

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

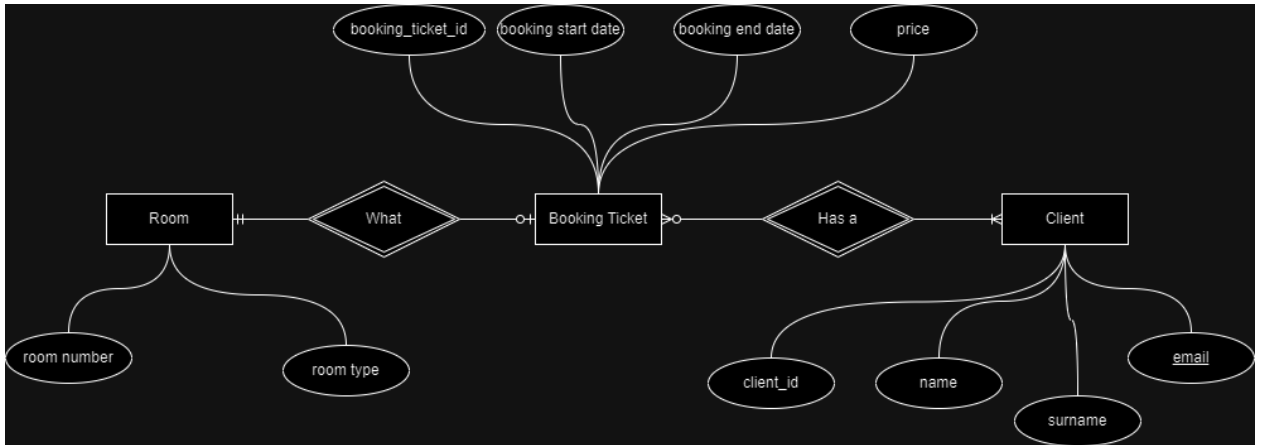


Рис. 1 - Концептуальна модель предметної області “Готельний бронювальний портал”

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

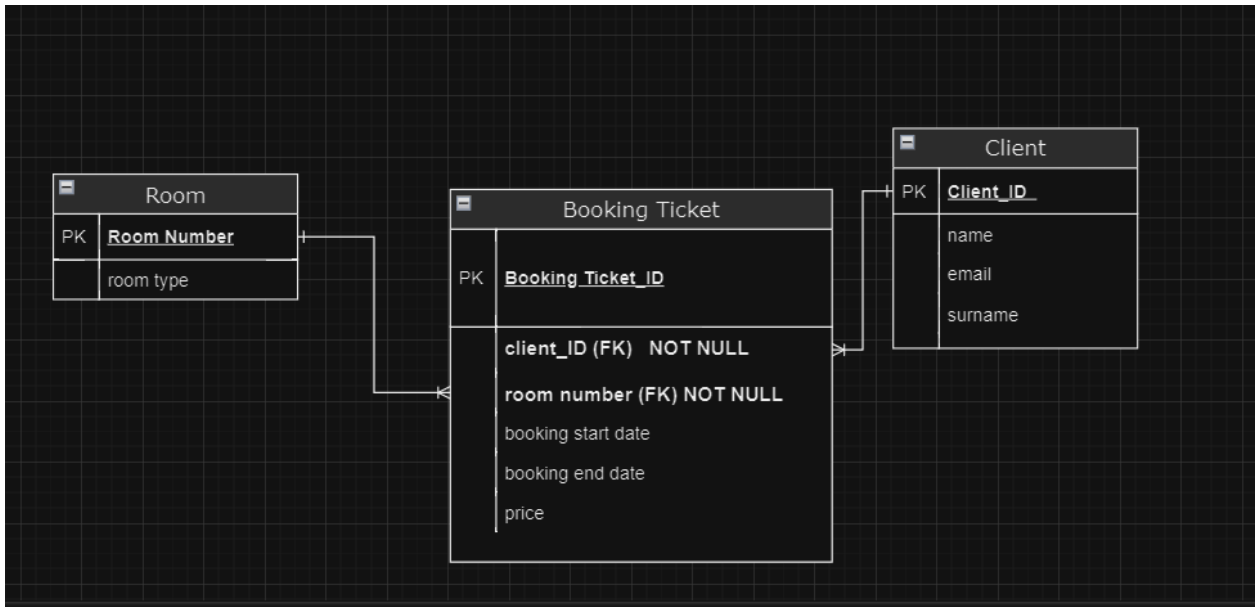


Рис. 2 – Логічна модель бази даних

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

Код класів програми:

```
3 usages  Dan23333333 *
class BookingTicket(Base):
    __tablename__ = 'booking_ticket'
    booking_id = Column(Integer, primary_key=True)
    client_id = Column(Integer, nullable=False)
    room_number = Column(Integer, nullable=False)
    booking_start_date = Column(Date, nullable=False)
    booking_end_date = Column(Date, nullable=False)
    price = Column(Numeric(precision=10, scale=2), nullable=False)
```

```

1 usage (1 dynamic)  ⚡ Dan23333333
def add_booking_ticket(self, booking_id, client_id, room_number, booking_start_date, booking_end_date, price):
    try:
        new_booking = BookingTicket(
            booking_id=booking_id,
            client_id=client_id,
            room_number=room_number,
            booking_start_date=booking_start_date,
            booking_end_date=booking_end_date,
            price=price
        )
        self.session.add(new_booking)
        self.session.commit()
        return True
    except Exception as e:
        self.session.rollback()
        print(f"Error when adding a booking: {str(e)}")
        return False

```

```

1 usage (1 dynamic)  ⚡ Dan23333333 *
def update_booking_ticket(self, booking_id, client_id, room_number, booking_start_date, booking_end_date, price):
    try:
        # Receive a booking from the database by its unique identifier
        booking = self.session.query(BookingTicket).filter_by(booking_id=booking_id).first()

        # Check if the reservation exists in the database
        if booking:
            # Update booking information
            booking.client_id = client_id
            booking.room_number = room_number
            booking.booking_start_date = booking_start_date
            booking.booking_end_date = booking_end_date
            booking.price = price

            self.session.commit()
            return True # Returns True if the update is successful
        else:
            return False # Returns False if no reservation is found
    except Exception as e:
        self.session.rollback()
        print(f"Error when updating a reservation: {str(e)}")
        return False # Returns False if an error occurs during the update

```

```

1 usage (1 dynamic)  ⚡ Dan23333333 *
def delete_booking_ticket(self, booking_id):
    try:
        # Receive a booking from the database by its unique identifier
        booking = self.session.query(BookingTicket).filter_by(booking_id=booking_id).first()

        # Check if the reservation exists in the database
        if booking:
            self.session.delete(booking)
            self.session.commit()
            return True # Returns True if the deletion is successful
        else:
            return False # Returns False if no reservation is found
    except Exception as e:
        self.session.rollback()
        print(f"Error when deleting a reservation: {str(e)}")
        return False # Returns False in case of an error during deletion

```

3 usages Dan23333333

```
class Client(Base):  
    __tablename__ = 'client'  
    client_id = Column(Integer, primary_key=True)  
    name = Column(String, nullable=False)  
    surname = Column(String, nullable=False)  
    email = Column(String, nullable=False)
```

1 usage (1 dynamic) Dan23333333

```
def add_client(self, client_id, name, surname, email):  
    try:  
        new_client = Client(  
            client_id=client_id,  
            name=name,  
            surname=surname,  
            email=email  
        )  
        self.session.add(new_client)  
        self.session.commit()  
        return True # Returns True if the update was successful  
    except Exception as e:  
        self.session.rollback()  
        print(f"Error when adding a client: {str(e)}")  
        return False # Returns False if insertion fails
```

1 usage (1 dynamic) Dan23333333 *

```
def update_client(self, client_id, name, surname, email):  
    try:  
        # Receive a booking from the database by its unique identifier  
        client = self.session.query(Client).filter_by(client_id=client_id).first()  
  
        if client:  
            # Update booking information  
            client.name = name  
            client.surname = surname  
            client.email = email  
  
            self.session.commit()  
            return True # Returns True if the update is successful  
        else:  
            return False # Returns False if no reservation is found  
    except Exception as e:  
        self.session.rollback()  
        print(f"Error when updating the client: {str(e)}")  
        return False # Returns False if insertion fails
```

```

1 usage (1 dynamic)  ⤴ Dan23333333 *
def delete_client(self, client_id):
    try:
        # Receive a booking from the database by its unique identifier
        client = self.session.query(Client).filter_by(client_id=client_id).first()

        # Check if the reservation exists in the database
        if client:
            self.session.delete(client)
            self.session.commit()
            return True # Returns True if the deletion is successful
        else:
            return False # Returns False if no reservation is found
    except Exception as e:
        self.session.rollback()
        print(f"An error when deleting a client breaks the foreign key restriction: {str(e)}")
        return False # Returns False if insertion fails

```

```

3 usages  ⤴ Dan23333333
class Room(Base):
    __tablename__ = 'room'
    room_number = Column(Integer, primary_key=True)
    room_type = Column(String, nullable=False)

```

```

1 usage (1 dynamic)  ⤴ Dan23333333
def add_room(self, room_number, room_type):
    try:
        new_room = Room(
            room_number=room_number,
            room_type=room_type
        )
        self.session.add(new_room)
        self.session.commit()
        return True # Returns True if the update was successful
    except Exception as e:
        self.session.rollback()
        print(f"Error when adding a room: {str(e)}")
        return False # Returns False if insertion fails

```

1 usage (1 dynamic) 👤 Dan23333333 *

```
def update_room(self, room_number, room_type):
    try:
        # Receive a booking from the database by its unique identifier
        room = self.session.query(Room).filter_by(room_number=room_number).first()

        if room:
            # Update booking information
            room.name = room_number
            room.surname = room_type

            self.session.commit()
            return True # Returns True if the update is successful
        else:
            return False # Returns False if no reservation is found
    except Exception as e:
        self.session.rollback()
        print(f"Error when updating a room: {str(e)}")
        return False # Returns False if insertion fails
```

1 usage (1 dynamic) 👤 Dan23333333 *

```
def delete_room(self, room_number):
    try:
        # Receive a booking from the database by its unique identifier
        room = self.session.query(Room).filter_by(room_number=room_number).first()

        # Check if the reservation exists in the database
        if room:
            self.session.delete(room)
            self.session.commit()
            return True # Returns True if the deletion is successful
        else:
            return False # Returns False if no reservation is found
    except Exception as e:
        self.session.rollback()
        print(f"Error when deleting a room: {str(e)}")
        return False # Returns False if insertion fails
```


Програма працює ідентично програмі з РГР.

Приклад отримання даних

```
Menu:
1. Add Booking Ticket
2. View Booking Tickets
3. Update Booking Ticket
4. Delete Booking Ticket
5. Add Client
6. View Clients
7. Update Client
8. Delete Client
9. Add Room
10. View Rooms
11. Update Room
12. Delete Room
13. Generate Random Data
14. Truncate All Tables
15. Display Analytics
16. Quit
Enter your choice: 2
Booking Tickets:
Booking ID: 2, Client ID: 3, Room Number: 1, Start Date: 2023-02-27, End Date: 2023-03-21, Price: 863
Booking ID: 4, Client ID: 3, Room Number: 1, Start Date: 2023-02-07, End Date: 2023-05-28, Price: 764
```

```
Menu:
1. Add Booking Ticket
2. View Booking Tickets
3. Update Booking Ticket
4. Delete Booking Ticket
5. Add Client
6. View Clients
7. Update Client
8. Delete Client
9. Add Room
10. View Rooms
11. Update Room
12. Delete Room
13. Generate Random Data
14. Truncate All Tables
15. Display Analytics
16. Quit
Enter your choice: 1
Enter booking ID: 1
Enter client ID: 3
Enter room number: 1
Enter booking start date (YYYY-MM-DD): 2022-02-27
Enter booking end date (YYYY-MM-DD): 202-02-11
Enter booking price: 12
Booking added successfully!
```

```
Menu:
1. Add Booking Ticket
2. View Booking Tickets
3. Update Booking Ticket
4. Delete Booking Ticket
5. Add Client
6. View Clients
7. Update Client
8. Delete Client
9. Add Room
10. View Rooms
11. Update Room
12. Delete Room
13. Generate Random Data
14. Truncate All Tables
15. Display Analytics
16. Quit
Enter your choice: 3
Enter booking ID: 2
Enter client ID: 3
Enter room number: 1
Enter booking start date (YYYY-MM-DD): 2023-11-27
Enter booking end date (YYYY-MM-DD): 2023-11-28
Enter booking price: 1
Booking updated successfully!
```

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:

```
CREATE TABLE documents (  
    id serial PRIMARY KEY,  
    title text,  
    content text,  
    keywords varchar(100)[]  
);  
  
INSERT INTO documents (title, content, keywords)  
SELECT  
    'Document ' || seq,  
    'Sample document content ' || seq,  
    array[  
        'keyword_' || (seq + floor(random()*20))::int,  
        'keyword_' || (seq + floor(random()*20))::int,  
        'keyword_' || (seq + floor(random()*20))::int  
    ]  
FROM generate_series(1,100000) seq;
```

Візьмемо за приклад фільтрацію даних. Фільтрація по полю keywords:

Без індекса:

```
21 -- Пошук по полю масиву без індексу|  
22 --Фільтрація по полю keywords:  
23 EXPLAIN ANALYZE SELECT * FROM documents WHERE keywords @> array['keyword_5']::varchar[];  
24  
25  
26
```

Data Output Messages Notifications



	QUERY PLAN
	text
1	Seq Scan on documents (cost=0.00..3608.96 rows=636 width=100) (actual time=19.400..19.401 rows=0 loops=...
2	Filter: (keywords @> '{keyword_5}':character varying[])
3	Rows Removed by Filter: 100000
4	Planning Time: 0.504 ms
5	Execution Time: 19.422 ms

З ідексом:

```
21
22 CREATE INDEX idxgin_keywords ON documents USING GIN (keywords);
23 -- Пошук по полю масиву без індексу
24 --Фільтрація по полю keywords:
25 EXPLAIN ANALYZE SELECT * FROM documents WHERE keywords @> array['keyword_5']::varchar[];
26
```

Data Output Messages Notifications



QUERY PLAN		text	
1	Bitmap Heap Scan on documents	(cost=19.87..1179.38 rows=500 width=100) (actual time=0.040..0.041 rows=0 loops=1)	
2	Recheck Cond: (keywords @> 'keyword_5')::character varying[])		
3	-> Bitmap Index Scan on idxgin_keywords	(cost=0.00..19.75 rows=500 width=0) (actual time=0.035..0.036 rows=0 loop...	
4	Index Cond: (keywords @> 'keyword_5')::character varying[])		
5	Planning Time: 0.667 ms		
6	Execution Time: 0.089 ms		

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:

```
CREATE TABLE documents (  
  id serial PRIMARY KEY,  
  title text,  
  
  content text,  
  random_number integer[],  
  keywords varchar(100)[]  
);  
  
INSERT INTO documents (title, content, random_number, keywords)  
SELECT  
  'Document ' || seq,  
  'Sample document content ' || seq,  
  array[  
    floor(random() * 1000),  
    floor(random() * 1000),  
    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(1,100000) seq;  
  
--CREATE INDEX idxgin_random_number ON documents USING GIN (random_number);  
  
-- Оновимо 10000 рядків  
EXPLAIN ANALYZE UPDATE documents SET random_number[1] = floor(random() * 1000) WHERE id BETWEEN 1 AND 10000;
```

	QUERY PLAN text	
1	Update on documents (cost=14.62..1439.40 rows=0 width=0) (actual time=77.465..77.466 rows=0 loops=1)	
2	-> Bitmap Heap Scan on documents (cost=14.62..1439.40 rows=617 width=38) (actual time=0.369..5.494 rows=10000 loops=1)	
3	Recheck Cond: ((id >= 1) AND (id <= 10000))	
4	Heap Blocks: exact=223	
5	-> Bitmap Index Scan on documents_pkey (cost=0.00..14.46 rows=617 width=0) (actual time=0.344..0.344 rows=10000 loop...	
6	Index Cond: ((id >= 1) AND (id <= 10000))	
7	Planning Time: 0.100 ms	
8	Execution Time: 77.513 ms	

З індексом:

	QUERY PLAN text	
1	Update on documents (cost=13.42..1251.75 rows=0 width=0) (actual time=109.173..109.174 rows=0 loops=1)	
2	-> Bitmap Heap Scan on documents (cost=13.42..1251.75 rows=500 width=38) (actual time=0.355..5.939 rows=10000 loops=1)	
3	Recheck Cond: ((id >= 1) AND (id <= 10000))	
4	Heap Blocks: exact=223	
5	-> Bitmap Index Scan on documents_pkey (cost=0.00..13.29 rows=500 width=0) (actual time=0.330..0.331 rows=10000 loop...	
6	Index Cond: ((id >= 1) AND (id <= 10000))	
7	Planning Time: 0.633 ms	
8	Execution Time: 109.226 ms	

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:

```
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,  
  '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(1,10000000) seq;  
  
--CREATE INDEX idx_brin ON documents USING BRIN (row_id);  
EXPLAIN ANALYZE SELECT * FROM documents WHERE row_id > 9000000 AND row_id < 10000000;
```

	QUERY PLAN	
	text	🔒
1	Gather (cost=1000.00..311866.08 rows=66377 width=108) (actual time=1861.908..2007.947 rows=999999 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on documents (cost=0.00..304228.38 rows=27657 width=108) (actual time=1756.134..1850.512 rows=333333 loop...	
5	Filter: ((row_id > 9000000) AND (row_id < 10000000))	
6	Rows Removed by Filter: 3000000	
7	Planning Time: 3.166 ms	
8	Execution Time: 2033.400 ms	

Приклад з використанням індексу:


```
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,  
  '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(1,10000000) seq;  
  
CREATE INDEX idx_brin ON documents USING BRIN (row_id);  
EXPLAIN ANALYZE SELECT * FROM documents WHERE row_id > 9000000 AND row_id < 10000000;
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..289757.00 rows=50000 width=108) (actual time=1577.792..1714.103 rows=999999 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on documents (cost=0.00..283757.00 rows=20833 width=108) (actual time=1495.542..1580.997 rows=333333 loop...	
5	Filter: ((row_id > 9000000) AND (row_id < 10000000))	
6	Rows Removed by Filter: 3000000	
7	Planning Time: 7.957 ms	
8	Execution Time: 1739.688 ms	

змінимо змінну row_id з впорядкованої на рандомну


Приклад без індекса

```
CREATE TABLE documents (  
  id serial PRIMARY KEY,  
  row_id int,  
  title text,  
  content text,  
  random_number integer,  
  keywords varchar(100[])  
  -- name tsvector  
);  
  
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  
  ]  
  -- to_tsvector (chr (trunc (65 + random() * 50)::int) || chr(trunc (65 + random() * 25)::int) || chr (trunc (65+ random() * 25)  
FROM generate_series(1,10000000) seq;  
  
--CREATE INDEX idx_brin ON documents USING BRIN (row_id);  
  
EXPLAIN ANALYZE SELECT * FROM documents WHERE row_id > 9000000 AND row_id < 10000000;
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..311866.08 rows=66377 width=108) (actual time=1559.289..1573.258 rows=0 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on documents (cost=0.00..304228.38 rows=27657 width=108) (actual time=1436.689..1436.689 rows=0 loop...	
5	Filter: ((row_id > 9000000) AND (row_id < 10000000))	
6	Rows Removed by Filter: 3333333	
7	Planning Time: 6.013 ms	
8	Execution Time: 1573.744 ms	

Приклад з індексом

```
CREATE TABLE documents (  
  id serial PRIMARY KEY,  
  row_id int,  
  title text,  
  content text,  
  random_number integer,  
  keywords varchar(100)[]  
  -- name tsvector  
);  
  
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  
  ]  
  -- to_tsvector (chr (trunc (65 + random() * 50)::int) || chr(trunc (65 + random() * 25)::int) || chr (trunc  
FROM generate_series(1,10000000) seq;  
  
CREATE INDEX idx_brin ON documents USING BRIN (row_id);  
  
EXPLAIN ANALYZE SELECT * FROM documents WHERE row_id > 9000000 AND row_id < 10000000;
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..289757.00 rows=50000 width=108) (actual time=1720.101..1738.616 rows=0 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on documents (cost=0.00..283757.00 rows=20833 width=108) (actual time=1609.396..1609.397 rows=0 loop...	
5	Filter: ((row_id > 9000000) AND (row_id < 10000000))	
6	Rows Removed by Filter: 3333333	
7	Planning Time: 247.668 ms	
8	Execution Time: 1739.445 ms	

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;
```

```
ПОВІДОМЛЕННЯ:  Обробка рядка з id=974  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=978  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=979  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=981  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=983  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=985  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=988  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=992  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=994  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=995  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=996  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=998  
ПОВІДОМЛЕННЯ:  Обробка рядка з id=1000  
ПОВІДОМЛЕННЯ:  Було видалено рядок з id=999  
DELETE 518
```

```
Query returned successfully in 20 secs 810 msec.
```

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']);
```

```
ПОВІДОМЛЕННЯ: Обробка рядка з id=988
ПОВІДОМЛЕННЯ: Обробка рядка з id=989
ПОВІДОМЛЕННЯ: Обробка рядка з id=990
ПОВІДОМЛЕННЯ: Обробка рядка з id=991
ПОВІДОМЛЕННЯ: Обробка рядка з id=992
ПОВІДОМЛЕННЯ: Обробка рядка з id=993
ПОВІДОМЛЕННЯ: Обробка рядка з id=994
ПОВІДОМЛЕННЯ: Обробка рядка з id=995
ПОВІДОМЛЕННЯ: Обробка рядка з id=996
ПОВІДОМЛЕННЯ: Обробка рядка з id=997
ПОВІДОМЛЕННЯ: Обробка рядка з id=998
ПОВІДОМЛЕННЯ: Обробка рядка з id=999
ПОВІДОМЛЕННЯ: Обробка рядка з id=1000
ПОВІДОМЛЕННЯ: Обробка рядка з id=1001
ПОВІДОМЛЕННЯ: Було вставлено новий рядок з id=1001
INSERT 0 1
```

Завдання 4

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і способ 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 виникнення таких проблем:

1. Втрачене оновлення

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

2. “Брудне” читання

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

3. Неповторюване читання

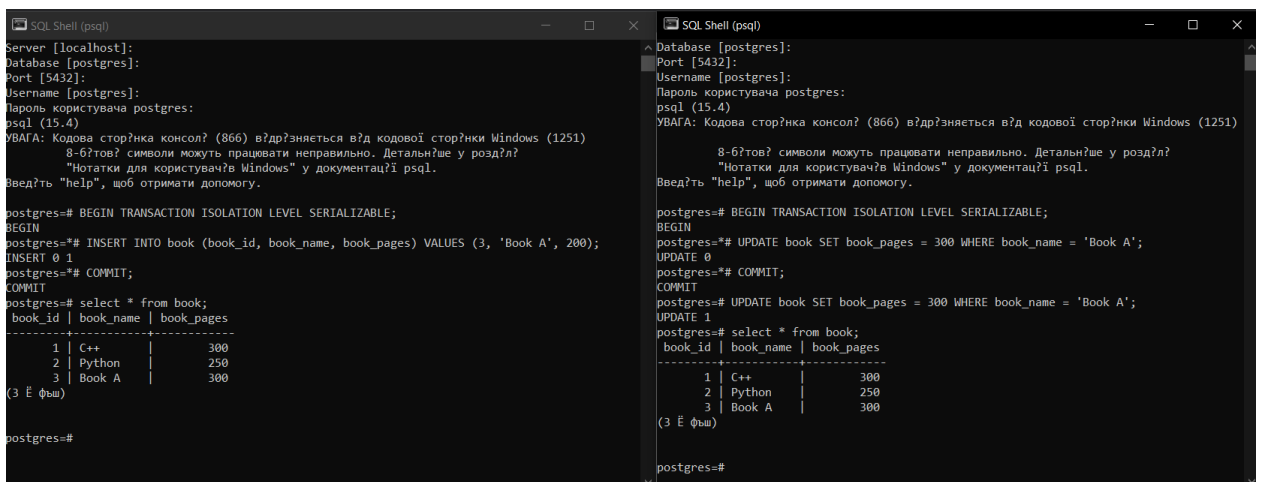
Ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

4. Фантомне читання

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

Serializable

Найбільш високий рівень ізолюваності; 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).



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (15.4)
УВАГА: Кодова сторінка консолі (866) відрізняється від кодової сторінки Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі?
"Нотатки для користувача Windows" у документації psql.
Введіть "help", щоб отримати допомогу.

postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
postgres=# INSERT INTO book (book_id, book_name, book_pages) VALUES (3, 'Book A', 200);
INSERT 0 1
postgres=# COMMIT;
COMMIT
postgres=# select * from book;
 book_id | book_name | book_pages
-----+-----+-----
      1 | C++       |        300
      2 | Python    |        250
      3 | Book A    |        300
(3 ř  w)

postgres=#
```

```
SQL Shell (psql)
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (15.4)
УВАГА: Кодова сторінка консолі (866) відрізняється від кодової сторінки Windows (1251)
8-бітові символи можуть працювати неправильно. Детальніше у розділі?
"Нотатки для користувача Windows" у документації psql.
Введіть "help", щоб отримати допомогу.

postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
postgres=# UPDATE book SET book_pages = 300 WHERE book_name = 'Book A';
UPDATE 0
postgres=# COMMIT;
COMMIT
postgres=# UPDATE book SET book_pages = 300 WHERE book_name = 'Book A';
UPDATE 1
postgres=# select * from book;
 book_id | book_name | book_pages
-----+-----+-----
      1 | C++       |        300
      2 | Python    |        250
      3 | Book A    |        300
(3 ř  w)

postgres=#
```

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).

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (15.4)
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=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
(4 rows)

postgres=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
postgres=# INSERT INTO book (book_id, book_name, book_pages) values (5, 'Golang', 440);
INSERT 0 1
postgres=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

postgres=# commit;
COMMIT
postgres=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль користувача postgres:
psql (15.4)
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=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
(4 rows)

postgres=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
(4 rows)

postgres=# select * from book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

postgres=#
```

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.

```
SQL Shell (psql)
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
postgres=# SELECT * FROM book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

postgres=# INSERT INTO book (book_id, book_name, book_pages) values (6, 'C#', 900);
INSERT 0 1
postgres=# SELECT * FROM book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
      6 | C#       |         900
(6 rows)

postgres=# commit;
COMMIT
postgres=#

SQL Shell (psql)
postgres=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
postgres=# SELECT * FROM book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

postgres=# SELECT * FROM book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
(5 rows)

postgres=# SELECT * FROM book;
 book_id | book_name | book_pages
-----
      2 | Python   |         250
      3 | Book A   |         300
      1 | C++      |         200
      4 | New Book |         150
      5 | Golang   |         440
      6 | C#       |         900
(6 rows)

postgres=# commit;
COMMIT
postgres=#
```

Посилання на репозиторій

https://github.com/Dan-live/c3s1_Lab_2_DB