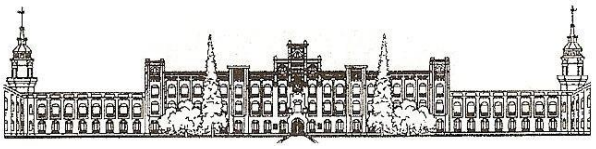


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Розрахункова графічна робота

з дисципліни «Бази даних»

«Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL»

Виконав студент групи: КВ-11

ПІБ: Гультяєв Дмитро Антонович

Telegram: @dimagultiaev

GitHub: Link

Перевірив: _____

Київ 2023

Постановка задачі:

1. Перетворити модуль “Модель” з шаблону MVC РГР 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.

Варіант:

5	*BTree, GIN*	*before update, delete*

Структура бази даних:

Сутності:

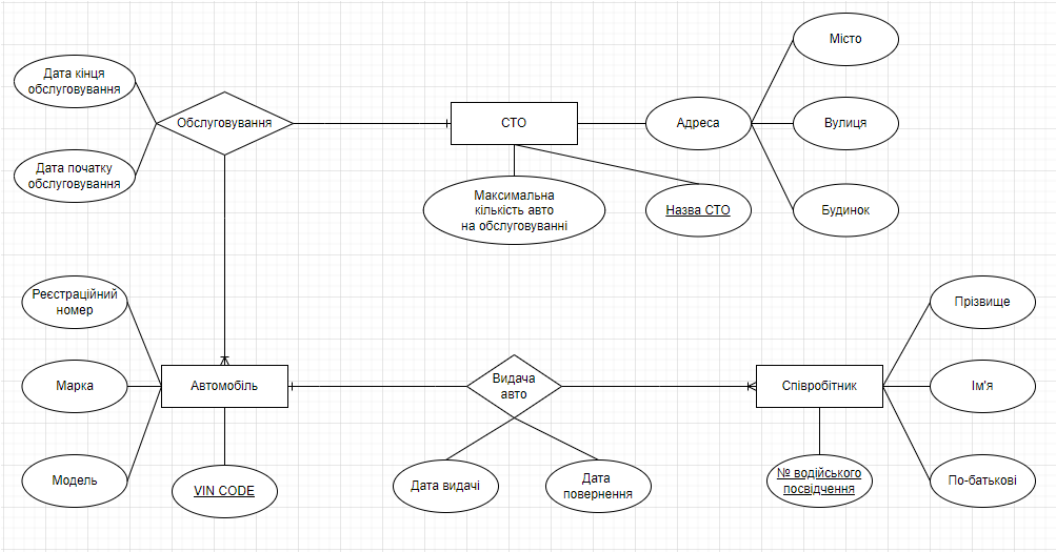
- Автомобіль (Cars) – ця сутність представляє автопарк компанії. У цій сутності ми маємо такі атрибути, як: vin, реєстраційний номер автомобіля, марку та модель.
- Співробітник (Employee) – ця сутність представляє працівників, які мають водійське посвідчення. Ці працівники можуть отримувати автомобілі в розпорядження з автопарку компанії. У цій сутності ми маємо такі атрибути, як: номер водійського посвідчення та ФІО.
- СТО (Service Station) – ця сутність представляє станцію технічного обслуговування, на якій обслуговується автопарк компанії. У цій сутності ми маємо такі атрибути, як: назва автосервісу, адреса та максимальна кількість автомобілей, яку автосервіс може обслуговувати одночасно.

Зв’язки:

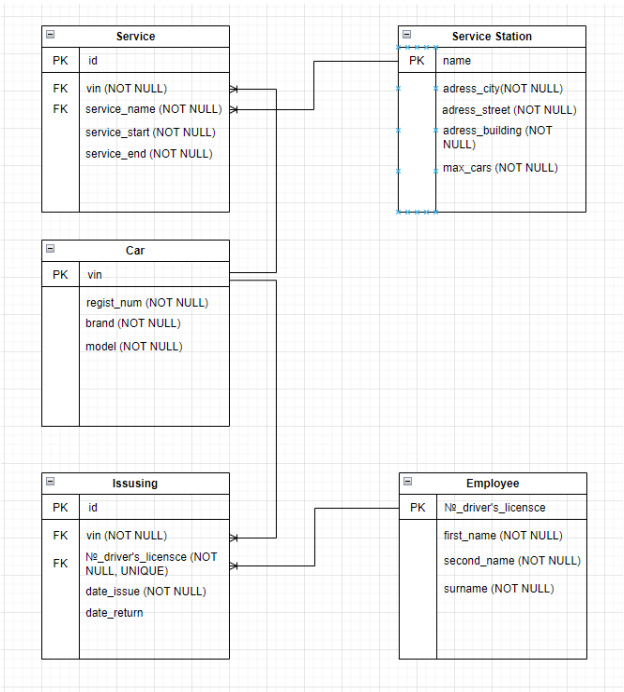
- 1:N – між автомобілем та співробітником (1 автомобіль може з часом переходити до різних співробітників).
- 1:N – між сервісом та автомобілем (1 сервіс обслуговує весь автопарк компанії).

Також я виділив такі зв’язки з атрибутами, як:

- Обслуговування (Service) – цей зв’язок поєднує сутність автомобіль та СТО
- Видача авто (Issuing) - цей зв’язок поєднує сутність автомобіль та співробітник



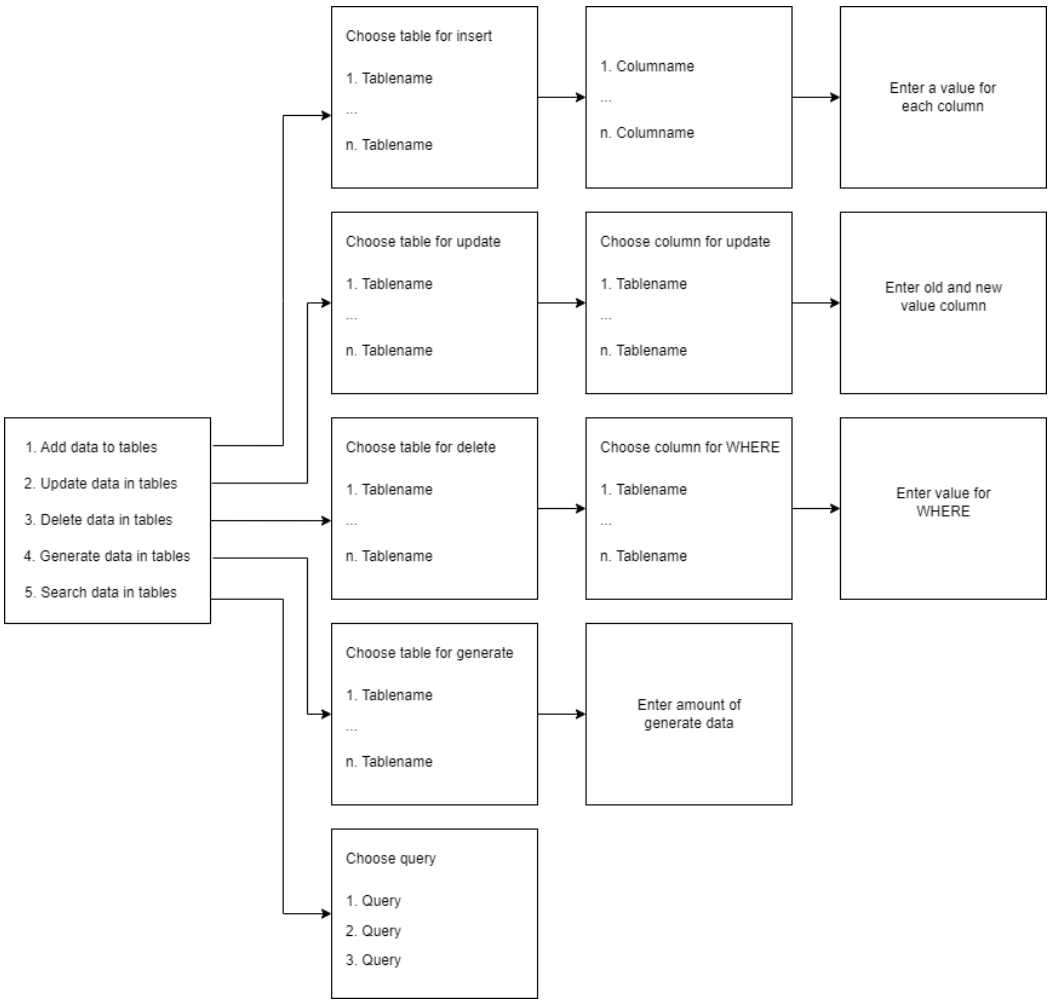
ER діаграма виконана за нотацією “Пташина лапка”



Структура бази даних

Опис меню користувача:

1) Add data to tables. Цей пункт призначений для додавання даних в таблиці. Після вибору цього пункту, користувач вибирає таблицю для вставки даних, а також вводить необхідні данні
2) Update data in tables. Цей пункт призначений для зміни даних в таблиці. Після вибору цього пункту, користувач вибирає таблицю, в якій треба змінити данні, вибирає потрібний аргумент таблиці для зміни, далі вводить старе та нове значення.
3) Delete data in tables. Цей пункт призначений для видалення даних в таблиці. Після вибору цього пункту, користувач вибирає таблицю, в якій треба видалити данні, вибирає потрібний аргумент таблиці для видалення, далі вводить значення цього елементу.
4) Generate data in tables. Цей пункт призначений для генерації даних в таблиці. Після вибору цього пункту, користувач вибирає таблицю, в якій треба згенерувати данні, а далі вводить кількість, яку необхідно згенерувати.
5) Search data in tables. Цей пункт призначений для реалізації пункту 3 завдання (пошук даних). Після вибору цього пункту, користувач вибирає один із 3 запитів, а далі вводить потрібні фільтри для цього запиту.



Графічна схема меню

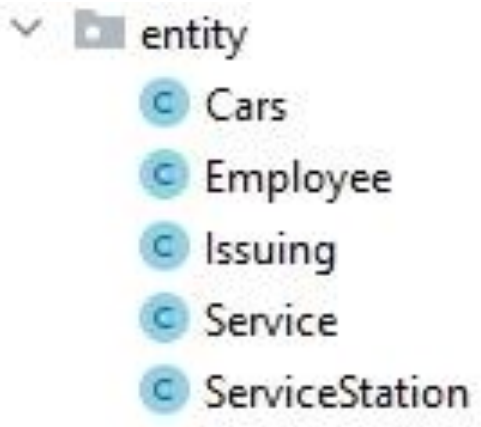
```
1. Add data to tables
2. Update data in tables
3. Delete data in tables
4. Generate data in tables
5. Search data in tables

Choose your action:
```

Головне меню в консолі

Для створення програми була обрана мова Java. А для посилання запитів до бази даних був вибраний інструмент JDBC

Завдання №1:

Було створено 5 сутностей, які відповідають за призначенням, назві та полями відповідним таблицям в базі даних.



ORM сутності

Код сутності Cars:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "\"Cars\"")
public class Cars {
    @Id
    private String vin;
    @Column(name = "regist_num")
    private String registrationNumber;
    private String brand;
    private String model;
}
```

Код сутності Employee:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "\"Employee\"")
public class Employee {
    @Id
    @Column(name = "№_drivers_license")
    private String driversLicense;
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "last_name")
    private String lastName;
    @Column(name = "surname")
    private String surName;
}
```

Код сутності Issuing:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "\"Issuing\"")
public class Issuing {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String vin;
    @Column(name = "№_drivers_licensce")
    private String driversLicense;
    @Column(name = "date_issue")
    private LocalDate dateIssue;
    @Column(name = "date_return", nullable = true)
    private LocalDate dateReturn;
}
```

Код сутності Service:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "\"Service\"")
public class Service {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private int id;
    private String vin;
    @Column(name = "service_name")
    private String serviceName;
    @Column(name = "service_start_date")
    private LocalDate serviceStartDate;
    @Column(name = "service_end_date")
    private LocalDate serviceEndDate;
}
```

Код сутності Service Station:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "\"Service Station\"")
public class ServiceStation {
    @Id
    private String name;
    @Column(name = "address_city")
    private String addressCity;
    @Column(name = "address_building")
    private String addressBuilding;
    @Column(name = "max_cars")
    private int maxCars;
}
```

Операція INSERT за допомогою ORM:

```
1. Add data to tables
2. Update data in tables
3. Delete data in tables
4. Generate data in tables
5. Search data in tables

Choose your action: 1

1. Cars
2. Employee
3. Issuing
4. Service
5. ServiceStation

Choose table: 1

Enter data for variables:
vin - 1D8HN44H98B152730
regist_num - KA8112BB
brand - BMW
model - 3
Hibernate:
    insert
    into
        "Cars"
        (brand, model, regist_num, vin)
    values
        (?, ?, ?, ?)
```

Також Hibernate (ORM) виводить запит, який він генерує до бази даних.

Операція UPDATE за допомогою ORM:

```
1. Add data to tables
2. Update data in tables
3. Delete data in tables
4. Generate data in tables
5. Search data in tables

Choose your action: 2

1. Cars
2. Employee
3. Issuing
4. Service
5. ServiceStation

Choose table: 3

Enter value of Primary Key:22
Hibernate:
    select
        i1_0.id,
        i1_0.date_issue,
        i1_0.date_return,
        i1_0."№_drivers_licensce",
        i1_0.vin
    from
        "Issuing" i1_0
    where
        i1_0.id=?

1. vin
2. №_drivers_licensce
3. date_issue
4. date_return
Choose column to update: 4

Enter value:
date_return - 2024-01-03
```

Також Hibernate (ORM) виводить проміжний запит, який він використовує для отримання множини колонок в заданій таблиці.

Операція DELETE за допомогою ORM:

```
1. Add data to tables
2. Update data in tables
3. Delete data in tables
4. Generate data in tables
5. Search data in tables

Choose your action: 3

1. Cars
2. Employee
3. Issuing
4. Service
5. ServiceStation

Choose table: 2

Enter value of Primary Key:1211307
Hibernate:
    select
        e1_0."№_drivers_license",
        e1_0.first_name,
        e1_0.last_name,
        e1_0.surname
    from
        "Employee" e1_0
    where
        e1_0."№_drivers_license"=?
Hibernate:
    delete
    from
        "Employee"
    where
        "№_drivers_license"=?
```

Також Hibernate (ORM) виводить запит, який він генерує для видалення рядка із бази даних.

Завдання №2:

BTree

Створимо таблицю для тестування і заповнемо її даними

```
CREATE TABLE btree_test_table (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50),
    age INTEGER
);

INSERT INTO btree_test_table (username, age)
SELECT
    md5(random()::text),
    FLOOR(RANDOM() * 96) + 5
FROM generate_series(1, 100000);
```

Для тестування використаємо такі запити

```
SELECT * FROM btree_test_table;
SELECT * FROM btree_test_table WHERE age = 25;
SELECT * FROM btree_test_table WHERE username LIKE 'a%';
SELECT * FROM btree_test_table WHERE age BETWEEN 30 and 40;
SELECT * FROM btree_test_table WHERE username LIKE '%9%';
```

Результати до створення BTree індекса

✓ Successfully run. Total query runtime: 389 msec. 100000 rows affected. ×

✓ Successfully run. Total query runtime: 394 msec. 1044 rows affected. ×

✓ Successfully run. Total query runtime: 129 msec. 6304 rows affected. ×

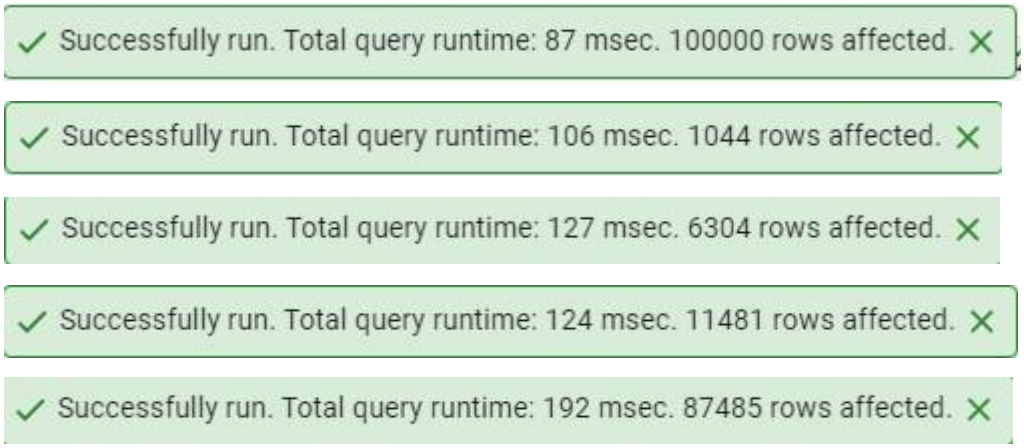
✓ Successfully run. Total query runtime: 453 msec. 11481 rows affected. ×

✓ Successfully run. Total query runtime: 130 msec. 87485 rows affected. ×

Створюємо BTree індекс

```
1  CREATE INDEX btree_index ON btree_test_table USING btree (age);
```

Результати після створення BTree індекса



Створивши індекс для стовпця age ми бачимо значне покращення запити по отриманню всієї таблиці, а також значно пришвидшилися запити пов’язані з стовпцем age. При роботі зі стовпцем name час виконання запиту є приблизно однаковим. Значне поркащення швидкості операцій з age пов'язано з тим, що BTree індекс ідеально підходить для даних, які можна чітко відсортувати (наприклад числа).

GIN

Створимо таблицю для тестування

```
CREATE TABLE gin_test_table (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50),
    age INTEGER
);

INSERT INTO gin_test_table (username, age)
SELECT
    md5(random()::text),
    FLOOR(RANDOM() * 96) + 5
FROM generate_series(1, 100000);
```

Для тестування використаємо такі запити

```
SELECT * FROM gin_test_table;
SELECT * FROM gin_test_table WHERE age = 25;
SELECT * FROM gin_test_table WHERE username LIKE 'a%';
SELECT * FROM gin_test_table WHERE age BETWEEN 30 and 40;
SELECT * FROM gin_test_table WHERE username LIKE '%9%';
```

Результати до створення GIN індекса

✓ Successfully run. Total query runtime: 70 msec. 100000 rows affected. ×

✓ Successfully run. Total query runtime: 55 msec. 1032 rows affected. ×

✓ Successfully run. Total query runtime: 60 msec. 6263 rows affected. ×

✓ Successfully run. Total query runtime: 159 msec. 11307 rows affected. ×

✓ Successfully run. Total query runtime: 185 msec. 87331 rows affected. ×

Створюємо GIN індекс

```
CREATE INDEX gin_index ON gin_test_table USING GIN (username);
```

Результати після створення GIN індекса

✓ Successfully run. Total query runtime: 90 msec. 100000 rows affected. ×

✓ Successfully run. Total query runtime: 138 msec. 1032 rows affected. ×

✓ Successfully run. Total query runtime: 139 msec. 6263 rows affected. ×

✓ Successfully run. Total query runtime: 72 msec. 11307 rows affected. ×

✓ Successfully run. Total query runtime: 92 msec. 87331 rows affected. ×

Спостерігаємо, що сильного покращення результатів не відбулося, а іноді результати навіть гірші, ніж до створення індексу. Скоріше за всього, GIN є не

ефективним, так як типи атрибутів таблиці не є оптимальними для роботи з цим типом індексу.

Завдання №3:

Створимо 3 таблиці:

```
CREATE TABLE employees_test (
    id serial PRIMARY KEY,
    name VARCHAR(100),
    salary numeric
);

CREATE TABLE salary_history (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    salary numeric
);

CREATE TABLE rem_workers(
    id SERIAL PRIMARY KEY,
    name VARCHAR(100)
);
```

Заповнимо таблицю employees_test даними:

```
INSERT INTO employees_test (name, salary)
VALUES
('Anna', 1200),
('Sergiy', 900),
('Andriy', 1500),
('Vasyl', 600),
('Anton', 1700);
```

	id [PK] integer	name character varying (100)	salary numeric
1	1	Anna	1200
2	2	Sergiy	900
3	3	Andriy	1500
4	4	Vasyl	600
5	5	Anton	1700

Створимо тригер:

```
CREATE OR REPLACE FUNCTION before_update_trigger()
RETURNS TRIGGER AS $$
DECLARE
    emp_record employees_test%ROWTYPE;
BEGIN
    IF TG_OP = 'UPDATE' THEN
        IF OLD.name LIKE 'A%' THEN
            INSERT INTO salary_history(name, salary) VALUES (OLD.name, OLD.salary);
            RETURN NEW;
        ELSE
            RAISE EXCEPTION 'The name does not start with the letter A';
        END IF;
    ELSIF TG_OP = 'DELETE' THEN
        FOR emp_record IN SELECT * FROM employees_test
        LOOP
            INSERT INTO rem_workers(name) VALUES (emp_record.name);
        END LOOP;
    END IF;
    RETURN NEW;

END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OR DELETE ON employees_test
FOR EACH ROW
EXECUTE FUNCTION before_update_trigger();
```

Робота тригера полягає в тому що перед update перевіряється чи починається ім’я оновлювального користувача на букву А, якщо так то він поміщається в таблицю salary_history. Якщо ні – то виводиться виняток, що ім’я починається не на А.

Під час операції delete ми перносимо всіх працівників, які лишилися в таблицю rem_workers за допомогою цикла.

Тригер не має чіткої бізнес логіки, але показує принцип роботи тригерів, умов, циклів та винятків.

Проведемо UPDATE:

```
1  UPDATE employees_test SET salary = 700 WHERE id = 4
```

Data Output | Messages | Notifications

```
ERROR: ПОМИЛКА:  The name does not start with the letter A
CONTEXT:  Функція PL/pgSQL before_update_trigger() рядок 9 в RAISE
```

Ми намагаємося провести UPDATE користувача у якого ім’я починається не на A, тому з’являється виняток і користувач на поміщається в таблицю salary_history.

```
UPDATE employees_test SET salary = 1400 WHERE id = 1
```

	id [PK] integer	name character varying (100)	salary numeric
1	1	Anna	1200

Якщо UPDATE Anna – то вона додється в таблицю salary_history.

Проведемо DELETE:

```
DELETE FROM employees_test WHERE id = 4;
```

	id [PK] integer	name character varying (100)	salary numeric
1	2	Sergiy	900
2	3	Andriy	1500
3	5	Anton	1700
4	1	Anna	1400

	id [PK] integer	name character varying (100)
1	1	Sergiy
2	2	Andriy
3	4	Anton
4	5	Anna

Отже, Vasyl видаляється із основної таблиці, а всі інші працівники додаються в таблицю rem_workers.

Завдання №4:

При паралельному виконанні транзакцій можливі виникненя таких проблем:

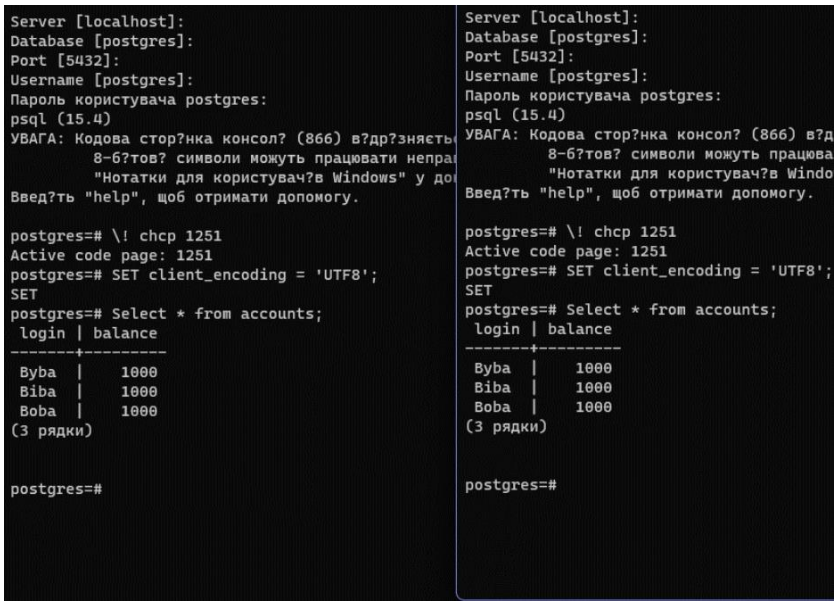
1. **Втрачене оновлення** Ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.

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.

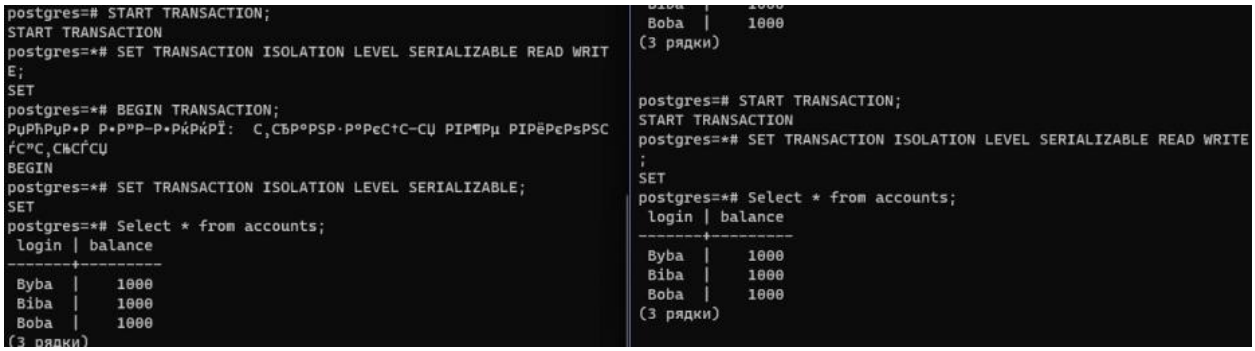
4. **Фантомне читання** Ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

Створимо таблицю:

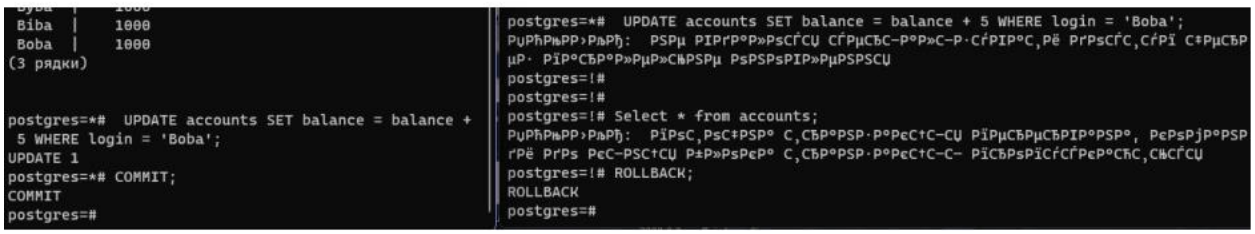


Serializable

Serializable (немає жодних побічних ефектів) забезпечується блокуванням і на запис, і на читання будь-якого блоку даних, з яким ми працюємо. Блокується навіть вставка даних, які можуть потрапити в блок, який ми прочитали. Таким чином, за рахунок низької конкурентності, забезпечується відсутність навіть фантомних читань.



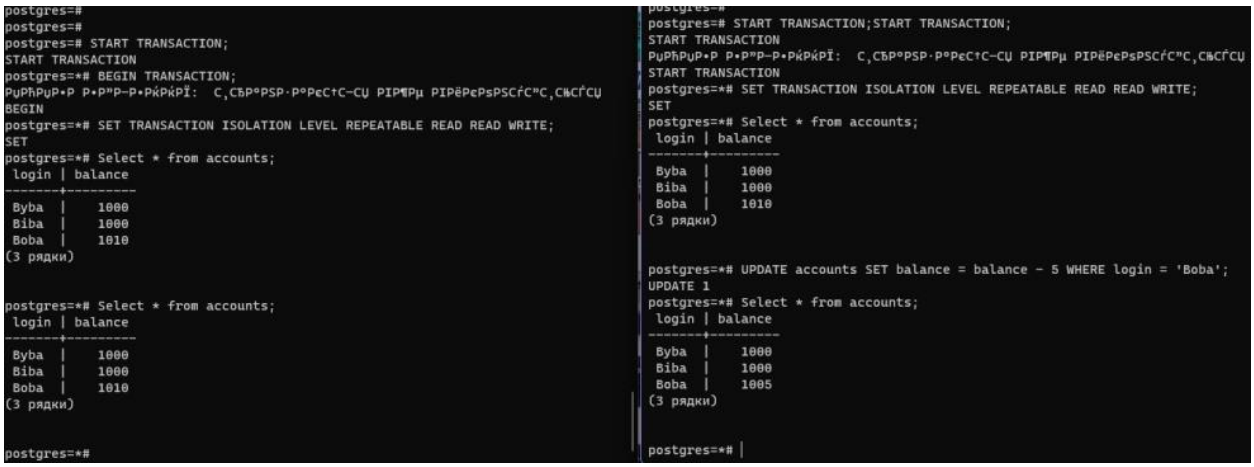
Дані в транзакціях ізольовані, а отже при оновленні даних одна транзакція блокується поки не закінчиться інша



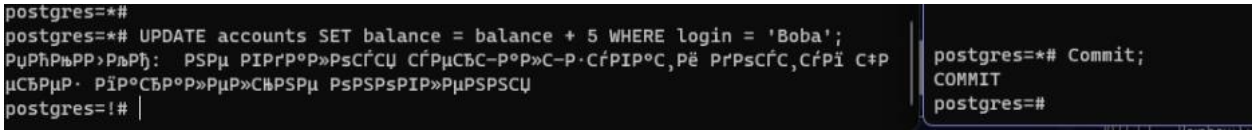
Repeatable Read

Repeatable read (тобто відсутність усього, крім фантомних читань) забезпечується блокуванням на запис даних (рядків), які ми намагаємося прочитати, за тим винятком, що блокування на запис працює до кінця транзакції, а не окремої операції. Одного разу "торкнувшись" блоку даних, транзакція блокує його зміну до кінця роботи, що забезпечує відсутність dirty reads і non-repeatable reads. Читання одного і тогож самого рядка в транзакції дає однакову відповідь.

Читання одного і тогож самого рядка в транзакції дає однакову відповідь.

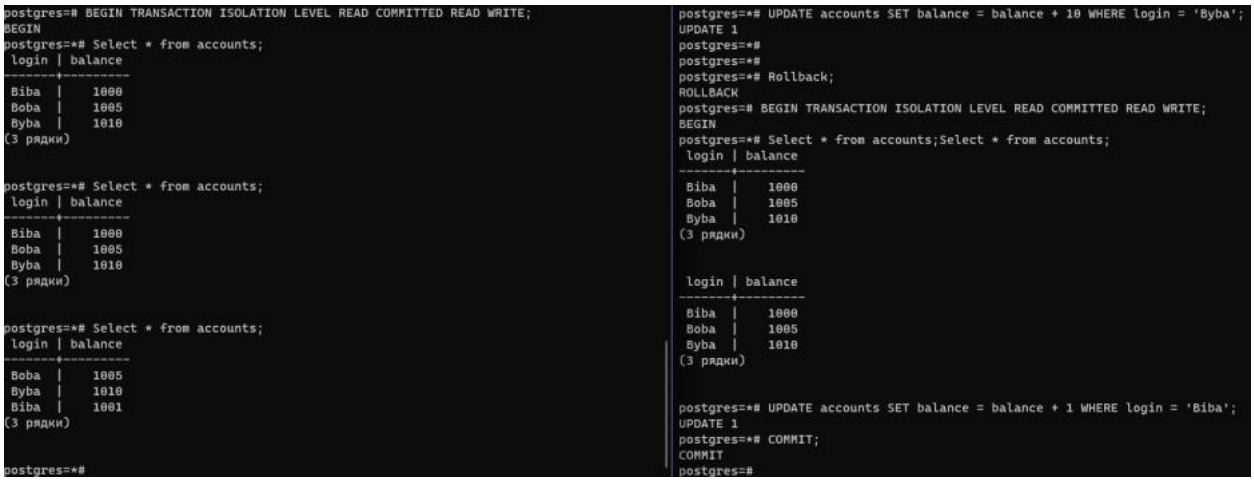


Відповідно транзакція може змінити данні тільки після закінчення іншої.



Read Committed

Read committed (тобто відсутність dirty reads) забезпечується блокуванням на запис даних (рядків), які ми намагаємося прочитати. Це блокування гарантує, що ми почекаємо завершення транзакцій, які вже змінюють наші дані, або змусимо їх почекати, поки ми будемо читати. У підсумку, ми точно прочитаємо тільки дані, які були закоммічені, уникнувши тим самим брудне читання. Цей режим - типовий приклад песимістичного блокування, оскільки ми блокуємо дані на запис, навіть якщо в них ніхто реально не пише.



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.