

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: КВ-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 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.

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.

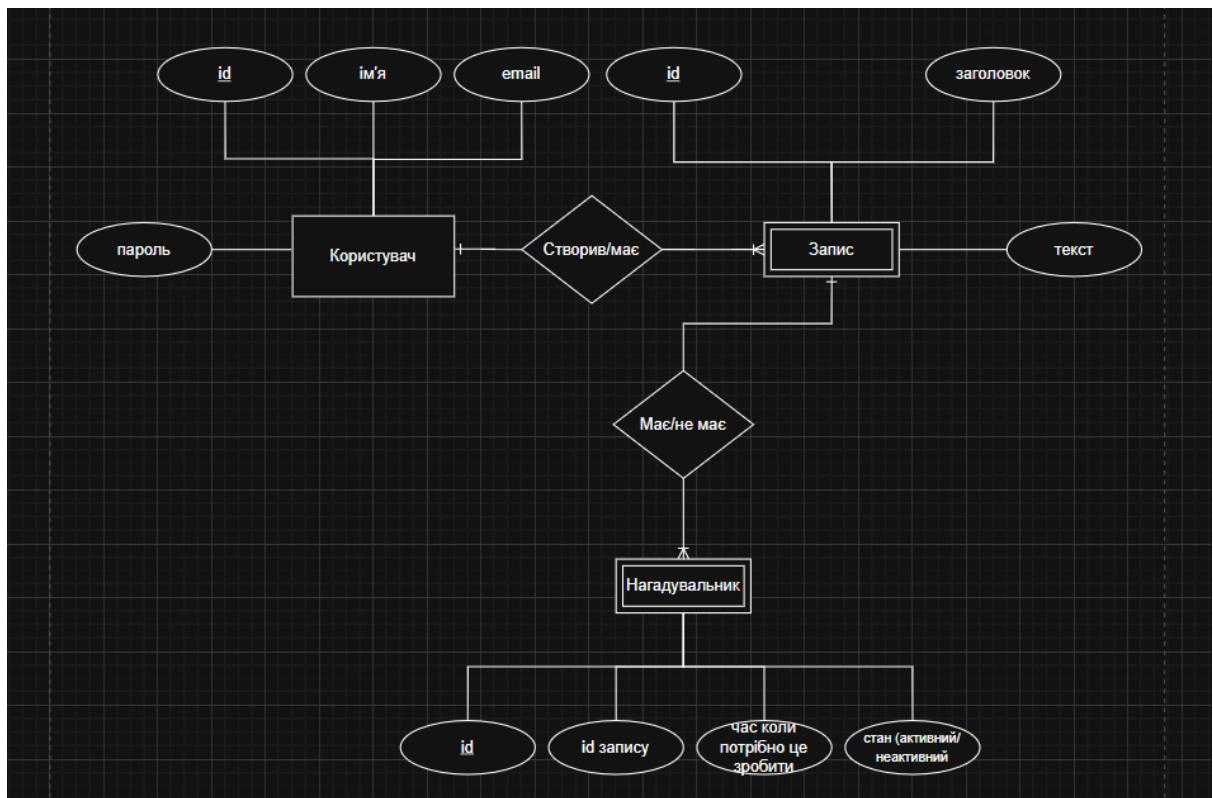
Зв'язки:

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

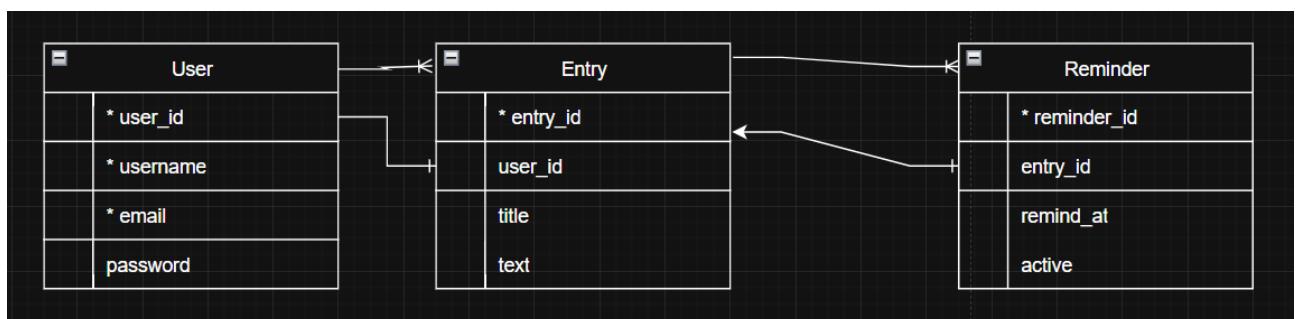
Один користувач може створити безліч записів, але кожен конкретний запис належить лише одному користувачу.

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 сущностей: **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 # --- ОГОЛОШЕННЯ 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
22
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")
```

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}"
170
```

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

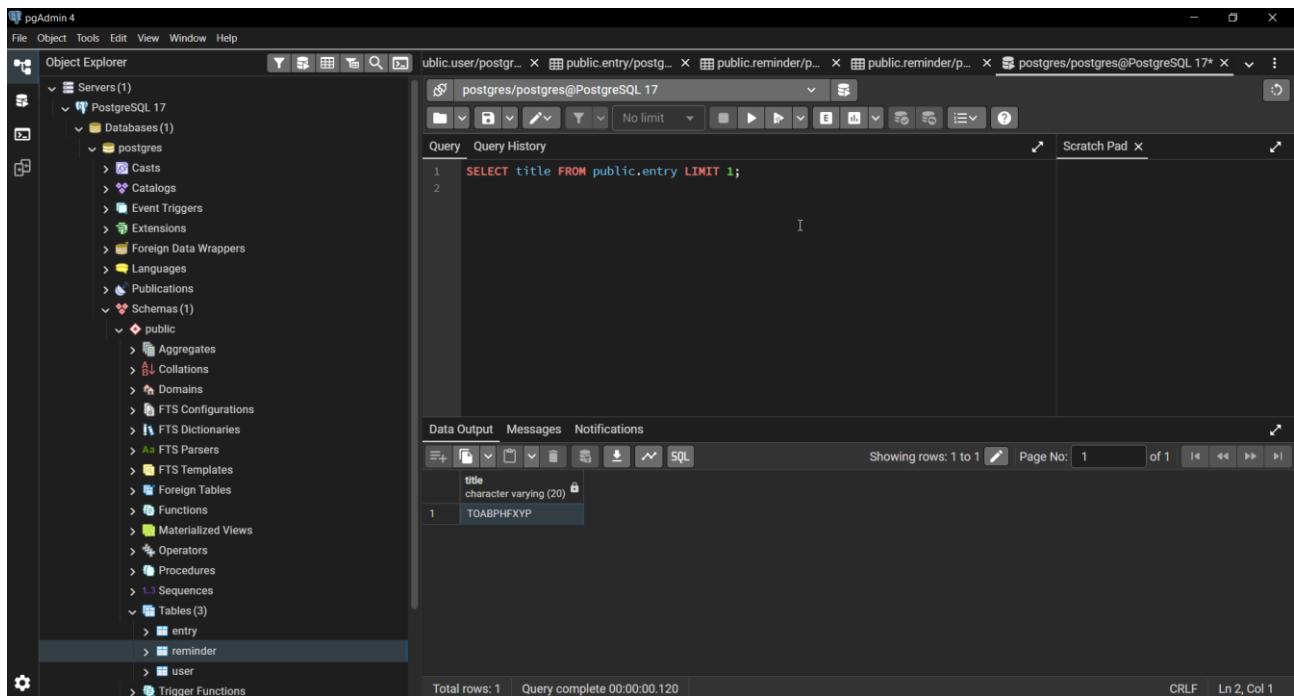
```

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



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

```

EXPLAIN ANALYZE
SELECT * FROM public.entry
WHERE title = 'TOABPHIFXYP';

```

Query Plan:

- Seq Scan on entry (cost=0.00..1986.00 rows=1 width=30) (actual time=0.017..7.625 rows=1 loops=1)
 - Filter: ((title::text = 'TOABPHIFXYP'::text))
 - Rows Removed by Filter: 99999
- Planning Time: 0.083 ms
- Execution Time: 7.641 ms

Total rows: 5 Query complete 00:00:00.580 Successfully run. Total query runtime: 580 msec. 5 rows affected.

Проведемо тестування пошуку за заданим діапазоном дат для Нагадувальника

```

EXPLAIN ANALYZE
SELECT * FROM public.reminder
WHERE remind_at BETWEEN '2025-06-01' AND '2025-08-01';

```

Query Plan:

- Seq Scan on reminder (cost=0.00..2137.00 rows=1 width=17) (actual time=5.825..5.826 rows=0 loops=1)
 - Filter: ((remind_at >= '2025-06-01 00:00:00'::timestamp without time zone) AND (remind_at <= '2025-08-01 00:00:00'::timestamp without time zone))
 - Rows Removed by Filter: 100000
- Planning Time: 0.817 ms
- Execution Time: 5.837 ms

Total rows: 5 Query complete 00:00:00.140 Successfully run. Total query runtime: 140 msec. 5 rows affected.

-- 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);`

```

-- Створюємо BTree для прискорення пошуку по Title
CREATE INDEX idx_entry_title_btree ON public.entry USING btree (title);

-- Створюємо BRIN для дат (дуже компактний індекс)
CREATE INDEX idx_reminder_date_brin ON public.reminder USING brin (remind_at);

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables' section of the 'entry' table, the 'idx_entry_title_btree' index is selected. In the main query editor, two SQL statements are shown for creating indexes. Below the query editor, the 'Data Output' tab displays the execution plan for the second statement, which shows a sequential scan on the 'reminder' table.

2. Тест зі створеними індексами BTree та BRIN

```

EXPLAIN ANALYZE
SELECT * FROM public.entry
WHERE title = 'TOABPHFXYP';

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, the 'entry' table is selected. In the main query editor, an EXPLAIN ANALYZE query is run to analyze the execution plan for a WHERE clause. The results show that the query uses the 'idx_entry_title_btree' index for the search. A green message at the bottom right indicates the query was successfully run and completed quickly.

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 блокі 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 (клasterизації). 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;
```

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;
```

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer tree is visible, showing the schema structure. In the center, a query editor window displays the SQL code for creating a trigger function. The code includes an INSERT INTO statement, a RETURN NEW; statement, an END IF; block, an EXCEPTION block that RAISES NOTICE, and a final RETURN NULL;. The code concludes with an END; and a \$\$ LANGUAGE plpgsql; declaration. Below the query editor, the Data Output tab shows a successful execution message: "Query returned successfully in 55 msec." At the bottom right, a status bar indicates "Query returned successfully in 55 msec. CRLF Ln 45, Col 25".

```
38    INSERT INTO public.audit_log (user_id, action_type, description)
39      VALUES (OLD.id, 'UPDATE', 'Зміна даних: ' || OLD.username || ' -> ' || NEW.username);
40
41      RETURN NEW;
42  END IF;
43
44  EXCEPTION
45    WHEN OTHERS THEN
46      RAISE NOTICE 'УВАГА: Транзакцію скасовано тригером. Причина: %', SQLERRM;
47      RETURN NULL;
48  END;
49
50  RETURN NULL;
51 END;
52 $$ 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)

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, the 'user' table under the 'public' schema is selected. In the main pane, a query is run:

```

1 UPDATE public."user" SET username = 'Updated_User_5' WHERE id = 5;
2 SELECT * FROM public.audit_log ORDER BY log_id DESC LIMIT 1;

```

The Data Output tab shows the results of the audit log query:

log_id	user_id	action_type	description	created_at
1	1	UPDATE	Зміна даних: Updated_User_5 > Updated_User_5	2025-12-01 21:21:09.948265

Total rows: 1 Query complete 00:00:00.080 CRLF Ln 1, Col 1

Приклад 2: Видалення звичайного користувача (Курсорний цикл)

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, the 'user' table under the 'public' schema is selected. In the main pane, a query is run:

```

1 DELETE FROM public."user" WHERE id = 10;
2
3 SELECT * FROM public.audit_log ORDER BY log_id DESC LIMIT 1;

```

The Data Output tab shows the results of the audit log query:

log_id	user_id	action_type	description	created_at
1	2	DELETE	Користувача видалено. Його запис...	2025-12-01 21:21:58.122939

Successfully run. Total query runtime: 152 msec. 1 rows affected. Total rows: 1 Query complete 00:00:00.153 CRLF Ln 2, Col 1

Приклад 3: Спроба видалити SuperAdmin (Обробка Exception)

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

```

public.audit_log/p... public.audit_log/p... public.user/postgr... public.audit_log/p... postgres/postgres@PostgreSQL 17* 
Object Explorer  postgres/postgres@PostgreSQL 17*  : 
Query History  Scratch Pad 
Query 1
2
DELETE FROM public."user" WHERE id = 1;
SELECT * FROM public."user" WHERE id = 1;

Data Output Messages Notifications
ПОВІДОМЛЕННЯ: УВАГА: Транзакція скасувана тригером. Причина: Заборонено видаляти або змінювати SuperAdmin (ID=1)

Successfully run. Total query runtime: 315 msec.
1 rows affected.

Total rows: 1 Query complete 00:00:00.315 CRLF Ln 1, Col 1

```

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 переходить 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 пройтися 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 переходить 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 Є фюь)

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 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 рядок)

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

postgres=# COMMIT;
COMMIT
postgres=#

```

Продемонстровано 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.

The screenshot shows two side-by-side PostgreSQL SQL shells. Both windows have identical configuration parameters at the top:

- 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.

The left window (Session A) contains the following commands and output:

```

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

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

postgres=# COMMIT;
COMMIT
postgres=#

```

The right window (Session B) contains the following commands and output:

```

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 рядок)

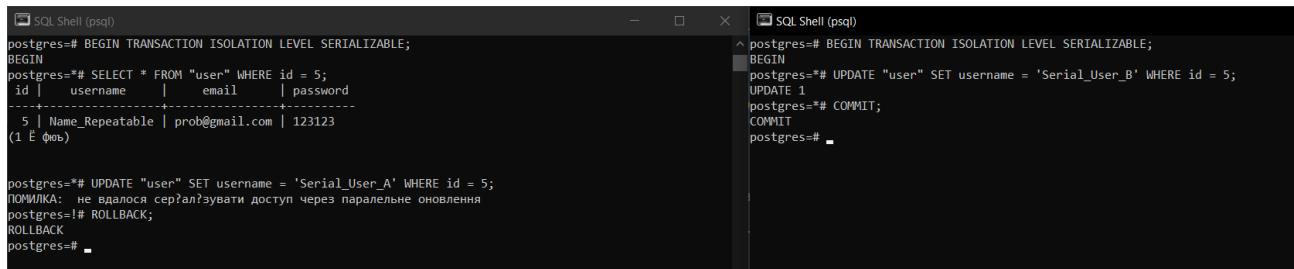
postgres=#

```

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

SERIALIZABLE (Конфлікт)

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



The screenshot shows two separate PostgreSQL sessions in SQL Shell (psql) windows.

Session 1 (Left):

```
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;
WARNING:  could not serialize access due to concurrent update
ROLLBACK;
postgres=#

```

Session 2 (Right):

```
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.