**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 і управління”

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

Виконав студент групи КВ-03

Статечний Сергій

***Репозиторій:***

<https://github.com/Code01KPI/Lab3>

Київ – 2022

**Лабораторна робота № 3.**

**Засоби оптимізації роботи СУБД PostgreSQL**

*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.

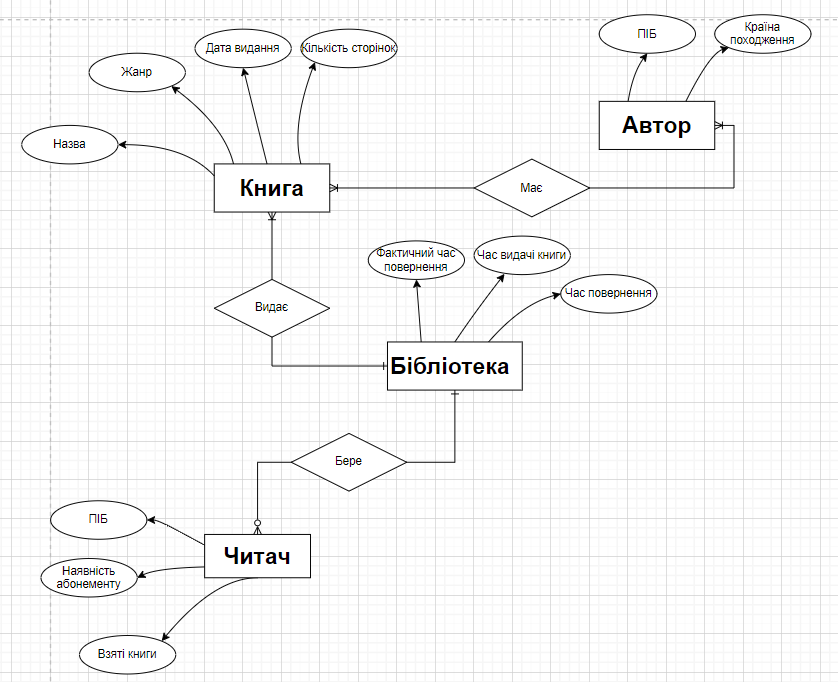
**Варіант – 21**





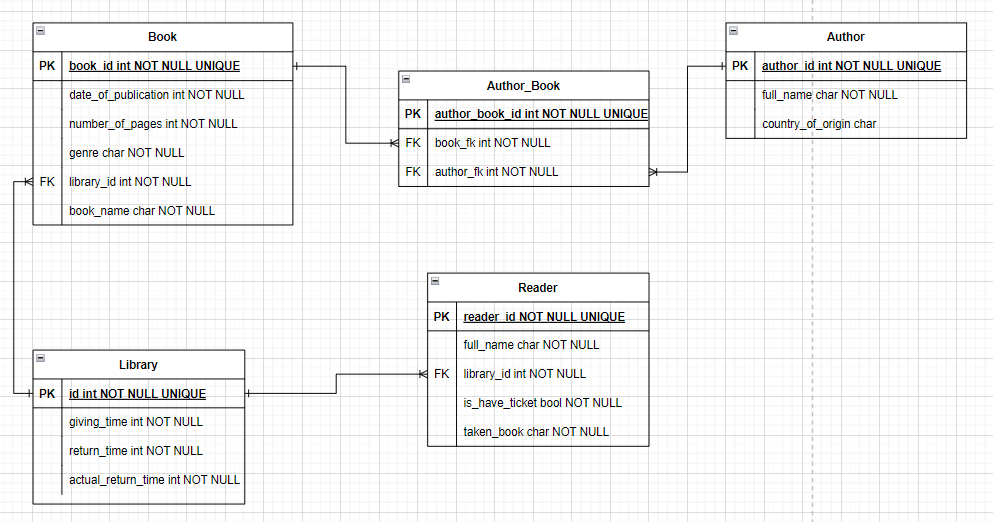
**Завдання 1**

*ER діаграма*



*Нотація “Пташина лапка”*

*Структура бази даних*



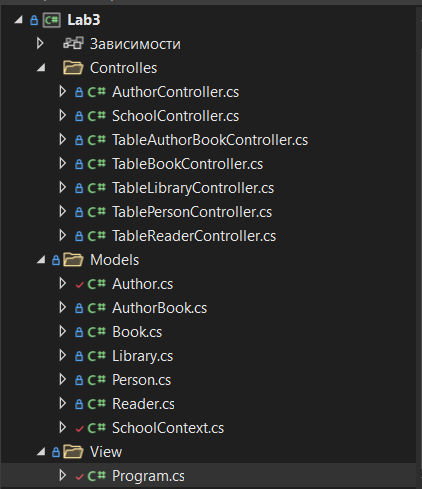
ER модель – бібліотека(видача книг читачам)

*Опис сутностей*

1. Автор – сутність описує автора певної книги. Серед атрибутів має ПІБ(виступає в ролі id) та країну походження .
2. Книга – сутність описує певну книгу. Атрибутами є назва книги(виступає в ролі id), а також жанр, дату видання і кількість сторінок.
3. Бібліотека – сутність описує бібліотеку. Атрибутами є час видачі книги, очікуваний час повернення та фактичний час повернення книги.
4. Читач – сутність описує певного читача. Атрибутами є ПІБ(виступає в ролі id), наявність абонементу та перелік взятих книг.

*Мова програмування та використані бібліотеки*

Програма реалізована на мові C#, з використанням Entity Framework.



Структура програми не зазнала змін. Модуль Models так само має 6 класів для 6 різних таблиці. SchoolContext – клас контексту даних, назва включає назву БД. Зв’язок між таблицями реалізований з допомогою зовнішніх ключів та навігаційних properties(виділив жовтим кольором).

**Клас *Author***

public partial class Author

{

public int AuthorId { get; set; }

public string FullName { get; set; } = null!;

public string CountryOfOrigin { get; set; } = null!;

public Author() { }

public Author (int id, string fullName, string countryOfOrigin)

{

AuthorId = id;

FullName = fullName;

CountryOfOrigin = countryOfOrigin;

}

public virtual ICollection<AuthorBook> AuthorBooks { get; } = new List<AuthorBook>();

}

**Клас *AuthorBook***

public partial class AuthorBook

{

public int AuthorBookId { get; set; }

public int BookFk { get; set; }

public int AuthorFk { get; set; }

public AuthorBook() { }

public AuthorBook(int id, int bFK, int aFK)

{

AuthorBookId = id;

BookFk = bFK;

AuthorFk = aFK;

}

public virtual Author AuthorFkNavigation { get; set; } = null!;

public virtual Book BookFkNavigation { get; set; } = null!;

}

**Клас *Book***

public partial class Book

{

public int BookId { get; set; }

public int? DateOfPublication { get; set; }

public int NumberOfPages { get; set; }

public string Genre { get; set; } = null!;

public int? BkLibraryId { get; set; }

public string BookName { get; set; } = null!;

public Book() { }

public Book(int id, int publicationDate, int pages, string genre, int? libId, string bookName)

{

BookId = id;

DateOfPublication = publicationDate;

NumberOfPages = pages;

Genre = genre;

BkLibraryId = libId;

BookName = bookName;

}

public virtual ICollection<AuthorBook> AuthorBooks { get; } = new List<AuthorBook>();

public virtual Library? BkLibrary { get; set; }

}

**Клас *Library***

public partial class Library

{

public int Id { get; set; }

public DateOnly GivingTime { get; set; }

public DateOnly ReturnTime { get; set; }

public DateOnly ActualReturnTime { get; set; }

public Library () { }

public Library (int id, DateOnly givingTime, DateOnly returnTime, DateOnly actualReturnTime)

{

Id = id;

GivingTime = givingTime;

ReturnTime = returnTime;

ActualReturnTime = actualReturnTime;

}

public virtual ICollection<Book> Books { get; } = new List<Book>();

public virtual ICollection<Reader> Readers { get; } = new List<Reader>();

}

**Клас *Reader***

public partial class Reader

{

public int Id { get; set; }

public int LibraryId { get; set; }

public int PersonId { get; set; }

public string TakenBook { get; set; } = null!;

public Reader() { }

public Reader(int id, int libId, int perId, string takenBook)

{

Id = id;

LibraryId = libId;

PersonId = perId;

TakenBook = takenBook;

}

public virtual Library Library { get; set; } = null!;

public virtual Person Person { get; set; } = null!;

}

**Клас *Person***

public partial class Person

{

public int PersonId { get; set; }

public string FullName { get; set; } = null!;

public bool IsHaveTicket { get; set; }

public Person() { }

public Person(int id, string fullName, bool isHaveTicket)

{

PersonId = id;

FullName = fullName;

IsHaveTicket = isHaveTicket;

}

public virtual ICollection<Reader> Readers { get; } = new List<Reader>();

}

**Приклад ORM запитів**

**Код реалізації запитів з допомогою ORM для таблиці *Author***

public override async Task InsertDataAsync()

{

if (author is not null)

{

using (schoolContext = new SchoolContext())

{

await schoolContext.AddAsync(author);

await schoolContext.SaveChangesAsync();

}

}

else

throw new ArgumentException("Author object is not set!", nameof(author));

}

public override async Task UpdateDataAsync()

{

if (author is not null)

{

using (schoolContext = new SchoolContext())

{

schoolContext.Authors.Update(author);

await schoolContext.SaveChangesAsync();

}

}

else

throw new ArgumentException("Author object is not set!", nameof(author));

}

public override async Task DeleteDataAsync(int id)

{

using (schoolContext = new SchoolContext())

{

var aBList = schoolContext.AuthorBooks.ToList().Where(ab => ab.AuthorFk == id);

schoolContext.AuthorBooks.RemoveRange(aBList);

await schoolContext.SaveChangesAsync();

schoolContext.Authors.Remove(schoolContext.Authors.First(a => a.AuthorId == id));

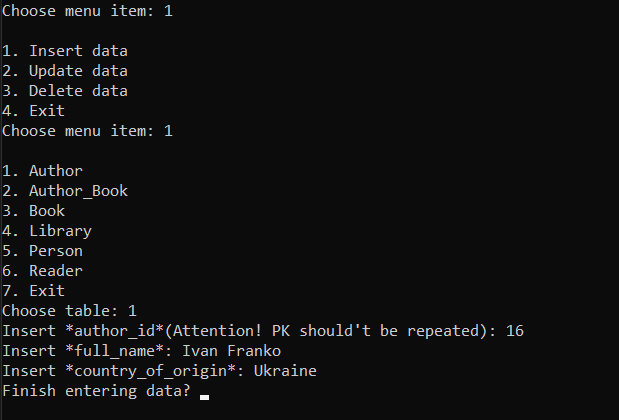
await schoolContext.SaveChangesAsync();

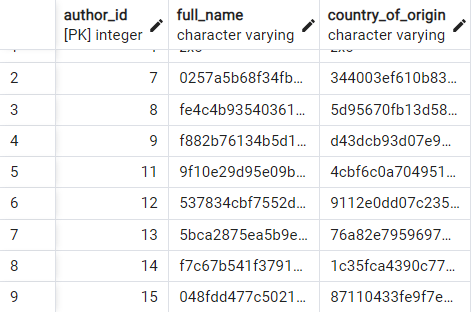
}

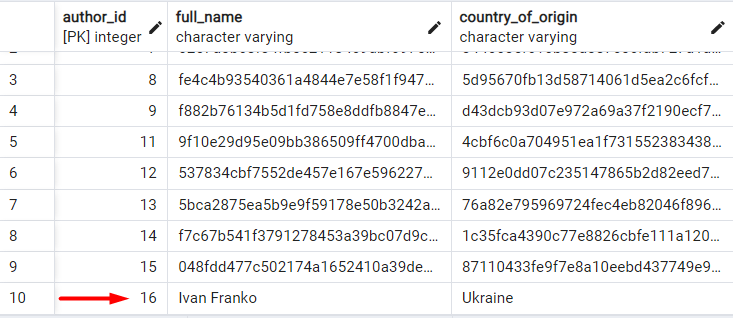
}

**Приклади запитів**

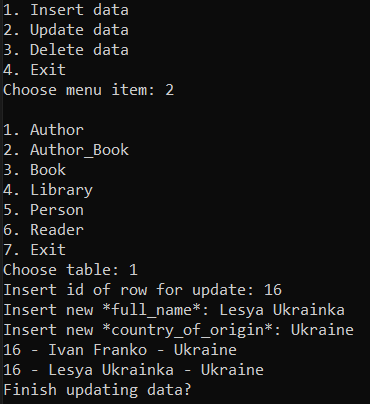
Вставка даних в таблицю *Author*

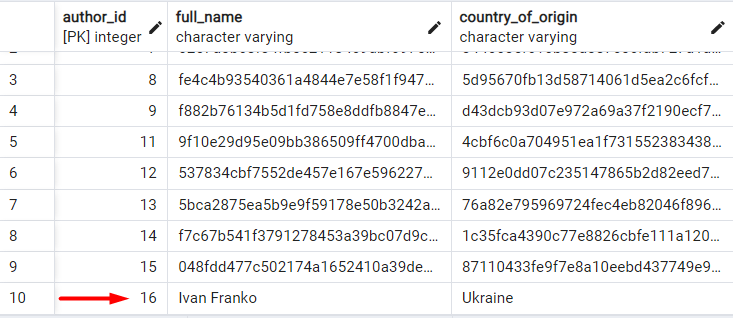


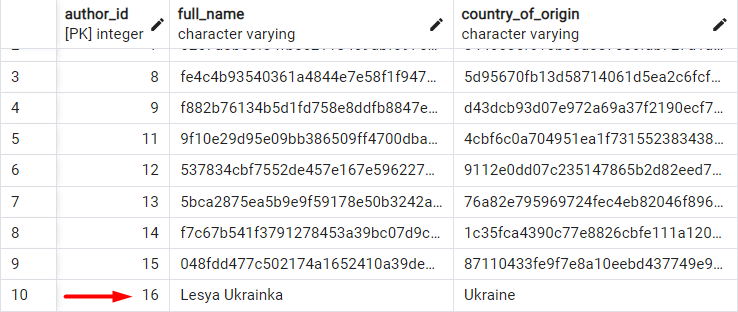




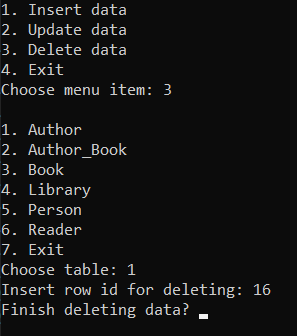
Редагування даних таблиці *Author*

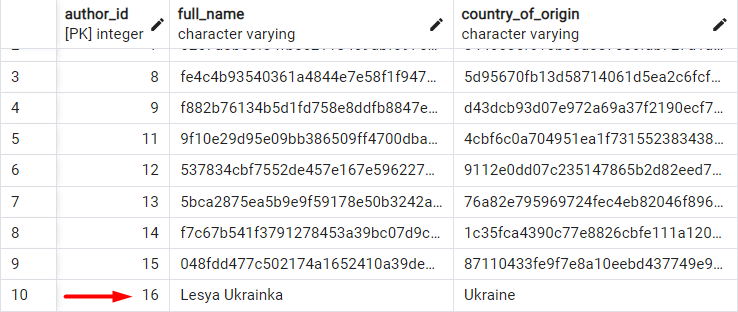


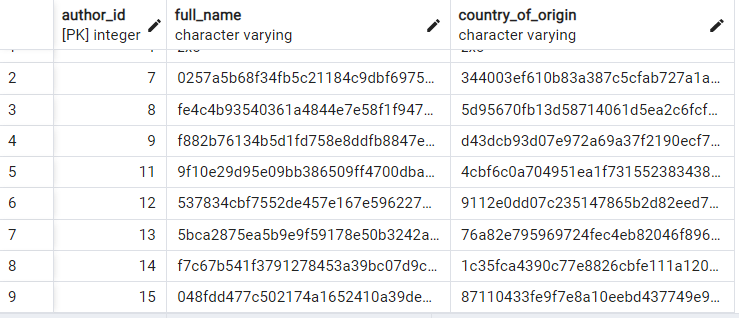




Вилучення даних з таблиці *Author*







**Завдання 2**

**BTree**

Індекс BTree призначений для даних, які можна відсортувати. Іншими словами, для типу даних мають бути визначені оператори «більше», «більше або дорівнює», «менше», «менше або дорівнює» та «дорівнює». Пошук починається з кореня вузла, і потрібно визначити, по якому з дочірніх вузлів спускатися. Знаючи ключи в корені, можна зрозуміти діапазони значень в дочірніх вузлах. Процедура повторюється до тих пір, поки не буде знайдено вузол, з якого можна отримати необхідні дані.

Для тестування даного індексу була створена додаткова таблиця *btree\_test,* в яку було записано 100000 рядків рандомізованих даних.

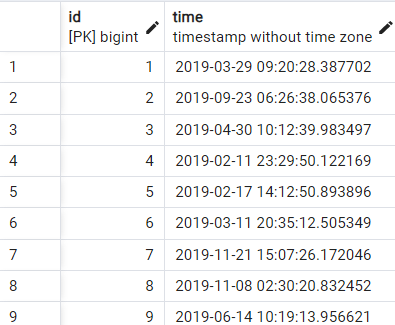
*SQL-запити створення та внесення даних в таблицю:*

CREATE TABLE "btree\_test"(id bigserial PRIMARY KEY, time timestamp);

INSERT INTO "btree\_test"(id, time)

SELECT generate\_series(1, 10000), timestamp '2019-01-01 00:00:00' + random() \* (timestamp '2021-01-01 00:00:00' - timestamp '2020-01-02

00:00:00');



*SQL-запити з використанням агрегатних функцій та групування:*

1 – explain analyze SELECT \* FROM "btree\_test" WHERE "id" % 5 = 0;

2 - explain analyze SELECT COUNT(\*) FROM "btree\_test" WHERE "id" % 5 = 0 AND "time" > '20190916';

3 – explain analyze SELECT AVG(id) FROM "btree\_test" WHERE "time" > '20190303' AND time < '20190909'

4 - explain analyze SELECT SUM(id), MIN(id) FROM "btree\_test" WHERE "time" > '20190101' AND "time"

<= '20191231' GROUP BY id % 2;

*Створення індексу:*

CREATE INDEX "btree\_time\_index" ON "btree\_test" ("id");

Час виконання запитів без та з індексом

|  |  |  |
| --- | --- | --- |
| Час | Без індексу | З індексом |
| Запит 1 | 0.849 ms | 0.856 ms |
| Запит 2 | 0.818 ms | 0.809 ms |
| Запит 3 | 3.889 ms | 1.509 ms |
| Запит 4 | 3.904 ms | 3.544 ms |

**Висновки**

*B-tree* індекс рекомендовано використовувати для операцій порівняння. Аналізуючи отримані результати часу запитів можна дійти висновку, що в загальному випадку використання даного індексу покращує показники. Це пов’язано з тим, що *B-tree* виходить неглибоким навіть для великих таблиць – через велику кількість сторінок(гілок).

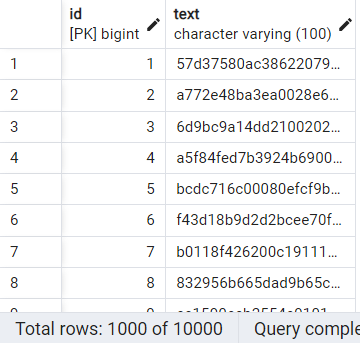
**Hash**

Хеш-індекс базується на принципі хеш функції. При пошуці в індексі ми обчислюємо хеш-функцію для ключа і отримуємо номер корзини в якій знаходяться потрібні дані. Після чого перебираються всі дані корзини і вибираються потрібні.

*SQL-запити створення та внесення даних в таблицю:*

CREATE TABLE "Hash\_test"(id bigserial PRIMARY KEY, text varchar(100));

INSERT INTO "Hash\_test"(id, text) SELECT generate\_series(1, 10000), md5(random()::text);



*SQL-запити з використанням фільтрації, агрегатних функцій та групування:*

1 - explain analyze SELECT \* FROM "Hash\_test" WHERE text = '57d37580ac386220796f5d486f771b91';

2 - explain analyze SELECT COUNT(\*) FROM "Hash\_test" WHERE text LIKE 'a7%';

3 - explain analyze SELECT AVG(id), SUM(id) FROM "Hash\_test" WHERE text LIKE 'b%' OR text = 'a5f84fed7b3924b690095e54ad315eca';

4 - explain analyze SELECT AVG(id), SUM(id) FROM "Hash\_test" WHERE id % 2 = 0 GROUP BY text LIKE 'd%';

*Створення індексу:*

CREATE INDEX "hash\_text\_index" ON "Hash\_test" USING hash("text");

Час виконання запитів без та з індексом

|  |  |  |
| --- | --- | --- |
| Час | Без індексу | З індексом |
| Запит 1 | 1.777 ms | 1.024 ms |
| Запит 2 | 2.295 ms | 2.189 ms |
| Запит 3 | 1.449 ms | 1.554 ms |
| Запит 4 | 5.681 ms | 2.991 ms |

**Висновки**

*Hash-*індекс показав хороші результати в цілому, особливо для операції порівняння =. Варто зазначити, що даний індекс не варто застосовувати для операцій перевірки інтервалів. Також потрібно враховувати можливість колізій – тому краще застосовувати хеш-індекс для унікальних даних.

**Завдання 3**

Для тестування тригера було творено дві додаткові таблиці, *trigger\_test(id, first\_name, age)* та *trigger\_log(id, test\_id, old\_first\_name, old\_age, changed\_on, is\_updated).* Тригер повинен спрацьовувати при командах видалення/редагування даних для першої таблиці – і записувати видалені або дані до редагування в другу таблицю. Стовпець *is\_updated* в таблиці *trigger\_log* зберігає логічний стан, true – над першою таблицею було здійсненне редагування даних, false – було здійснене видалення даних.

*SQL-запити для створення та заповнення даними таблиць:*

CREATE TABLE trigger\_test(

id INT GENERATED ALWAYS AS IDENTITY,

first\_name VARCHAR(20) NOT NULL,

age INT NOT NULL

);

CREATE TABLE trigger\_log(

id INT GENERATED ALWAYS AS IDENTITY,

test\_id INT NOT NULL,

old\_first\_name VARCHAR(20) NOT NULL,

old\_age INT NOT NULL,

changed\_on TIMESTAMP(6) NOT NULL,

is\_updated BOOLEAN NOT NULL

);

INSERT INTO trigger\_test(first\_name, age) VALUES

('Andrew', 20), ('Anton', 30), ('Serhii', 40);

*Тригер:*

CREATE OR REPLACE FUNCTION trigger\_func()

RETURNS TRIGGER AS $$

BEGIN

IF (tg\_op = 'DELETE') THEN

IF OLD.first\_name = 'Andrew' THEN

RAISE EXCEPTION 'Exc delete';

ELSEIF OLD.first\_name <> 'Andrew' THEN

INSERT INTO trigger\_log(test\_id, old\_first\_name, old\_age, changed\_on, is\_updated)

VALUES(OLD.id, OLD.first\_name, OLD.age, now(), 'FALSE'::BOOLEAN);

RETURN OLD;

END IF;

END IF;

IF(tg\_op = 'UPDATE') THEN

IF OLD.first\_name = 'Andrew' THEN

RAISE EXCEPTION 'Exc update';

ELSEIF OLD.first\_name <> 'Andrew' THEN

INSERT INTO trigger\_log(test\_id, old\_first\_name, old\_age, changed\_on, is\_updated)

VALUES(OLD.id, OLD.first\_name, OLD.age, now(), 'TRUE'::BOOLEAN);

END IF;

RETURN NEW;

END IF;

END;

$$ LANGUAGE PLPGSQL;

CREATE TRIGGER trigger1

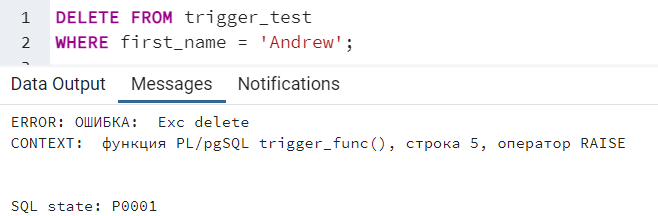
BEFORE DELETE OR UPDATE

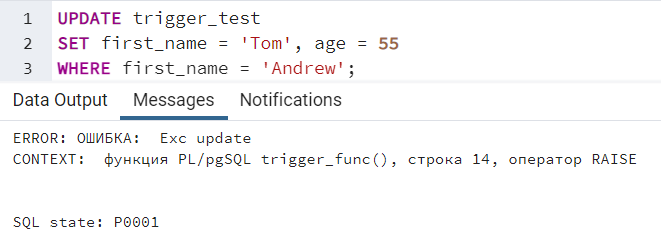
on trigger\_test

FOR EACH ROW EXECUTE PROCEDURE trigger\_func();

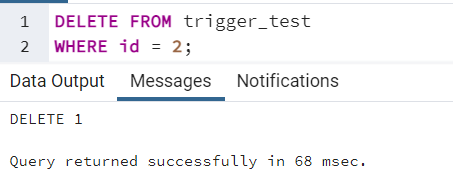
*Перевірка роботи тригера:*

1. Спроба видалення/редагування забороненого рядка(перевірка роботи обробки виключних ситуацій)

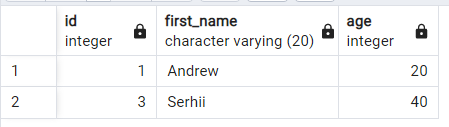




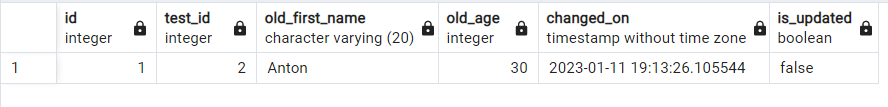
1. Видалення з таблиці:



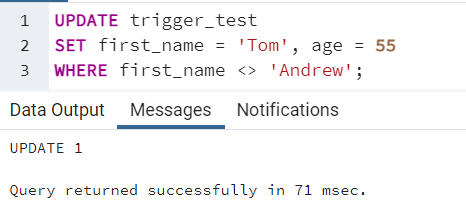
*trigger\_test:*



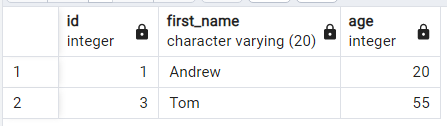
*trigger\_log:*



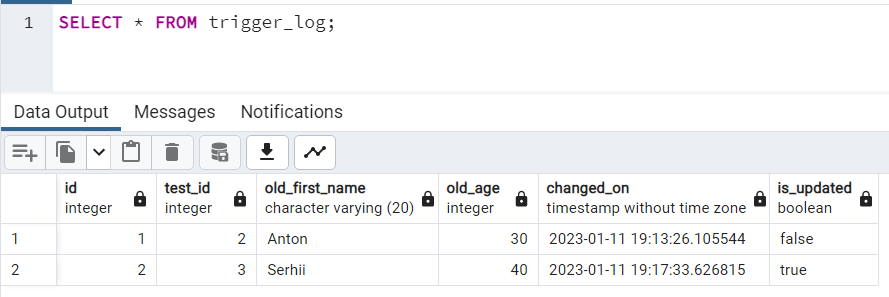
1. Редагування:



*trigger\_test:*



*trigger\_log:*



**Завдання 4**

В рамках даного завдання була створенна таблиця *transactions(int id, int number, varchar(100) text).* Також в неї були додані дані.

*SQL-запит створення та внесення даних в таблицю transactions:*

CREATE TABLE transactions(

id INT GENERATED ALWAYS AS IDENTITY,

numb INT NOT NULL,

text VARCHAR(100) NOT NULL

);

INSERT INTO transactions(numb, text) VALUES

(12, 'zxczxvzxc'),

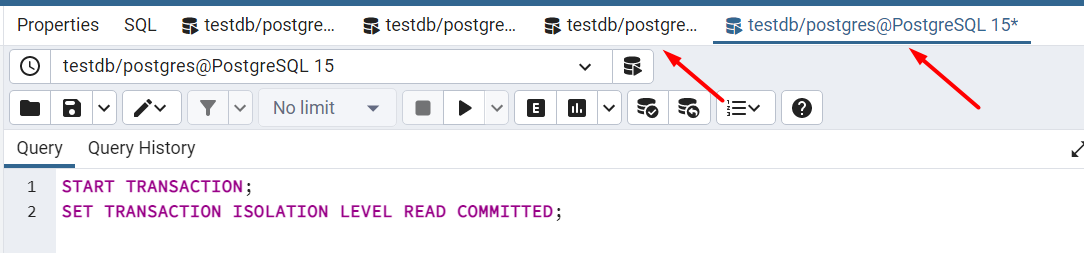
(23, 'fadasdasda'),

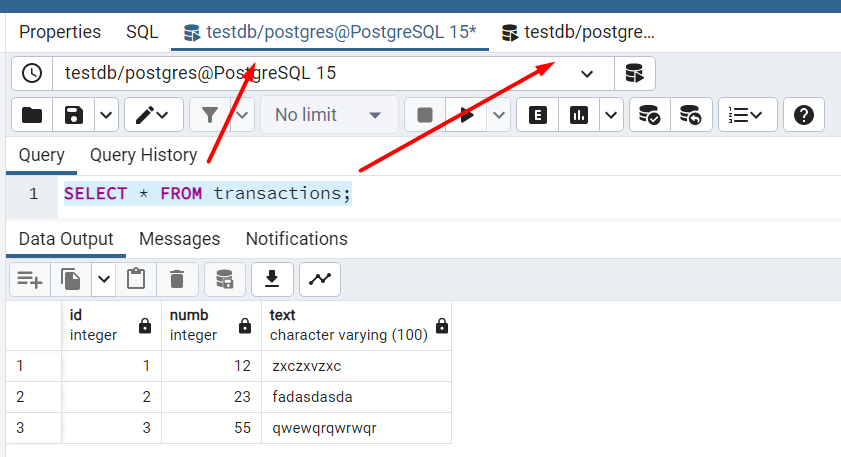
(55, 'qwewqrqwrwqr')

*READ COMMITTED*

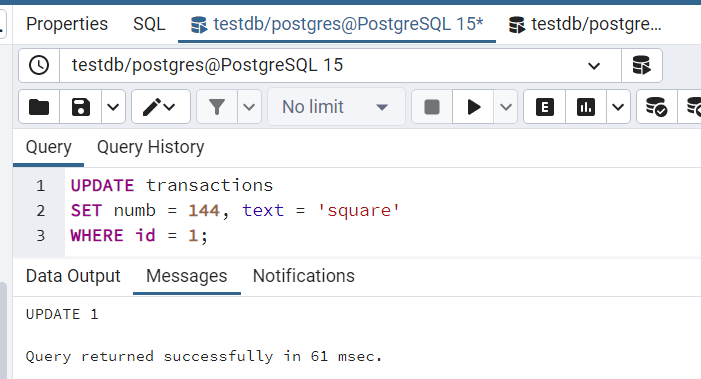
На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою *COMMIT* або *ROLLBACK*).

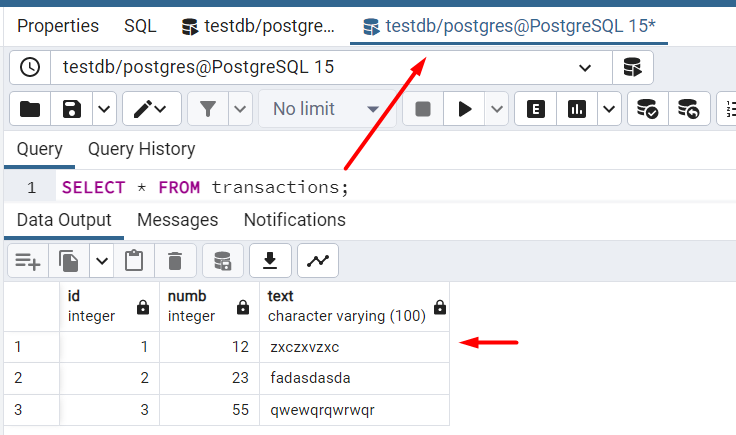
Спочатку у транзакціях 1 і 2 таблиця має однаковий стан. Якщо у транзакції 1 виконати редагування одного рядка, то в транзакції 2 цих змін не буде помітно, поки в першій транзакції не буде команди commit. Таким чином, феномен «*брудного читання*» на цьому рівні ізоляції неможливий.

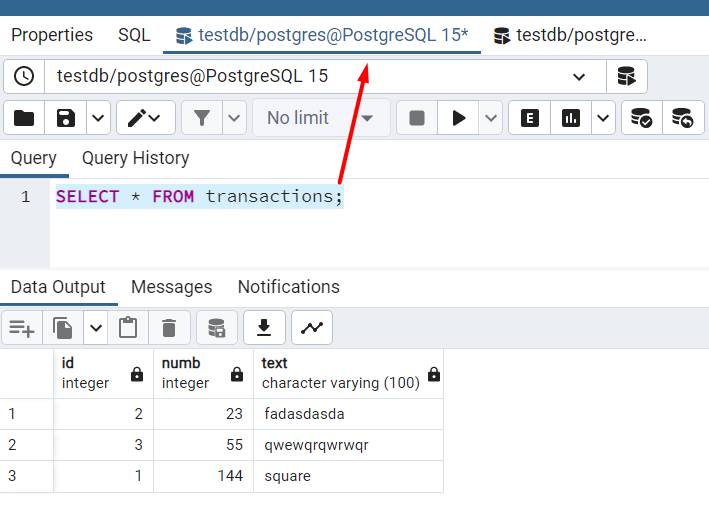




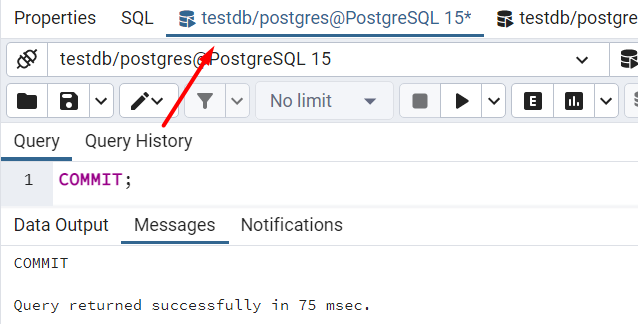
*Виконуємо редагування:*

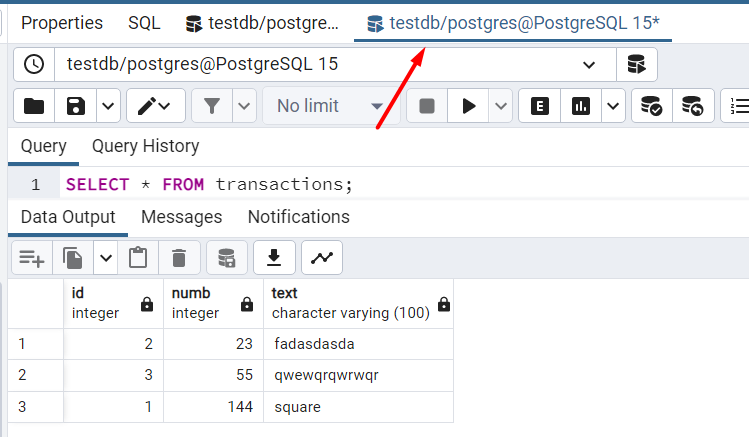






Тепер дослідимо феномен «*фантомного читання*».

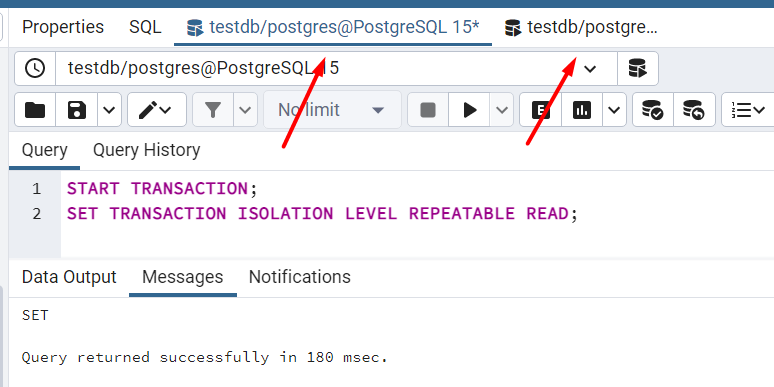


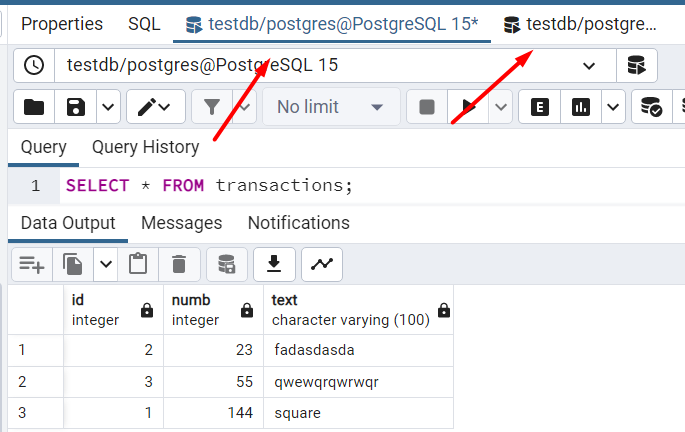


Після команди *COMMIT* у першій транзакції у другій ми побачимо, що зміни були внесені і збережені в обох транзакціях.

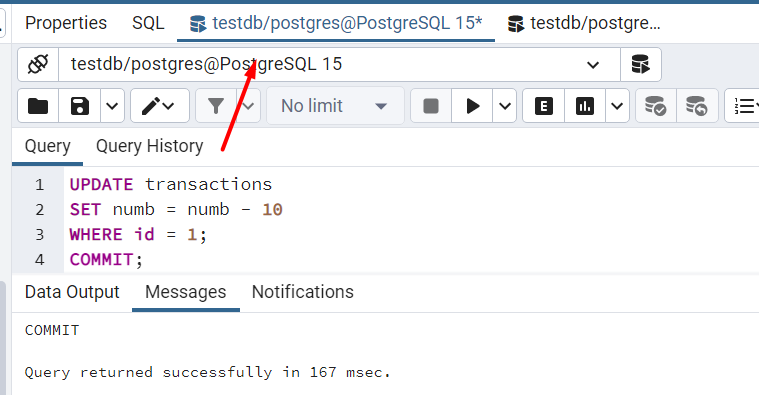
*REPEATABLE READ*

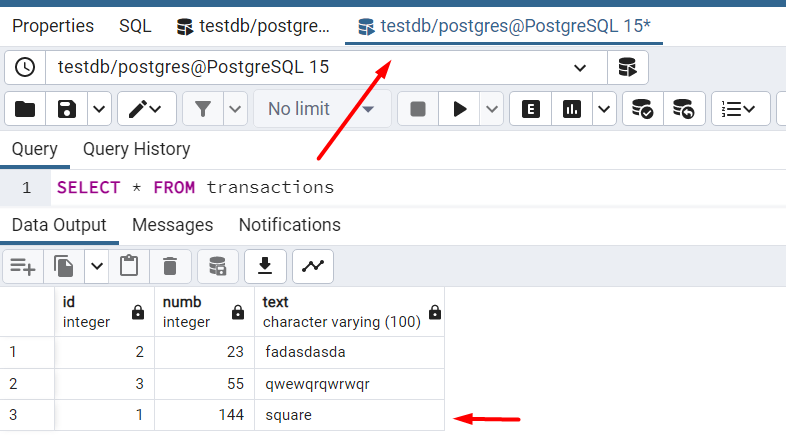
Почнемо дві транзакції на рівні ізоляції *REPEATEBLE READ*. У першій транзакції обираємо запис з *id = 1*. Тепер змінимо значення *numb(id =1)* та виконаємо команду *COMMIT*. У другій транзакції ніяких змін із цим рядком немає, хоча команда *COMMIT* була виконана. Це сталося через використання рівня ізоляції *REPEATEBLE READ*, тобто один і той самий запит має повертати той самий результат. Це призводить до того, що феномен «*неповторного читання*» неможливий на цьому рівні ізоляції.



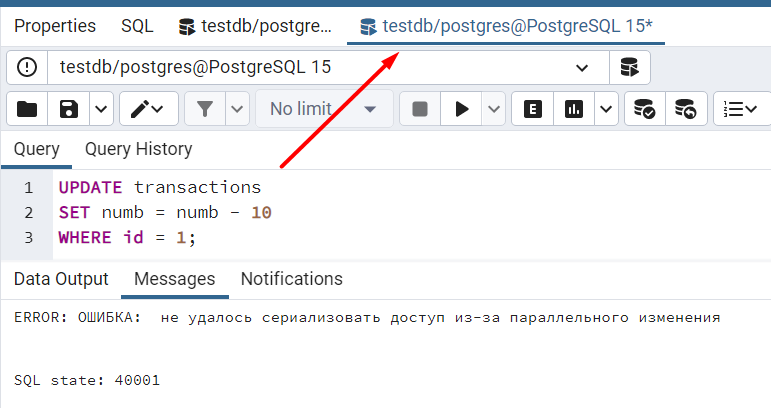


*Виконуємо редагування:*

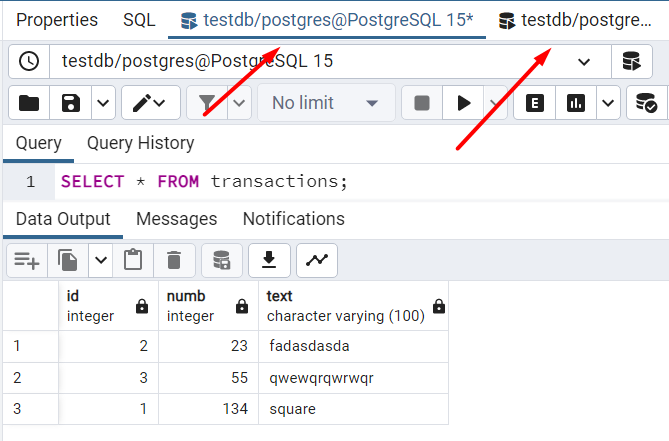


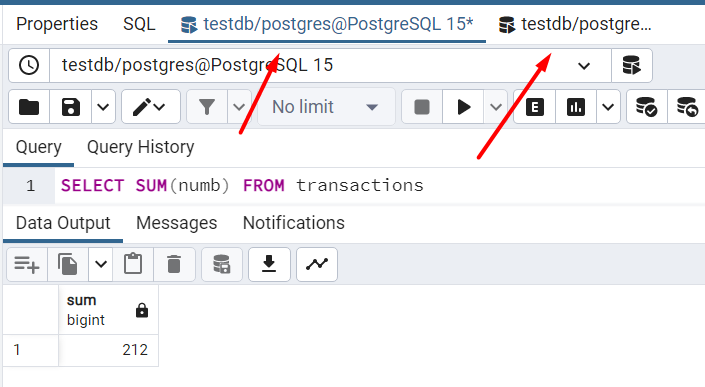


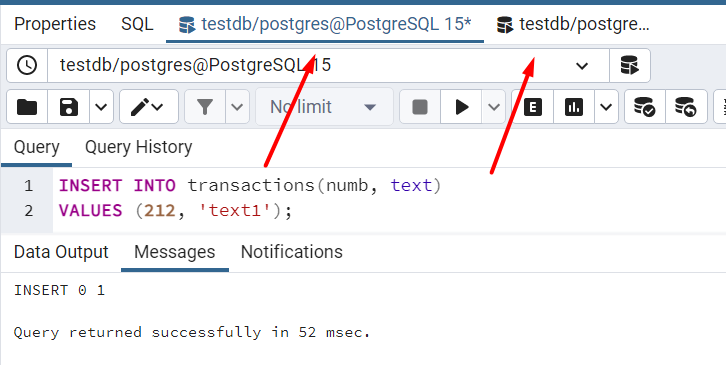
Якщо спробувати так само відрегадувати рядок в другій транзакції буде помилка. Це є перевагою рівня ізоляції *REPEATABLE READ* оскільки захищає від повторного редагування даних.

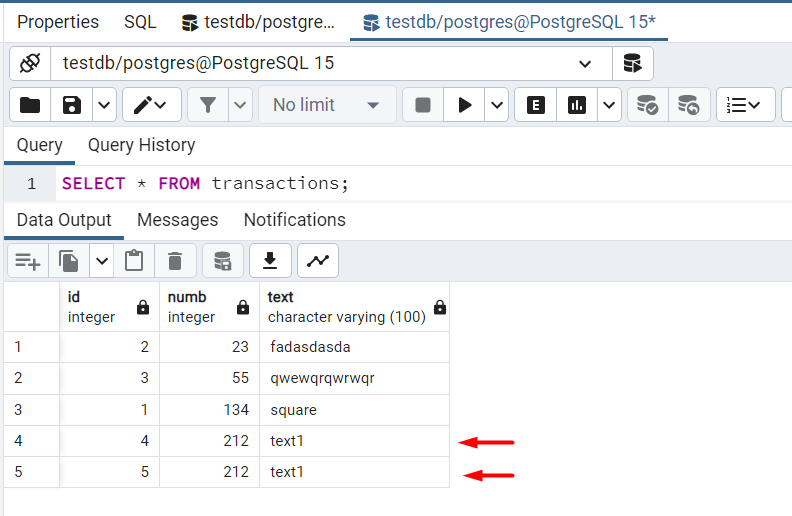


Тепер дослідимо аномалію серіалізації на рівні ізоляції *REPEATABLE READ*. Для цього запустимо дві транзації. У першій виведемо всі рядки і порахуємо суму стовпчика *numb* у всіх записах та додамо новий рядок з сумою в таблицю. Якщо у другій транзакції повторити ті ж самі операції, то стан таблиці на 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. Після виконання команди *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.



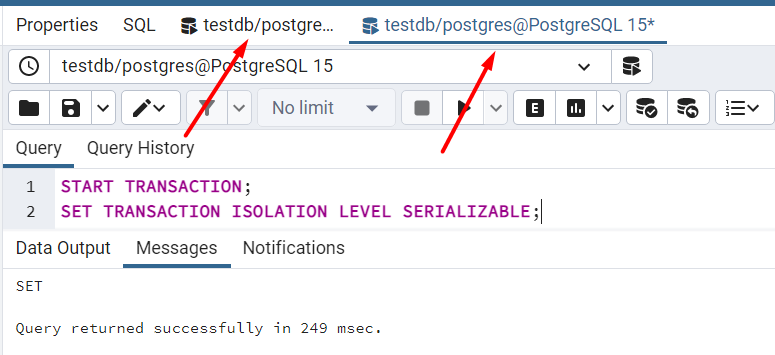


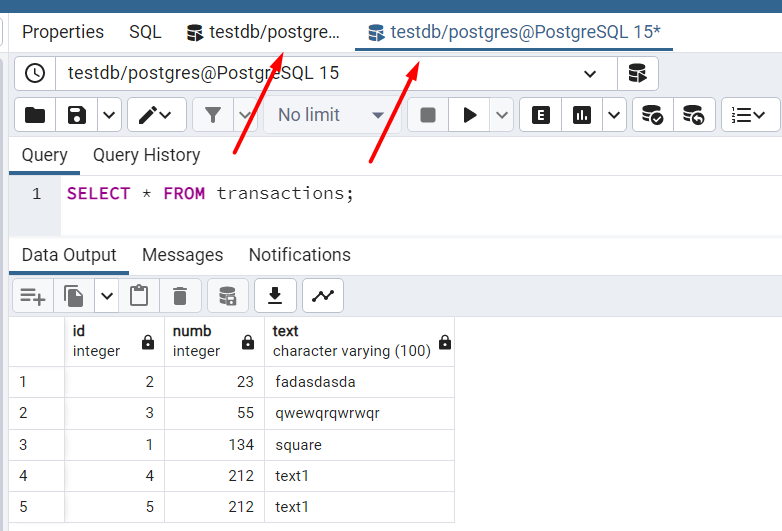




*SERIALIZABLE*

Запустимо дві транзакції на рівні ізоляції *SERIALIZABLE*. 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. Виправити таку ситуацію можна з домогою команди *ROLLBACK,* після її виконання зміни відбудуться і в другій транзакції.





*Виконуємо видалення в обох транзакціях:*

