

**Національний технічний університет України  
“Київський політехнічний інститут ім. 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”**

Виконав: студент 3 курсу ФПМ групи KB-21  
Гуманіцький Андрій (5 варіант)

Перевірів(-ла):

Оцінка:

**Київ – 2024**

**Мета:** здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL

**Завдання:**

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.

**Посилання в телеграмі:** <https://t.me/axepent>

**Посилання на репозиторій:** <https://github.com/Axepent/BD>

**Опис предметної області**

Обрана предметна область - система обліку екзаменаційних балів студентів. Система має дозволити викладачам та студентам слідкувати за загальною успішністю студентів. Це допомагає формувати звіти про результати навчання та забезпечувати прозорість процесу оцінювання.

**Опис сутностей:**

1. Студент (Student)

- `student\_id` (первинний ключ)
- `first\_name` (ім'я)
- `last\_name` (прізвище)
- `com\_method` (спосіб комунікації)

Призначення: збереження даних про студентів

2. Викладач (Teacher)

- `teacher\_id` (первинний ключ)
- `first\_name` (ім'я)
- `last\_name` (прізвище)
- `com\_method` (спосіб комунікації)

Призначення: збереження даних про викладачів

### 3. Іспит (Exam)

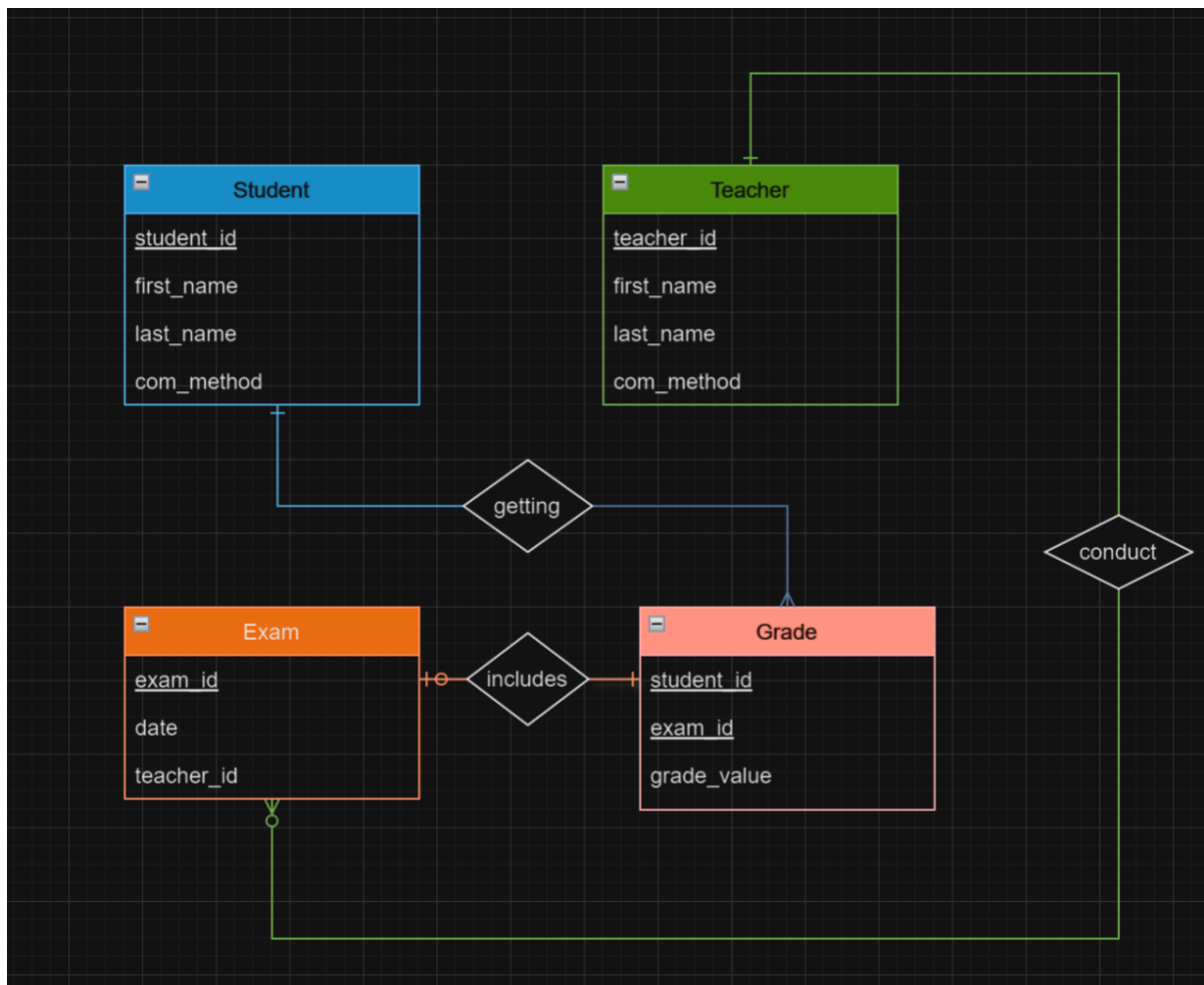
- `exam\_id` (первинний ключ)
- `teacher\_id` (посилання на вчителя)
- `date` (дата проведення іспиту)

Призначення: збереження даних про іспити

### 4. Оцінка (Grade)

- `student\_id` (посилання на студента)
- `exam\_id` (посилання на іспит)
- `grade\_value` (оцінка студента, числове значення)

Призначення: збереження оцінок студентів за конкретні іспити

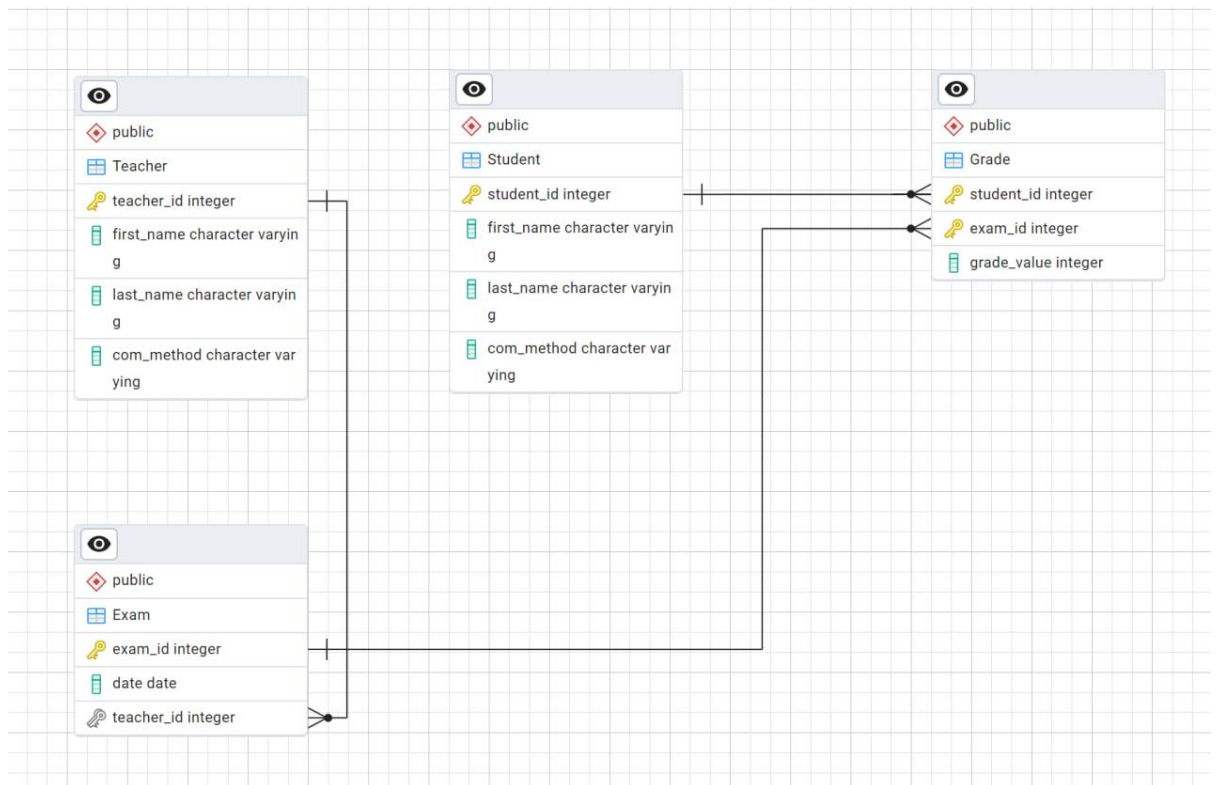


## Хід роботи

### 1. Демонстрація роботи коду після перетворення

```
Menu:  
1. Add Student  
2. View Students  
3. Update Student  
4. Delete Student  
5. Add Teacher  
6. View Teachers  
7. Update Teacher  
8. Delete Teacher  
9. Add Exam  
10. View Exams  
11. Update Exam  
12. Delete Exam  
13. Add Grade  
14. View Grades  
15. Update Grade  
16. Delete Grade  
17. Quit  
Enter your choice: █
```

Меню не було змінено, оскільки зміна була лише в частині “model.py”, яка не впливає на інтерфейс взаємодії з користувачем



Ілюстрація зав'язків в базі даних

```

Enter your choice: 13
Enter student ID: 1
Enter exam ID: 1234
Enter grade value: 77
Grade added successfully!
  
```

Query		Query History	
1	SELECT	*	FROM public."Grade"
2	ORDER BY	student_id ASC, exam_id ASC	LIMIT 100
Data Output			
	student_id [PK] integer	exam_id [PK] integer	grade_value integer
1	1	477	67
2	1	1000	99
3	1	1234	77
4	1	2070	74

Додавання даних в таблицю "Grade" та відповідна перевірка в pgAdmin 4

```

Enter your choice: 3
Enter student ID: 1
Enter column to update (first_name, last_name, com_method): com_method
Enter new value: jonhy.col@yum.com
Student updated successfully!

```

Query Query History

```

1 SELECT * FROM public."Student"
2 ORDER BY student_id ASC LIMIT 100

```

Data Output Messages Notifications

	student_id [PK] integer	first_name character varying	last_name character varying	com_method character varying
1	1	Eren	Yeager	jonhy.col@yum.com

```

Enter your choice: 15
Enter student ID: 1
Enter exam ID: 1234
Enter column to update (grade_value): grade_value
Enter new value: 88
Grade updated successfully!

```

Query Query History

```

1 SELECT * FROM public."Grade"
2 ORDER BY student_id ASC, exam_id ASC LIMIT 100

```

Data Output Messages Notifications

	student_id [PK] integer	exam_id [PK] integer	grade_value integer
1	1	477	67
2	1	1000	99
3	1	1234	88
4	1	2070	74

Зміна уже існуючих даних з таблиць “Student” та “Grade” і відповідна перевірка даних в pgAdmin4

```
Enter your choice: 16
Enter student ID: 1
Enter exam ID: 2070
Grade deleted successfully!
```

[Query](#) [Query History](#)

```
1  SELECT * FROM public."Grade"
2  ORDER BY student_id ASC, exam_id ASC LIMIT 100
```

[Data Output](#) [Messages](#) [Notifications](#)

	student_id [PK] integer	exam_id [PK] integer	grade_value integer
1	1	477	67
2	1	1000	99
3	1	1234	88
4	2	573	100

Видалення даних з таблиці "Grade" та відповідна перевірка в pgAdmin 4

## 2. Аналіз індексів

5	<i>BTree, GIN</i>	<i>before update, delete</i>
---	-------------------	------------------------------

### Індекс GIN

GIN (Generalized Inverted Index) — це індекс, який використовується для роботи з даними, що складаються з елементів (неатомарних значень). Замість індексації самих значень, GIN індексує окремі елементи, пов'язуючи кожен із них зі списком рядків таблиці, де цей елемент зустрічається.

Query Query History

```
1 CREATE INDEX gin_string_idx ON "GIN" USING GIN (string gin_trgm_ops);
```

Створення індексу в спеціально підготовленій для цього тестовій таблиці

```
1 SET enable_indexscan = off;
2
3 SELECT * FROM "GIN"
4 WHERE string LIKE 'a%'
5 ORDER BY id ASC
6 LIMIT 10;
7
```

✓ Successfully run. Total query runtime: 123 msec. 10 rows affected. ✕

```
1 SET enable_indexscan = on;
2
3 SELECT * FROM "GIN"
4 WHERE string LIKE 'a%'
5 ORDER BY id ASC
6 LIMIT 10;
7
```

✓ Successfully run. Total query runtime: 67 msec. 10 rows affected. ✕

Змінюючи дозвіл на використання індексів для запиту, можна побачити суттєву різницю у продуктивності. Це обумовлено тим, що



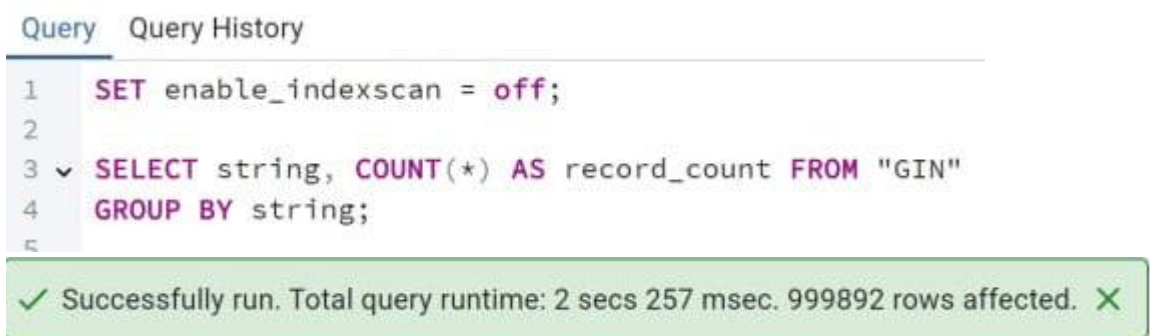
GIN-індекс оптимізує вибірку невеликої кількості рядків, які відповідають умові LIKE 'a%'. При використанні індексу значно скорочується кількість необхідних операцій для пошуку даних у порівнянні з повним скануванням таблиці



The screenshot shows a database query interface with a 'Query' tab selected. The query text is as follows:

```
1 SET enable_indexscan = on;  
2  
3 SELECT string, COUNT(*) AS record_count FROM "GIN"  
4 GROUP BY string;  
5
```

On the right side, there is a button labeled 'Execute query' with 'Alt' and 'F5' shortcuts. Below the query, a green status bar indicates: '✓ Successfully run. Total query runtime: 2 secs 284 msec. 999892 rows affected. ✕'



The screenshot shows the same database query interface, but with the query text modified to:

```
1 SET enable_indexscan = off;  
2  
3 SELECT string, COUNT(*) AS record_count FROM "GIN"  
4 GROUP BY string;  
5
```

Below the query, a green status bar indicates: '✓ Successfully run. Total query runtime: 2 secs 257 msec. 999892 rows affected. ✕'

Цей приклад демонструє, що використання індексів може бути недоцільним у певних сценаріях. У даному випадку запит охоплює всі записи таблиці, тому індекс GIN не дає жодної переваги. Обидва запити, з дозволом на використання індексів і без нього, мають однаковий час виконання. Це відбувається через те, що створений індекс не застосовується для повного сканування таблиці

## Індекс BTREE

Індекс BTree (В-дерево) використовується для даних, які можна сортувати. Він ефективний для пошуку значень за умовами порівняння ( $>$ ,  $>=$ ,  $<$ ,  $<=$ ,  $=$ ).

Ключові особливості:

1. Збалансованість: усі листкові сторінки однаково віддалені від кореня, що забезпечує однаковий час пошуку.
2. Розгалуженість: кожна сторінка індексу вміщує велику кількість записів (сотні), що мінімізує глибину дерева.
3. Впорядкованість: дані в індексі впорядковані за зростанням, що дозволяє ефективно отримувати впорядковані набори даних.

Query	Query History
1	<b>CREATE INDEX</b> idx_student_id <b>ON</b> "Student" <b>USING</b> BTREE (student_id);
2	

Створення індексу в існуючій таблиці “Student”

```
1  SET enable_indexscan = off;
2
3  SELECT * FROM "Student"
4  WHERE student_id BETWEEN 2000070 AND 2000085
5     AND first_name LIKE 'D%';
6
```

✓ Successfully run. Total query runtime: 63 msec. 3 rows affected. ✕

```
1  SET enable_indexscan = on;
2
3  SELECT * FROM "Student"
4  WHERE student_id BETWEEN 2000070 AND 2000085
5     AND first_name LIKE 'D%';
6
```

✓ Successfully run. Total query runtime: 49 msec. 3 rows affected. ✕

Змінюючи дозвіл на використання індексів для запиту, можна побачити помітну різницю у часі виконання. Ця різниця обумовлена

тим, що індекс дозволяє значно звузити область пошуку завдяки сортуванню та впорядкованості даних у структурі BTree. У випадку з маленькою вибіркою даних, як в даному прикладі, використання індексу виявляється ефективним

[Query](#) [Query History](#)

1 **EXPLAIN ANALYZE**

2 **SELECT** \*

3 **FROM** "Student"

4 **WHERE** student\_id **BETWEEN** 9999 **AND** 10010;

5

[Data Output](#) [Messages](#) [Notifications](#)

QUERY PLAN

text

1 Index Scan using idx\_student\_id on "Student" (cost=0.29..8.53 rows=12 width=28) (actual time=0.024..0.026 rows=12 loop...

2 Index Cond: ((student\_id >= 9999) AND (student\_id <= 10010))

3 Planning Time: 1.797 ms

4 Execution Time: 0.233 ms

У даному випадку, через відсутність пошуку по полю 'first\_name' (для якого індекс не було створено), швидкість виконання запиту стає ще помітнішою, ніж у минулому прикладі. Це демонструє важливість "точкового" використання індексів. Індеси мають бути створені для тих полів, які часто використовуються у фільтрах або умовах сортування, щоб максимально зменшити область пошуку та підвищити ефективність виконання запитів

### 3. Розробка триггеру

5	<i>BTree, GIN</i>	<i>before update, delete</i>
---	-------------------	------------------------------

```
Query Query History
1 CREATE OR REPLACE FUNCTION trigger_function()
2 RETURNS TRIGGER AS $$
3 DECLARE
4     changes text[];
5 BEGIN
6     IF TG_OP = 'UPDATE' THEN
7         changes := ARRAY[]::text[];
8         IF OLD.first_name IS DISTINCT FROM NEW.first_name THEN
9             changes := array_append(changes, 'first_name');
10        END IF;
11        IF OLD.last_name IS DISTINCT FROM NEW.last_name THEN
12            changes := array_append(changes, 'last_name');
13        END IF;
14        IF OLD.com_method IS DISTINCT FROM NEW.com_method THEN
15            changes := array_append(changes, 'com_method');
16        END IF;
17
18        INSERT INTO student_log (student_id, operation, old_first_name, old_last_name, old_com_method, change_time, changes)
19        VALUES (NEW.student_id, 'UPDATE', OLD.first_name, OLD.last_name, OLD.com_method, NOW(), changes);
20        RETURN NEW;
21    ELSEIF TG_OP = 'DELETE' THEN
22        changes := ARRAY['all deleted'];
23        INSERT INTO student_log (student_id, operation, old_first_name, old_last_name, old_com_method, change_time, changes)
24        VALUES (OLD.student_id, 'DELETE', OLD.first_name, OLD.last_name, OLD.com_method, NOW(), changes);
25        RETURN OLD;
26    ELSE
27        RETURN NEW;
28    END IF;
29 END;
30 $$ LANGUAGE plpgsql;
```

При оновленні або видаленні даних з таблиці “Student” в новостворену таблицю “student\_log” будуть заноситися дані про студента, що були попередньо, до зміни, а також в додатковій колонці відображаються поля що були змінені, або ж у випадку видалення видається “all deleted”

Query

Query History

1

▼

SELECT \* FROM public."Student"

2

ORDER BY student\_id ASC LIMIT 100

3

Data Output

Messages

Notifications

≡+

📄

▼

📋





▼

🗑

🗄

⬇

📈

	student_id [PK] integer 	first_name character varying 	last_name character varying 	com_method character varying 
1	1	Eren	Yeager	jonhy.col@yum.com
2	2	Carrot	Brain	veg@ht.com
3	3	Tomato	OTD	otd@example.com

Зміна даних відразу двох студентів, для пересвідчення в можливості занотувати відразу декілька змін в таблиці та перевірка дій тригери при дії “update”

Query

Query History

1

SELECT \* FROM public.student\_log

2

ORDER BY log\_id ASC LIMIT 100

3

Data Output

Messages

Notifications

	log_id [PK] integer	operation character varying (10)	student_id integer	old_first_name character varying (100)	old_last_name character varying (100)	old_com_method character varying (100)	change_time timestamp without time zone	changes text[]
1	23	UPDATE	3	Potato	OTD	otd@example.com	2024-12-01 20:51:43.335921	{first_name}
2	24	UPDATE	2	Tomato	Carrot	veg@ht.com	2024-12-01 20:51:43.335921	{first_name,last_name}

Відповідне логування цих подій після збереження змін в основній таблиці “Student”

Query Query History

1

2

3

SELECT \* FROM public."Student"

ORDER BY student\_id ASC LIMIT 100

Data Output Messages Notifications

≡+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

	student_id [PK] integer	first_name character varying	last_name character varying	com_method character varying
1	1	Eren	Yeager	jonhy.col@yum.com
2	2	Carrot	Brain	veg@ht.com

Видалення одного зі студентів з таблиці “Student” та перевірка дій тригери при дії “delete”

Query

Query History

<

Відповідне логування цієї події в таблиці “student\_log”

## 4. Рівні ізоляції

```
Query  Query History
1  CREATE TABLE test_table (
2      Number SERIAL PRIMARY KEY,
3      Text VARCHAR(20)
4  );
```

```
Data Output  Messages  Notifications
CREATE TABLE
```

Створення таблицки для тестування

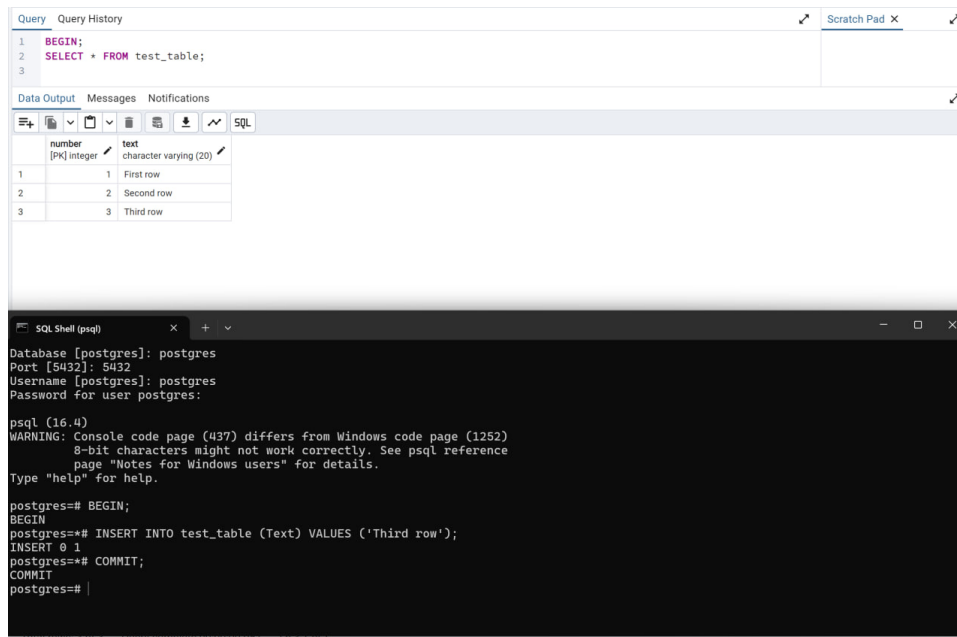
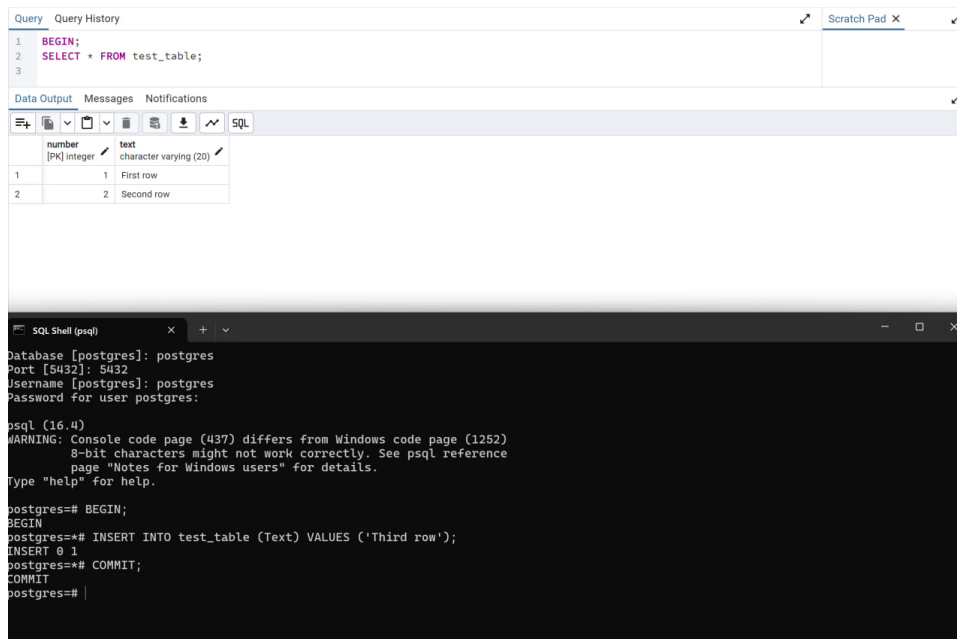
```
Query  Query History
1  SELECT * FROM test_table;
```

```
Data Output  Messages  Notifications
```

	number [PK] integer	text character varying (20)
1	1	First row
2	2	Second row

Початково занесенні до таблицки дані

# READ COMMITTED



PostgreSQL запобігає "брудному читанню", показуючи кожній транзакції знімок даних, які були зафіксовані до початку поточної операції читання. Це означає, що не можливо побачити зміни, які ще не були зафіксовані.

На першому скріншоті транзакція додає нові дані, але ми їх не бачимо допоки не буде проведена зміна даних та оновлена сторінка.



Цей рівень ізоляції є найпоширенішим і підходить для більшості транзакцій, оскільки він забезпечує баланс між узгодженістю та продуктивністю.

## REPEATABLE READ

The screenshot displays a PostgreSQL client interface with a query window, a data output pane, and a SQL shell window.

**Query Window:**

```
1 BEGIN;
2 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
3 SELECT * FROM test_table;
4
```

**Data Output Pane:**

	number [PK] integer	text character varying (20)
1	1	First row
2	2	Second row
3	3	Third row

**SQL Shell (psql):**

```
Database [postgres]: postgres
Port [5432]: 5432
Username [postgres]: postgres
Password for user postgres:

psql (16.4)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

postgres=# BEGIN;
BEGIN
postgres=# INSERT INTO test_table (Text) VALUES ('Third row');
INSERT 0 1
postgres=# COMMIT;
COMMIT
postgres=#
```

The second part of the screenshot shows the same client after an update operation:

**Query Window:**

```
1 BEGIN;
2 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
3 SELECT * FROM test_table;
4
```

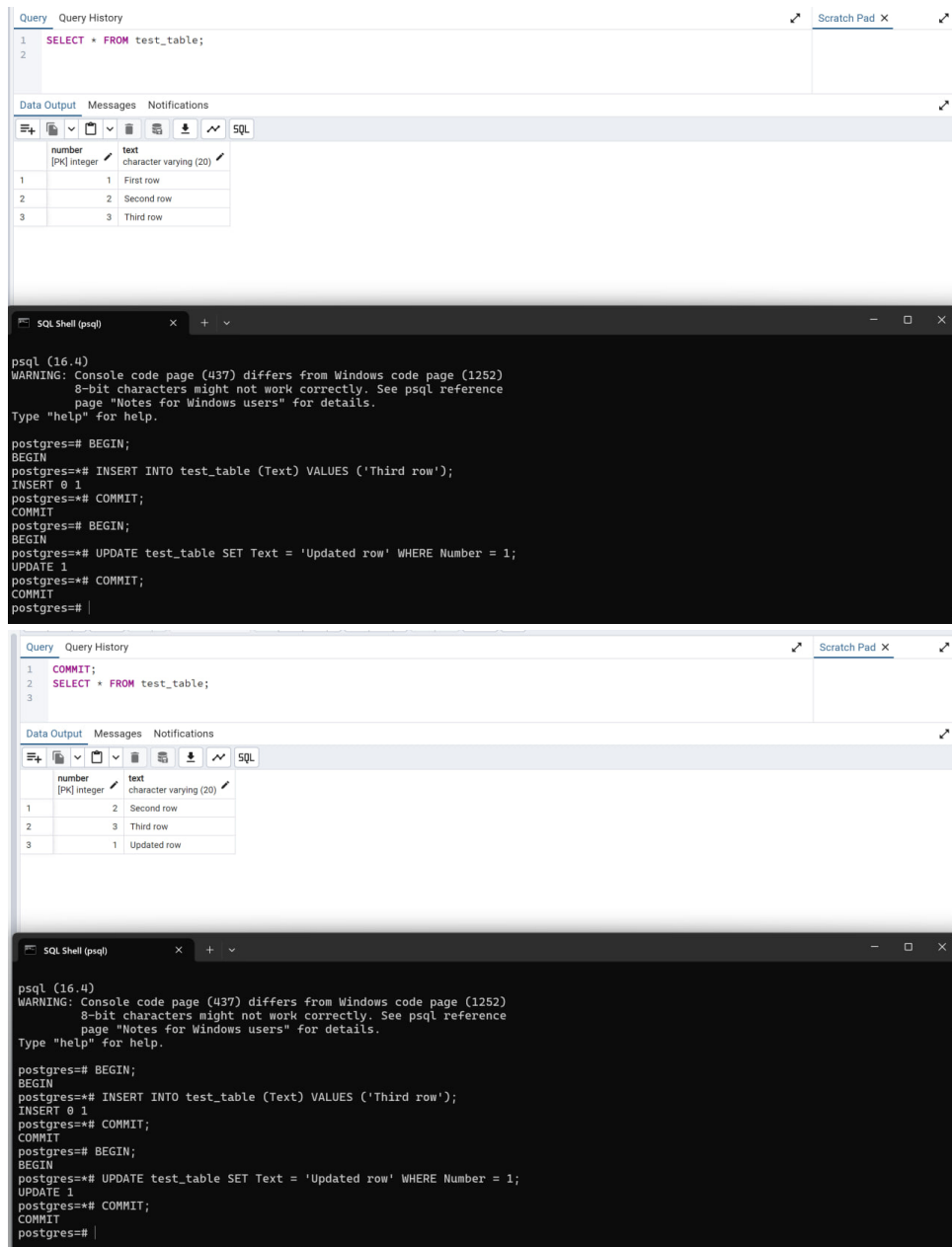
**Data Output Pane:**

	number [PK] integer	text character varying (20)
1	1	First row
2	2	Second row
3	3	Third row

**SQL Shell (psql):**

```
psql (16.4)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

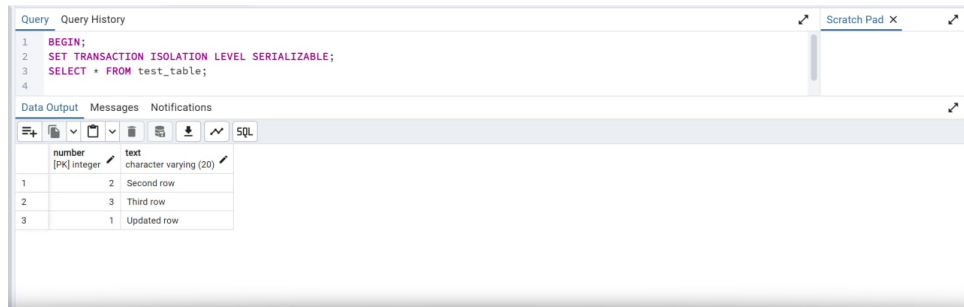
postgres=# BEGIN;
BEGIN
postgres=# INSERT INTO test_table (Text) VALUES ('Third row');
INSERT 0 1
postgres=# COMMIT;
COMMIT
postgres=# BEGIN;
BEGIN
postgres=# UPDATE test_table SET Text = 'Updated row' WHERE Number = 1;
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=#
```



Рівень ізоляції REPEATABLE READ працює таким чином, що будь-які зміни, внесені іншими транзакціями після початку нашої, ігноруватимуться до моменту завершення (або початку нової) транзакції. Цей рівень ізоляції не гарантує захист від фантомних читань (появи нових рядків, які відповідають критеріям пошуку під час повторного запиту).

Зі скріншотів ми можемо побачити відсутність феномену фантомного читання, оскільки поки не було виконано команди “COMMIT” в pgAdmin4 ми не бачили зміни в таблиці.

# SERIALIZABLE

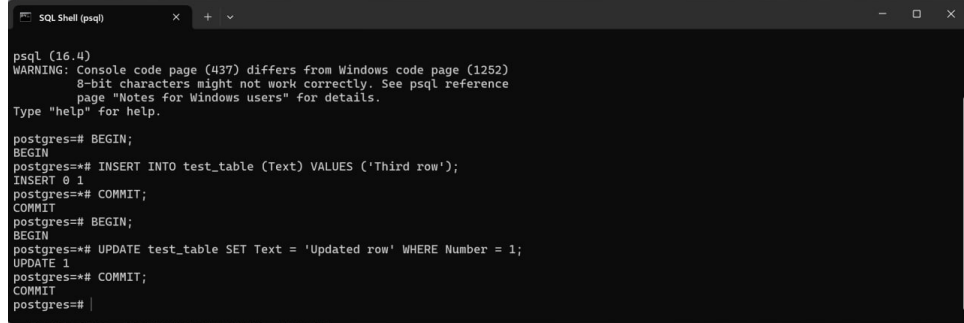


Query Query History

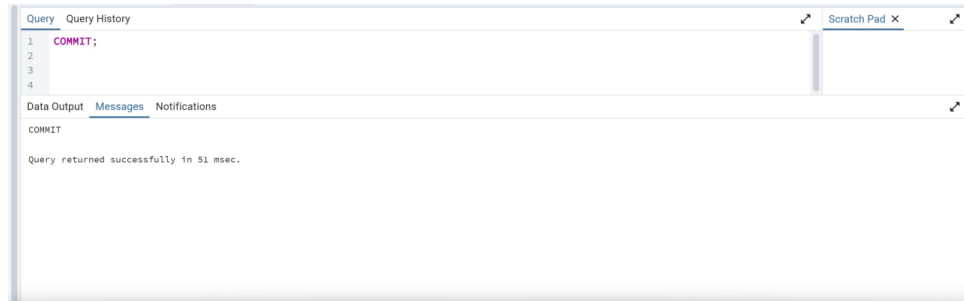
```
1 BEGIN;  
2 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
3 SELECT * FROM test_table;  
4
```

Data Output Messages Notifications

number [PK] integer	text character varying (20)
1	2 Second row
2	3 Third row
3	1 Updated row



```
psql (16.4)  
WARNING: Console code page (437) differs from Windows code page (1252)  
8-bit characters might not work correctly. See psql reference  
page "Notes for Windows users" for details.  
Type "help" for help.  
  
postgres=# BEGIN;  
BEGIN  
postgres=# INSERT INTO test_table (Text) VALUES ('Third row');  
INSERT 0 1  
postgres=# COMMIT;  
COMMIT  
postgres=# BEGIN;  
BEGIN  
postgres=# UPDATE test_table SET Text = 'Updated row' WHERE Number = 1;  
UPDATE 1  
postgres=# COMMIT;  
COMMIT  
postgres=#
```



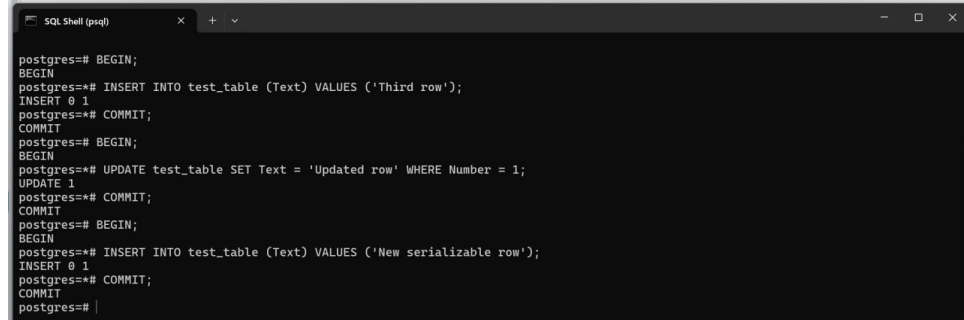
Query Query History

```
1 COMMIT;  
2  
3  
4
```

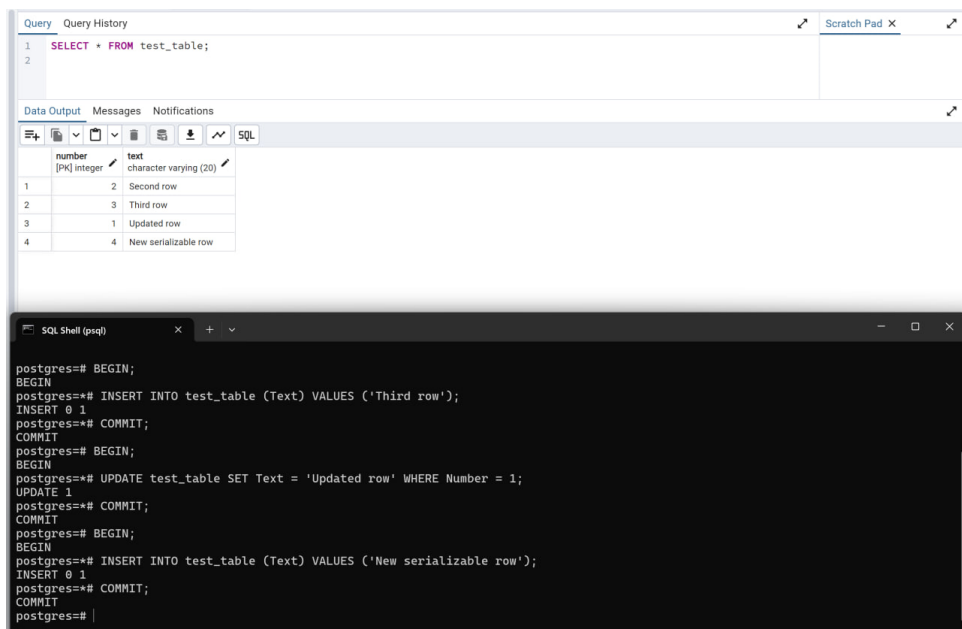
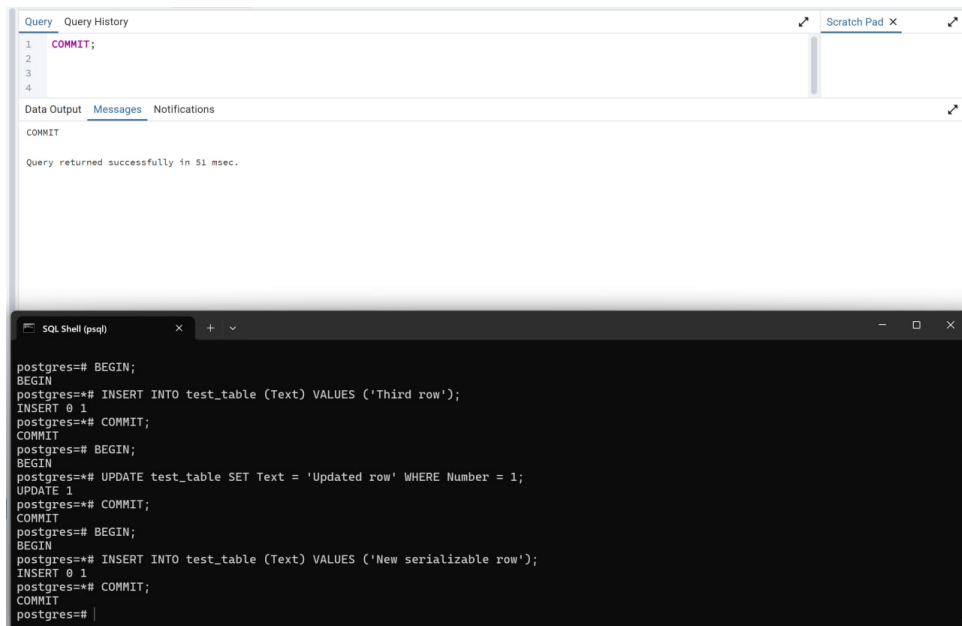
Data Output Messages Notifications

COMMIT

Query returned successfully in 51 msec.



```
postgres=# BEGIN;  
BEGIN  
postgres=# INSERT INTO test_table (Text) VALUES ('Third row');  
INSERT 0 1  
postgres=# COMMIT;  
COMMIT  
postgres=# BEGIN;  
BEGIN  
postgres=# UPDATE test_table SET Text = 'Updated row' WHERE Number = 1;  
UPDATE 1  
postgres=# COMMIT;  
COMMIT  
postgres=# BEGIN;  
BEGIN  
postgres=# INSERT INTO test_table (Text) VALUES ('New serializable row');  
INSERT 0 1  
postgres=# COMMIT;  
COMMIT  
postgres=#
```



Рівень ізоляції **SERIALIZABLE** забезпечує виконання транзакцій таким чином, ніби вони виконуються послідовно. Цей рівень забезпечує максимальну відповідність і незалежність даних, але може впливати на продуктивність через підвищення кількості блокувань або відхилень транзакцій.

Поки в обох сесіях не буде виконано операція “COMMIT” ми не будемо мати доступу до даних, що було створено в паралельній роботі з базою даних.