

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

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

### **Лабораторна робота № 3**

*з дисципліни*

**«Бази даних та засоби управління»**

**Тема: «Засоби оптимізації роботи СУБД PostgreSQL»**

Виконала студентка групи:

КВ-03 Віннікова У. В.

Перевірив: Петрашенко А. В.

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 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.

#### *Вимоги до пункту завдання №1*

Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlalchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:М, М:М та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

#### *Вимоги до пункту завдання №2*

Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5 прикладів запитів SELECT (із виведенням результатуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).

### *Вимоги до пункту завдання №3*

Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

### *Вимоги до пункту завдання №4*

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і способ їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у різних вікнах pgAdmin4 і виконати послідовність запитів INSERT, UPDATE або DELETE у обох транзакціях, що доводять наявність або відсутність певних феноменів.

### **Варіант №2**

<i>№ варіанта</i>	<i>Види індексів</i>	<i>Умови для тригера</i>
2	<i>Hash, BRIN</i>	<i>after insert, update</i>

URL репозиторію з вихідним кодом: <https://github.com/UlianaVinnikova/database.git>

Модель «сутність-зв'язок» предметної галузі «Магазин» («Shop»):

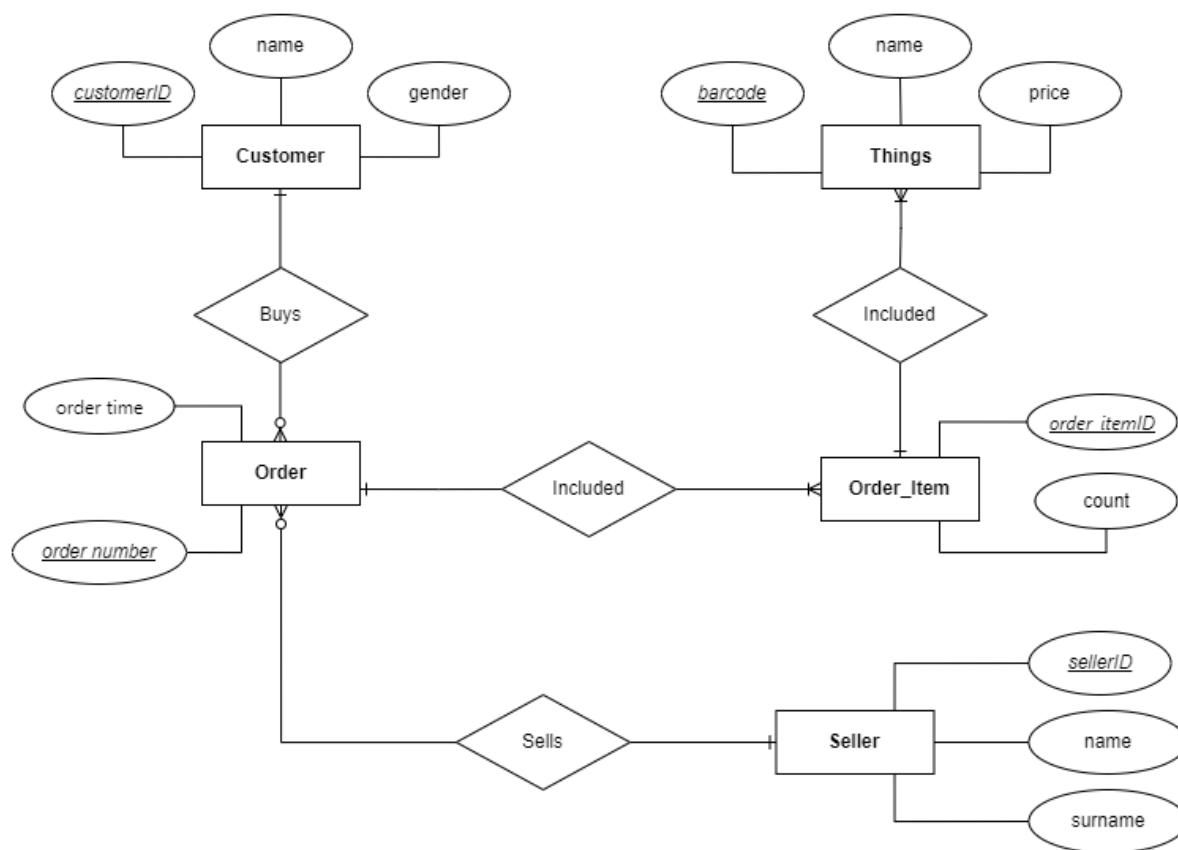


Рисунок 1. ER-діаграма побудована за нотацією «Crow`s foot»

### Сутності з описом призначення:

Предметна галузь «Shop» включає в себе 5 сутностей, кожна сутність містить декілька атрибутів:

1. Customer (customerID, name, gender).
2. Seller (sellerID, name, surname).
3. Things (barcode, name, price).
4. Order (order number, order time).
5. Order\_Item (order\_itemID, count).

Сутність Customer описує покупців, які завітали до даного магазину. Кожен покупець містить інформацію про свій ID, ім'я та стать.

Сутність Seller описує робітників, а точніше продавців магазину. Робітник має унікальний ID, ім'я та прізвище.

Сутність Things відповідає за речі, які продаються в магазині та входять в замовлення. У кожній речі є штрих-код, назва товару та ціна за товар.

Сутність Order це замовлення, яке може купити покупець та продати продавець. Замовлення має власний номер та час замовлення.

Сутність Order\_Item є залежною сутністю від Order. Order\_Item має ID та відповідає за кількість товару, який входить в замовлення.

Модель «сутність-зв'язок» у схемі бази даних PostgreSQL:

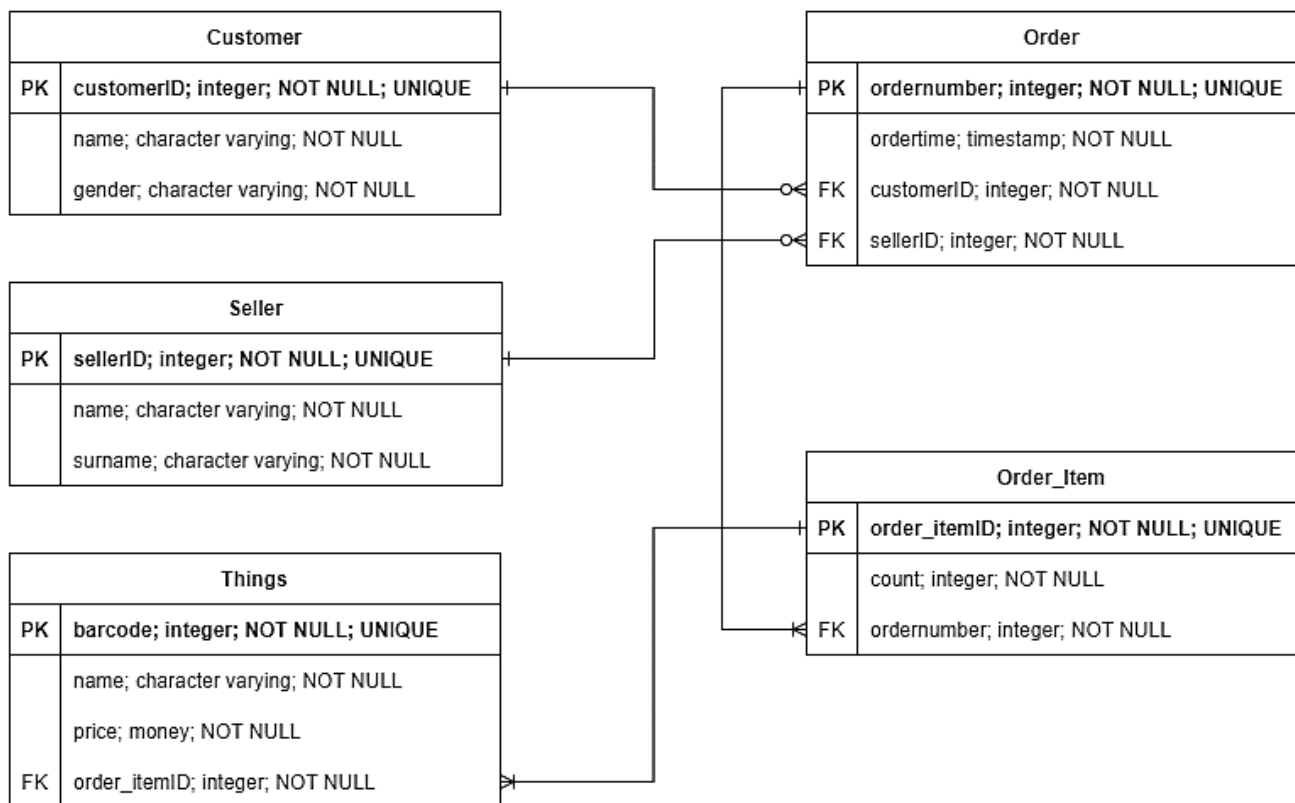


Рисунок 2. Схема бази даних у графічному вигляді

**Середовище для відлагодження SQL-запитів до бази даних – PgAdmin4.**

**Мова програмування – Python 3.10**

**Середовище розробки програмного забезпечення – PyCharm Community Edition.**

**Бібліотека взаємодії з PostgreSQL - Psycopg2, для реалізації моделі ORM використовувалася бібліотека SQLAlchemy.**

## Завдання 1

У даній лабораторній роботі було реалізовано 5 класів відповідно до 5 існуючих таблиць:

1. Customer;
2. Seller;
3. Things;
4. Order;
5. Order\_Item.

Таблиця customer має стовпчики: customerID (ідентифікатор), name (ім'я), gender (стать), а також зв'язок 1:N із таблицею order, тому в класі Customer встановлений зв'язок relationship ("Order").

Програмна реалізація класу Customer:

```
class Customer(Base):
    __tablename__ = 'customer'
    customerID = Column(Integer, primary_key=True)
    name = Column(String)
    gender = Column(String)

    order = relationship("Order")

    def __init__(self, customerID, name, gender):
        self.customerID = customerID
        self.name = name
        self.gender = gender

    def __repr__(self):
        return f"<Customer(customerID={self.customerID}, name={self.name}, gender={self.gender})>"
```

Таблиця seller має стовпчики: sellerID (ідентифікатор), name (ім'я), surname (прізвище), а також зв'язок 1:N із таблицею order, тому в класі Seller встановлений зв'язок relationship ("Order").

Програмна реалізація класу Seller:

```
class Seller(Base):
    __tablename__ = 'seller'
    sellerID = Column(Integer, primary_key=True)
    name = Column(String)
    surname = Column(String)

    order = relationship("Order")
```

```

def __init__(self, sellerID, name, surname):
    self.sellerID = sellerID
    self.name = name
    self.surname = surname

def __repr__(self):
    return f"<Seller(sellerID={self.sellerID}, name={self.name}, surname={self.surname})>"

```

Таблиця things має стовпчики: barcode (ідентифікатор), name (назва товару), price (ціна), order\_itemID (зовнішній ключ, який пов'язує товар з кількістю замовленого товару).

Програмна реалізація класу Things:

```

class Things(Base):
    __tablename__ = 'things'
    barcode = Column(Integer, primary_key=True)
    name = Column(String)
    price = Column(Numeric)

    order_itemID = Column(Integer, ForeignKey('order_item.order_itemID'))

    def __init__(self, barcode, name, price, order_itemID):
        self.barcode = barcode
        self.name = name
        self.price = price
        self.order_itemID = order_itemID

    def __repr__(self):
        return f"<Things(barcode={self.barcode}, name={self.name}, price={self.surname})>"

```

Таблиця order має стовпчики: ordernumber (ідентифікатор), ordertime (час в який було куплено/продано замовлення), customerID (зовнішній ключ, який пов'язує покупця з замовленням), sellerID (зовнішній ключ, який пов'язує продавця з замовленням). Зв'язок 1:N із таблицею order\_item, тому в класі Order встановлений зв'язок relationship ("Order\_Item").

Програмна реалізація класу Order:

```

class Order(Base):
    __tablename__ = 'order'
    ordernumber = Column(Integer, primary_key=True)

```

```

ordertime = Column(TIMESTAMP)

customerID = Column(Integer, ForeignKey('customer.customerID'))
sellerID = Column(Integer, ForeignKey('seller.sellerID'))

order_item = relationship("Order_Item")

def __init__(self, ordernumber, ordertime, customerID, sellerID):
    self.ordernumber = ordernumber
    self.ordertime = ordertime
    self.customerID = customerID
    self.sellerID = sellerID

def __repr__(self):
    return f"<Order(ordernumber={self.ordernumber},
ordertime={self.ordertime}, customerID={self.customerID},
sellerID={self.sellerID})>"

```

Таблиця `order_item` має стовпчики: `order_itemID` (ідентифікатор), `count` (кількість певного товару), `ordernumber` (зовнішній ключ, який пов'язує кількість даного товару який входить в замовлення). Також зв'язок 1:N із таблицею `things`, тому в класі `Order_Item` встановлений зв'язок `relationship` ('Things').

### Програмна реалізація класу `Order_Item`:

```

class Order_Item(Base):
    __tablename__ = 'order_item'
    order_itemID = Column(Integer, primary_key=True)
    count = Column(Integer)

    ordernumber = Column(Integer, ForeignKey('order.ordernumber'))

    things = relationship("Things")

    def __init__(self, order_itemID, count, ordernumber):
        self.order_itemID = order_itemID
        self.count = count
        self.ordernumber = ordernumber

    def __repr__(self):
        return f"<Order_Item(order_itemID={self.order_itemID},
count={self.count}, ordernumber={self.ordernumber})>"

```



## MENU

1. Show one table
2. Show all table
3. Insert data
4. Delete data
5. Update data
6. Exit

Головне меню для користувача складається з шести пунктів.

Перший пункт (1. Show one table) пропонує виведення однієї таблиці за вибором:

Choose an option: 1

- 1: customer
- 2: seller
- 3: things
- 4: order
- 5: order\_item

Choose the table number:

Перед виведенням даних, користувач обирає, яку саме таблицю потрібно вивести.

Після цього на екрані виводяться всі рядки і стовпчики з обраної таблиці БД.

Другий пункт (2. Show all table) це виведення всіх таблиць. Послідовно виводяться усі таблиці БД, після чого користувач знову повертається до головного меню і може обрати нову опцію для взаємодії з таблицями бази даних.

Третій пункт (3. Insert data) пропонує внесення даних:

Choose an option: 3

- 1: customer
- 2: seller
- 3: things
- 4: order
- 5: order\_item

Choose the table number:

Спочатку потрібно обрати, для якої таблиці буде відбуватися внесення, користувач вводить номер таблиці. Після цього користувач вводить дані для кожного атрибуту.

Четвертий пункт (4. Delete data) пропонує видалення даних:

```
Choose an option: 4
```

```
1: customer
```

```
2: seller
```

```
3: things
```

```
4: order
```

```
5: order_item
```

```
Choose the table number:
```

Спочатку потрібно обрати, для якої таблиці буде відбуватися видалення. Тому користувач вводить номер, що відповідає певній таблиці. Після цього користувач вводить ідентифікатор рядка, який потрібно видалити. Потім відбувається видалення даних відповідного рядка.

П'ятий пункт (5. Update data) пропонує редагування даних:

```
Choose an option: 5
```

```
1: customer
```

```
2: seller
```

```
3: things
```

```
4: order
```

```
5: order_item
```

```
Choose the table number:
```

Спочатку потрібно обрати, для якої таблиці буде відбуватися редагування. Тому користувач вводить номер, що відповідає певній таблиці. Після цього користувач обирає, які саме дані редагувати, вводить нові дані, які записуються в таблицю.

Восьмий пункт (6. Exit) пропонує вихід з програми. Закривається з'єднання і програма завершується.

```
Choose an option: 6
```

```
The program is over!
```

## Приклади запитів у вигляді ORM

**Запит вставки** реалізовано за допомогою функції `insert`. Спочатку у меню користувач обирає опцію вставки, далі обирає таблицю, до якої хоче додати запис і вводить необхідні дані.

Таблиця “seller” до вставки:

Data output   Messages   Notifications			
	sellerID [PK] integer	name character varying (20)	surname character varying (20)
1	1	Olga	Pavlenko
2	2	Iryna	Tkachenko
3	3	Mykola	Kuzmenko

Choose the table number: 2

Seller ID = 4

Seller name = *Natalia*

Seller surname = *Koval*

Таблиця “seller” після вставки:

	sellerID [PK] integer	name character varying (20)	surname character varying (20)
1	1	Olga	Pavlenko
2	2	Iryna	Tkachenko
3	3	Mykola	Kuzmenko
4	4	Natalia	Koval

Лістинг функцій `insert` для кожної таблиці:

```
@staticmethod
def insert_table1(customerID: int, name: str, gender: str) -> None:
    customer = Customer(customerID=customerID, name=name,
gender=gender)
    s.add(customer)
    s.commit()

@staticmethod
def insert_table2(sellerID: int, name: str, surname: str) -> None:
    seller = Customer(sellerID=sellerID, name=name, surname=surname)
```

```

s.add(seller)
s.commit()

@staticmethod
def insert_table3(barcode: int, name: str, price: float, order_itemID:
int) -> None:
    things = Things(barcode=barcode, name=name, price=price,
order_itemID=order_itemID)
    s.add(things)
    s.commit()

@staticmethod
def insert_table4(ordernumber: int, ordertime: datetime.datetime,
customerID, sellerID) -> None:
    order = Order(ordernumber=ordernumber, ordertime=ordertime,
customerID=customerID, sellerID=sellerID)
    s.add(order)
    s.commit()

@staticmethod
def insert_table5(order_itemID: int, count: int, ordernumber: int) ->
None:
    order_item = Order_Item(order_itemID=order_itemID, count=count,
ordernumber=ordernumber)
    s.add(order_item)
    s.commit()

```

**Запит видалення** реалізовано за допомогою функції delete. Спочатку користувач обирає таблицю, з якої потрібно видалити дані. Потім потрібно ввести номер ідентифікатора рядка для видалення.

Особливу увагу потрібно приділити зв'язкам між таблицями. Наприклад, таблиці order\_item та things пов'язані зв'язком 1:N через зовнішній ключ order\_itemID у таблиці things. Тому якщо користувач обирає запис про предмет, з яким є зв'язок у таблиці things, то відповідні записи видаляться в обох таблицях.

Таблиця “order\_item” до видалення:

	order_itemID [PK] integer	count integer	ordernumber integer
1	10	2	534
2	20	18	729

Таблиця “things” до видалення:

	barcode [PK] integer	name character varying (20)	price money	order_itemID integer
1	456	jacket	850,50 ?	10
2	789	boots	1 011,25 ?	20

Choose the table number: 5

Attribute to delete order\_itemID = 10

Таблиця “order\_item” після видалення:

	order_itemID [PK] integer	count integer	ordernumber integer
1	20	18	729

Таблиця “things” після видалення:

	barcode [PK] integer	name character varying (20)	price money	order_itemID integer
1	789	boots	1 011,25 ?	20

Лістинг функцій delete для кожної таблиці:

```
@staticmethod
def delete_table1(customerID) -> None:
    customer = s.query(Customer).filter_by(customerID=customerID).one()
    s.delete(customer)
    s.commit()

@staticmethod
def delete_table2(sellerID) -> None:
    seller = s.query(Seller).filter_by(sellerID=sellerID).one()
    s.delete(seller)
    s.commit()

@staticmethod
def delete_table3(barcode) -> None:
    things = s.query(Things).filter_by(barcode=barcode).one()
    s.delete(things)
    s.commit()

@staticmethod
def delete_table4(ordernumber) -> None:
    order = s.query(Order).filter_by(ordernumber=ordernumber).one()
    s.delete(order)
    s.commit()

@staticmethod
def delete_table5(order_itemID) -> None:
    order_item =
s.query(Order_Item).filter_by(order_itemID=order_itemID).one()
    s.delete(order_item)
    s.commit()
```

**Запит редагування** реалізовано за допомогою функції update. Спочатку користувач обирає, у якій таблиці потрібно змінити запис і за яким ідентифікатором. Також потрібно обрати атрибут, що редагується.

Таблиця “order” до редагування:

	ordernumber [PK] integer	ordertime timestamp without time zone	customerID integer	sellerID integer
1	534	2023-01-01 17:48:02	1	2
2	646	2022-10-13 12:34:47	3	1
3	729	2022-07-18 14:56:02	2	3

```
Choose the table number: 4
```

```
Attribute to update (where) ordernumber = 646
```

```
1: ordertime
```

```
Choose the number of attribute: 1
```

```
New value of attribute = 2023-01-14 15:09:45
```

Таблиця “order” після редагування:

	ordernumber [PK] integer	ordertime timestamp without time zone	customerID integer	sellerID integer
1	534	2023-01-01 17:48:02	1	2
2	646	2023-01-14 15:09:45	3	1
3	729	2022-07-18 14:56:02	2	3

Лістинг функцій update для кожної таблиці:

```
@staticmethod
    def update_table1(customerID: int, name: str, gender: str) -> None:

s.query(Customer).filter_by(customerID=customerID).update({Customer.name:
name, Customer.gender: gender})
s.commit()

    @staticmethod
    def update_table2(sellerID: int, name: str, surname: str) -> None:
        s.query(Seller).filter_by(sellerID=sellerID).update({Seller.name:
name, Seller.surname: surname})
        s.commit()

    @staticmethod
```

```

    def update_table3(barcode: int, name: str, price: float) -> None:
        s.query(Things).filter_by(barcode=barcode).update({Things.name:
name, Things.price: price})
        s.commit()

    @staticmethod
    def update_table4(ordernumber: int, ordertime: datetime.datetime) ->
None:

s.query(Order).filter_by(ordernumber=ordernumber).update({Order.ordertime:
ordertime})
        s.commit()

    @staticmethod
    def update_table5(order_itemID: int, count: int) -> None:

s.query(Order_Item).filter_by(order_itemID=order_itemID).update({Order_Item
.count: count})
        s.commit()

```

## Завдання 2

Індекс – це спеціальна структура даних, яка зберігає групу ключових значень та покажчиків. Індекс використовується для управління даними. Для тестування індексів було створено окремі таблиці у базі даних test з 1000000 записів.

### Hash

Для дослідження індексу була створена таблиця hash\_test, яка має дві колонки: “id” та “string”.

```
CREATE TABLE "hash_test"("id" bigserial PRIMARY KEY, "string" varchar(100));
INSERT INTO "hash_test"("id", "string")
SELECT generate_series as "id", md5(random()::text)
FROM generate_series(1, 1000000)
```

Data output			Messages	Notifications
	id [PK] bigint	string character varying (100)		
1	1	fb988bf3851ee006fa...		
2	2	fdc953fe515460b0985...		
3	3	45c5d106e03e7a1d24...		
4	4	89b126f0ac56cdd7b9...		
5	5	34a36d0136e6d7e0ad...		
6	6	b4ed1e41f7726fae2f0...		
7	7	fd9c03cdf9fb11247a29...		
8	8	cfc92ed1e2cd37bf870...		
9	9	f2205a615d5de95ada...		
10	10	2aa2b4b37a7d517280...		
11	11	10dfb072467c34b01e...		
12	12	9b09bd317b657b4336...		
13	13	02f2cb5c7147867d4d...		
14	14	7c984e59b59231b0a6...		
15	15	93b34edbd5bdb268de...		
Total rows: 1000 of 1000000			Query compl	

Тестування на чотирьох запитах:

```
SELECT COUNT(*) FROM "hash_test" WHERE "string" = 'b4ed1e41f7726fae2f02169e7cdaaed1';
SELECT AVG("id") FROM "hash_test" WHERE "string" LIKE 'b%' OR "string" = '9b09bd317b657b43360c96844324e6ad';
SELECT COUNT(*) FROM "hash_test" WHERE "string" LIKE 'f2%';
explain SELECT SUM("id") FROM "hash_test" WHERE "id" % 4 = 0 GROUP BY "string" LIKE '%ce1';
```







Створення індексу:

```
CREATE INDEX "hash_index" ON "hash_test" USING hash("text");  
CREATE INDEX
```





Query returned successfully in 5 secs 3 msec.

### Результати виконання запитів:

Без індекса hash:

- Запит 1:  Successfully run. Total query runtime: 86 msec. 1 rows affected.
- Запит 2:  Successfully run. Total query runtime: 215 msec. 1 rows affected.
- Запит 3:  Successfully run. Total query runtime: 227 msec. 1 rows affected.
- Запит 4:  Successfully run. Total query runtime: 250 msec. 2 rows affected.

З індексом hash:

- Запит 1:  Successfully run. Total query runtime: 57 msec. 1 rows affected.
- Запит 2:  Successfully run. Total query runtime: 203 msec. 1 rows affected.
- Запит 3:  Successfully run. Total query runtime: 171 msec. 1 rows affected.
- Запит 4:  Successfully run. Total query runtime: 264 msec. 2 rows affected.

З наведених даних можна побачити, що в 3 з 4 випадків виконання запитів за допомогою індексів виконується трохи швидше. Індекс hash має свої певні особливості від яких залежить час виконання конкретного запиту:

- не можна використовувати дані в індексі, щоб уникнути читання рядків;
- не можна використовувати для сортування, оскільки рядки у ньому не зберігаються у відсортованому порядку;
- hash-індекси не підтримують пошук за частковим ключем, так як hash-коди обчислюються для всього значення, що індексується;
- hash-індекси підтримують лише порівняння на рівність, що використовують оператори =, IN() та <=>;
- доступ до даних у хеш-індекс дуже швидкий, якщо немає великої кількості колізій
- деякі операції обслуговування індексу можуть виявитися повільними, якщо кількість колізій велика.

## ***BRIN***

BRIN розшифровується як індекс діапазону блоків (Block Range Index). BRIN призначений для обробки великих таблиць, у яких певні стовпці мають якусь природну кореляцію зі своїми фізичним розташуванням у таблиці.

Індекси BRIN відповідають на запити за допомогою звичайного сканування по бітовій карті, повертаючи всі кортежі всіх сторінок у кожному діапазоні, якщо зведена інформація, збережена в індексі, узгоджується з умовами запиту.

Виконавець запиту відповідає за перевірку цих кортежів і скидання тих, що не відповідають умовам запиту тобто, ці індекси неточні. Але оскільки індекс BRIN дуже малий, сканування індексу додає мало накладних витрат у порівнянні з послідовним скануванням, при цьому допомагаючи уникнути сканування великих областей таблиці, які безумовно не містять відповідних кортежів. Конкретні дані, які зберігатиме індекс BRIN, а також конкретні запити, які цей індекс зможе задовольнити, залежать від класу операторів, вибраного для кожного стовпця індексу. Типи даних з лінійним порядком сортування можуть мати класи операторів, які зберігають мінімальне та максимальне значення у кожному діапазоні блоків; геометричні типи можуть зберігати рамку для всіх об'єктів в діапазоні блоків.

Для дослідження індексу була створена таблиця brin\_test, яка має дві колонки: "id" та "string".

```
CREATE TABLE "brin_test"("id" bigserial PRIMARY KEY, "string" varchar(100));
INSERT INTO "brin_test"("id", "string")
SELECT generate_series as "id", md5(random()::text)
FROM generate_series(1, 1000000)
```

Data output

Messages

Notifications

≡+

📄

▼



📋

🗑️

🗄️

⬇️

📈

	id [PK] bigint 	string character varying (100) 
1	1	6ef9e0826d63548360...
2	2	030d86a0c5a3d1a2ea...
3	3	6a34f4624bc063cad8...
4	4	4a531873819c43d816...
5	5	f942ee430a5706482aa...
6	6	be6f50cdd193f8ca442...
7	7	4db261d522ef325deb...
8	8	a4f12076e9c2a56841f...
9	9	67f5ac0b3b7aa0570af...
10	10	737fdd18fef463ada14...
11	11	1736c01571736c0157...

Total rows: 1000 of 1000000

Query complete

Створення індексу:

```
CREATE INDEX "brin_index" ON "brin_test" USING brin("string");
```

```
CREATE INDEX
```

```
Query returned successfully in 1 secs 574 msec.
```

Результати виконання запитів:

Без індекса brin:

Запит 1:

✓ Successfully run. Total query runtime: 208 msec. 1 rows affected.

Запит 2:

✓ Successfully run. Total query runtime: 67 msec. 1 rows affected.

Запит 3:

✓ Successfully run. Total query runtime: 198 msec. 1 rows affected.

Запит 4:

✓ Successfully run. Total query runtime: 194 msec. 8695 rows affected.

З індексом brin:

Запит 1:

✓ Successfully run. Total query runtime: 158 msec. 1 rows affected.

Запит 2:

✓ Successfully run. Total query runtime: 46 msec. 1 rows affected.

Запит 3:

✓ Successfully run. Total query runtime: 154 msec. 1 rows affected.

Запит 4:

✓ Successfully run. Total query runtime: 182 msec. 8695 rows affected.

З наведених даних можна побачити, що виконання запитів за допомогою індексів виконується на декілька msec швидше, ніж без індексів.

### Завдання 3

Умови для тригера: after insert, update.

Для тестування тригера було створено дві таблиці в базі даних test: таблиця trigger\_test з атрибутами trigger\_testID (ідентифікатор) trigger\_testName (ім'я), trigger\_test\_log з атрибутами id (ідентифікатор), trigger\_test\_log\_ID (зовнішній ключ для зв'язку з таблицею trigger\_test), trigger\_test\_log\_name (ім'я).

Тригер спрацьовує після операції вставки (after insert) та під час операції редагування (update). Серед усіх записів таблиці trigger\_test у курсорному циклі 20 обираються ті, що мають ідентифікатори кратні 2. Якщо цей ідентифікатор також кратний 3, то висвічується повідомлення, що число ділиться на 2 і 3. Також якщо ідентифікатор кратний 2 і 3, то в таблицю trigger\_test\_log вставляються рядки з цими ідентифікаторами та відповідними іменами. В іншому випадку (якщо число не ділиться на 3, але ділиться на 2), викликається повідомлення - «Число парне» і в таблицю trigger\_test\_log вставляються рядки з цими ідентифікаторами та відповідними іменами. Далі з атрибуту trigger\_test\_log\_name видаляються набори символів 'log'. Якщо число не ділиться на 2, то висвічується повідомлення «Число непарне» і виконується редагування в курсорному циклі: для всіх записів таблиці trigger\_test\_log, що мають в назві сполучення букв '\_id' потрібно замінити ім'я на '\_' та trigger\_test\_log\_name та '\_log'. Тригер спрацьовує, якщо викликати операцію вставки (insert) або редагування (update).

Нижче на скріншотах продемонстровано коректну роботу тригера.

Створення таблиць:

```
CREATE TABLE "trigger_test"(  
    "trigger_testID" bigserial PRIMARY KEY,  
    "trigger_testName" text  
);
```

```
CREATE TABLE "trigger_test_log"(  
    "id" bigserial PRIMARY KEY,  
    "trigger_test_log_ID" bigint,  
    "trigger_test_log_name" text  
);
```

Data output	Messages	Notifications
<div> <div>≡+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div>		
	trigger_testID [PK] bigint	trigger_testName text

Data output

Messages

Notifications

≡+

📄

▼

📋

🗑️

🗄️

⬇️

📈

id

[PK] bigint

✎

trigger\_test\_log\_ID

bigint

✎

trigger\_test\_log\_name

text

✎

Створення тригера:

```
CREATE OR REPLACE FUNCTION after_insert_func() RETURNS TRIGGER AS $trigger$
DECLARE
CURSOR_LOG CURSOR FOR SELECT * FROM "trigger_test_log";
row_ "trigger_test_log" % ROWTYPE;
BEGIN
IF NEW."trigger_testID" % 2 = 0 THEN
IF NEW."trigger_testID" % 3 = 0 THEN
RAISE NOTICE 'trigger_testID is multiple of 2 and 3';
FOR row_ IN CURSOR_LOG LOOP
-- UPDATE "trigger_test_log" SET "trigger_test_log_name"='_' || row_."trigger_test_log_name" || '_log' WHERE "id"=row_."id";
INSERT INTO "trigger_test_log"("trigger_test_log_ID", "trigger_test_log_name")
VALUES (NEW."trigger_testID", NEW."trigger_testName");
END LOOP;
RETURN NEW;
ELSE
RAISE NOTICE 'trigger_testID is even';
INSERT INTO "trigger_test_log"("trigger_test_log_ID", "trigger_test_log_name")
VALUES (NEW."trigger_testID", NEW."trigger_testName");
UPDATE "trigger_test_log" SET "trigger_test_log_name" = trim(BOTH '_log' FROM "trigger_test_log_name");
RETURN NEW;
END IF;
ELSE
RAISE NOTICE 'trigger_testID is odd';
FOR row_ IN CURSOR_LOG LOOP
UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' || row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
END LOOP;
RETURN NEW;
END IF;
END;
$trigger$ LANGUAGE plpgsql;
CREATE TRIGGER after_insert_test
AFTER INSERT OR UPDATE ON "trigger_test"
FOR EACH ROW EXECUTE PROCEDURE after_insert_func();
```

CREATE TRIGGER

Query returned successfully in 59 msec.

Команди, що ініціюють виконання тригера:

```
CREATE TRIGGER "after_insert_update_trigger"
AFTER INSERT OR UPDATE ON "trigger_test"
FOR EACH ROW
EXECUTE procedure after_insert_func();
```

Початковий вміст таблиці trigger\_test було задано запитом:

```
INSERT INTO trigger_test("trigger_testName")
VALUES ('test1'), ('test2'), ('test3'), ('test4'), ('test5'), ('test6');
```

Data output Messages Notifications			
	trigger_testID [PK] bigint		trigger_testName text
1	1		test1
2	2		test2
3	3		test3
4	4		test4
5	5		test5
6	6		test6

Data output Messages Notifications			
	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	_test2_log
2	2	4	_test4_log
3	3	6	test6
4	4	6	test6

```
UPDATE "trigger_test" SET "trigger_testName" = "trigger_testName" || '_2' WHERE "trigger_testID" % 2 = 0
```

Data output Messages Notifications

	trigger_testID [PK] bigint	trigger_testName text
1	1	test1
2	2	test2_2
3	3	test3
4	4	test4_2
5	5	test5
6	6	test6_2

Data output Messages Notifications

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	test2
2	2	4	test4
3	3	6	test6
4	4	6	test6
5	5	2	test2_2
6	6	4	test4_2
7	7	6	test6_2
8	8	6	test6_2
9	9	6	test6_2
10	10	6	test6_2
11	11	6	test6_2
12	12	6	test6_2



## Завдання 4

Для цього завдання знадобилася окрема таблиця “transactions” з атрибутами id (ідентифікатор), number (число), string (текст). Також було додано три записи за допомогою запиту вставки insert.

```
CREATE TABLE "transactions"(  
    "id" bigserial PRIMARY KEY,  
    "number" bigint,  
    "string" text  
);  
  
INSERT INTO "transactions"("number", "string")  
VALUES (111, 'strstr1'), (122, 'strstr2'), (133, 'strstr3');
```

### Read committed (читання фіксованих даних)

Прийнятий за замовчуванням рівень для Microsoft SQL Server. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT).

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.

Dashboard Properties SQL Statistics Dependencies Dependents **test/postgres@PostgreSQL 14\*** test/postgres...

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 START TRANSACTION;  
2  
3 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
4
```

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres...

**test/postgres@PostgreSQL 14\***

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 SELECT * FROM "transactions";
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	1	111	strsr1
2	2	122	strsr2
3	3	133	strsr3

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres...

**test/postgres@PostgreSQL 14\***

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 UPDATE "transactions"  
2 SET "number" = 200, "string" = 'strsrstrsr'  
3 WHERE "id" = 2
```

Data output Messages Notifications

UPDATE 1

Query returned successfully in 74 msec.

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

1 **SELECT** \* **FROM** "transactions";

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	1	111	strsr1
2	3	133	strsr3
3	2	200	strsrtsr

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres... test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

Query Query History

1 **SELECT** \* **FROM** "transactions";

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	1	111	strsr1
2	3	133	strsr3
3	2	200	strsrtsr

## Феномен «фантомного чтения»

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

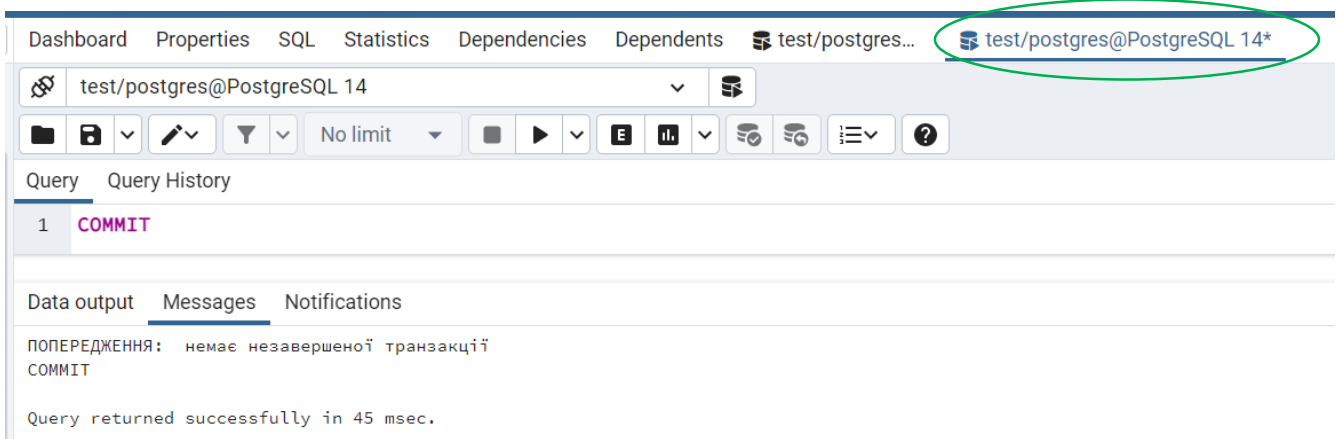
Query Query History

1 **COMMIT**

Data output Messages Notifications

ПОПЕРЕДЖЕННЯ: немає незавершеної транзакції  
COMMIT

Query returned successfully in 64 msec.

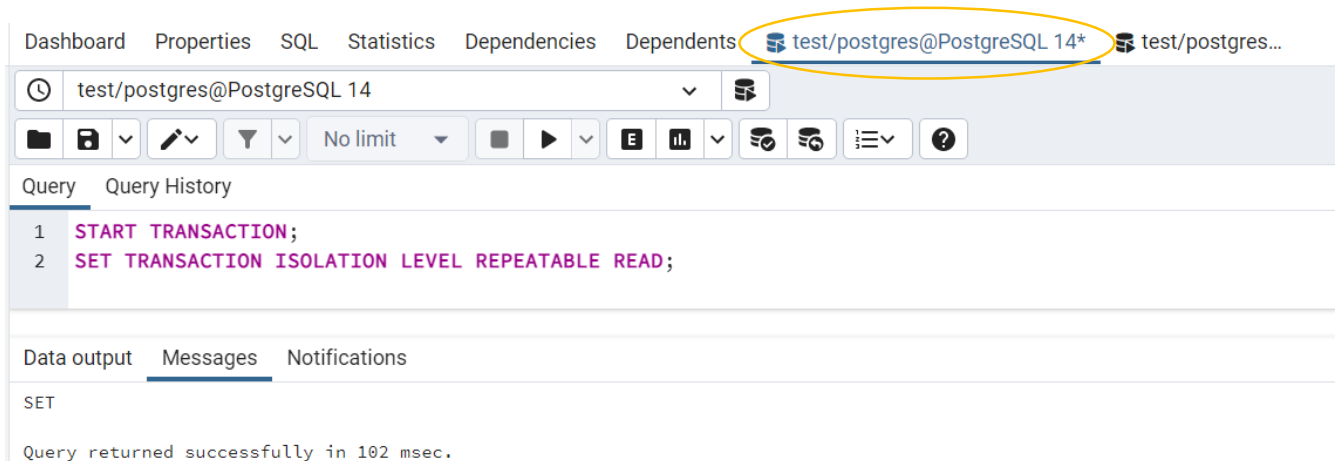


Після команди COMMIT бачимо, що зміни були внесені і збережені в обох транзакціях.

### 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).

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.



Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

```
1 SELECT * FROM "transactions"
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	1	111	strsr1
2	3	133	strsr3
3	2	200	strsrtsr

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

```
1 UPDATE "transactions"
2 SET "number" = "number" - 21
3 WHERE "id" = 1;
4 COMMIT
```

Data output Messages Notifications

ПОПЕРЕДЖЕННЯ: немає незавершеної транзакції  
COMMIT

Query returned successfully in 73 msec.

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres... test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

Query Query History

```
1 SELECT * FROM "transactions"
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	3	133	strsr3
2	2	200	strsrtsr
3	1	90	strsr1

Якщо спробувати в другій транзакції виконати запит редагування того самого рядка і відняти від numeric 10, то нам висвітиться помилка через паралельні зміни в транзакціях. Це є перевагою repeatable read.

Дослідимо аномалію «серіалізації». На рівні ізоляції repeatable read запустимо дві транзакції. У першій виведемо всі рядки і порахуємо суму стовпчика numeric у всіх записах. Додаємо запис із цим значенням в таблицю. Якщо у другій транзакції повторити ті ж самі операції, то стан таблиці на початку ще не змінений, сума буде такою ж, як у першій транзакції. Таким чином, ми додамо до таблиці такий самий рядок, як і першій транзакції. Виконуючи commit в обох транзакціях, ми побачимо два однакових записи в таблиці. Це і є феномен «серіалізації», що пояснюється серійним виконанням двох транзакцій однієї за одною, причому порядок виконання транзакції неважливий.

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

```
1 SELECT * FROM "transactions"
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	3	133	strstr3
2	2	200	strstrtsr
3	1	90	strsr1

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

```
1 SELECT SUM("number") FROM "transactions"
```

Data output Messages Notifications

	sum numeric
1	423

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

Query Query History

```

1 INSERT INTO "transactions"("number", "string")
2 VALUES (423, 'strstr423')

```

Data output Messages Notifications

INSERT 0 1

Query returned successfully in 45 msec.

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres... test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

Query Query History

```

1 SELECT * FROM "transactions"

```

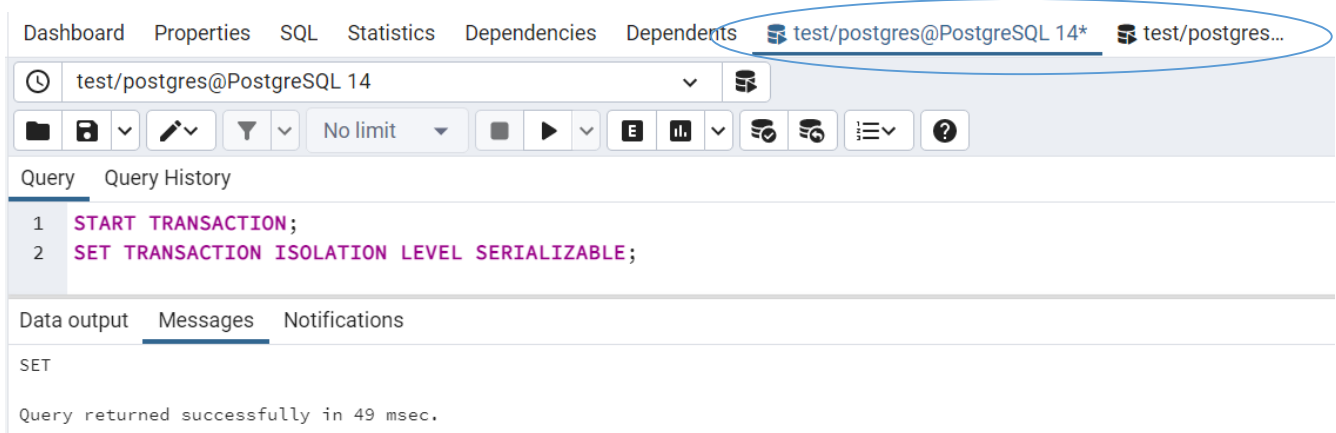
Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	3	133	strstr3
2	2	200	strstrstr
3	1	90	strstr1
4	4	423	strstr423
5	5	423	strstr423

## 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.

Запустимо дві транзакції на рівні Serializable. Спочатку стан таблиці однаковий. У першій транзакції видалимо рядок з `id = 5`. Якщо у другій транзакції спробувати зробити ті ж операції, то ми повинні будемо очікувати, доки перша транзакція не завершиться. Коли команда `commit` у першій транзакції виконана, у другій виникає помилка через паралельне видалення. Це неможливо, оскільки якщо запис уже видалений в першій транзакції, то видалити рядок з неіснуючим ідентифікатором неможливо. Для зупинення та відміни змін використаємо команду `rollback`, і після цього бачимо, що зміни внесені і в другу транзакцію.



test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

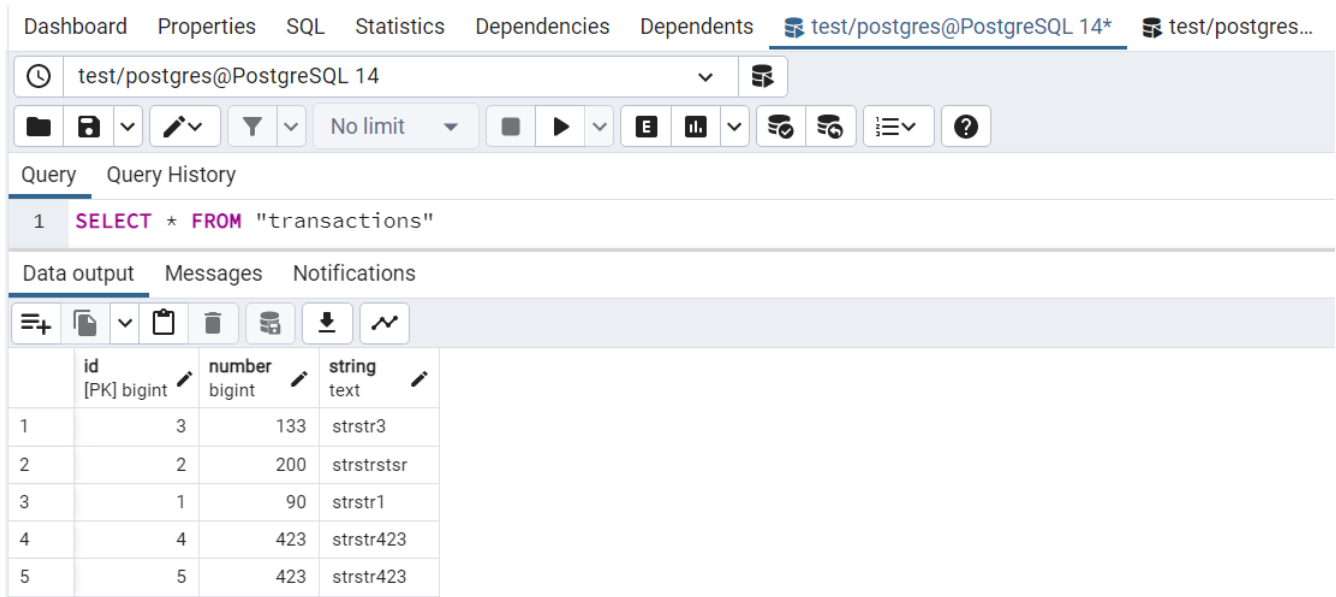
Query

```
1 START TRANSACTION;  
2 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Data output Messages Notifications

SET

Query returned successfully in 49 msec.



test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

Query

```
1 SELECT * FROM "transactions"
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	3	133	strsr3
2	2	200	strsrstr
3	1	90	strsr1
4	4	423	strsr423
5	5	423	strsr423



Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 DELETE FROM "transactions" WHERE "id" = 5
```

Data output Messages Notifications

DELETE 1

Query returned successfully in 47 msec.

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres... test/postgres@PostgreSQL 14\*

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 DELETE FROM "transactions" WHERE "id" = 5
```

Data output Messages Notifications

	id [PK] bigint	number bigint	string text
1	3	133	strstr3
2	2	200	strstrstr
3	1	90	strstr1
4	4	423	strstr423
5	5	423	strstr423

Waiting for the query to complete...

Dashboard Properties SQL Statistics Dependencies Dependents test/postgres@PostgreSQL 14\* test/postgres...

test/postgres@PostgreSQL 14

No limit

Query Query History

```
1 COMMIT
```

Data output Messages Notifications

COMMIT

Query returned successfully in 49 msec.

test/postgres@PostgreSQL 14



No limit



E



Query Query History

```
1 ROLLBACK
2 SELECT * FROM "transactions"
```

Data output Messages Notifications



	id [PK] bigint	number bigint	string text
1	3	133	strsr3
2	2	200	strsrtsr
3	1	90	strsr1
4	4	423	strsr423