

**Національний технічний університет України
“Київський політехнічний інститут імені Ігоря
Сікорського ”**

**Факультет прикладної математики
Кафедра системного програмування і спеціалізованих
комп'ютерних систем**

Лабораторна робота №2

**з дисципліни
“Бази даних та засоби управління”**

ТЕМА: «Засоби оптимізації роботи СУБД PostgreSQL»

Група: KB-11

Виконав: Кухта Данило

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.

12	<i>BTree, GIN</i>	<i>after update, insert</i>
----	-------------------	-----------------------------

URL репозиторію <https://github.com/joe1i/DB-labs.git>

Завдання №1

Перетворити модуль “Модель” з шаблону MVC PTP у вигляд об’єктно-реляційної проєкції (ORM)

Модель «сутність-зв’язок» предметної галузі, для проектування бази даних «Платформа для продажу та покупки мистецьких творів»:

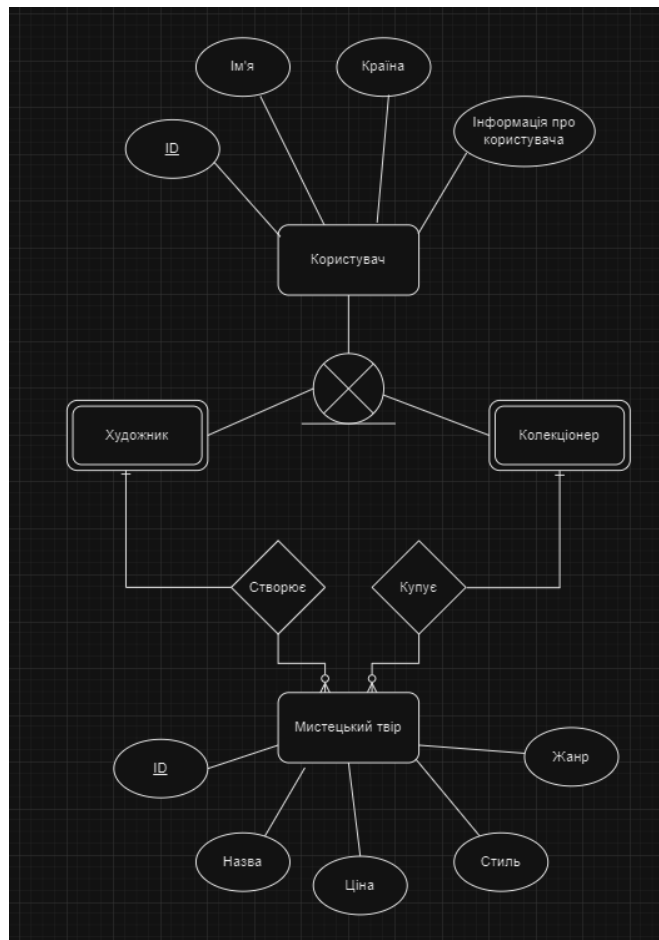


Рис. 1. ER-діаграма побудована за нотацією “Пташина лапка”

Модель “сутність-зв’язок” у схемі бази даних PostgreSQL:

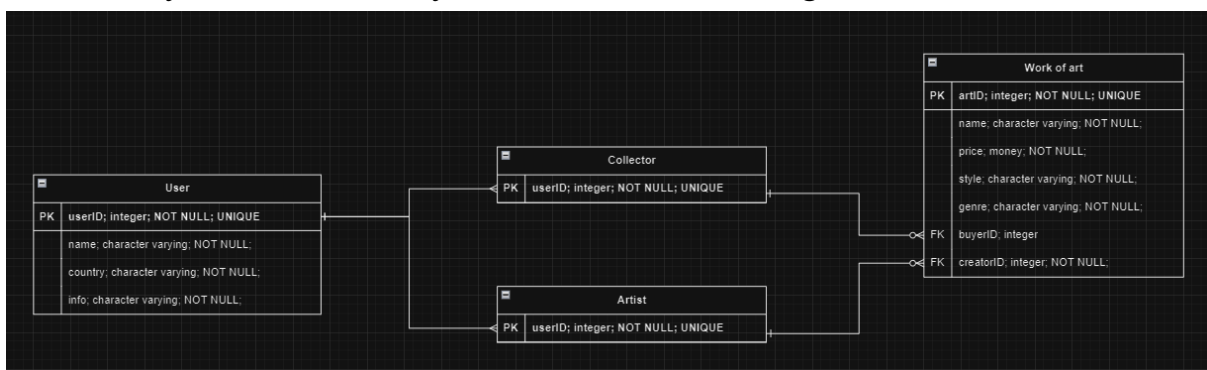


Рис. 2. Схема бази даних у графічному вигляді

Класи ORM і зв'язки між ними:

Сутність **Користувач(User)** у вигляді класу ORM

```
class UserModel(Base):  
    __tablename__ = 'User'  
    userID = Column(Integer, primary_key=True)  
    name = Column(String)  
    country = Column(String)  
    info = Column(String)  
    artist = relationship('ArtistModel', uselist=False, back_populates='user')  
    collector = relationship('CollectorModel', uselist=False, back_populates='user')
```

Сутність **Художник(Artist)** у вигляді класу ORM

```
class ArtistModel(Base):  
    __tablename__ = 'Artist'  
    userID = Column(Integer, ForeignKey('User.userID'), primary_key=True)  
    user = relationship('UserModel', back_populates='artist')  
    art_created = relationship('ArtModel', foreign_keys='ArtModel.creatorID',  
back_populates='creator')
```

Сутність **Колекціонер(Collector)** у вигляді класу ORM

```
class CollectorModel(Base):  
    __tablename__ = 'Collector'  
    userID = Column(Integer, ForeignKey('User.userID'), primary_key=True)  
    user = relationship('UserModel', back_populates='collector')  
    art_purchased = relationship('ArtModel', foreign_keys='ArtModel.buyerID',  
back_populates='buyer')
```

Сутність **Мистецький твір (Work of art)** у вигляді класу ORM

```
class ArtModel(Base):  
    __tablename__ = 'Work of art'  
    artID = Column(Integer, primary_key=True)  
    name = Column(String)  
    price = Column(Integer)  
    style = Column(String)  
    genre = Column(String)  
    buyerID = Column(Integer, ForeignKey('Collector.userID'))  
    creatorID = Column(Integer, ForeignKey('Artist.userID'))  
    buyer = relationship('CollectorModel', foreign_keys=[buyerID],  
back_populates='art_purchased')  
    creator = relationship('ArtistModel', foreign_keys=[creatorID],  
back_populates='art_created')
```

Приклади запитів у вигляді ORM:

Виведення всіх користувачів:

```
def get_all_users(self):  
  
    Session = sessionmaker(bind=self.engine)  
  
    session = Session()  
  
    users = session.query(UserModel.userID, UserModel.name, UserModel.country,  
UserModel.info).all()  
  
    return users
```

Видалення Художника:

```
def remove_artist(self, user_id):  
  
    Session = sessionmaker(bind=self.engine)  
  
    session = Session()  
  
    session.query(ArtModel).filter_by(creatorID=user_id).delete()  
    session.query(ArtistModel).filter_by(userID=user_id).delete()  
  
    session.query(UserModel).filter_by(userID=user_id).delete()  
  
    session.commit()  
  
    session.close()
```

Додавання Колекціонера:

```
def add_collector(self, name, country, info):  
  
    Session = sessionmaker(bind=self.engine)  
  
    session = Session()  
  
    new_user = UserModel(name=name, country=country, info=info)  
    session.add(new_user)  
  
    session.commit()  
  
    new_collector = CollectorModel(user=new_user)  
    session.add(new_collector)  
  
    session.commit()  
  
    return new_collector.userID
```

Редагування даних Мистецького твору:

```
def edit_art(self, art_id, name, price, style, genre, buyerID, creatorID):  
  
    Session = sessionmaker(bind=self.engine)  
  
    session = Session()  
  
    art = session.query(ArtModel).filter_by(artID=art_id).first()  
  
    if art:  
  
        art.name = name  
  
        art.price = price  
  
        art.style = style  
  
        art.genre = genre  
  
        art.creatorID = creatorID  
  
        art.buyerID = buyerID  
  
        session.commit()  
  
    session.close()
```

Загалом перетворено такі запити:

- додавання Користувача;
- виведення всіх Користувачів;
- редагування Користувача;
- додавання Художника;
- виведення всіх Художників;
- виведення конкретного Художника;
- видалення Художника;
- додавання Колекціонера;
- виведення всіх Колекціонерів;
- виведення конкретного Колекціонера;
- видалення Колекціонера;
- додавання Мистецького твору;
- виведення всіх Мистецьких творів;
- виведення конкретного Мистецького твору;
- редагування Мистецького твору;
- видалення Мистецького твору;

Завдання №2

Створити та проаналізувати різні типи індексів у PostgreSQL

BTREE

Створення індексу BTREE в таблиці “Work of art” для стовпця “style”:

```
1 CREATE INDEX btree_index_art_styles ON "Work of art"(style);
2
```

Data Output Messages Notifications

CREATE INDEX

Query returned successfully in 349 msec.

Виведення Мистецьких творів з таблиці “Work of art” (кількість рядків – **50000**) зі стилем ‘Abstract’.

Без індексу:

```
1 EXPLAIN ANALYZE SELECT * FROM "Work of art" WHERE style='Abstract';
```

Data Output Messages Notifications



QUERY PLAN

text

1	Seq Scan on "Work of art" (cost=0.00..1223.00 rows=2503 width=64) (actual time=0.420..24.761 rows=2536 loops...
2	Filter: ((style)::text = 'Abstract')::text)
3	Rows Removed by Filter: 47466
4	Planning Time: 1.634 ms
5	Execution Time: 25.118 ms

З індексом:

```
1 EXPLAIN ANALYZE SELECT * FROM "Work of art" WHERE style='Abstract';
```

Data Output Messages Notifications



QUERY PLAN

text

1	Bitmap Heap Scan on "Work of art" (cost=31.69..660.98 rows=2503 width=64) (actual time=0.253..1.356 rows=2536 loops=1)
2	Recheck Cond: ((style)::text = 'Abstract')::text)
3	Heap Blocks: exact=591
4	-> Bitmap Index Scan on btree_index_art_styles (cost=0.00..31.06 rows=2503 width=0) (actual time=0.173..0.173 rows=2536 loop...
5	Index Cond: ((style)::text = 'Abstract')::text)
6	Planning Time: 0.108 ms
7	Execution Time: 1.453 ms

Виведення Мистецьких творів з таблиці “Work of art” (кількість рядків – **100**) зі стилем ‘Abstract’.

Без індексу:

1	EXPLAIN ANALYZE SELECT * FROM "Work of art" WHERE style='Abstract'
Data Output Messages Notifications	
	QUERY PLAN text
1	Seq Scan on "Work of art" (cost=0.00..600.25 rows=9 width=65) (actual time=0.275..0.287 rows=9 loops...
2	Filter: ((style)::text = 'Abstract'::text)
3	Rows Removed by Filter: 91
4	Planning Time: 0.082 ms
5	Execution Time: 0.301 ms

З індексом:

1	EXPLAIN ANALYZE SELECT * FROM "Work of art" WHERE style='Abstract'
Data Output Messages Notifications	
	QUERY PLAN text
1	Bitmap Heap Scan on "Work of art" (cost=4.21..37.02 rows=9 width=65) (actual time=0.026..0.029 rows=9 loops=1)
2	Recheck Cond: ((style)::text = 'Abstract'::text)
3	Heap Blocks: exact=2
4	-> Bitmap Index Scan on btree_index_art_styles (cost=0.00..4.21 rows=9 width=0) (actual time=0.020..0.020 rows=9 loop...
5	Index Cond: ((style)::text = 'Abstract'::text)
6	Planning Time: 0.086 ms
7	Execution Time: 0.051 ms

GIN

Для демонстрації роботи *GIN*-індекса, створимо таблицю “documents”:

Створення таблиці documents з JSONB-стовпцем:

> artists	Query	Query History
> countries	1	CREATE TABLE documents (
> documents	2	id serial PRIMARY KEY,
> genres	3	data jsonb
> names	4);
> nouns	Data Output	Messages
> styles		Notifications
> surnames	CREATE TABLE	
> tasks	Query returned successfully in 162 msec.	

Заповнення даними:

Query	Query History
1 INSERT INTO documents (data)	
2 SELECT	
3 jsonb_build_object(
4 'title', 'Document ' i,	
5 'content', chr(trunc(65 + random()*26)::int) chr(trunc(65 + random()*26)::int) chr(trunc(65 + random()*26)::int)	
6)	
7 FROM	
8 generate_series(1, 50000) AS i;	
9	
Data Output	Messages
	Notifications
INSERT 0 50000	
Query returned successfully in 495 msec.	

Створення індексу GIN в таблиці “documents” для стовпця “data”:

1 CREATE INDEX gin_index_data ON documents USING GIN(data);		
2		
Data Output	Messages	Notifications
CREATE INDEX		
Query returned successfully in 547 msec.		

Пошук документа в таблиці “documents” (кількість рядків – **50000**), у якого поле data містить JSON-об'єкт з полем "content", яке має значення "SDK".

Без індексу:

1	EXPLAIN ANALYZE SELECT * FROM documents WHERE data @> '{"content": "SDK"}';
Data Output Messages Notifications	
	QUERY PLAN text
1	Seq Scan on documents (cost=0.00..1141.00 rows=5 width=53) (actual time=0.057..18.785 rows=4 loops...
2	Filter: (data @> '{"content": "SDK"}::jsonb)
3	Rows Removed by Filter: 49996
4	Planning Time: 0.110 ms
5	Execution Time: 18.806 ms

З індексом:

1	EXPLAIN ANALYZE SELECT * FROM documents WHERE data @> '{"content": "SDK"}';
Data Output Messages Notifications	
	QUERY PLAN text
1	Bitmap Heap Scan on documents (cost=28.04..46.62 rows=5 width=53) (actual time=0.147..0.152 rows=4 loops=1)
2	Recheck Cond: (data @> '{"content": "SDK"}::jsonb)
3	Heap Blocks: exact=4
4	-> Bitmap Index Scan on gin_index_data (cost=0.00..28.04 rows=5 width=0) (actual time=0.140..0.140 rows=4 loops...
5	Index Cond: (data @> '{"content": "SDK"}::jsonb)
6	Planning Time: 0.103 ms
7	Execution Time: 0.172 ms

Пошук документа в таблиці “documents” (кількість рядків – **100**), у якого поле data містить JSON-об'єкт з полем "content", яке має значення "SDK".

Без індексу:

1	EXPLAIN ANALYZE SELECT * FROM documents WHERE data @> '{"content": "KIK"}'
Data Output Messages Notifications	
	QUERY PLAN text
1	Seq Scan on documents (cost=0.00..3.25 rows=1 width=50) (actual time=0.022..0.042 rows=1 loops...
2	Filter: (data @> '{"content": "KIK"}':jsonb)
3	Rows Removed by Filter: 99
4	Planning Time: 0.089 ms
5	Execution Time: 0.054 ms

З індексом:

1	EXPLAIN ANALYZE SELECT * FROM documents WHERE data @> '{"content": "KIK"}'
2	INDEX SCAN on documents
Data Output Messages Notifications	
	QUERY PLAN text
1	Seq Scan on documents (cost=0.00..3.25 rows=1 width=50) (actual time=0.014..0.028 rows=1 loops...
2	Filter: (data @> '{"content": "KIK"}':jsonb)
3	Rows Removed by Filter: 99
4	Planning Time: 0.107 ms
5	Execution Time: 0.037 ms


! Тут видно, що для 100 рядків індекс не використовувався.

Завдання №3

Розробити тригер бази даних PostgreSQL


Для демонстрації роботи тригера *after insert, update*, створимо таблицю “Elite artists”, у яку будуть додаватись Художники, в яких є хоча б один Мистецький твір з ціною (price) більше 999999:


Створення таблиці “Elite artists”:


 Elite artists

General Columns Advanced Constraints Parameters Security SQL



Name Elite artists



Owner  postgres

Schema  public

Tablespace  no default

Columns +

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
 	artistID	integer v			<input type="checkbox"/>	<input type="checkbox"/>	

	Name	Columns	Referenced Table
 	fk_artistID	(artistID) -> (userID)	public.Artist

```
ALTER TABLE "Elite artists"  
ADD CONSTRAINT unique_artist_id UNIQUE ("artistID");
```

Створення тригера:

```
1 CREATE OR REPLACE FUNCTION after_insert_update_trigger()
2 RETURNS TRIGGER AS $$
3 DECLARE
4     current_row RECORD;
5     my_cursor CURSOR FOR SELECT * FROM "Work of art" WHERE NEW."creatorID" = "Work of art"."creatorID";
6 BEGIN
7     OPEN my_cursor;
8     LOOP
9         FETCH my_cursor INTO current_row;
10        EXIT WHEN NOT FOUND;
11
12        IF current_row.price > 999999 THEN
13            -- Якщо ціна твору > 999999, вставляємо його автора в "Elite artists"
14            INSERT INTO "Elite artists" ("artistID")
15            VALUES (current_row."creatorID")
16            ON CONFLICT ("artistID") DO NOTHING;
17        ELSE
18            -- Якщо ціна <= 999999, перевіряємо, чи існує твір цього ж автора з ціною > 999999
19            IF NOT EXISTS (
20                SELECT 1
21                FROM "Work of art"
22                WHERE "creatorID" = current_row."creatorID"
23                AND price > 999999
24            ) AND EXISTS (
25                SELECT 1
26                FROM "Elite artists"
27                WHERE "artistID" = current_row."creatorID"
28            ) THEN
29                -- Якщо немає такого твору, але в "Elite artists" є автор, вилючаємо його з "Elite artists"
30                DELETE FROM "Elite artists" WHERE "artistID" = current_row."creatorID";
31            END IF;
32        END IF;
33
34    END LOOP;
35    CLOSE my_cursor;
36
37    RETURN NEW;
38 END;
39 $$ LANGUAGE plpgsql;
40
41 CREATE TRIGGER after_insert_update_trigger
42 AFTER INSERT OR UPDATE ON "Work of art"
43 FOR EACH ROW
44 EXECUTE FUNCTION after_insert_update_trigger();
```

Приклад №1

Дані в таблиці “Work of art” до оновлення:

1

SELECT * FROM "Work of art" WHERE "creatorID"=2432272

Data Output

Messages

Notifications

	artID [PK] integer	name character varying (50)	price integer	style character varying (50)	genre character varying (50)	buyerID integer	creatorID integer
1	70634	Radiant Horizon Converging	39814	Realism	Portrait	2476270	2432272
2	81568	Surreal Whisper Awakening	7799	Realism	Landscape	[null]	2432272

Оновлення:

```
1 UPDATE "Work of art" SET price = 1000000 WHERE "artID"=70634;
```

Дані в таблиці “Work of art” після оновлення:

1

SELECT * FROM "Work of art" WHERE "creatorID"=2432272

Data Output

Messages










Notifications


	artID [PK] integer	name character varying (50)	price integer	style character varying (50)	genre character varying (50)	buyerID integer	creatorID integer
1	70634	Radiant Horizon Converging	1000000	Realism	Portrait	2476270	2432272
2	81568	Surreal Whisper Awakening	7799	Realism	Landscape	[null]	2432272

Дані в таблиці “Elite artists” після оновлення:

1 **SELECT** * **FROM** public."Elite artists"

Data Output Messages Notifications



	artistID integer 	
1	2432272	

Приклад №2

Оновлення:

```
1 UPDATE "Work of art" SET price = 1000000 WHERE "artID"=81568;
```

Дані в таблиці “Work of art” після оновлення:

```
1 SELECT * FROM "Work of art" WHERE "creatorID"=2432272
```

	artID [PK] integer	name character varying (50)	price integer	style character varying (50)	genre character varying (50)	buyerID integer	creatorID integer
1	70634	Radiant Horizon Converging	1000000	Realism	Portrait	2476270	2432272
2	81568	Surreal Whisper Awakening	1000000	Realism	Landscape	[null]	2432272

Дані в таблиці “Elite artists” після оновлення:

```
1 SELECT * FROM public."Elite artists"
```

	artistID integer
1	2432272

Приклад №3

Оновлення:

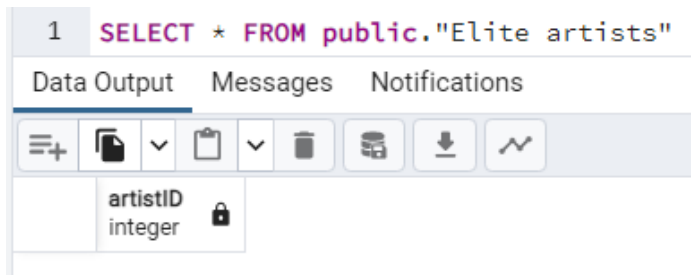
```
UPDATE "Work of art" SET price = 10000 WHERE "artID"=70634;  
UPDATE "Work of art" SET price = 10000 WHERE "artID"=81568;
```

Дані в таблиці “Work of art” після оновлення:

```
48 SELECT * FROM "Work of art" WHERE "creatorID"=2432272
```

	artID [PK] integer	name character varying (50)	price integer	style character varying (50)	genre character varying (50)	buyerID integer	creatorID integer
1	70634	Radiant Horizon Converging	10000	Realism	Portrait	2476270	2432272
2	81568	Surreal Whisper Awakening	10000	Realism	Landscape	[null]	2432272

Дані в таблиці “Elite artists” після оновлення:



Тригер спрацьовує після додавання або оновлення даних у таблиці “Work of art”. Якщо ціна (price) доданого/оновленого Мистецького твору Художника більша 999999, то він додається у таблицю “Elite artists” (якщо його там немає). А якщо ціна менша і у всіх інших творів Художника теж, то він вилучається з таблиці “Elite artists”(якщо він там є).

Завдання №4

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL

Існують різні проблеми, які можуть виникнути при паралельному виконанні транзакцій, і вони є частинами так званих феноменів ізоляції транзакцій. Коротко розглянемо кожен з них:

1. *Втрачене оновлення (Lost Update):*

Це виникає, коли дві транзакції одночасно читають один і той самий запис, а потім кожна з них вносить свої зміни, перезаписуючи зміни іншої транзакції. Як результат, оновлення однієї транзакції може бути втрачено.

2. *"Брудне" читання (Dirty Read):*

Це виникає, коли одна транзакція читає зміни, які були здійснені іншою транзакцією, але яка ще не була закінчена (не була зафіксована). Це може призвести до прочитання "брудних" або непідтверджених даних.

3. *Неповторюване читання (Non-Repeatable Read):*

Це виникає, коли одна транзакція читає той самий запис двічі, і між цими двома читаннями інша транзакція вносить зміни, що призводять до різниці у значеннях, прочитаних обома разами.

4. *Фантомне читання (Phantom Read):*

Це виникає, коли одна транзакція читає набір записів, а інша транзакція вставляє новий запис, який відповідає критеріям пошуку першої транзакції. Як результат, перша транзакція бачить "фантомні" записи, які з'явилися після початку її виконання.

Ці проблеми підсилюють необхідність правильної ізоляції транзакцій для забезпечення консистентності та вірності даних при паралельному виконанні. Рівні ізоляції транзакцій, такі як READ COMMITTED, REPEATABLE READ і SERIALIZABLE, намагаються уникнути або пом'якшити ці феномени забезпеченням різних рівнів видимості та блокування даних.

READ COMMITTED

"READ COMMITTED" - це рівень ізоляції в транзакційних системах баз даних. Під час використання цього рівня, транзакція може читати лише ті дані, які вже були збережені (committed) іншими транзакціями. Це означає, що транзакція не буде бачити змін, які виконуються в інших транзакціях, які ще не завершилися.

Одна з особливостей "READ COMMITTED" полягає в тому, що вона не блокує інші транзакції від читання тих самих даних. Інші транзакції можуть читати ті ж самі рядки, але вони не будуть бачити змін, які виконуються в поточній транзакції, поки та не буде фіксована (committed).

➤ Arts Platform/postgres@PostgreSQL 15	✕	➤ Arts Platform/postgres@PostgreSQL 15
Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 (5 rows)		Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 (5 rows)

Оновлюємо дані в таблиці "Work of art" (далі спрацьовує тригер з минулого пункту та додає Мистецький твір у таблицю "Elite artists"):

➤ Arts Platform/postgres@PostgreSQL 15	✕	➤ Arts Platform/postgres@PostgreSQL 15
Arts Platform=# BEGIN; BEGIN Arts Platform=# UPDATE "Work of art" SET price = 1000000 Arts Platform-## WHERE "artID"=64103; UPDATE 1 Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 (6 rows)		Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 (5 rows)

>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# COMMIT; COMMIT	>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 (6 rows)
--	--

Зміни відобразились у правому вікні тільки після команди “COMMIT”.

REPEATABLE READ

Під час використання цього рівня, транзакція блокує доступ інших транзакцій до даних, які вона читає, до завершення власної роботи. Це забезпечує стабільність даних протягом усього життєвого циклу транзакції, уникаючи конфліктів із іншими операціями. Однак цей рівень ізоляції не гарантує відсутності фантомних рядків.

>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 (7 rows)	>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 (7 rows)
--	--

Оновлюємо дані у лівому вікні:

>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# UPDATE "Work of art" SET price = 1000000 Arts Platform=# WHERE "artID"=64106; UPDATE 1 Arts Platform=#	>_ Arts Platform/postgres@PostgreSQL 15 Arts Platform=# Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 (7 rows)
--	--

Arts Platform/postgres@PostgreSQL 15	Arts Platform/postgres@PostgreSQL 15
Arts Platform=# COMMIT; COMMIT Arts Platform=#	Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 (7 rows)

Після команди “COMMIT” у лівому вікні, дані у правому не змінилися.

Arts Platform/postgres@PostgreSQL 15	Arts Platform/postgres@PostgreSQL 15
Arts Platform=# Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 2456812 (8 rows) Arts Platform=#	Arts Platform=# Arts Platform=# COMMIT; COMMIT Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 2456812 (8 rows)

Після команди “COMMIT” у правому вікні, дані у ньому змінилися.

SERIALIZABLE

Рівень ізоляції SERIALIZABLE є найвищим рівнем ізоляції транзакцій в базі даних. Цей рівень надає найбільш строгі гарантії щодо одночасності та консистентності даних, але при цьому може призводити до більшого використання ресурсів та зниження продуктивності через велику кількість блокувань.

Arts Platform/postgres@PostgreSQL 15	Arts Platform/postgres@PostgreSQL 15
Arts Platform=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 (6 rows)	Arts Platform=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 (6 rows)

Оновлюємо дані перше у лівому вікні, потім у правому:

Arts Platform/postgres@PostgreSQL 15	Arts Platform/postgres@PostgreSQL 15
2450449 2439656 (6 rows)	2439450 2436896 2450449 2439656 (6 rows)
Arts Platform=# UPDATE "Work of art" SET price = 1000000 Arts Platform=# WHERE "artID"=64104; UPDATE 1 Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 2432358 (7 rows)	Arts Platform=# SELECT * FROM "Elite artists"; artistID ----- 2432272 2447264 2439450 2436896 2450449 2439656 (6 rows)
Arts Platform=#	Arts Platform=# UPDATE "Work of art" SET price = 1000000 Arts Platform=# WHERE "artID"=64104;

Видно, що при оновленні даних у правому вікні, це вікно блокується, поки у лівому вікні не будуть зафіксовані зміни.

Arts Platform/postgres@PostgreSQL 15	Arts Platform/postgres@PostgreSQL 15
2432358 (7 rows)	2439656 (6 rows)
Arts Platform=# COMMIT; COMMIT Arts Platform=# Arts Platform=#	Arts Platform=# UPDATE "Work of art" SET price = 1000000 Arts Platform=# WHERE "artID"=64104; ПОМИЛКА: не вдалося сер?ал?зувати доступ через паралельне оновлення Arts Platform=!#