НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №2

з дисципліни
«Бази даних і засоби управління»

Група: КВ-23

Виконав: Булавчук Д.

Оцінка:

Київ – 2024

Засоби оптимізації роботи СУБД PostgreSQL

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

Завдання роботи полягає у наступному:

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

Вимоги до пункту завдання №1

Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlAlchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:M, M:M та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

Вимоги до пункту завдання №2

Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного

представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5 прикладів запитів SELECT (із виведенням результуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).

Вимоги до пункту завдання №3

Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

Вимоги до пункту завдання №4

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і спосіб їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у різних вікнах pgAdmin4 і виконати послідовність запитів INSERT, UPDATE або DELETE у обох транзакціях, що доводять наявність або відсутність певних феноменів.

№ варіанта	*Види індексів*	*Умови для тригера*
4	*GIN, BRIN*	*after delete, insert*

1. **Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проекції (ORM).**

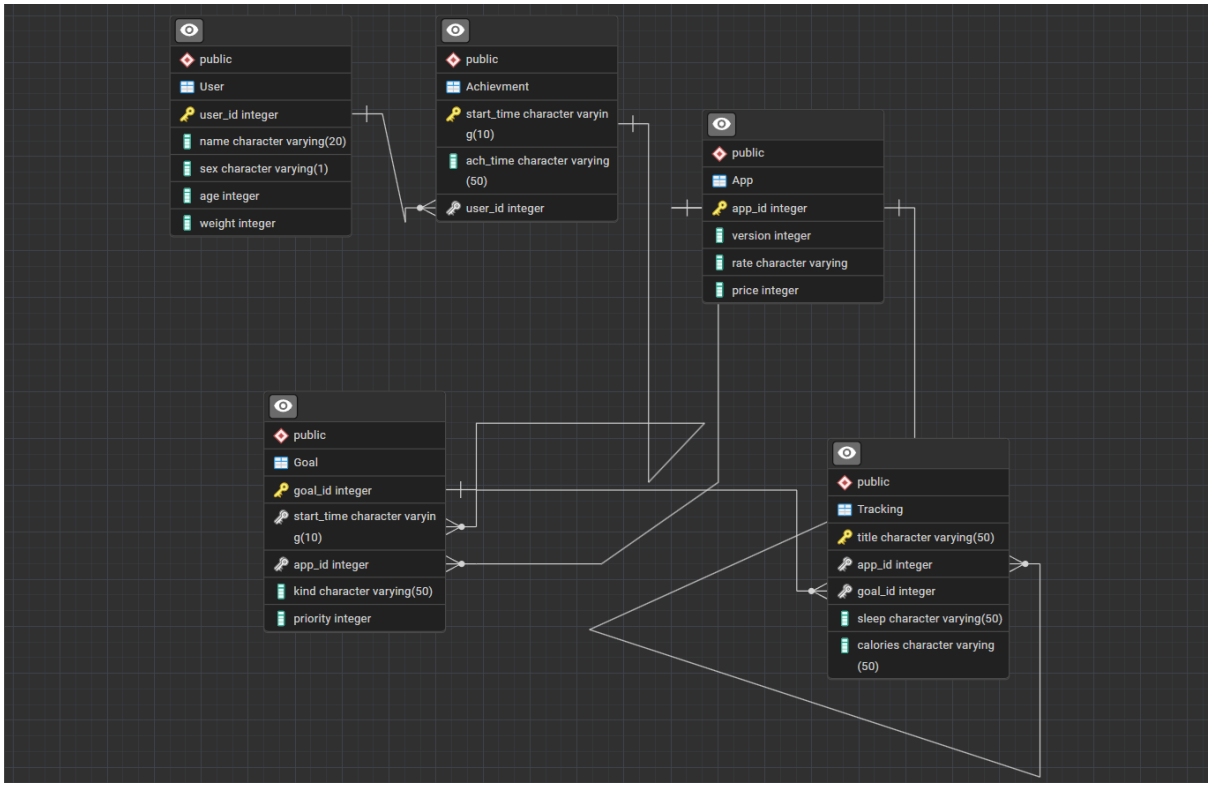

Рис. 1 - Схема бази даних

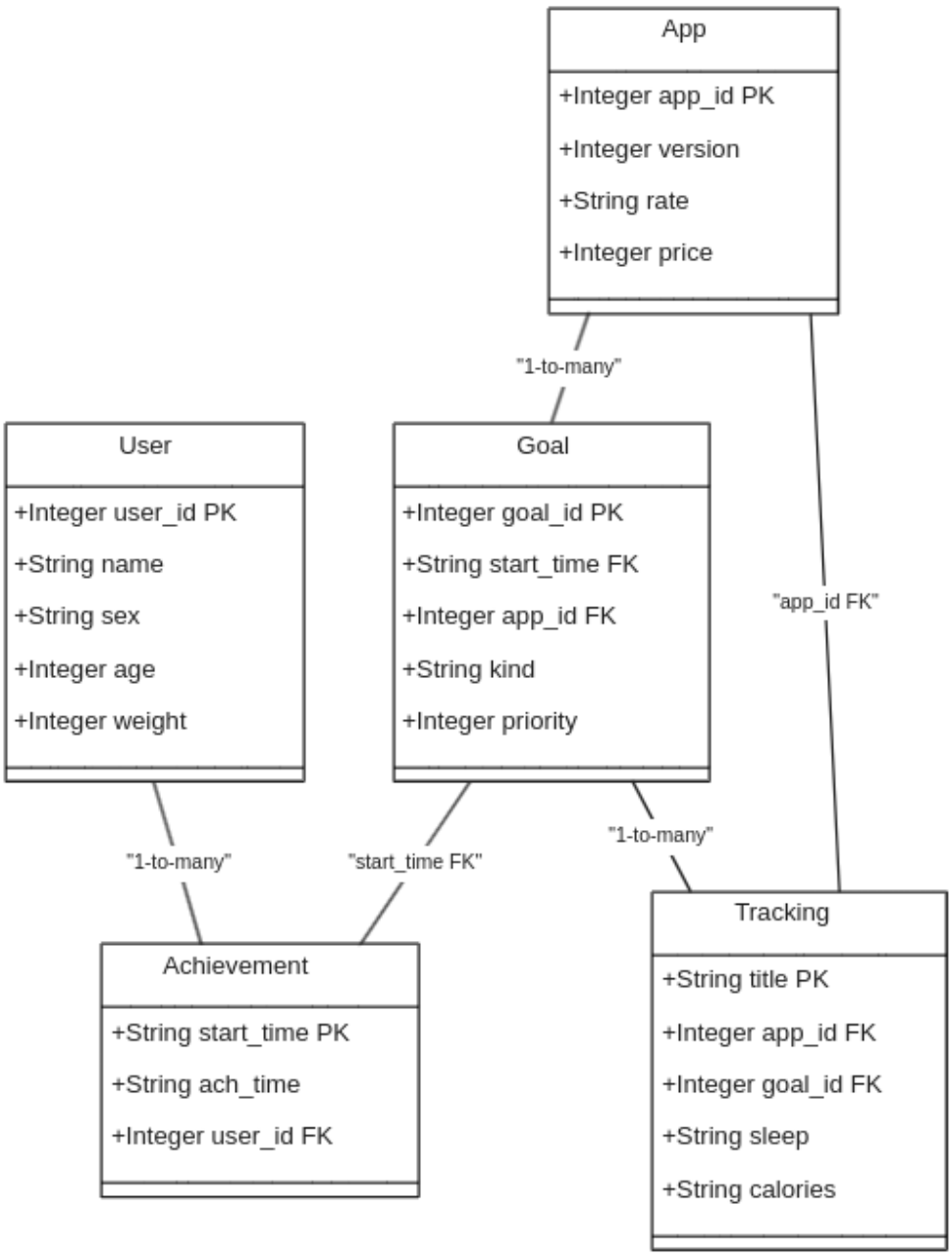

Рис. 2 - Класи ORM

Приклади запитів у вигляді ORM

```
def add_user(self, name, sex, age, weight):
    session = self.Session()
    user = User(name=name, sex=sex, age=age, weight=weight)
    session.add(user)
    session.commit()
    session.close()

  def get_users(self):
    session = self.Session()
    users = session.query(User).all()
    session.close()
    return [(u.user_id, u.name, u.sex, u.age, u.weight) for u in users]

  def update_user(self, user_id, name, sex, age, weight):
    session = self.Session()
    user = session.query(User).filter(User.user_id == user_id).first()
    if user:
      user.name = name
      user.sex = sex
      user.age = age
      user.weight = weight
      session.commit()
    session.close()

  def delete_user(self, user_id):
    session = self.Session()
    user = session.query(User).filter(User.user_id == user_id).first()
    if user:
      session.delete(user)
```

```
    session.commit()
session.close()
```

2. Створити та проаналізувати різні типи індексів у PostgreSQL.

GIN Index

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT *
3  FROM public."Goal"
4  WHERE kind LIKE '%fitness%'
5  ORDER BY priority;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=1.03..1.04 rows=1 width=168) (actual time=0.015..0.016 rows=0 loops=1)
2	Sort Key: priority
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on "Goal" (cost=0.00..1.02 rows=1 width=168) (actual time=0.009..0.009 rows=0 loop…
5	Filter: ((kind)::text ~~ '%fitness%'::text)
6	Rows Removed by Filter: 22
7	Planning Time: 0.112 ms
8	Execution Time: 0.024 ms

Query | Query History

```
1  CREATE INDEX idx_goal_kind_gin
2  ON public."Goal"
3  USING GIN (kind gin_trgm_ops);
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 56 msec.
```

Query Query History

```
1  EXPLAIN ANALYZE
2  SELECT *
3  FROM public."Goal"
4  WHERE kind LIKE '%fitness%'
5  ORDER BY priority;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Sort (cost=1.28..1.29 rows=1 width=168) (actual time=0.013..0.014 rows=0 loops=1)
2	Sort Key: priority
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on "Goal" (cost=0.00..1.27 rows=1 width=168) (actual time=0.010..0.010 rows=0 loop…
5	Filter: ((kind)::text ~~ '%fitness%'::text)
6	Rows Removed by Filter: 22
7	Planning Time: 0.158 ms
8	Execution Time: 0.025 ms

BRIN Index

Query Query History

```
1  CREATE INDEX idx_goal_priority_brin
2  ON public."Goal"
3  USING BRIN (priority);
```

Data Output Messages Notifications

```
CREATE INDEX

Query returned successfully in 51 msec.
```

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT *
3  FROM public."Goal"
4  WHERE kind LIKE '%fitness%'
5  ORDER BY priority;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=1.28..1.29 rows=1 width=168) (actual time=0.011..0.011 rows=0 loops=1)
2	Sort Key: priority
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on "Goal" (cost=0.00..1.27 rows=1 width=168) (actual time=0.008..0.008 rows=0 loop…
5	Filter: ((kind)::text ~~ '%fitness%'::text)
6	Rows Removed by Filter: 22
7	Planning Time: 0.127 ms
8	Execution Time: 0.020 ms

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT *
3  FROM public."Goal"
4  WHERE priority > 5
5  ORDER BY priority;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=1.37..1.39 rows=7 width=168) (actual time=0.012..0.013 rows=2 loops=1)
2	Sort Key: priority
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on "Goal" (cost=0.00..1.27 rows=7 width=168) (actual time=0.007..0.008 rows=2 loop…
5	Filter: (priority > 5)
6	Rows Removed by Filter: 20
7	Planning Time: 0.068 ms
8	Execution Time: 0.022 ms

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT app_id, COUNT(*) AS goal_count
3  FROM public."Goal"
4  GROUP BY app_id
5  HAVING COUNT(*) > 10;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=1.33..1.60 rows=7 width=12) (actual time=0.028..0.029 rows=0 loops=1)
2	Group Key: app_id
3	Filter: (count(*) > 10)
4	Batches: 1 Memory Usage: 24kB
5	Rows Removed by Filter: 21
6	-> Seq Scan on "Goal" (cost=0.00..1.22 rows=22 width=4) (actual time=0.007..0.009 rows=22 loop...
7	Planning Time: 0.093 ms
8	Execution Time: 0.055 ms

1. Пошук тексту з сортуванням

До створення індексу GIN запит використовував послідовний перегляд таблиці (Seq Scan). Умови фільтрації виконувалися на кожному рядку, що уповільнює роботу на великих обсягах даних. Після створення індексу GIN суттєвого покращення не відбулося, оскільки оператор LIKE '%text%' обмежує можливості використання цього типу індексу. Для таких умов GIN не є оптимальним, адже його основне призначення – повнотекстовий пошук або специфічні шаблони.

2. Фільтрація числових значень із сортуванням

До створення індексу BRIN запит також використовував Seq Scan, переглядаючи всі рядки таблиці. Після створення BRIN індексу час виконання зменшився, оскільки BRIN ефективно обмежив кількість блоків даних, які потрібно перевірити, завдяки діапазонному індексуванню. Це особливо корисно для великих таблиць із числовими або впорядкованими даними.

3. Агрегація та групування

Запити, які використовують агрегатні функції, наприклад підрахунок кількості записів у групах (COUNT(*)), виконувалися за допомогою хеш-агрегації. Тут індекси не змогли суттєво вплинути на продуктивність, оскільки агрегатні функції більше залежать від самого механізму обробки даних. Проте індекси можуть допомогти, якщо разом із групуванням використовується фільтрація.

Висновки

1. **GIN індекс** корисний лише для специфічних запитів, таких як повнотекстовий пошук або оператори типу @@. Для шаблонів LIKE '%text%' його ефективність обмежена.

2. **BRIN індекс** значно покращує продуктивність запитів, що працюють із впорядкованими або числовими даними, особливо у великих таблицях.
3. Для запитів із агрегатними функціями індекси мають обмежений вплив, але можуть бути корисними в поєднанні з умовами фільтрації.
4. Використання індексів має сенс лише тоді, коли вони відповідають характеру запитів. Неправильно підібрані індекси можуть не вплинути на продуктивність або навіть уповільнити оновлення даних у таблиці.

Результати тестів показали, що індекси BRIN підходять для більшості сценаріїв, пов’язаних із числовими фільтраціями, тоді як GIN варто використовувати в конкретних ситуаціях текстового пошуку.

3. Розробити тригер бази даних PostgreSQL.

```
-- Створення тригера для обробки вставки та видалення записів у таблиці public."Achievement"
CREATE OR REPLACE FUNCTION handle_achievement_changes() RETURNS TRIGGER
AS $$
DECLARE
  total_achievements INT;
BEGIN
  -- Дії після вставки
  IF TG_OP = 'INSERT' THEN
    -- Підрахунок загальної кількості досягнень для користувача
    SELECT COUNT(*) INTO total_achievements
    FROM public."Achievement"
    WHERE user_id = NEW.user_id;

    -- Якщо досягнень більше 5, оновлюємо rate у таблиці public."App"
    IF total_achievements > 5 THEN
      UPDATE public."App"
      SET rate = 'High'
      WHERE app_id IN (
        SELECT app_id
        FROM public."Goal"
        WHERE goal_id IN (
          SELECT goal_id FROM public."Goal" WHERE start_time = NEW.start_time
        )
      );
    END IF;
  END IF;

  -- Дії після видалення
```

```
  IF TG_OP = 'DELETE' THEN
    -- Перевірка, чи залишилися ще досягнення для користувача
    SELECT COUNT(*) INTO total_achievements
    FROM public."Achievement"
    WHERE user_id = OLD.user_id;

    -- Якщо досягнень більше немає, оновлюємо rate у таблиці public."App"
    IF total_achievements = 0 THEN
      UPDATE public."App"
      SET rate = 'Low'
      WHERE app_id IN (
        SELECT app_id
        FROM public."Goal"
        WHERE goal_id IN (
          SELECT goal_id FROM public."Goal" WHERE start_time = OLD.start_time
        )
      );
    END IF;
  END IF;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Створення тригера
CREATE TRIGGER achievement_trigger
AFTER INSERT OR DELETE
ON public."Achievement"
FOR EACH ROW
EXECUTE FUNCTION handle_achievement_changes();
```

4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

1. READ COMMITTED

Поведінка:

- За замовчуванням цей рівень ізоляції використовується в PostgreSQL.
- Дозволяє **неповторювані зчитування** і **фантомні зчитування**.

Кроки:

1. У **Транзакції 1**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM public."Achievement";
```

2. У **Транзакції 2**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO public."Achievement" (start_time, ach_time, user_id) VALUES
('12:00', '3 hours', 1);
COMMIT;
```

3. Повертаємося до **Транзакції 1**:

```
SELECT * FROM public."Achievement";
COMMIT;
```

Спостереження:

- Перший запит `SELECT` у Транзакції 1 **не бачить** нового рядка.
- Другий запит `SELECT` у Транзакції 1 **бачить** новий рядок, доданий у Транзакції 2.

Феномен:

- **Неповторюване зчитування**: Значення, яке було прочитане раніше, змінюється після коміту іншої транзакції.

2. REPEATABLE READ

Поведінка:

- Запобігає **неповторюваним зчитуванням**, але дозволяє **фантомні зчитування**.

Кроки:

1. У **Транзакції 1**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM public."Achievement";
```

2. У **Транзакції 2**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
INSERT INTO public."Achievement" (start_time, ach_time, user_id) VALUES
('13:00', '4 hours', 1);
COMMIT;
```

3. Повертаємося до **Транзакції 1**:

```
SELECT * FROM public."Achievement";
COMMIT;
```

Спостереження:

- Транзакція 1 **не бачить** рядок, доданий у Транзакції 2, навіть після її коміту.

Феномен:

- **Немає неповторюваних зчитувань**: Дані залишаються послідовними протягом транзакції.

3. SERIALIZABLE
Поведінка:

- Найвищий рівень ізоляції.
- Запобігає **фантомним зчитуванням**, забезпечуючи серіалізоване виконання транзакцій.

Кроки:

1. У **Транзакції 1**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM public."Achievement";
```

2. У **Транзакції 2**:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO public."Achievement" (start_time, ach_time, user_id) VALUES
('14:00', '5 hours', 2);
COMMIT;
```

3. Повертаємося до **Транзакції 1**:

```
SELECT * FROM public."Achievement";
COMMIT;
```

Спостереження:

- Транзакція 1 **не бачить** нового рядка, доданого Транзакцією 2.
- Якщо Транзакція 1 спробує додати або змінити дані, що перетинаються

Феномен:

- **Немає фантомних зчитувань**: Нові рядки недоступні для активних транзакцій.

github:
telegram: https://t.me/lightblue_shark