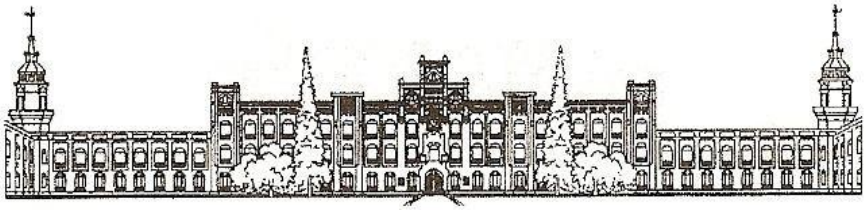


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Лабораторна робота №2

з дисципліни «Бази даних»

«Засоби оптимізації роботи СУБД PostgreSQL»

Виконав студент групи: КВ-33

ПІБ: Малойван В. Р.

Перевірив: Павловський В. І.

Київ 2025

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.

Завдання за варіантом:

10	*Hash, BRIN*	*after delete, insert*

Опис предметної галузі:

Тема бази даних: Мобільний додаток для фітнес-трекінгу та здоров’я.

Коротка характеристика: Додаток призначений для відстеження фізичної активності користувачів, контролю стану здоров’я, а також для взаємодії з іншими користувачами у соціальному форматі. Система дозволяє реєструвати тренування, зберігати дані про вправи та показники здоров’я, підтримувати соціальні зв’язки.

Розробка концептуальної моделі

При створенні даної бази даних виділено такі сутності:

1. **Користувач (User)** – представляє клієнтів мобільного додатку:
   – Атрибути: ім'я, прізвище, електронна пошта, дата реєстрації, телефон.

2. **Тренування (Workout)** – представляє заняття спортом, які виконує користувач:
   – Атрибути: тип тренування (біг, силові вправи, йога тощо), дата, час.

3. **Відвідування тренувань (User_Workout)** – відображає факт участі користувачів у тренуваннях:
   – Атрибути: дата відвідування, час відвідування.

4. **Показники здоров’я (Health Metrics)** – представляє дані про фізичний стан користувача:

– Атрибути: кількість кроків, пульс, витрачені калорії, дата вимірювання.

Зв’язки:

Зв’язок «Користувач» - «Тренування»:

– Тип зв’язку: M до N (один користувач може брати участь у багатьох тренуваннях; одне тренування може виконуватись багатьма користувачами).

– Реалізується через проміжну сутність User_Workout.

Зв’язок «Відвідування тренувань» - «Показники здоров’я»:

– Тип зв’язку: 1 до 1 (кожне відвідування має лише один набір показників; кожен набір показників відноситься лише до одного відвідування).

Зв’язок «Користувач» - «Користувач»:

– Тип зв’язку: 1 до N (кожен користувач може мати багато друзів, і кожен інший користувач також може бути другом багатьох).

– Атрибути зв’язку: статус (підтверджено/очікує), дата (дата встановлення дружби).

Графічне подання концептуальної моделі «Сутність-зв’язок» зображено на рисунку 1

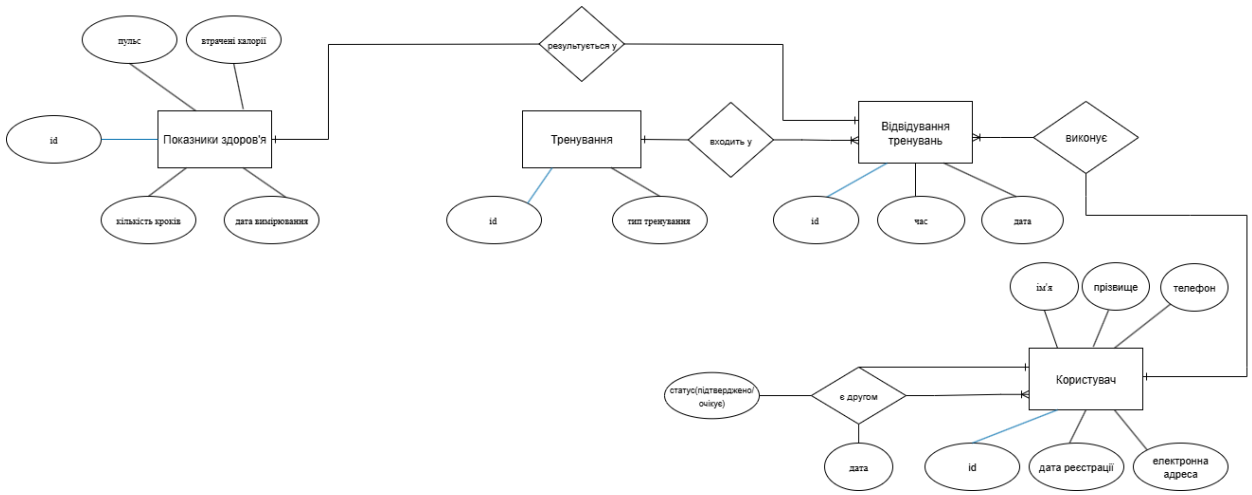


Рисунок 1 – Графічне подання концептуальної моделі «Сутність-зв’язок»

Графічне подання логічної моделі «Сутність-зв’язок» зображено на рисунку 2

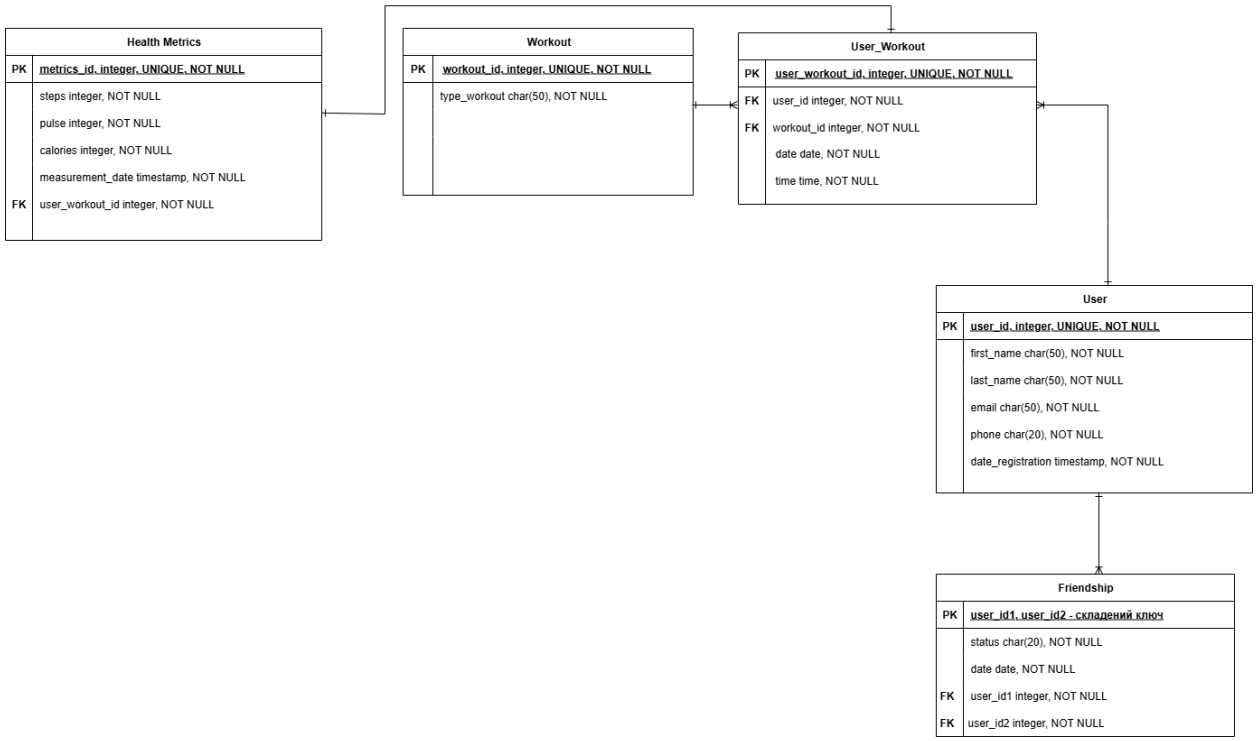


Рисунок 2 – Графічне подання логічної моделі «Сутність-зв’язок»

Хід роботи

Завдання №1. Демонстрація роботи коду після перетворення у вигляд ORM

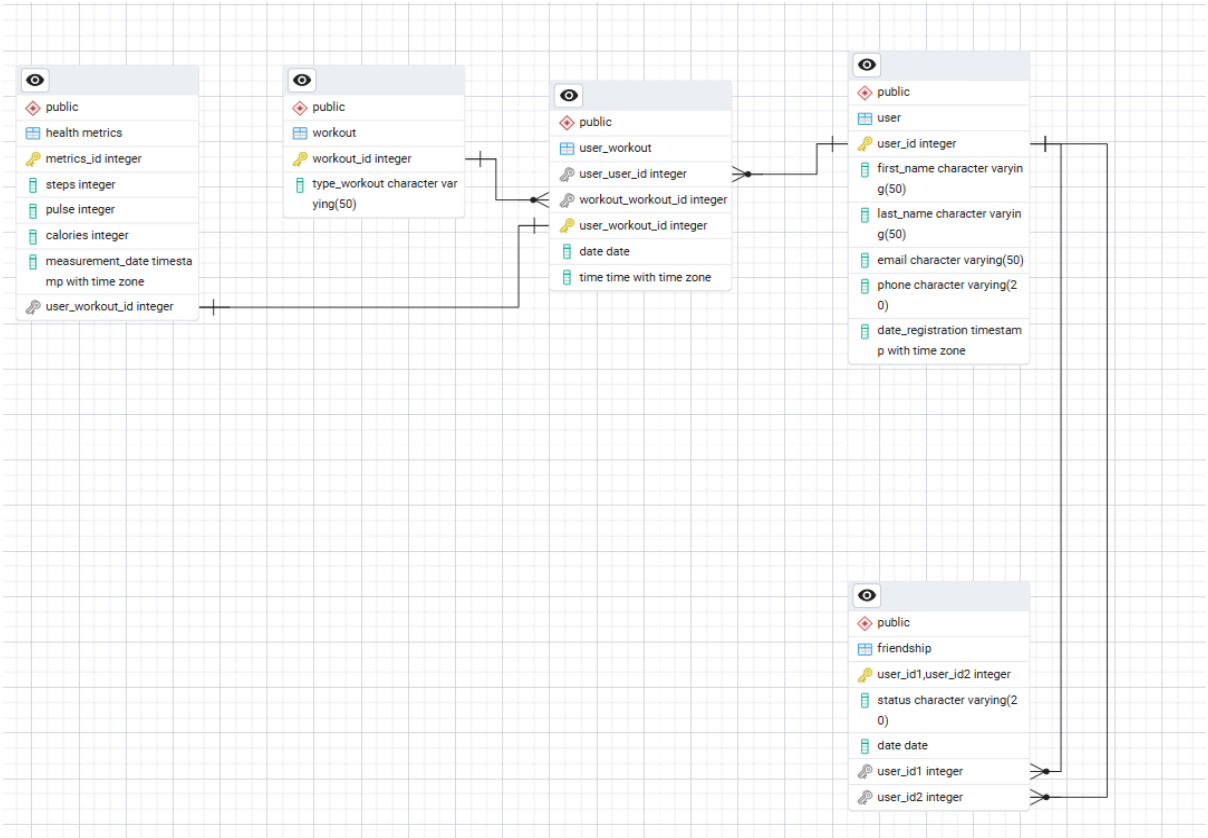


Рисунок 3 – Ілюстрація зв’язків в базі даних

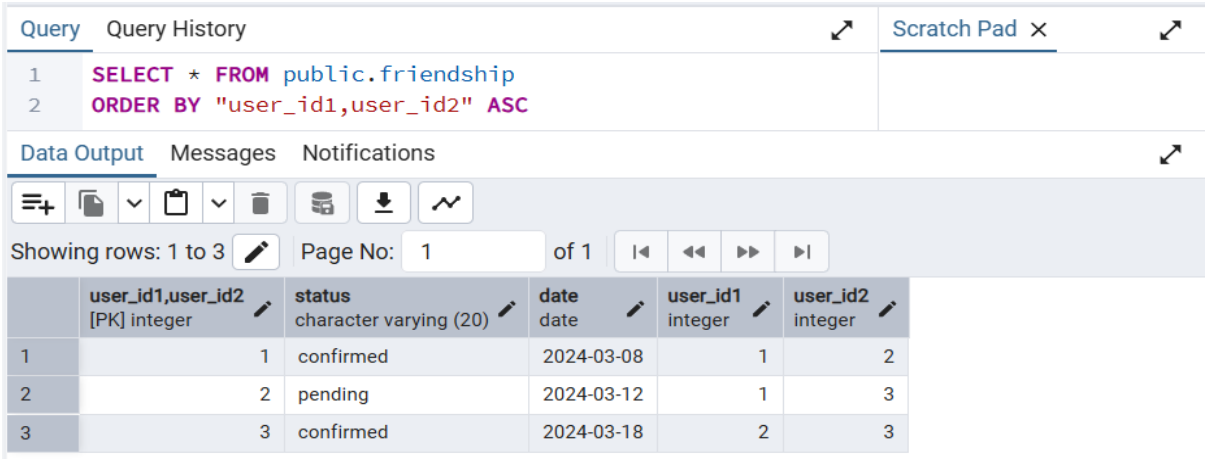
Меню за умовою завдання залишилось не зміненим:

```
--- Fitness App Main Menu ---
1. Show all data (Task 1)
2. Add data (Task 1)
3. Update data (Task 1)
4. Delete data (Task 1)
5. Generate random data (Task 2)
6. Search data (Task 3)
0. Quit

Enter your choice:
```

1. Підменю перегляду даних:

```
--- Show All Data (Task 1) ---
1. Show All Users
2. Show All Workouts
3. Show All User Workouts
4. Show All Health Metrics
5. Show All Friendships
0. Back to Main Menu

Enter your choice: 5

--- Friendships ---
Friendship PK ID: 1
User 1 ID: 1
User 2 ID: 2
Status: confirmed
Date: 2024-03-08

Friendship PK ID: 2
User 1 ID: 1
User 2 ID: 3
Status: pending
Date: 2024-03-12

Friendship PK ID: 3
User 1 ID: 2
User 2 ID: 3
Status: confirmed
Date: 2024-03-18
```

Відповідна перевірка у pgAdmin4:



Query | Query History | Scratch Pad

```
SELECT * FROM public.friendship
ORDER BY "user_id1,user_id2" ASC
```

Data Output | Messages | Notifications

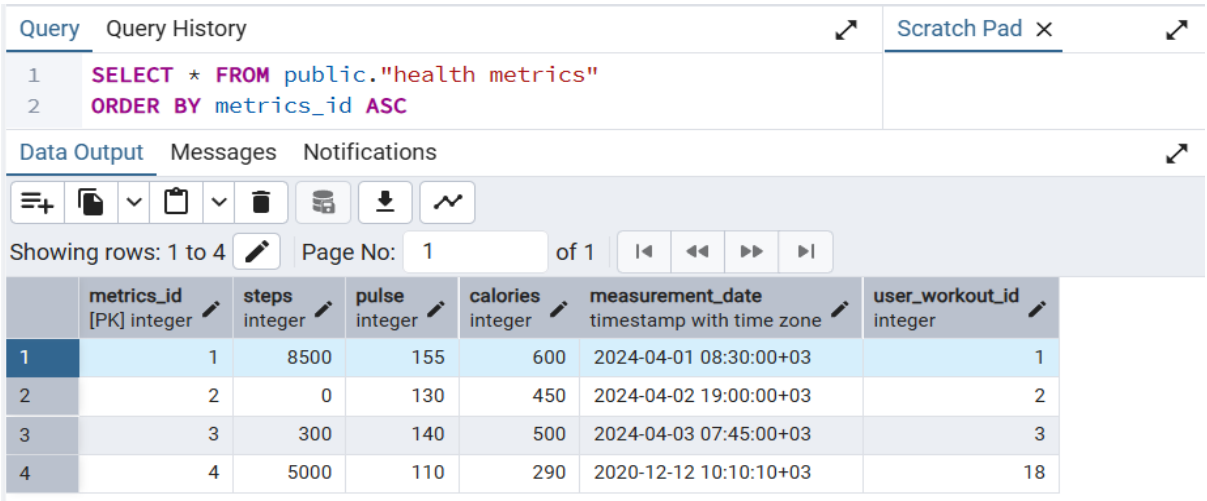
Showing rows: 1 to 3 | Page No: 1 of 1

	user_id1,user_id2 [PK] integer	status character varying (20)	date date	user_id1 integer	user_id2 integer
1	1	confirmed	2024-03-08	1	2
2	2	pending	2024-03-12	1	3
3	3	confirmed	2024-03-18	2	3

2. Підменю додавання даних:

```
--- Add data (Task 1) ---
1. Add User
2. Add Workout
3. Add User Workout
4. Add Health Metric
5. Add Friendship
0. Back to Main Menu

Enter your choice: 4
Enter user_workout_id (must exist and be unique): 18
Enter steps: 5000
Enter pulse: 110
Enter calories: 290
Enter measurement datetime (YYYY-MM-DD HH:MM:SS+TZ): 2020-12-12 10:10:10+03

1 health_metric added successfully.
```

Відповідна перевірка у pgAdmin4:



Query | Query History | Scratch Pad

```
SELECT * FROM public."health metrics"
ORDER BY metrics_id ASC
```

Data Output | Messages | Notifications

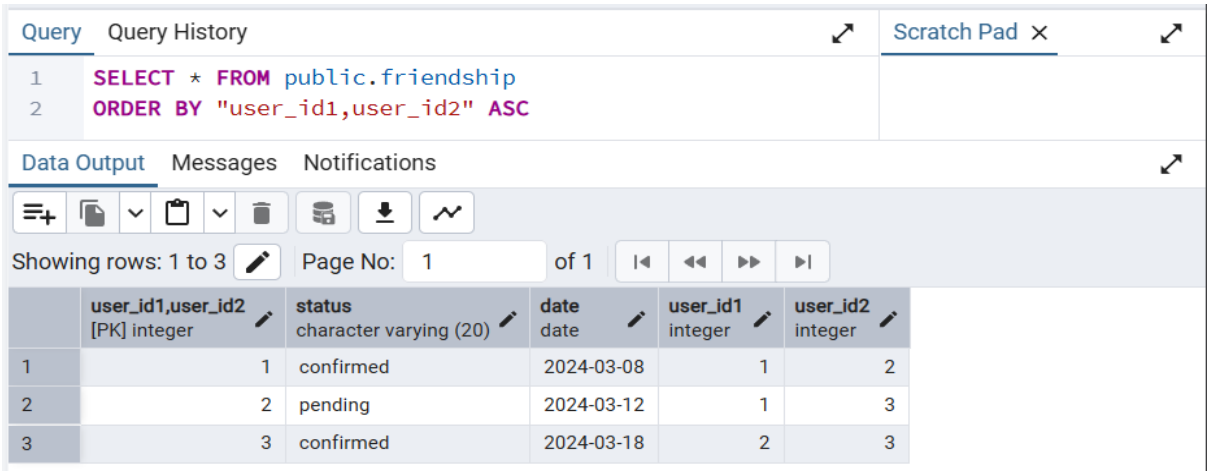
Showing rows: 1 to 4 | Page No: 1 of 1

	metrics_id [PK] integer	steps integer	pulse integer	calories integer	measurement_date timestamp with time zone	user_workout_id integer
1	1	8500	155	600	2024-04-01 08:30:00+03	1
2	2	0	130	450	2024-04-02 19:00:00+03	2
3	3	300	140	500	2024-04-03 07:45:00+03	3
4	4	5000	110	290	2020-12-12 10:10:10+03	18

3. Підменю оновлення даних:

```
--- Update data (Task 1) ---
1. Update User
2. Update Workout
3. Update User Workout
4. Update Health Metric
5. Update Friendship
0. Back to Main Menu

Enter your choice: 5
Updating friendship requires identifying the record by BOTH user IDs.
Enter user_id1 of the friendship: 1
Enter user_id2 of the friendship: 2
Enter NEW status (e.g., 'accepted'): pending

1 row updated.
```

Відповідна перевірка перед оновленням у pgAdmin4:



Query | Query History | Scratch Pad

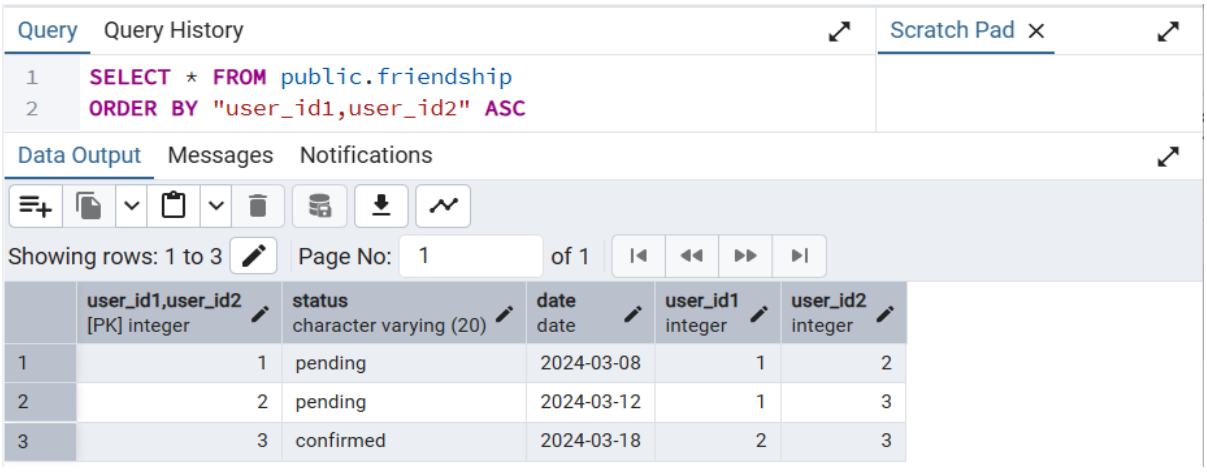
```
SELECT * FROM public.friendship
ORDER BY "user_id1,user_id2" ASC
```

Data Output | Messages | Notifications

Showing rows: 1 to 3 | Page No: 1 of 1

	user_id1,user_id2 [PK] integer	status character varying (20)	date date	user_id1 integer	user_id2 integer
1	1	confirmed	2024-03-08	1	2
2	2	pending	2024-03-12	1	3
3	3	confirmed	2024-03-18	2	3

Відповідна перевірка після оновлення у pgAdmin4:



Query | Query History | Scratch Pad

```
SELECT * FROM public.friendship
ORDER BY "user_id1,user_id2" ASC
```

Data Output | Messages | Notifications

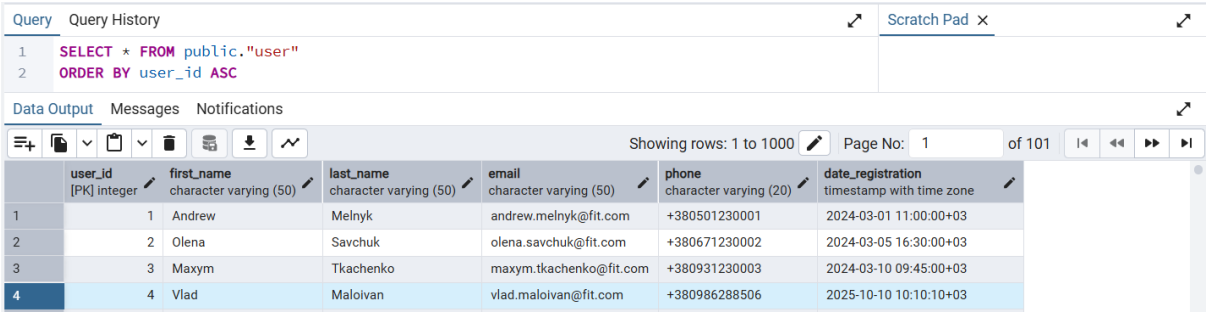
Showing rows: 1 to 3 | Page No: 1 of 1

	user_id1,user_id2 [PK] integer	status character varying (20)	date date	user_id1 integer	user_id2 integer
1	1	pending	2024-03-08	1	2
2	2	pending	2024-03-12	1	3
3	3	confirmed	2024-03-18	2	3

4. Підменю видалення даних:

```
--- Delete data (Task 1) ---
Enter '0' to exit.
Enter table_name to delete from: user
Enter field to delete by: user_id
Enter user_id value to delete: 4

1 rows deleted successfully!
```

Відповідна перевірка перед видаленням у pgAdmin4:



Query | Query History | Scratch Pad

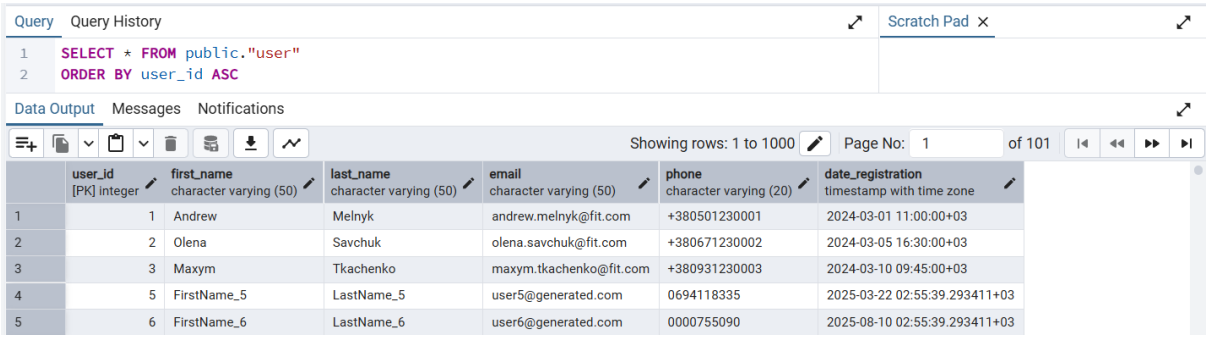
```
SELECT * FROM public."user"
ORDER BY user_id ASC
```

Data Output | Messages | Notifications

Showing rows: 1 to 1000 | Page No: 1 of 101

	user_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (50)	phone character varying (20)	date_registration timestamp with time zone
1	1	Andrew	Melnyk	andrew.melnyk@fit.com	+380501230001	2024-03-01 11:00:00+03
2	2	Olena	Savchuk	olena.savchuk@fit.com	+380671230002	2024-03-05 16:30:00+03
3	3	Maxym	Tkachenko	maxym.tkachenko@fit.com	+380931230003	2024-03-10 09:45:00+03
4	4	Vlad	Maloivan	vlad.maloivan@fit.com	+380986288506	2025-10-10 10:10:10+03

Відповідна перевірка після видалення у pgAdmin4:



Query | Query History | Scratch Pad

```
SELECT * FROM public."user"
ORDER BY user_id ASC
```

Data Output | Messages | Notifications

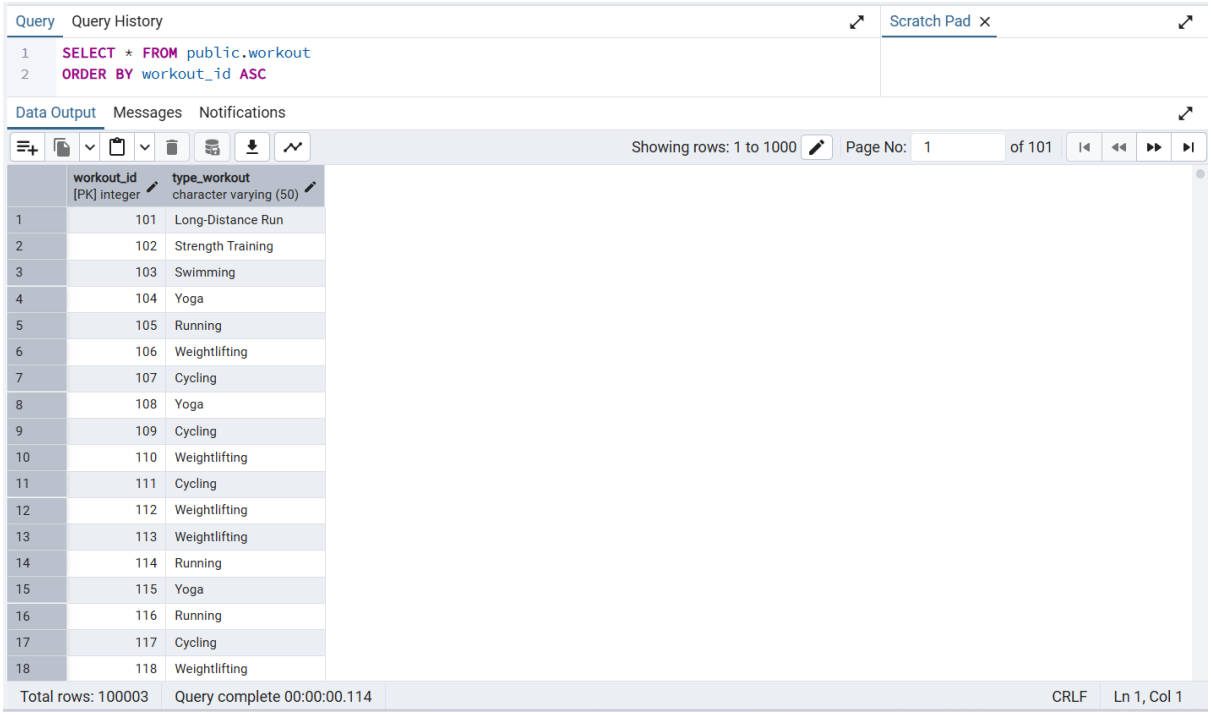
Showing rows: 1 to 1000 | Page No: 1 of 101

	user_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (50)	phone character varying (20)	date_registration timestamp with time zone
1	1	Andrew	Melnyk	andrew.melnyk@fit.com	+380501230001	2024-03-01 11:00:00+03
2	2	Olena	Savchuk	olena.savchuk@fit.com	+380671230002	2024-03-05 16:30:00+03
3	3	Maxym	Tkachenko	maxym.tkachenko@fit.com	+380931230003	2024-03-10 09:45:00+03
4	5	FirstName_5	LastName_5	user5@generated.com	0694118335	2025-03-22 02:55:39.293411+03
5	6	FirstName_6	LastName_6	user6@generated.com	0000755090	2025-08-10 02:55:39.293411+03

5. Підменю генерування даних:

```
--- Generate Random Data (Task 2) ---
1. Generate Users
2. Generate Workouts
3. Generate User Workouts (needs Users/Workouts)
4. Generate Health Metrics (needs User Workouts)
5. Generate Friendships (needs Users)
0. Back to Main Menu

Enter your choice: 2
Enter number of records to generate: 100000

100000 workouts generated successfully.
```

Відповідна перевірка у pgAdmin4:



Query | Query History | Scratch Pad

```
SELECT * FROM public.workout
ORDER BY workout_id ASC
```

Data Output | Messages | Notifications

Showing rows: 1 to 1000 | Page No: 1 of 101

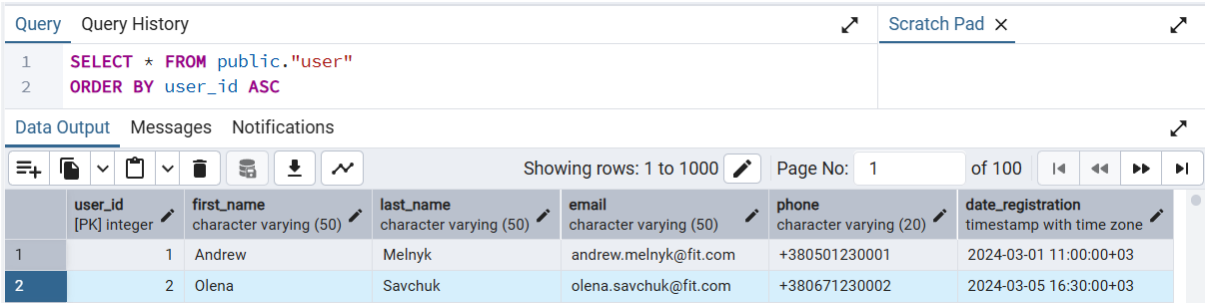
	workout_id [PK] integer	type_workout character varying (50)
1	101	Long-Distance Run
2	102	Strength Training
3	103	Swimming
4	104	Yoga
5	105	Running
6	106	Weightlifting
7	107	Cycling
8	108	Yoga
9	109	Cycling
10	110	Weightlifting
11	111	Cycling
12	112	Weightlifting
13	113	Weightlifting
14	114	Running
15	115	Yoga
16	116	Running
17	117	Cycling
18	118	Weightlifting

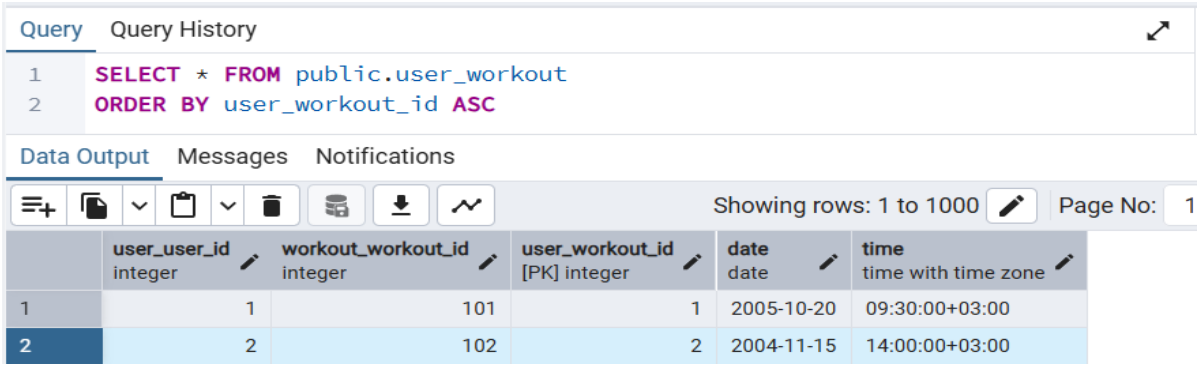
Total rows: 100003 | Query complete 00:00:00.114 | CRLF | Ln 1, Col 1

6. Підменю пошуку даних:

```
--- Search data (Task 3) ---
--- Basic Search (1 Table) ---
1. Search Users
2. Search Workouts
3. Search User Workouts
4. Search Health Metrics
5. Search Friendships
--- Complex Search (JOIN / GROUP BY) ---
6. Find Workouts by User Name (JOIN)
7. Find Users by Pulse Rate (JOIN)
8. Show Workout Count per User (GROUP BY)
----------------------------------------
0. Back to Main Menu

Enter your choice: 6
Enter User First Name pattern (e.g., 'Vla%'): Olena

1 rows found.


--- Workouts Found (by User Name) ---
User: Olena Savchuk
Workout: Strength Training
Date: 2004-11-15
Time: 14:00:00+03:00
```

Відповідна перевірка у pgAdmin4:





Фрагменти коду:

Нижче наведені приклади, що ілюструють функціональні можливості додатку та особливості моделі коду.

Клас

```
class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, primary_key=True)
    first_name = Column(String(50))
    last_name = Column(String(50))
    email = Column(String(50))
    phone = Column(String(20))
    date_registration = Column(DateTime(timezone=True))

    workouts = relationship("UserWorkout", back_populates="user")
```

Цей фрагмент демонструє вигляд запису таблиці для подальшої роботи з нею за допомогою коду

Перегляд даних

```
def get_all_users(self):
    try:
        result = self.session.query(
            User.user_id, User.first_name, User.last_name,
            User.email, User.phone, User.date_registration
        ).order_by(User.user_id).all()
        return result
```

```
    except Exception as e:
        print(f"Error: {e}")
        self.session.rollback()
        return []
```

Ця функція get_all_users отримує всі записи таблиці user та повертає їх у вигляді списку результатів.

Параметри: Функція не приймає жодних параметрів.

Запит: Виконується SQL-запит до таблиці user, який вибирає такі поля: user_id, first_name, last_name, email, phone, date_registration.

Дані сортуються за user_id у порядку зростання: .order_by(User.user_id).

Усі результати повертаються методом .all().

Результат: Функція повертає список усіх користувачів із відповідними полями. У разі успішного виконання ніякі повідомлення не виводяться.

Обробка помилок: Якщо виникає помилка, функція:

- виводить повідомлення "Error: <опис>"
- виконує rollback()
- повертає порожній список

Особливості: Це проста функція перегляду даних, яка не змінює таблицю, а лише повертає впорядкований список користувачів.

Створення і внесення даних

```
    def add_user(self, first_name, last_name, email, phone, date_registration):
    try:
        max_id = self.session.query(func.max(User.user_id)).scalar() or 0
        new_id = max_id + 1

        new_obj = User(
            user_id=new_id, first_name=first_name, last_name=last_name,
            email=email, phone=phone, date_registration=date_registration
        )
        self.session.add(new_obj)
        self.session.commit()
        print(f"\n1 user added successfully.")
    except Exception as e:
        self.session.rollback()
        print(f"Error: {e}")
```

Ця функція add_user додає новий запис про користувача у таблицю User.

Параметри: first_name, last_name, email, phone, date_registration — значення для полів таблиці, що визначають ім’я, прізвище, електронну пошту, номер телефону та дату реєстрації користувача.

Запит: Функція виконує запит до бази даних для визначення максимального значення user_id у таблиці: func.max(User.user_id). На основі отриманого значення формується новий user_id, який дорівнює максимальному плюс один. Потім створюється новий об’єкт User із цим ідентифікатором та переданими параметрами, після чого він додається у сесію і зберігається в базі за допомогою commit().

Результат: У разі успішного виконання функція додає один новий запис у таблицю user та виводить повідомлення про успішне додавання

Обробка помилок: У випадку виникнення помилки відбувається відкат транзакції (rollback()), а користувачу виводиться повідомлення з текстом помилки.

Особливості: Функція вручну забезпечує унікальність значення user_id, самостійно визначаючи наступний ідентифікатор на основі найбільшого існуючого. Сесія не закривається автоматично.

Оновлення даних

```
def update_user(self, user_id, first_name, last_name, email, phone):
try:
    q = self.session.query(User).filter(User.user_id == user_id)
    if q.first():
        q.update({
            User.first_name: first_name, User.last_name: last_name,
            User.email: email, User.phone: phone
        })
        self.session.commit()
        print(f"\n1 row updated.\n")
    else:
        print("User not found")
except Exception as e:
    self.session.rollback()
    print(f"Error: {e}")
```

Ця функція update_user оновлює інформацію про користувача у таблиці User на основі переданого user_id.

Параметри:

– user_id — ID користувача, дані якого потрібно оновити.
– first_name, last_name, email, phone — нові значення для відповідних полів таблиці.

Запит: Функція виконує пошук запису в таблиці user за вказаним user_id: self.session.query(User).filter(User.user_id == user_id). Якщо запис існує,

виконується оновлення полів first_name, last_name, email і phone методом update(), після чого зміни зберігаються за допомогою commit().

Результат: Якщо запис знайдено, функція виводить повідомлення:1 row updated. Якщо запис із таким user_id не знайдено, виводить: User not found

Обробка помилок: У разі виникнення помилки транзакція скасовується (rollback()), а користувачу виводиться повідомлення з описом помилки.

Особливості: Функція перевіряє існування запису перед оновленням, що запобігає спробам змінювати неіснуючі дані. Сесія не закривається автоматично.

Видалення даних

```
def delete_data(self, table_name, field, value):
    try:
        model_map = {
            'user': User, 'workout': Workout, 'user_workout': UserWorkout,
            'health': HealthMetric, '"health metrics"': HealthMetric,
            'friendship': Friendship
        }
        ModelClass = model map.get(table name)
        if not ModelClass:
            print("Unknown table")
            return

        field_attr = getattr(ModelClass, field)

        rows_deleted = self.session.query(ModelClass).filter(field_attr ==
value).delete()
        self.session.commit()
        print(f"\n{rows_deleted} rows deleted successfully! ")
    except Exception as e:
        self.session.rollback()
        print(f"Error: {e}")
```

Ця функція delete_data видаляє один або кілька записів із таблиці, вказаної у параметрі table_name, на основі умови field = value.

Параметри:

table_name — назва таблиці, з якої потрібно виконати видалення.

field — назва поля, за яким здійснюється фільтрація.

value — значення, яке має відповідати полю field для видалення записів.

Запит: Функція використовує внутрішню карту відповідності model_map, щоб знайти модель SQLAlchemy, яка відповідає назві таблиці. Якщо відповідної моделі немає — виводиться повідомлення "Unknown table" і операція завершується. Далі отримує атрибут поля: getattr(ModelClass, field).

Після цього виконує видалення всіх записів, у яких значення вибраного поля дорівнює value, і застосовує зміни через commit().

Результат: Функція виводить кількість видалених рядків: X rows deleted successfully! Якщо таблицю не знайдено, виводиться повідомлення "Unknown table".

Обробка помилок: У разі виникнення помилки транзакція скасовується (rollback()), а користувачу виводиться повідомлення з текстом помилки.

Особливості: Функція перевіряє існування таблиці перед виконанням операції. Видалення виконується для всіх записів, що відповідають умові, але функція не перевіряє окремо існування поля або конкретного запису — якщо поле вказане неправильно, помилка буде перехоплена блоком except.

Генерація даних

```
def generate_users(self, num_users):
    try:
        max_id = self.session.query(func.max(User.user_id)).scalar() or 0
        new_users = []

        for i in range(1, int(num_users) + 1):
            uid = max_id + i
            u = User(
                user_id=uid,
                first_name=f'Name_{uid}',
                last_name=f'Surname_{uid}',
                email=f'user{uid}@gen.com',
                phone=str(random.randint(100000000, 999999999)),
                date_registration=datetime.date.today() -
datetime.timedelta(days=random.randint(0, 365))
            )
            new_users.append(u)

        # Bulk insert
        self.session.add_all(new_users)
        self.session.commit()
        print(f"\n{num_users} users generated successfully.\n")
    except Exception as e:
        self.session.rollback()
        print(f"Generation error: {e}")
```

Функція generate_users генерує вказану кількість нових користувачів та додає їх до таблиці User, використовуючи автоматично сформовані значення для полів. Отримання максимального user_id: Виконується запит: func.max(User.user_id). Щоб визначити найбільший існуючий user_id у таблиці User. Якщо таблиця порожня, результатом буде None, тому значення замінюється на 0. Нові користувачі отримують послідовні ідентифікатори, починаючи з max_id + 1.

Генерація користувачів:

Для кожного користувача створюється об’єкт User зі значеннями:

- user_id — унікальний, зростаючий для кожного нового запису.
- first_name — у форматі Name_<uid>.
- last_name — у форматі Surname_<uid>.
- email — у форматі user<uid>@gen.com.
- phone — випадкове дев’ятизначне число.
- date_registration — випадкова дата в межах останніх 365 днів від сьогодні.

Усі згенеровані користувачі додаються до списку new_users.

Додавання користувачів: Пакетне додавання здійснюється методом: self.session.add_all(new_users). Після чого транзакція фіксується через commit(). У разі успіху виводиться повідомлення: <num_users> users generated successfully.

Обробка помилок: Якщо виникла помилка, транзакція скасовується (rollback()), а на екран виводиться повідомлення: Generation error: <опис помилки>

Особливості: Функція уникає конфліктів user_id, обчислюючи нові ідентифікатори на основі максимального існуючого значення. Пакетне додавання (add_all) підвищує ефективність роботи, а транзакційний підхід гарантує цілісність даних у разі помилки.

Пошук даних

```
def search_users_by_pulse(self, min_pulse):
    try:
        # DISTINCT, JOINs
        query = self.session.query(
            User.user_id, User.first_name, User.last_name, HealthMetric.pulse
        ).select_from(User) \
            .join(UserWorkout, User.user_id == UserWorkout.user_user_id) \
            .join(HealthMetric, UserWorkout.user_workout_id ==
HealthMetric.user_workout_id) \
            .filter(HealthMetric.pulse > min_pulse) \
            .distinct() \
            .order_by(HealthMetric.pulse.desc())

        result = query.all()
        print(f"\n{len(result)} users found.\n")
        return result
    except Exception as e:
        print(f"Error: {e}")
        return []
```

Ця функція search_users_by_pulse виконує пошук користувачів, у яких значення пульсу перевищує задане значення min_pulse.

Параметри:

min_pulse — мінімальне значення пульсу, яке повинно бути перевищено, щоб користувач потрапив у результати пошуку.

Запит: Функція формує SQL-запит із використанням JOIN між трьома таблицями: User, User_Workout, Health Metrics.

Запит вибирає такі поля: User.user_id, User.first_name, User.last_name, HealthMetric.pulse. Фільтрація виконується умовою: HealthMetric.pulse > min_pulse

Додатково застосовуються:

- distinct() — усуває дублікати користувачів у результатах
- order_by(HealthMetric.pulse.desc()) — сортує результати за пульсом у порядку спадання

Результат отримується методом .all().

Результат: Функція виводить кількість знайдених користувачів: X users found. Повертає список отриманих результатів. Якщо сталася помилка — повертає порожній список.

Обробка помилок: У випадку виникнення помилки виводиться повідомлення: Error: <опис помилки>. Транзакція не модифікується (оскільки це лише SELECT-запит), і функція повертає порожній список.

Особливості: Функція використовує JOIN для отримання зв’язаних даних між таблицями користувачів, їх тренувань і метрик здоров’я. Унікальність результатів забезпечується за допомогою distinct(), а зручність перегляду — сортуванням у порядку спадання пульсу.

Завдання №2. Аналіз індексів

Індекс BRIN

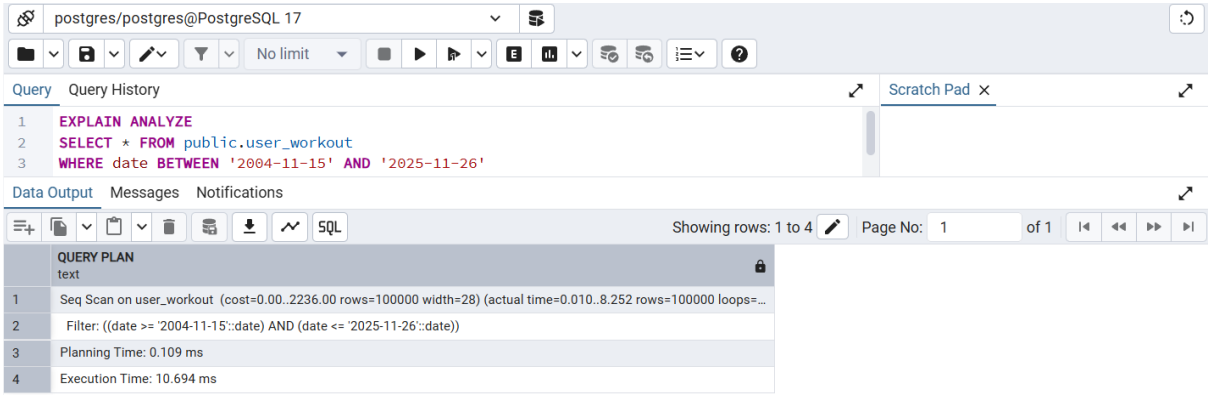
Індекс BRIN (Block Range INdex) використовується для дуже великих таблиць, у яких значення поля мають природну кореляцію з фізичним розташуванням рядків у таблиці (наприклад: дати, часові мітки, послідовні

ідентифікатори). Він особливо ефективний у випадках, коли дані зберігаються у порядку, що приблизно відповідає значенням у стовпці.

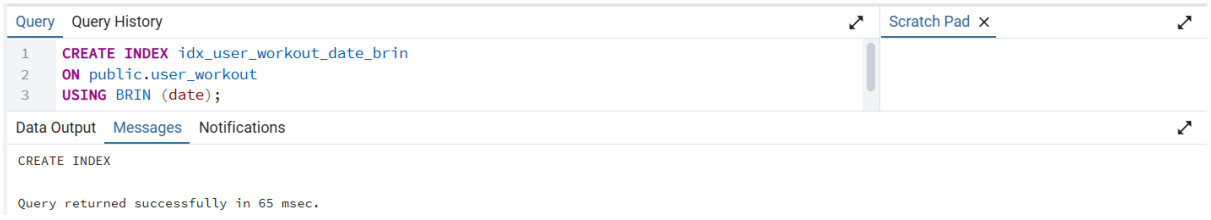
Ключові особливості:

1. Працює з діапазонами блоків: BRIN зберігає мінімальне та максимальне значення для кожного діапазону фізичних сторінок таблиці, а не окремі значення кожного рядка. Це робить індекс дуже компактним.
2. Надзвичайно малий розмір: BRIN зберігає лише кілька статистичних значень для кожного діапазону сторінок, тому його обсяг дуже малий. Завдяки цьому індекс займає мінімум місця на диску та майже не навантажує пам’ять, що робить його особливо вигідним для робочих баз із великими таблицями.
3. Оптимізований для послідовних даних: BRIN працює найкраще тоді, коли дані вставляються у природному порядку, наприклад, за датою створення, timestamp або зростальними id. Завдяки такій впорядкованості діапазони блоків охоплюють чіткі, логічні інтервали значень.

Точний пошук (Equality):



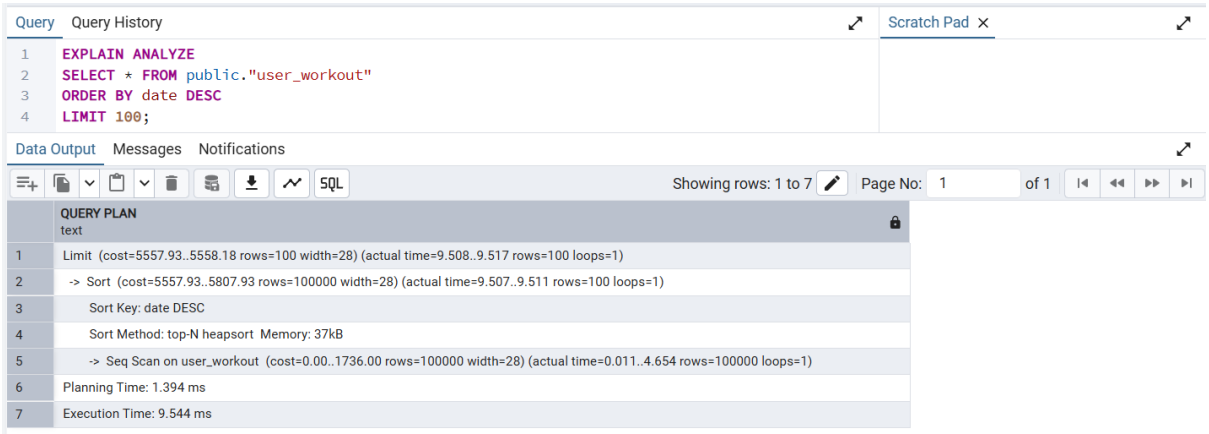
Створення індексу BRIN:



Пошук з індексом BRIN:

Швидше виконання запиту при використанні BRIN обумовлене тим, що індекс дозволяє швидко визначити, які діапазони блоків потенційно містять потрібні значення, та прочитати лише ці блоки. У випадку великої таблиці це значно зменшує кількість даних, які потрібно сканувати. У невеликих таблицях ефект менш помітний, але індекс усе одно працює коректно.

Сортування та Ліміт:



У даному випадку, навіть попри наявність BRIN-індексу, запит продовжує виконувати повне сортування результатів у пам’яті. Це пояснюється тим, що BRIN не зберігає впорядкованої структури значень — він містить лише мінімальні та максимальні значення для окремих діапазонів сторінок. Тому під час операції ORDER BY date DESC PostgreSQL не має можливості використати BRIN для отримання даних у вже відсортованому вигляді.

У результаті система виконує Seq Scan по всій таблиці та додає окремий етап сортування (Sort), що чітко видно у плані запиту. Така поведінка демонструє, що BRIN-індекси не призначені для оптимізації операцій сортування. Вони ефективні в інших ситуаціях — зокрема, коли потрібно швидко відфільтрувати великі діапазони значень, але не тоді, коли важлива збережена впорядкованість даних.

Цей приклад наголошує на необхідності враховувати призначення BRIN-індексів. Якщо поле використовується у запитах, де критичною є швидка

вибірка у відсортованому порядку, BRIN не забезпечить очікуваного прискорення.

Індекс HASH

Індекс HASH Індекс HASH використовується для оптимізації запитів, що виконують пошук за точним співпадінням (рівність). Він базується на структурі хеш-таблиці та є ефективним вибором, коли необхідно швидко знайти конкретний рядок за унікальним ключем (наприклад: номер телефону, email, UUID або артикул товару), ігноруючи порядок даних.

Ключові особливості:

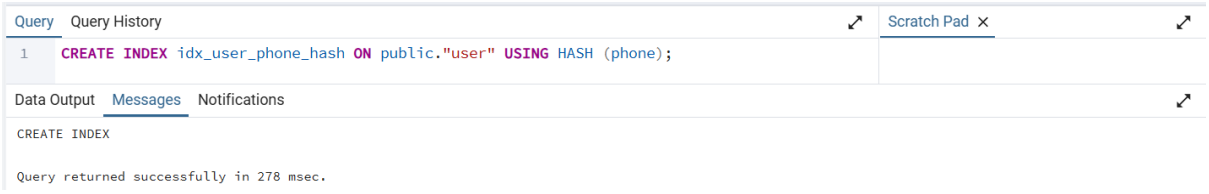
1. Максимальна швидкість точного пошуку: HASH-індекс забезпечує теоретично найшвидший доступ до даних (складність O(1)) при операціях перевірки на рівність (оператор =). Система обчислює хеш-код шуканого значення і миттєво переходить до відповідного блоку даних, оминаючи необхідність проходити через рівні дерева, як це відбувається у B-Tree індексах.

2. Обмежена сфера застосування: На відміну від B-Tree або BRIN, HASH-індекс підтримує лише оператор рівності (=). Він абсолютно не підходить для пошуку діапазонів (<, >, BETWEEN), пошуку за частиною рядка (LIKE) або сортування (ORDER BY), оскільки хеш-функція розподіляє значення хаотично, і логічний порядок даних втрачається.

3. Компактність для довгих ключів: Цей тип індексу може бути вигіднішим за B-Tree при індексації довгих рядкових даних (наприклад, довгих URL-адрес або текстових ключів). Замість зберігання повного довгого рядка у структурі індексу, HASH зберігає лише компактний хеш-код (зазвичай 4 байти), що дозволяє економити місце на диску при роботі з великими текстовими полями.

Точний пошук (Equality):

Створення індексу HASH:



Пошук з індексом HASH:

У проведеному запиті спостерігається значний приріст швидкості, оскільки умова відбору ґрунтується на точній відповідності значення поля phone. Для таких випадків HASH-індекс працює максимально ефективно: система одразу переходить до потрібного рядка, оминаючи перегляд зайвих блоків. Це показує, що HASH-індекс доцільно застосовувати саме тоді, коли потрібен швидкий доступ за конкретним ключем, а не аналіз цілого діапазону даних.

Агрегація та групування (JOIN):

Query | Query History | Scratch Pad

```
1  EXPLAIN ANALYZE
2  SELECT w.type_workout, COUNT(uw.user_workout_id) as total_sessions
3  FROM public.workout w
4  JOIN public.user_workout uw ON w.workout_id = uw.workout_workout_id
5  GROUP BY w.type_workout
6  ORDER BY total_sessions DESC;
```

Data Output | Messages | Notifications

Showing rows: 1 to 14 | Page No: 1 of 1

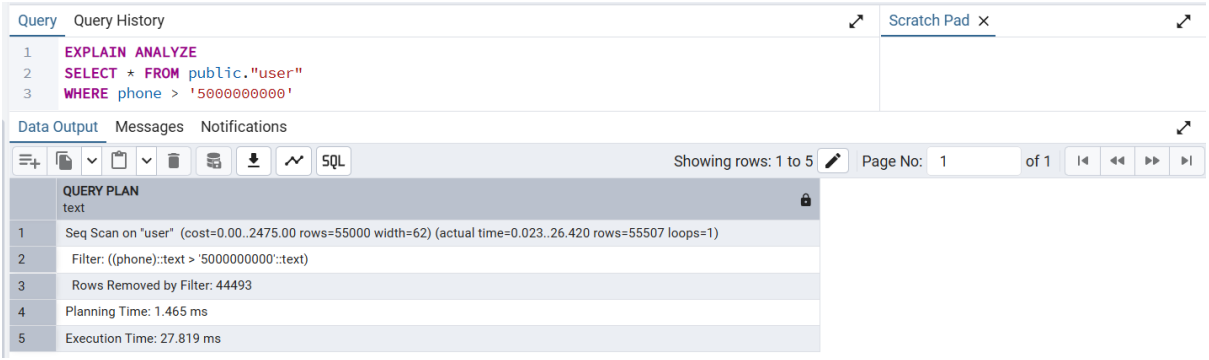
	QUERY PLAN text
1	Sort (cost=5313.72..5313.73 rows=6 width=16) (actual time=72.456..72.459 rows=7 loops=1)
2	Sort Key: (count(uw.user_workout_id)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=5313.58..5313.64 rows=6 width=16) (actual time=72.436..72.439 rows=7 loops=1)
5	Group Key: w.type_workout
6	Batches: 1 Memory Usage: 24kB
7	-> Hash Join (cost=2815.07..4813.58 rows=100000 width=12) (actual time=18.978..58.635 rows=100000 loops=1)
8	Hash Cond: (uw.workout_workout_id = w.workout_id)
9	-> Seq Scan on user_workout uw (cost=0.00..1736.00 rows=100000 width=8) (actual time=0.012..5.198 rows=100000 loops=1)
10	-> Hash (cost=1565.03..1565.03 rows=100003 width=12) (actual time=18.804..18.805 rows=100003 loops=1)
11	Buckets: 131072 Batches: 1 Memory Usage: 5518kB
12	-> Seq Scan on workout w (cost=0.00..1565.03 rows=100003 width=12) (actual time=0.008..7.736 rows=100003 loops=1)
13	Planning Time: 6.257 ms
14	Execution Time: 73.147 ms

У даному випадку механізм Hash Join, який обрав планувальник, демонструє типову поведінку під час виконання аналітичних запитів, що включають агрегування та групування. Завдяки використанню хеш-таблиць система може ефективно зіставляти рядки між таблицями Workout та User_Workout за умовою рівності ключів. Такий підхід добре масштабується, оскільки під час хешування значень PostgreSQL значно зменшує кількість порівнянь, які потрібно виконати при об’єднанні великих наборів даних.

Hash-індекс може бути корисним у подібних сценаріях, оскільки він оптимізує доступ до значень, що беруть участь у рівності (=), на які спирається сам Hash Join. Це дозволяє швидше формувати хеш-таблицю та зменшити навантаження на послідовне сканування таблиці. Проте при виконанні агрегування та групування основний час все одно витрачається на побудову hash-структур та подальшу агрегацію, що помітно у плані виконання.

Цей приклад демонструє, що HASH-індекси можуть відігравати роль у прискоренні операцій JOIN саме у випадках, коли об’єднання базується на точній відповідності ключів. Але при цьому вони не впливають на подальші етапи сортування й агрегування, які залишаються найбільш ресурсоємними частинами запиту.

Діапазонне порівняння:



У цьому прикладі видно, що запит із умовою діапазону (phone > '500000000') повністю обходить HASH-індекс, навіть якщо він існує. Це пояснюється природою хешування: хеш-таблиця не зберігає логічного порядку значень і не дозволяє визначити, які ключі є “більшими” чи “меншими”. Оскільки хеш-функція перетворює кожне значення на незалежний хеш-код, планувальник не може використати такий індекс для операцій типу <, >, <= або >=.

У результаті PostgreSQL змушений виконувати повне послідовне сканування таблиці (Seq Scan) та фільтрацію рядків уже під час проходження. Це суттєво збільшує вартість запиту, особливо при роботі з великими наборами даних, де використання відповідного індексу могло би значно зменшити кількість перевірок.

Приклад підкреслює, що HASH-індекси доцільно використовувати виключно для точних відповідностей (=). Якщо ж поле застосовується у діапазонних умовах, індекс цього типу не лише не допоможе, а й буде повністю проігнорований планувальником, що робить його неефективним для подібних сценаріїв.

Завдання №3. Розробка тригеру

1. Створення таблиці User_log для запису подій insert та after delete в таблиці User:

```
Query  Query History

1  CREATE TABLE user_log (
2      log_id SERIAL PRIMARY KEY,
3      user_id INTEGER,
4      operation VARCHAR(20) NOT NULL,
5      old_first_name VARCHAR(50),
6      old_last_name VARCHAR(50),
7      old_email VARCHAR(50),
8      old_phone VARCHAR(20),
9      old_date_registration TIMESTAMP WITH TIME ZONE,
10     change_time TIMESTAMP DEFAULT NOW()
11 );

Data Output  Messages  Notifications

CREATE TABLE

Query returned successfully in 47 msec.
```

2. Створення тригерної функції (Logic):

```
Query  Query History                                                    Scratch Pad

1  CREATE OR REPLACE FUNCTION log_user_changes()
2  RETURNS TRIGGER AS $$
3  DECLARE
4      rec RECORD;
5  BEGIN
6      FOR rec IN SELECT user_id FROM public."user" LIMIT 1 LOOP
7          NULL;
8      END LOOP;
9
10     IF (TG_OP = 'INSERT') THEN
11         INSERT INTO user_log (user_id, operation, change_time)
12         VALUES (NEW.user_id, 'INSERT', NOW());
13
14     ELSIF (TG_OP = 'DELETE') THEN
15         INSERT INTO user_log (user_id, operation, old_first_name, old_last_name, old_email, old_phone, ol
16         VALUES (OLD.user_id, 'DELETE', OLD.first_name, OLD.last_name, OLD.email, OLD.phone, OLD.date_regi
17     END IF;
18
19     RETURN NULL;
20
21 EXCEPTION WHEN OTHERS THEN
22     RETURN NULL;
23 END;
24 $$ LANGUAGE plpgsql;

Data Output  Messages  Notifications

CREATE FUNCTION

Query returned successfully in 41 msec.
```

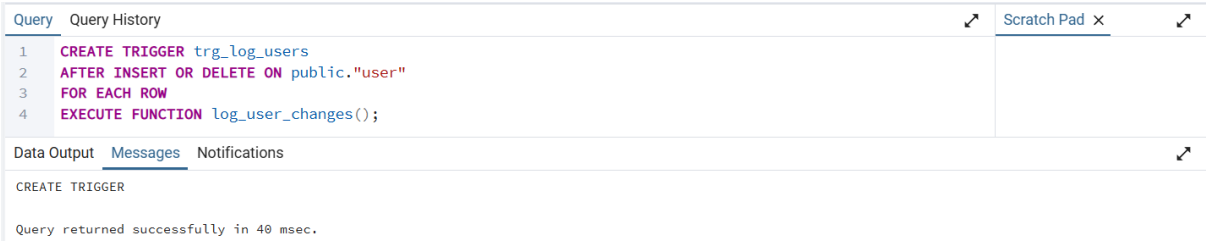
При додаванні нового запису до таблиці User, умова IF (TG_OP = 'INSERT') ідентифікує операцію вставки. Тригер автоматично фіксує цю подію в журналі User_log, зберігаючи ідентифікатор нового користувача (NEW.user_id), тип операції та поточний час зміни.

При видаленні запису з таблиці User, спрацьовує умова ELSIF (TG_OP = 'DELETE'). Тригер перехоплює дані видаленого користувача (змінна OLD) та

архівує їх у таблиці User_log, зберігаючи ідентифікатор, особисті дані та час операції для історії аудиту.

Код тригеру, тепер при оновлені або видалені даних з таблиці User спочатку дані будут продубльовані в таблиці User_log. В полі «operation» зазначено дію, а поле «change_time» відповідно показує час зміни даних.

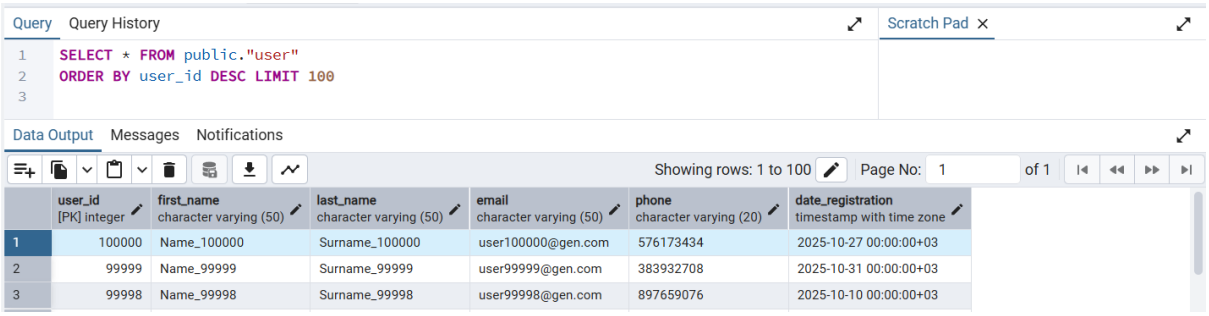
3. Створення тригера за допомогою команди trg_log_users та прикріплення тригеру до відповідної таблиці:



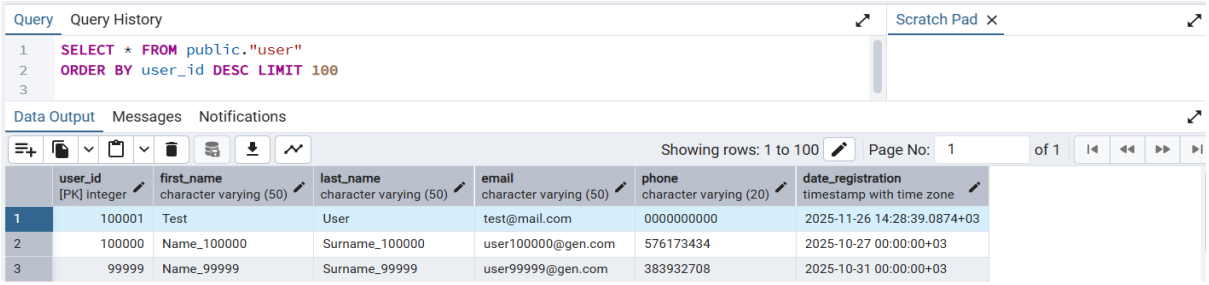
Умова insert для тригера

Виконання insert:

Початкові дані в таблиці User:



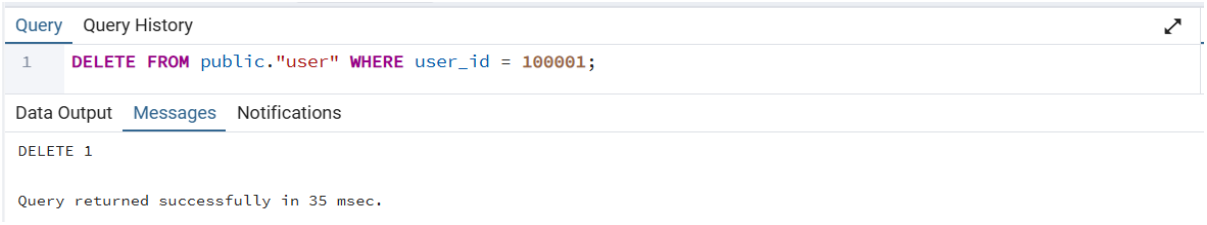
Результат роботи тригера після додавання користувача (insert) до таблиці User:



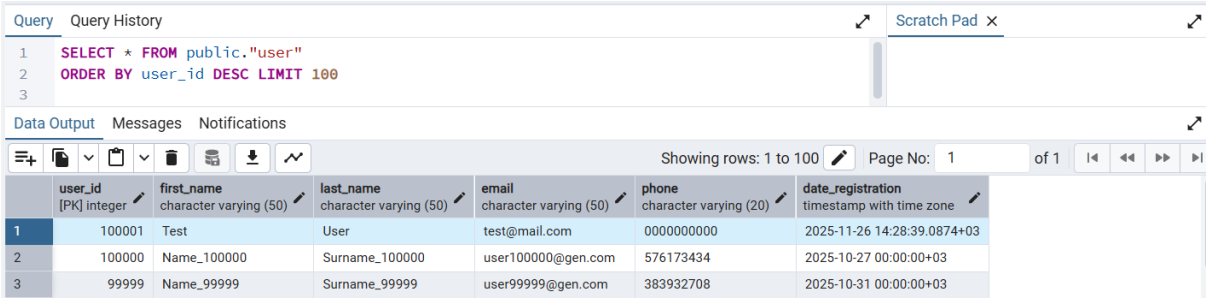
На наведеному скріншоті продемонстровано результат виконання команди insert у таблицю User. Тригер автоматично перехопив подію додавання нового запису та створив відповідний рядок у таблиці аудиту User_log.

Умова after delete для тригера

Виконання after delete:

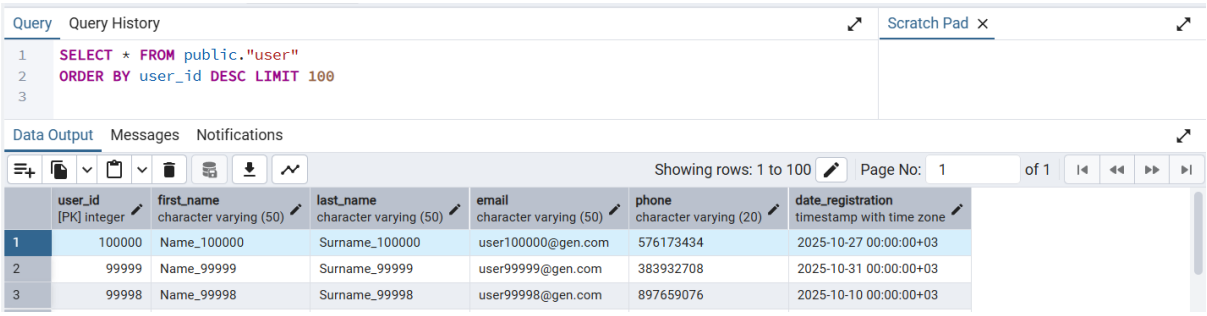


Початкові дані в таблиці User:



	user_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (50)	phone character varying (20)	date_registration timestamp with time zone
1	100001	Test	User	test@mail.com	0000000000	2025-11-26 14:28:39.0874+03
2	100000	Name_100000	Surname_100000	user100000@gen.com	576173434	2025-10-27 00:00:00+03
3	99999	Name_99999	Surname_99999	user99999@gen.com	383932708	2025-10-31 00:00:00+03

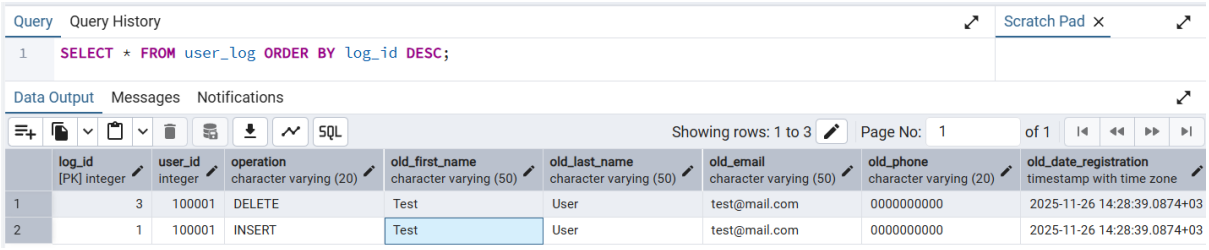
Результат роботи тригера після видалення користувача (after delete) в таблиці User:



	user_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (50)	phone character varying (20)	date_registration timestamp with time zone
1	100000	Name_100000	Surname_100000	user100000@gen.com	576173434	2025-10-27 00:00:00+03
2	99999	Name_99999	Surname_99999	user99999@gen.com	383932708	2025-10-31 00:00:00+03
3	99998	Name_99998	Surname_99998	user99998@gen.com	897659076	2025-10-10 00:00:00+03

На наведеному скріншоті продемонстровано результат виконання команди delete у таблицю User. Тригер автоматично перехопив подію видалення запису та створив відповідний рядок у таблиці аудиту User_log.

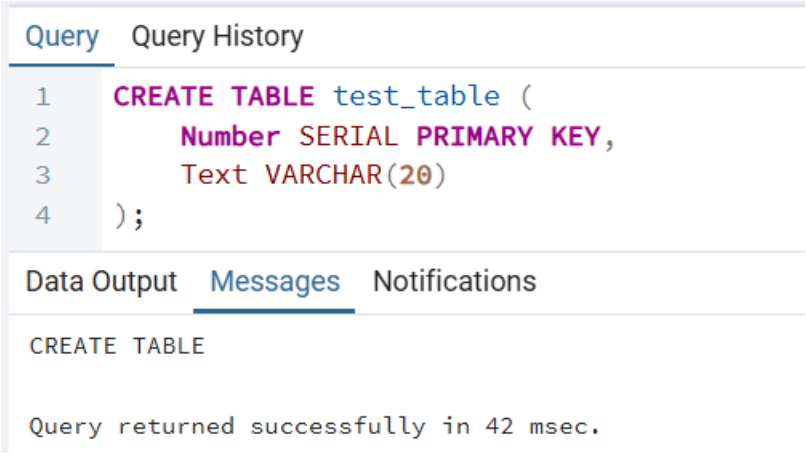
Перевірка занесення даних до логувальної таблиці User_log:



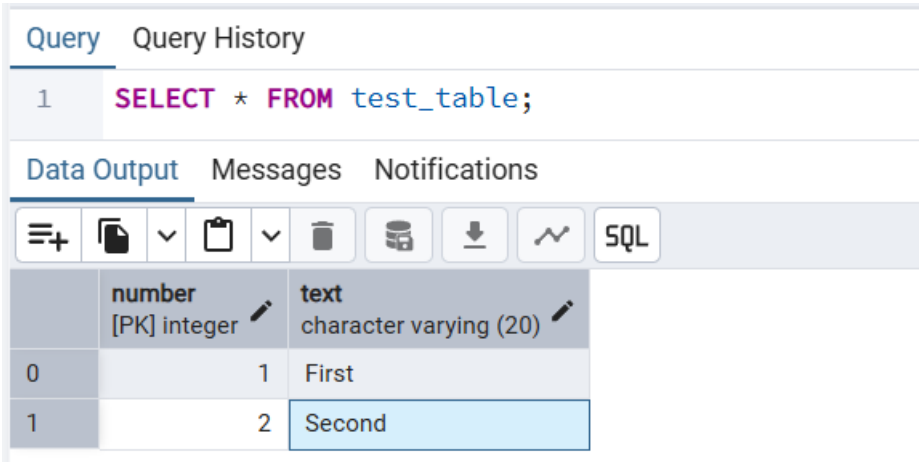
Проведене тестування на кількох різних вхідних даних показало, що тригер стабільно та коректно реагує на всі сценарії, правильно обробляє записи й коректно відпрацьовує помилки. На основі результатів можна зробити висновок, що тригер реалізовано правильно та працює відповідно до поставлених вимог.

Завдання №4. Рівні ізоляції

1. Створення таблиці Test_table для тестування:

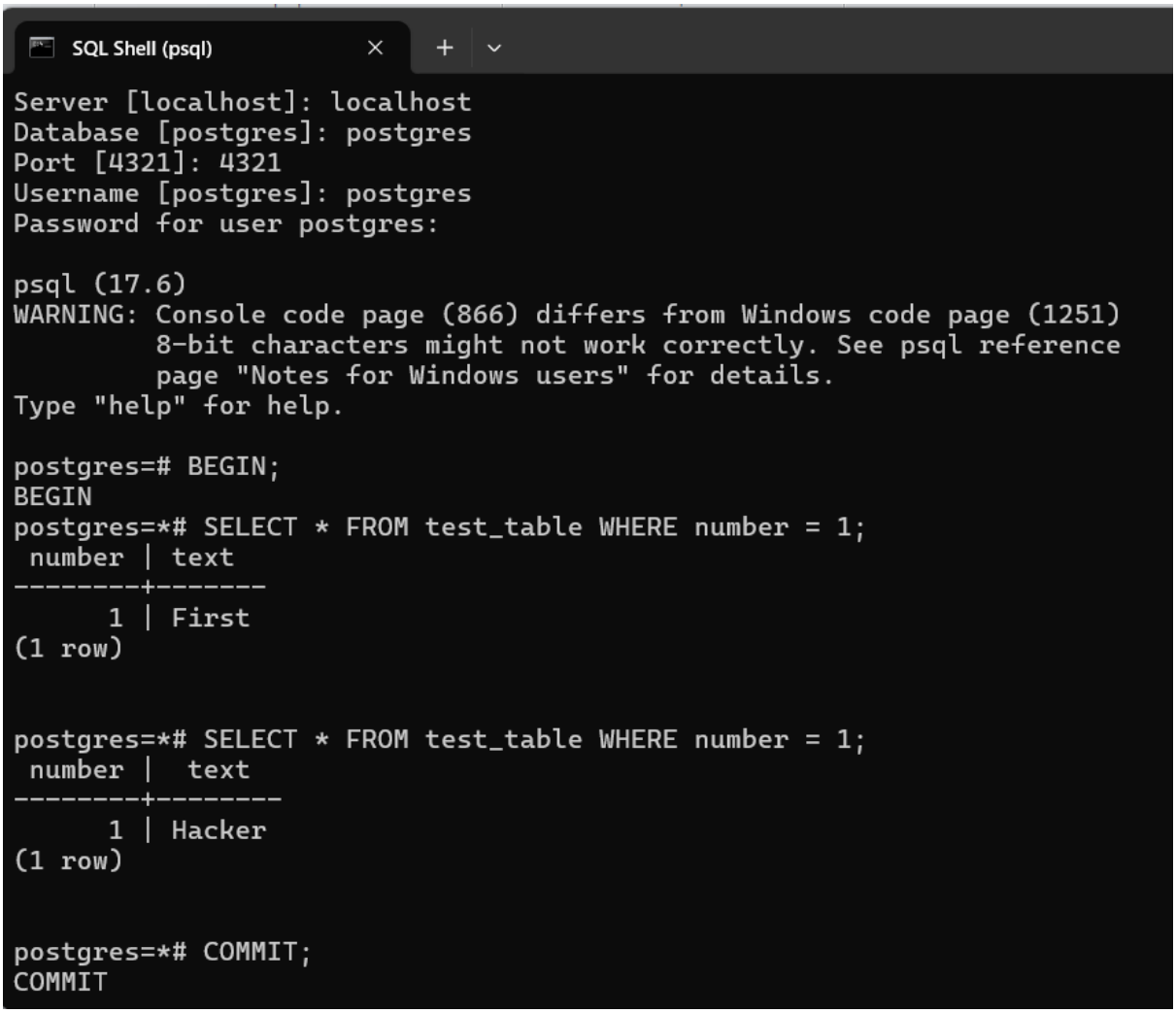


2. Початково занесенні до таблиці дані:

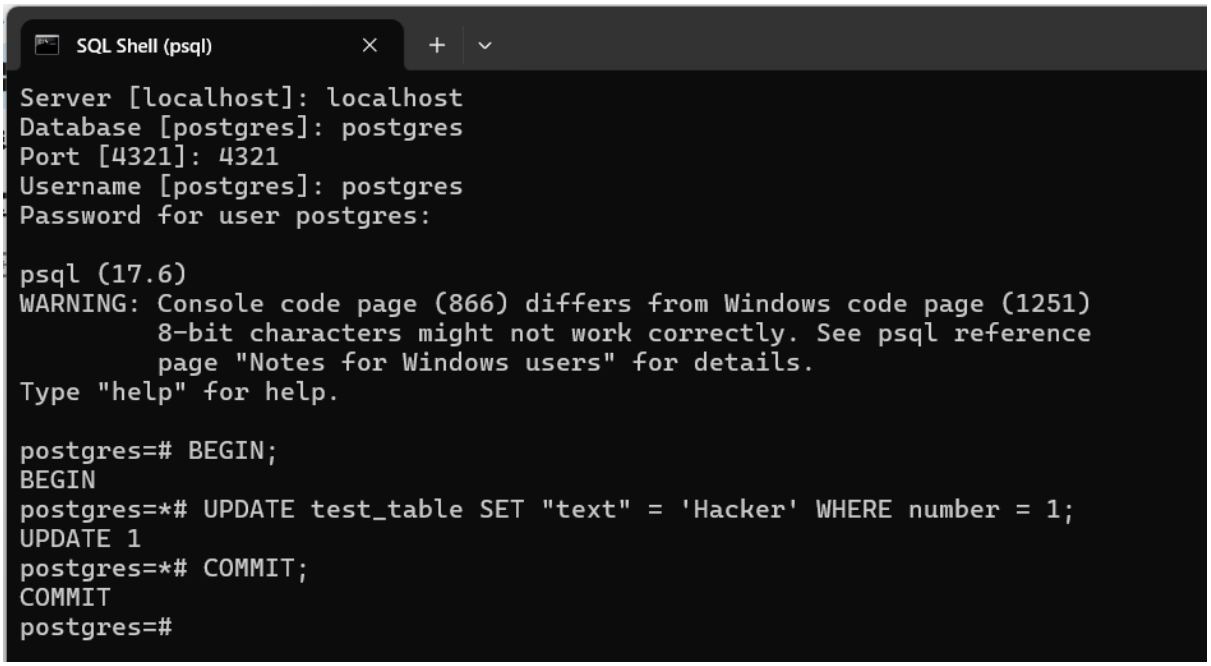


READ COMMITTED

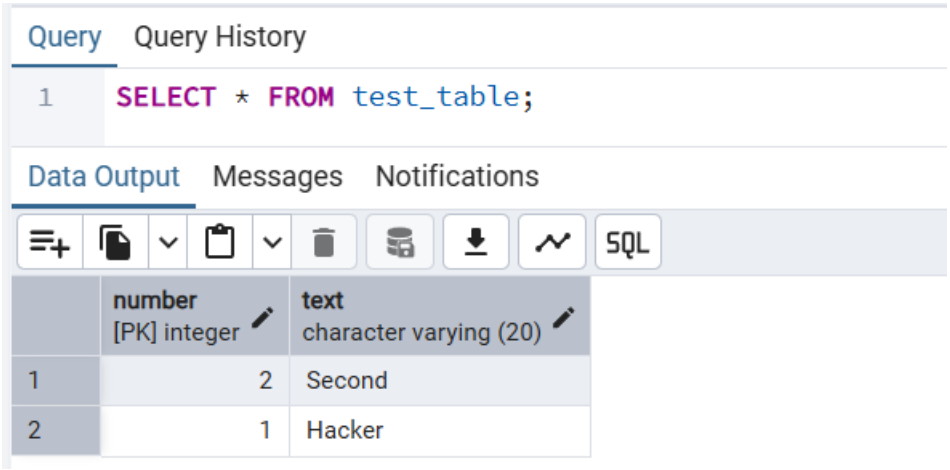
1. Демонстрація рівня READ COMMITED. Спочатку отримуємо початкові дані таблиці. Бачимо, що паралельна транзакція успішно змінила дані на 'Hacker' і зафіксувала їх, поточна транзакція починає бачити нове значення 'Hacker', забезпечуючи читання даних.



2. Нижче продемонстрована транзакція, яка успішно змінила дані (UPDATE) на 'Hacker' і зафіксувала їх.



Кінцеві дані в таблиці:



Рівень ізоляції READ COMMITTED (за замовчуванням у PostgreSQL) запобігає «брудному читанню», гарантуючи, що транзакція бачить лише зафіксовані дані. Однак, він не захищає від змін, внесених іншими транзакціями під час нашої роботи.

В результаті продемонстровано феномен неповторюваного читання:

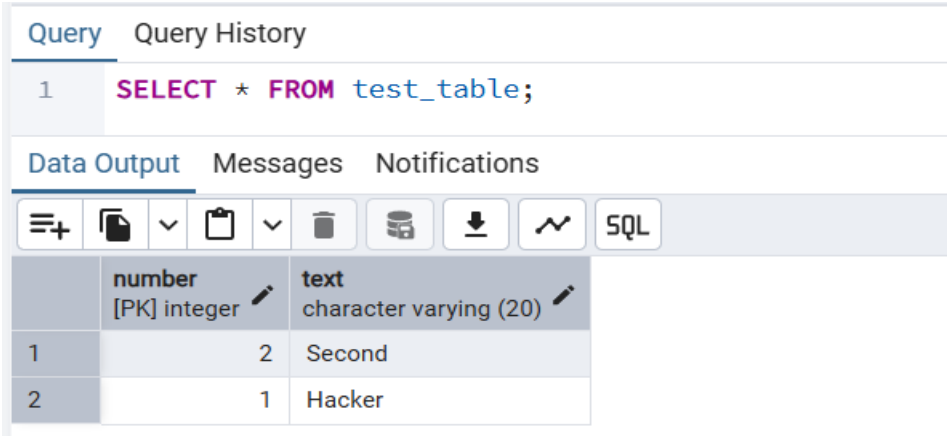
1. Спочатку транзакція зчитала значення 'First'.

2. Паралельна транзакція змінила його на 'Hacker' і виконала фіксацію (COMMIT).

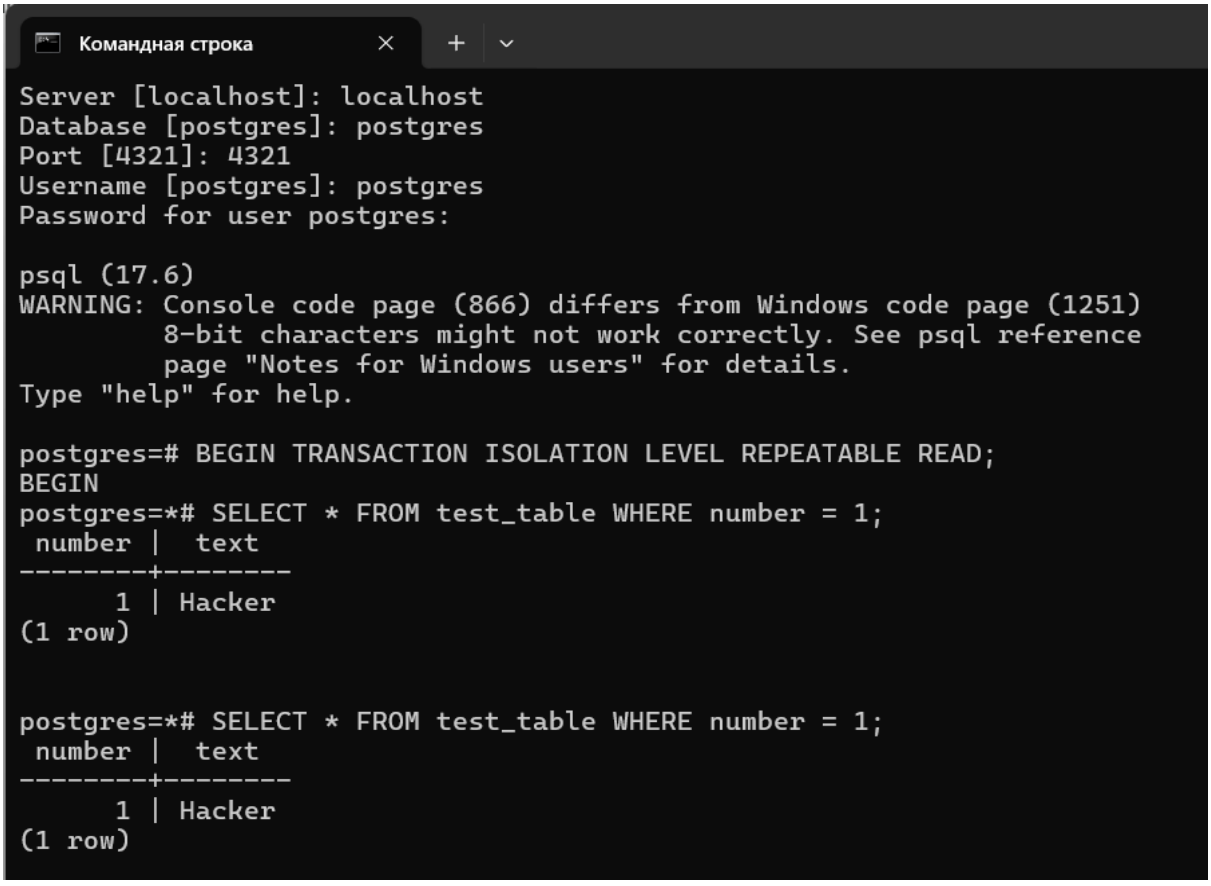
3. Повторний запит у першій транзакції одразу відобразив нове значення 'Hacker', хоча сама транзакція ще не завершилася.

REPEATABLE READ

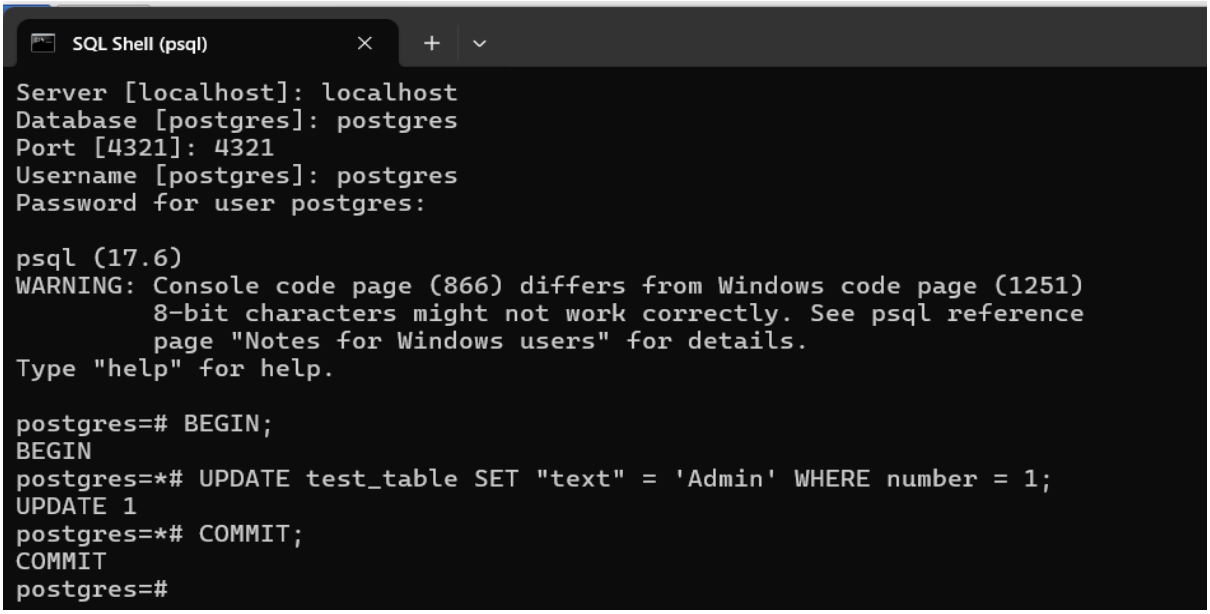
Початкові дані в таблиці:



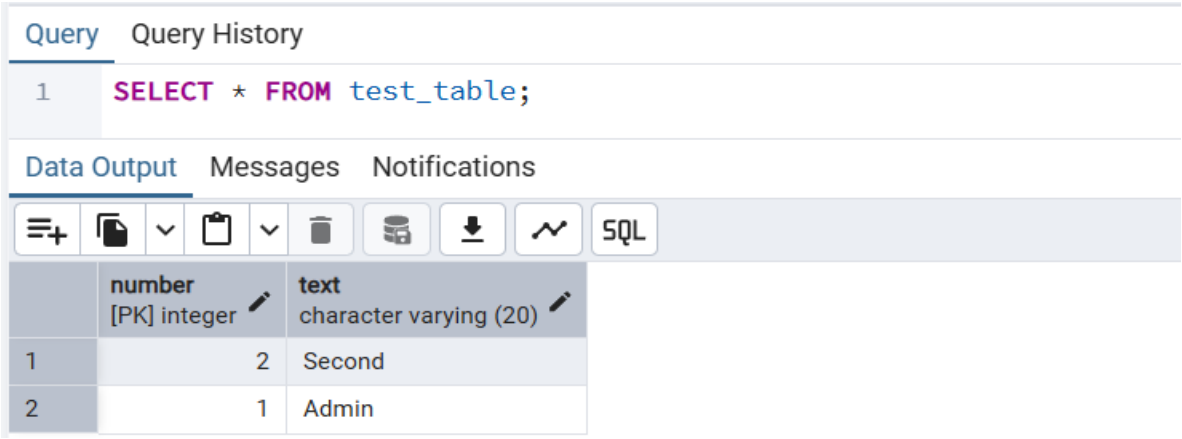
1. Демонстрація рівня REPEATABLE READ. Попри те, що паралельна транзакція успішно змінила дані на 'Admin' і зафіксувала їх, поточна транзакція продовжує бачити старе значення 'Hacker', забезпечуючи ізоляцію даних.



2. Нижче продемонстрована транзакція, яка успішно змінила дані (UPDATE) на 'Admin' і зафіксувала їх.



Кінцеві дані в таблиці:



Рівень ізоляції REPEATABLE READ працює за принципом створення знімка даних (snapshot) станом на момент початку самої транзакції, а не кожного окремого запиту.

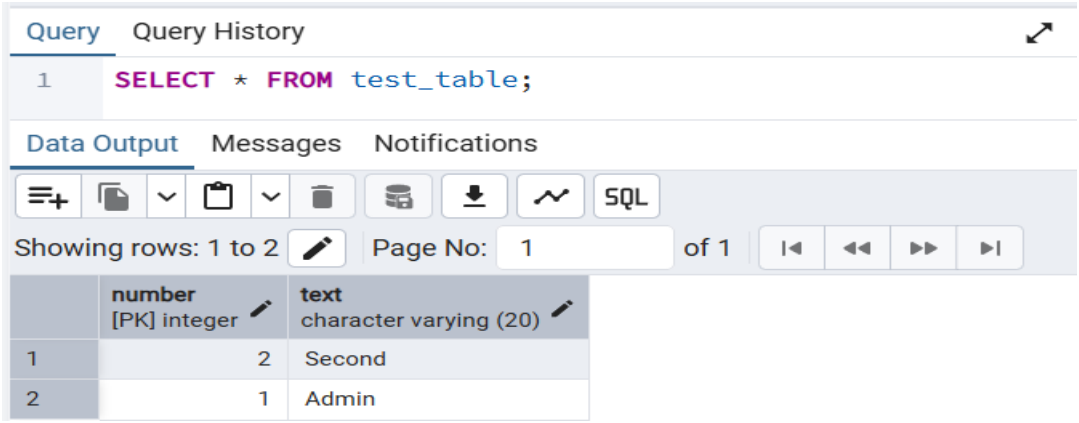
В результаті продемонстровано уникнення феномену «неповторюваного читання»:

1. Поточна транзакція зафіксувала стан даних зі значенням 'Hacker'.
2. Паралельна транзакція виконала оновлення (UPDATE) цього запису і успішно зафіксувала зміни (COMMIT).
3. Попри це, повторний запит у нашій поточній транзакції продовжує повертати старе значення 'Hacker'.

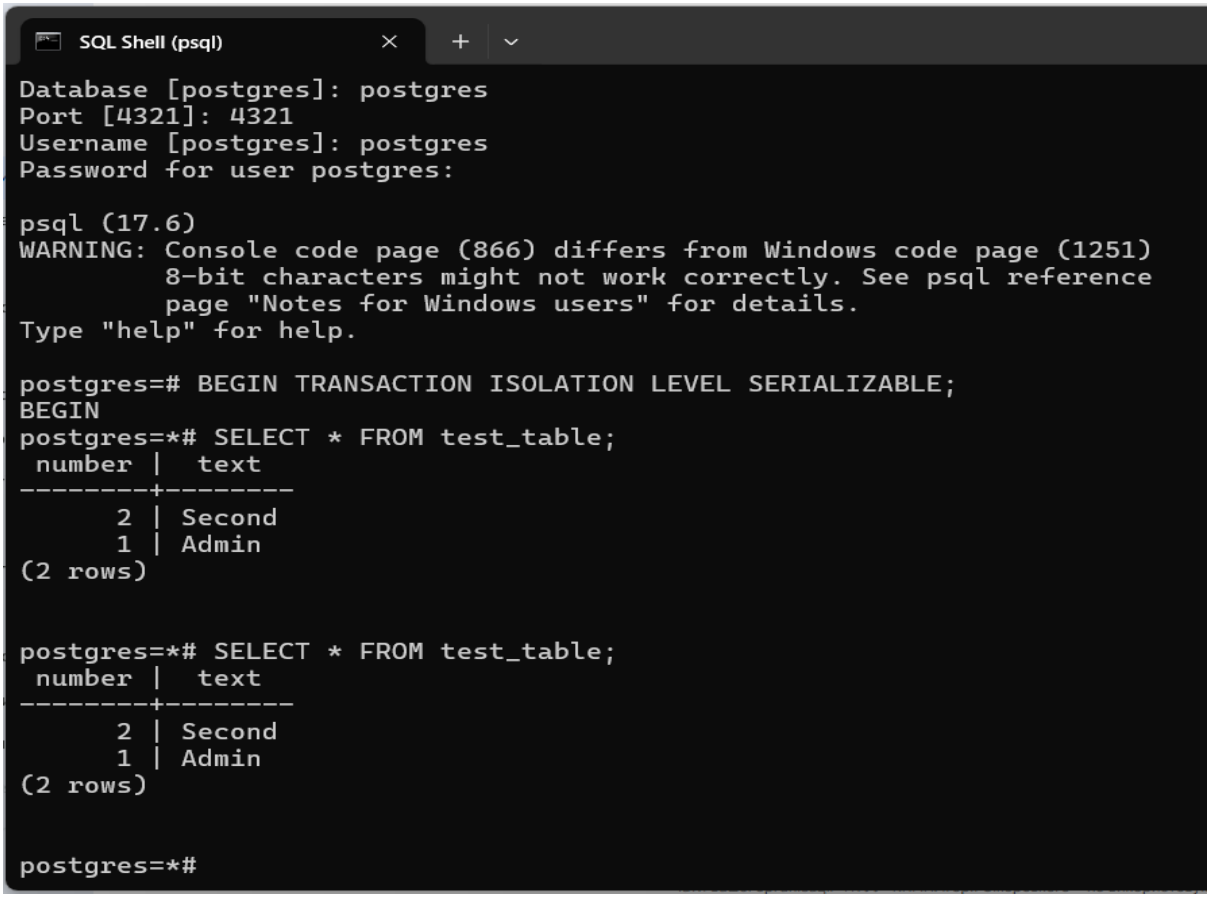
Це підтверджує, що будь-які зміни, внесені іншими транзакціями після початку нашої роботи, ігноруються. Такий механізм гарантує, що дані не зміняться непередбачувано протягом роботи нашої транзакції.

SERIALIZABLE

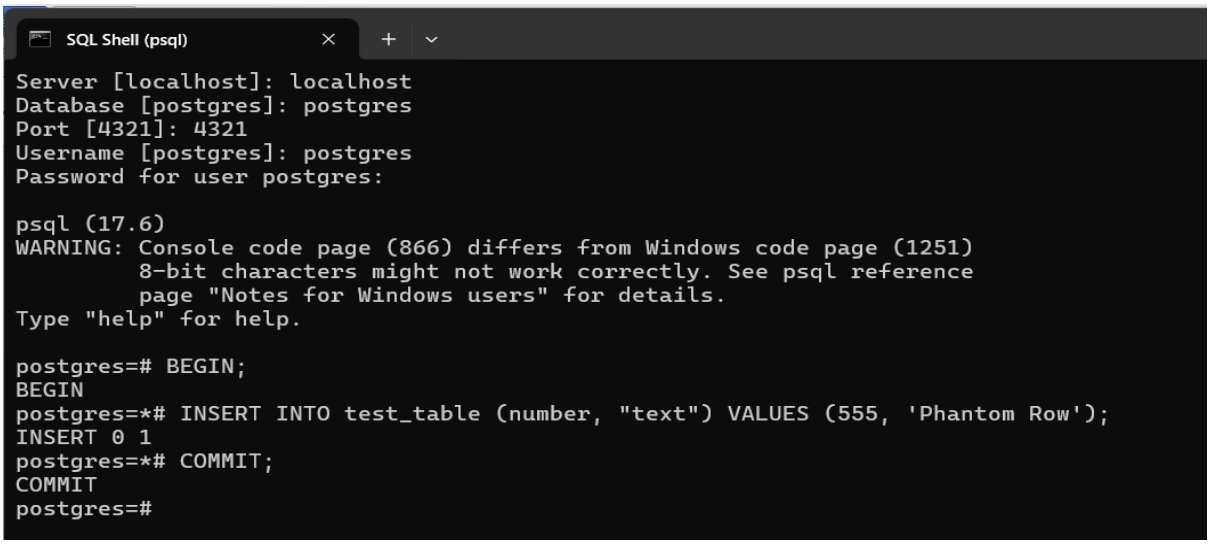
Початкові дані в таблиці:



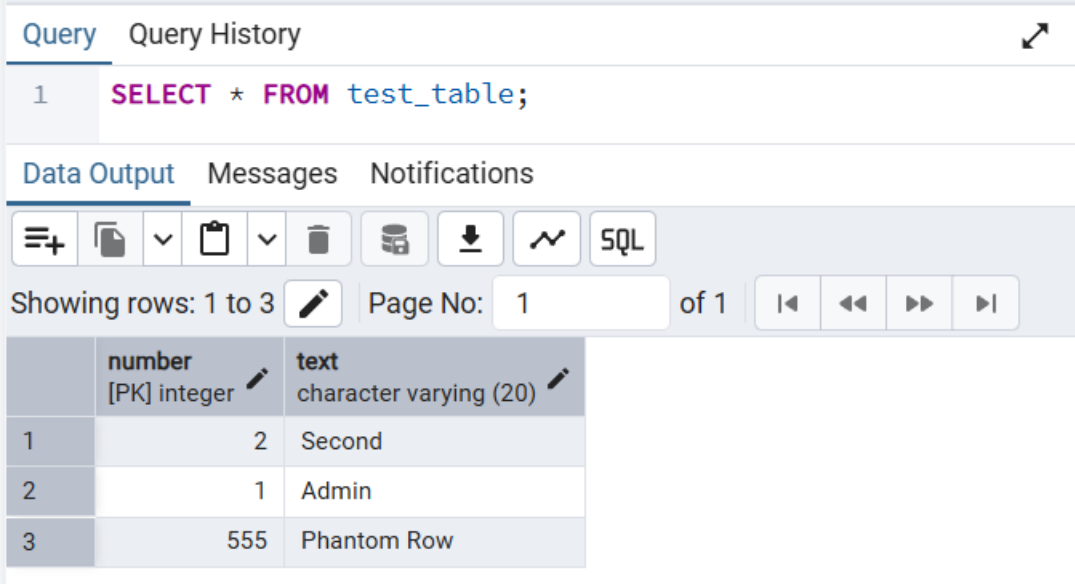
1. Демонстрація рівня SERIALIZABLE. Попри те, що паралельна транзакція успішно додала новий запис (фантомний рядок) і зафіксувала зміни, поточна транзакція не бачить цього нового рядка, забезпечуючи повну ізоляцію та захист від фантомного читання.



2. Нижче продемонстрована транзакція, яка успішно виконала вставку (INSERT) нового рядка в таблицю і зафіксувала зміни.



Кінцеві дані в таблиці:



Рівень ізоляції SERIALIZABLE забезпечує найвищий ступінь ізоляції, емулюючи послідовне виконання транзакцій (одна за одною). Цей рівень гарантує максимальну узгодженість та незалежність даних, повністю виключаючи феномени «брудного», «неповторюваного» та «фантомного» читання, хоча це може впливати на продуктивність через суворіші механізми блокування.

В результаті продемонстровано запобігання феномену фантомного читання:

1. Поточна транзакція виконала запит на вибірку всіх рядків таблиці.

2. Паралельна транзакція додала абсолютно новий запис (INSERT) і успішно зафіксувала зміни (COMMIT).

3. Повторний запит у поточній транзакції не виявив нового рядка (так званого "фантома").

Це підтверджує, що наша транзакція працює з ізольованим знімком даних і захищена від появи нових записів, створених іншими користувачами під час нашої роботи.

Оновлений код model.py

```
import time
import random
import datetime
from functools import wraps
from sqlalchemy import create_engine, Column, Integer, String, Date, Time,
DateTime, ForeignKey, func, text
from sqlalchemy.orm import sessionmaker, declarative_base, relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, primary_key=True)
    first_name = Column(String(50))
    last_name = Column(String(50))
    email = Column(String(50))
    phone = Column(String(20))
    date_registration = Column(DateTime(timezone=True))

    workouts = relationship("UserWorkout", back_populates="user")


class Workout(Base):
    __tablename__ = 'workout'
    workout_id = Column(Integer, primary_key=True)
    type_workout = Column(String(50))

    user_links = relationship("UserWorkout", back_populates="workout")


class UserWorkout(Base):
    __tablename__ = 'user_workout'
    user_workout_id = Column(Integer, primary_key=True)
    user_user_id = Column(Integer, ForeignKey('user.user_id'))
    workout_workout_id = Column(Integer, ForeignKey('workout.workout_id'))
    date = Column(Date)
    time = Column(Time(timezone=True))

    user = relationship("User", back_populates="workouts")
    workout = relationship("Workout", back_populates="user_links")
    health_metric = relationship("HealthMetric", uselist=False,
back_populates="user_workout")


class HealthMetric(Base):
```

```
    __tablename__ = 'health metrics'
    metrics_id = Column(Integer, primary_key=True)
    user_workout_id = Column(Integer,
ForeignKey('user_workout.user_workout_id'))
    steps = Column(Integer)
    pulse = Column(Integer)
    calories = Column(Integer)
    measurement_date = Column(DateTime(timezone=True))

    user_workout = relationship("UserWorkout", back_populates="health_metric")


class Friendship(Base):
    __tablename__ = 'friendship'
    id_composite = Column("user_id1,user_id2", Integer, primary_key=True)
    user_id1 = Column(Integer, ForeignKey('user.user_id'))
    user_id2 = Column(Integer, ForeignKey('user.user_id'))
    status = Column(String(20))
    date = Column(Date)


def timeit(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        elapsed_time = (end_time - start_time) * 1000
        # print(f"\n[INFO] Function '{func.__name__}' executed in
{elapsed_time:.4f} milliseconds\n")
        return result

    return wrapper


class Model:
    def __init__(self):
        try:
            self.engine =
create_engine("postgresql://postgres:1234@localhost:4321/postgres")
            Session = sessionmaker(bind=self.engine)
            self.session = Session()
            print("Database connection successful (SQLAlchemy).")
        except Exception as e:
            print(f"FATAL: Database connection error: {e}")
            self.session = None

    def close_connection(self):
        if self.session:
            self.session.close()

    def _to_tuples(self, query_result):
        return query_result


    def get_all_users(self):
        try:
            result = self.session.query(
                User.user_id, User.first_name, User.last_name,
                User.email, User.phone, User.date_registration
            ).order_by(User.user_id).all()
```

```
            return result
        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    def get_all_workouts(self):
        try:
            result = self.session.query(
                Workout.workout_id, Workout.type_workout
            ).order_by(Workout.workout_id).all()
            return result
        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    def get_all_user_workouts(self):
        try:
            result = self.session.query(
                UserWorkout.user_workout_id, UserWorkout.user_user_id,
                UserWorkout.workout_workout_id, UserWorkout.date,
UserWorkout.time
            ).order_by(UserWorkout.user_workout_id).all()
            return result
        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    def get_all_health_metrics(self):
        try:
            result = self.session.query(
                HealthMetric.metrics_id, HealthMetric.user_workout_id,
                HealthMetric.steps, HealthMetric.pulse,
                HealthMetric.calories, HealthMetric.measurement_date
            ).order_by(HealthMetric.metrics_id).all()
            return result
        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    def get_all_friendships(self):
        try:
            result = self.session.query(
                Friendship.id_composite, Friendship.user_id1,
                Friendship.user_id2, Friendship.status, Friendship.date
            ).order_by(Friendship.date).all()
            return result
        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    @timeit
    def get_data_in_range(self, request):
        try:
            commands = request.split(' ')
            table_name = commands[0]
            field_name = commands[1]
```

```
            start_val = commands[2]
            end_val = commands[3]
            order_field = commands[4]

            model_map = {
                'user': User,
                'workout': Workout,
                'user_workout': UserWorkout,
                'health': HealthMetric,
                '"health metrics"': HealthMetric,
                'friendship': Friendship
            }

            ModelClass = model_map.get(table_name)
            if not ModelClass:
                print("Unknown table")
                return []

            field_attr = getattr(ModelClass, field_name)
            order_attr = getattr(ModelClass, order_field)

            query = self.session.query(ModelClass).filter(
                field_attr.between(start_val, end_val)
            ).order_by(order_attr)

            objs = query.all()
            print(f"\n{len(objs)} rows found.\n")

            result = []
            for obj in objs:
                row = [getattr(obj, col.name) for col in obj.__table__.columns]
                result.append(tuple(row))
            return result

        except Exception as e:
            print(f"Error: {e}")
            self.session.rollback()
            return []

    @timeit
    def get_data_by_field_like(self, request):
        try:
            commands = request.split(' ')
            table_name = commands[0]
            req_field = commands[1]
            search_req = commands[2]
            order_field = commands[3]

            model_map = {
                'user': User, 'workout': Workout, 'user_workout': UserWorkout,
                'health': HealthMetric, 'friendship': Friendship
            }
            ModelClass = model_map.get(table_name)
            if not ModelClass: return []

            field_attr = getattr(ModelClass, req_field)
            order_attr = getattr(ModelClass, order_field)

            objs = self.session.query(ModelClass).filter(
                field_attr.like(f'%{search_req}%')
            ).order_by(order_attr).all()
```

```
            print(f"\n{len(objs)} rows found.\n")

            result = []
            for obj in objs:
                row = [getattr(obj, col.name) for col in obj.__table__.columns]
                result.append(tuple(row))
            return result
        except Exception as e:
            print(f"Error: {e}")
            return []

    @timeit
    def search_workouts_by_username(self, name_pattern):
        try:
            query = self.session.query(
                User.first_name, User.last_name, Workout.type_workout,
UserWorkout.date, UserWorkout.time
            ).join(UserWorkout, User.user_id == UserWorkout.user_user_id) \
                .join(Workout, UserWorkout.workout_workout_id ==
Workout.workout_id) \
                .filter(User.first_name.like(f'%{name_pattern}%')) \
                .order_by(User.last_name, UserWorkout.date)

            result = query.all()
            print(f"\n{len(result)} rows found.\n")
            return result
        except Exception as e:
            print(f"Error: {e}")
            return []

    @timeit
    def search_users_by_pulse(self, min_pulse):
        try:
            query = self.session.query(
                User.user_id, User.first_name, User.last_name,
HealthMetric.pulse
            ).select_from(User) \
                .join(UserWorkout, User.user_id == UserWorkout.user_user_id) \
                .join(HealthMetric, UserWorkout.user_workout_id ==
HealthMetric.user_workout_id) \
                .filter(HealthMetric.pulse > min_pulse) \
                .distinct() \
                .order_by(HealthMetric.pulse.desc())

            result = query.all()
            print(f"\n{len(result)} users found.\n")
            return result
        except Exception as e:
            print(f"Error: {e}")
            return []

    @timeit
    def get_workout_counts_by_user(self):
        try:
            query = self.session.query(
                User.user_id, User.first_name, User.last_name,
                func.count(UserWorkout.user_workout_id).label('workout_count')
            ).outerjoin(UserWorkout, User.user_id == UserWorkout.user_user_id) \
                .group_by(User.user_id, User.first_name, User.last_name) \
                .order_by(text('workout_count DESC'))  # text допомагає
```

```
сортувати за аліасом

            result = query.all()
            print(f"\n{len(result)} users found.\n")
            return result
        except Exception as e:
            print(f"Error: {e}")
            return []

    def add_user(self, first_name, last_name, email, phone, date_registration):
        try:
            max_id = self.session.query(func.max(User.user_id)).scalar() or 0
            new_id = max_id + 1

            new_obj = User(
                user_id=new_id, first_name=first_name, last_name=last_name,
                email=email, phone=phone, date_registration=date_registration
            )
            self.session.add(new_obj)
            self.session.commit()
            print(f"\n1 user added successfully.")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def add_workout(self, type_workout):
        try:
            max_id = self.session.query(func.max(Workout.workout_id)).scalar()
or 0
            new_obj = Workout(workout_id=max_id + 1, type_workout=type_workout)
            self.session.add(new_obj)
            self.session.commit()
            print(f"\n1 workout added successfully.")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def add_user_workout(self, user_id, workout_id, date, time_val):
        try:
            max_id =
self.session.query(func.max(UserWorkout.user_workout_id)).scalar() or 0
            new_obj = UserWorkout(
                user_workout_id=max_id + 1, user_user_id=user_id,
                workout_workout_id=workout_id, date=date, time=time_val
            )
            self.session.add(new_obj)
            self.session.commit()
            print(f"\n1 user_workout added successfully.")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def add_health_metric(self, user_workout_id, steps, pulse, calories,
measurement_date):
        try:
            max_id =
self.session.query(func.max(HealthMetric.metrics_id)).scalar() or 0
            new_obj = HealthMetric(
                metrics_id=max_id + 1, user_workout_id=user_workout_id,
                steps=steps, pulse=pulse, calories=calories,
measurement_date=measurement_date
```

```
            )
            self.session.add(new_obj)
            self.session.commit()
            print(f"\n1 health_metric added successfully.")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def add_friendship(self, user_id1, user_id2, status, date):
        try:
            max_id =
self.session.query(func.max(Friendship.id_composite)).scalar() or 0
            new_obj = Friendship(
                id_composite=max_id + 1, user_id1=user_id1, user_id2=user_id2,
                status=status, date=date
            )
            self.session.add(new_obj)
            self.session.commit()
            print(f"\n1 friendship added successfully.")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def update_user(self, user_id, first_name, last_name, email, phone):
        try:
            q = self.session.query(User).filter(User.user_id == user_id)
            if q.first():
                q.update({
                    User.first_name: first_name, User.last_name: last_name,
                    User.email: email, User.phone: phone
                })
                self.session.commit()
                print(f"\n1 row updated.\n")
            else:
                print("User not found")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def update_workout(self, workout_id, type_workout):
        try:
            q = self.session.query(Workout).filter(Workout.workout_id ==
workout_id)
            if q.first():
                q.update({Workout.type_workout: type_workout})
                self.session.commit()
                print(f"\n1 row updated.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def update_user_workout(self, user_workout_id, user_id, workout_id, date,
time_val):
        try:
            q =
self.session.query(UserWorkout).filter(UserWorkout.user_workout_id ==
user_workout_id)
            if q.first():
                q.update({
                    UserWorkout.user_user_id: user_id,
UserWorkout.workout_workout_id: workout_id,
```

```
                    UserWorkout.date: date, UserWorkout.time: time_val
                })
                self.session.commit()
                print(f"\n1 row updated.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def update_health_metric(self, metrics_id, steps, pulse, calories,
measurement_date):
        try:
            q = self.session.query(HealthMetric).filter(HealthMetric.metrics_id
== metrics_id)
            if q.first():
                q.update({
                    HealthMetric.steps: steps, HealthMetric.pulse: pulse,
                    HealthMetric.calories: calories,
HealthMetric.measurement_date: measurement_date
                })
                self.session.commit()
                print(f"\n1 row updated.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def update_friendship(self, user_id1_key, user_id2_key, new_status):
        try:
            q = self.session.query(Friendship).filter(
                Friendship.user_id1 == user_id1_key,
                Friendship.user_id2 == user_id2_key
            )
            if q.first():
                q.update({Friendship.status: new_status})
                self.session.commit()
                print(f"\n1 row updated.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")

    def delete_data(self, table_name, field, value):
        try:
            model_map = {
                'user': User, 'workout': Workout, 'user_workout': UserWorkout,
                'health': HealthMetric, '"health metrics"': HealthMetric,
                'friendship': Friendship
            }
            ModelClass = model_map.get(table_name)
            if not ModelClass:
                print("Unknown table")
                return

            field_attr = getattr(ModelClass, field)

            rows_deleted = self.session.query(ModelClass).filter(field_attr ==
value).delete()
            self.session.commit()
            print(f"\n{rows_deleted} rows deleted successfully! ")
        except Exception as e:
            self.session.rollback()
            print(f"Error: {e}")
```

```
    @timeit
    def generate_users(self, num_users):
        try:
            max_id = self.session.query(func.max(User.user_id)).scalar() or 0
            new_users = []

            for i in range(1, int(num_users) + 1):
                uid = max_id + i
                u = User(
                    user_id=uid,
                    first_name=f'Name_{uid}',
                    last_name=f'Surname_{uid}',
                    email=f'user{uid}@gen.com',
                    phone=str(random.randint(100000000, 999999999)),
                    date_registration=datetime.date.today() -
datetime.timedelta(days=random.randint(0, 365))
                )
                new_users.append(u)

            self.session.add_all(new_users)
            self.session.commit()
            print(f"\n{num_users} users generated successfully.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Generation error: {e}")

    @timeit
    def generate_workouts(self, num_workouts):
        try:
            max_id = self.session.query(func.max(Workout.workout_id)).scalar()
or 0
            types = ['Running', 'Weightlifting', 'Yoga', 'Cycling']
            new_data = []

            for i in range(1, int(num_workouts) + 1):
                new_data.append(Workout(
                    workout_id=max_id + i,
                    type_workout=random.choice(types)
                ))

            self.session.add_all(new_data)
            self.session.commit()
            print(f"\n{num_workouts} workouts generated successfully.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Generation error: {e}")

    @timeit
    def generate_user_workouts(self, num_records):
        try:
            u_ids = [r[0] for r in self.session.query(User.user_id).all()]
            w_ids = [r[0] for r in self.session.query(Workout.workout_id).all()]

            if not u_ids or not w_ids:
                print("Need users and workouts first.")
                return

            max_id =
self.session.query(func.max(UserWorkout.user_workout_id)).scalar() or 0
            new_data = []
```

```
            for i in range(1, int(num_records) + 1):
                new_data.append(UserWorkout(
                    user_workout_id=max_id + i,
                    user_user_id=random.choice(u_ids),
                    workout_workout_id=random.choice(w_ids),
                    date=datetime.date.today() -
datetime.timedelta(days=random.randint(0, 30)),
                    time=datetime.time(random.randint(0, 23), random.randint(0,
59))
                ))

            self.session.add_all(new_data)
            self.session.commit()
            print(f"\n{num_records} user_workouts generated successfully.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Generation error: {e}")

    @timeit
    def generate_health_metrics(self, num_records):
        try:
            uw_ids = [r[0] for r in
self.session.query(UserWorkout.user_workout_id).limit(int(num_records)).all()]

            max_id =
self.session.query(func.max(HealthMetric.metrics_id)).scalar() or 0
            new_data = []

            for i, uw_id in enumerate(uw_ids):
                new_data.append(HealthMetric(
                    metrics_id=max_id + i + 1,
                    user_workout_id=uw_id,
                    steps=random.randint(1000, 20000),
                    pulse=random.randint(60, 180),
                    calories=random.randint(100, 1000),
                    measurement_date=datetime.date.today()
                ))

            self.session.add_all(new_data)
            self.session.commit()
            print(f"\nMetrics generated (attempted).\n")
        except Exception as e:
            self.session.rollback()
            print(f"Generation error: {e}")

    @timeit
    def generate_friendships(self, num_records):
        try:
            u_ids = [r[0] for r in self.session.query(User.user_id).all()]
            if len(u_ids) < 2: return

            max_id =
self.session.query(func.max(Friendship.id_composite)).scalar() or 0
            new_data = []

            for i in range(1, int(num_records) + 1):
                u1 = random.choice(u_ids)
                u2 = random.choice(u_ids)
                if u1 == u2: continue

                new_data.append(Friendship(
```

```
                    id_composite=max_id + i,
                    user_id1=u1,
                    user_id2=u2,
                    status=random.choice(['pending', 'accepted', 'blocked']),
                    date=datetime.date.today()
                ))

            self.session.add_all(new_data)
            self.session.commit()
            print(f"\nFriendships generated.\n")
        except Exception as e:
            self.session.rollback()
            print(f"Generation error: {e}")
```

Посилання на GitHub: https://github.com/MaloivanVladyslav/maloiva_lab2