



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 3 курсу

ФПМ групи КВ-11

Рибалка Денис

Перевірив:

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.

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: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.

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 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.

Варіант 22

22	Hash, BRIN	after delete, insert
----	------------	----------------------

Середовище для відлагодження SQL-запитів до бази даних – PgAdmin4.

Мова програмування – Python 3.10.11

Середовище розробки програмного забезпечення – VS Code

Бібліотека взаємодії з PostgreSQL - Psycopg2

URL репозиторію з вихідним кодом - <https://github.com/denisrybalka-kpi/lab2-bd>

Модель «сутність-зв'язок» предметної галузі для обліку експонатів у музеї.

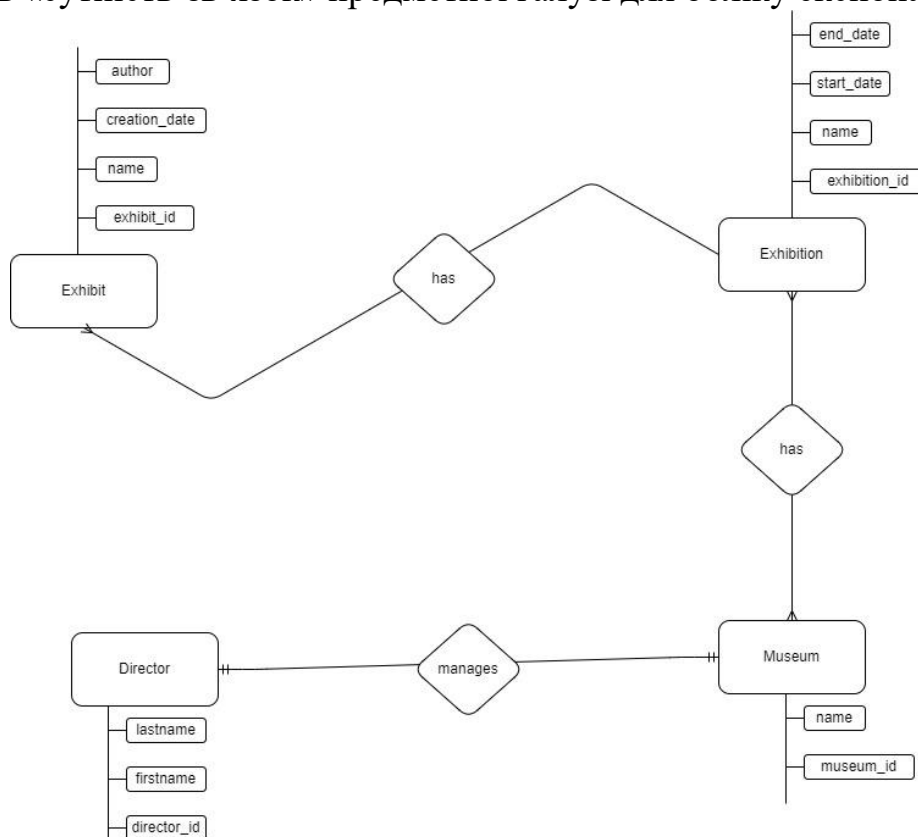


Рисунок 1. ER-діаграма побудована за нотацією «Crow's foot»

У цій моделі кожна з сутностей має свої атрибути, а зв'язки між ними подані наступним чином:

- Зв'язок "Director-Museum" (1:1):
Музей може мати лише одного директора, а директор може управляти лише одним музеєм. Отже це зв'язок «1:1» між **Музеєм** та **Директором**.
- Зв'язок "Museum-Exhibition" (M:N):
Між **Музеєм** і **Виставкою** може існувати зв'язок "багато до багатьох" (M:N), оскільки виставка може проводитись у багатьох музеях, а у музеї може проводитись багато виставок.
- Зв'язок "Exhibition-Exhibit" (1:N):
Між **Виставкою** і **Експонатом** може існувати зв'язок "один до багатьох" (1:N), оскільки експонат може належати тільки до однієї виставки, а на виставці може бути представлено багато експонатів.

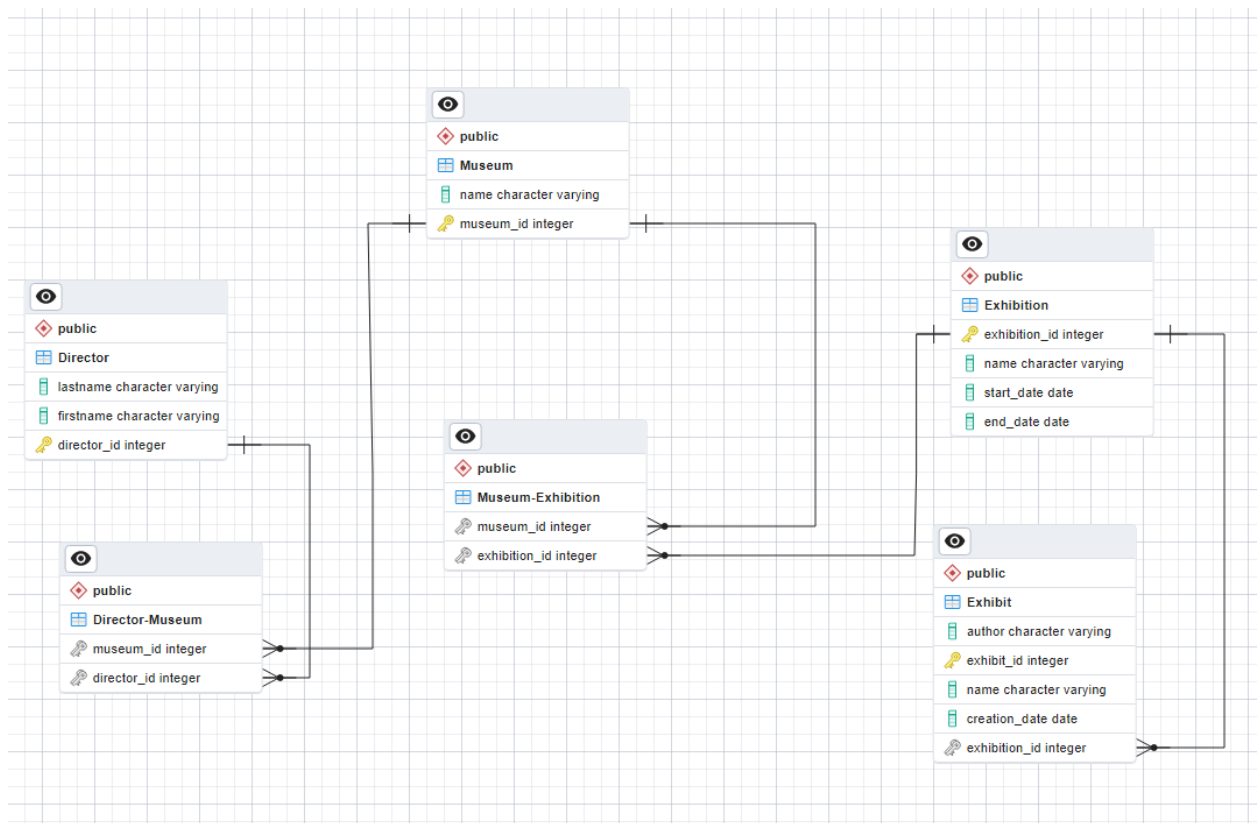


Рисунок. 2 Схема бази даних PostgreSQL на основі ER-моделі для системи обліку експонатів у музеї

Завдання 1

У лабораторній роботі було реалізовано 4 класи відповідно до існуючих таблиць:

1. Exhibit
2. Exhibition
3. Museum
4. Director

Програмна реалізація класів:

```

class Exhibit(Entity, Base):
    __tablename__ = 'Exhibit'
    exhibit_id = Column(Integer, primary_key=True)
    exhibition_id = Column(Integer)
    name = Column(String)
    author = Column(String)
    creation_date = Column(Date)

class Exhibition(Entity, Base):
    __tablename__ = 'Exhibition'
    exhibition_id = Column(Integer, primary_key=True)
    name = Column(String)
    start_date = Column(Date)
    end_date = Column(Date)
  
```

```

class Museum(Entity, Base):
    __tablename__ = 'Museum'
    museum_id = Column(Integer, primary_key=True)
    name = Column(String)

class Director(Entity, Base):
    __tablename__ = 'Director'
    director_id = Column(Integer, primary_key=True)
    firstname = Column(String)
    lastname = Column(String)

```

Та модифіковано клас Entity.

Програмна реалізація класу Entity:

```

class Entity:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    @classmethod
    def get_input_data(cls, isUpdate):
        input_data = {}
        for prop, prop_type in cls.__init__.__annotations__.items():
            if not (isUpdate and ('_id' in prop)):
                value = cls.get_valid_input(f"Enter {prop} ({prop_type.__name__}): ", prop_type)
                input_data[prop] = value
        return input_data

    @classmethod
    def print_properties(self):
        for i, prop in self.__init__.__annotations__.items():
            Printer.print_text(f"{i}. {prop}")

    @classmethod
    def getName(cls):
        return cls.__name__

    @staticmethod
    def get_valid_input(prompt: str, data_type: type):
        while True:
            try:
                if data_type == date:
                    user_input = date.fromisoformat(input(prompt))
                    user_input = user_input.strftime('%Y-%m-%d')
                else:
                    user_input = data_type(input(prompt))

                return user_input
            except ValueError:
                Printer.print_error(f"Invalid input. Please enter a valid {data_type.__name__}.")
        def __repr__(self):
            props = ', '.join([f"{key}={getattr(self, key)}" for key in self.__dict__])
            return f"<{self.__class__.__name__}({props})>"

```

```
PS C:\Users\denis\OneDrive\Робочий стол\5 SEM\БД\rgr> python main.py
Navigation Menu
 1. Show One Table
 2. Insert Data
 3. Delete Data
 4. Update Data
 5. Select Data
 6. Randomize Data
 7. Exit
Enter your choice: █
```

Рисунок. 3 Головне меню програми

Головне меню для користувача складається з семи пунктів.

Приклади запитів у вигляді ORM

Введення даних (2. Insert data) дозволяє користувачеві додавати нові записи в обрану таблицю. Користувач вибирає таблицю і вводить дані для кожного атрибуту.

Реалізація запиту вставки у вигляді ORM

```
def insert_data(self, table_name, data):
    try:
        if table_name == "Exhibit":
            exhibit = Exhibit(**data)
            self.session.add(exhibit)
        elif table_name == "Exhibition":
            exhibition = Exhibition(**data)
            self.session.add(exhibition)
        elif table_name == "Museum":
            museum = Museum(**data)
            self.session.add(museum)
        elif table_name == "Director":
            director = Director(**data)
            self.session.add(director)

        self.session.commit()
        Printer.print_success("Successfully added to {}
table".format(table_name))
    except Exception as e:
        self.session.rollback()
        Printer.print_error(f"Error: {e}")
    finally:
        self.session.close()
```

```

Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: 2
1. Exhibit
2. Exhibition
3. Museum
4. Director
Enter table index: 3
Enter museum_id (int): 2334
Enter name (str): museum
:) Successfully added to Museum table
Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: █

```

Видалення даних (3. Delete data) дозволяє користувачеві вилучати існуючі записи з обраної таблиці. Користувач обирає таблицю, потім вводить ідентифікатор рядка для видалення.

Реалізація запиту видалення у вигляді ORM

```

def delete_data(self, table_name, id):
    try:
        item_deleted = False

        if table_name == "Exhibit":
            exhibit =
self.session.query(Exhibit).filter_by(exhibit_id=id).first()
            if exhibit:
                self.session.delete(exhibit)
                item_deleted = True

        elif table_name == "Exhibition":
            exhibition =
self.session.query(Exhibition).filter_by(exhibition_id=id).first()
            if exhibition:
                self.session.delete(exhibition)
                item_deleted = True

        elif table_name == "Museum":
            museum =
self.session.query(Museum).filter_by(museum_id=id).first()
            if museum:
                self.session.delete(museum)
                item_deleted = True

        elif table_name == "Director":

```



```

        director =
self.session.query(Director).filter_by(director_id=id).first()
        if director:
            self.session.delete(director)
            item_deleted = True

        if item_deleted:
            self.session.commit()
            Printer.print_success(f"Removed successfully from
{table_name} element with id: {id}", 5)
        else:
            Printer.print_error(f"No {table_name} with
{table_name.lower()}_id {id} found.")

    except Exception as e:
        self.session.rollback()
        Printer.print_error(f"Error: {e}", 5)

    finally:
        self.session.close()

```

```

Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: 3
1. Exhibit
2. Exhibition
3. Museum
4. Director
Enter table index: 3
Enter id: 2334
:) Removed successfully from Museum element with id: 2334
Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: █

```

Редагування даних (4. Update data) дозволяє користувачеві змінювати існуючі дані в обраній таблиці. Користувач обирає таблицю, потім вказує, які саме дані редагувати та вводить нові значення.

Реалізація запиту редагування у вигляді ORM

```

def update_data(self, table_name, id, new_data):
    try:
        item_updated = False

        if table_name == "Exhibit":
            exhibit =
self.session.query(Exhibit).filter_by(exhibit_id=id).first()
            if exhibit:
                for key, value in new_data.items():
                    setattr(exhibit, key, value)
                item_updated = True

            elif table_name == "Exhibition":
                exhibition =
self.session.query(Exhibition).filter_by(exhibition_id=id).first()
                if exhibition:
                    for key, value in new_data.items():
                        setattr(exhibition, key, value)
                    item_updated = True

            elif table_name == "Museum":
                museum =
self.session.query(Museum).filter_by(museum_id=id).first()
                if museum:
                    for key, value in new_data.items():
                        setattr(museum, key, value)
                    item_updated = True

            elif table_name == "Director":
                director =
self.session.query(Director).filter_by(director_id=id).first()
                if director:
                    for key, value in new_data.items():
                        setattr(director, key, value)
                    item_updated = True

        if item_updated:
            self.session.commit()
            Printer.print_success(f"Successfully updated {table_name}
table", 5)
        else:
            Printer.print_error(f"No {table_name} with
{table_name.lower()}_id {id} found.")

    except Exception as e:
        self.session.rollback()
        Printer.print_error(f"Error: {e}", 5)

    finally:
        self.session.close()

```

```

Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: 4
1. Exhibit
2. Exhibition
3. Museum
4. Director
Enter table index: 1
Enter id: 2
Enter name (str): updated
Enter author (str): upd
Enter creation_date (date): 2012-12-12
:) Successfully updated Exhibit table
Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: █

```

Вибірковий пошук (5. Select data) дозволяє користувачеві шукати дані за атрибутами з декількох таблиць. Користувач обирає запит і вводить параметри пошуку.

Реалізація запиту вибіркового пошуку у вигляді ORM

```

def select_data(self, selected_options):
    option_index = selected_options['option_index']
    data = selected_options['data']

    try:
        if option_index == 1:
            author_name = data['author_name']

            query =
self.session.query(Exhibit.name.label('exhibit_name'),
Exhibition.name.label('exhibition_name'),
                        Exhibit.author)\
                        .join(Exhibition, Exhibition.exhibition_id
== Exhibit.exhibition_id)\
                        .filter(Exhibit.author == author_name)

            begin = int(time.time() * 1000)
            result_objects = query.all()
            end = int(time.time() * 1000) - begin

            Printer.print_info(f"Request took {end} ms")

            Printer.print_text('fetched {}'.format(result_objects))

        elif option_index == 2:
            exhibition_id = data['exhibition_id']

```

```

        query = self.session.query(Exhibit.author)\
                                .join(Exhibition, Exhibition.exhibition_id
== Exhibit.exhibition_id)\
                                .filter(Exhibition.exhibition_id ==
exhibition_id)\
                                .group_by(Exhibit.author)

        begin = int(time.time() * 1000)
        result_objects = query.all()
        end = int(time.time() * 1000) - begin

        Printer.print_info(f"Request took {end} ms")

        result_objects = [{'author_name': author_name} for
(author_name,) in result_objects]

        Printer.print_text('fetched authors:
{}'.format(result_objects))

    except Exception as e:
        Printer.print_error(f"Error: {e}", 5)

    finally:
        self.session.close()

```

```

Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: 5
1. Show Exhibit by author and Exhibitions they represented at
2. Show list of authors that are represented at specific Exhibition
Enter row index (1-2): 1
Enter Author name: upd
# Request took 0 ms
fetched [('updated', 'ZWZ', 'upd')]
Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: █

```

Випадкові дані (6. Randomize data) дозволяє користувачеві додавати випадкові дані до таблиць Exhibit та Exhibition.

Реалізація запиту генерування випадкових даних у вигляді ORM

```

def randomize_data(self, count):
    try:
        current_exhibition_id =
self.session.query(func.coalesce(func.max(Exhibition.exhibition_id), 0) +
1).scalar()

```

```

        for _ in range(count):
            exhibition = Exhibition(
                exhibition_id=current_exhibition_id,
                name=''.join(chr(math.trunc(65 + random.random() * 26))
                for _ in range(3))),
                start_date=(datetime.now() +
timedelta(days=random.uniform(0, 365))).date(),
                end_date=(datetime.now() +
timedelta(days=(random.uniform(0, 365) + 30))).date()
                )
            current_exhibition_id += 1
            self.session.add(exhibition)

        self.session.commit()

        current_exhibit_id =
self.session.query(func.coalesce(func.max(Exhibit.exhibit_id), 0) +
1).scalar()

        for _ in range(count):
            exhibit = Exhibit(
                exhibit_id=current_exhibit_id,
                author=''.join(chr(math.trunc(65 + random.random() * 26))
                for _ in range(3))),
                creation_date=(datetime.now() -
timedelta(days=random.uniform(0, 365))).date(),
                name=''.join(chr(math.trunc(65 + random.random() * 26))
                for _ in range(3))),
            exhibition_id=self.session.query(Exhibition.exhibition_id).order_by(func.rand
om()).first()[0]
            )
            current_exhibit_id += 1
            self.session.add(exhibit)

        self.session.commit()

        Printer.print_success(f"Successfully randomized {count} records
for Exhibition and Exhibit tables", 5)

    except Exception as e:
        self.session.rollback()
        Printer.print_error(f"Error: {e}", 5)

    finally:
        self.session.close()

```

```

Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit
Enter your choice: 6
Enter how many rows do you want to add: 10
:) Successfully randomized 10 records for Exhibition and Exhibit tables
Navigation Menu
1. Show One Table
2. Insert Data
3. Delete Data
4. Update Data
5. Select Data
6. Randomize Data
7. Exit

```

Завдання 2

Індекс – це спеціальна структура даних, яка утримує в собі зв'язок між ключовими значеннями та відповідними покажчиками на записи в базі даних.

Використання індексів є важливою стратегією для оптимізації управління даними та прискорення виконання запитів. З метою тестування ефективності індексів були створені відокремлені таблиці у базі даних "test" з 1 000 000 записами.

Hash

Для дослідження індексу була створена таблиця hash_test, яка має дві колонки: «id» та «string»

Запит на створення таблиці «hash_test»

```
CREATE TABLE "hash_test"("id" bigserial PRIMARY KEY, "string" varchar(100));  
INSERT INTO "hash_test"("id", "string")  
SELECT generate_series as "id", md5(random()::text)  
FROM generate_series(1,1000000)
```











Створена таблиця:

	id [PK] bigint	string character varying (100)
1	1	16e9f44fa61b21412c812a1399deae4b
2	2	e724b7364b35d3179300e468174c83...
3	3	13f46cac367a64a065fda3f15a715ef6
4	4	1a8126648d4cb728a4ab2e6e3955fdd2
5	5	6cff743c7c64d1f4eef30957de2e9fa0
6	6	42d3993dfa0306528f501dc21f83ddd3
7	7	de644abba946c5d99956221e423267fc
8	8	2c60f83f1b99e1808ef08816132b461e
9	9	c89ca00856e335e3af3b1b17570a332d
10	10	5b4daf8cbdf89194a31df88e752b7ff4
11	11	50fd83b97db1576ba92846536b6ec982
12	12	b1e234086f9e5acfd639d90e77453989
13	13	5aca4ac519e9b9fe034449c3ca3ed8ec
14	14	dbdf9b2862a4059062e6a400dc6ba834
15	15	8fd0d2ef80331f3464a143fd09fd9a92
16	16	dcaacb0c84470c6007539b097aea7271
17	17	1b416bfe3aaaa2f3957efe0020bde484
18	18	57b73087c7db07af0ee90d06cf416ccb
19	19	5cef060a631088e5d64bf501976da162
20	20	b260927a2eead4b5711e14a95ce3a3c5
21	21	ded75cce991d68e45f956a37f2e0bb44
22	22	04dd92d7c53b58bf568c05b9ee938248
Total rows: 1000 of 1000000		Query complete 00:00:00.506

Тестування на п'яти запитах:

```
SELECT COUNT(*) FROM "hash_test" WHERE "string" =  
'1f237e17667e3f4ec5d3defccb6fe525';  
SELECT AVG("id") FROM "hash_test" WHERE "string" LIKE '1f' OR "string" =  
'1f237e17667e3f4ec5d3defccb6fe525';  
SELECT COUNT(*) FROM "hash_test" WHERE "string" LIKE '5d3';  
SELECT * FROM "hash_test" WHERE "string" = 'fe52';  
EXPLAIN SELECT SUM("id") FROM "hash_test" WHERE "id" % 14 = 0 GROUP BY  
"string" LIKE 'fe52';
```











Результати виконання запитів без індексу «hash»:

1.  Successfully run. Total query runtime: 397 msec. 10 rows affected. 
2.  Successfully run. Total query runtime: 459 msec. 10 rows affected. 
3.  Successfully run. Total query runtime: 395 msec. 10 rows affected. 
4.  Successfully run. Total query runtime: 475 msec. 10 rows affected. 
5.  Successfully run. Total query runtime: 478 msec. 10 rows affected. 

Створемо індекс «hash»:

```
CREATE INDEX "hash_index" ON "hash_test" USING hash("string");
```

Результати виконання з індексом «hash»:

1.  Successfully run. Total query runtime: 88 msec. 10 rows affected. 
2.  Successfully run. Total query runtime: 83 msec. 10 rows affected. 
3.  Successfully run. Total query runtime: 38 msec. 10 rows affected. 
4.  Successfully run. Total query runtime: 70 msec. 10 rows affected. 
5.  Successfully run. Total query runtime: 74 msec. 10 rows affected. 

Використання індексу "HASH" відзначилося суттєвим покращенням швидкодії виконання запитів. Без індексу середні значення часу запитів становили 350-450 мс, тоді як із застосуванням хеш-індексу ці значення

зменшилися до 30-80 мс. Такий великий приріст ефективності пояснюється тим, що хеш-індекс прискорює пошук за ключовими значеннями, забезпечуючи більш ефективний доступ до даних.

Індекс hash має свої певні особливості від яких залежить час виконання конкретного запиту:

- не можна використовувати дані в індексі, щоб уникнути читання рядків;
- не можна використовувати для сортування, оскільки рядки у ньому не зберігаються у відсортованому порядку;
- hash-індекси не підтримують пошук за частковим ключем, так як hash-коди обчислюються для всього значення, що індексується;
- hash-індекси підтримують лише порівняння на рівність, що використовують оператори `=`, `IN()` та `<=>`;
- доступ до даних у хеш-індекс дуже швидкий, якщо немає великої кількості колізій;
- деякі операції обслуговування індексу можуть виявитися повільними, якщо кількість колізій велика;

BRIN

BRIN (Block Range Index) — це аббревіатура, що означає індекс діапазону блоків. Індекс BRIN спроектований для ефективної обробки великих таблиць, в яких певні стовпці мають природну кореляцію з фізичним розташуванням даних у таблиці. Робота індексів BRIN ґрунтується на ідеї сканування по бітовій карті. При відповіді на запити, індекс BRIN використовує звичайне сканування, повертаючи всі кортежі всіх сторінок у кожному діапазоні, якщо зведена інформація, збережена в індексі, відповідає умовам запиту. Хоча індекс BRIN може бути неточним, йому властивий невеликий розмір, що робить його ефективним для сканування порівняно невеликої кількості даних. Виконавець запиту відповідає за перевірку кортежів та відкидання тих, що не відповідають умовам запиту. Основні

переваги індексу BRIN полягають у тому, що він мінімізує накладні витрати порівняно з послідовним скануванням, а також уникає сканування великих областей таблиці, де гарантовано немає відповідних кортежів. Типи даних з лінійним порядком сортування можуть використовувати індекс BRIN для зберігання мінімального та максимального значення у кожному діапазоні блоків. У випадку геометричних типів дані можуть зберігати рамку для всіх об'єктів у діапазоні блоків. Загалом, використання індексу BRIN варто розглядати для оптимізації роботи з великими таблицями, особливо там, де існує природна кореляція між значеннями стовпців та їх фізичним розташуванням.

Для дослідження індексу була створена таблиця `brin_test`, яка має дві колонки: «`id`» та «`string`»

Запит на створення таблиці «`brin_test`»

```
CREATE TABLE "brin_test"("id" bigserial PRIMARY KEY, "string" varchar(100));
INSERT INTO "brin_test"("id", "string")
SELECT generate_series as "id", md5(random()::text)
FROM generate_series(1,1000000)
```

	id [PK] bigint	string character varying (100)
1	1	8aa1d4d6eb6af55b6a5dda6aa7f81c...
2	2	7f400ab55833caf2d0504ec624e7e5...
3	3	8e86bd65038ec16c22fc25fdd9ea79...
4	4	d14fd5239a58dd091a5615b36c290...
5	5	70eec02a4476ec5b1b0335042ca4d...
6	6	081afead7fbd863354bc24cc64ec23...
7	7	ab43aa34daa6ee3cf25f4422d5bc16...
8	8	98e3229580ba349f07a1b2005bda0...
9	9	070a2fb1ae0b05e89f1ee648a14472...
10	10	b064f484d746c8cab2b2f9abd2c213...
11	11	c87e55b9436ef3f375208eeb49292c9f
12	12	09a15ea333f2f21f995d5434458950...
13	13	6731e373b064dc9ff3367351c4a8b5...
14	14	66ec19bec4fe321dc99004d3ff764ec2
15	15	66224a0909726c8b0fd1f2f3aa17330
16	16	ed9f4d92136fd4540a25e2961e923c...
17	17	166f4c3431d9dcdc09d06d9213957f...
18	18	25d87cde4c69118af00e1485b18880...
19	19	39e2e68ade21c956d5b2946baa3c1...
20	20	7564cf00a04a3a650a2c68bc25c59e...
21	21	3c855180ddc46d7c26c886d8606af1...
22	22	944988057bb72cc201ef75e2299835...
Total rows: 1000 of 1000000		Query complete 00:00:00.565

Тестування на п'яти запитах:

```
SELECT COUNT(*) FROM "brin_test" WHERE "string" =  
'6731e373b064dc9ff3367351c4a8b5ec';  
SELECT AVG("id") FROM "brin_test" WHERE "string" LIKE 'lf' OR "string" =  
'6731e373b064dc9ff3367351c4a8b5ec';  
SELECT COUNT(*) FROM "brin_test" WHERE "string" LIKE '9ff3';  
SELECT * FROM "brin_test" WHERE "string" = '4a8b';  
EXPLAIN SELECT SUM("id") FROM "brin_test" WHERE "id" % 6 = 0 GROUP BY  
"string" LIKE 'fe52';
```

Результати виконання запитів без індексу «brin»:

1. ✓ Successfully run. Total query runtime: 577 msec. 10 rows affected. ✕
2. ✓ Successfully run. Total query runtime: 474 msec. 10 rows affected. ✕
3. ✓ Successfully run. Total query runtime: 633 msec. 10 rows affected. ✕
4. ✓ Successfully run. Total query runtime: 583 msec. 10 rows affected. ✕
5. ✓ Successfully run. Total query runtime: 483 msec. 10 rows affected. ✕

Створемо індекс «brin»:

```
CREATE INDEX "brin_index" ON "brin_test" USING brin("string");
```

Результати виконання з індексом «brin»:

1. ✓ Successfully run. Total query runtime: 444 msec. 10 rows affected. ✕
2. ✓ Successfully run. Total query runtime: 385 msec. 10 rows affected. ✕
3. ✓ Successfully run. Total query runtime: 564 msec. 10 rows affected. ✕
4. ✓ Successfully run. Total query runtime: 544 msec. 10 rows affected. ✕
5. ✓ Successfully run. Total query runtime: 478 msec. 10 rows affected. ✕

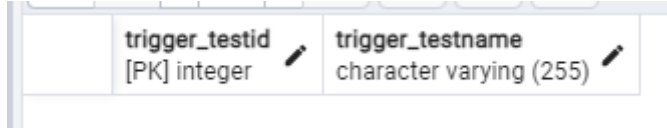
Із цих результатів можна побачити, що запити з використанням індексу «brin» виконуються в середньому на 50-70мс швидше ніж без нього.

Завдання 3

Спершу, створимо таблиці та тригери за варіантом - *after delete, insert*.

Створення таблиць:

```
CREATE TABLE trigger_test (  
    trigger_testID SERIAL PRIMARY KEY,  
    trigger_testName VARCHAR(255)  
);
```



trigger_testid	trigger_testname
[PK] integer	character varying (255)

Створення тригерів:

```
CREATE OR REPLACE FUNCTION trigger_after_insert_function()  
RETURNS TRIGGER AS $$  
DECLARE  
    rec record;  
BEGIN  
    RAISE NOTICE 'Trigger called after insertion';  
  
    IF NEW.trigger_testID % 2 = 0 THEN  
        RAISE NOTICE 'Identifier is even';  
    ELSE  
        RAISE NOTICE 'Identifier is not even';  
    END IF;  
  
    FOR rec IN SELECT * FROM trigger_test  
    LOOP  
        RAISE NOTICE 'Processing record: %', rec.trigger_testName;  
  
        BEGIN  
            RAISE EXCEPTION 'Example of an exception situation';  
        EXCEPTION  
            WHEN others THEN  
                RAISE NOTICE 'An exception situation occurred: %', SQLERRM;  
        END;  
    END LOOP;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trigger_after_insert  
AFTER INSERT ON trigger_test  
FOR EACH ROW EXECUTE FUNCTION trigger_after_insert_function();  
  
CREATE OR REPLACE FUNCTION trigger_after_delete_function()  
RETURNS TRIGGER AS $$  
BEGIN  
    RAISE NOTICE 'Trigger called after deletion';  
  
    IF OLD.trigger_testID % 2 = 0 THEN  
        RAISE NOTICE 'Identifier is even';  
    ELSE  
        RAISE NOTICE 'Identifier is not even';  
    END IF;  
  
    RETURN OLD;
```

```

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_after_delete
AFTER DELETE ON trigger_test
FOR EACH ROW EXECUTE FUNCTION trigger_after_delete_function();

```

52

Data Output	Messages	Notifications
-------------	----------	---------------

CREATE TRIGGER

Query returned successfully in 35 msec.

Додаємо елемент в таблицю «trigger_test»:

```

INSERT INTO trigger_test (trigger_testID, trigger_testName) VALUES (1,
'TestRecord');

```

Отримуємо результат виконання триггеру «insert»:

Data Output	Messages	Notifications
-------------	----------	---------------

ЗАМЕЧАНИЕ: Trigger called after insertion

ЗАМЕЧАНИЕ: Identifier is not even

ЗАМЕЧАНИЕ: Processing record: TestRecord

ЗАМЕЧАНИЕ: An exception situation occurred: Example of an exception situation

INSERT 0 1

Query returned successfully in 36 msec.

Додаємо ще один елемент в таблицю «trigger_test»:

```
INSERT INTO trigger_test (trigger_testID, trigger_testName) VALUES (2,  
'TestRecord 2');
```

Отримуємо результат виконання триггеру «insert»:

Data Output	Messages	Notifications
ЗАМЕЧАНИЕ: Trigger called after insertion		
ЗАМЕЧАНИЕ: Identifier is even		
ЗАМЕЧАНИЕ: Processing record: TestRecord		
ЗАМЕЧАНИЕ: An exception situation occurred: Example of an exception situation		
ЗАМЕЧАНИЕ: Processing record: TestRecord 2		
ЗАМЕЧАНИЕ: An exception situation occurred: Example of an exception situation		
INSERT 0 1		
Query returned successfully in 35 msec.		

Видаємо елемент в таблицю «trigger_test»:

```
DELETE FROM trigger_test WHERE trigger_testID = 1;
```

Отримуємо результат виконання триггеру «after delete»:

Data Output	Messages	Notifications
ЗАМЕЧАНИЕ: Trigger called after deletion		
ЗАМЕЧАНИЕ: Identifier is not even		
DELETE 1		
Query returned successfully in 39 msec.		