**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**

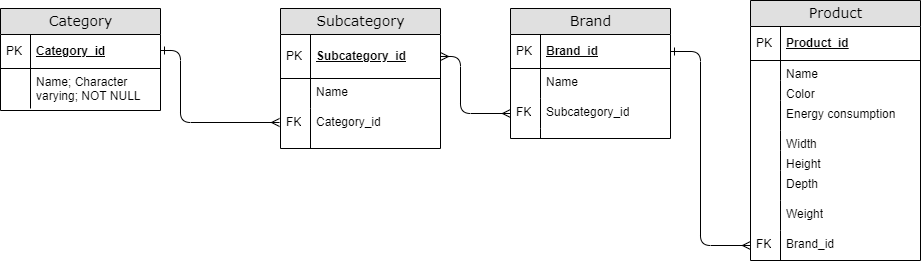


Посилання на телеграм та репозиторій:

<https://t.me/jemapel_sasuke_uchiwa>

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

**Відомості про обрану предметну галузь з лабораторної роботи №1**

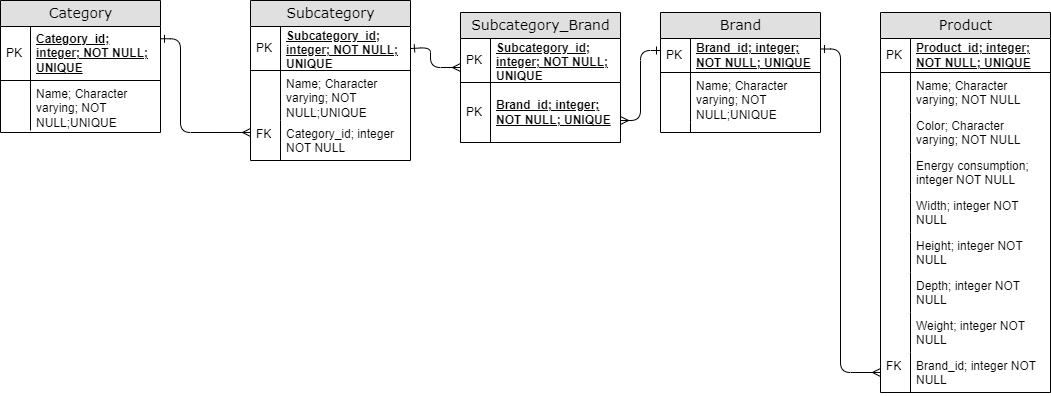


ER-діагарама побудована за нотацією «Crow`s foot».

**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 РГР 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)

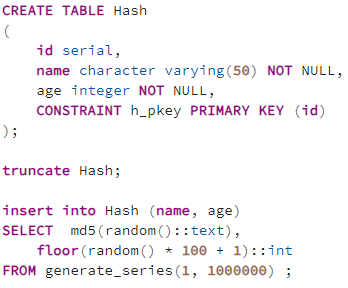
Програма працює ідентично розрахунково-графічній роботі.

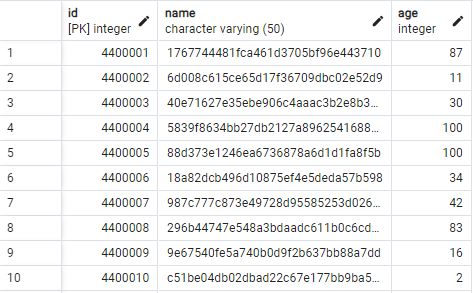
Приклад отримання всіх даних з таблиці SubCategory:

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

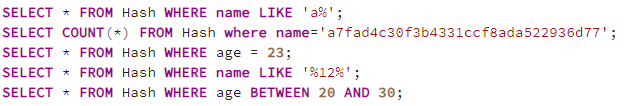
**Завдання №2**

**Hash-індекс**





Тестування на 5-х запитах:



Результати без індексу:

1:



2:



3:



4:



5:



Створимо індекс:



Результати з індексом:

1:



2:



3:



4:



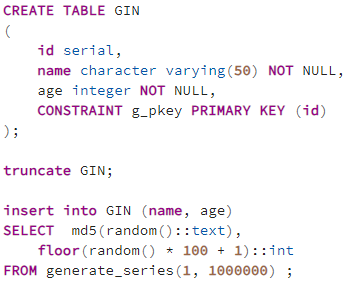
5:

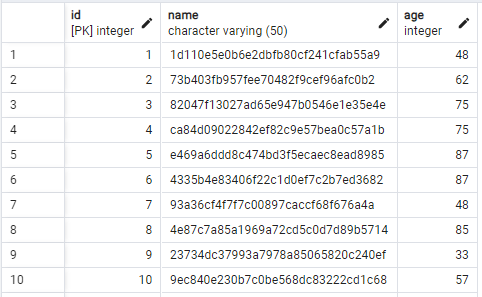


Індекс hash ефективний для простих порівнянь, і не підходить для діапазонів або не є сильно ефективним рішенням для використання функцій LIKE.

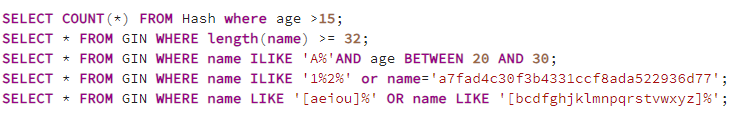
Оскільки шукаючи точні значення хеш-функція може швидко визначити, в якому сегменті таблиці знаходиться потрібний рядок. Це дозволяє уникнути необхідності сканувати всю таблицю, що значно прискорює пошук.

**Gin-індекс**





Тестування на 5-х запитах:



Результати без індексу:

1:



2:



3:



4:



5:



Створимо індекс:



Результати з індексом:

1:



2:



3:



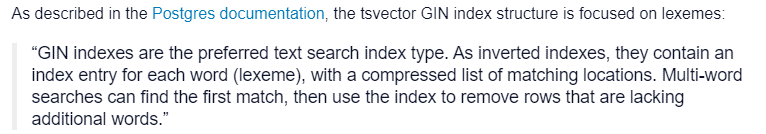
4:



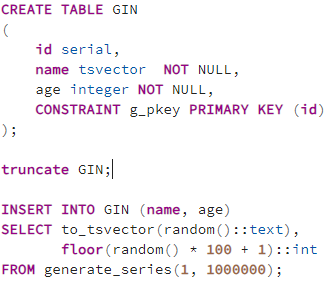
5:



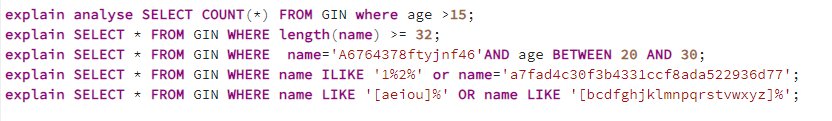
Індекс gin ,скоріше за все, виявляється неекфективним через невідповідність типів атрибутів таблиці.



Якщо змінити тип поля таблиці то результат стане краще:



Тестування на 5-х запитах:



Результати без індексу:

1:



2:



3:



Створимо індекс:



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

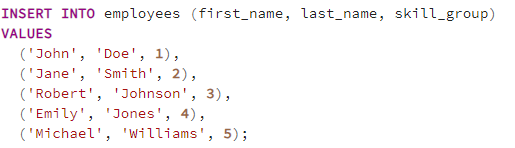
**FOR** **EACH** **ROW**

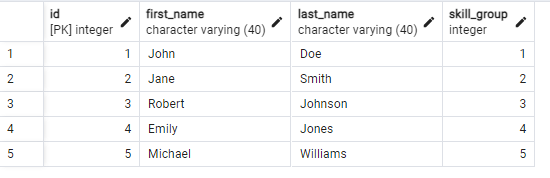
**EXECUTE** **FUNCTION** delete\_insert\_func();

Тригер працює наступним чином, після видалення робітника з рівнем навичок більше 2 він потрапляю до таблиці звільнених співробітників.

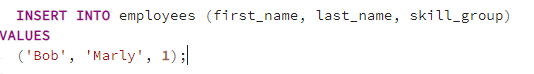
При вставці нових значень неможливо вставити співробтника у якого ім’я або прізвище коротше 2 символів, також придодаванні нового співробітника у старих підвищується рівень навичок. (не питайте чому)

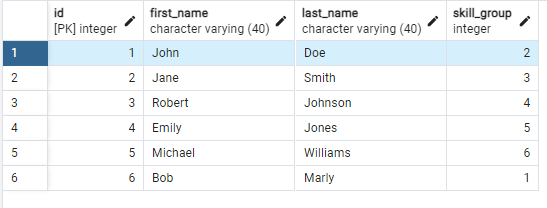
Додамо 5 рядків до employees:



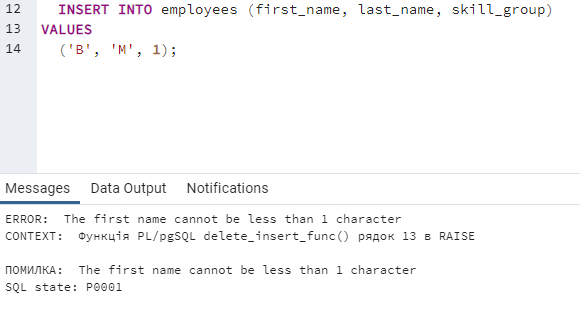


Тепер додамо 1 рядок:

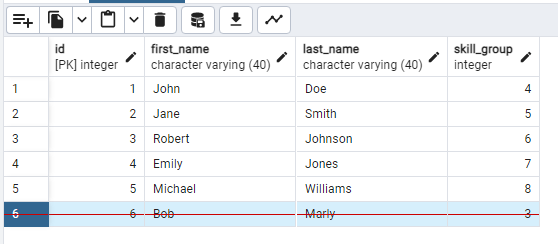


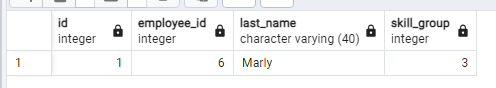


Неможливо вставити рядок з ім’ям/прізвищем у якого меньше 2 символів:



Видалимо Боба:





**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**

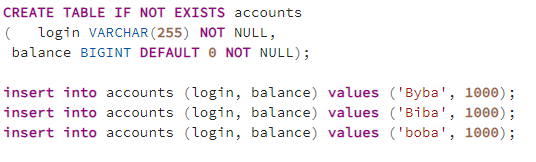
Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

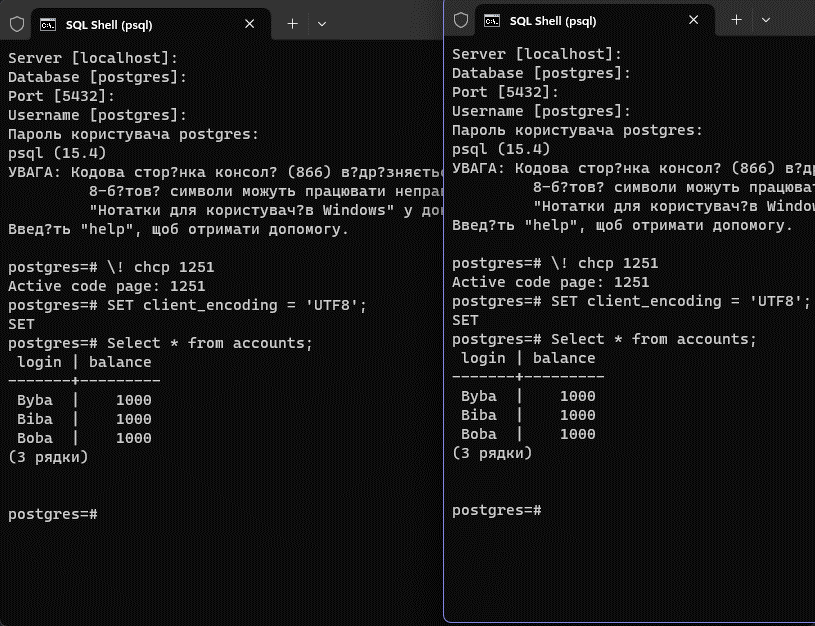
**3. Неповторюване читання**

Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

**4. Фантомне читання**

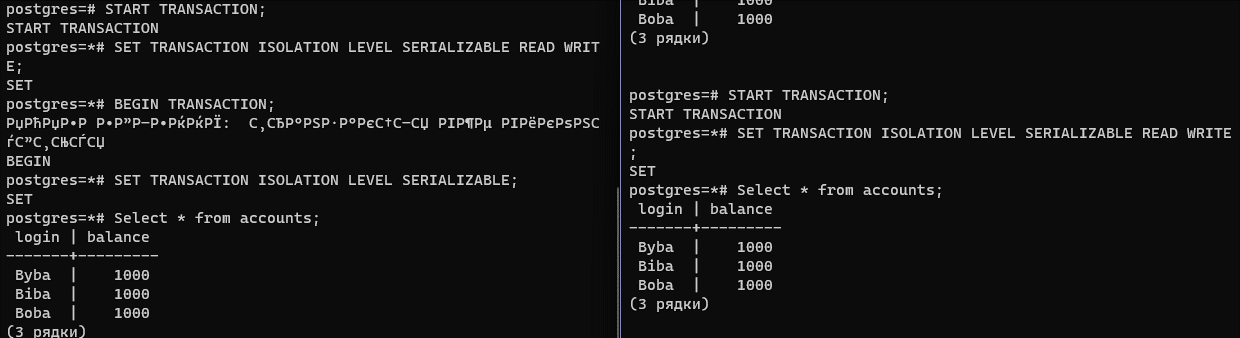
Ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.



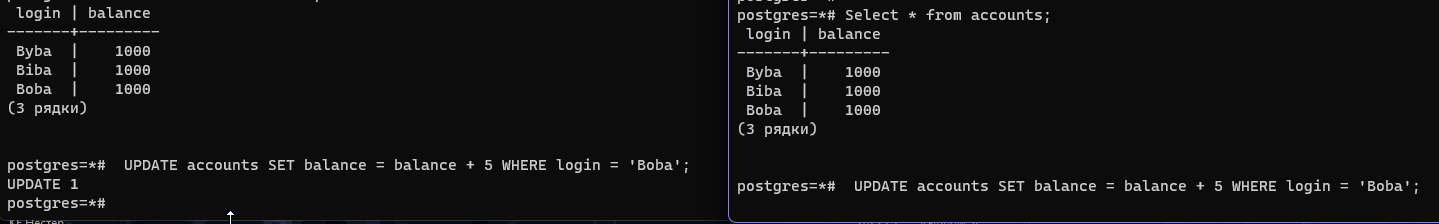


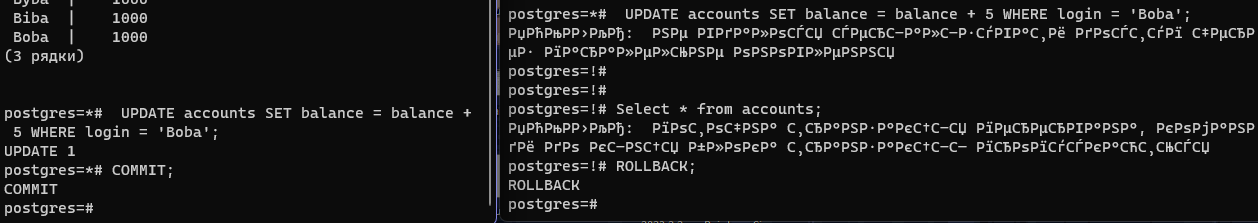
**Serializable**

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:

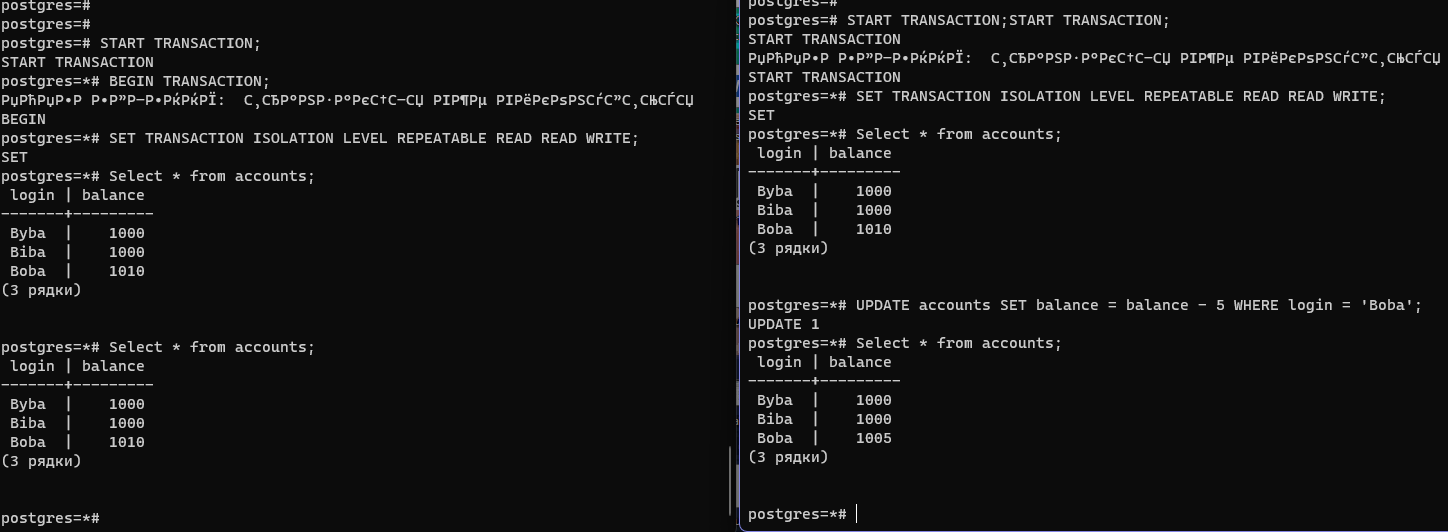




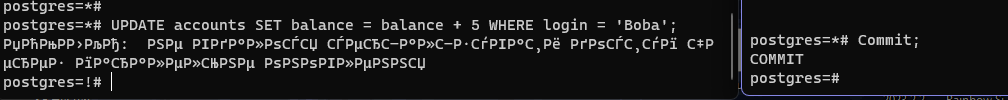
**Repeatable Read**

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.

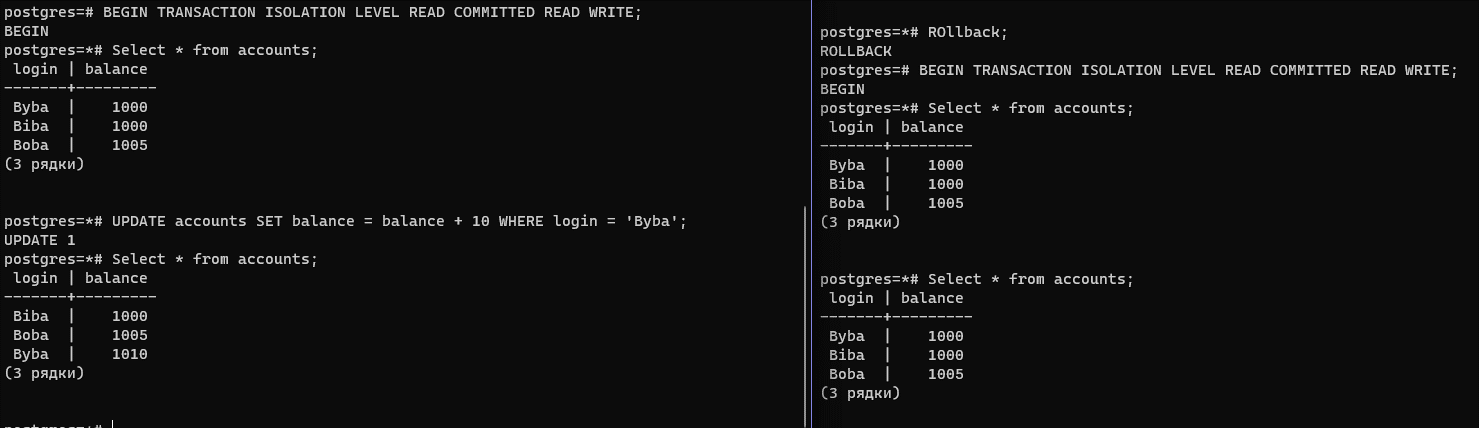


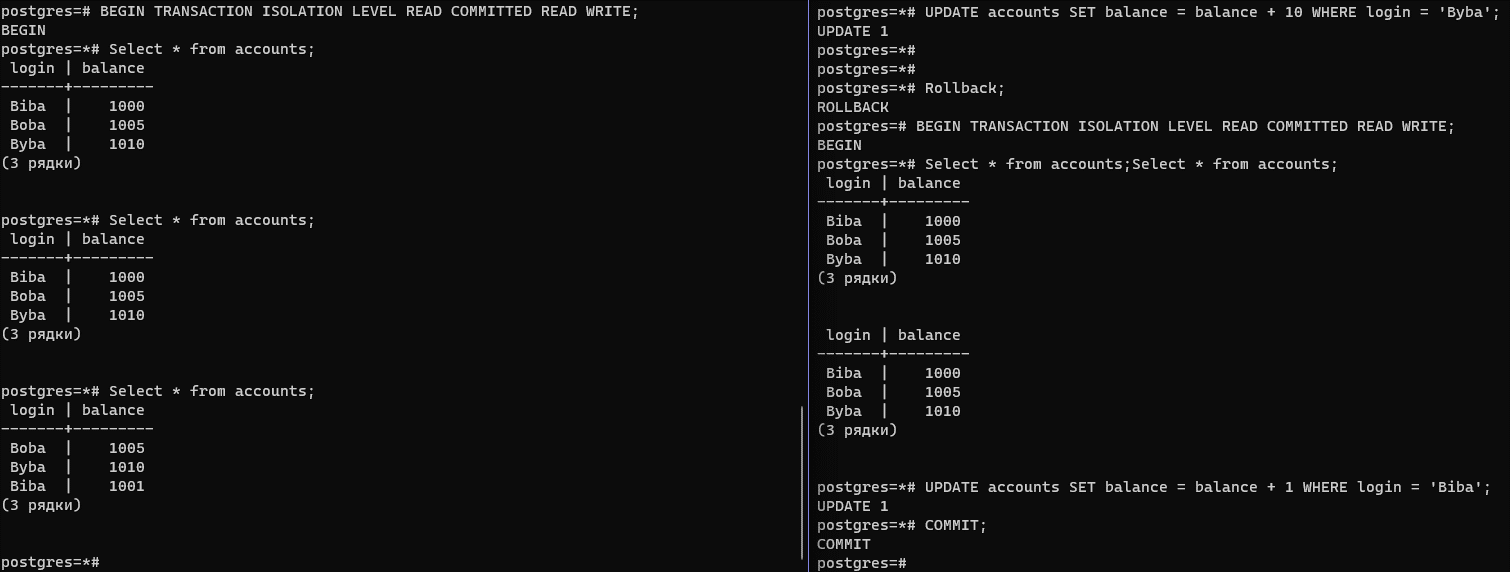
Відповідно транзакція може змінити данні тільки після закінчення іншої.



**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.





**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.