

**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 KB-03  
Статечний Сергій

*Репозиторій:*

<https://github.com/Code01KPI/DBLAB3>

### Лабораторна робота № 3.

#### Засоби оптимізації роботи СУБД PostgreSQL

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

Завдання роботи полягає у наступному:

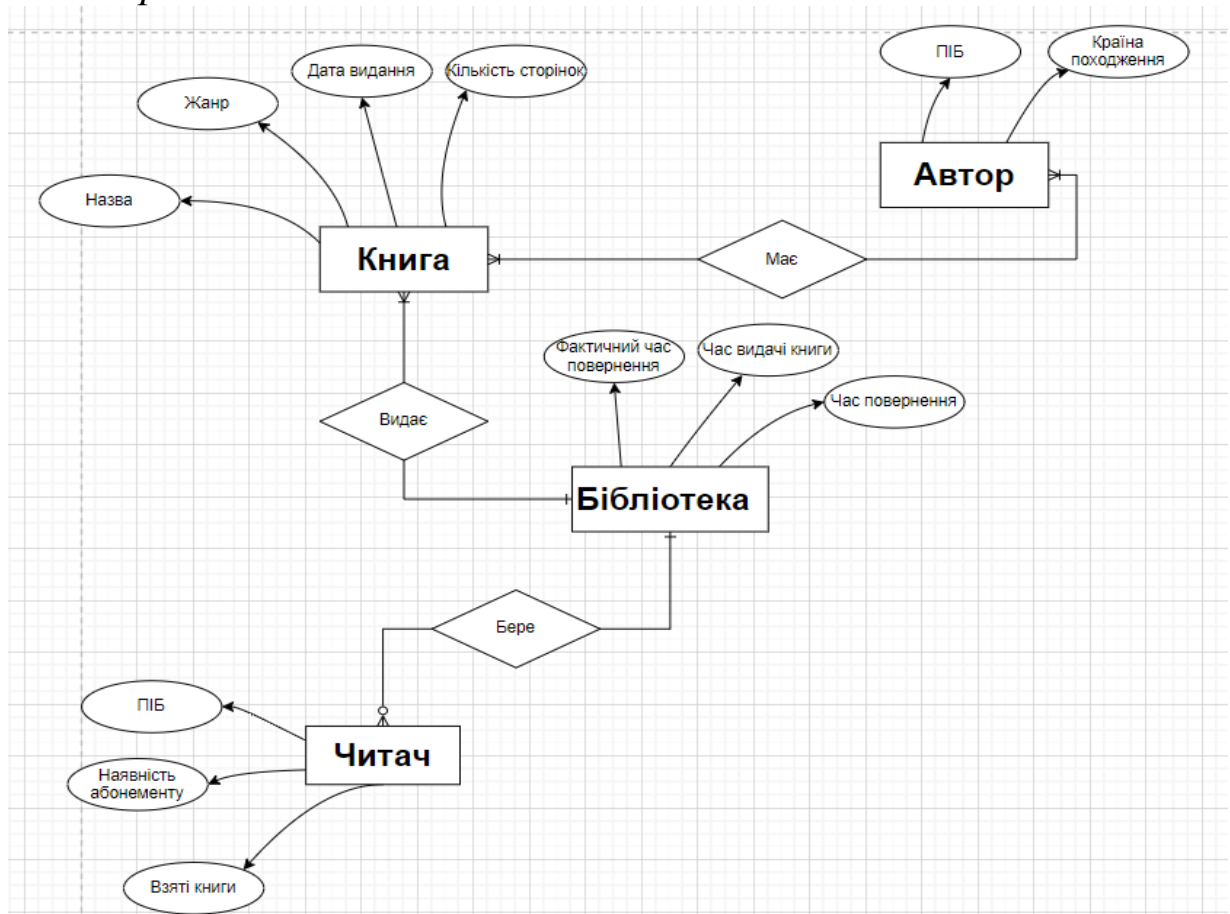
1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

#### Варіант – 21

№ варіанта	Види індексів	Умови для тригера
21	Btree, Hash	before delete, update

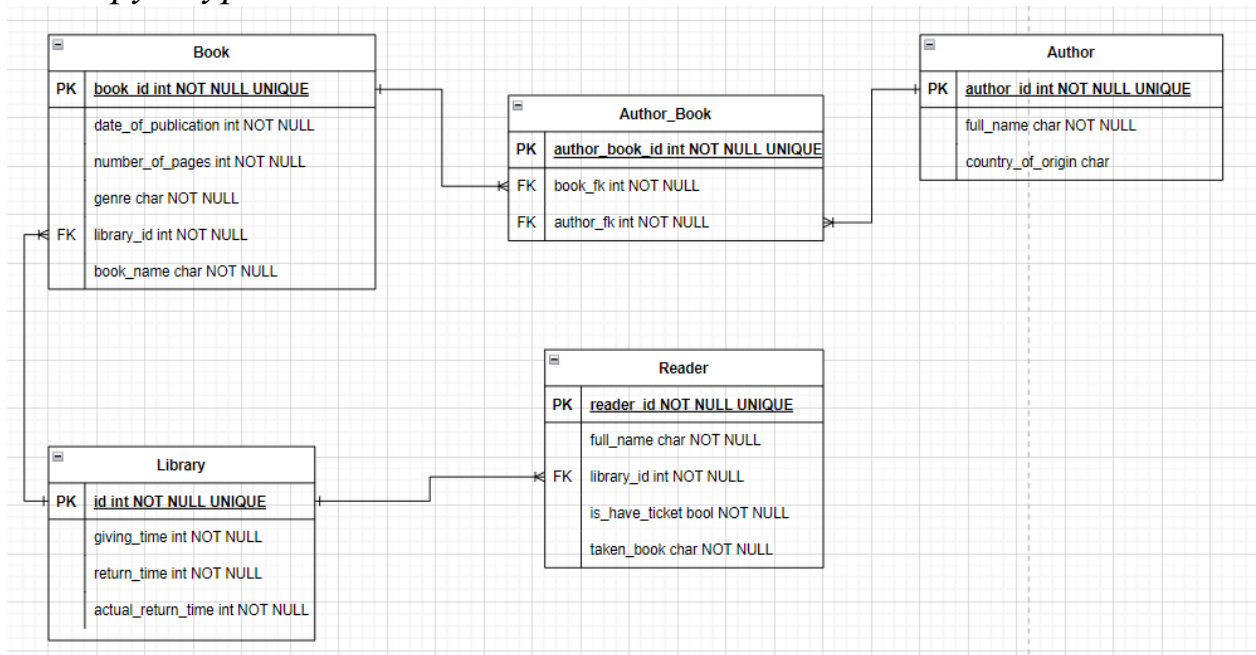
#### Завдання 1

ER діаграма



Нотація “Пташина лапка”

## Структура бази даних



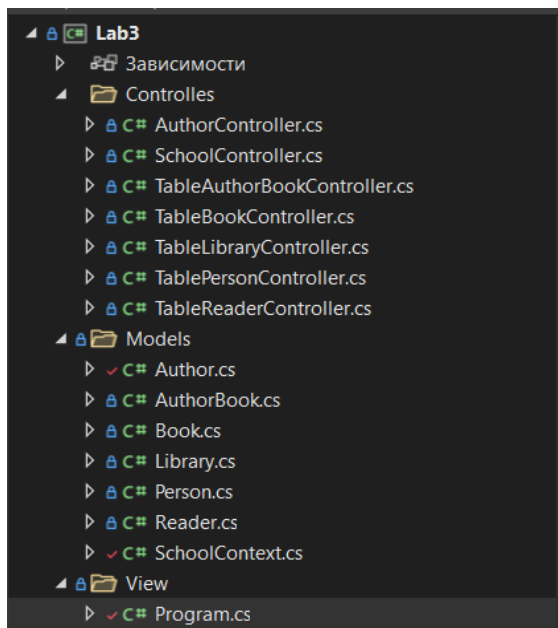
ER модель – бібліотека(видача книг читачам)

### 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*

```

Choose menu item: 1
1. Insert data
2. Update data
3. Delete data
4. Exit
Choose menu item: 1
1. Author
2. Author_Book
3. Book
4. Library
5. Person
6. Reader
7. Exit
Choose table: 1
Insert *author_id*(Attention! PK should't be repeated): 16
Insert *full_name*: Ivan Franko
Insert *country_of_origin*: Ukraine
Finish entering data? █

```

	author_id [PK] integer	full_name character varying	country_of_origin character varying
2	7	0257a5b68f34fb...	344003ef610b83...
3	8	fe4c4b93540361...	5d95670fb13d58...
4	9	f882b76134b5d1...	d43dcb93d07e9...
5	11	9f10e29d95e09b...	4cbf6c0a704951...
6	12	537834cbf7552d...	9112e0dd07c235...
7	13	5bca2875ea5b9e...	76a82e7959697...
8	14	f7c67b541f3791...	1c35fca4390c77...
9	15	048fdd477c5021...	87110433fe9f7e...

	author_id [PK] integer	full_name character varying	country_of_origin character varying
3	8	fe4c4b93540361a4844e7e58f1f947...	5d95670fb13d58714061d5ea2c6fcf...
4	9	f882b76134b5d1fd758e8ddf8847e...	d43dcb93d07e972a69a37f2190ecf7...
5	11	9f10e29d95e09bb386509ff4700dba...	4cbf6c0a704951ea1f731552383438...
6	12	537834cbf7552de457e167e596227...	9112e0dd07c235147865b2d82eed7...
7	13	5bca2875ea5b9e9f59178e50b3242a...	76a82e795969724fec4eb82046f896...
8	14	f7c67b541f3791278453a39bc07d9c...	1c35fca4390c77e8826cbfe111a120...
9	15	048fdd477c502174a1652410a39de...	87110433fe9f7e8a10eebd437749e9...
10	 16	Ivan Franko	Ukraine

## Редагування даних таблиці *Author*


```


1. Insert data
2. Update data
3. Delete data
4. Exit
Choose menu item: 2

1. Author
2. Author_Book
3. Book
4. Library
5. Person
6. Reader
7. Exit
Choose table: 1
Insert id of row for update: 16
Insert new *full_name*: Lesya Ukrainka
Insert new *country_of_origin*: Ukraine
16 - Ivan Franko - Ukraine
16 - Lesya Ukrainka - Ukraine
Finish updating data?

```



	author_id [PK] integer	full_name character varying	country_of_origin character varying
3	8	fe4c4b93540361a4844e7e58f1f947...	5d95670fb13d58714061d5ea2c6fcf...
4	9	f882b76134b5d1fd758e8ddfb8847e...	d43dcb93d07e972a69a37f2190ecf7...
5	11	9f10e29d95e09bb386509ff4700dba...	4cbf6c0a704951ea1f731552383438...
6	12	537834cbf7552de457e167e596227...	9112e0dd07c235147865b2d82eed7...
7	13	5bca2875ea5b9e9f59178e50b3242a...	76a82e795969724fec4eb82046f896...
8	14	f7c67b541f3791278453a39bc07d9c...	1c35fca4390c77e8826cbfe111a120...
9	15	048fdd477c502174a1652410a39de...	87110433fe9f7e8a10eebd437749e9...
10	 16	Ivan Franko	Ukraine

	author_id [PK] integer	full_name character varying	country_of_origin character varying
3	8	fe4c4b93540361a4844e7e58f1f947...	5d95670fb13d58714061d5ea2c6fcf...
4	9	f882b76134b5d1fd758e8ddfb8847e...	d43dcb93d07e972a69a37f2190ecf7...
5	11	9f10e29d95e09bb386509ff4700dba...	4cbf6c0a704951ea1f731552383438...
6	12	537834cbf7552de457e167e596227...	9112e0dd07c235147865b2d82eed7...
7	13	5bca2875ea5b9e9f59178e50b3242a...	76a82e795969724fec4eb82046f896...
8	14	f7c67b541f3791278453a39bc07d9c...	1c35fca4390c77e8826cbfe111a120...
9	15	048fdd477c502174a1652410a39de...	87110433fe9f7e8a10eebd437749e9...
10	 16	Lesya Ukrainka	Ukraine


Вилучення даних з таблиці *Author*

```

1. Insert data
2. Update data
3. Delete data
4. Exit
Choose menu item: 3

1. Author
2. Author_Book
3. Book
4. Library
5. Person
6. Reader
7. Exit
Choose table: 1
Insert row id for deleting: 16
Finish deleting data? _

```

	author_id [PK] integer	full_name character varying	country_of_origin character varying
3	8	fe4c4b93540361a4844e7e58f1f947...	5d95670fb13d58714061d5ea2c6fcf...
4	9	f882b76134b5d1fd758e8ddfb8847e...	d43dcb93d07e972a69a37f2190ecf7...
5	11	9f10e29d95e09bb386509ff4700dba...	4cbf6c0a704951ea1f731552383438...
6	12	537834cbf7552de457e167e596227...	9112e0dd07c235147865b2d82eed7...
7	13	5bca2875ea5b9e9f59178e50b3242a...	76a82e795969724fec4eb82046f896...
8	14	f7c67b541f3791278453a39bc07d9c...	1c35fca4390c77e8826cbfe111a120...
9	15	048fdd477c502174a1652410a39de...	87110433fe9f7e8a10eebd437749e9...
10	 16	Lesya Ukrainka	Ukraine

	author_id [PK] integer	full_name character varying	country_of_origin character varying
2	7	0257a5b68f34fb5c21184c9dbf6975...	344003ef610b83a387c5cfab727a1a...
3	8	fe4c4b93540361a4844e7e58f1f947...	5d95670fb13d58714061d5ea2c6fcf...
4	9	f882b76134b5d1fd758e8ddfb8847e...	d43dcb93d07e972a69a37f2190ecf7...
5	11	9f10e29d95e09bb386509ff4700dba...	4cbf6c0a704951ea1f731552383438...
6	12	537834cbf7552de457e167e596227...	9112e0dd07c235147865b2d82eed7...
7	13	5bca2875ea5b9e9f59178e50b3242a...	76a82e795969724fec4eb82046f896...
8	14	f7c67b541f3791278453a39bc07d9c...	1c35fca4390c77e8826cbfe111a120...
9	15	048fdd477c502174a1652410a39de...	87110433fe9f7e8a10eebd437749e9...

## 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 ключі 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');
```

	id [PK] bigint	time timestamp without time zone
1	1	2019-03-29 09:20:28.387702
2	2	2019-09-23 06:26:38.065376
3	3	2019-04-30 10:12:39.983497
4	4	2019-02-11 23:29:50.122169
5	5	2019-02-17 14:12:50.893896
6	6	2019-03-11 20:35:12.505349
7	7	2019-11-21 15:07:26.172046
8	8	2019-11-08 02:30:20.832452
9	9	2019-06-14 10:19:13.956621

*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);
```

	id [PK] bigint	text character varying (100)
1	1	57d37580ac38622079...
2	2	a772e48ba3ea0028e6...
3	3	6d9bc9a14dd2100202...
4	4	a5f84fed7b3924b6900...
5	5	bcdc716c00080efcf9b...
6	6	f43d18b9d2d2bcee70f...
7	7	b0118f426200c19111...
8	8	832956b665dad9b65c...
9	9	...
Total rows: 1000 of 10000		Query complete

*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 створено 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 здійснене 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 DELETE FROM trigger_test
2 WHERE first_name = 'Andrew';

```

Data Output Messages Notifications

ERROR: ОШИБКА: Exc delete  
CONTEXT: функция PL/pgSQL trigger\_func(), строка 5, оператор RAISE

SQL state: P0001

```

1 UPDATE trigger_test
2 SET first_name = 'Tom', age = 55
3 WHERE first_name = 'Andrew';

```

Data Output Messages Notifications

ERROR: ОШИБКА: Exc update  
CONTEXT: функция PL/pgSQL trigger\_func(), строка 14, оператор RAISE

SQL state: P0001

2. Видалення з таблиці:

```
1 DELETE FROM trigger_test
2 WHERE id = 2;
```

Data Output Messages Notifications

DELETE 1

Query returned successfully in 68 msec.

*trigger\_test:*

	id integer	first_name character varying (20)	age integer
1	1	Andrew	20
2	3	Serhii	40

*trigger\_log:*

	id integer	test_id integer	old_first_name character varying (20)	old_age integer	changed_on timestamp without time zone	is_updated boolean
1	1	2	Anton	30	2023-01-11 19:13:26.105544	false

### 3. Редагування:

```
1 UPDATE trigger_test
2 SET first_name = 'Tom', age = 55
3 WHERE first_name <> 'Andrew';
```

Data Output Messages Notifications

UPDATE 1

Query returned successfully in 71 msec.

*trigger\_test:*

	id integer	first_name character varying (20)	age integer
1	1	Andrew	20
2	3	Tom	55



*trigger\_log:*

1

SELECT \* FROM trigger\_log;

Data Output

Messages

Notifications

	id integer	test_id integer	old_first_name character varying (20)	old_age integer	changed_on timestamp without time zone	is_updated boolean
1	1	2	Anton	30	2023-01-11 19:13:26.105544	false
2	2	3	Serhii	40	2023-01-11 19:17:33.626815	true

## 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, 'qwewqrqrwrqr')
```

## 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.



Properties SQL testdb/postgre... testdb/postgre... testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

No limit

Query Query History

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

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	1	12	zxczvxzxc
2	2	23	fadasdasda
3	3	55	qwewqrqwrwqr

*Виконуємо редагування:*

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 UPDATE transactions  
2 SET numb = 144, text = 'square'  
3 WHERE id = 1;
```

Data Output Messages Notifications

UPDATE 1

Query returned successfully in 61 msec.

Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

No limit

Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	1	12	zxczvxzxc
2	2	23	fadasdasda
3	3	55	qwewqrqwrwqr

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	144	square

Тепер дослідимо феномен «фантомного читання».

The top screenshot shows a PostgreSQL client interface with the query `COMMIT;` executed successfully. The message pane displays "COMMIT" and "Query returned successfully in 75 msec." A red arrow points to the database connection dropdown menu.

The bottom screenshot shows the same interface with the query `SELECT * FROM transactions;` executed. The message pane displays the query result. A red arrow points to the database connection dropdown menu.

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	144	square

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 *REPEATABLE 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 *REPEATABLE 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.

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 START TRANSACTION;  
2 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Data Output Messages Notifications

SET

Query returned successfully in 180 msec.

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

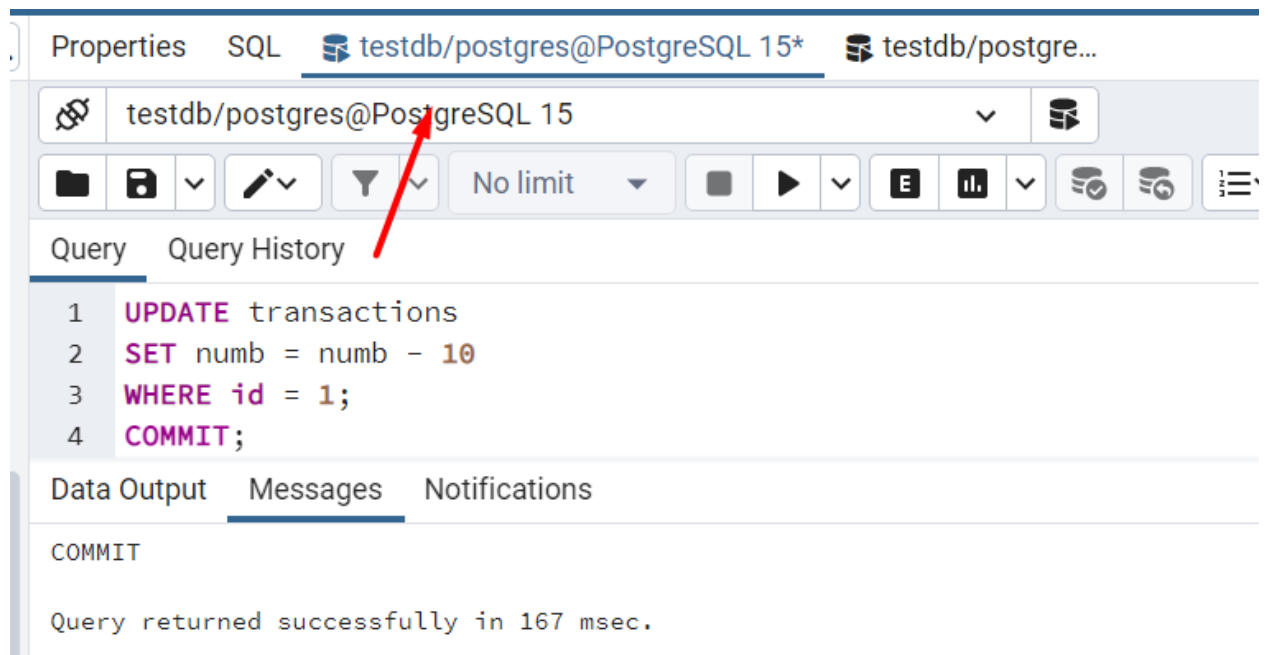
Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	144	square

Виконуємо редагування:



Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

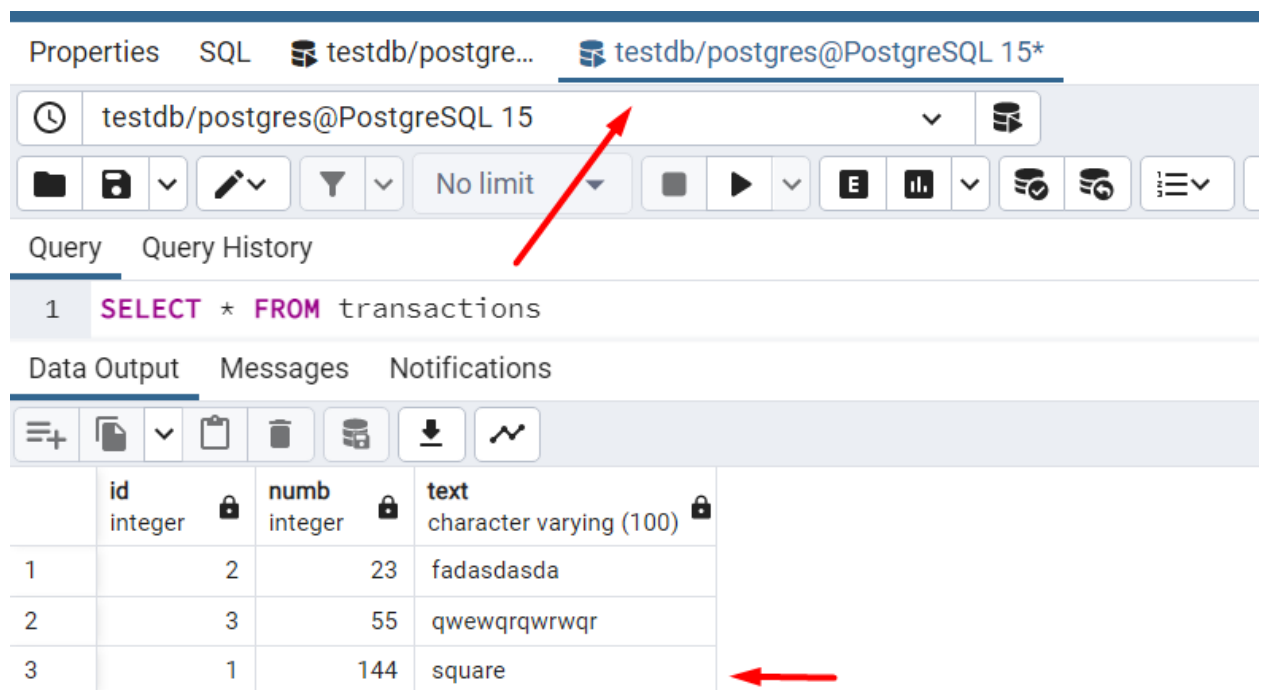
Query Query History

```
1 UPDATE transactions
2 SET numb = numb - 10
3 WHERE id = 1;
4 COMMIT;
```

Data Output Messages Notifications

COMMIT

Query returned successfully in 167 msec.



Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

Query Query History

```
1 SELECT * FROM transactions
```

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	144	square

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

Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

Query Query History

```
1 UPDATE transactions
2 SET numb = numb - 10
3 WHERE id = 1;
```

Data Output Messages Notifications

ERROR: ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения

SQL state: 40001

Тепер дослідимо аномалію серіалізації на рівні ізоляції *REPEATABLE READ*. Для цього запустимо дві транзакції. 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.

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	134	square

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 SELECT SUM(num) FROM transactions
```

Data Output Messages Notifications

	sum	
	bigint	
1	212	

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 INSERT INTO transactions(num, text)
2 VALUES (212, 'text1');
```

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 52 msec.

Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

Query Query History

1 **SELECT \* FROM transactions;**

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	134	square
4	4	212	text1
5	5	212	text1

## 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 допомогою команди *ROLLBACK*, після її виконання зміни відбудуться і в другій транзакції.

Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

Query Query History

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

Data Output Messages Notifications

SET

Query returned successfully in 249 msec.



Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

No limit

Query Query History

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

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	134	square
4	4	212	text1
5	5	212	text1

*Виконуємо видалення в обох транзакціях:*

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 DELETE FROM transactions
2 WHERE id = 5;
```

Data Output Messages Notifications

DELETE 1

Query returned successfully in 47 msec.

Properties SQL testdb/postgre... testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 DELETE FROM transactions
2 WHERE id = 5;
```

Data Output Messages Notifications

	id integer	numb integer	text character varying (100)
1	2	23	fadasdasda
2	3	55	qwewqrqwrwqr
3	1	134	square
4	4	212	text1
5	5	212	text1

Waiting for the query to complete...

Properties SQL testdb/postgres@PostgreSQL 15\* testdb/postgre...

testdb/postgres@PostgreSQL 15

No limit

Query Query History

```
1 COMMIT;
```

Data Output Messages Notifications

COMMIT

Query returned successfully in 69 msec.

Properties   SQL   testdb/postgre...   testdb/postgres@PostgreSQL 15\*

testdb/postgres@PostgreSQL 15

No limit

Query   Query History

```
1 DELETE FROM transactions
2 WHERE id = 5;
```

Data Output   Messages   Notifications

ERROR: ОШИБКА: не удалось сериализовать доступ из-за параллельного удаления

SQL state: 40001