Національний технічний університет України
“Київський політехнічний інститут”

Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп’ютерних систем

Лабораторна робота №2

з дисципліни

“ "Бази даних та засоби управління”

ТЕМА: “ Засоби оптимізації роботи СУБД PostgreSQL

"

Група: КВ-11

Виконала: Нестерук А.О.

Оцінка:

Київ – 2023

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Завдання роботи полягає у наступному:

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 16

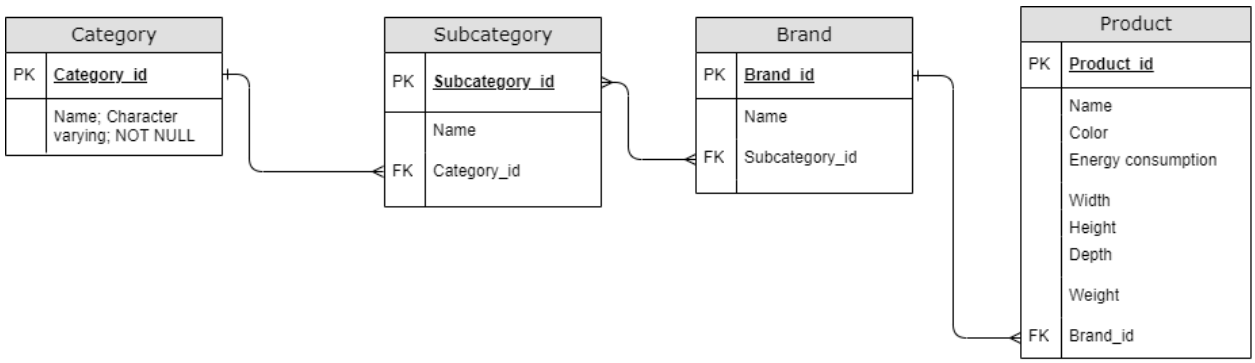
16	*GIN, Hash*	*after delete, insert*

Посилання на телеграм та репозиторій:

https://t.me/jemapel_sasuke_uchiwa

https://github.com/FLeD-jk/LAB2

Відомості про обрану предметну галузь з лабораторної роботи №1

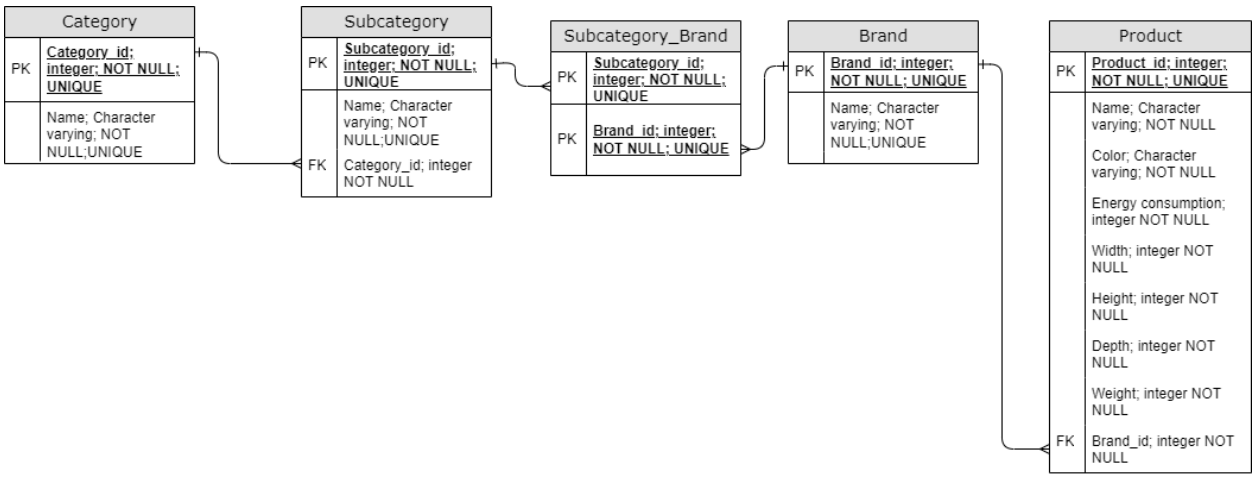


ER-діагарама побудована за нотацією «Crow`s foot».

Опис предметної галузі

Дана предметна галузь реалізує електронний довідник для зберігання технічних характеристик товарів.

Перетворення моделі у схему бази даних



Завдання №1

Для перетворити модуля “Model” з шаблону MVC PГР у вигляд об’єктно-реляційної проекції (ORM) було використано бібліотеку SQLAlchemy.

Класи ORM:

```
class Category(Base):
    __tablename__ = 'Category'

    Category_id = Column(Integer, primary_key=True)
    Name = Column(String(30), unique=True, nullable=False)
    subcategories = relationship('SubCategory')

class SubCategory(Base):
    __tablename__ = 'SubCategory'

    SubCategory_id = Column(Integer, primary_key=True)
    Name = Column(String(30), unique=True, nullable=False)
    Category_id = Column(Integer,
ForeignKey('Category.Category_id'))
    brands = relationship('Brand', secondary='SubCategory_Brand')

class Brand(Base):
    __tablename__ = 'Brand'

    Brand_id = Column(Integer, primary_key=True)
    Name = Column(String(30), unique=True, nullable=False)
    products = relationship('Product')
```

```
class Product(Base):
    __tablename__ = 'Product'

    Product_id = Column(Integer, primary_key=True)
    Name = Column(String(30), nullable=False)
    Color = Column(String(30), nullable=False)
    Width = Column(Integer, nullable=False)
    Height = Column(Integer, nullable=False)
    Deepth = Column(Integer, nullable=False)
    Weight = Column(Integer, nullable=False)
    Energy_consumption = Column(Integer, nullable=False)
    Brand_id = Column(Integer, ForeignKey('Brand.Brand_id'))

class SubCategory_Brand(Base):
    __tablename__ = 'SubCategory_Brand'

    SubCategory_id = Column(Integer,
ForeignKey('SubCategory.SubCategory_id'), primary_key=True)
    Brand_id = Column(Integer, ForeignKey('Brand.Brand_id'),
primary_key=True)
```

Програма працює ідентично розрахунково-графічній роботі.
Приклад отримання всіх даних з таблиці SubCategory:

```
subcategories = self.session.query(SubCategory).all()
```

Завдання №2

Hash-індекс

```
CREATE TABLE Hash
(
    id serial,
    name character varying(50) NOT NULL,
    age integer NOT NULL,
    CONSTRAINT h_pkey PRIMARY KEY (id)
);

truncate Hash;

insert into Hash (name, age)
SELECT  md5(random()::text),
    floor(random() * 100 + 1)::int
FROM generate_series(1, 1000000) ;
```

	id [PK] integer	name character varying (50)	age integer
1	4400001	1767744481fca461d3705bf96e443710	87
2	4400002	6d008c615ce65d17f36709dbc02e52d9	11
3	4400003	40e71627e35ebe906c4aaac3b2e8b3...	30
4	4400004	5839f8634bb27db2127a8962541688...	100
5	4400005	88d373e1246ea6736878a6d1d1fa8f5b	100
6	4400006	18a82dcb496d10875ef4e5deda57b598	34
7	4400007	987c777c873e49728d95585253d026...	42
8	4400008	296b44747e548a3bdaadc611b0c6cd...	83
9	4400009	9e67540fe5a740b0d9f2b637bb88a7dd	16
10	4400010	c51be04db02dbad22c67e177bb9ba5...	2

Тестування на 5-х запитах:

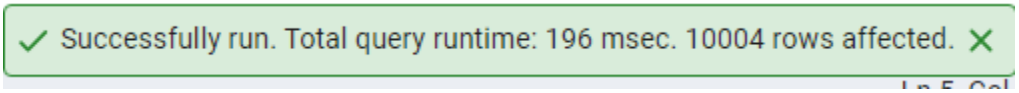
```
SELECT * FROM Hash WHERE name LIKE 'a%';
SELECT COUNT(*) FROM Hash where name='a7fad4c30f3b4331ccf8ada522936d77';
SELECT * FROM Hash WHERE age = 23;
SELECT * FROM Hash WHERE name LIKE '%12%';
SELECT * FROM Hash WHERE age BETWEEN 20 AND 30;
```

Результати без індексу:

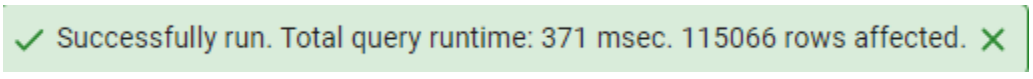
1:

2:

3:



4:



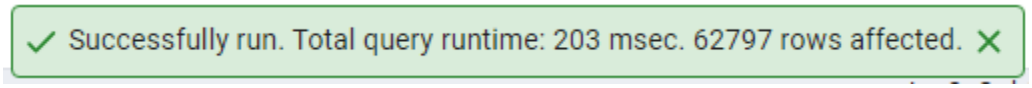
5:

Створимо індекс:

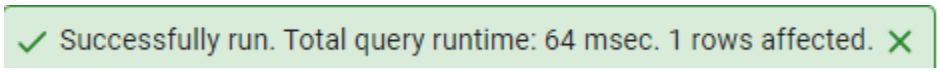
```
create index index_hash on Hash (name);
```

Результати з індексом:

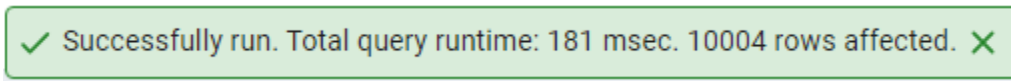
1:



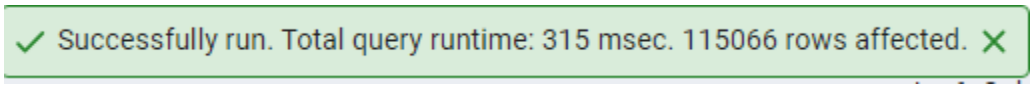
2:



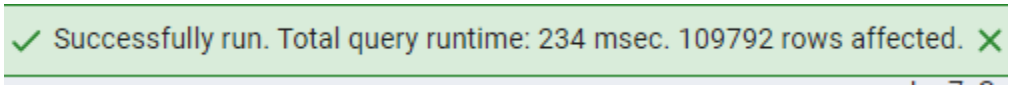
3:



4:



5:



Індекс hash ефективний для простих порівнянь, і не підходить для діапазонів або не є сильно ефективним рішенням для використання функцій LIKE. Оскільки шукаючи точні значення хеш-функція може швидко визначити, в якому сегменті таблиці знаходиться потрібний рядок. Це дозволяє уникнути необхідності сканувати всю таблицю, що значно прискорює пошук.

Gin-індекс

```
CREATE TABLE GIN
(
    id serial,
    name character varying(50) NOT NULL,
    age integer NOT NULL,
    CONSTRAINT g_pkey PRIMARY KEY (id)
);

truncate GIN;

insert into GIN (name, age)
SELECT  md5(random()::text),
    floor(random() * 100 + 1)::int
FROM generate_series(1, 1000000) ;
```

	id [PK] integer	name character varying (50)	age integer
1	1	1d110e5e0b6e2dbfb80cf241cfab55a9	48
2	2	73b403fb957fee70482f9cef96afc0b2	62
3	3	82047f13027ad65e947b0546e1e35e4e	75
4	4	ca84d09022842ef82c9e57bea0c57a1b	75
5	5	e469a6ddd8c474bd3f5ecaec8ead8985	87
6	6	4335b4e83406f22c1d0ef7c2b7ed3682	87
7	7	93a36cf4f7f7c00897caccf68f676a4a	48
8	8	4e87c7a85a1969a72cd5c0d7d89b5714	85
9	9	23734dc37993a7978a85065820c240ef	33
10	10	9ec840e230b7c0be568dc83222cd1c68	57

Тестування на 5-х запитах:

```
SELECT COUNT(*) FROM Hash where age >15;
SELECT * FROM GIN WHERE length(name) >= 32;
SELECT * FROM GIN WHERE name ILIKE 'A%'AND age BETWEEN 20 AND 30;
SELECT * FROM GIN WHERE name ILIKE '1%2%' or name='a7fad4c30f3b4331ccf8ada522936d77';
SELECT * FROM GIN WHERE name LIKE '[aeiou]%' OR name LIKE '[bcdfghjklmnpqrstvwxyz]%';
```

Результати без індексу:

1:

✓ Successfully run. Total query runtime: 235 msec. 1 rows affected. ×

2:

✓ Successfully run. Total query runtime: 860 msec. 1000000 rows affected. ×

3:

✓ Successfully run. Total query runtime: 1 secs 11 msec. 6805 rows affected. ×

4:

✓ Successfully run. Total query runtime: 1 secs 115 msec. 53944 rows affected. ×

5:

✓ Successfully run. Total query runtime: 295 msec. 0 rows affected. ×

Створимо індекс:

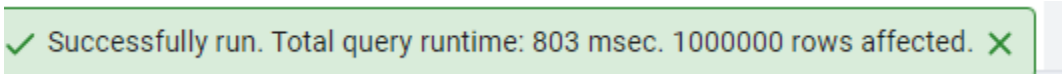
```
CREATE INDEX gin_index ON GIN (name, age);
```

Результати з індексом:

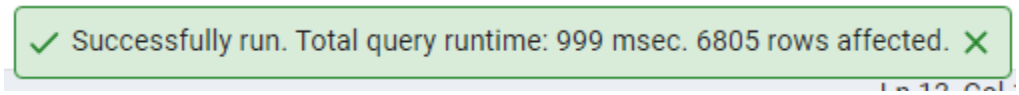
1:

✓ Successfully run. Total query runtime: 204 msec. 1 rows affected. ×

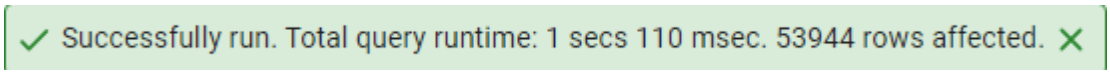
2:



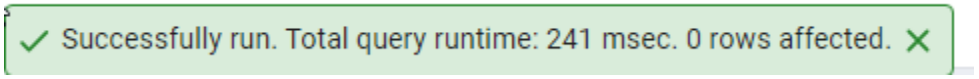
3:



4:



5:



Індекс gin ,скоріше за все, виявляється неекфективним через невідповідність типів атрибутів таблиці.

As described in the Postgres documentation, the tsvector GIN index structure is focused on lexemes:

> “GIN indexes are the preferred text search index type. As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations. Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words.”

Якщо змінити тип поля таблиці то результат стане краще:

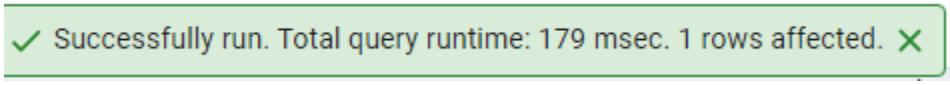
```
CREATE TABLE GIN
(
    id serial,
    name tsvector  NOT NULL,
    age integer NOT NULL,
    CONSTRAINT g_pkey PRIMARY KEY (id)
);

truncate GIN;

INSERT INTO GIN (name, age)
SELECT to_tsvector(random()::text),
       floor(random() * 100 + 1)::int
FROM generate_series(1, 1000000);
```

Тестування на 5-х запитах:

```
explain analyse SELECT COUNT(*) FROM GIN where age >15;
explain SELECT * FROM GIN WHERE length(name) >= 32;
explain SELECT * FROM GIN WHERE  name='A6764378ftyjnf46'AND age BETWEEN 20 AND 30;
explain SELECT * FROM GIN WHERE name ILIKE '1%2%' or name='a7fad4c30f3b4331ccf8ada522936d77';
explain SELECT * FROM GIN WHERE name LIKE '[aeiou]%' OR name LIKE '[bcdfghjklmnpqrstvwxyz]%';
```

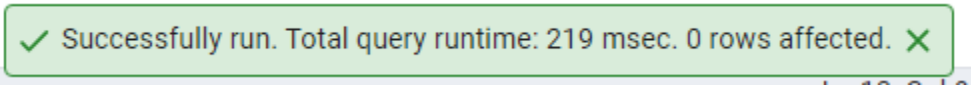
Результати без індексу:

1:



2:

3:

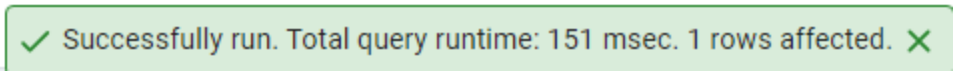


Створимо індекс:

```
CREATE INDEX gin_index ON GIN (name, age);
```

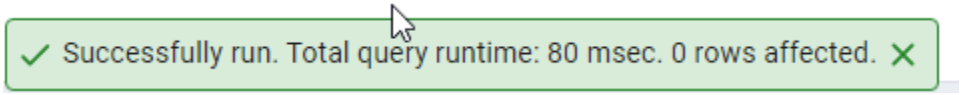
Результати з індексом:

1:



2:

3:



Але в такому випадку неможливо використовувати функції LIKE, ILIKE для tsvector.

Завдання №3

Тригери *after delete, insert*

Таблиці:

```
CREATE TABLE employees (
   id INT GENERATED ALWAYS AS IDENTITY,
   first_name VARCHAR(40) NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   skill_group INT NOT NULL,
   PRIMARY KEY(id)
);


CREATE TABLE dismissed_employees (
   id INT GENERATED ALWAYS AS IDENTITY,
   employee_id INT NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   skill_group INT NOT NULL
```

```
);
```

Тригер:

```
CREATE OR REPLACE FUNCTION delete_insert_func()
RETURNS TRIGGER AS $$
DECLARE
    old_employee employees%ROWTYPE;
BEGIN
    IF old.skill_group>2 THEN
        INSERT INTO dismissed_employees (employee_id, last_name, skill_group)
        VALUES (OLD.id, OLD.last_name, OLD.skill_group);
        RETURN NEW;
    ELSIF TG_OP = 'INSERT'  THEN
        IF length(NEW.first_name) <= 1 OR NEW.first_name IS NULL THEN
            RAISE EXCEPTION 'The first name cannot be less than 1 characters';
            return NULL;
        END IF;
        IF length(NEW.last_name) <= 1 OR NEW.last_name IS NULL THEN
            RAISE EXCEPTION 'The last name cannot be less than 1 characters';
            return NULL;
        END IF;
        FOR old_employee IN SELECT * FROM employees LOOP
            UPDATE employees SET skill_group = old_employee.skill_group + 1
WHERE id = old_employee.id;
        END LOOP;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER employees_insert_delete_trigger
AFTER DELETE OR INSERT ON employees
FOR EACH ROW
EXECUTE FUNCTION delete_insert_func();
```

Тригер працює наступним чином, після видалення робітника з рівнем навичок більше 2 він потрапляю до таблиці звільнених співробітників. При вставці нових значень неможливо вставити співробтника у якого ім’я або прізвище коротше 2 символів, також придодаванні нового співробітника у старих підвищується рівень навичок. (не питайте чому)
Додамо 5 рядків до employees:

```
INSERT INTO employees (first_name, last_name, skill_group)
VALUES
  ('John', 'Doe', 1),
  ('Jane', 'Smith', 2),
  ('Robert', 'Johnson', 3),
  ('Emily', 'Jones', 4),
  ('Michael', 'Williams', 5);
```

	id [PK] integer	first_name character varying (40)	last_name character varying (40)	skill_group integer
1	1	John	Doe	1
2	2	Jane	Smith	2
3	3	Robert	Johnson	3
4	4	Emily	Jones	4
5	5	Michael	Williams	5

Тепер додамо 1 рядок:

```
  INSERT INTO employees (first_name, last_name, skill_group)
VALUES
  ('Bob', 'Marly', 1);
```

	id [PK] integer	first_name character varying (40)	last_name character varying (40)	skill_group integer
1	1	John	Doe	2
2	2	Jane	Smith	3
3	3	Robert	Johnson	4
4	4	Emily	Jones	5
5	5	Michael	Williams	6
6	6	Bob	Marly	1

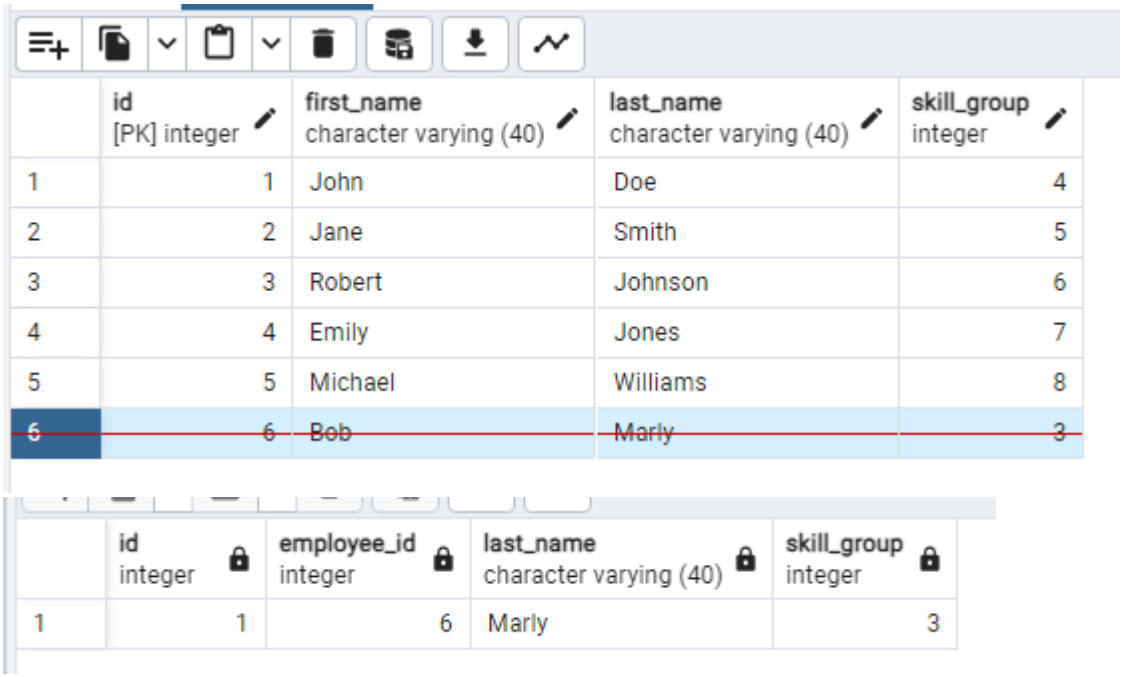
Неможливо вставити рядок з ім’ям/прізвищем у якого меньше 2 символів:

```
12    INSERT INTO employees (first_name, last_name, skill_group)
13  VALUES
14    ('B', 'M', 1);
```

Messages Data Output Notifications

```
ERROR:  The first name cannot be less than 1 character
CONTEXT:  Функція PL/pgSQL delete_insert_func() рядок 13 в RAISE

ПОМИЛКА:  The first name cannot be less than 1 character
SQL state: P0001
```

Видалимо Боба:



	id [PK] integer	first_name character varying (40)	last_name character varying (40)	skill_group integer
1	1	John	Doe	4
2	2	Jane	Smith	5
3	3	Robert	Johnson	6
4	4	Emily	Jones	7
5	5	Michael	Williams	8
6	6	Bob	Marly	3

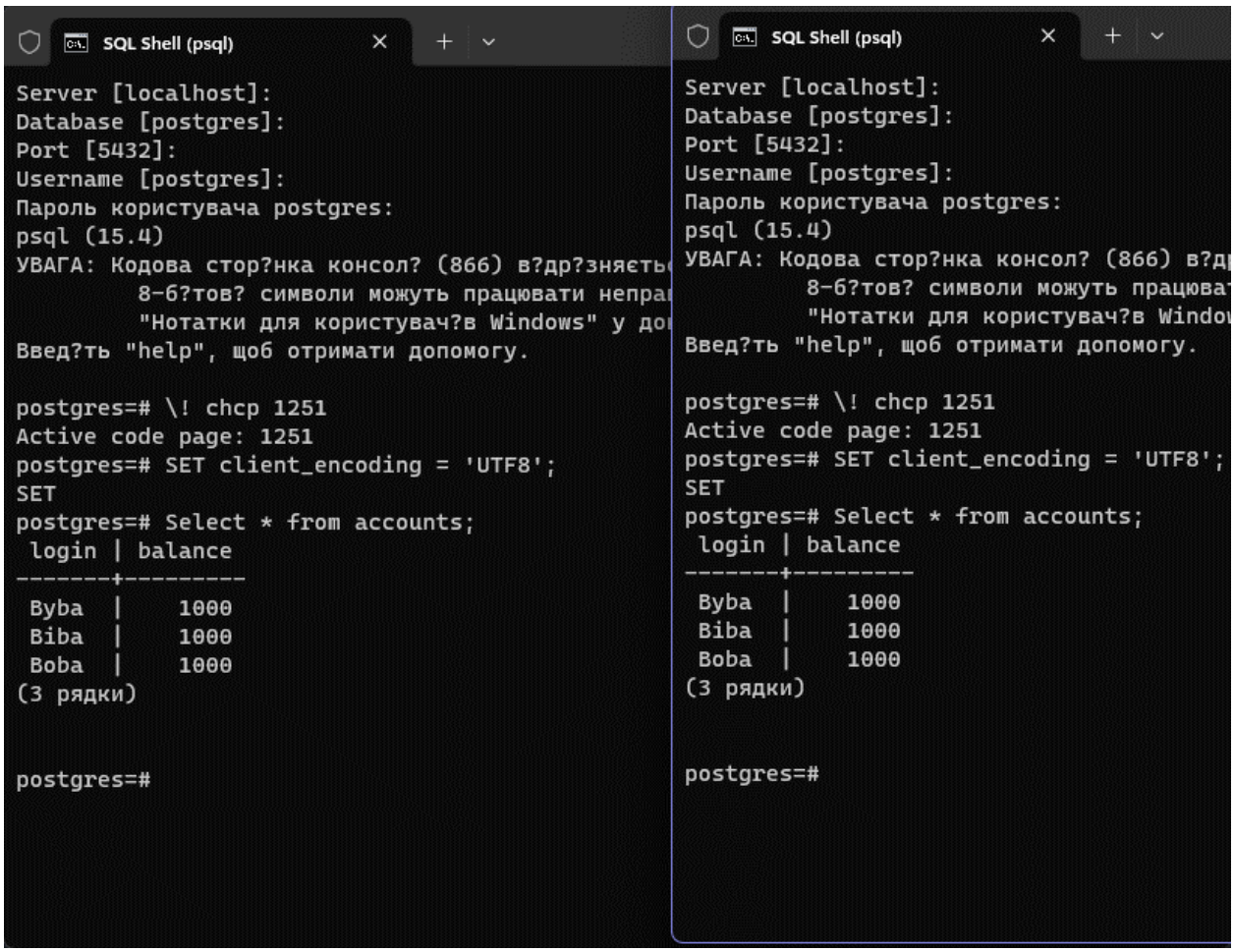
	id integer	employee_id integer	last_name character varying (40)	skill_group integer
1	1	6	Marly	3

Завдання №4

При паралельному виконанні транзакцій можливі виникненя таких проблем:

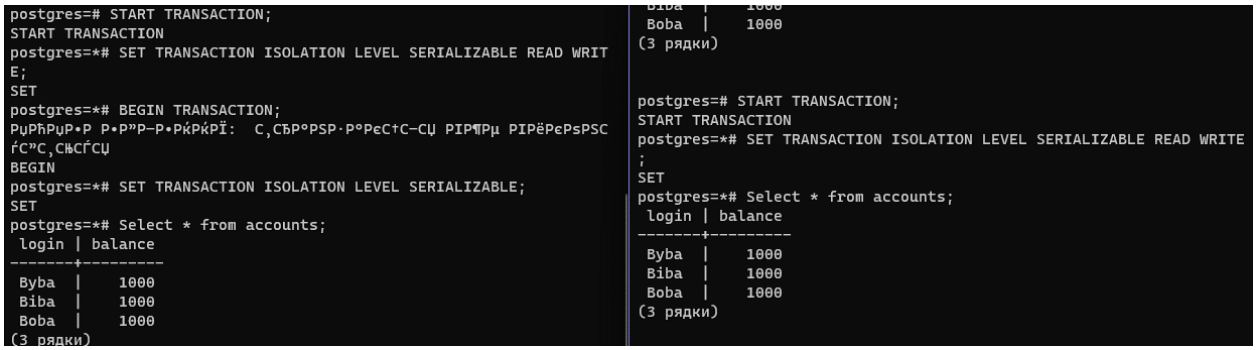
1. Втрачене оновлення
Ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.
2. «Брудне» читання
Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).
3. Неповторюване читання
Ситуація, коли при повторному читанні 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.

```
CREATE TABLE IF NOT EXISTS accounts
(   login VARCHAR(255) NOT NULL,
 balance BIGINT DEFAULT 0 NOT NULL);

insert into accounts (login, balance) values ('Byba', 1000);
insert into accounts (login, balance) values ('Biba', 1000);
insert into accounts (login, balance) values ('boba', 1000);
```

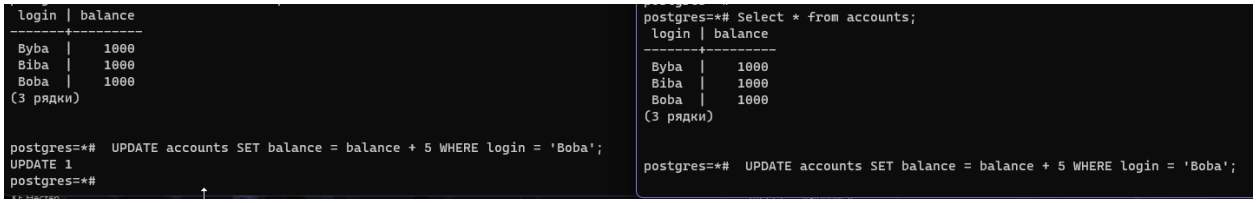


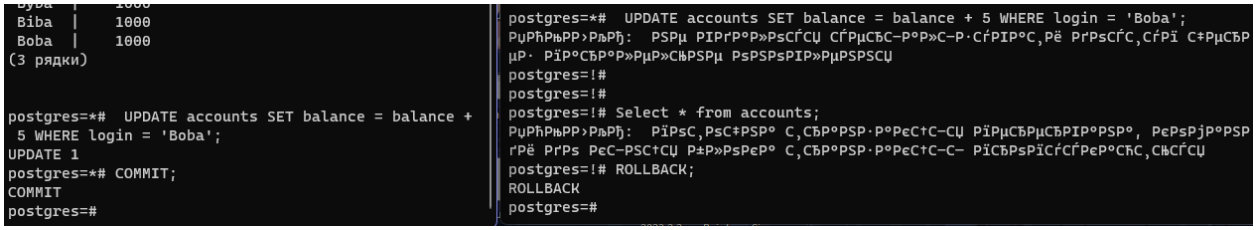
Serializable

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. При оновленні даних одна транзакція блокується поки не закінчиться інша:

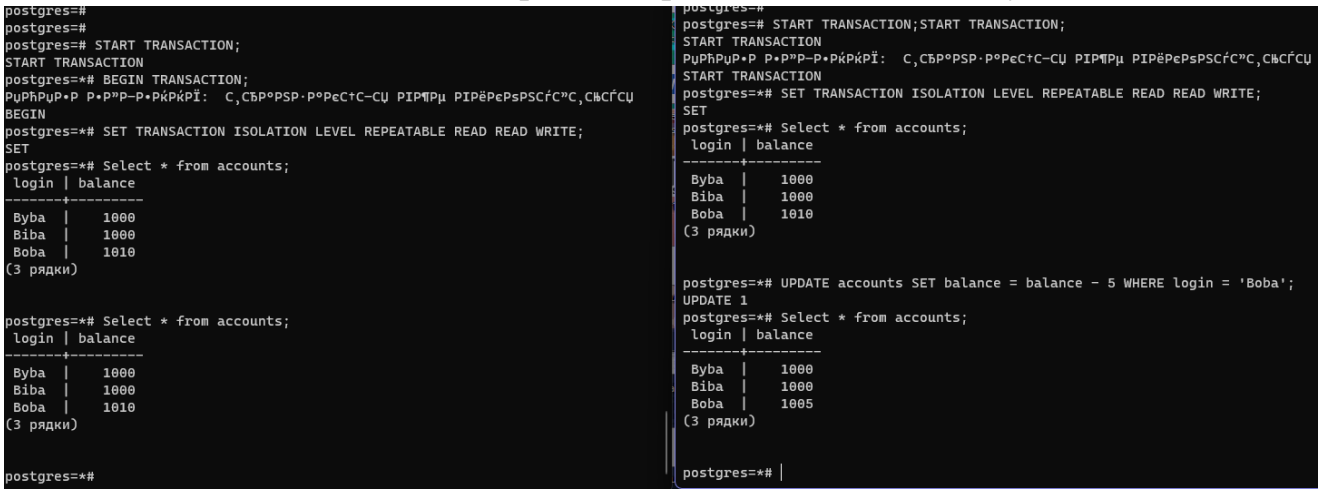




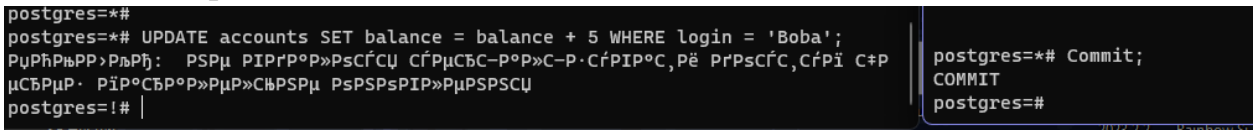
Repeatable Read

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.

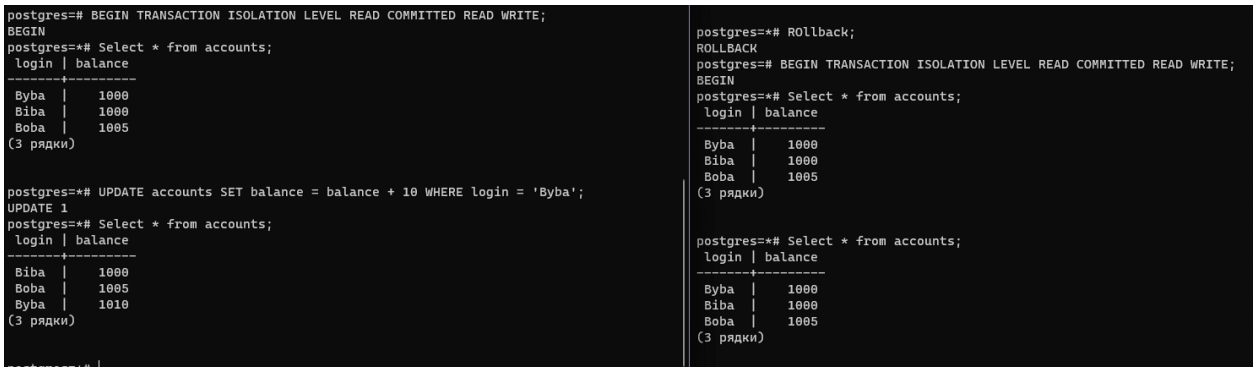


Відповідно транзакція може змінити данні тільки після закінчення іншої.



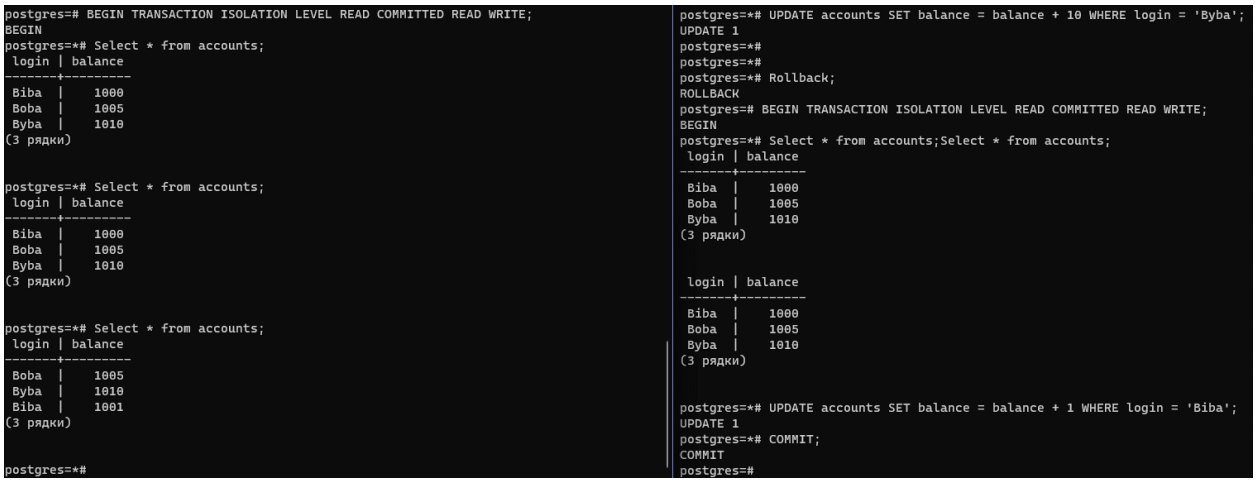
Read Committed

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.



```
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
postgres=*# Select * from accounts;
 login | balance
-------+---------
 Byba  |    1000
 Biba  |    1000
 Boba  |    1005
(3 рядки)


postgres=*# UPDATE accounts SET balance = balance + 10 WHERE login = 'Byba';
UPDATE 1
postgres=*# Select * from accounts;
 login | balance
-------+---------
 Biba  |    1000
 Boba  |    1005
 Byba  |    1010
(3 рядки)
```

```
postgres=*# ROllback;
ROLLBACK
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
postgres=*# Select * from accounts;
 login | balance
-------+---------
 Byba  |    1000
 Biba  |    1000
 Boba  |    1005
(3 рядки)


postgres=*# Select * from accounts;
 login | balance
-------+---------
 Byba  |    1000
 Biba  |    1000
 Boba  |    1005
(3 рядки)
```



```
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
postgres=*# Select * from accounts;
 login | balance
-------+---------
 Biba  |    1000
 Boba  |    1005
 Byba  |    1010
(3 рядки)


postgres=*# Select * from accounts;
 login | balance
-------+---------
 Biba  |    1000
 Boba  |    1005
 Byba  |    1010
(3 рядки)


postgres=*# Select * from accounts;
 login | balance
-------+---------
 Boba  |    1005
 Byba  |    1010
 Biba  |    1001
(3 рядки)


postgres=*#
```

```
postgres=*# UPDATE accounts SET balance = balance + 10 WHERE login = 'Byba';
UPDATE 1
postgres=*#
postgres=*#
postgres=*# Rollback;
ROLLBACK
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
postgres=*# Select * from accounts;Select * from accounts;
 login | balance
-------+---------
 Biba  |    1000
 Boba  |    1005
 Byba  |    1010
(3 рядки)


 login | balance
-------+---------
 Biba  |    1000
 Boba  |    1005
 Byba  |    1010
(3 рядки)


postgres=*# UPDATE accounts SET balance = balance + 1 WHERE login = 'Biba';
UPDATE 1
postgres=*# COMMIT;
COMMIT
postgres=#
```

Read Uncommitted

Нічого не відбувається в режимі read uncomitted. Тобто нічого не блокується і не створюються снепшоти, транзакція просто читає все що хоче. Не присутній в postgres.