**Лабораторна робота № 3**

«**Засоби оптимізації роботи СУБД PostgreSQL**»

***з дисципліни***

**“БАЗИ ДАНИХ ТА ЗАСОБИ УПРАВЛІННЯ”**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Група: КВ-01**

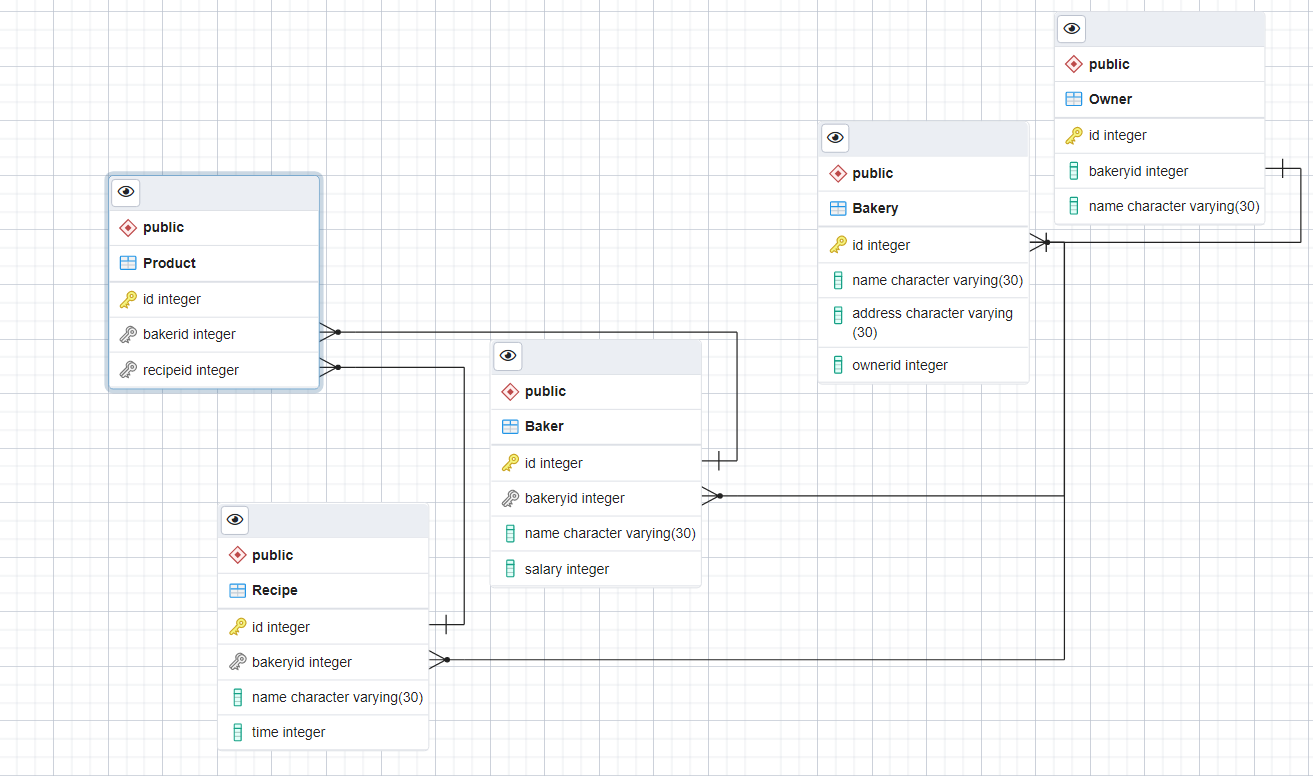
**Виконала: Бондарчук Марія**

**Варіант 2**

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 «Пекарня»**



**Рисунок 1 – Логічна модель бази даних**

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Класи ORM**

'use strict';  
const {  
 Model  
} = ***require***('sequelize');  
***module***.exports = (sequelize, DataTypes) => {  
 class Baker extends Model {  
  
 static *associate*(models) {  
 Baker.*belongsTo*(models.Bakery);  
 }  
 }  
  
 Baker.*init*({  
 bakery\_id: DataTypes.INTEGER,  
 name: DataTypes.STRING,  
 salary: DataTypes.INTEGER  
 }, {  
 sequelize,  
 underscored: true,  
 modelName: 'Baker',  
 });  
 return Baker;  
};

'use strict';  
const {  
 Model  
} = ***require***('sequelize');  
***module***.exports = (sequelize, DataTypes) => {  
 class Bakery extends Model {  
  
 static *associate* (models) {  
 Bakery.*belongsTo*(models.Owner);  
 Bakery.*hasMany*(models.Baker);  
 Bakery.*hasMany*(models.Product);  
 Bakery.*hasMany*(models.Recipe);  
 }  
 }  
  
 Bakery.*init*({  
 owner\_id: DataTypes.INTEGER,  
 name: DataTypes.STRING,  
 address: DataTypes.STRING  
 }, {  
 sequelize,  
 underscored: true,  
 modelName: 'Bakery'  
 });  
 return Bakery;  
};

'use strict';  
const {  
 Model  
} = ***require***('sequelize');  
***module***.exports = (sequelize, DataTypes) => {  
 class Owner extends Model {  
  
 static *associate* (models) {  
 Owner.*hasMany*(models.Bakery);  
 }  
 }  
  
 Owner.*init*({  
 first\_name: DataTypes.STRING,  
 last\_name: DataTypes.STRING  
 }, {  
 sequelize,  
 underscored: true,  
 modelName: 'Owner'  
 });  
 return Owner;  
};

'use strict';  
const {  
 Model  
} = ***require***('sequelize');  
***module***.exports = (sequelize, DataTypes) => {  
 class Product extends Model {  
 static *associate*(models) {  
 Product.*belongsTo*(models.Bakery);  
 Product.*belongsTo*(models.Recipe);  
 }  
 }  
 Product.*init*({  
 bakery\_id: DataTypes.INTEGER,  
 recipe\_id: DataTypes.INTEGER  
 }, {  
 sequelize,  
 underscored: true,  
 modelName: 'Product',  
 });  
 return Product;  
};

'use strict';  
const {  
 Model  
} = ***require***('sequelize');  
***module***.exports = (sequelize, DataTypes) => {  
 class Recipe extends Model {  
  
 static *associate*(models) {  
 Recipe.*belongsTo*(models.Bakery);  
 Recipe.*hasMany*(models.Product);  
 }  
 }  
 Recipe.*init*({  
 bakery\_id: DataTypes.INTEGER,  
 name: DataTypes.STRING,  
 ingredients: DataTypes.STRING  
 }, {  
 sequelize,  
 underscored: true,  
 modelName: 'Recipe',  
 });  
 return Recipe;  
};

**Hash**

**Створення таблиці та її заповнення:**

CREATE TABLE test\_table (

id integer,

annotation text COLLATE "C"

);

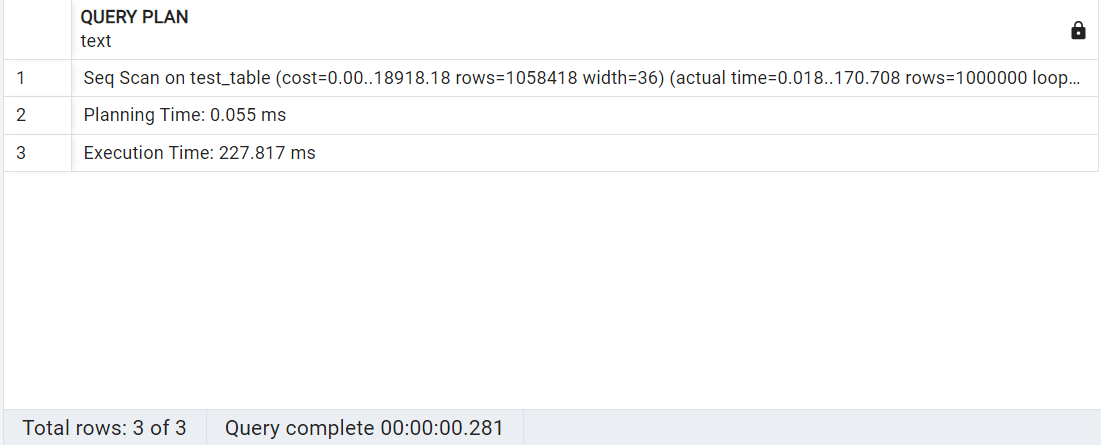
INSERT INTO test\_table(id, annotation)

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

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

ORDER BY **random**();

**Вибір даних без індексу:**



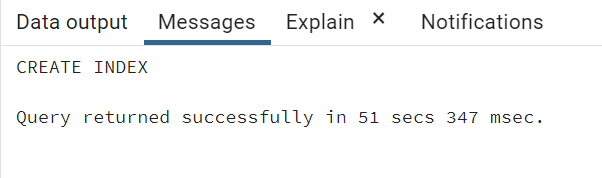
**Сворюємо індекс:**

Спочатку за такою командою:

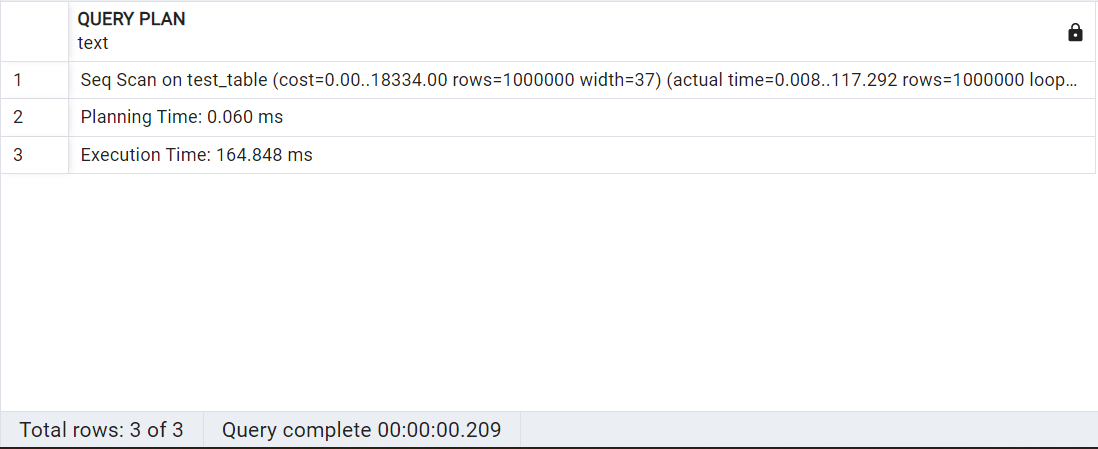
CREATE EXTENSION pg\_trgm;

А потім:

CREATE INDEX h\_idx ON test\_table USING HASH (annotation);



**Вибір даних з створеним індексом:**



|  |  |
| --- | --- |
| **До індексації** | **Після індексації** |
| 227 ms | 164 ms |
| 240 ms | 158 ms |
| 184 ms | 152 ms |
| 243 ms | 175 ms |
| 263 ms | 146 ms |
| 232 ms | 139 ms |
| 213 ms | 164 ms |
| 195 ms | 154 ms |
| 251 ms | 147 ms |

|  |  |
| --- | --- |
| **227 ms** | **155 ms** |

В даному випадку індексація не значно пришвидшила роботу запиту, проте це можна пояснити незначним об’ємом даних у таблиці, особливостями роботи операційної системи. Тому для маленьких баз даних немає необхідності у використанні індексів. Проте індексація повинна пришвидшити пошук у таблицях з великим об’ємом даних, на великих проектах.

# BRIN

**Створення таблиці та її заповнення:**

CREATE TABLE temperature\_log(

log\_id serial,

sensor\_id int,

log\_timestamp timestamp without time zone,

temperature int

);

INSERT INTO temperature\_log(sensor\_id, log\_timestamp, temperature)

VALUES (1,

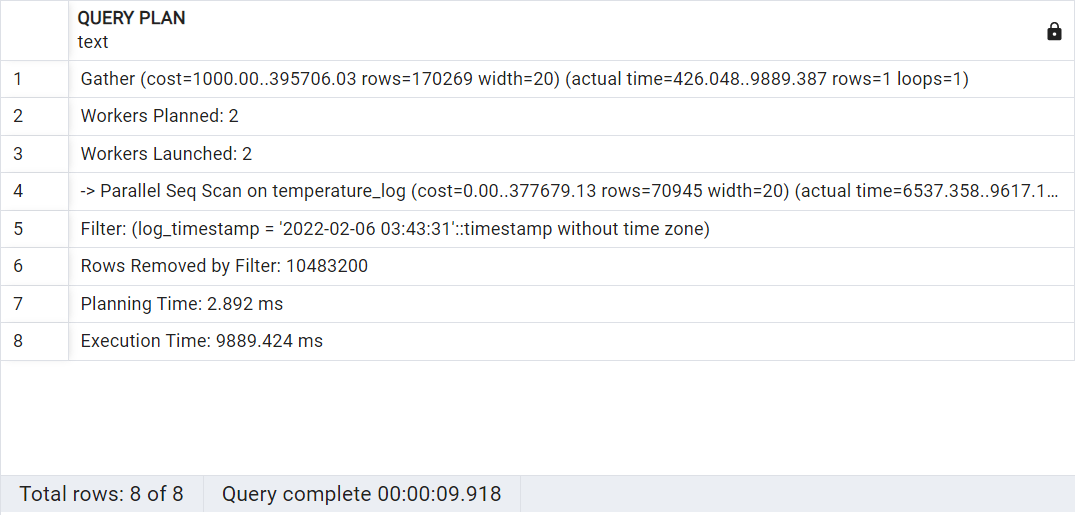
generate\_series('2022-01-01'::timestamp,

'2022-12-31'::timestamp,

'1 second'),

**round**(**random**()\*100)::int);

**Вибір даних без індексу:**

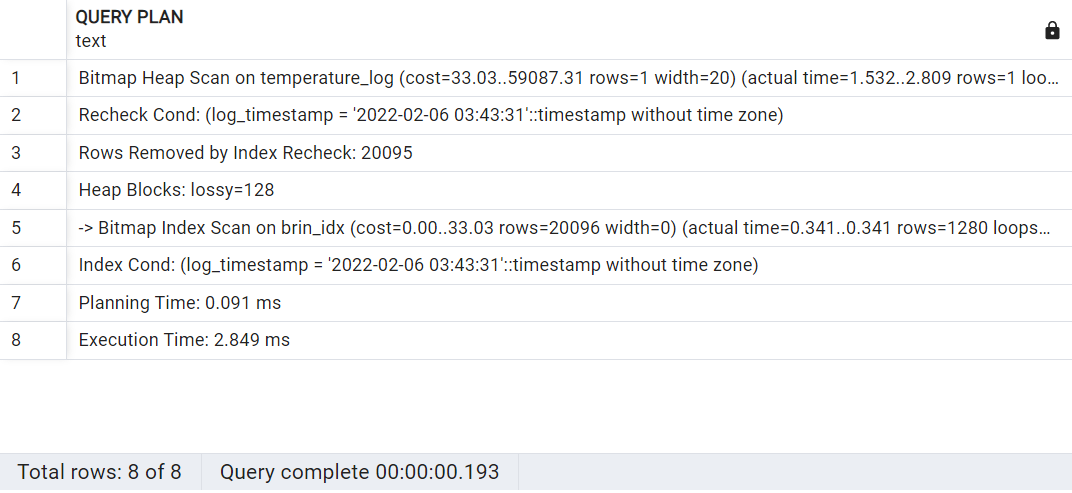


**Сворюємо індекс:**

CREATE INDEX brin\_idx ON temperature\_log

USING brin(log\_timestamp) WITH (pages\_per\_range = 128);

**Вибір даних з створеним індексом:**



|  |  |
| --- | --- |
| **До індексації** | **Після індексації** |
| 9889 ms | 2.849 ms |
| 10213 ms | 4.750 ms |
| 10198 ms | 1.470 ms |
| 9998 ms | 2.594 ms |
| 10897 ms | 2.379 ms |
| 10634 ms | 2.805 ms |
| 9788 ms | 3.696 ms |
| 8763 ms | 1.515 ms |
| 10451 ms | 1.449 ms |

|  |  |
| --- | --- |
| **10 203.4 ms** | **2.611 ms** |

Індексація brin значно пришвидшила пошук по таблиці.

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

CREATE FUNCTION baker\_stamp() RETURNS trigger AS

$baker\_stamp$

BEGIN

IF TG\_OP = 'INSERT' THEN

INSERT INTO operations (operation, timestamp)

VALUES (TG\_OP, current\_timestamp);

RETURN NEW;

END IF;

IF

NEW.name IS NULL THEN

RAISE EXCEPTION 'name cannot be null';

END IF;

IF

NEW.name IS NULL THEN

RAISE EXCEPTION 'name cannot be null';

END IF;

IF

NEW.salary IS NULL THEN

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

END IF;

IF

NEW.salary < 0 THEN

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

END IF;

INSERT INTO operations (operation, timestamp)

VALUES (TG\_OP, current\_timestamp);

RETURN NEW;

END;

$baker\_stamp$

LANGUAGE plpgsql;

CREATE TRIGGER baker\_stamp

AFTER INSERT OR

UPDATE

ON bakers

FOR EACH ROW

EXECUTE FUNCTION baker\_stamp();

**Принцип роботи тригеру**

Тригер спрацьовує перед редагуванням або після додавання в таблиці «baker». При редагуванні даних відбувається ряд перевірок:

* На наявність імені
* На коректність заробітньої платні.

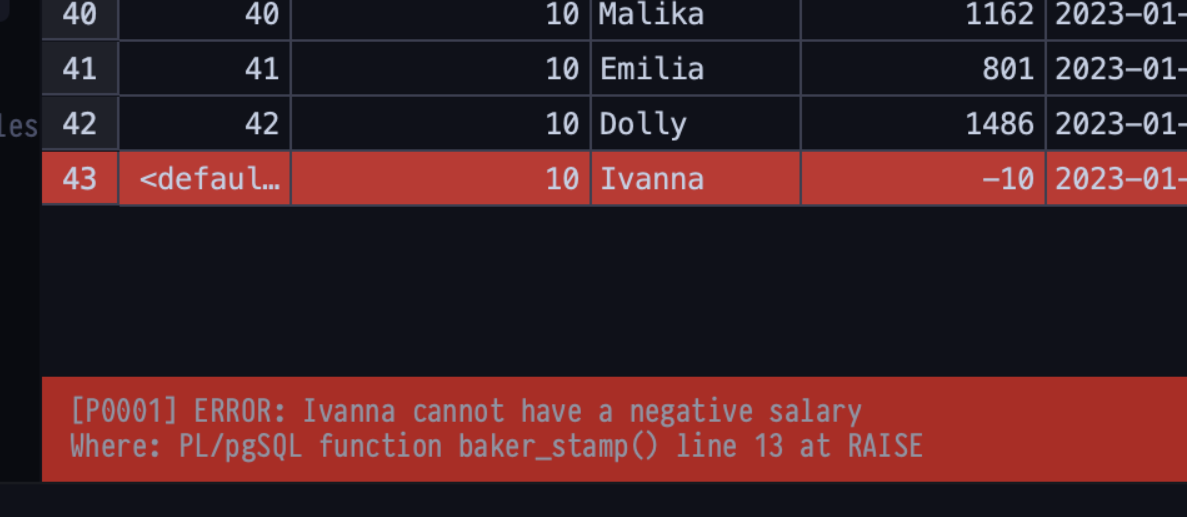
Якщо в настілок перевірки виявлені некоректно введені дані – користувачу виводиться відповідне повідомлення з текстом помилки, додатково створюється запис в таблиці operations. Також запис в таблиці operations створюється після запису в таблицю baker.

**Спробуємо задати некоректні дані:**

UPDATE bakers

SET salary = -1500

WHERE id = 1;

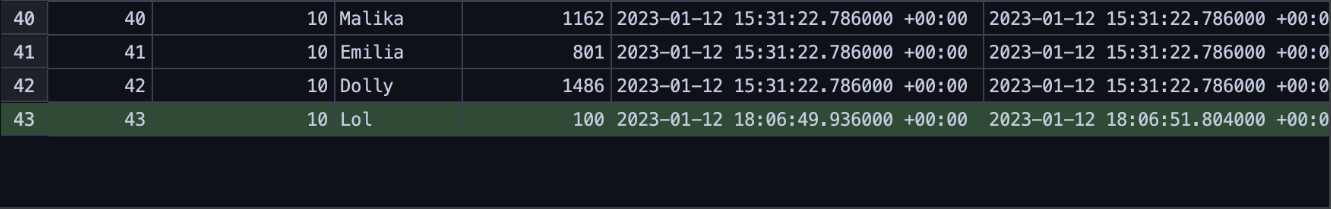


**Вводимо коректні дані:**

UPDATE bakers

SET salary = 10

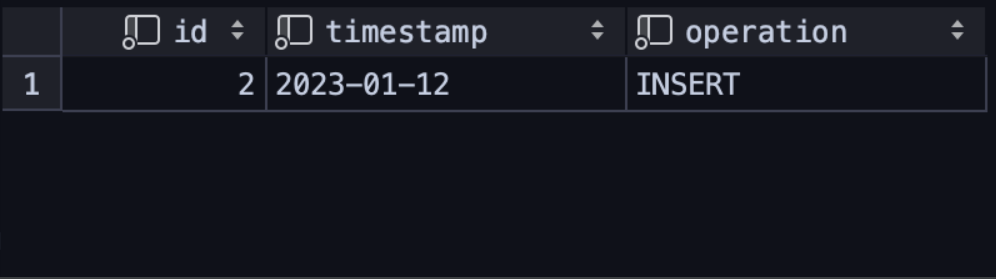
WHERE id = 43;



**Сворення нового рядка в таблиці bakers:**

INSERT INTO bakers (id, bakery\_id, name, salary, created\_at, updated\_at)

VALUES (19992, 1, 'Vova', 1299, CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP)



Тригер створює запис в таблицю operations

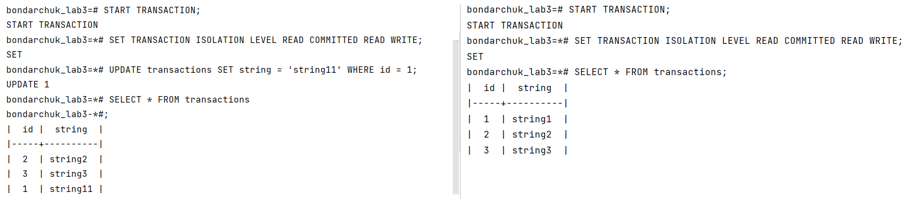
**Стандарт SQL-92**

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

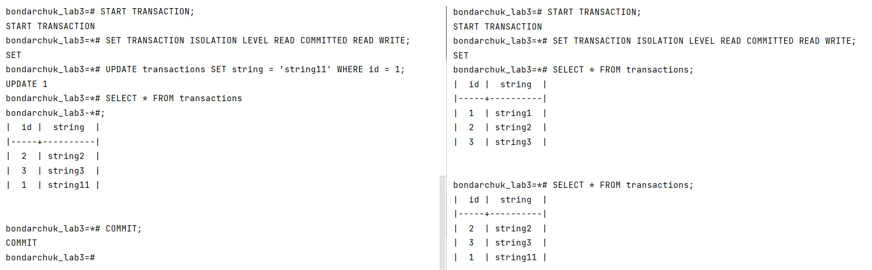
Read Committed — це стандартний рівень ізоляції в PostgreSQL. Коли транзакція використовує цей рівень ізоляції, запит SELECT (без пропозиції FOR UPDATE/SHARE) бачить лише дані, передані до початку запиту; він ніколи не бачить ані незакріплені дані, ані зміни, внесені під час виконання запиту одночасними транзакціями.

По суті, запит SELECT бачить знімок бази даних на момент початку виконання запиту. Однак SELECT бачить наслідки попередніх оновлень, виконаних у власній транзакції, навіть якщо вони ще не зафіксовані.

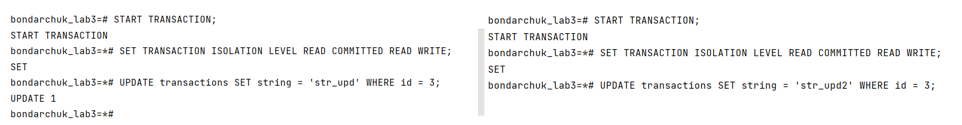
Також можна зауважити, що дві послідовні команди SELECT можуть бачити різні дані, навіть якщо вони знаходяться в одній транзакції, якщо інші транзакції фіксують зміни після запуску першого SELECT і до початку другого SELECT.



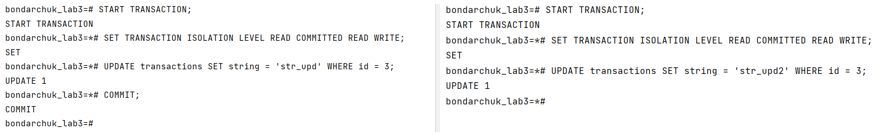
У лівій консолі змінюємо один рядок, після чого, у правій консолі було виводиться таблицю, яка залишилася незмінною.



Після того, як у лівій консолі ми робимо COMMIT, оновлені дані починають відображатися і у правій консолі.



У лівій консолі розпочинаємо та не завершуємо транзакцію, після чого у лівій консолі виконуємо іншу дію і отримуємо очікуване завершення лівої транзакції.



Після завершення лівої транзакції, операція правої завершилась успішно.

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

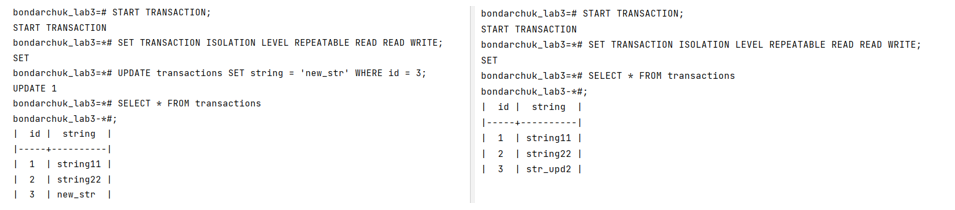
Цей рівень ізоляції дозволяє брудне читання. Одна транзакція може побачити незафіксовані зміни, внесені іншою транзакцією.

Щоб підтримувати найвищий рівень ізоляції, СУБД зазвичай отримує блокування даних, що може призвести до втрати паралельності та великих накладних витрат на блокування. Цей рівень ізоляції ігнорує цю властивість.

У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.

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

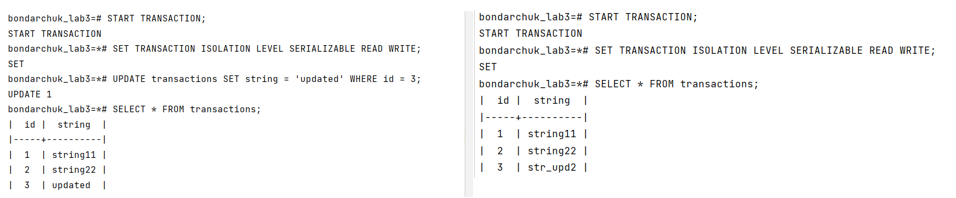
Рівень ізоляції повторюваного читання бачить лише дані, передані до початку транзакції; він ніколи не бачить незафіксовані дані або зміни, внесені під час виконання транзакцій одночасними транзакціями. (Однак запит бачить наслідки попередніх оновлень, виконаних у його власній транзакції, навіть якщо вони ще не зафіксовані.) Це більш сильна гарантія, ніж вимагається стандартом SQL для цього рівня ізоляції, і запобігає всім феноменам , за винятком аномалій серіалізації. Як згадувалося вище, це спеціально дозволено стандартом, який описує лише мінімальний захист, який повинен забезпечувати кожен рівень ізоляції.



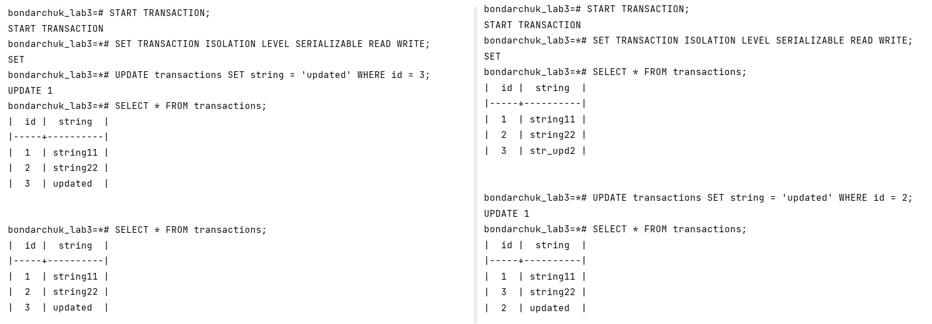
Після того, як у лівій консолі було оновлено таблицю, то у правій вона не змінилась.

1. **Serializable (впорядкованість)**

Рівень ізоляції Serializable забезпечує найсуворішу ізоляцію транзакцій. Цей рівень емулює послідовне виконання транзакцій для всіх здійснених транзакцій; ніби транзакції виконувалися одна за одною, послідовно, а не одночасно. Однак, як і рівень повторюваного читання, програми, які використовують цей рівень, повинні бути готові повторювати транзакції через помилки серіалізації. Насправді цей рівень ізоляції працює точно так само, як повторюване читання, за винятком того, що він також відстежує умови, які можуть призвести до того, що виконання одночасного набору серіалізованих транзакцій буде несумісним із усіма можливими послідовними (по одному) виконаннями цих транзакцій. Цей моніторинг не вводить жодних блокувань, окрім тих, які існують у повторюваному читанні, але є певні додаткові витрати на моніторинг, і виявлення умов, які можуть спричинити аномалію серіалізації, спричинить помилку серіалізації.



У лівій консолі змінюємо один рядок, після чого, у правій консолі виводиться таблиця, яка залишилася незмінною.



Після цього, змінюємо рядок у правій транзакції і виводимо таблиці. Як бачимо, таблиці є незалежними одна від одної.

**Посилання на GitHub:**

<https://github.com/mariabondarchuk/DB_lab_3>