



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп'ютерних  
систем**

**Лабораторна робота №2**

**з дисципліни Баз даних і засоби управління**

*на тему: “Засоби оптимізації роботи СУБД PostgreSQL”*

Виконала:

студентка III курсу

групи KB-12

Павленко Л.П.

Перевірив: Павловський В. І.

Київ – 2023

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

*Завдання роботи полягає у наступному:*

1. Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

*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 результуючих 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 спосіб 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.

### *Вимоги до інструментарію*

1. Бібліотека для реалізації ORM - [SQLAlchemy для Python](#) або інша з подібною функціональністю.
2. Середовище для відлагодження SQL-запитів до бази даних – pgAdmin 4.
3. СУБД - PostgreSQL 13-15.

### *Завдання за варіантом №16:*

16	GIN, Hash	after delete, insert
----	-----------	----------------------

## Завдання №1

### Схеми бази даних «Електронний каталог для зберігання музичних нот та композицій»

### Логічна модель бази даних «Електронний каталог для зберігання музичних нот та композицій»

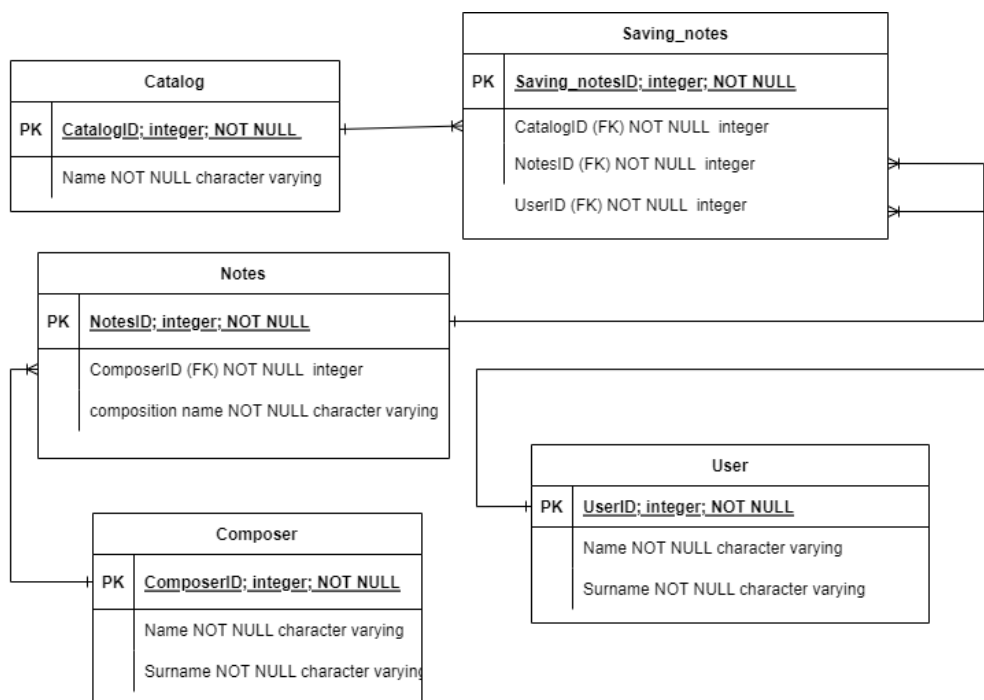


Рисунок 1 – Схема бази даних

## Класи ORM

```
class Catalog(Base):  
    __tablename__ = 'Catalog'  
    CatalogID = Column(Integer, primary_key=True)  
    catalog_name = Column(String)
```

Рисунок 3 – Клас «Catalog»

```
class Composer(Base):  
    __tablename__ = 'Composer'  
    ComposerID = Column(Integer, primary_key=True)  
    Name = Column(String)  
    Surname = Column(String)
```

Рисунок 4 – Клас «Composer»

```
class Notes(Base):  
    __tablename__ = 'Notes'  
    NotesID = Column(Integer, primary_key=True)  
    ComposerID = Column(Integer, ForeignKey('Composer.ComposerID'))  
    composition_name = Column(String)
```

Рисунок 5 – Клас «Notes»

```
class User(Base):  
    __tablename__ = 'User'  
    UserID = Column(Integer, primary_key=True)  
    Name = Column(String)  
    Surname = Column(String)
```

Рисунок 6 – Клас «User»

```
class Saving_notes(Base):
    __tablename__ = 'Saving_notes'
    Saving_notesID = Column(Integer, primary_key=True)
    CatalogID = Column(Integer, ForeignKey('Catalog.CatalogID'))
    NotesID = Column(Integer, ForeignKey('Notes.NotesID'))
    UserID = Column(Integer, ForeignKey('User.UserID'))
```

Рисунок 7 – Клас «Saving\_notes»

На рисунках 3, 4, 5, 6, 7 зображені класи ORM а також в них прописані зв'язки, де таблиці «Notes» та «Composer» мають зв'язок багато до багатьох, а таблиці «Notes», «Catalog» та «User» мають зв'язок з таблицею «Saving\_notes» також багато до багатьох, що відповідають таблицям бази даних.

### Приклади запитів у вигляді ORM

```
def add_catalog(self, catalog_id, name):
    s = Session()
    check = s.query(exists().where(Catalog.CatalogID == catalog_id)).scalar()
    if (check):
        print("Error! This identifier already exists")
    else:
        catalog = Catalog(
            CatalogID=catalog_id,
            catalog_name=name
        )
        s.add(catalog)
        s.commit()
        print("User added successfully!")
    s.close()
```

Рисунок 8 – Функція внесення даних в таблицю «Catalog»

```
def update_catalog(self, catalog_id, catal_name):
    s = Session()
    catalog = s.query(Catalog).filter_by(CatalogID=catalog_id).first()
    if catalog:
        catalog.catalog_name = catal_name
        s.commit()
        print("Catalog updated successfully!")
    else:
        print("Error! Catalog does not exist")
    s.close()
```

Рисунок 9 – Функція редагування в таблиці «Catalog»

```
def delete_catalog(self, catalog_id):
    with session_scope() as s:
        saving_notes = s.query(Saving_notes).filter_by(CatalogID=catalog_id).all()
        if saving_notes:
            for notes in saving_notes:
                s.delete(notes)
            s.commit()
            print("Catalog deleted successfully!")

        catalog = s.query(Catalog).get(catalog_id)
        if catalog:
            s.delete(catalog)
            s.commit()
            print("Catalog deleted successfully!")
        else:
            print("Error! Catalog does not exist")
```

Рисунок 10 – Функція видалення в таблиці «Catalog»

На рисунках 8, 9 та 10 зображені функції в яких виконуються запити у вигляді ORM.

## Завдання №2

Спочатку для використання індексу GIN змінимо тип даних в стовпці «Name» з character varying на tsvector в таблиці «User».

За допомогою функції яка зображена на рисунку 11 ми генеруємо випадковим чином дані:

```
def gener_add_user(self, num1, num2):
    c = self.conn.cursor()
    num = 0
    for i in range(num1, num2 + 1):
        c.execute(query: 'SELECT * FROM "User" WHERE "UserID" = %s', vars: (i,))
        check = c.fetchall()
        if check:
            print("UserID %s already exists", i)
            num = 1

    if num == 0:
        # c.execute('INSERT INTO "User" ("UserID", "Name", "Surname") SELECT generate_series as UserID, chr(trunc(65+random()*25)::int) || chr(trunc(65+random()*25)::int) as Name, chr(trunc(65+random()*25)::int) || chr(trunc(65+random()*25)::int) as Surname FROM generate_series(%s, %s)', vars: (num1, num2))
        c.execute(query: 'INSERT INTO "User" ("UserID", "Name", "Surname") '
            'SELECT generate_series as "UserID", to_tsvector(chr(trunc(65+random()*50)::int) || chr(trunc(65+random()*25)::int)) as Name, '
            'chr(trunc(65+random()*25)::int) as Name, chr(trunc(65+random()*25)::int) || chr(trunc(65+random()*25)::int) as Surname '
            'FROM generate_series(%s, %s)', vars: (num1, num2))
        self.conn.commit()
    else:
        print("Error! Identifiers already exist")
```

Рисунок 11 – Функція «gener\_add\_user»



## Запити без використання індекса Hash

Query	Query History	Scratch Pad
1	<code>explain analyze select * from "User" order by "Surname";</code>	
<div> Data Output Messages Notifications </div> <div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑</div> <div>🗄</div> <div>⬇</div> <div>📈</div> </div> </div>		
<div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div>		
1	Sort (cost=136536.79..139036.77 rows=999991 width=23) (actual time=7167.896..7428.845 rows=999991 loops=1)	
2	Sort Key: "Surname"	
3	Sort Method: external merge Disk: 33040kB	
4	-> Seq Scan on "User" (cost=0.00..16369.91 rows=999991 width=23) (actual time=0.010..323.868 rows=999991 loops=1)	
5	Planning Time: 0.115 ms	
6	Execution Time: 7493.972 ms	

Рисунок 12 – Запит, який містить сортування без використання індексу

Query	Query History	Scratch Pad
1	<code>explain analyze select * from "User" WHERE "Surname" = 'PR';</code>	
<div> Data Output Messages Notifications </div> <div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑</div> <div>🗄</div> <div>⬇</div> <div>📈</div> </div> </div>		
<div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div>		
1	Gather (cost=1000.00..12736.99 rows=1587 width=23) (actual time=0.448..440.194 rows=1601 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on "User" (cost=0.00..11578.29 rows=661 width=23) (actual time=0.008..83.191 rows=534 loop=1)	
5	Filter: (("Surname")::text = 'PR'::text)	
6	Rows Removed by Filter: 332797	
7	Planning Time: 0.179 ms	
8	Execution Time: 440.505 ms	

Рисунок 13 – Запит, який містить фільтрацію без використання індексу

На рисунках 12 та 13 зображено запити, які містять сортування та фільтрацію.

## Створення Hash індексу

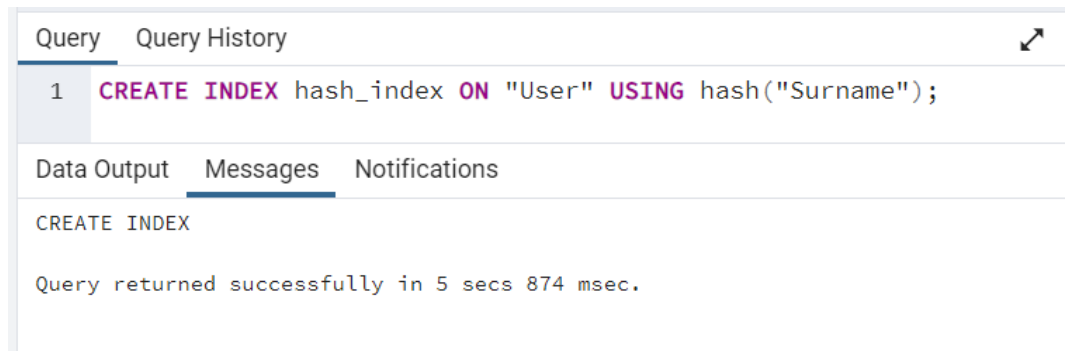


Рисунок 15 – Створення Hash індексу

На рисунку 15 зображено створення Hash індексу.

## Запити з використанням індекса «Hash»

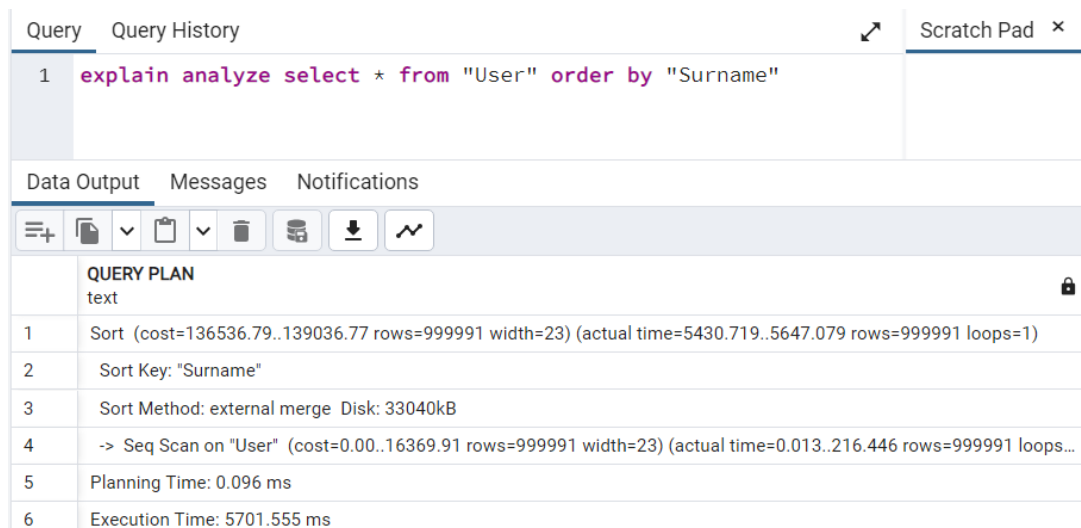


Рисунок 16 – Запит, який містить сортування з використанням Hash індексу

Query	Query History	Scratch Pad
1	<code>explain analyze select * from "User" WHERE "Surname" = 'PR';</code>	
Data Output Messages Notifications		
<div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div>		
	<b>QUERY PLAN</b>	🔒
	text	
1	Gather (cost=1000.00..12736.99 rows=1587 width=23) (actual time=0.399..180.205 rows=1601 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on "User" (cost=0.00..11578.29 rows=661 width=23) (actual time=0.186..83.385 rows=534 loop...	
5	Filter: (("Surname")::text = 'PR')::text)	
6	Rows Removed by Filter: 332797	
7	Planning Time: 0.123 ms	
8	Execution Time: 180.477 ms	

Рисунок 17 – Запит, який містить фільтрацію з використанням Nash індексу

На рисунках 16 та 17 показано запити сортування та фільтрації з використання Nash індексу. Дивлячись на час виконання цих запитів, можемо сказати, що запити там де використовується Nash індекс виконуються набагато швидше ніж ті, в яких він не використовується.

## Запити без використання індекса GIN

Query	Query History	Scratch Pad
1	<code>explain analyze select count(*) from "User" group by "Name";</code>	
<div> Data Output Messages Notifications </div> <div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>📦</div> <div>⬇️</div> <div>📈</div> </div> <div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div> </div>		
1	GroupAggregate (cost=133117.29..150617.13 rows=999991 width=24) (actual time=2116.578..3265.858 rows=16875 loo...	
2	Group Key: "Name"	
3	-> Sort (cost=133117.29..135617.27 rows=999991 width=16) (actual time=2116.492..2839.015 rows=999991 loops=1)	
4	Sort Key: "Name"	
5	Sort Method: external merge Disk: 20368kB	
6	-> Seq Scan on "User" (cost=0.00..16369.91 rows=999991 width=16) (actual time=0.025..158.958 rows=999991 loop...	
7	Planning Time: 0.175 ms	
8	Execution Time: 3274.958 ms	

Рисунок 18 – Запит, який містить агрегатну функцію без використання GIN індексу

Query	Query History	Scratch Pad
1	<code>explain analyze select * from "User" order by "Name" ASC;</code>	
<div> Data Output Messages Notifications </div> <div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>📦</div> <div>⬇️</div> <div>📈</div> </div> <div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div> </div>		
1	Sort (cost=136536.79..139036.77 rows=999991 width=23) (actual time=2789.658..3171.884 rows=999991 loops=1)	
2	Sort Key: "Name"	
3	Sort Method: external merge Disk: 33040kB	
4	-> Seq Scan on "User" (cost=0.00..16369.91 rows=999991 width=23) (actual time=0.016..212.110 rows=999991 loops...	
5	Planning Time: 0.110 ms	
6	Execution Time: 3235.702 ms	

Рисунок 19 – Запит, який містить сортування без використання GIN індексу

На рисунка 18 та 19 зображені запити, які містять агрегатні функції та сортування, без використання

### Створення GIN індекса

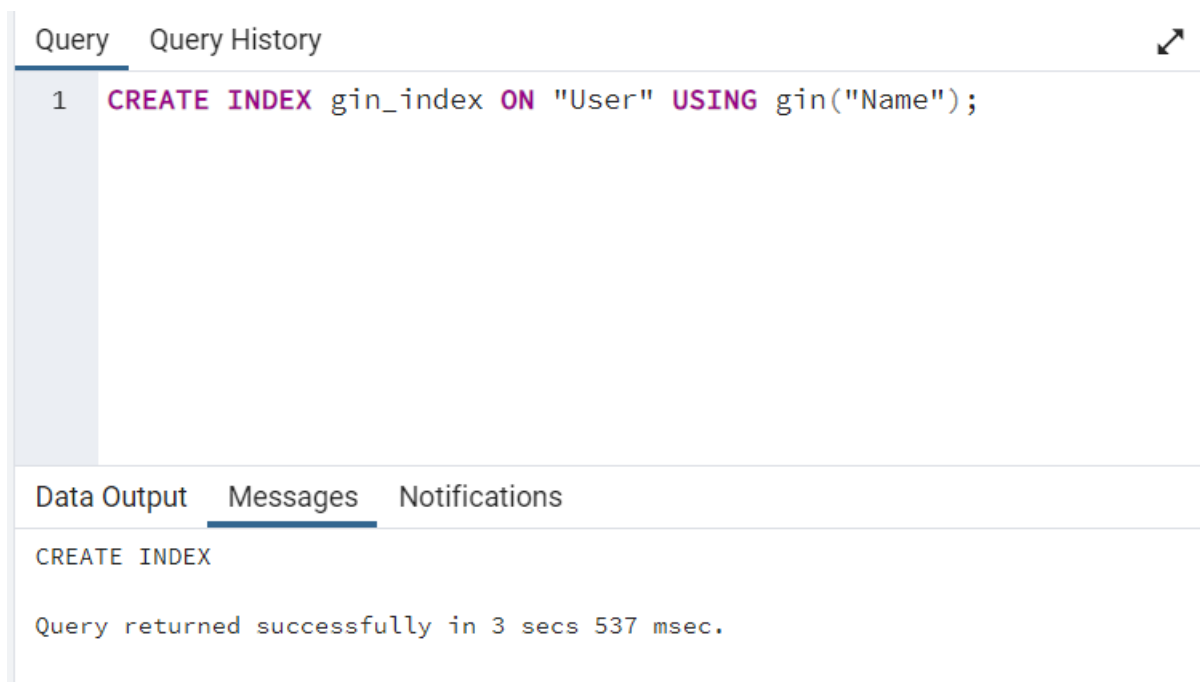


Рисунок 19 – Створення GIN індексу

На рисунку 19 зображено створення GIN індекса.



На рисунках 21 та 22 зображено запити, які містять агрегатну функцію та сортування, з використанням GIN індекса.

Для порівняння створимо меншу за обсягом таблицю «users\_1», де 300 рядків.

Виконаємо запит без використання індексу GIN, результат виконання ми бачимо на рисунку 23.

Query	Query History	
1	<code>explain analyze select count(*) from users_1 group by name;</code>	
Data Output	Messages	Notifications
<div><div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div></div></div></div>		
	<div><div>QUERY PLAN</div><div>text</div><div></div></div>	
1	GroupAggregate (cost=17.34..22.59 rows=300 width=24) (actual time=0.168..0.267 rows=297 loops=1)	
2	Group Key: name	
3	-> Sort (cost=17.34..18.09 rows=300 width=16) (actual time=0.164..0.175 rows=300 loops=1)	
4	Sort Key: name	
5	Sort Method: quicksort Memory: 34kB	
6	-> Seq Scan on users_1 (cost=0.00..5.00 rows=300 width=16) (actual time=0.008..0.043 rows=300 loops=1)	
7	Planning Time: 1.565 ms	
8	Execution Time: 0.294 ms	

Рисунок 23 – Запит, який містить агрегатну функцію без використання GIN індексу в таблиці «users\_1»

Далі створюємо індекс GIN, це зображена на рисунку 24.

Query	Query History
1	<code>create index gin_inx on users_1 using gin(name);</code>
2	

Data Output	Messages	Notifications
CREATE INDEX		
Query returned successfully in 53 msec.		

Рисунок 24 – Створення GIN індексу в таблиці «users\_1»

Після створення індексу виконуємо такий же запит, який зображений на рисунку 25, і тоді ми бачимо що з використанням індексу для маленької за обсягом таблиці час на виконання запиту набагато більший.

Query Query History	
1	<code>explain analyze select count(*) from users_1 group by name;</code>
Data Output Messages Notifications	
<div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div> </div>	
	<b>QUERY PLAN</b> text <span>🔒</span>
1	GroupAggregate (cost=17.34..22.59 rows=300 width=24) (actual time=1.075..1.309 rows=297 loops=1)
2	Group Key: name
3	-> Sort (cost=17.34..18.09 rows=300 width=16) (actual time=1.006..1.030 rows=300 loops=1)
4	Sort Key: name
5	Sort Method: quicksort Memory: 34kB
6	-> Seq Scan on users_1 (cost=0.00..5.00 rows=300 width=16) (actual time=0.051..0.082 rows=300 loops=1)
7	Planning Time: 4.745 ms
8	Execution Time: 3.509 ms



## Завдання №3

Для використання триггеру де виконуються умови: «after delete» та «after insert» створимо дві таблиці: «users\_tr\_1» та «users\_tr\_2», в кожній з яких буде 150 рядків, це зображено на рисунку 26.



Рисунок 26 – Створення таблиць «users\_tr\_1» та «users\_tr\_2»

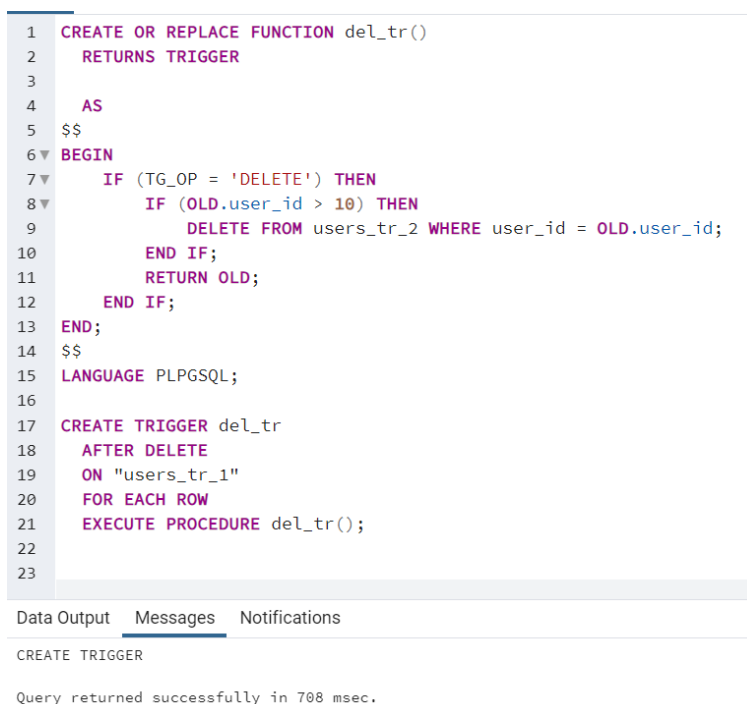


Рисунок 27 – Створення тригера «del\_tr»

Даний тригер «del\_tr» передбачає умову, що після видалення певних рядків з таблиці «users\_tr\_1», то в таблиці «users\_tr\_2» будуть видалятися такі самі рядки, але при умові, що user\_id більше 10. Дію даного тригера ми можемо спостерігати після видалення елементів які кратні 2, видалення зображене на рисунку 28.

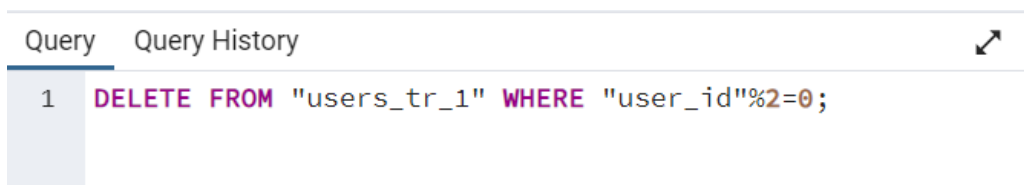


Рисунок 28 –Видалення рядків за умовою, що user\_id кратний 2

The screenshot shows a SQL query editor with two tabs: 'Query' and 'Query History'. The 'Query' tab is active, displaying a two-line SQL statement: `1 SELECT * FROM public.users_tr_1` and `2 ORDER BY user_id ASC`. Below the query, there are three tabs: 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, showing a table of results. The table has three columns: 'user\_id [PK] Integer', 'name character varying', and 'surname character varying'. The results are listed in 17 rows, with 'user\_id' values ranging from 1 to 33 in increments of 2, and corresponding 'name' and 'surname' values.

	user_id [PK] Integer	name character varying	surname character varying
1	1	MH	EA
2	3	VB	VF
3	5	JM	SW
4	7	PP	NM
5	9	KA	HM
6	11	OW	UE
7	13	FU	BQ
8	15	SC	RS
9	17	OQ	VS
10	19	FR	BS
11	21	FY	UN
12	23	JB	TH
13	25	NE	ME
14	27	GV	XW
15	29	BV	MR
16	31	AJ	BD
17	33	LF	HV
...	...	...	...

Рисунок 29 –Таблиця «users\_tr\_1» після виконання запиту видалення

Query

Query History

1

SELECT \* FROM public.users\_tr\_2

2

ORDER BY user\_id ASC

Data Output

Messages

Notifications

≡+

▼

▼

	user_id [PK] integer	name character varying	surname character varying
1	1	HU	HY
2	2	LV	OA
3	3	SC	PC
4	4	WJ	AX
5	5	KD	PJ
6	6	HQ	TE
7	7	WP	VF
8	8	UG	HI
9	9	CO	IL
10	10	BI	WF
11	11	NH	AO
12	13	DW	SV
13	15	XW	OD
14	17	MP	VO
15	19	TJ	AA
16	21	XN	NF
17	23	VW	NR

Рисунок 30 – Таблиця «users\_tr\_2» після виконання запиту видалення

Після виконання запиту видалення ми дійсно спростерігаємо що в таблиці «users\_tr\_1» видалились всі рядки де user\_id кратний 2, тоді як в таблиці «users\_tr\_2» видалились рядки, де user\_id більше 10 та кратний 2. Це зображується на рисунках 29 та 30

```

Query  Query History
1  CREATE OR REPLACE FUNCTION ins_tr()
2  RETURNS TRIGGER
3
4  AS
5  $$
6  BEGIN
7      IF (TG_OP = 'INSERT') THEN
8          IF (NEW.user_id % 5 = 0) THEN
9              INSERT INTO users_tr_2(user_id, name, surname) VALUES (NEW.user_id,NE
10             END IF;
11             RETURN NEW;
12         END IF;
13     END;
14 $$
15 LANGUAGE PLPGSQL;
16
17 CREATE TRIGGER ins_tr
18 AFTER INSERT
19 ON "users_tr_1"
20 FOR EACH ROW
21 EXECUTE PROCEDURE ins_tr();

```

Data Output Messages Notifications

CREATE TRIGGER

Query returned successfully in 2 secs 475 msec.

Рисунок 31 – Створення тригера «ins\_tr»

На рисунку 31 зображено створення тригера який виконується після вставки даних в таблицю. Та також записується умова що якщо user\_id рядка таблиці «users\_tr\_1», який вставляється, кратний 5, то такий самий рядок вставляється в таблицю «users\_tr\_2». Для того щоб перевірити роботу тригера вставляємо рядок в таблицю «users\_tr\_1» з значеннями (200, 'oKJ', 'rty'), це зображено на рисунку 32.

```

Query  Query History
1  INSERT INTO users_tr_1(user_id, name, surname) VALUES (200,'oKJ','rty');

```

Рисунок 32 – Запит вставки рядка в таблицю «users\_tr\_1»

Після вставки ми можемо спостерігати що в таблицях «users\_tr\_1» та «users\_tr\_2» на рисунках 33 та 34 ми бачимо що оскільки 200 кратне 5, то такий самий рядок додався і до таблиці «users\_tr\_2».

	<b>user_id</b> [PK] integer	<b>name</b> character varying	<b>surname</b> character varying
68	135	QK	PW
69	137	RH	YJ
70	139	FB	RI
71	141	IE	SW
72	143	KG	KD
73	145	NK	LN
74	147	GT	TF
75	149	MT	XM
76	155	LKJ	UIO
77	200	oKJ	rty

Рисунок 33 – Таблиця «users\_tr\_1» після виконання запиту вставки

	<b>user_id</b> [PK] integer	<b>name</b> character varying	<b>surname</b> character varying
72	133	NV	MX
73	135	MH	MU
74	137	CN	BD
75	139	OW	GO
76	141	SV	KX
77	143	VQ	IH
78	145	IX	PC
79	147	BK	EF
80	149	FA	QJ
81	200	oKJ	rty

Рисунок 33 – Таблиця «users\_tr\_2» після виконання запиту вставки

## Завдання №4

### Рівень ізоляції транзакцій READ COMMITTED

```

SQL Shell (psql)
Server [localhost]:
Database [postgres]: Deanery
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (16.0)
УВАГА: Кодова сторінка консолі? (866) введіть кодову сторінку Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі?
"Нотатки для користувачів Windows" у документації psql.
Введіть "help", щоб отримати допомогу.

Deanery=# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
-----
read committed
(1 Б фмб)

Deanery=# select * from "Catalog";
 catalog_id | catalog_name
-----
3 | Арґументи вчашш ІюґрЕер
2 | Арґументи вчашш фм ЁвЕшяш
1 | Арґументи вчашш фм ІюЕЕхя|рзю
6 | xmas
4 | vitamin
5 | song
(6 Б фм|т)

Deanery=# BEGIN;
BEGIN
Deanery=# UPDATE "Catalog" SET "catalog_name"='valse' where "CatalogID"=4;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=# BEGIN;
BEGIN
Deanery=# UPDATE "Catalog" SET "catalog_name"='rock' where "CatalogID"=4;
UPDATE 1
Deanery=# commit;
COMMIT
  
```

```

SQL Shell (psql)
Server [localhost]:
Database [postgres]: Deanery
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (16.0)
УВАГА: Кодова сторінка консолі? (866) введіть кодову сторінку Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі?
"Нотатки для користувачів Windows" у документації psql.
Введіть "help", щоб отримати допомогу.

Deanery=# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
-----
read committed
(1 Б фмб)

Deanery=# select * from "Catalog";
 catalog_id | catalog_name
-----
3 | Арґументи вчашш ІюґрЕер
2 | Арґументи вчашш фм ЁвЕшяш
1 | Арґументи вчашш фм ІюЕЕхя|рзю
6 | xmas
5 | song
4 | valse
(6 Б фм|т)

Deanery=# select * from "Catalog";
 catalog_id | catalog_name
-----
3 | Арґументи вчашш ІюґрЕер
2 | Арґументи вчашш фм ЁвЕшяш
1 | Арґументи вчашш фм ІюЕЕхя|рзю
6 | xmas
5 | song
4 | rock
(6 Б фм|т)
  
```

Рисунок 35 – Виконання запиту редагування даних в таблиці при рівні ізоляції READ COMMITTED

```

Deanery=# begin;
BEGIN
Deanery=# insert into "Catalog" values (10, 'songl');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=#
  
```

```

Deanery=# select * from "Catalog";
 catalog_id | catalog_name
-----
3 | Арґументи вчашш ІюґрЕер
2 | Арґументи вчашш фм ЁвЕшяш
1 | Арґументи вчашш фм ІюЕЕхя|рзю
6 | lololol
7 | symphony
8 | opera
9 | jk
11 | hjk
10 | songl
(9 Б фм|т)
  
```

Рисунок 36 – Виконання запиту вставки даних в таблицю при рівні ізоляції READ COMMITTED



## Рівень ізоляції транзакцій REPEATABLE READ

На рисунку 38 ми бачимо, що транзакція повторно зчитує деякі дані та виявляє, що вони були змінені іншою транзакцією з моменту першого зчитування. Тоді як на рисунку при рівні ізоляції REPEATABLE READ ми бачимо що при повторному зчитуванні дані не змінюються після виконання запиту іншою транзакцією.

```

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Архивування файлів
2 | Архивування файлів
1 | Архивування файлів
6 | lololol
7 | symphony
8 | opera
9 | jk
10 | song1
(8 E 0)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Архивування файлів
2 | Архивування файлів
1 | Архивування файлів
6 | lololol
7 | symphony
8 | opera
10 | song1
9 | xmas song
(8 E 0)

Deanery=# begin;
BEGIN
Deanery=# UPDATE "Catalog" SET "catalog_name"='xmas song' WHERE "CatalogID"=9;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=#
  
```

Рисунок 38 – Виконання запиту редагування даних в таблиці при рівні ізоляції READ COMMITTED

```

SQL Shell (psql)
Deanery=# begin transaction isolation level repeatable read;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Архивування файлів
2 | Архивування файлів
1 | Архивування файлів
6 | lololol
7 | symphony
8 | opera
10 | song1
9 | xmas song
(8 E 0)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Архивування файлів
2 | Архивування файлів
1 | Архивування файлів
6 | lololol
7 | symphony
8 | opera
10 | song1
9 | xmas song
(8 E 0)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# UPDATE "Catalog" SET "catalog_name"='xmas' WHERE "CatalogID"=6;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=#
  
```

Рисунок 39 – Виконання запиту редагування даних в таблиці при рівні ізоляції REPEATABLE READ



На рисунках 40 та 41 ми спостерігаємо також що при рівні ізоляції READ COMMITTED при повторному зчитуванні, після вставки рядка дані вже змінені, на відміну від ситуації, коли використовується рівень ізоляції REPEATABLE READ.

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Арґрыоу ьечшы йоўрЕер
2 | Арґрыоу ьечшы фы ёёЕшыы
1 | Арґрыоу ьечшы фы йёЕёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
(8 ̲ фь|т)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Арґрыоу ьечшы йоўрЕер
2 | Арґрыоу ьечшы фы ёёЕшыы
1 | Арґрыоу ьечшы фы йёЕёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
11 | sirens
(9 ̲ фь|т)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (11,'sirens');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=#
  
```

Рисунок 40 – Виконання запиту вставки даних в таблицю при рівні ізоляції с

```

SQL Shell (psql)
Deanery=# begin transaction isolation level repeatable read;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Арґрыоу ьечшы йоўрЕер
2 | Арґрыоу ьечшы фы ёёЕшыы
1 | Арґрыоу ьечшы фы йёЕёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
11 | sirens
(9 ̲ фь|т)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
3 | Арґрыоу ьечшы йоўрЕер
2 | Арґрыоу ьечшы фы ёёЕшыы
1 | Арґрыоу ьечшы фы йёЕёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
11 | sirens
(9 ̲ фь|т)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (12,'solo');
INSERT 0 1
Deanery=#
  
```

Рисунок 41 – Виконання запиту вставки даних в таблицю при рівні ізоляції REPEATABLE READ

На рисунках 43 та 44 зображені транзакції де при рівнях READ COMMITTED та REPEATABLE READ виконується запит видалення рядка, як і в випадку редагування та вставки рядка при рівні ізоляції READ COMMITTED при повторному зчитуванні, після вставки рядка дані вже змінені, на відміну від ситуації, коли використовується рівень ізоляції REPEATABLE READ .

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----+-----
3 | Арёрыоу ьечшыш |юўрЁЁр
2 | Арёрыоу ьечшыш фы ёёёшыяш
1 | Арёрыоу ьечшыш фы |юёёхя|рэю
7 | symphony
8 | opera
10 | song1
6 | xmas
11 | sirens
12 | solo
(9 ̲ фь|т)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----+-----
3 | Арёрыоу ьечшыш |юўрЁЁр
2 | Арёрыоу ьечшыш фы ёёёшыяш
1 | Арёрыоу ьечшыш фы |юёёхя|рэю
7 | symphony
8 | opera
10 | song1
6 | xmas
11 | sirens
(8 ̲ фь|т)

```

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=12;
DELETE 1
Deanery=# commit;
COMMIT
Deanery=#

```

Рисунок 43 – Виконання запиту видалення даних з таблиці при рівні ізоляції READ COMMITTED

```

SQL Shell (psql)
Deanery=# begin transaction isolation level repeatable read;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----+-----
3 | Арёрыоу ьечшыш |юўрЁЁр
2 | Арёрыоу ьечшыш фы ёёёшыяш
1 | Арёрыоу ьечшыш фы |юёёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
11 | sirens
12 | solo
(10 ̲ фь|т)

Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----+-----
3 | Арёрыоу ьечшыш |юўрЁЁр
2 | Арёрыоу ьечшыш фы ёёёшыяш
1 | Арёрыоу ьечшыш фы |юёёхя|рэю
7 | symphony
8 | opera
10 | song1
9 | xmas song
6 | xmas
11 | sirens
12 | solo
(10 ̲ фь|т)

```

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=9;
DELETE 1
Deanery=# commit;
COMMIT
Deanery=#

```

Рисунок 44 – Виконання запиту видалення даних з таблиці при рівні ізоляції REPEATABLE READ

## Рівень ізоляції транзакцій **SERIALIZABLE**

На рисунку 45 зображено в Виконання запити вставки даних в таблицю при рівні ізоляції **READ COMMITTED** і ми бачимо що при двох одночасних транзакціях були внесені дані, що може призвести після зміни даних до неточності оскільки в одній з транзакцій можуть відображатись неточні дані, саме такий випадок зображується також на рисунках 47 та 49. Для того щоб виправити цю ситуацію можна використати рівень ізоляції **SERIALIZABLE**, який застосовується на рисунках 46, 48, 50.

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
3 | Арґриуоу ьечшыш ІюґрЕер
2 | Арґриуоу ьечшыш фы ьЕЕшыш
1 | Арґриуоу ьечшыш фы ІюЕЕхяІрзю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(10 ̲ фыІр)

Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (15,'ol');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
3 | Арґриуоу ьечшыш ІюґрЕер
2 | Арґриуоу ьечшыш фы ьЕЕшыш
1 | Арґриуоу ьечшыш фы ІюЕЕхяІрзю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(13 ̲ фыІр)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(12 ̲ фыІр)

Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (17,'ui');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
3 | Арґриуоу ьечшыш ІюґрЕер
2 | Арґриуоу ьечшыш фы ьЕЕшыш
1 | Арґриуоу ьечшыш фы ІюЕЕхяІрзю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(13 ̲ фыІр)
  
```

Рисунок 45 – Виконання запити вставки даних в таблицю при рівні ізоляції **READ COMMITTED**

```

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
3 | Арґриуоу ьечшыш ІюґрЕер
2 | Арґриуоу ьечшыш фы ьЕЕшыш
1 | Арґриуоу ьечшыш фы ІюЕЕхяІрзю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(13 ̲ фыІр)

Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (18,'rt');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=#

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
3 | Арґриуоу ьечшыш ІюґрЕер
2 | Арґриуоу ьечшыш фы ьЕЕшыш
1 | Арґриуоу ьечшыш фы ІюЕЕхяІрзю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(13 ̲ фыІр)

Deanery=# INSERT INTO "Catalog" ("CatalogID", "catalog_name") VALUES (19,'yu');
INSERT 0 1
Deanery=# commit;
COMMIT
Deanery=#
ПОМИЛКА: не вдалося серґанізувати доступ через залежність читання/запису серед транзакцій
ДЕТАЛІ: Reason code: Canceled on identification as a pivot, during commit attempt.
ПІДКАЗКА: Транзакція може завершитися успішно, якщо повторити спробу.
Deanery=#
  
```

Рисунок 46 – Виконання запити вставки даних в таблицю при рівні ізоляції **SERIALIZABLE**

The image shows two side-by-side screenshots of a SQL Shell (psql) window. The left screenshot shows the initial state of the 'Catalog' table with 13 rows. The right screenshot shows the same window after an UPDATE operation has been executed, changing the 'catalog\_name' of the row with 'CatalogID' 13 to 'xs'. The output of the SELECT query is identical in both screenshots, showing the state of the table after the update.

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

Deanery=# UPDATE "Catalog" SET "catalog_name"='sx' WHERE "CatalogID"=13;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
13 | sr
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

Deanery=# UPDATE "Catalog" SET "catalog_name"='xs' WHERE "CatalogID"=13;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

```

Рисунок 47 – Виконання запиту редагування даних в таблиці при рівні ізоляції  
READ COMMITTED

The image shows two side-by-side screenshots of a SQL Shell (psql) window. The left screenshot shows the initial state of the 'Catalog' table with 14 rows. The right screenshot shows the same window after an UPDATE operation has been executed, changing the 'catalog\_name' of the row with 'CatalogID' 14 to 're'. The output of the SELECT query is identical in both screenshots, showing the state of the table after the update.

```

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

Deanery=# UPDATE "Catalog" SET "catalog_name"='do' WHERE "CatalogID"=14;
UPDATE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
14 | lo
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
3 | Арёмоу ьечшыш ЮўрЁёр
2 | Арёмоу ьечшыш фы Ёёшьяш
1 | Арёмоу ьечшыш фы ЮёЁхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 Ё фь|т)

Deanery=# UPDATE "Catalog" SET "catalog_name"='re' WHERE "CatalogID"=14;
ПОМИЛКА: не вдалося сер?ал?зувати доступ через паралельне оновлення
Deanery=# rollback;
ROLLBACK
Deanery=#

```

Рисунок 48 – Виконання запиту редагування даних в таблиці при рівні ізоляції  
SERIALIZABLE

```

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 ё фь|т)

Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=16;
DELETE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(12 ё фь|т)

SQL Shell (psql)
Deanery=# begin;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
15 | ol
16 | po
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(14 ё фь|т)

Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=15;
DELETE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(12 ё фь|т)

```

Рисунок 49 – Виконання запиту видалення даних з таблиці при рівні ізоляції  
READ COMMITTED

```

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(12 ё фь|т)

Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=13;
DELETE 1
Deanery=# commit;
COMMIT
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
17 | ui
18 | rt
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(11 ё фь|т)

SQL Shell (psql)
Deanery=# begin transaction isolation level serializable;
BEGIN
Deanery=# select * from "Catalog";
CatalogID | catalog_name
-----
17 | ui
18 | rt
13 | xs
14 | do
3 | Арёрыоу ьечыш ЁўрЁёр
2 | Арёрыоу ьечыш фы ёёёшяш
1 | Арёрыоу ьечыш фы Ёёёхя|рэю
7 | symphony
8 | opera
10 | songl
6 | xmas
11 | sirens
(12 ё фь|т)

Deanery=# DELETE FROM "Catalog" WHERE "CatalogID"=14;
DELETE 1
Deanery=# commit;
ПOMИЛКА: не вдалося сер?ал?зувати доступ через залежн?сть читання/запису серед транзакц?й
ДЕТАЛ??: Reason code: Canceled on identification as a pivot, during commit attempt.
ПРДКАЗКА: Транзакц?я може завершитися усп?шно, якщо повторити спробу.
Deanery=#

```

Рисунок 50 – Виконання запиту видалення даних з таблиці при рівні ізоляції  
SERIALIZABLE

Telegram: [https://t.me/lumi\\_789](https://t.me/lumi_789)

GitHub: [https://github.com/liudapavlenko/lab\\_2.git](https://github.com/liudapavlenko/lab_2.git)