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»**

Тема: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав: студент III курсу

ФПМ групи КВ-13

Горбик Д.В.

Київ – 2023

*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.

**Варіант 4**

У другому завданні проаналізувати індекси *GIN, BRIN.*

Умова для тригера – *after delete, insert*

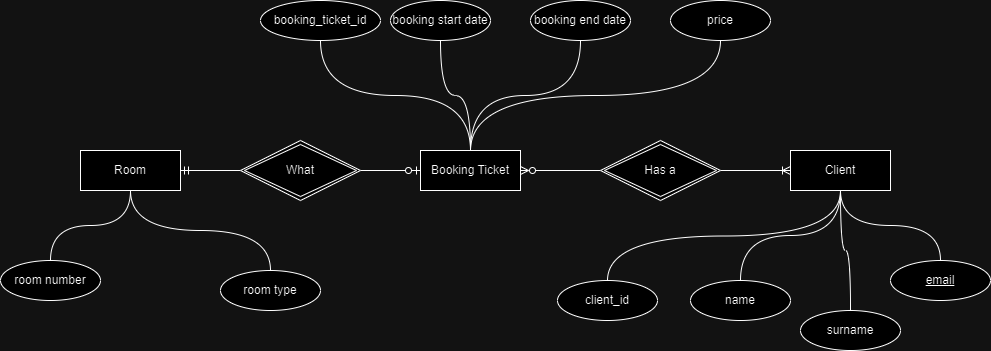
**Завдання 1**

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:М, М:М 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.

**Інформація про модель та структуру бази даних**

Рис. 1 - Концептуальна модель предметної області “Готельний бронювальний портал”

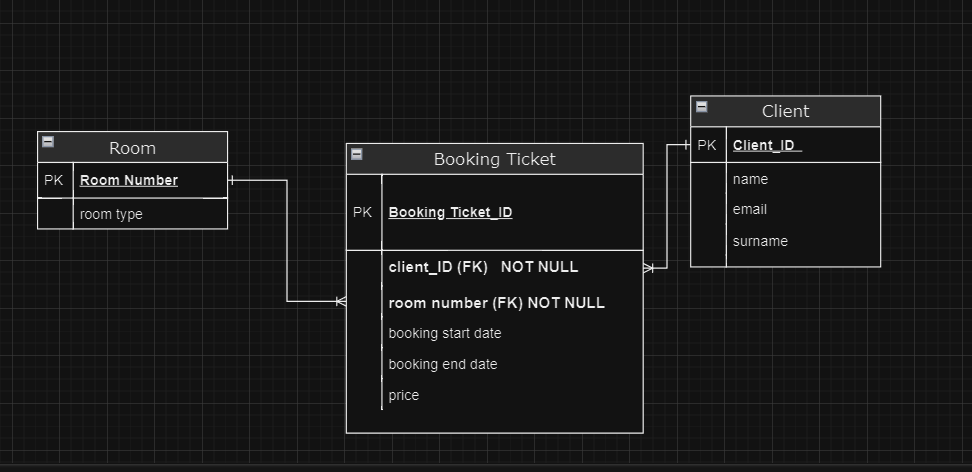
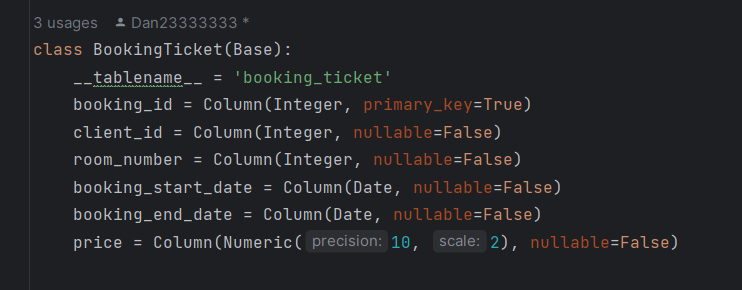
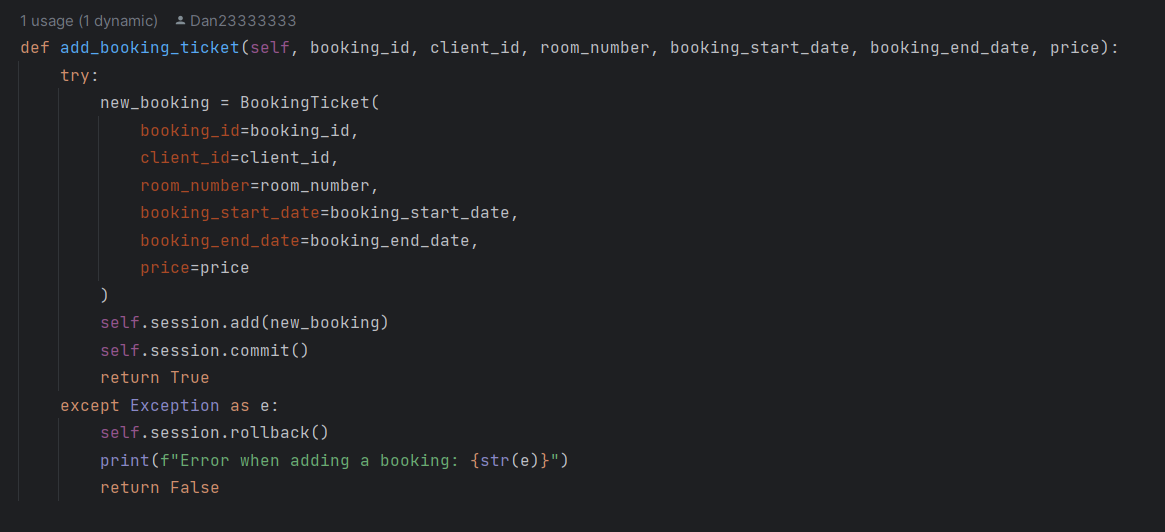
Нижче (Рис. 2) наведено логічну модель бази даних:

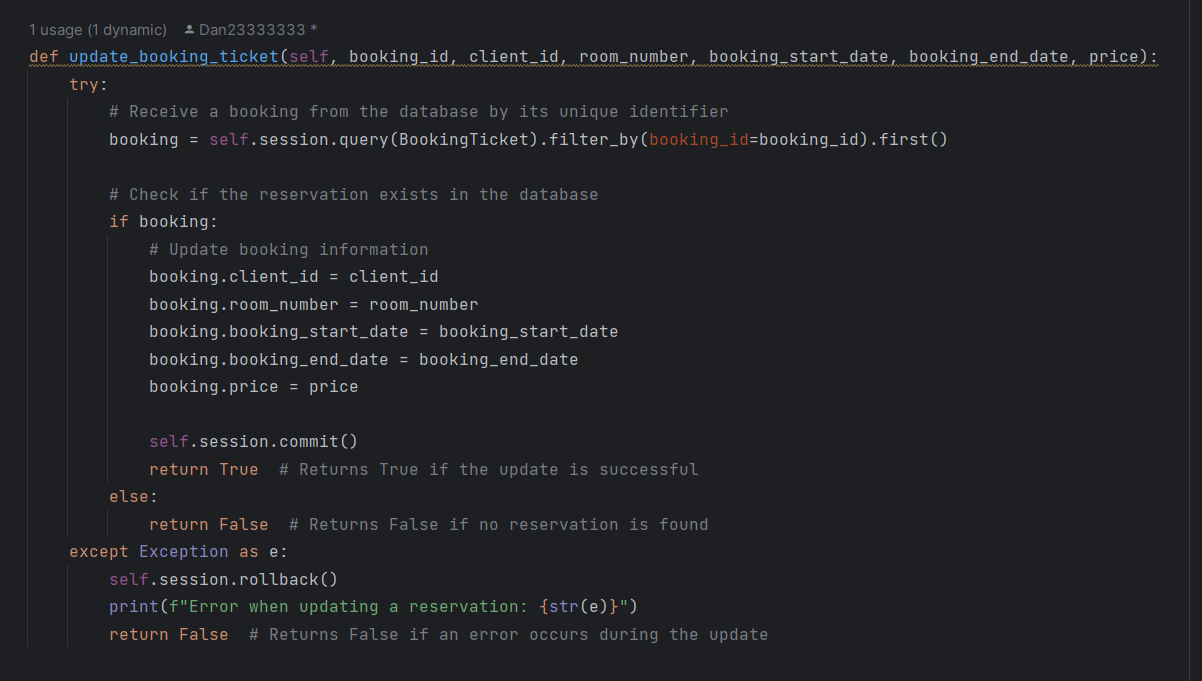
Рис. 2 – Логічна модель бази даних

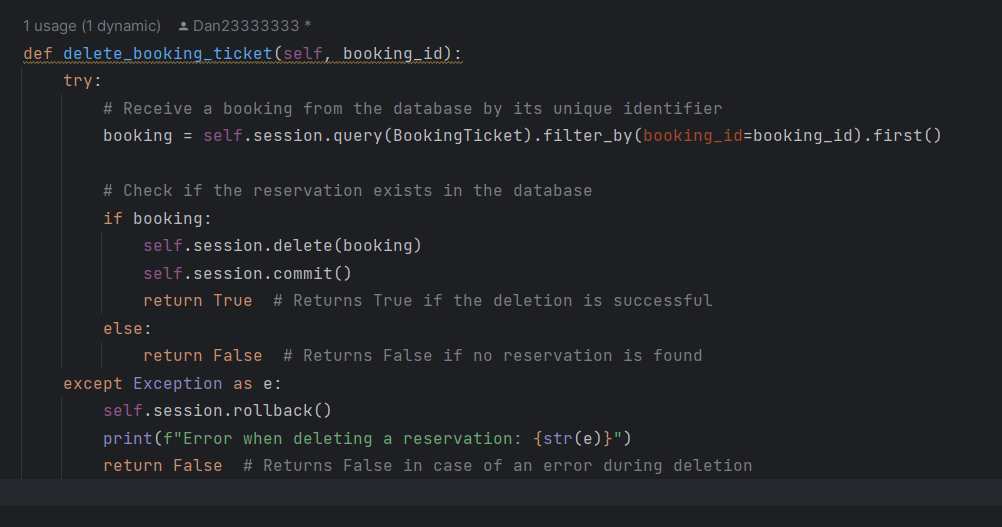
Для перетворення модуля “Model” програми, створеного в РГР, у вигляд об’єктно-реляційної моделі було використано бібліотеку “ SQLAlchemy”

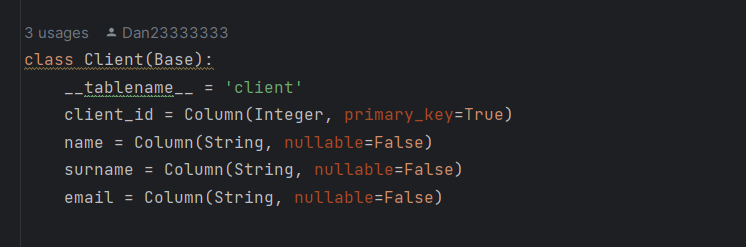
Код класів програми:

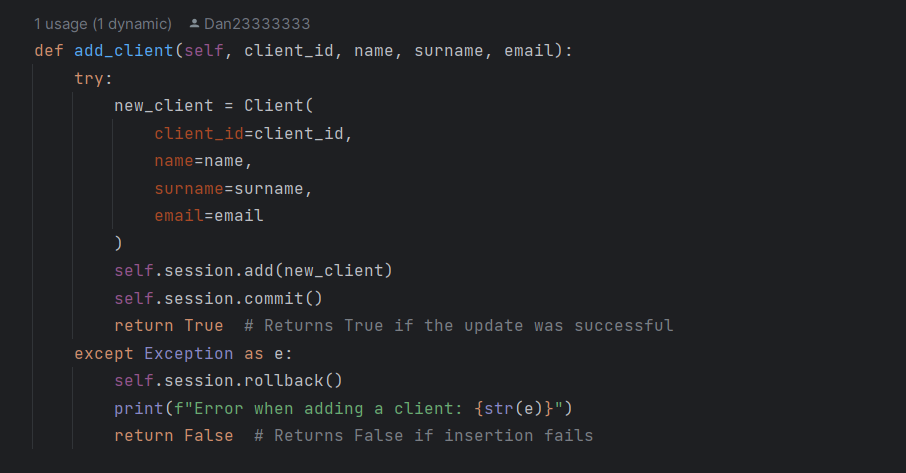


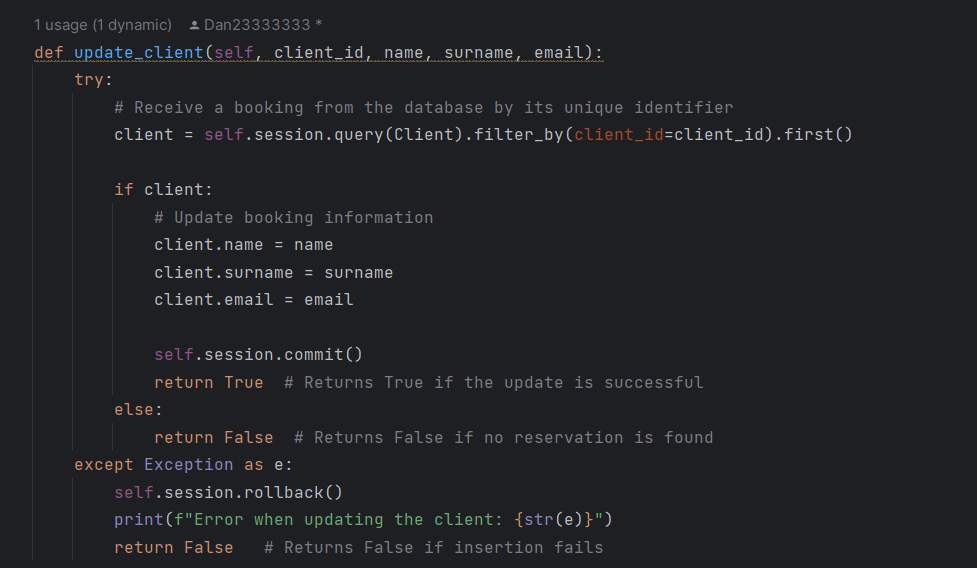


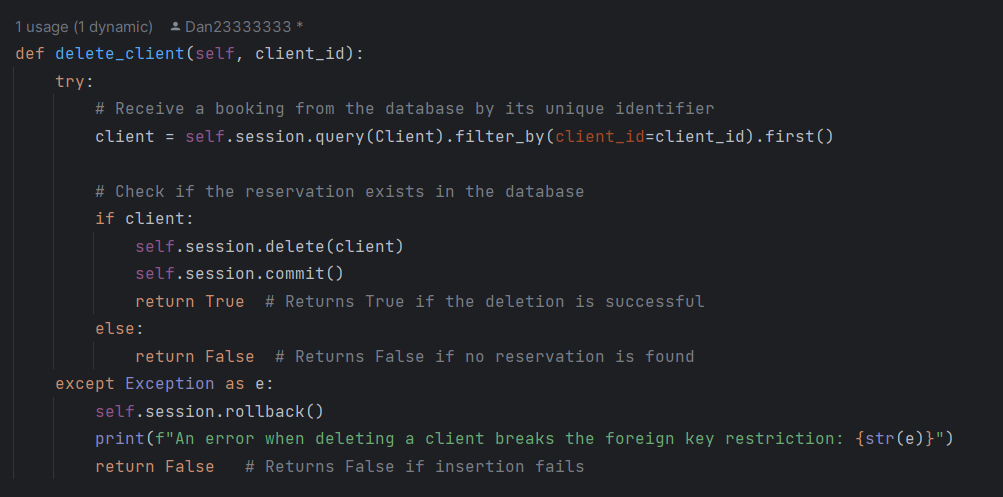


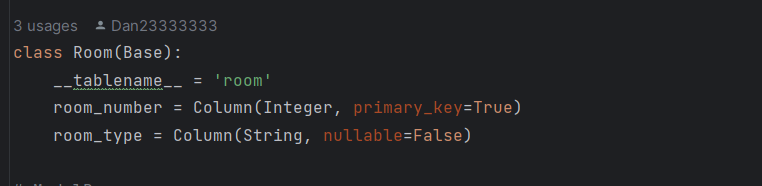


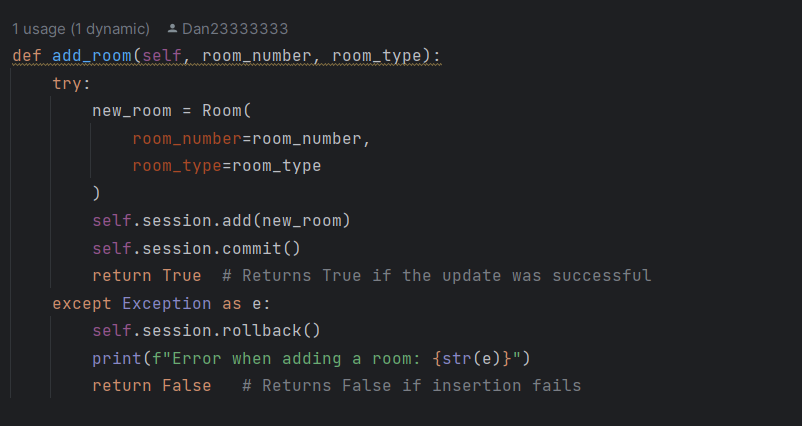


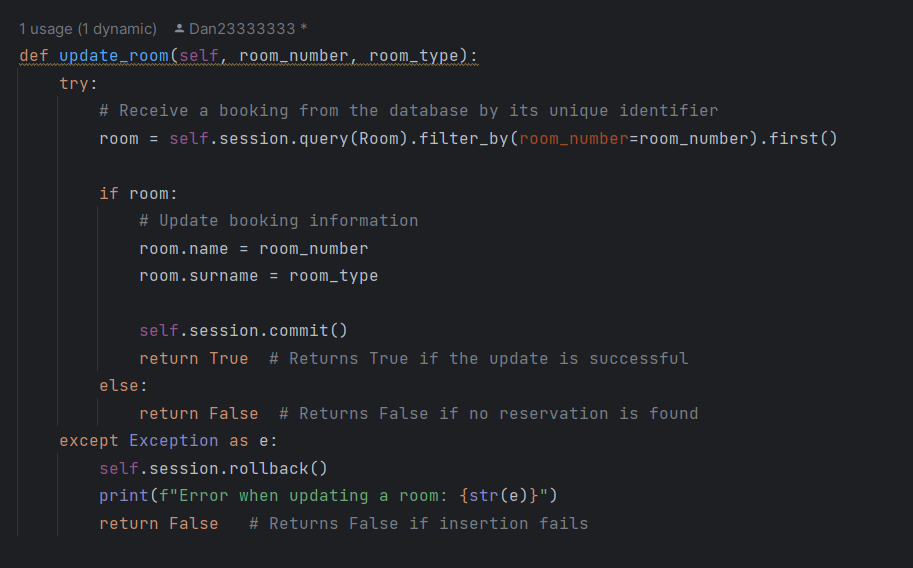


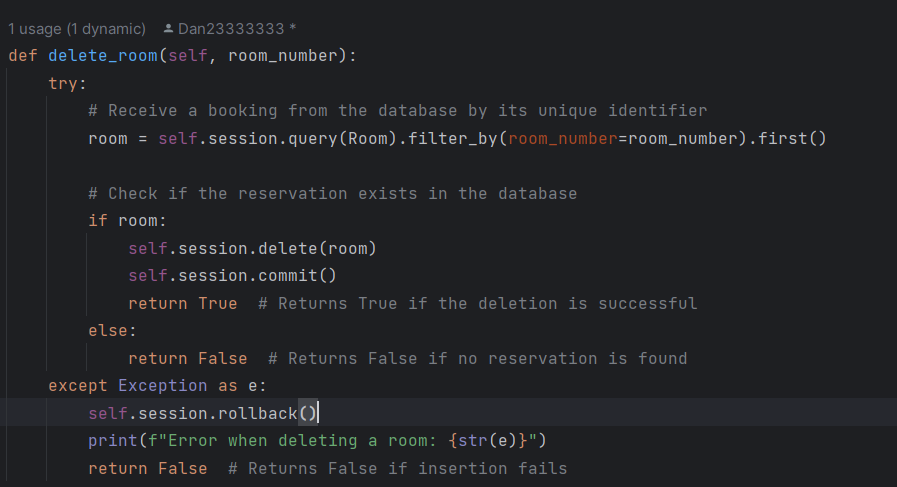






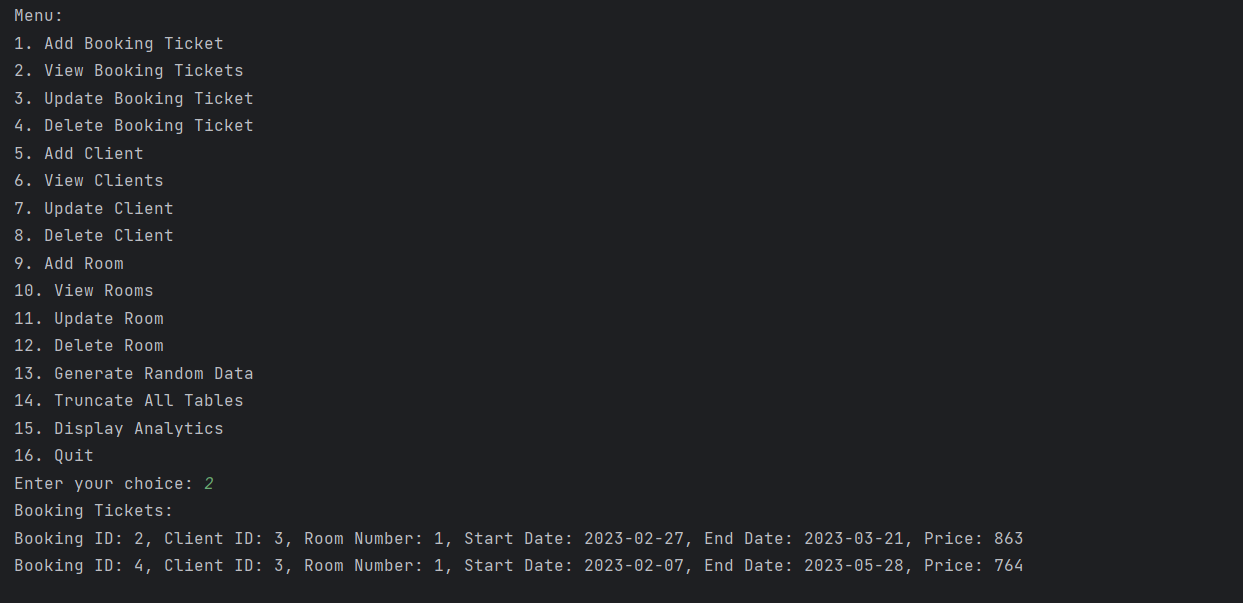


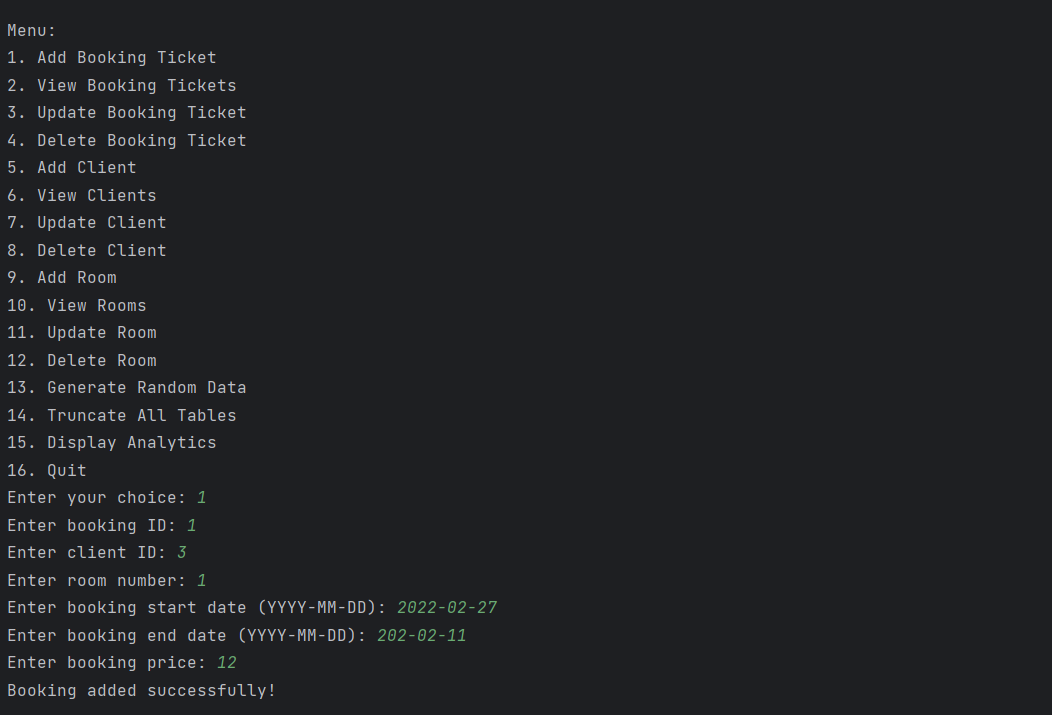


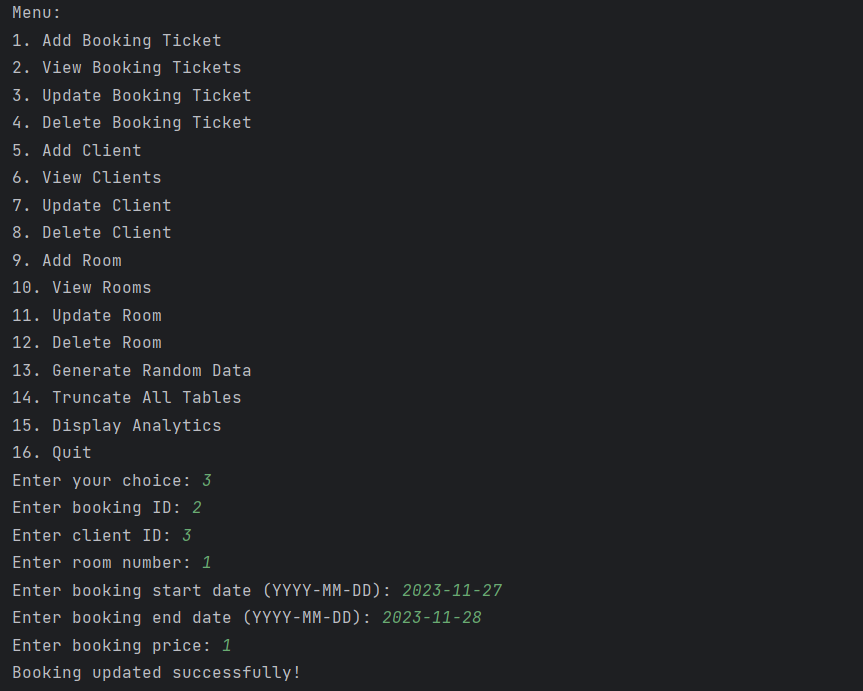


Програма працює ідентично програмі з РГР.

Приклад отримання даних







**Завдання 2**

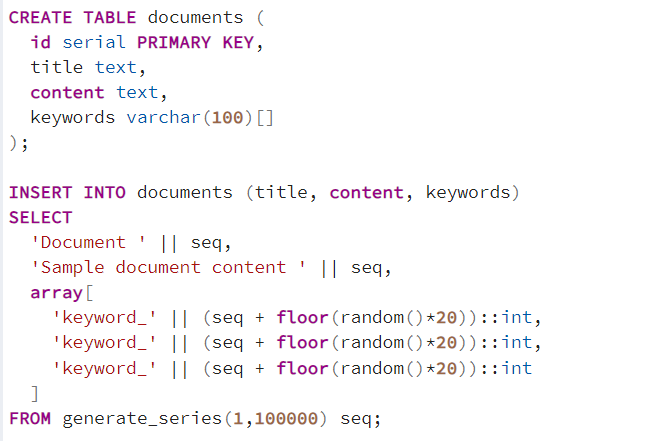
Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування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

**Варіант GIN, BRIN**

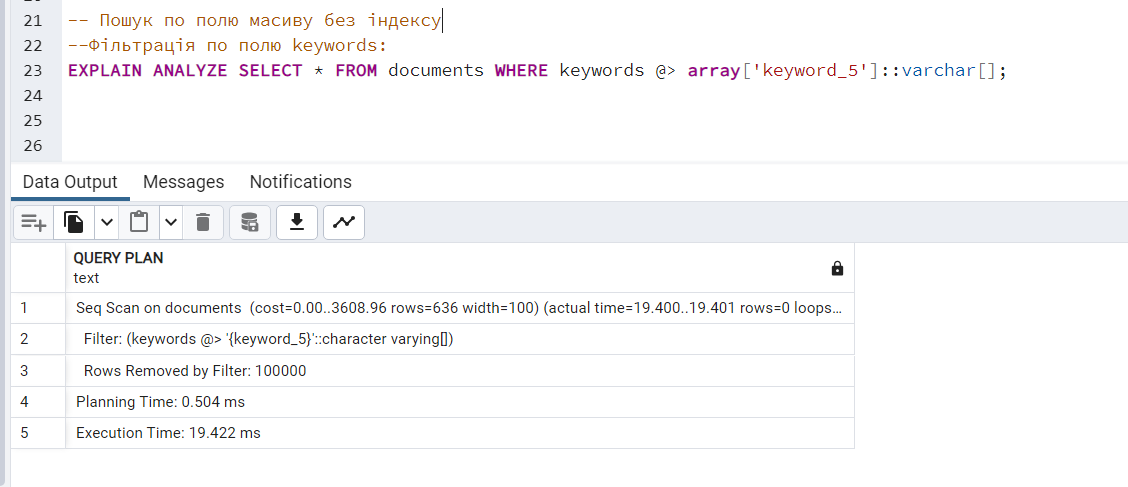
GIN

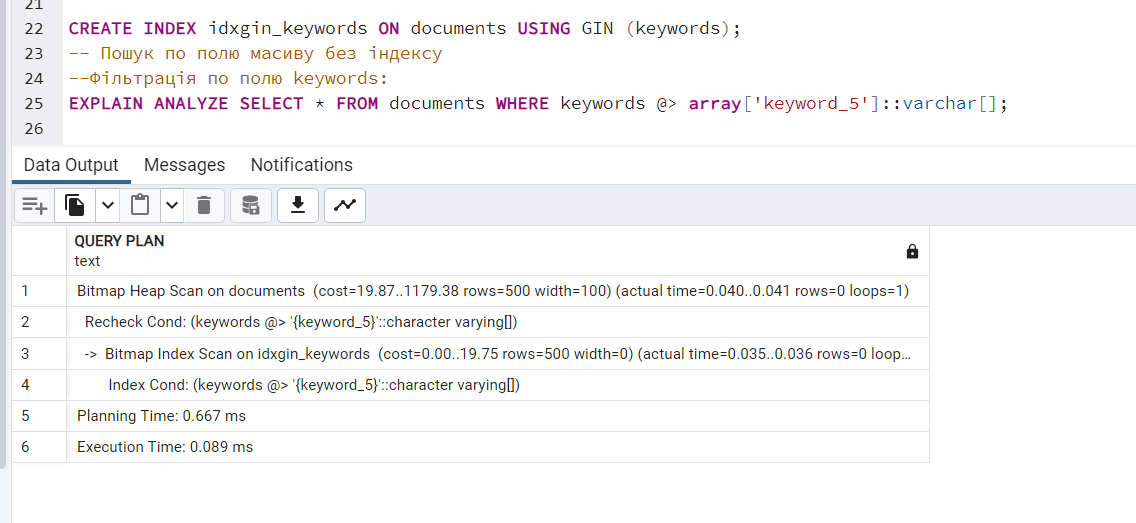
Для дослідження індексу була створена таблиця. У таблицю було занесено 1000000 записів.

Створення таблиці та її заповнення:



Візьмемо за приклад фільтрацію даних. Фільтрація по полю keywords:

Без індекса:  


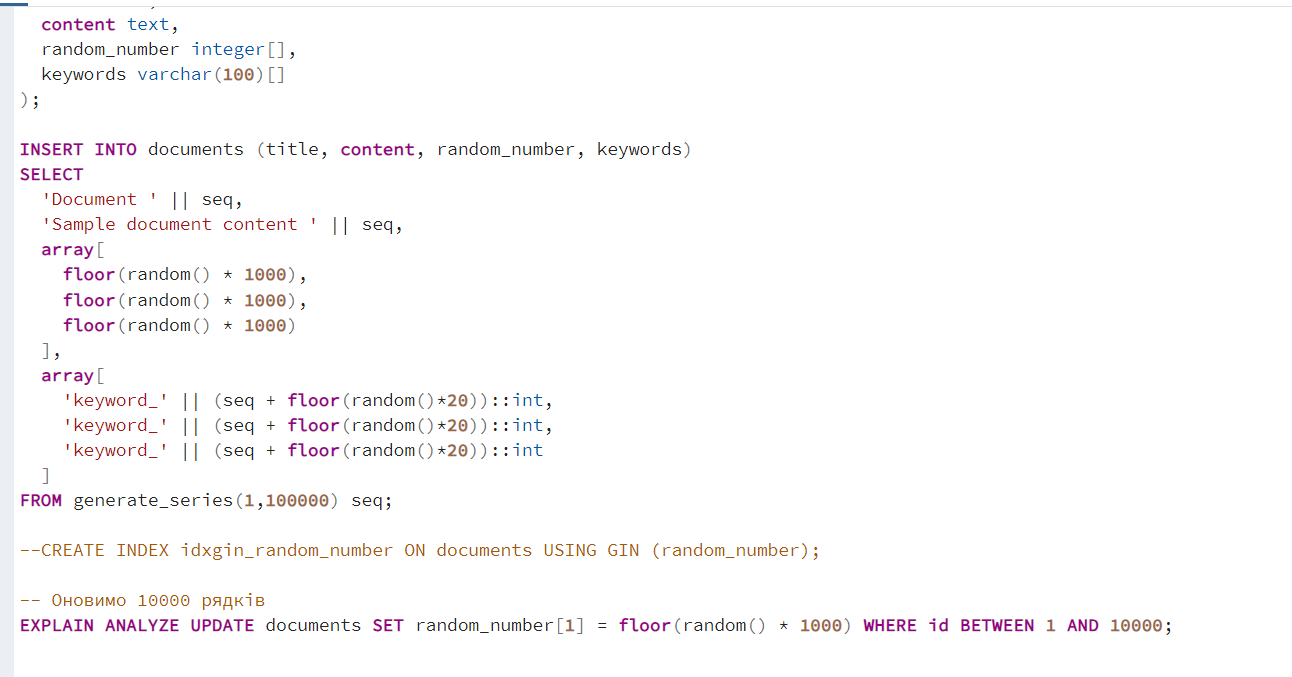
З ідексом:  


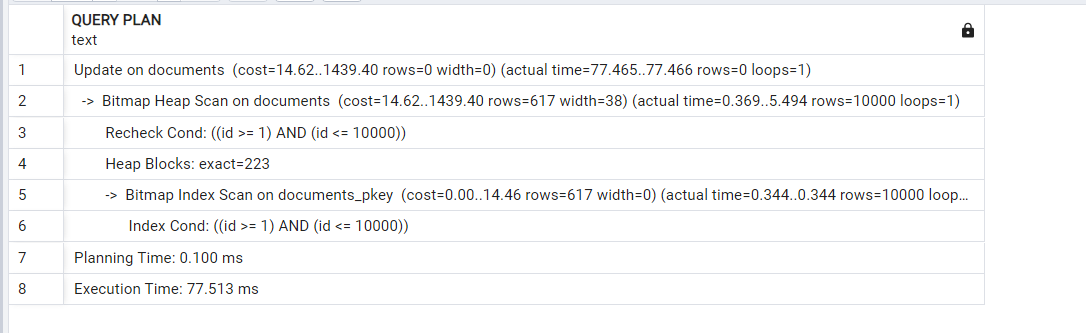
Недоліки GIN індексів.  
Індекси GIN є особливими, оскільки вони часто містять кілька записів індексу в кожному рядку, що вставляється. Це важливо для реалізації варіантів використання, які підтримує GIN, але спричиняє одну суттєву проблему: оновлення індексу обходиться дорого.

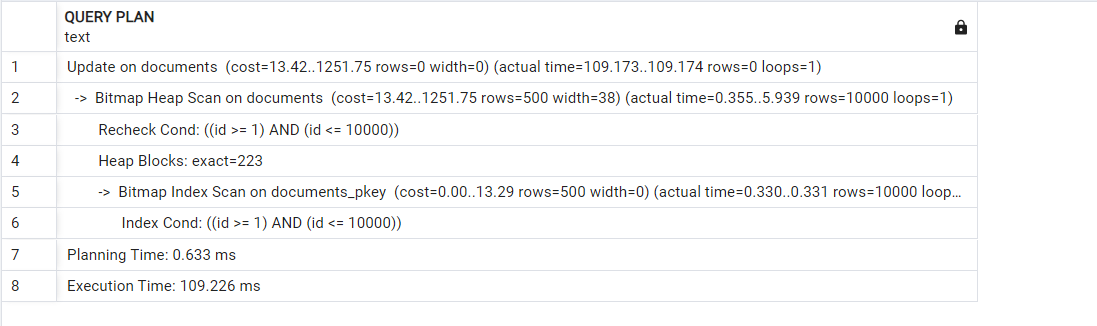
Візьмемо за приклад Оновлення 10000 рядків.

Без індекса:







З індексом:  


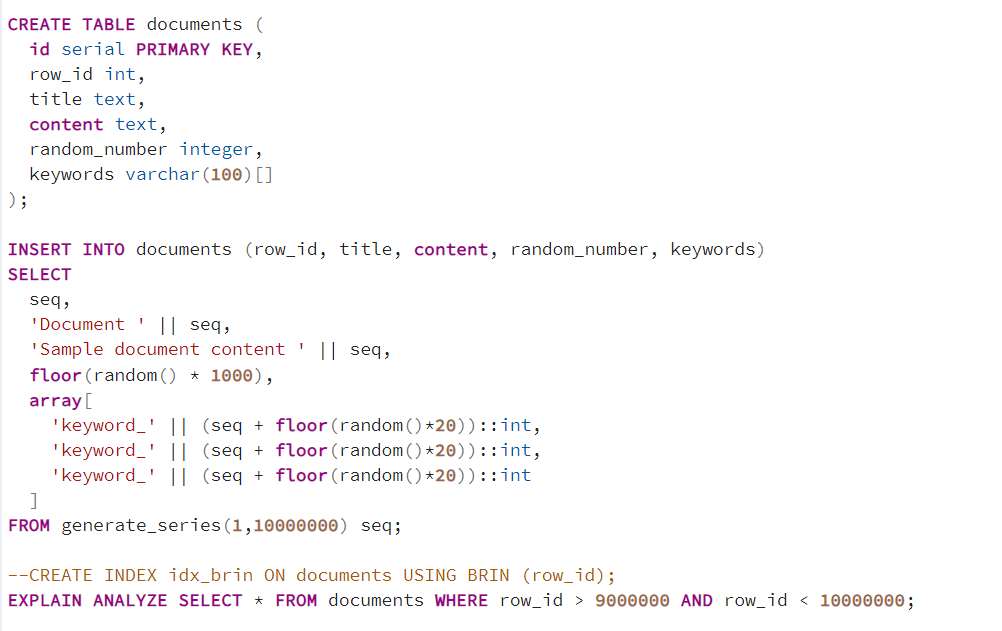
Висновок.

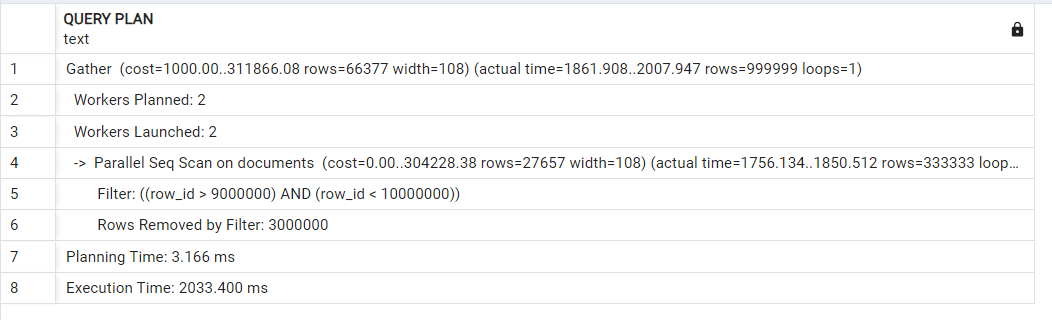
"Тип індексу GIN був розроблений для роботи з типами даних, які можна розділити, і ви хочете шукати значення окремих компонентів (елементи масиву, лексеми в текстовому документі тощо)" - Том Лейн  
Отже, ми можемо сказати він працює з типами даних, значення яких не є атомарними, а складаються з елементів. При цьому індексуються не самі значення, а окремі елементи; кожен елемент посилається на ті значення, в яких він зустрічається.

У нашому дослідженні ми довели, що GIN індекс є ефективним для роботи з повнотекстовим пошуком і через високу вартість оновлення індексу при вставці/оновленні даних, не є ефективним.

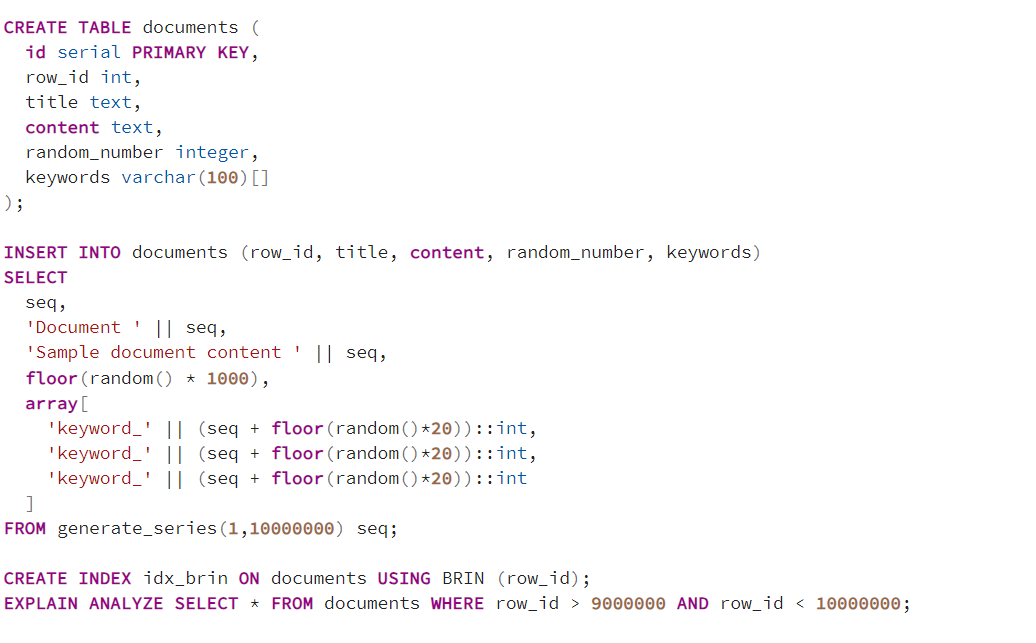
BRIN

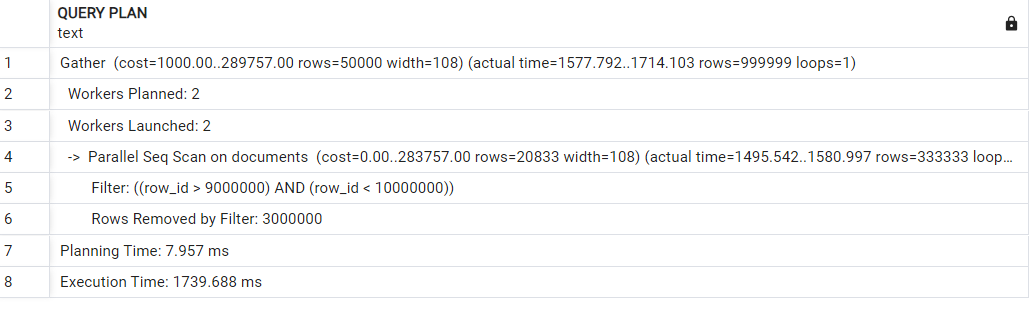
Для дослідження індексу була створена таблиця. У таблицю було занесено 10000000 записів.

Приклад без індекса:  




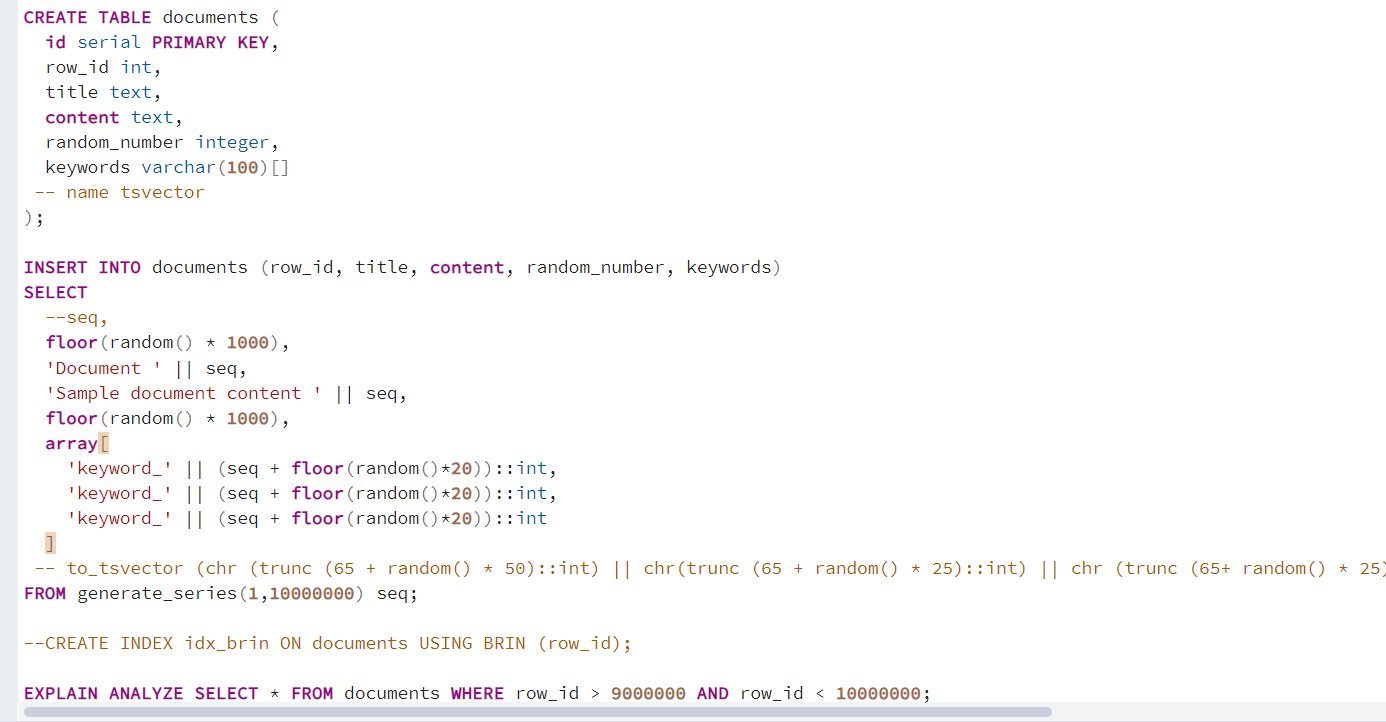
Приклад з використанням індексу:

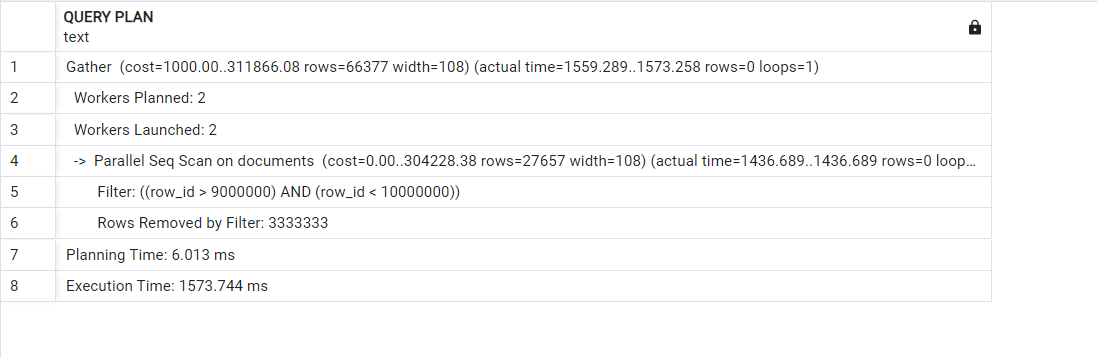




змінемо змінну row\_id з впорядкованої на рандомну

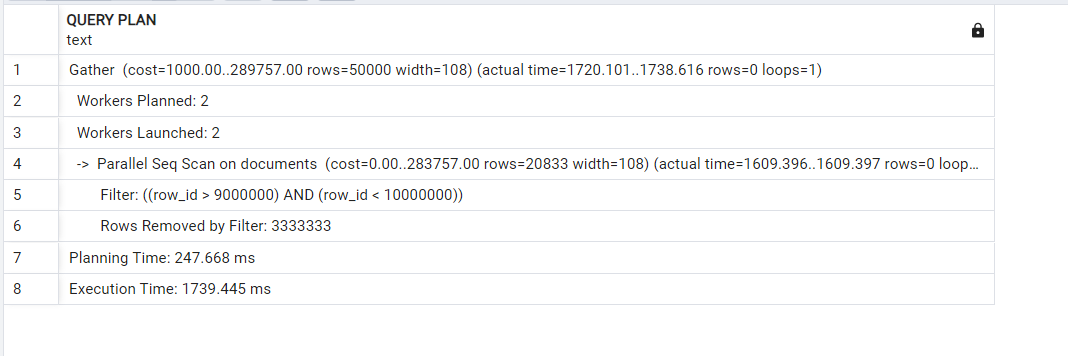
Приклад без індекса





Приклад з індексом





Висновок.

BRIN-індекси призначені для обробки дуже великих таблиць і спроектовані для роботи з даними, які мають "природну кореляцію" за певним критерієм впорядкування, наприклад, коли значення у стовпці рівномірно зростають або спадають. В таких випадках BRIN може швидко ідентифікувати блоки даних, які містять в собі потрібні значення або діапазони значень, що забезпечує більш ефективний пошук і зменшення кількості даних, які потрібно обробляти.

У нашому дослідженні ми використовували великі таблиці і довели, що при роботі з не впорядкованими даними BRIN індекс втрачає свою ефективність.

**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.

Умова для тригера – after delete, insert.

Тригер для видалення

**DROP** **TABLE** **IF** **EXISTS** documents;

**CREATE** **TABLE** documents (

**id** **serial** **PRIMARY** **KEY**,

row\_id **int**,

title text,

content text,

random\_number **integer**,

keywords **varchar**(100)[]

);

**INSERT** **INTO** documents (row\_id, title, content, random\_number, keywords)

**SELECT**

--seq,

**floor**(**random**() \* 1000),

'Document ' || seq,

'Sample document content ' || seq,

**floor**(**random**() \* 1000),

**array**[

'keyword\_' || (seq + **floor**(**random**()\*20))::int,

'keyword\_' || (seq + **floor**(**random**()\*20))::int,

'keyword\_' || (seq + **floor**(**random**()\*20))::int

]

**FROM** generate\_series(1,1000) seq;

**CREATE** **OR** **REPLACE** **FUNCTION** example\_trigger\_function()

**RETURNS** **TRIGGER** **AS** $$

**DECLARE**

some\_cursor **CURSOR** **FOR** **SELECT** \* **FROM** documents;

some\_row documents%**ROWTYPE**;

**BEGIN**

**BEGIN**

**OPEN** some\_cursor;

**LOOP**

**FETCH** some\_cursor **INTO** some\_row;

**EXIT** **WHEN** **NOT** **FOUND**;

**RAISE** NOTICE 'Обробка рядка з id=%', some\_row.**id**;

**END LOOP**;

**CLOSE** some\_cursor;

-- операція вставки (insert)

**IF** TG\_OP = 'INSERT' **THEN**

**RAISE** NOTICE 'Було вставлено новий рядок з **id**=%', NEW.id;

-- операція видалення (delete)

ELSIF TG\_OP = '**DELETE**' THEN

RAISE NOTICE 'Було видалено новий рядок з id=%', **OLD**.**id**;

**END IF**;

**RETURN** **NEW**;

**EXCEPTION**

**WHEN** others **THEN**

**RAISE** **EXCEPTION** 'Сталася помилка: %', **SQLERRM**;

**END**;

**END**;

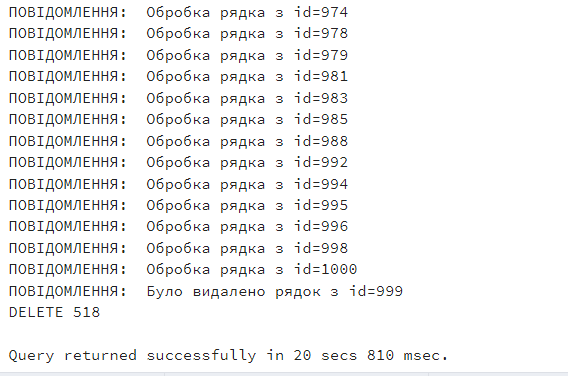
$$ **LANGUAGE** plpgsql;

**CREATE** **TRIGGER** example\_trigger

**AFTER** **INSERT** **OR** **DELETE** **ON** documents

**FOR** **EACH** **ROW** **EXECUTE** **FUNCTION** example\_trigger\_function();

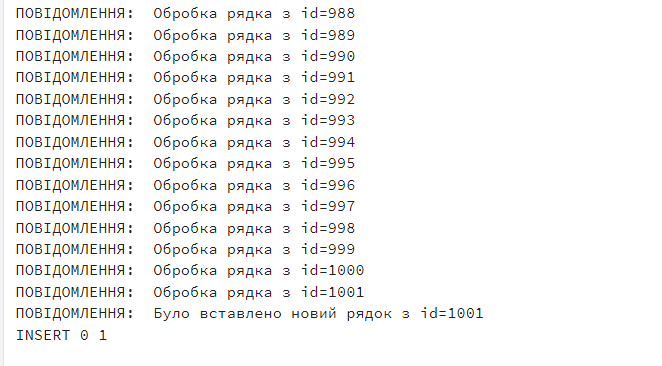
**DELETE** **FROM** documents **WHERE** random\_number > 500;



Тригер для вставки аналогічний видаленню, тільки в кінці   
- Вставка нового запису для демонстрації тригера

**INSERT** **INTO** documents (row\_id, title, content, random\_number, keywords)

**VALUES** (123, 'New Document', 'Content of the new document', 300, **ARRAY**['keyword\_1', 'keyword\_2']);



**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.

Рівні ізоляції транзакцій - це певний компроміс між швидкістю того, як інші побачать зміни і надійністю. Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у виборі між високою узгодженістю даних між транзакціями та швидкістю виконання цих транзакцій.

При паралельному виконанні транзакцій можливі винекнення таких проблем:

1. Втрачене оновлення

Ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.

1. “Брудне” читання

Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

1. Неповторюване читання

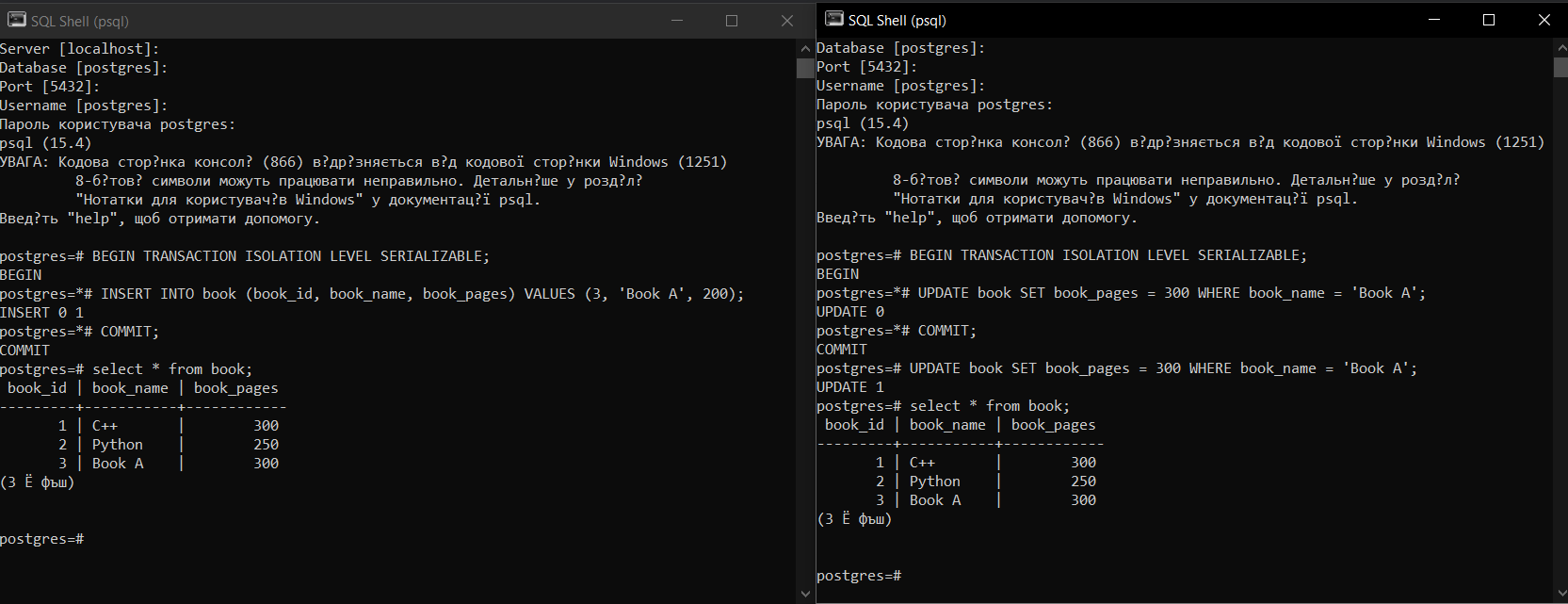
Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

1. Фантомне читання

Ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

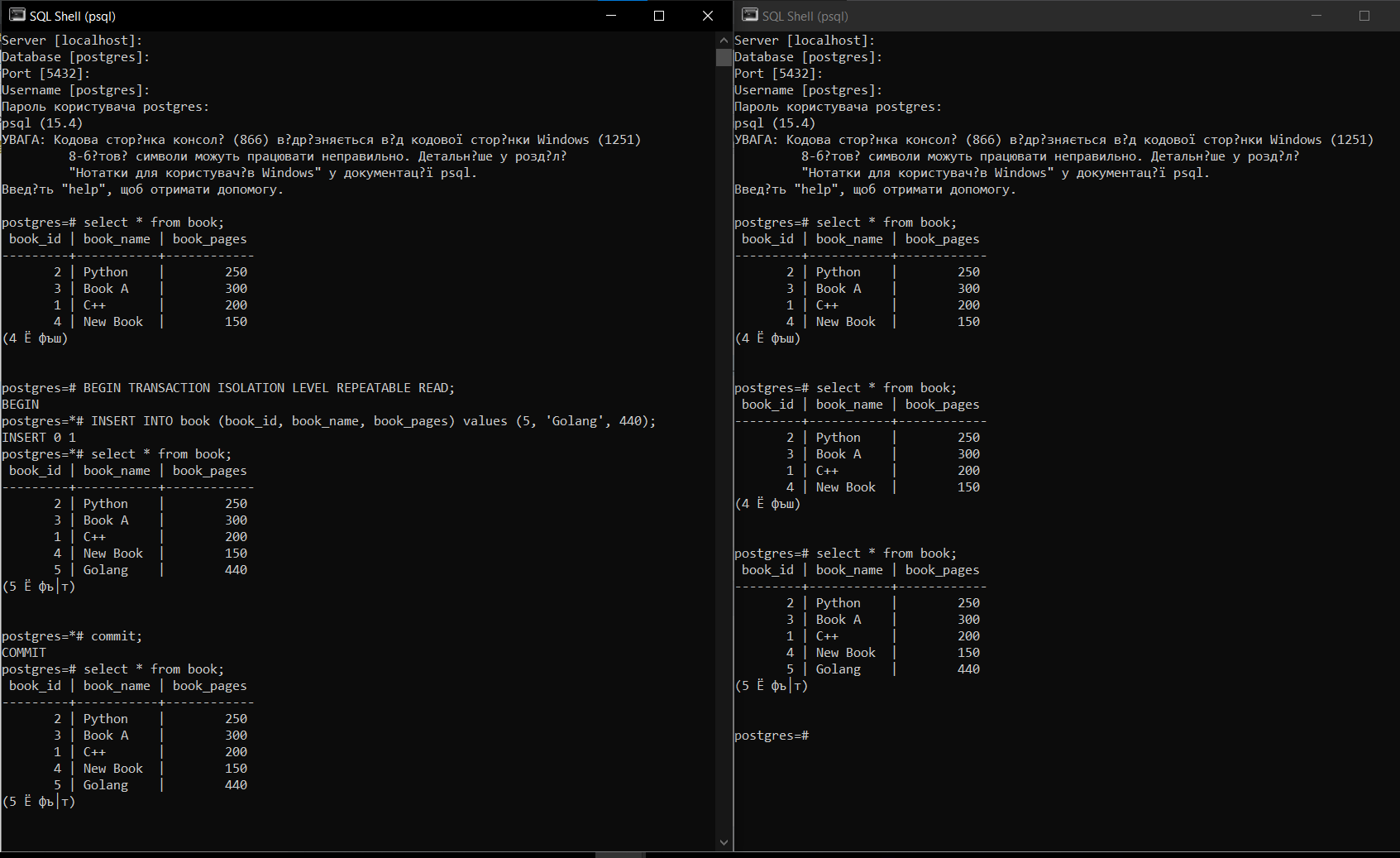
**Serializable**

Найбільш високий рівень ізольованості; транзакції повністю ізолюються одна від одної. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими, що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку).



**Repeatable read**

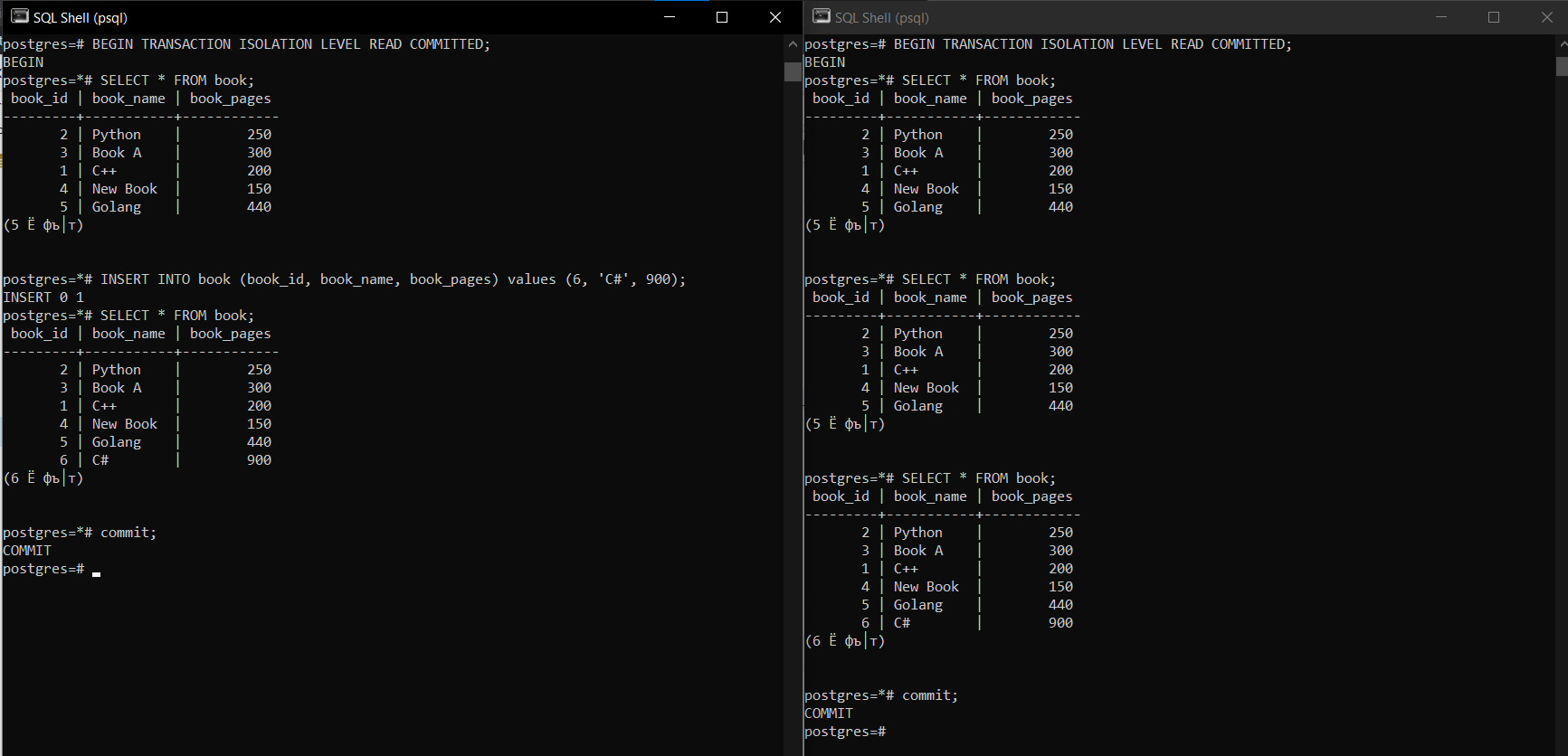
Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані).



**Read committed**

Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання.

Простіше кажучи, read committed - це рівень ізоляції, який гарантує, що будь-які прочитані дані будуть закомічені в момент читання.



Посилання на репозиторій   
https://github.com/Dan-live/c3s1\_Lab\_2\_DB