Date)

class project\_worker(Base):

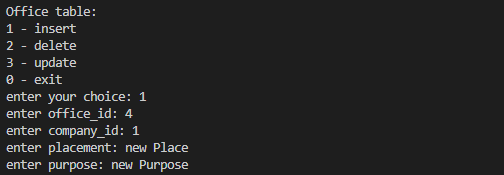
    \_\_tablename\_\_ = "project\_worker"

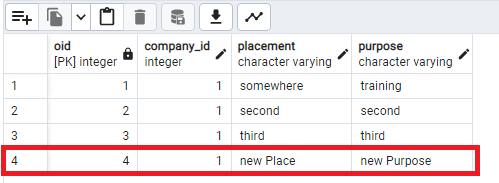
    project\_id = Column(Integer, ForeignKey("project.pid"), primary\_key = True)

    employer\_id = Column(Integer, ForeignKey("employer.eid"), primary\_key = True)

**Результати виконання операцій**

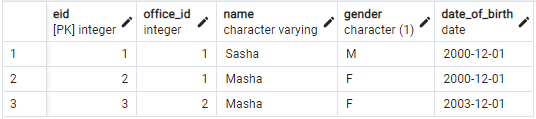
**Додавання нового рядку до таблиці “office”**



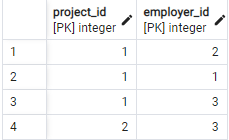


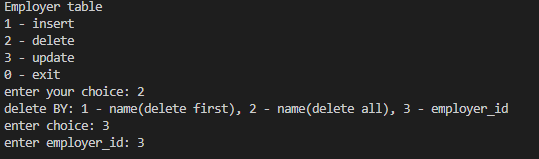
**Видалення рядку з таблиці “employer”**

до видалення

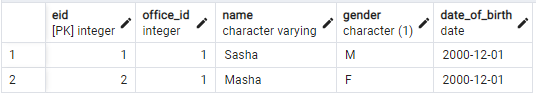


таблиця project\_worker

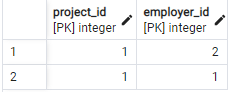




після видалення

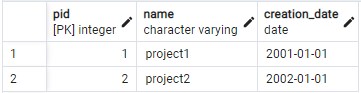


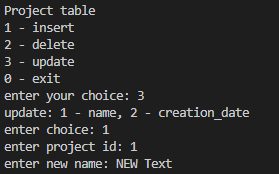
всі рядки пов’язані з видаленим робітником також зникли в таблиці project\_worker



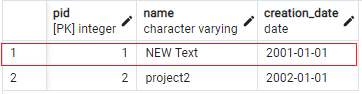
**Оновлення атрибуту “name” в рядку таблиці “project”**

початковий вигляд





після оновлення



**GIN**

Для дослідження індексу була створена таблиця, яка має дві колонки: id типу integer і annotation типу text. Колонка annotation проіндексована як GIN. У таблицю занесено 1000000 записів.

**Створення таблицю та її заповнення:**

CREATE TABLE gin\_table (

id integer,

annotation text COLLATE "C"

);

INSERT INTO gin\_table(id, annotation)

SELECT s.id, md5(**random**()::text)

FROM generate\_series(1, 1000000) AS s(id)

ORDER BY **random**();

**Вибір даних без індексу:**



**Створюємо індекс:**

CREATE INDEX gin\_idx

ON gin\_table

USING gin(annotation gin\_trgm\_ops);

**Вибір даних з створеним індексом:**



**Brin**

Для дослідження індексу була створена таблиця, яка має дві колонки: id типу bigserial і annotation типу text. Колонка id проіндексована як Brin. У таблицю занесено 1000000 записів.

**Створення таблицю та її заповнення:**

CREATE TABLE hash\_table(

id integer,

annotation text COLLATE "C"

);

INSERT INTO hash\_table(id, annotation)

SELECT s.id, md5(**random**()::text)

FROM generate\_series(1, 1000000) AS s(id)

ORDER BY **random**();

**Вибір даних без індексу:**

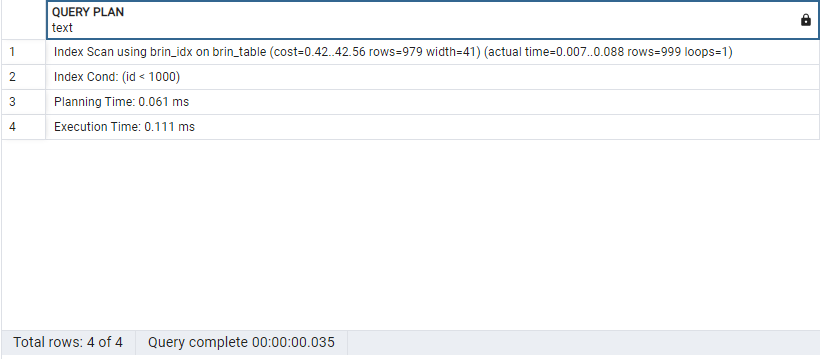


**Сворюємо індекс:**

CREATE INDEX brin\_idx

ON brin\_table USING BRIN(id);

**Вибір даних з створеним індексом:**

****

Основна сфера застосування методу GIN – прискорення повнотекстового пошуку. Якщо дані змінюються не часто, шукати треба швидко – GIN буде мати кращі результати, ніж GiST.

Brin ефективно використовується для пошуку в природно-впорядкованих таблиць. Дозволяє пропускати великі розділи таблиць, які не можуть містити відповідні значення.

**Тригер бази даних PostgreSQL. Умова для тригера – after insert, delete**

Створимо допоміжні таблиці

CREATE TABLE employee

(

employee\_id INT NOT NULL primary key,

FirstName varchar(20),

LastName varchar(20),

ins\_time timestamp

);

CREATE TABLE employee\_audit

(

employee\_id INT NOT NULL,

FirstName varchar(20),

LastName varchar(20),

operation\_time timestamp

);

**Створимо функцію (insert)**

CREATE OR REPLACE FUNCTION insert\_func()

RETURNS trigger AS

$$

BEGIN

IF NEW.salary < 0 THEN

RAISE EXCEPTION '% cannot have a negative salary', NEW.name;

END IF;

INSERT INTO employee\_audit (operation, employee\_id, name, salary, operation\_time)

VALUES('INSERT', NEW.employee\_id,NEW.name, NEW.salary, now());

RETURN NEW;

END;

$$

LANGUAGE plpgsql;

**Створимо функцію (delete)**

CREATE OR REPLACE FUNCTION delete\_func()

RETURNS trigger AS

$$

BEGIN

INSERT INTO employee\_audit (operation, employee\_id, name, salary, operation\_time)

VALUES('DELETE', OLD.employee\_id, OLD.name, OLD.salary, now());

RETURN NEW;

END;

$$

LANGUAGE plpgsql;

**Тригер(after insert)**

CREATE OR REPLACE TRIGGER InsertTrigger

after INSERT ON employee

FOR EACH ROW EXECUTE FUNCTION insert\_func();

**Тригер(after delete)**

CREATE OR REPLACE TRIGGER DeleteTrigger

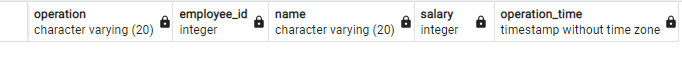
after DELETE ON employee

FOR EACH ROW EXECUTE FUNCTION delete\_func();

**Принцип роботи тригеру**

Тригер після вставлення або видалення в таблиці «employee». При вставленні або видаленні данних з таблиці «employee» у таблицю «employee\_audit» буде заноситись рядок про тип операції, часова мітка виконання та дані про робітника.

Таблиця “employee\_audit” до виконання операцій (insert, delete)

 **Виконаємо наступні операції:**

INSERT INTO employee (employee\_id, name, salary, ins\_time)

VALUES (1, 'Sasha', 1000, now());

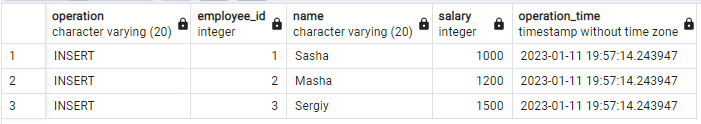
INSERT INTO employee (employee\_id, name, salary, ins\_time)

VALUES (2, 'Masha', 1200, now());

INSERT INTO employee (employee\_id, name, salary, ins\_time)

VALUES (3, 'Sergiy', 1500, now());

Таблиця “employee\_audit” після виконання операцій (insert)

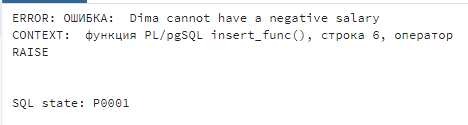


**Спробуємо виконати некоректну операцію:**

INSERT INTO employee (employee\_id, name, salary, ins\_time)

VALUES (4, 'Dima', -1200, now());

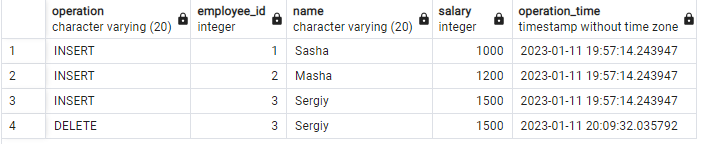
**отримуємо помилку, а таблиці ніяк не змінились**

****

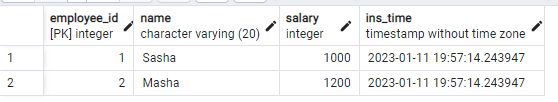
**Виконаємо наступну операції:**

DELETE from employee where employee\_id = 3;

Таблиця “employee\_audit” до виконання операцій (delete)



Таблиця “employee” до виконання операцій (delete)



**Рівні ізоляції транзакцій у PostgreSQL**

Створимо та заповнимо таблицю:

CREATE TABLE transactions (

id bigserial,

text\_ varchar(10)

);

INSERT INTO transactions(text\_)

VALUES ('text1'), ('text2'), ('text3');

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. Ізольованість транзакції показує те, наскільки сильно впливають одне на одну паралельно виконувані транзакції.

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

Найвищу швидкість виконання та найнижчу узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та найвищу узгодженість — serializable.

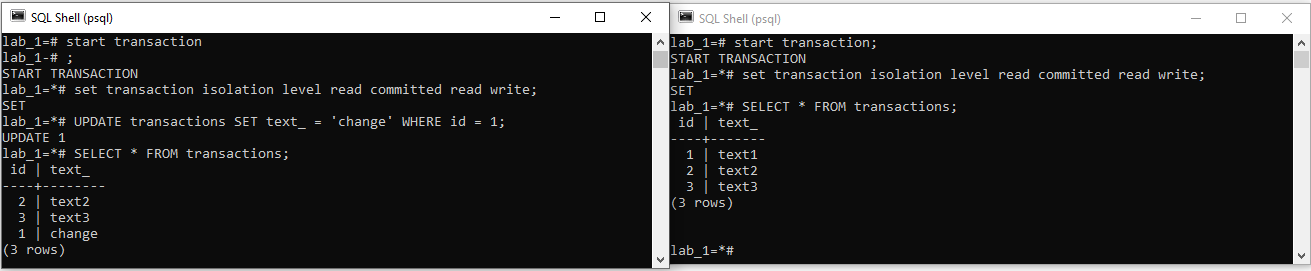
При паралельному виконанні транзакцій можливі виникнення таких проблем:

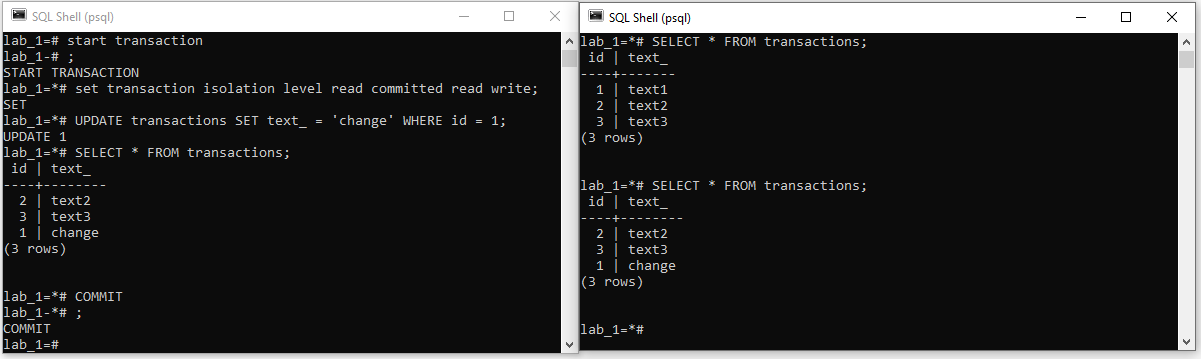
1. Втрачене оновлення - ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається
2. «Брудне» читання- читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться)
3. Фантомне читання - ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків
4. Неповторюване читання - ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими

**Стандарт SQL-92**

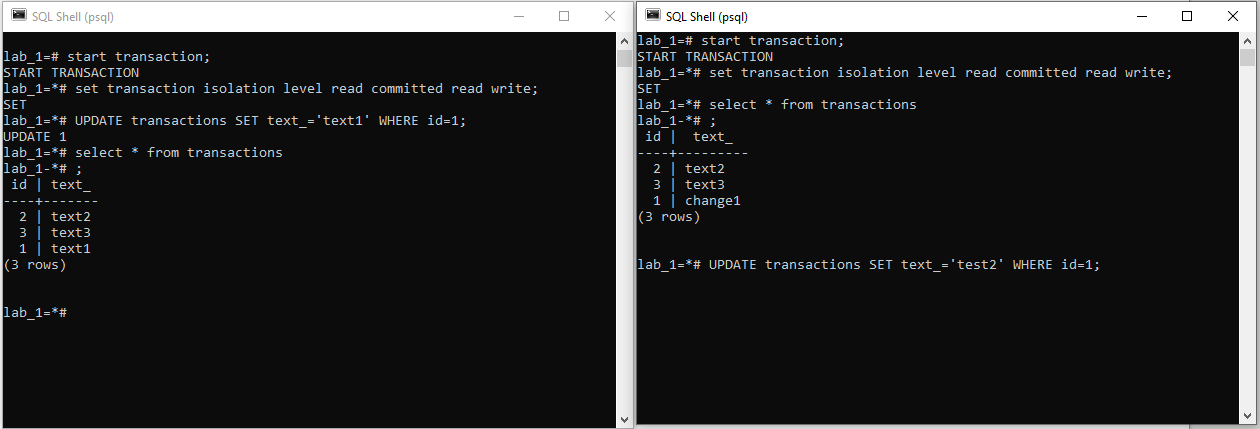
1. **Read committed (читання фіксованих даних)**

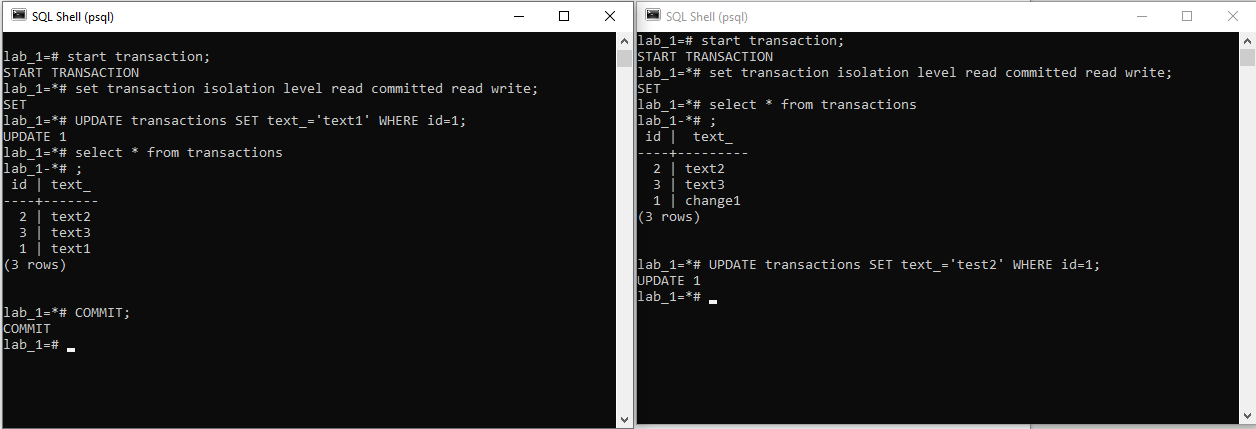
Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку,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

 У лівій консолі було змінено рядок таблиці, після чого було виведено таблицю в консолі справа, як бачимо, таблиця залишилася незмінною



Після виконання команди “COMMIT” у лівій консолі, зміни почали відображатись і для правої консолі

 У лівій консолі було розпочато, проте не закінчено транзакцію, після чого у правій консолі виконати іншу дію і отримали очікування закінчення дії лівої консолі.

 Після закінчення транзакції в лівій консолі, операція в правій завершилась успішно.

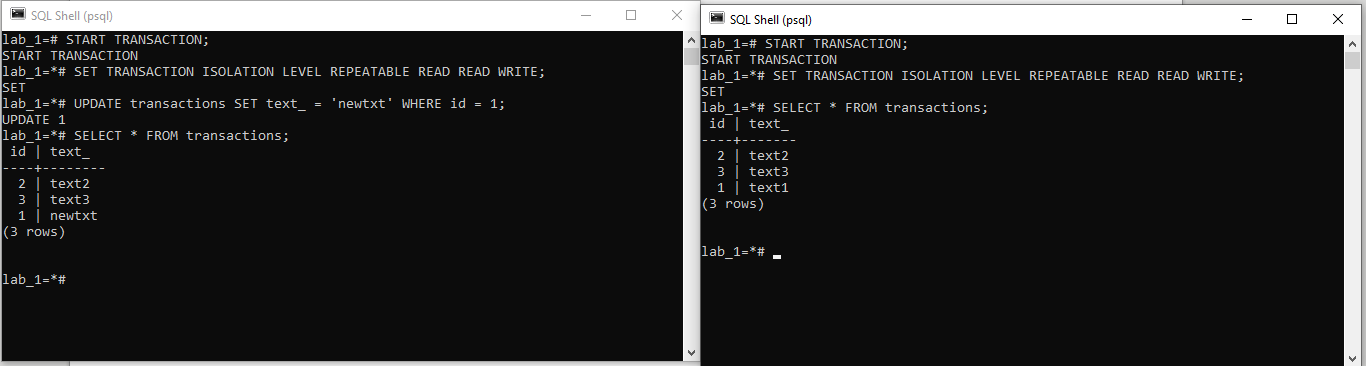
1. **Read uncommitted (читання незафіксованих даних)**

Найнижчий рівень ізоляції. Він гарантує тільки відсутність втрачених оновлень. На цьому рівні транзакції не є ізольованими один від одної. Якщо декілька транзакцій змінювали один той самий рядок то в кінцевому результаті рядок матиме значення останньої успішно виконаної транзакції.

PostgreSQL не підтримує даний вид ізоляції

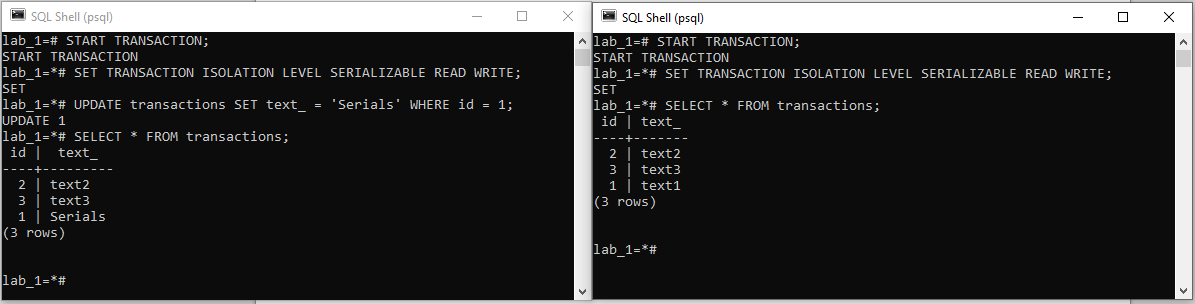
1. **Repeatable read (повторюваність читання)**

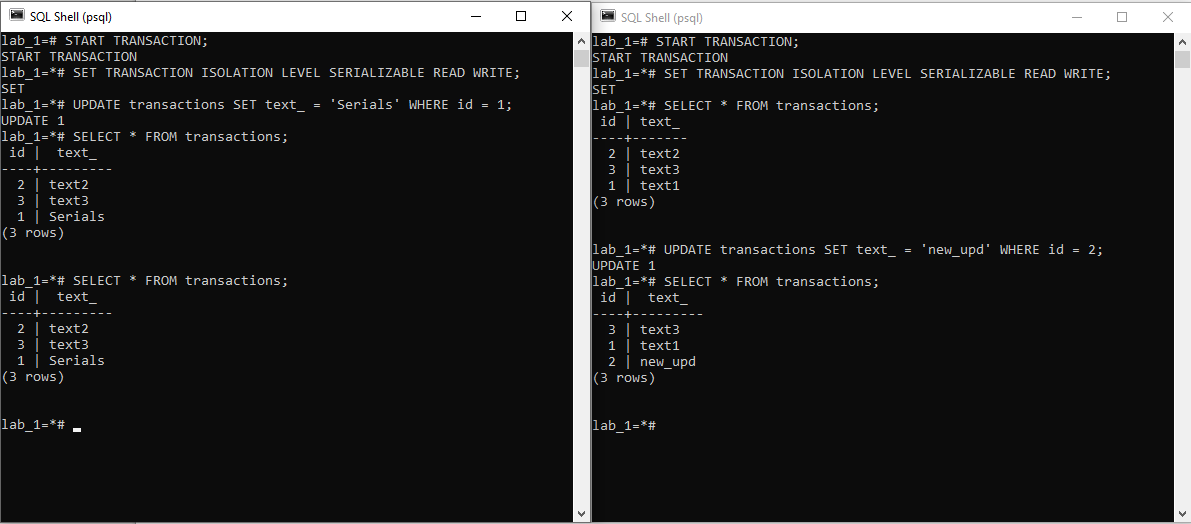
Рівень, при якому читання одного і того ж рядку/рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані)

Після оновлення таблиці у лівій консолі, у правій вона не змінилась

1. **Serializable (впорядкованість)**

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

 Після зміни рядку таблиці у лівій консолі, таблиця в правій консолі залишилась без змін



Потім було змінено рядок таблиці в правій консолі, після даної операції таблиця в лівій консолі не змінилась, як бачимо таблиці є незалежними одне від одної