



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

Лабораторна робота №2

«Засоби оптимізації роботи СУБД PostgreSQL»

З дисципліни «Бази даних та засоби управління»

Виконав студент групи: KB-32

ПІБ: Клубук Максим Віталійович

Контакт в телеграм: @ReinlaughT

Github: [посилання](#)

Перевірив: _____

Мета роботи

здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Варіант

Електронний журнал для ведення особистого щоденника.

Постановка задачі

1. Перетворити модуль “Модель” з шаблону MVC PGP 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.

Хід роботи

Структура бази даних:

Сутності:

Користувач (User) – ця сутність представляє зареєстровану особу в системі, яка є власником інформації. Це головна сутність, навколо якої будується робота програми. У цій сутності ми маємо такі атрибути, як:

id (Integer) – унікальний ідентифікатор користувача.

username (Varchar) – логін для входу в систему.

email (Varchar) – електронна пошта для контакту та ідентифікації.

password (Varchar) – зашифрований пароль для доступу.

Запис (Entry) – ця сутність представляє конкретну нотатку або сторінку щоденника. Це основний контент, який створює користувач. У цій сутності ми маємо такі атрибути, як:

entry_id (Integer) – унікальний ідентифікатор запису.

title (Varchar) – заголовок нотатки.

text (Varchar) – основний текст або зміст нотатки.

user_id (Integer) – посилання на власника (користувача).

Нагадування (Reminder) – ця сутність представляє заплановане сповіщення, яке логічно прив'язане до конкретного запису (наприклад, нагадати про важливу зустріч, описану в нотатці). У цій сутності ми маємо такі атрибути, як:

reminder_id (Integer) – унікальний ідентифікатор нагадування.

remind_at (Timestamp) – точна дата та час спрацювання нагадування.

active (Boolean) – статус (активне/виконане).

entry_id (Integer) – посилання на запис, до якого створено нагадування.

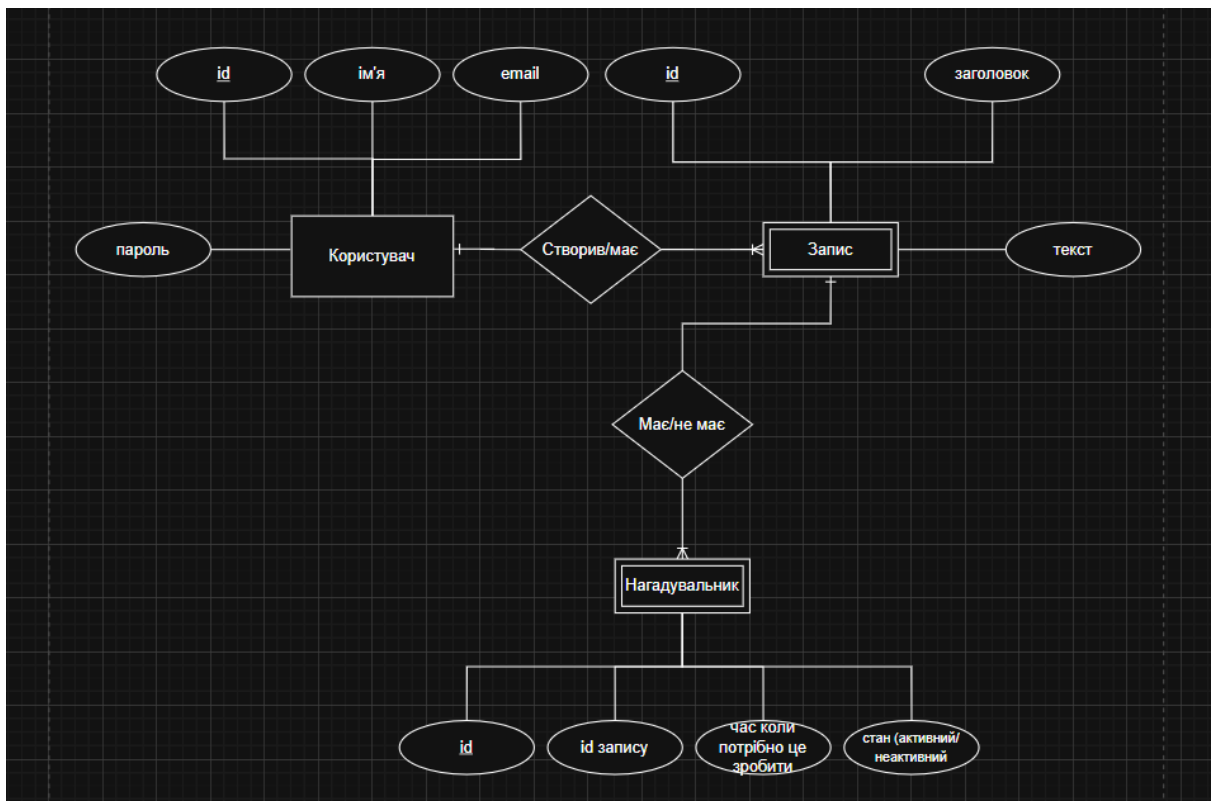
Зв'язки:

1:N (Один до багатьох) – між Користувачем та Записом.

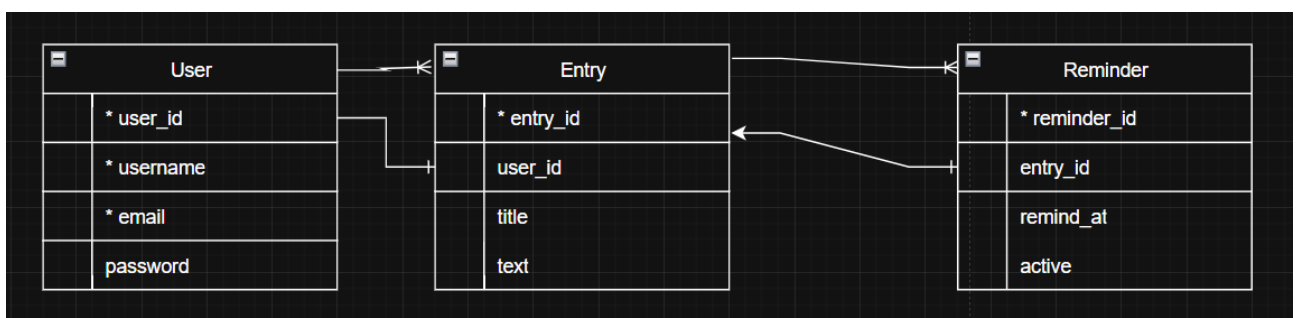
Один користувач може створити безліч записів, але кожен конкретний запис належить лише одному користувачу.

1:N (Один до багатьох) – між Записом та Нагадуванням.

До одного запису можна створити декілька нагадувань (наприклад, за тиждень до події і в день події), але кожне нагадування стосується лише одного конкретного запису.



ER діаграма виконана за нотацією «Пташина лапка»



Структура бази даних

Завдання 1:

Для реалізації об'єктно-реляційної проекції (ORM) використано бібліотеку **SQLAlchemy**. Структура бази даних складається з трьох сутностей: **User**, **Entry** та **Reminder**, які пов'язані між собою відношеннями "Один-до-Багатьох" (1:M).

```
1 import time
2 from sqlalchemy import create_engine, Column, Integer, String, Boolean, DateTime, ForeignKey, func, text, select
3 from sqlalchemy.orm import declarative_base, sessionmaker, relationship, joinedload
4 from sqlalchemy.exc import IntegrityError, SQLAlchemyError
5
6 # --- ОГЛОШЕННЯ БАЗОВИХ КЛАСІВ ORM ---
7 Base = declarative_base()
8
9
10 class User(Base): 27 usages
11     __tablename__ = 'user'
12     __table_args__ = {'schema': 'public'}
13
14     id = Column(*args: Integer, primary_key=True) # Autoincrement за замовчуванням
15     username = Column(*args: String(50), unique=True, nullable=False)
16     email = Column(*args: String(100), unique=True, nullable=False)
17     password = Column(*args: String(100), nullable=False)
18
19     # Зв'язок 1:M (User -> Entries)
20     entries = relationship("Entry", back_populates="user", cascade="all, delete-orphan")
21
```

```
23 class Entry(Base): 19 usages
24     __tablename__ = 'entry'
25     __table_args__ = {'schema': 'public'}
26
27     entry_id = Column(*args: Integer, primary_key=True)
28     title = Column(*args: String(50), nullable=False)
29     text = Column(*args: String, nullable=False)
30     user_id = Column(*args: Integer, ForeignKey('public.user.id'), nullable=False)
31
32     # Зв'язки
33     user = relationship("User", back_populates="entries")
34     reminders = relationship("Reminder", back_populates="entry", cascade="all, delete-orphan")
35
36
37 class Reminder(Base): 19 usages
38     __tablename__ = 'reminder'
39     __table_args__ = {'schema': 'public'}
40
41     reminder_id = Column(*args: Integer, primary_key=True)
42     entry_id = Column(*args: Integer, ForeignKey('public.entry.entry_id'), nullable=False)
43     remind_at = Column(*args: DateTime, nullable=False)
44     active = Column(*args: Boolean, default=True)
45
46     # Зв'язки
47     entry = relationship("Entry", back_populates="reminders")
48
49
```

Реалізація основних CRUD-операцій та складного пошуку засобами SQLAlchemy

1. Вставка даних (INSERT)

Створення нового об'єкта та додавання його до сесії.

```

103 # --- ДОДАВАННЯ (CREATE) ---
104 def add_user(self, user_id, username, email, password): 1 usage
105     try:
106         # Якщо user_id передано явно, використовуємо його, інакше автоінкремент
107         new_user = User(username=username, email=email, password=password)
108         if user_id is not None:
109             new_user.id = user_id
110
111         self.session.add(new_user)
112         self.session.commit()
113         return "Користувача успішно додано."
114     except IntegrityError:
115         self.session.rollback()
116         return "Помилка: Такий ID, email або username вже існує."
117     except Exception as e:
118         self.session.rollback()
119         return f"Помилка: {e}"
120

```

2. Вибірка даних (SELECT) з фільтрацією та сортуванням

Отримання списку записів з обмеженням кількості (TOP-10).

```

82 def get_top_users(self, limit=10): 1 usage
83     try:
84         users = self.session.query(User).order_by(User.id).limit(limit).all()
85         return [(u.id, u.username, u.email) for u in users]
86     except Exception as e:
87         return str(e)

```

3. Редагування даних (UPDATE)

Пошук об'єкта за ID, зміна його атрибутів та збереження змін.

```

534 # --- РЕДАГУВАННЯ (UPDATE) ---
535 def update_user(self, current_id, new_id, new_username, new_email, new_password): 1 usage
536     try:
537         user = self.session.get(User, current_id)
538         if not user:
539             return "Користувача не знайдено."
540
541         # Оновлюємо поля
542         user.id = new_id
543         user.username = new_username
544         user.email = new_email
545         user.password = new_password
546
547         self.session.commit()
548         return "Користувача успішно оновлено."
549     except IntegrityError:
550         self.session.rollback()
551         return "Помилка: Такий ID, Username або Email вже зайняті, або ID використовується в FK."
552     except Exception as e:
553         self.session.rollback()
554         return f"Помилка оновлення: {e}"
555

```

4. Видалення даних (DELETE)

Видалення об'єкта. Завдяки налаштуванню `cascade="all, delete-orphan"` у моделі, видалення `User` автоматично видалить пов'язані `Entry` та `Reminder`.

```
154 # --- ВИДАЛЕННЯ (DELETE) ---
155 def delete_user(self, user_id): 1 usage
156     try:
157         user = self.session.get(User, user_id)
158         if not user:
159             return "ID не знайдено."
160
161         self.session.delete(user)
162         self.session.commit()
163         return f"Користувача {user_id} видалено."
164     except IntegrityError:
165         self.session.rollback()
166         return "Неможливо видалити: є пов'язані записи (спробуйте каскадне видалення)."
167     except Exception as e:
168         self.session.rollback()
169         return f"Помилка: {e}"
```

5. Складний запит (JOIN, Aggregation, Filtering)

Гнучкий пошук, що об'єднує три таблиці (`User` -> `Entry` -> `Reminder`) та рахує кількість записів.

```
381 # --- ГНУЧКИЙ ПОШУК ---
382 def search_flexible(self, filters): 1 usage
383     start_time = time.time()
384     try:
385         query = self.session.query(
386             User.id,
387             User.username,
388             User.email,
389             func.count(func.distinct(Entry.entry_id)),
390             func.count(func.distinct(Reminder.reminder_id))
391         ).outerjoin(Entry, User.id == Entry.user_id) \
392             .outerjoin(Reminder, Entry.entry_id == Reminder.entry_id)
393
394         # Застосування фільтрів
395         if filters.get('username'):
396             query = query.filter(User.username.ilike(f"%{filters['username']}%"))
397         if filters.get('email'):
398             query = query.filter(User.email.ilike(f"%{filters['email']}%"))
399         if filters.get('title'):
400             query = query.filter(Entry.title.ilike(f"%{filters['title']}%"))
401         if filters.get('text'):
402             query = query.filter(Entry.text.ilike(f"%{filters['text']}%"))
403         if filters.get('is_active') is not None:
404             query = query.filter(Reminder.active == filters['is_active'])
405         if filters.get('date_from') and filters.get('date_to'):
406             query = query.filter(Reminder.remind_at.between(filters['date_from'], filters['date_to']))
407
408         # Групування
409         query = query.group_by(User.id, User.username, User.email).order_by(User.id)
```

```

410
411         results = query.all()
412         # Результат вже є списком кортежів (tuple), конвертація не потрібна
413
414         exec_time = (time.time() - start_time) * 1000
415         return results, exec_time
416     except Exception as e:
417         return str(e), 0
418

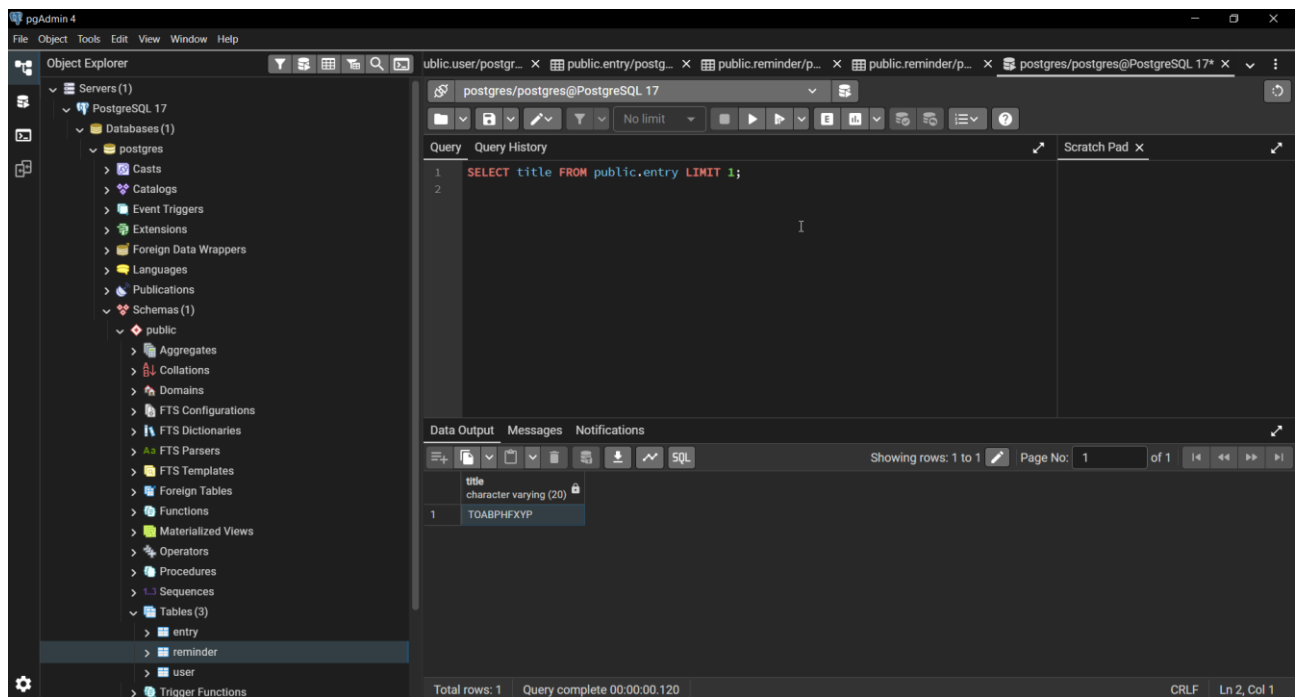
```

Завдання 2:

Варіант №9: Типи індексів **B-Tree** та **BRIN**.

Тестові дані: Для проведення експерименту було згенеровано по **100,000 записів** у таблицях User, Entry та Reminder за допомогою створеного в минулій РГР Python-скрипта.

1. Виконання запиту до створення індексів



The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer shows the database structure, including the 'public' schema and its tables. The main pane displays a query window with the following SQL query:

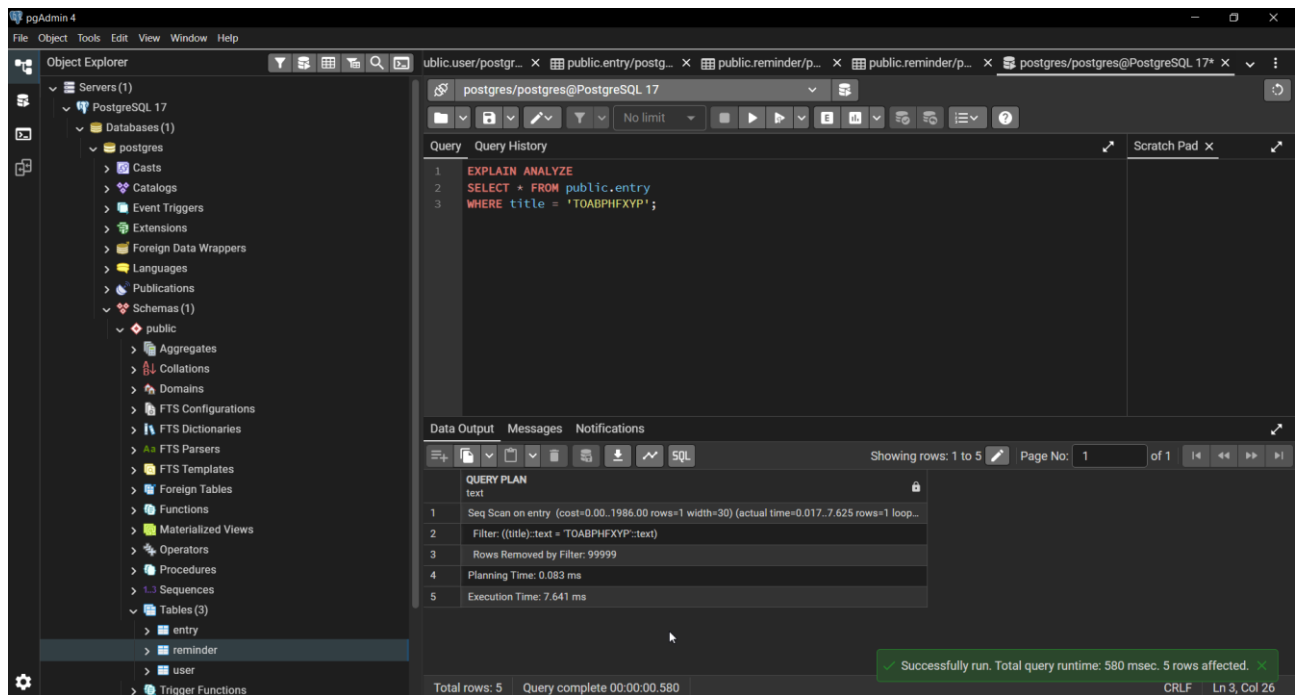
```
SELECT title FROM public.entry LIMIT 1;
```

The query has been executed, and the results are shown in the Data Output pane. The results table has one row with the title 'TOABPHFXYP'.

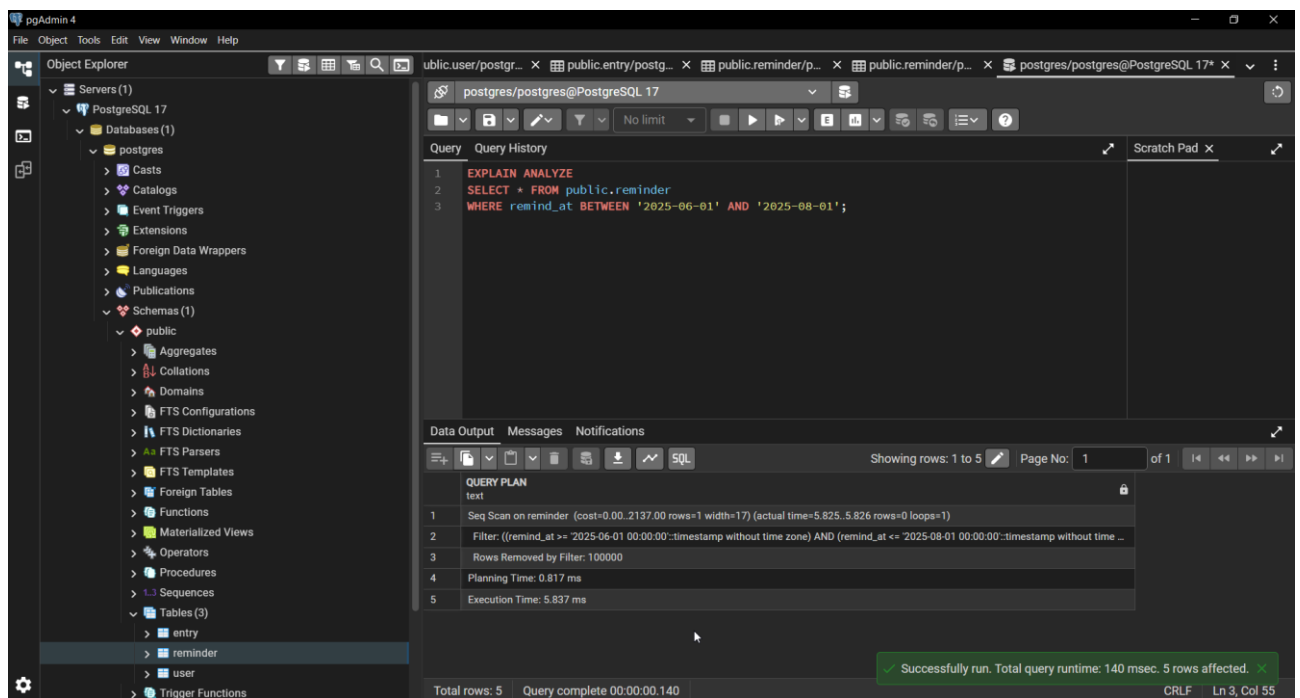
| title |
|------------|
| TOABPHFXYP |

The status bar at the bottom indicates 'Total rows: 1' and 'Query complete 00:00:00.120'.

Проведемо тестування пошуку за заданим “title” для Запису, а потім створимо



Проведемо тестування пошуку за заданим діапазоном дат для Нагадувальника

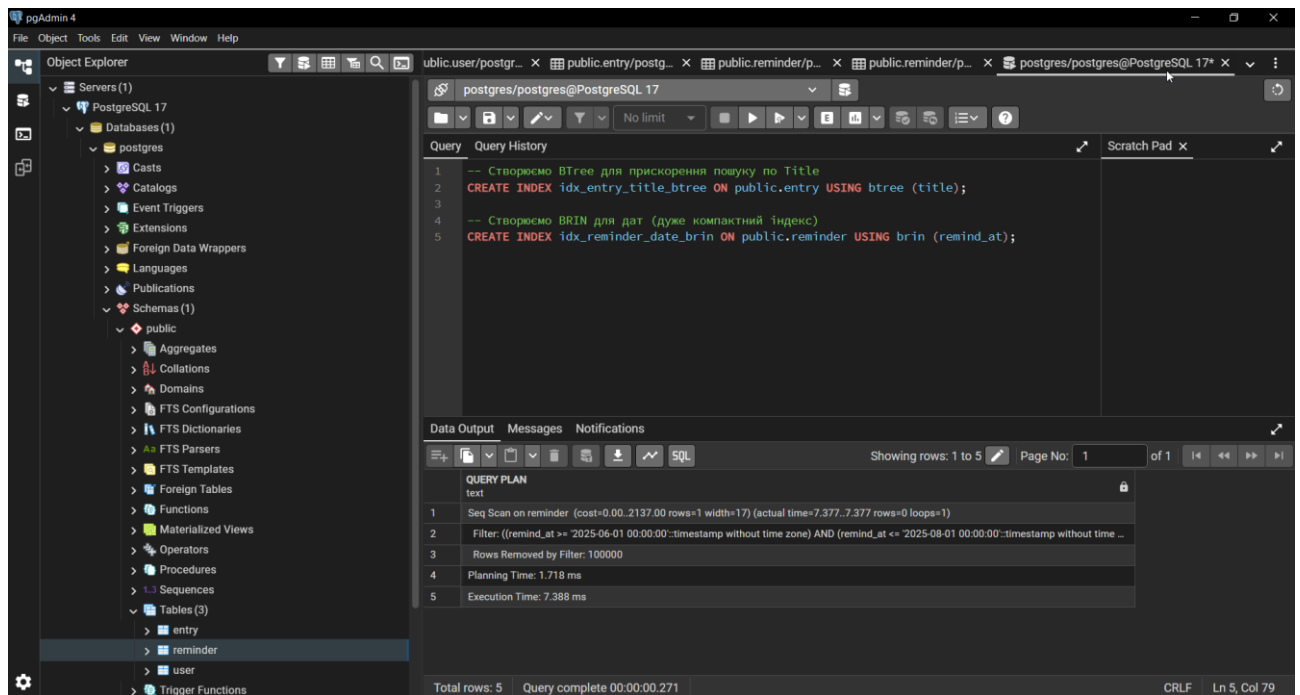


-- 1. Створення індексу B-Tree для текстового поля (пошук по назві)

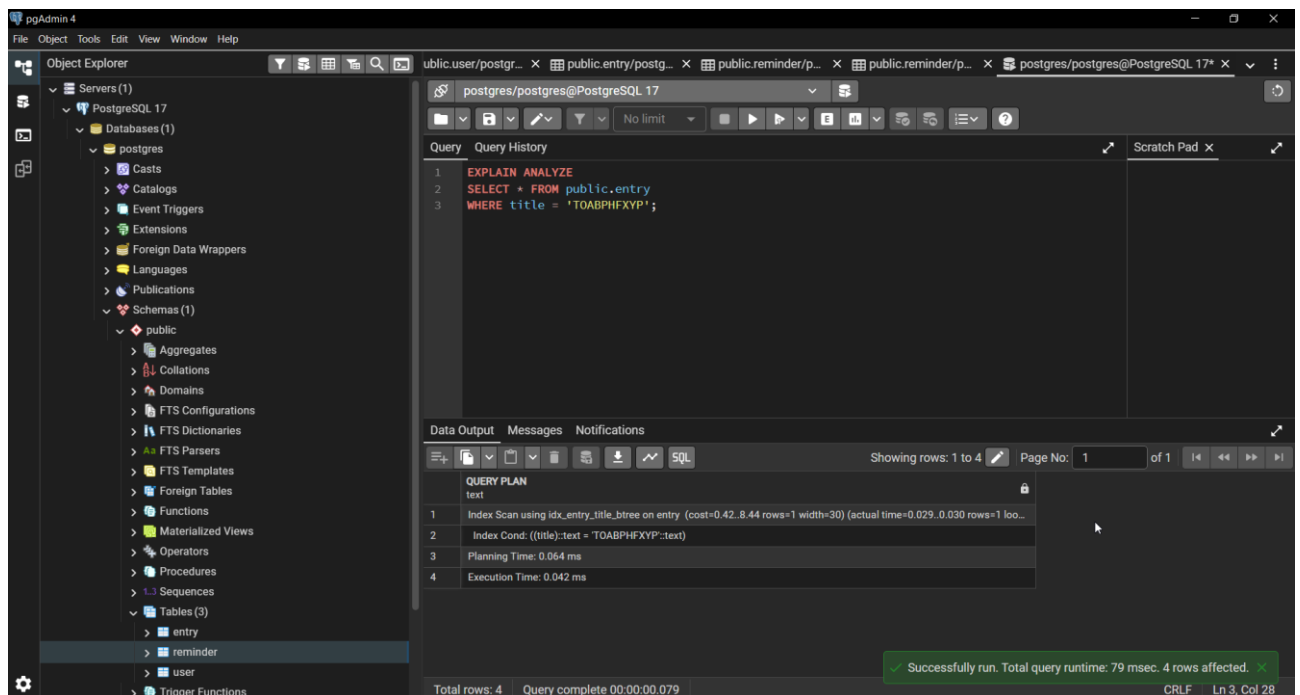
CREATE INDEX idx_entry_title_btree ON public.entry USING btree (title);

-- 2. Створення індексу BRIN для поля дати

CREATE INDEX idx_reminder_date_brin ON public.reminder USING brin (remind_at);



2. Тест зі створеними індексами BTree та BRIN

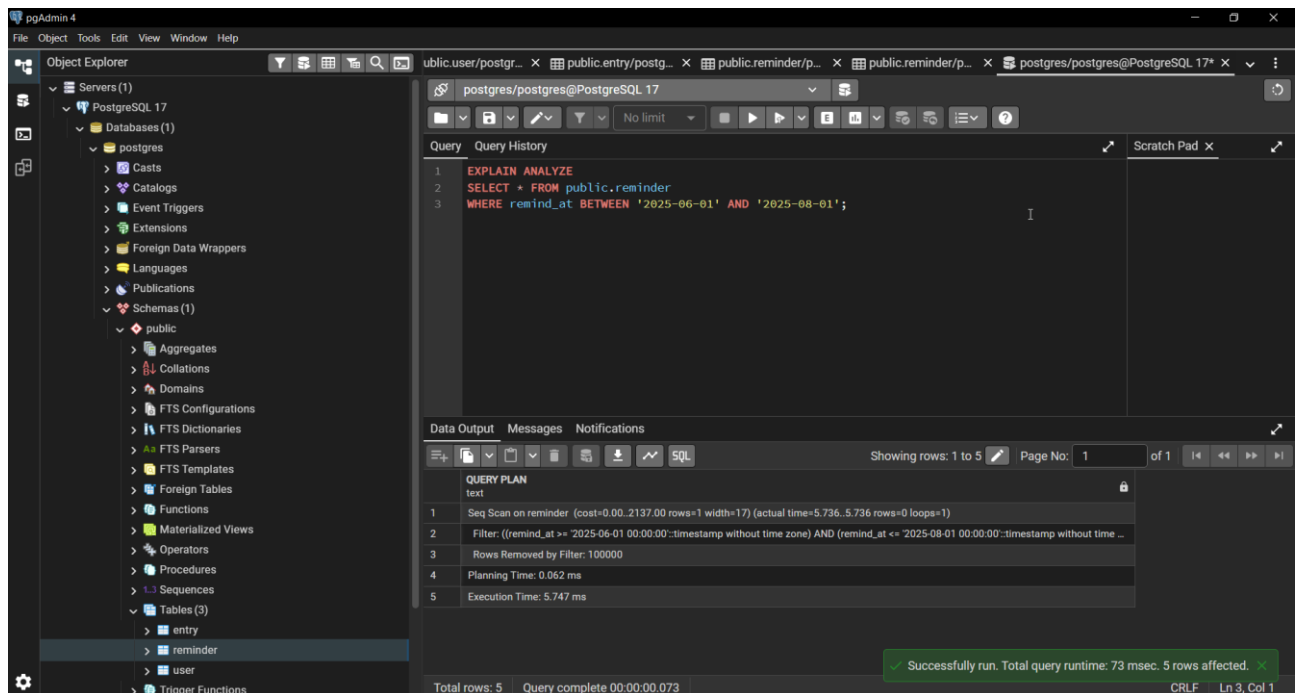


До індексації: Планувальник використовував Seq Scan (послідовне сканування всієї таблиці). Час виконання був значним, оскільки БД перебирала всі 100,000 рядків.

Після створення B-Tree: Планувальник перейшов на Index Scan (сканування індексу). Час виконання зменшився на кілька порядків (майже миттєво).

Пояснення:

Індекс **B-Tree** (збалансоване дерево) зберігає посилання на всі рядки у вигляді деревоподібної структури. Це дозволяє знаходити конкретне значення за логарифмічний час $O(\log n)$, не переглядаючи всю таблицю. Це універсальний тип індексу, який ефективний навіть для хаотично розкиданих даних.



Результати:

До індексації: Використовувався Seq Scan.

Після створення BRIN: Значного приросту продуктивності **не зафіксовано** (або планувальник продовжив використовувати Seq Scan, не дав виграшу у часі).

Аналіз причини (Чому BRIN не спрацював):

Індекс **BRIN (Block Range Index)** працює шляхом збереження лише мінімального та максимального значення для блоку сторінок на диску. Він ефективний лише тоді, коли дані в таблиці фізично впорядковані (корелюють з їх розташуванням у файлі).

Оскільки тестові дані генерувалися скриптом за допомогою функції генерування «рандомізованих» даних, то і дати записувалися в таблицю хаотично. У результаті, в кожному фізичному блоці діапазон значень (Min-Max) перекривав майже весь можливий період часу. Це зробило індекс BRIN неефективним, оскільки він не зміг відсіяти блоки даних, і базі довелося читати майже всю таблицю.

Висновок по BRIN:

Експеримент довів, що використання індексу BRIN є **недоцільним** для даних, які не мають фізичного впорядкування (кластеризації). Для ефективної роботи BRIN необхідна попередня обробка таблиці командою CLUSTER, що фізично відсортує рядки на диску.

3. Загальний висновок

Порівняння двох типів індексів показало:

B-Tree: Є найкращим вибором для загальних задач. Він займає більше місця на диску, але гарантує високу швидкість пошуку ($=$, $>$, $<$) незалежно від фізичного порядку даних.

BRIN: Є спеціалізованим індексом для надвеликих таблиць ("Big Data", логи). Він займає мізерно мало місця, але **вимагає, щоб дані були фізично впорядковані**. У випадку

випадкового розподілу даних (як у цій лабораторній роботі) його використання не дає переваг.

Завдання 3:

Створимо таблицю, куди тригер записуватиме інформацію про події.

| Query | Query History |
|-------|---|
| 1 | CREATE TABLE IF NOT EXISTS public.audit_log (|
| 2 | log_id SERIAL PRIMARY KEY, |
| 3 | user_id INT, |
| 4 | action_type VARCHAR(20), |
| 5 | description TEXT, |
| 6 | created_at TIMESTAMP DEFAULT NOW() |
| 7 |); |

Розробимо текст тригерної функції, який містить усі вимоги: IF (умови), CURSOR (цикл по записах), EXCEPTION (перехоплення помилок).

```
CREATE OR REPLACE FUNCTION trg_func_user_audit()
```

```
RETURNS TRIGGER AS $$
```

```
DECLARE
```

```
-- Оголошення змінних
```

```
entry_rec RECORD; -- Змінна для запису з курсора
```

```
titles_list TEXT := ''; -- Рядок для накопичення назв записів
```

```
-- Оголошення курсора
```

```
cur_entries CURSOR FOR
```

```
SELECT title FROM public.entry WHERE user_id = OLD.id;
```

```
BEGIN
```

```
BEGIN
```

```
IF OLD.id = 1 THEN
```

```
RAISE EXCEPTION 'Заборонено видаляти або змінювати SuperAdmin (ID=1)!';
```

END IF;

IF (TG_OP = 'DELETE') THEN

OPEN cur_entries;

LOOP

FETCH cur_entries INTO entry_rec;

EXIT WHEN NOT FOUND;

-- Накопичення тексту

titles_list := titles_list || entry_rec.title || ' ';

END LOOP;

CLOSE cur_entries;

-- Запис в лог

INSERT INTO public.audit_log (user_id, action_type, description)

VALUES (OLD.id, 'DELETE', 'Користувача видалено. Його записи: ' || titles_list);

RETURN OLD;

ELSIF (TG_OP = 'UPDATE') THEN

INSERT INTO public.audit_log (user_id, action_type, description)

VALUES (OLD.id, 'UPDATE', 'Зміна даних: ' || OLD.username || ' -> ' || NEW.username);

RETURN NEW;

END IF;

EXCEPTION

WHEN OTHERS THEN

```
RAISE NOTICE 'УВАГА: Транзакцію скасовано тригером. Причина: %', SQLERRM;
```

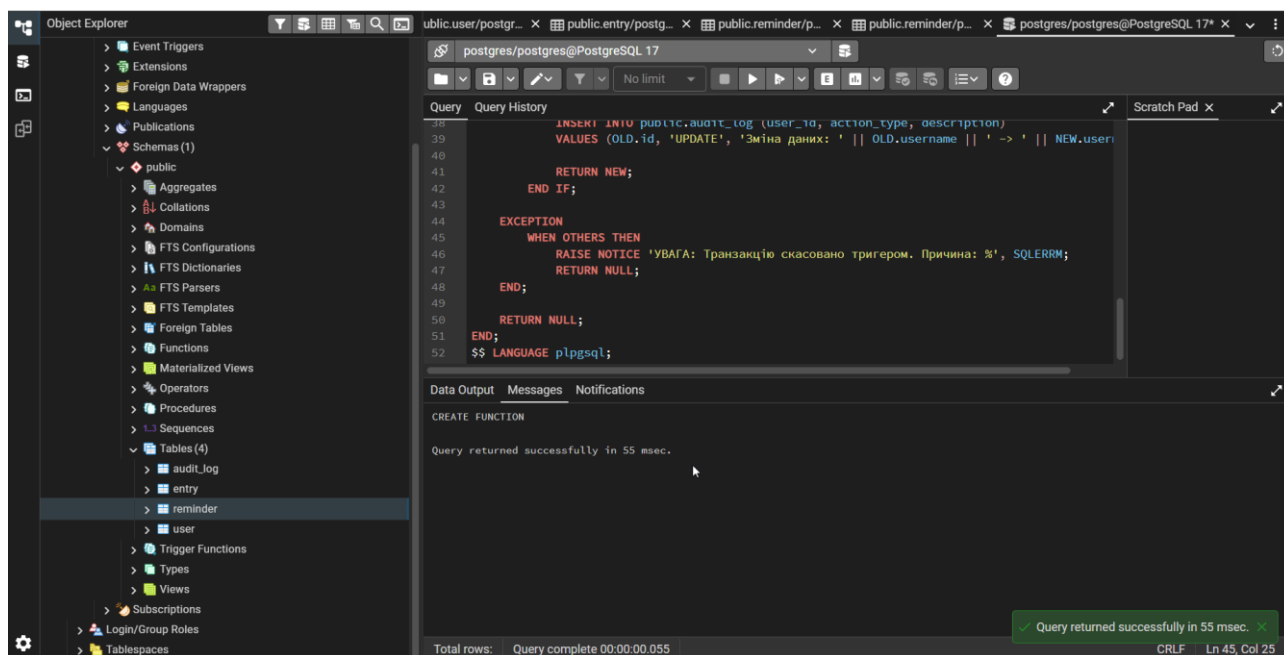
```
RETURN NULL;
```

```
END;
```

```
RETURN NULL;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

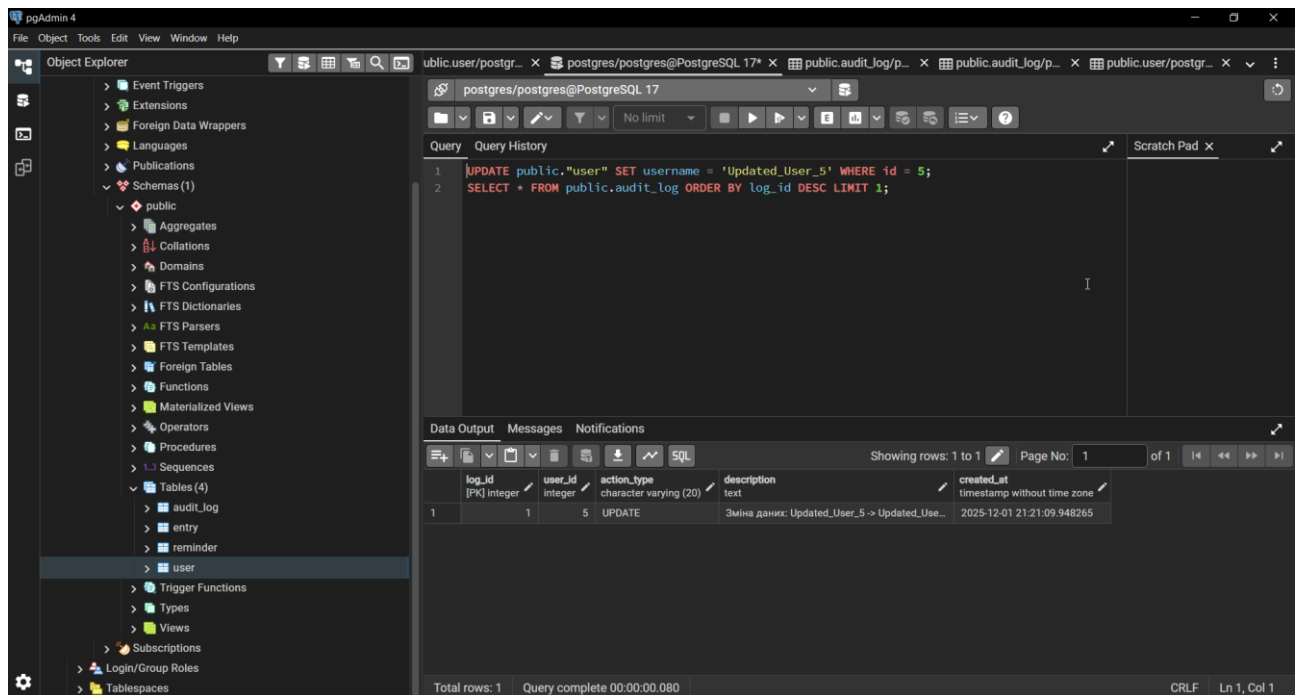


Створюємо тригер, та прив'язуємо функцію до таблиці user

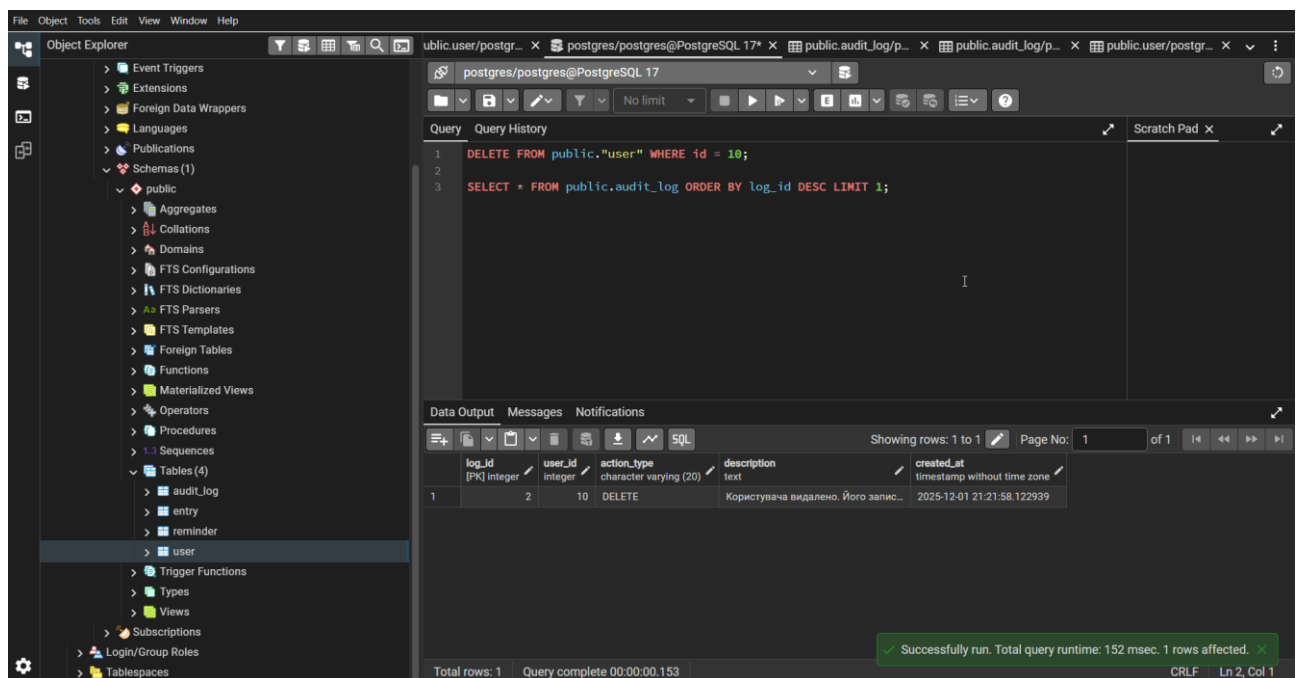
```
1 CREATE TRIGGER trg_before_user_changes
2 BEFORE DELETE OR UPDATE ON public."user"
3 FOR EACH ROW
4 EXECUTE FUNCTION trg_func_user_audit();
```

Демонстрація роботи

Приклад 1: Оновлення звичайного користувача (Успішне)

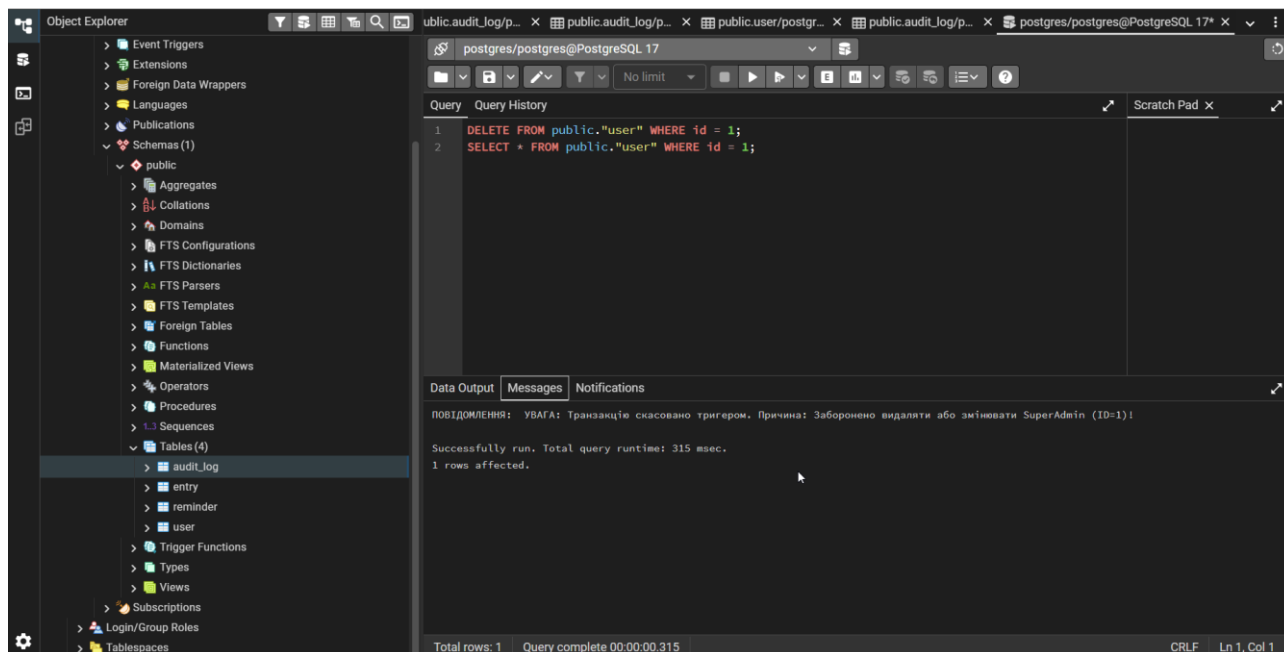


Приклад 2: Видалення звичайного користувача (Курсорний цикл)



Приклад 3: Спроба видалити SuperAdmin (Обробка Exception)

Тригер має заборонити дію і вивести повідомлення, але не впасти з помилкою (червоним), тому що ми зробили EXCEPTION WHEN OTHERS.



Отже, розробили та імплементували тригер бази даних `trg_before_user_changes` для таблиці `User`, який спрацьовує **перед** операціями видалення (`DELETE`) або оновлення (`UPDATE`).

Принцип роботи тригера:

Автоматичний аудит змін (`UPDATE`):

При спробі змінити дані користувача, тригер перехоплює подію до моменту фактичного запису в базу. Він порівнює старе значення (`OLD.username`) та нове (`NEW.username`), формує текстовий опис змін і зберігає цей запис у спеціальній таблиці `audit_log`. Це дозволяє мати повну історію змін облікових записів.

Збір залежних даних перед видаленням (`DELETE`):

Оскільки використовується подія `BEFORE DELETE`, запис користувача ще існує в базі в момент запуску тригера. Це дозволило використати **курсорний цикл (`CURSOR`)**, щоб пройти по пов'язаній таблиці `Entry` і зібрати заголовки всіх записів, що належать цьому користувачу. Цей список додається в лог як архівна інформація ("Що саме було втрачено разом з користувачем").

Захист критичних даних та обробка виключень:

Реалізовано логіку захисту облікового запису адміністратора (з `ID=1`).

При спробі змінити або видалити цей запис, тригер генерує виключну ситуацію (`RAISE EXCEPTION`).

Завдяки блоку **`EXCEPTION WHEN OTHERS`**, помилка не призводить до аварійної зупинки всієї системи ("крешу"). Натомість тригер перехоплює помилку, виводить попередження (`NOTICE`) і повертає `NULL`.

Повернення `NULL` у тригері типу `BEFORE` призводить до **скасування конкретної операції**, залишаючи дані недоторканими.

Завдання 4:

READ COMMITTED (Брудне/Неповторюване читання)

Це рівень за замовчуванням. Одна транзакція бачить зміни іншої одразу після коміту.

УВАГА: Кодова сторінка консолі (866) відрізняється від кодової сторінки Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі

"Нотатки для користувачів Windows" у документації psql.

Введіть "help", щоб отримати допомогу.

```
postgres=# BEGIN;
BEGIN
postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Serial_User_B
(1 ř фюъ)

postgres=#
```

Server [localhost]:

Database [postgres]:

Port [5432]:

Username [postgres]:

Пароль користувача postgres:

psql (17.6)

УВАГА: Кодова сторінка консолі (866) відрізняється від кодової сторінки Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі

"Нотатки для користувачів Windows" у документації psql.

Введіть "help", щоб отримати допомогу.

```
postgres=# BEGIN;
BEGIN
postgres=# UPDATE "user" SET username = 'Name_Change_ByA' WHERE id = 5;
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=#
```

```

Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (17.6)
УВАГА: Кодова сторінка консол? (866) в?др?зняється в?д кодової стор?нки Windows (1251)
      8-6?тов? символи можуть працювати неправильно. Детальн?ше у розд?л?
      "Нотатки для користувач?в Windows" у документац?ї psql.
Введ?ть "help", щоб отримати допомогу.

postgres=# BEGIN;
BEGIN
postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Serial_User_B
(1 ? ?ю?)

postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Name_Change_ByA
(1 ? ?ю?)

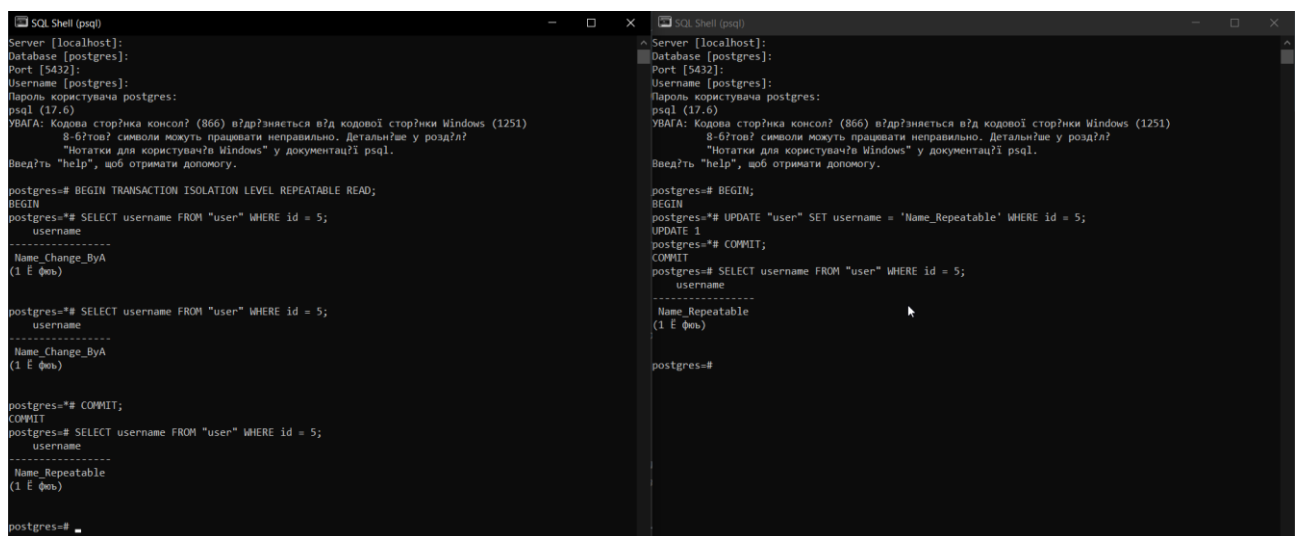
postgres=# COMMIT;
COMMIT
postgres=#

```

Продemonстровано феномен неповторюваного читання. У першій сесії (Transaction A) було виконано вибірку даних. Паралельна сесія (Transaction B) змінила ці дані та виконала COMMIT. Повторна вибірка у сесії A показала нові дані, що порушує цілісність "погляду" на дані в межах однієї транзакції.

REPEATABLE READ (Стабільний знімок)

Транзакція "заморожує" стан даних на початку. Зміни сусідів ігноруються.



```

SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (17.6)
УВАГА: Кодова сторінка консол? (866) в?др?зняється в?д кодової стор?нки Windows (1251)
      8-6?тов? символи можуть працювати неправильно. Детальн?ше у розд?л?
      "Нотатки для користувач?в Windows" у документац?ї psql.
Введ?ть "help", щоб отримати допомогу.

postgres=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Name_Change_ByA
(1 ? ?ю?)

postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Name_Change_ByA
(1 ? ?ю?)

postgres=# COMMIT;
COMMIT
postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Name_Repeatable
(1 ? ?ю?)

postgres=#

```

```

SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (17.6)
УВАГА: Кодова сторінка консол? (866) в?др?зняється в?д кодової стор?нки Windows (1251)
      8-6?тов? символи можуть працювати неправильно. Детальн?ше у розд?л?
      "Нотатки для користувач?в Windows" у документац?ї psql.
Введ?ть "help", щоб отримати допомогу.

postgres=# BEGIN;
BEGIN
postgres=# UPDATE "user" SET username = 'Name_Repeatable' WHERE id = 5;
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=# SELECT username FROM "user" WHERE id = 5;
      username
-----
 Name_Repeatable
(1 ? ?ю?)

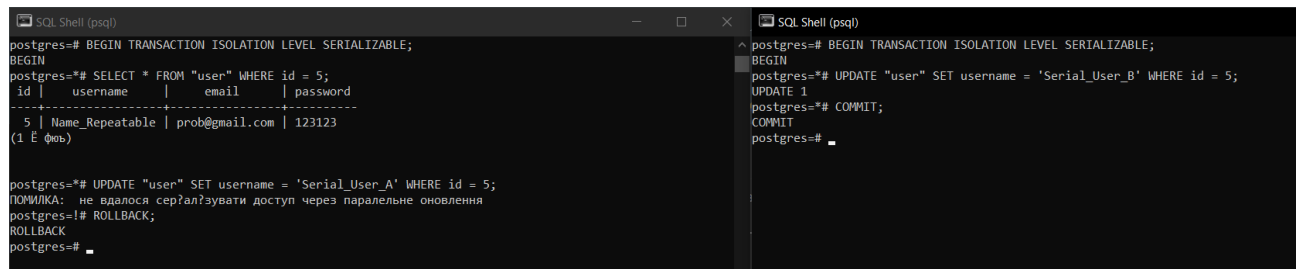
postgres=#

```

Встановлення цього рівня ізоляції дозволило уникнути феномену неповторюваного читання. Транзакція А створила "знімок" бази даних на момент початку. Оновлення даних, зроблене паралельною транзакцією Б, не вплинуло на результати вибірки в транзакції А до її завершення.

SERIALIZABLE (Конфлікт)

Найсуворіший рівень. Якщо є ризик конфлікту, транзакція "вбивається".



```
SQL Shell (psql)
postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
postgres=# SELECT * FROM "user" WHERE id = 5;
 id | username | email | password
-----+-----+-----+-----
  5 | Name_Repeatable | prob@gmail.com | 123123
(1 row)

postgres=# UPDATE "user" SET username = 'Serial_User_A' WHERE id = 5;
ПОВИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# ROLLBACK;
ROLLBACK
postgres=#
```

```
SQL Shell (psql)
postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
postgres=# UPDATE "user" SET username = 'Serial_User_B' WHERE id = 5;
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=#
```

Продемонстровано механізм запобігання аномаліям серіалізації. При спробі двох транзакцій одночасно змінити один і той самий рядок на рівні SERIALIZABLE, СУБД виявила конфлікт залежностей. Транзакція, яка спробувала виконати UPDATE другою, була автоматично перервана з помилкою `could not serialize access due to concurrent update`. Це гарантує повну узгодженість даних.