



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 **Бази даних і засоби управління**

на тему: “ Засоби оптимізації роботи СУБД PostgreSQL”

Виконав: студент групи КВ-23

Перетяцько Богдан

Телеграм: <https://t.me/akk3rm4n>

Перевірив: Петрашенко А.В.

Київ – 2024

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

Завдання роботи полягає у наступному:

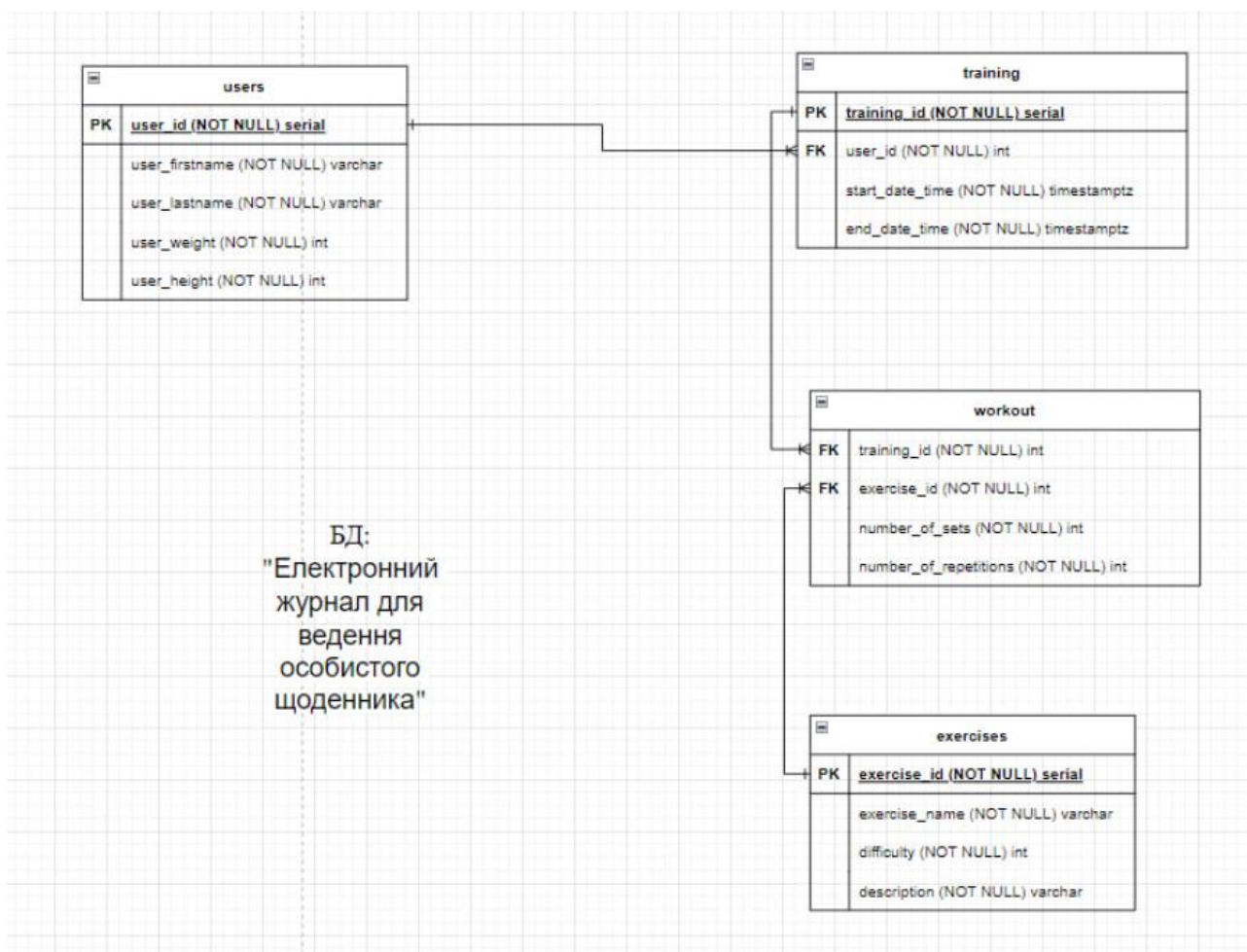
1. Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

## Варіант 21

21	<i>Btree, Hash</i>	<i>before delete, update</i>
----	--------------------	------------------------------

### Виконання роботи

Логічна модель (схема) “Система управління та аналізу даних в галузі робототехніки”



## Класи ORM

```
class User(Base):
    __tablename__ = 'users'
    user_id = Column(Integer, primary_key=True)
    user_firstname = Column(String, nullable=False)
    user_lastname = Column(String, nullable=False)
    user_weight = Column(Integer, nullable=False)
    user_height = Column(Integer, nullable=False)

class Exercise(Base):
    __tablename__ = 'exercises'
    exercise_id = Column(Integer, primary_key=True)
    exercise_name = Column(String, nullable=False)
    difficulty = Column(Integer, nullable=False)
    description = Column(String, nullable=False)

class Training(Base):
    __tablename__ = 'training'
    training_id = Column(Integer, primary_key=True)
    start_date_time = Column(DateTime, nullable=False)
    end_date_time = Column(DateTime, nullable=False)
    user_id = Column(Integer, ForeignKey('users.user_id'))

class Workout(Base):
    __tablename__ = 'workout'
    workout_id = Column(Integer, primary_key=True)
    training_id = Column(Integer, ForeignKey('training.training_id'),
nullable=False)
    exercise_id = Column(Integer, ForeignKey('exercises.exercise_id'),
nullable=False)
    number_of_sets = Column(Integer, nullable=False)
    number_of_repetitions = Column(Integer, nullable=False)
```

## Приклади запитів у вигляді ORM

Фрагмент програми для введення даних в таблицю:

```
def add_data(table_name, data_dict):
    model = get_model(table_name)
    new_record = model(**data_dict)
    session.add(new_record)
    session.commit()
    return new_record
```

Фрагмент програми для видалення даних з таблиці:

```
def delete_data(table_name, record_id):
    model = get_model(table_name)

    try:
        record = session.query(model).get(record_id)

        if record is None:
            raise ValueError(f"Запис із ID {record_id} не знайдено у таблиці {table_name}.")

        session.delete(record)
        session.commit()
    except Exception as e:
        session.rollback()
        raise e
```

Фрагмент програми для оновлення даних в таблиці:

```
def update_data(table_name, record_id, updates):
    model = get_model(table_name)

    try:
        record = session.query(model).get(record_id)

        if record is None:
            raise ValueError(f"Запис із ID {record_id} не знайдено у таблиці {table_name}.")

        for column, value in updates.items():
            if not hasattr(record, column):
                raise ValueError(f"Колонка '{column}' не існує в таблиці {table_name}.")
            setattr(record, column, value)

        session.commit()
    except Exception as e:
        session.rollback()
        raise e
```

Фрагмент програми для генерування даних в таблиці:

```
def generate_data(table, rows_count): # генерує і додає дані в таблицю
    data_dict = {}
    model = get_model(table)
    try:
        column_types = get_table_columns_and_types(table, engine)
        for i in range(rows_count):
            for key, value in column_types.items():
                reference_table_name = get_referred_table_by_column(key, table, engine)

                if reference_table_name is not None:
                    data_dict[key] = generate_random_value(value, reference_table_name, key)
                else:
                    data_dict[key] = generate_random_value(value)
```

```
        new_record = model(**data_dict)
        session.add(new_record)
        session.commit()
    except Exception as e:
        session.rollback()
        raise e
```

Фрагмент програми для отримання імен та типів стовпчиків таблиці:

```
def get_table_columns_and_types(table_name, engine):
    inspector = inspect(engine)

    columns = inspector.get_columns(table_name)
    pk_columns = inspector.get_pk_constraint(table_name)['constrained_columns']

    column_types = {column['name']: str(column['type']) for column in columns if
column['name'] not in pk_columns}
    return column_types
```

Фрагмент програми для отримання випадкового значення

```
def generate_random_value(column_type, reference_table_name=None,
column_name=None):
    column_type = column_type.lower()

    if column_name:
        return get_random_foreign_key_value(reference_table_name, column_name)

    if column_type == 'text':
        return ''.join(random.choices(string.ascii_letters, k=10))

    elif column_type == 'integer':
        return random.randint(1, 100)

    elif column_type == 'timestamp':
        now = datetime.now()
        random_days = random.randint(0, 30)
        random_seconds = random.randint(0, 86400)
        return now - timedelta(days=random_days, seconds=random_seconds)

    else:
        raise ValueError(f"Невідомий тип колонки: {column_type}")
```

Фрагмент для отримання випадкового значення з таблиці на яку посилається зовнішній ключ

```
def get_random_foreign_key_value(table_name, column_name):
    result = session.execute(text(f"SELECT {column_name} FROM {table_name}"))
    values = [row[0] for row in result.fetchall()]

    if not values:
        raise ValueError(f"Немає доступних значень у таблиці {table_name} для
колонки {column_name}.")

    # Повертаємо випадкове значення
    return random.choice(values)
```

Фрагмент для отримання назви таблиці на яку посилається зовнішній ключ

```
def get_referred_table_by_column(column_name, table_name, engine):
    inspector = inspect(engine)

    foreign_keys = inspector.get_foreign_keys(table_name)

    for fk in foreign_keys:

        if column_name in fk['constrained_columns']:
            referred_table = fk['referred_table']
            return referred_table
```

```
return None
```

## Створення індексів

### BTree

```
CREATE INDEX index_btree ON users USING BTree (user_weight)
```


### Hash

```
CREATE INDEX index_hash ON users USING Hash (user_weight)
```

## Приклад 1: Просте фільтрування

Без індексів:

Query		Query History
1	EXPLAIN ANALYSE	
2	SELECT * FROM users	
3	WHERE user_weight BETWEEN 10 AND 50	
4		

	QUERY PLAN	
	text	
1	Seq Scan on users (cost=0.00..2334.21 rows=40852 width=32) (actual time=0.010..6.306 rows=41028 loops...	
2	Filter: ((user_weight >= 10) AND (user_weight <= 50))	
3	Rows Removed by Filter: 58986	
4	Planning Time: 0.970 ms	
5	Execution Time: 7.223 ms	

3 индексом BTree:

Query		Query History
1	EXPLAIN ANALYSE	
2	SELECT * FROM users	
3	WHERE user_weight BETWEEN 10 AND 50	
4		
QUERY PLAN		
text		
1	Bitmap Heap Scan on users (cost=571.03..2017.81 rows=40852 width=32) (actual time=1.088..4.217 rows=41028 loops=1)	
2	Recheck Cond: ((user_weight >= 10) AND (user_weight <= 50))	
3	Heap Blocks: exact=834	
4	-> Bitmap Index Scan on index_btree (cost=0.00..560.81 rows=40852 width=0) (actual time=1.027..1.027 rows=41028 loop...	
5	Index Cond: ((user_weight >= 10) AND (user_weight <= 50))	
6	Planning Time: 0.143 ms	
7	Execution Time: 5.181 ms	



З індексом Hash:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT * FROM users
3	WHERE user_weight BETWEEN 10 AND 50
4	

	QUERY PLAN	text	
1	Seq Scan on users	(cost=0.00..2334.21 rows=40852 width=32) (actual time=0.012..6.396 rows=41028 loops...	
2	Filter: ((user_weight >= 10) AND (user_weight <= 50))		
3	Rows Removed by Filter: 58986		
4	Planning Time: 0.070 ms		
5	Execution Time: 7.304 ms		

Як бачимо індекс у цьому запиті не використовувався тому що Хеш-коди не мають порядку, тому неможливо визначити діапазон значень.

## Приклад 2: Агрегатні функції

Без індексів:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT SUM(user_weight) FROM users
3	WHERE user_weight = 10
4	

	QUERY PLAN	text	
1	Aggregate	(cost=2086.58..2086.59 rows=1 width=8) (actual time=4.686..4.687 rows=1 loops=1)	
2	-> Seq Scan on users	(cost=0.00..2084.18 rows=960 width=4) (actual time=0.021..4.646 rows=1022 loops...	
3	Filter: (user_weight = 10)		
4	Rows Removed by Filter: 98992		
5	Planning Time: 0.419 ms		
6	Execution Time: 4.700 ms		

З індексом BTree:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT SUM(user_weight) FROM users
3	WHERE user_weight = 10
4	

	QUERY PLAN text	
1	Aggregate (cost=27.49..27.50 rows=1 width=8) (actual time=0.277..0.278 rows=1 loops=1)	
2	-> Index Only Scan using index_btree on users (cost=0.29..25.09 rows=960 width=4) (actual time=0.188..0.245 rows=1022 loop...	
3	Index Cond: (user_weight = 10)	
4	Heap Fetches: 0	
5	Planning Time: 0.589 ms	
6	Execution Time: 0.294 ms	

З індексом Hash:

Query Query History

```
1 1 ✓ EXPLAIN ANALYSE
2 SELECT SUM(user_weight) FROM users
3 WHERE user_weight = 10
4
```


	QUERY PLAN text	
1	Aggregate (cost=924.77..924.78 rows=1 width=8) (actual time=0.623..0.623 rows=1 loops=1)	
2	-> Bitmap Heap Scan on users (cost=35.44..922.37 rows=960 width=4) (actual time=0.122..0.589 rows=1022 loops=1)	
3	Recheck Cond: (user_weight = 10)	
4	Heap Blocks: exact=581	
5	-> Bitmap Index Scan on index_hash (cost=0.00..35.20 rows=960 width=0) (actual time=0.078..0.078 rows=1022 loop...	
6	Index Cond: (user_weight = 10)	
7	Planning Time: 0.737 ms	
8	Execution Time: 0.642 ms	

## Приклад 3: Групування

Без індексів:


Query Query History

```
1 1 ✓ EXPLAIN ANALYSE
2 SELECT sum(user_height) FROM users
3 WHERE user_weight = 40
4 GROUP BY user_weight
```

	QUERY PLAN	
	text	
1	GroupAggregate (cost=0.00..2086.59 rows=1 width=12) (actual time=4.900..4.901 rows=1 loops=1)	
2	-> Seq Scan on users (cost=0.00..2084.18 rows=960 width=8) (actual time=0.012..4.856 rows=1017 loops=1)	
3	Filter: (user_weight = 40)	
4	Rows Removed by Filter: 98997	
5	Planning Time: 0.448 ms	
6	Execution Time: 4.917 ms	

### З индексом BTree:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT sum(user_height) FROM users
3	WHERE user_weight = 40
4	GROUP BY user_weight

	QUERY PLAN	
	text	
1	GroupAggregate (cost=11.73..901.08 rows=1 width=12) (actual time=0.615..0.616 rows=1 loops=1)	
2	-> Bitmap Heap Scan on users (cost=11.73..898.67 rows=960 width=8) (actual time=0.108..0.567 rows=1017 loops=1)	
3	Recheck Cond: (user_weight = 40)	
4	Heap Blocks: exact=602	
5	-> Bitmap Index Scan on index_btree (cost=0.00..11.49 rows=960 width=0) (actual time=0.064..0.064 rows=1017 loop=1)	
6	Index Cond: (user_weight = 40)	
7	Planning Time: 0.656 ms	
8	Execution Time: 0.637 ms	

### З индексом Hash:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT sum(user_height) FROM users
3	WHERE user_weight = 40
4	GROUP BY user_weight

	QUERY PLAN text	
1	GroupAggregate (cost=35.44..924.78 rows=1 width=12) (actual time=0.610..0.610 rows=1 loops=1)	
2	-> Bitmap Heap Scan on users (cost=35.44..922.37 rows=960 width=8) (actual time=0.120..0.570 rows=1017 loops=1)	
3	Recheck Cond: (user_weight = 40)	
4	Heap Blocks: exact=602	
5	-> Bitmap Index Scan on index_hash (cost=0.00..35.20 rows=960 width=0) (actual time=0.077..0.077 rows=1017 loop...	
6	Index Cond: (user_weight = 40)	
7	Planning Time: 0.390 ms	
8	Execution Time: 0.631 ms	

## Приклад 4: Сортвання

Без індексів:

Query	Query History
1	EXPLAIN ANALYSE
2	SELECT * FROM users
3	WHERE user_weight = 40
4	ORDER BY user_height asc

	QUERY PLAN text	
1	Sort (cost=2131.73..2134.13 rows=960 width=32) (actual time=4.868..4.927 rows=1017 loops=1)	
2	Sort Key: user_height	
3	Sort Method: quicksort Memory: 88kB	
4	-> Seq Scan on users (cost=0.00..2084.18 rows=960 width=32) (actual time=0.011..4.762 rows=1017 loops=1)	
5	Filter: (user_weight = 40)	
6	Rows Removed by Filter: 98997	
7	Planning Time: 0.403 ms	
8	Execution Time: 4.963 ms	

З індексом BTree:

Query Query History	
1	EXPLAIN ANALYSE
2	SELECT * FROM users
3	WHERE user_weight = 40
4	ORDER BY user_height asc
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Sort (cost=946.22..948.62 rows=960 width=32) (actual time=0.633..0.661 rows=1017 loops=1)
2	Sort Key: user_height
3	Sort Method: quicksort Memory: 88kB
4	-> Bitmap Heap Scan on users (cost=11.73..898.67 rows=960 width=32) (actual time=0.126..0.540 rows=1017 loops=1)
5	Recheck Cond: (user_weight = 40)
6	Heap Blocks: exact=602
7	-> Bitmap Index Scan on index_btree (cost=0.00..11.49 rows=960 width=0) (actual time=0.082..0.082 rows=1017 loop...
8	Index Cond: (user_weight = 40)
9	Planning Time: 0.564 ms
10	Execution Time: 0.701 ms

3 индексом Hash:

Query Query History	
1	EXPLAIN ANALYSE
2	SELECT * FROM users
3	WHERE user_weight = 40
4	ORDER BY user_height asc
	<div>QUERY PLAN</div> <div>text</div> <div></div>
1	Sort (cost=969.93..972.33 rows=960 width=32) (actual time=0.951..0.989 rows=1017 loops=1)
2	Sort Key: user_height
3	Sort Method: quicksort Memory: 88kB
4	-> Bitmap Heap Scan on users (cost=35.44..922.37 rows=960 width=32) (actual time=0.128..0.814 rows=1017 loops=1)
5	Recheck Cond: (user_weight = 40)
6	Heap Blocks: exact=602
7	-> Bitmap Index Scan on index_hash (cost=0.00..35.20 rows=960 width=0) (actual time=0.073..0.073 rows=1017 loop...
8	Index Cond: (user_weight = 40)
9	Planning Time: 0.511 ms
10	Execution Time: 1.033 ms

Результати:

Як бачимо на прикладах вище майже завжди найшвидшим варіантом був **BTree** індекс. По ідеї найшвидшим у всіх випадках крім першого (вибірка з діапазону) мав бути **Hash index**. Але через те що в мене було замало рекордів в таблиці найшвидше запити виконував саме **Btree** індекс. Точно можна сказати що для мого розміру таблиці навіть індекс **Hash** виконує запити набагато швидше ніж якщо виконання проходить не використовуючи індекси.

## **Btree-індекс (Balanced Tree)**

Гарно працює з:

1. Діапазонні запити
2. Сортування
3. Точний пошук
4. Комбіновані запити

## **Hash-індекс**

Гарно працює з:

1. Точний пошук за ключем
2. Запити з оператором рівності
3. Швидкий доступ до окремих записів

## Розробка тригерів

Створення функції для тригера:

```
CREATE OR REPLACE FUNCTION delete_update_trigger()
RETURNS TRIGGER AS $$
DECLARE
    rec RECORD;
BEGIN
    IF TG_OP = 'DELETE' THEN
        RAISE NOTICE 'deleted user with ID: %', OLD.user_id;
        IF EXISTS (SELECT 1 FROM training WHERE user_id = OLD.user_id) THEN
            FOR rec IN
                SELECT * FROM training WHERE user_id = OLD.user_id
            LOOP
                RAISE NOTICE 'delete record with training_id: % for user_id = %',
                    rec.training_id, rec.user_id;
            END LOOP;
        END IF;
        RETURN OLD;
    ELSIF TG_OP = 'UPDATE' THEN
        RAISE NOTICE 'Updated user with ID: %', NEW.user_id;
        RETURN NEW;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'An error occurred in delete_update_trigger trigger: %s', SQLERRM;
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Створення тригера для таблиці фабрика:

```
CREATE TRIGGER delete_update_trigger_t
BEFORE DELETE OR UPDATE ON users
FOR EACH ROW
```

```
EXECUTE FUNCTION delete_update_trigger();
```

Видалення рядка з таблиці users:

```
3
4 delete from users
5 where user_id = 85741
```

Data Output Messages Notifications

ПОВІДОМЛЕННЯ: deleted user with ID: 85741

ПОВІДОМЛЕННЯ: delete record with training\_id: 12 for user\_id = 85741

DELETE 1

Query returned successfully in 109 msec.

Звідси видно, що тригер спрацював, оскільки вивелося повідомлення про видалення.

Зміна рядка в таблиці users:

Query Query History

```
1 UPDATE users
2 SET user_firstname = 'Bohdan',
3     user_lastname = 'Peretatiako',
4     user_weight = '63',
5     user_height = '179'
6 WHERE user_id = 330
7
```

Data Output Messages Notifications

ПОВІДОМЛЕННЯ: Updated user with ID: 330

UPDATE 1

Query returned successfully in 31 msec.

Звідси видно, що тригер спрацював, оскільки вивелося повідомлення про зміну.



```
7
8 select * from users
9 where user_id = 330
10
```

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>					
	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
1	330	Bohdan	Peretatiako	63	179

## Використання рівнів ізоляції

### READ COMMITTED

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
1	1	John	Doe	75	180
2	4	bohdan	peretiatko	13	14
3	6	ikki	dsa	12	13
4	16	XLASX	QMMBS	61	47
5	17	SCOGQ	MAAVC	98	15
6	18	IMYRC	JVHBL	41	9
7	19	TXXWA	NSCYI	71	54
8	20	AKTZV	AQTGP	34	63
9	22	Max	Isagi	50	150
10	24	asd	dsa	15	15
11	25	xYjVX753D8	OwY8cGdcLo	14	61
12	26	wWeDZ8WmVp	fi10L34i4J	39	15
13	27	SgIIVloBqa	RhunAziokX	20	35

## Вікно 1:

Query	Query History
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2	
3	BEGIN;
4	
5	UPDATE users
6	SET user_firstname = 'Bohdan',
7	user_lastname = 'Peretatiako',
8	user_weight = '63',
9	user_height = '179'
10	WHERE user_id = 27

---

ПОВІДОМЛЕННЯ: Updated user with ID: 27  
UPDATE 1

Query returned successfully in 60 msec.

## Вікно 2:

Query	Query History
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2	
3	BEGIN;
4	
5	SELECT * FROM users WHERE user_id = 27

Data Output Messages Notifications						
<div> <div>☰+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div>						
	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer	
1	27	SgllVloBqa	RhunAziokX	20	35	

Звідси видно, що без фіксації змін у першому вікні, у другому вікні немає змін і запис залишився таким самим.

Вікно 1:

Query Query History	
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2	
3	COMMIT;

Data Output Messages Notifications	
COMMIT	
Query returned successfully in 69 msec.	

Вікно 2:

Query Query History	
1	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2	
3	SELECT * FROM users WHERE user_id = 27

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
1	27	Bohdan	Peretatiako	63	179

Тепер видно, що після фіксації у першому вікні, у другому вікні відображаються зміни, запис оновлено.

## REPEATABLE READ

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
5	17	SCOGQ	MAAVC	98	15
6	18	IMYRC	JVHBL	41	9
7	19	TXXWA	NSCYI	71	54
8	20	AKTZV	AQTGP	34	63
9	22	Max	Isagi	50	150
10	24	asd	dsa	15	15
11	25	xYjVX753D8	OwY8cGdcLo	14	61
12	26	wWeDZ8WmVp	fi10L34i4J	39	15
13	27	Bohdan	Peretatiako	63	179
14	28	fzqQVWkTgT	kjZitwYzHZ	65	33
15	29	SUfNAJWYfH	aklHzYGyYu	99	37
16	30	YFBBrwrwfv	VmhekLABCy	13	84
17	31	SiwhRaPdDi	VhfUpRWerb	89	8

Вікно 1:

Query	Query History
1	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2	
3	BEGIN;
4	
5	SELECT * FROM users WHERE user_id = 30;
6	

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
1	30	YFBBrrwfv	VmhekLABCy	13	84

Вікно 2:

```

3 BEGIN;
4
5 UPDATE users
6 SET user_firstname = 'Bohdan',
7     user_lastname = 'Peretatiako',
8     user_weight = '63',
9     user_height = '179'
10 WHERE user_id = 30;
11
12 COMMIT;

```

Data Output	Messages	Notifications
ПОВІДОМЛЕННЯ: Updated user with ID: 30 COMMIT  Query returned successfully in 30 msec.		

Змінюємо таблицю з фіксацією

Вікно 1:

```

4
5 SELECT * FROM users WHERE user_id = 30;
6
7 COMMIT;

```

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
1	30	YFBBrrwfv	VmhekLABCy	13	84

Видно, що навіть після фіксації змін у другому вікні, у першому вікні не відображаються зміни.

## SERIALIZABLE

	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
9	22	Max	Isagi	50	150
10	24	asd	dsa	15	15
11	25	xYjVX753D8	OwY8cGdcLo	14	61
12	26	wWeDZ8WmVp	fi10L34i4J	39	15
13	27	Bohdan	Peretatiako	63	179
14	28	fzqQVWkTgT	kjZitwYzHZ	65	33
15	29	SUfNAJWYfH	akIHzyGyYu	99	37
16	30	Peretiatko	Bohdan	55	104
17	31	SiwhRaPdDi	VhfUpRWerb	89	8
18	32	fYOnmWwwwS	xuDesDrvqA	37	27
19	33	GcJNUAZAXx	GJPmzwuelF	36	58
20	34	vOrWYdDNZS	zqdpHSAJxG	77	10
21	35	ypLOUSHjqD	otWvjYmmrB	20	60

## Вікно 1:

Query

Query History

1

2

3

4

5

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;

DELETE FROM users WHERE user\_id = 32;

Data Output

Messages

Notifications

ПОВІДОМЛЕННЯ: deleted user with ID: 32

DELETE 1

Query returned successfully in 31 msec.

Видаляємо рядок без фіксації зміни.

## Вікно 2:

Query

Query History

1

2

3

4

5

6

7

8

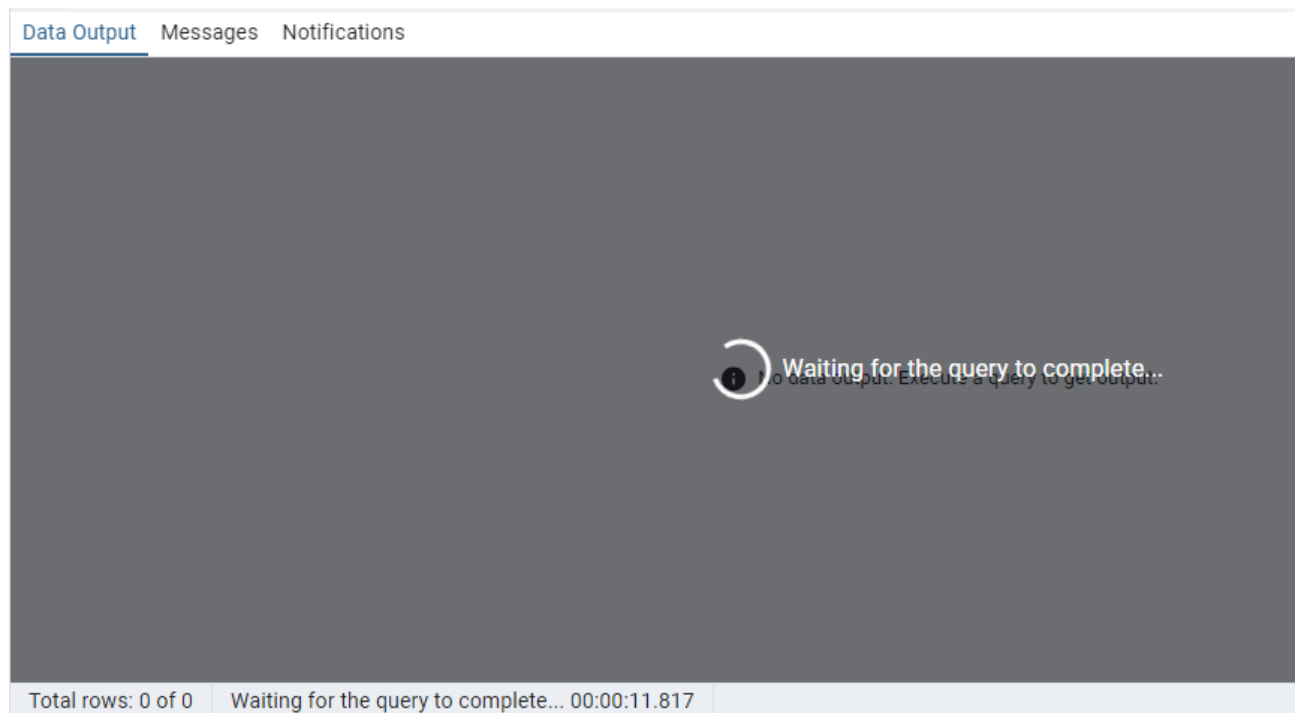
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;

▼ INSERT INTO users (user\_id, user\_firstname, user\_lastname, user\_weight, user\_height)

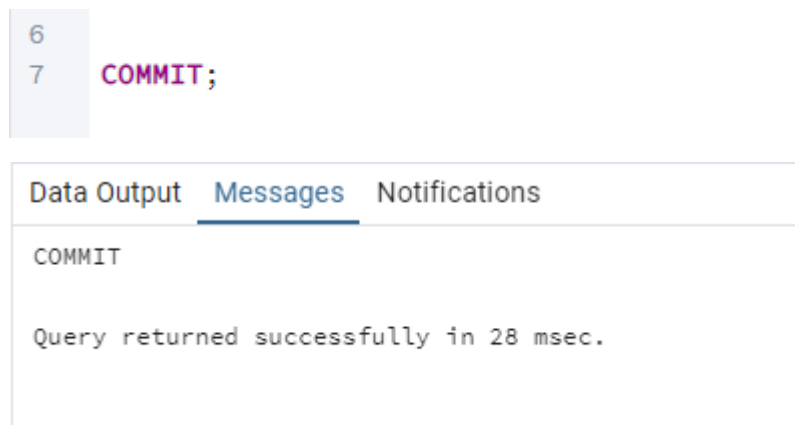
VALUES (32, 'firstname', 'lastname', 11, 11);

COMMIT;



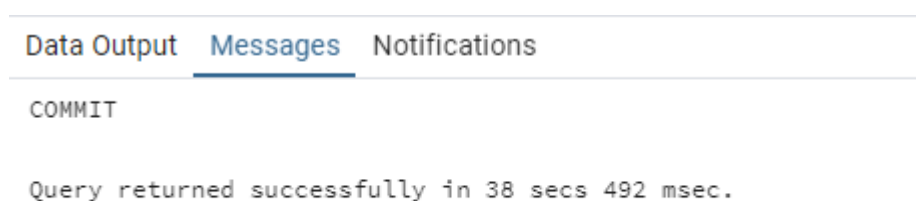
Як видно без фіксації зміни у першому вікні, у другому вікні запит не може виконатися.

Вікно 1:



Робимо фіксацію зміни.

Вікно 2:





	user_id [PK] integer	user_firstname text	user_lastname text	user_weight integer	user_height integer
9	22	Max	Isagi	50	150
10	24	asd	dsa	15	15
11	25	xYjVX753D8	OwY8cGdcLo	14	61
12	26	wWeDZ8WmVp	fi10L34i4J	39	15
13	27	Bohdan	Peretatiako	63	179
14	28	fzqQVWkTgT	kjZitwYzHZ	65	33
15	29	SUfNAJWYfH	akIHzyGyYu	99	37
16	30	Peretiatko	Bohdan	55	104
17	31	SiwhRaPdDi	VhfUpRWerb	89	8
18	32	firstname	lastname	11	11
19	33	GcJNUAZAXx	GJPmzwuelF	36	58
20	34	vOrWYdDNZS	zqdpHSAJxG	77	10
21	35	ypLOUSHjqD	otWvjYmmrB	20	60
22	36	WfzmWbxMBw	EEVTmRzPjy	81	87

Тепер у другому вікні запит зміг виконатися, оскільки відбулася фіксація змін у першому вікні.

## READ COMMITTED:

- **Підходить для:**
  - Сценаріїв, де важливіша швидкість транзакцій, ніж абсолютна консистентність даних.
  - Застосунків, які можуть працювати з даними, що могли змінитися іншими транзакціями, поки поточна транзакція ще не завершена.
  - Наприклад: системи аналітики в реальному часі, де допустимі незначні відхилення в даних.
- **Особливості:**
  - Уникає "брудних читань", але допускає "неповторювані читання" (доступ до змінених даних).

## REPEATABLE READ:

- **Підходить для:**
  - Сценаріїв, де важливо, щоб результати багаторазового читання залишалися незмінними під час виконання транзакції.
  - Застосунків, які працюють із даними в межах однієї транзакції та вимагають відсутності "неповторюваних читань".
  - Наприклад: системи обліку запасів, де критично уникати змін у проміжку транзакцій.
- **Особливості:**

- Гарантує стабільність прочитаних даних, але допускає "фантомні читання" (додавання нових рядків).

## **SERIALIZABLE:**

- **Підходить для:**

- Сценаріїв, де необхідна максимальна консистентність даних та повна ізоляція транзакцій.
- Критично важливих систем, де навіть найменші аномалії можуть призвести до серйозних проблем.
- Наприклад: банківські операції, фінансові системи, бухгалтерські програми.

- **Особливості:**

- Повністю уникає всіх аномалій: "брудних читань", "неповторюваних читань" та "фантомних читань".
- Найбільш ресурсозатратний рівень, оскільки створює ілюзію, що транзакції виконуються послідовно.