

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

Проводов Д. В.

Перевірив: Павловський В. І.

Київ – 2022

**Варіант 11**

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:

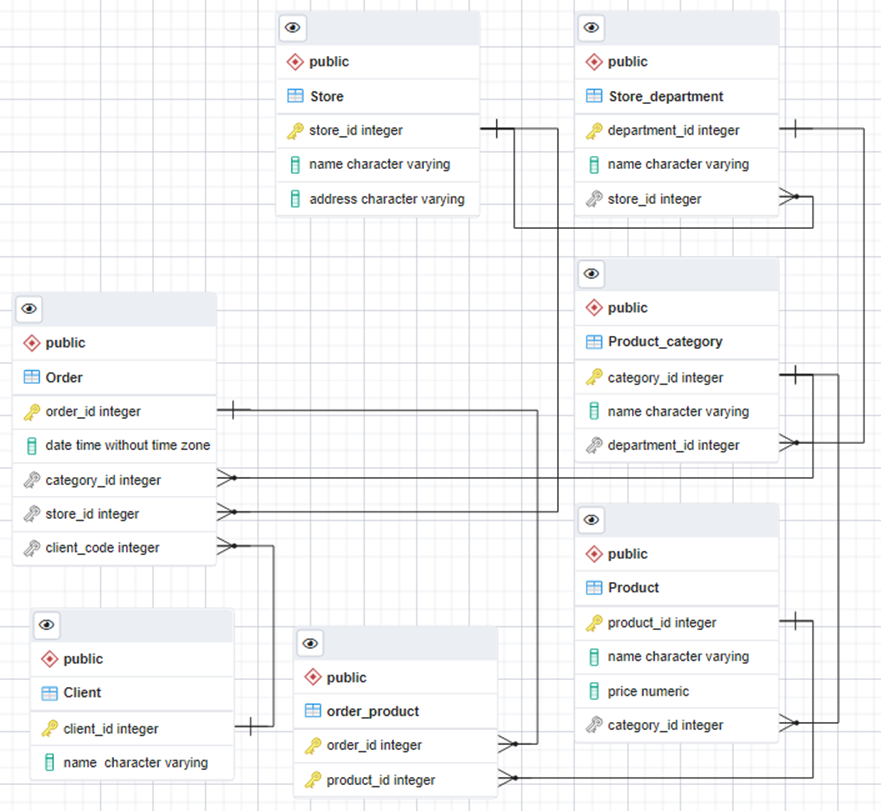


Рисунок 1 – Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Класи ORM**

class Client(Base):  
 \_\_tablename\_\_ = 'Client'  
 client\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
 super(Client, self).\_\_init\_\_()  
  
  
class Order(Base):  
 \_\_tablename\_\_ = 'Order'  
 order\_id = Column(Integer, primary\_key=True)  
 date = Column(sqlalchemy.Time)  
 category\_id = Column(Integer, ForeignKey('Product\_category.category\_id'))  
 client\_id = Column(Integer, ForeignKey('Client.client\_id'))  
 store\_id = Column(Integer, ForeignKey('Store.store\_id'))  
  
 category = relationship("ProductCategory", foreign\_keys=[category\_id])  
 client = relationship("Client", foreign\_keys=[client\_id])  
 store = relationship("Store", foreign\_keys=[store\_id])  
  
 def \_\_init\_\_(self, date, category\_id, client\_id, store\_id):  
 self.date = date  
 self.category\_id = category\_id  
 self.client\_id = client\_id  
 self.store\_id = store\_id  
 super(Order, self).\_\_init\_\_()  
  
  
class Product(Base):  
 \_\_tablename\_\_ = 'Product'  
 product\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
 price = Column(Integer)  
 category\_id = Column(Integer, ForeignKey('Product\_category.category\_id'))  
  
 category = relationship("ProductCategory", foreign\_keys=[category\_id])  
  
 def \_\_init\_\_(self, name, price, category\_id):  
 self.name = name  
 self.price = price  
 self.category\_id = category\_id  
 super(Product, self).\_\_init\_\_()  
  
  
class ProductCategory(Base):  
 \_\_tablename\_\_ = 'Product\_category'  
 category\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
 department\_id = Column(Integer, ForeignKey('Store\_department.department\_id'))  
  
 department = relationship("StoreDepartment", foreign\_keys=[department\_id])  
  
 def \_\_init\_\_(self, name, department\_id):  
 self.name = name  
 self.department\_id = department\_id  
 super(ProductCategory, self).\_\_init\_\_()  
  
  
class Store(Base):  
 \_\_tablename\_\_ = 'Store'  
 store\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
 address = Column(String)  
  
 def \_\_init\_\_(self, name, address):  
 self.name = name  
 self.address = address  
 super(Store, self).\_\_init\_\_()  
  
  
class StoreDepartment(Base):  
 \_\_tablename\_\_ = 'Store\_department'  
 department\_id = Column(Integer, primary\_key=True)  
 name = Column(String)  
 store\_id = Column(Integer, ForeignKey('Store.store\_id'))  
  
 store = relationship("Store", foreign\_keys=[store\_id])  
  
 def \_\_init\_\_(self, name, store\_id):  
 self.name = name  
 self.store\_id = store\_id  
 super(StoreDepartment, self).\_\_init\_\_()  
  
  
class OrderProduct(Base):  
 \_\_tablename\_\_ = 'order\_product'  
 order\_id = Column(Integer, ForeignKey('Order.order\_id'), primary\_key=True)  
 product\_id = Column(Integer, ForeignKey('Product.product\_id'), primary\_key=True)  
  
 order = relationship("Order", foreign\_keys=[order\_id])  
 product = relationship("Product", foreign\_keys=[product\_id])  
  
 def \_\_init\_\_(self, order\_id, product\_id):  
 self.order\_id = order\_id  
 self.product\_id = product\_id  
 super(OrderProduct, self).\_\_init\_\_()

**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 таблиці та її заповнення:**

CREATE TABLE gin\_table (

id integer,

annotation text COLLATE "C"

);

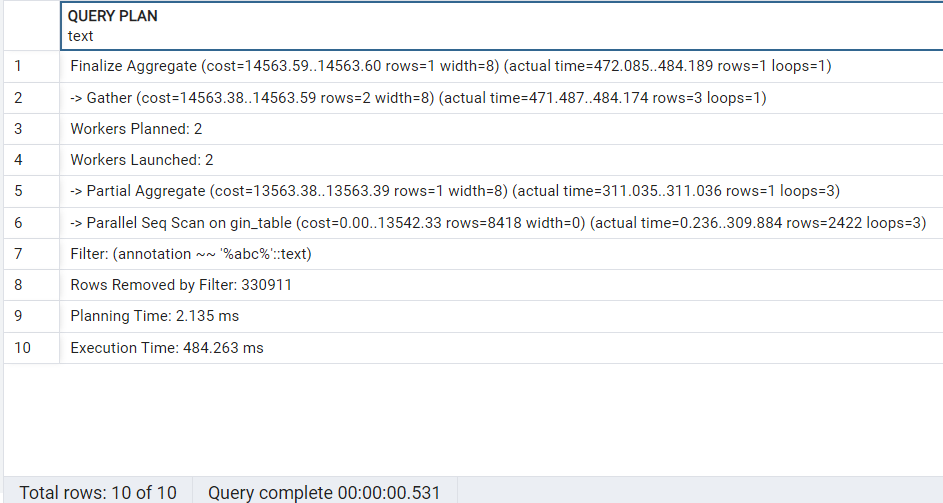
INSERT INTO gin\_table(id, annotation)

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

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

ORDER BY **random**();

**Вибір даних без індексу:**



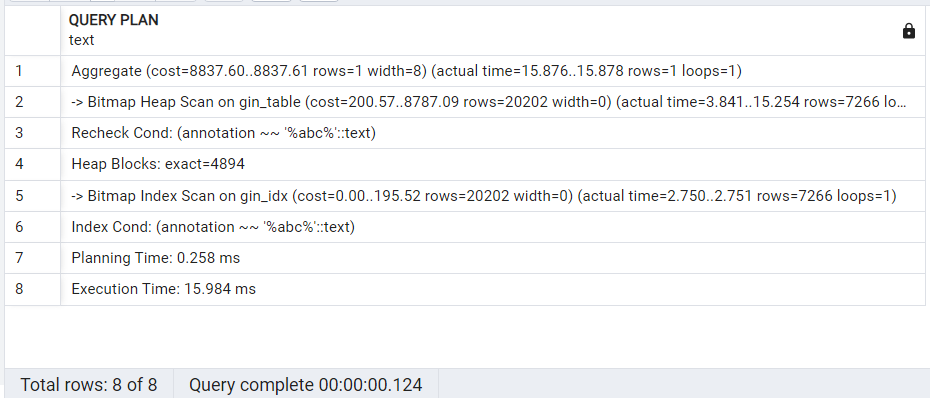
**Сворюємо індекс:**

CREATE INDEX gin\_idx

ON gin\_table

USING gin(annotation gin\_trgm\_ops);

**Вибір даних з створеним індексом:**



**Hash**

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

**Створення таблиці та її заповнення:**

CREATE TABLE hash\_table(

id integer,

annotation text COLLATE "C"

);

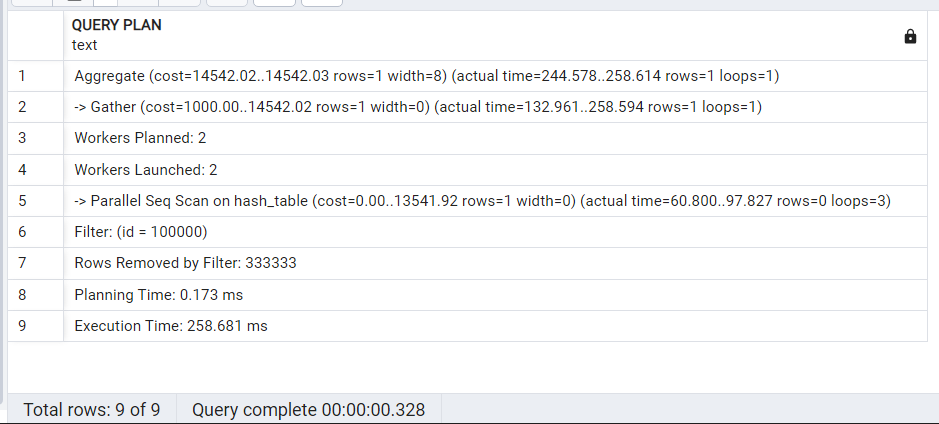
INSERT INTO hash\_table(id, annotation)

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

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

ORDER BY **random**();

**Вибір даних без індексу:**



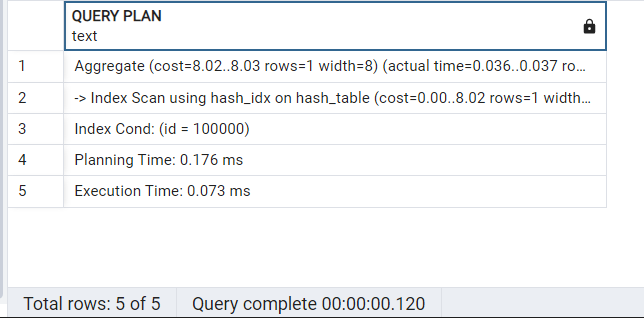
**Сворюємо індекс:**

CREATE INDEX hash\_idx

ON hash\_table

USING hash("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.

Hash використовують тільки з простими умовами рівності. Планувальник запитів може застосувати хеш-індекс, тільки якщо індексований стовпець бере участь у порівнянні з оператором =.

**Тригер бази даних PostgreSQL. Умова для тригера – before update, delete**

**Таблиці:**

CREATE TABLE employee (

id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),

username varchar NOT NULL,

salary integer NOT NULL,

last\_update timestamp

);

INSERT INTO employee (username, salary, last\_update)

VALUES ('Billy', 1000, current\_timestamp);

CREATE TABLE employee\_audit (

operation varchar(10) NOT NULL,

stamp timestamp NOT NULL,

employee\_id integer NOT NULL,

username varchar NOT NULL,

salary integer NOT NULL

);

**Функція:**

CREATE OR REPLACE FUNCTION employee\_stamp() RETURNS trigger AS $employee\_stamp$

BEGIN

IF (TG\_OP = 'DELETE') THEN

INSERT INTO employee\_audit SELECT 'DELETE', now(), OLD.id, OLD.username, OLD.salary;

ELSIF (TG\_OP = 'UPDATE') THEN

IF NEW.salary < 0 THEN

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

END IF;

INSERT INTO employee\_audit SELECT 'UPDATE', now(), NEW.id, NEW.username, NEW.salary;

END IF;

NEW.last\_update := current\_timestamp;

RETURN NEW;

END;

$employee\_stamp$ LANGUAGE plpgsql;

**Тригер:**

CREATE TRIGGER employee\_stamp BEFORE UPDATE OR DELETE ON employee

FOR EACH ROW EXECUTE FUNCTION employee\_stamp();

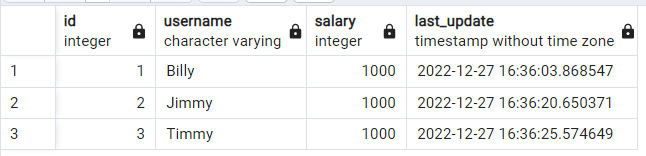
**Принцип роботи тригеру**

Тригер спрацьовує перед редагуванням або видаленням в таблиці «employee». При редагуванні або видаленні данних з таблиці «employee» у таблицю «employee\_audit» буде заноситись рядок про тип операції, часова мітка виконання та дані про робітника. Якщо обрана операція була UPDATE, тоді ще проводиться додаткова перевірка на ввід некоректних даних (зарплатня не може бути менше 0).

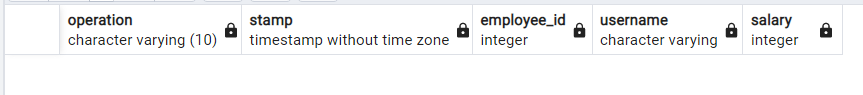
***Приклад***

**Таблиці до змін:**

employee



employee\_audit



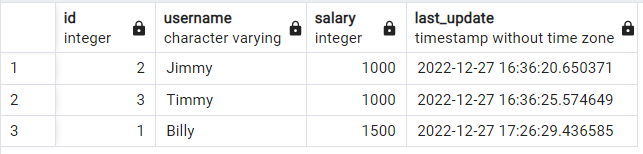
**Виконаємо оновлення одного рядка:**

UPDATE employee

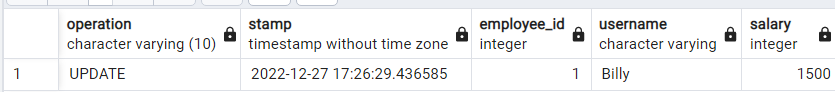
SET salary = 1500

WHERE id = 1;

employee



employee\_audit



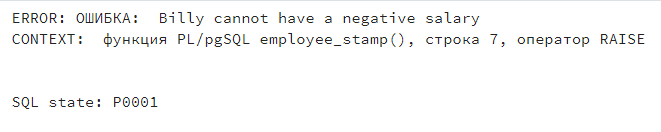
**Спробуємо задати некоректні дані:**

UPDATE employee

SET salary = -1500

WHERE id = 1;

Отримуємо помилку, а самі таблиці ніяк не змінились

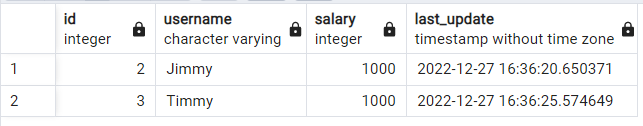


**Виконаємо видалення рядка:**

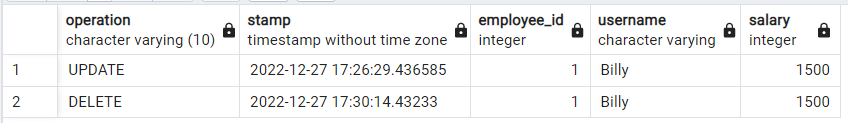
DELETE FROM employee

WHERE id = 1;

employee



employee\_audit



**Рівні ізоляції транзакцій у PostgreSQL**

**Створимо та заповнимо таблицю:**

CREATE TABLE transactions (

id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),

string varchar(10)

);

INSERT INTO transactions(string)

VALUES ('string1'), ('string2'), ('string3');

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім невиконуються. 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.

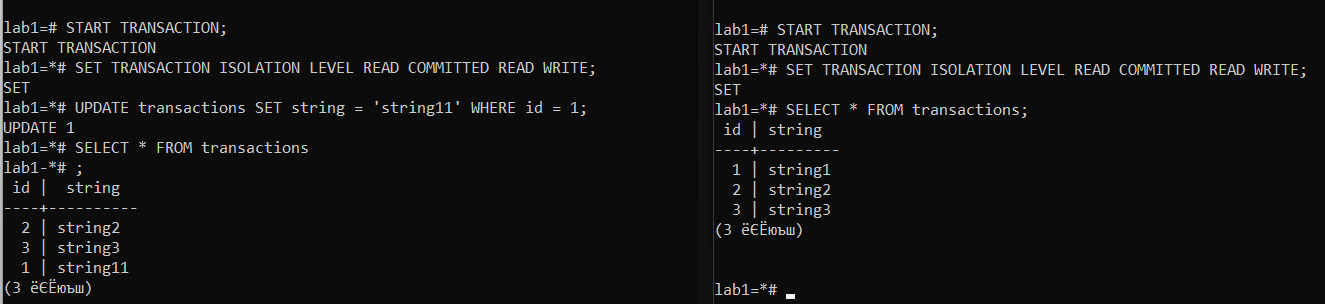
При паралельному виконанні транзакцій можливі виникненя таких проблем:

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 виявляються зміненими.
4. Фантомне читання - ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

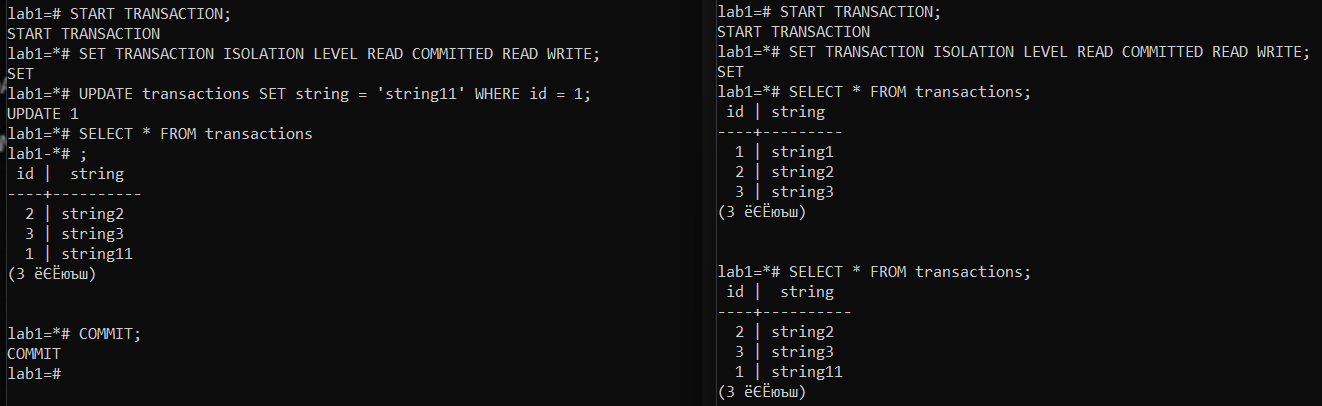
**Стандарт SQL-92**

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

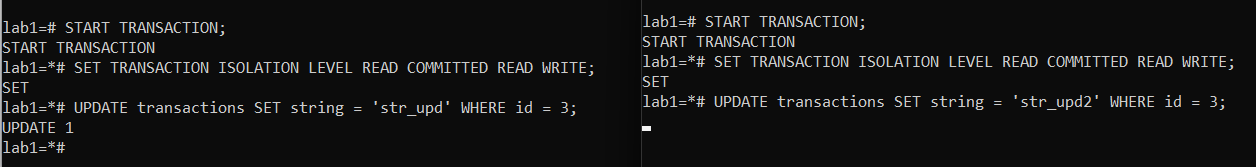
Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані.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



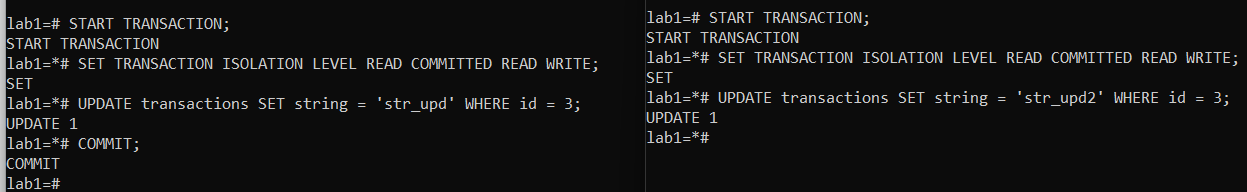
У лівій консолі було змінено один рядок, після чого, у правій консолі було виведено таблицю, як бачимо, таблиця залишилася незмінною.



Після того, як у лівій консолі ми зробили COMMIT, оновлені дані почали відображатися і у правій консолі.



У лівій консолі було розпочато і не завершено транзакцію, після чого у лівій консолі спробували виконати іншу дію і отримали очікування завершення лівої транзакції.



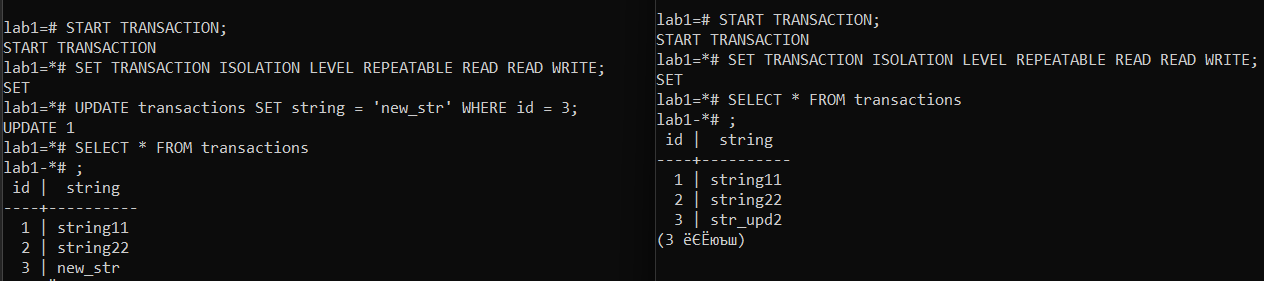
Після завершення лівої транзакції, операція правої завершилась успішно.

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 транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.

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

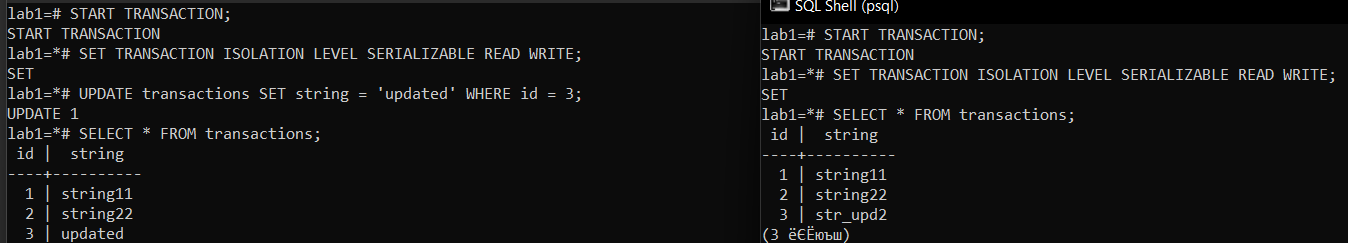
Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані).



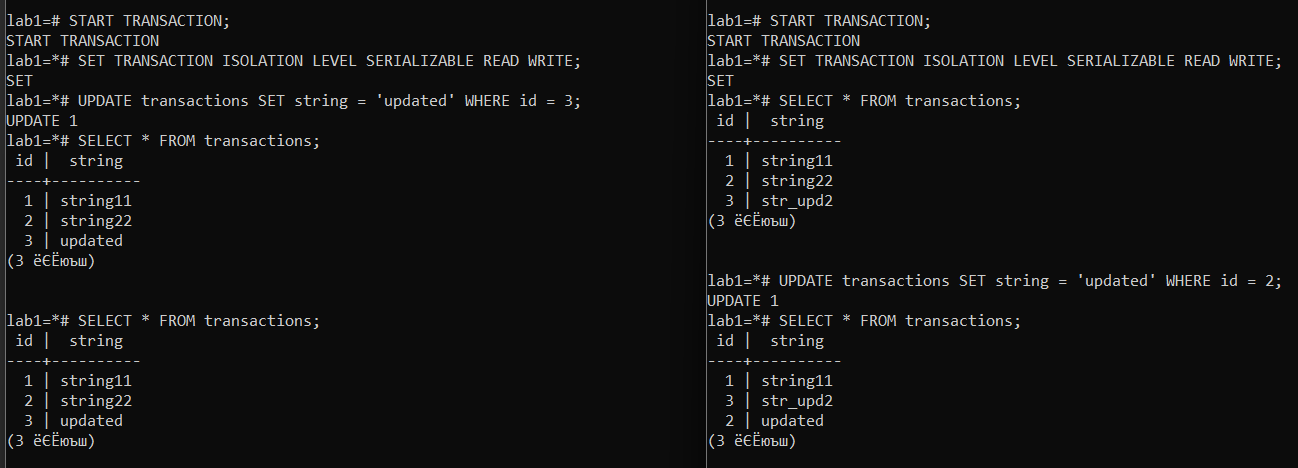
Після того, як у лівій консолі було оновлено таблицю, то у правій вона не змінилась.

1. **Serializable (впорядкованість)**

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



У лівій консолі було змінено один рядок, після чого, у правій консолі було виведено таблицю, як бачимо, таблиця залишилася незмінною.



Після цього, було змінено рядок у правій транзакції і виведено обидві таблиці. Як бачимо, таблиці є незалежними одна від одної.