



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих
комп'ютерних систем**

Лабораторна робота №3

з дисципліни Баз даних і засоби управління

на тему: “Засоби оптимізації роботи СУБД PostgreSQL”

Виконав:

студент III курсу

групи КВ-94

Мартинюк А. О.

Перевірив:

Петрашенко А. В.

Київ – 2021

Постановка задачі

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

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 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 15

<i>№ варіанта</i>	<i>Види індексів</i>	<i>Умови для тригера</i>
<i>15</i>	<i>Hash, BRIN</i>	<i>before delete, update</i>

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом:

<https://github.com/amartinuk30/databases/tree/main/lab3>

Відомості про обрану предметну галузь з лабораторної роботи №1

Обрана галузь передбачає облік пропозицій щодо трансферу гравців футбольного(або іншого) клубу.

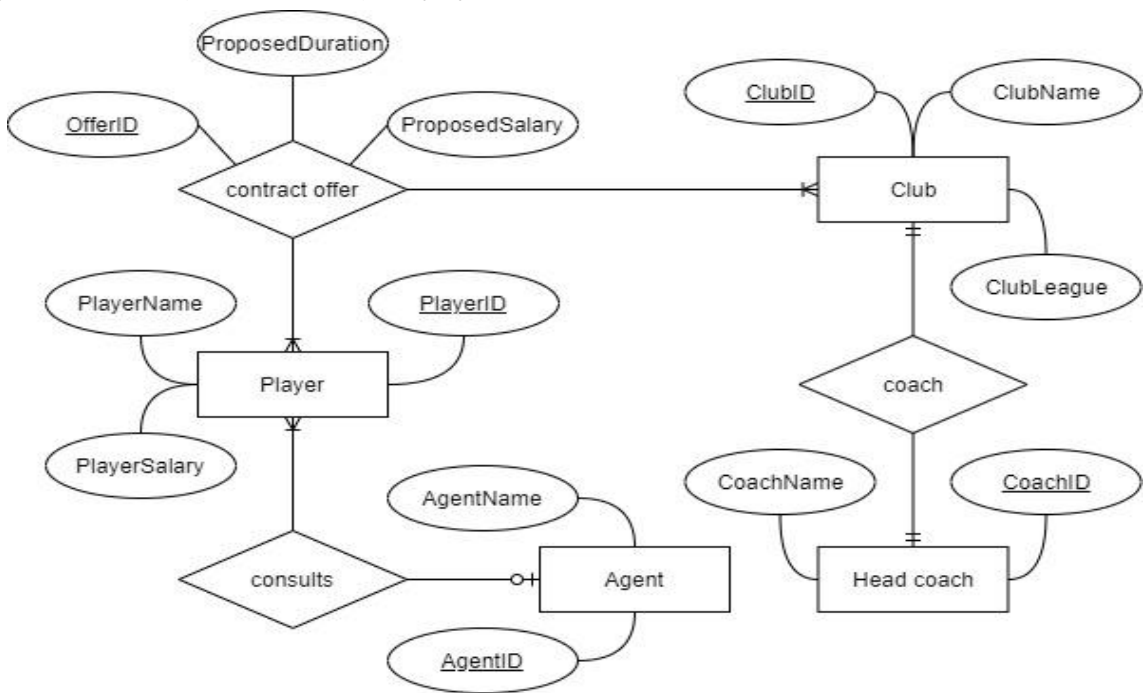


Рис.1 - ER-діаграма, побудована за нотацією Чена

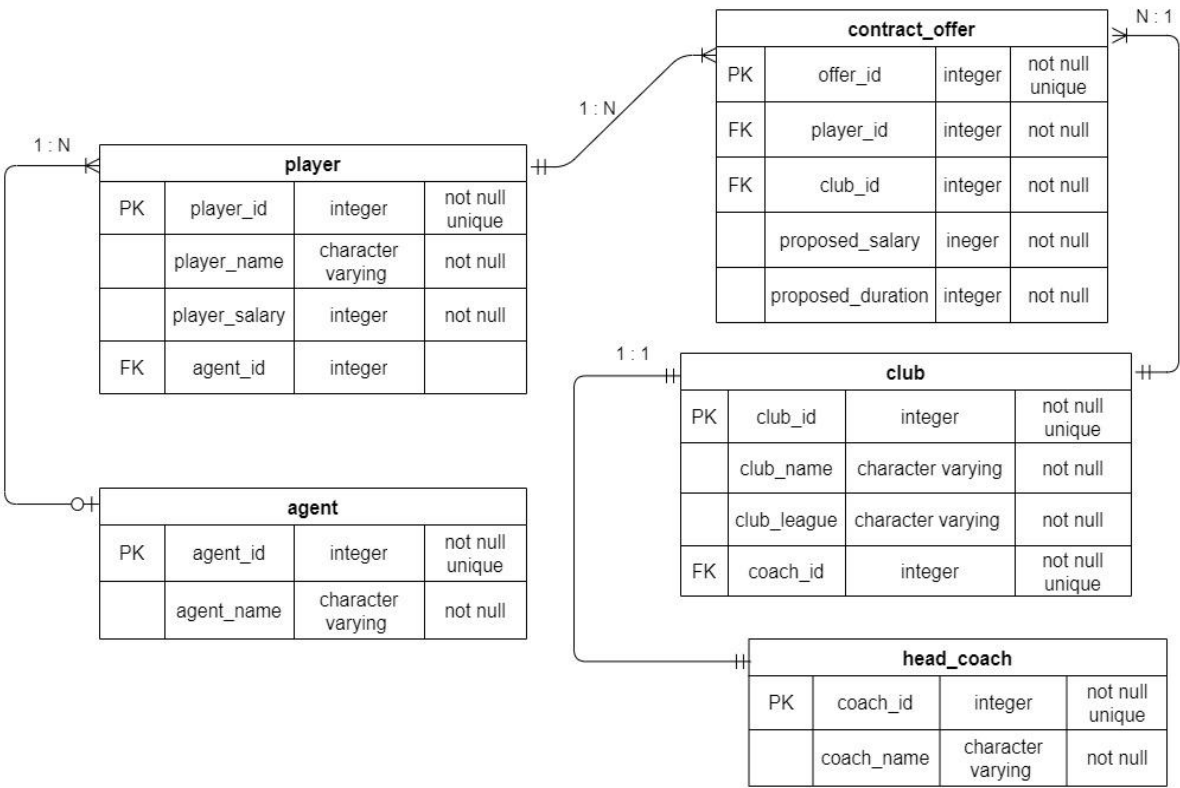


Рис. 2 - Схема бази даних

Таблиця 1. Опис структури БД

Сутність/Зв'язок	Атрибути	Тип атрибутів
Agent - містить інформацію про агента, який співпрацює із гравцем	agent_id - ідентифікатор агента; agent_name - ім'я та прізвище агента.	integer (числовий) char varying (рядок)
Player - містить інформацію про гравця, по якому була зроблена пропозиція	player_id - ідентифікатор гравця player_name - ім'я та прізвище гравця player_salary - актуальна зарплата гравця agent_id - особистий ідентифікатор агента, який співпрацює з гравцем (FK)	integer (числовий) char varying (рядок) integer (числовий) integer (числовий)
Contract offer - містить інформацію про деталі пропонованого контракту	offer_id - ідентифікатор пропозиції player_id - унікальний ідентифікатор гравця, якому зроблена ця пропозиція (FK); club_id - унікальний ідентифікатор клубу, який робить трансферну пропозицію (FK); proposed_salary - запропонована зарплата; proposed_duration - запропонований термін дії контракту.	integer (числовий) integer (числовий) integer (числовий) integer (числовий) integer (числовий)
Club - містить інформацію про клуб, який робить трансферну пропозицію	club_id - ідентифікатор клубу; club_name - назва команди; club_league - ліга, у якій виступає команда; coach_id - ідентифікатор головного тренера, який тренує клуб (FK).	integer (числовий) char varying (рядок) char varying (рядок) integer (числовий)
Head Coach - містить інформацію про головного тренера клубу, який робить пропозицію;	coach_id - ідентифікатор головного тренера; coach_name - ім'я та прізвище головного тренера.	integer (числовий) char varying (рядок)

В обраній предметній галузі маємо обробку трансферних пропозицій діючим гравцям футбольної команди в конкретне трансферне вікно (період, коли можливе заключення таких угод). Оскільки дані стосуються конкретної події та конкретного часового періоду, то актуальність даних забезпечується на той момент, у який була зроблена трансферна пропозиція.

Для побудови бази даних було використано наступні сутності :

1. **Player** з атрибутами PlayerID, PlayerName, PlayerSalary (актуальна зарплата гравця). Ця сутність відповідає за ідентифікацію гравця, по якому була отримана пропозиція трансферу.
2. **Club** з атрибутами ClubID, ClubName, ClubLeague. Ця сутність відповідає за ідентифікацію клубу, який зробив пропозицію щодо трансферу.
3. **Head coach** з атрибутами CoachID, CoachName. Дана сутність визначає головного тренера, який тренує клуб, що пропонує трансфер.
4. **Agent** з атрибутами AgentID, AgentName. Дана сутність визначає агента, який співпрацює із гравцем/гравцями.

Завдання №1

Класи ORM у реалізованому модулі Model

```
class Agent(Model.Base):
    __tablename__ = 'agent'
    agent_id = Column(Integer, primary_key=True)
    agent_name = Column(String(32))

    def __init__(self, agent_id, agent_name):
        self.agent_id = agent_id
        self.agent_name = agent_name
```

```
class Player(Model.Base):
    __tablename__ = 'player'
    player_id = Column(Integer, primary_key=True)
    player_name = Column(String(32))
    player_salary = Column(Integer)
    agent_id = Column(Integer, ForeignKey('agent.agent_id'))

    club_s = relationship("Contract_offer", back_populates="player_r")
    agent = relationship("Agent")

    def __init__(self, player_id, player_name, player_salary, agent_id):
        self.player_id = player_id
        self.player_name = player_name
        self.player_salary = player_salary
        self.agent_id = agent_id
```

```
class Contract_offer(Model.Base):
    __tablename__ = 'contract_offer'
    offer_id = Column(Integer, primary_key=True)
    player_id = Column(Integer, ForeignKey('player.player_id'))
    club_id = Column(Integer, ForeignKey('club.club_id'))
    proposed_salary = Column(Integer)
    proposed_duration = Column(Integer)

    club_r = relationship("Club", back_populates="player_s")
    player_r = relationship("Player", back_populates="club_s")

    def __init__(self, offer_id, player_id, club_id, proposed_salary, proposed_duration):
        self.offer_id = offer_id
        self.player_id = player_id
        self.club_id = club_id
        self.proposed_salary = proposed_salary
        self.proposed_duration = proposed_duration
```

```

class Club(Model.Base):
    __tablename__ = 'club'
    club_id = Column(Integer, primary_key=True)
    club_name = Column(String(32))
    club_league = Column(String(32))
    coach_id = Column(Integer, ForeignKey('head_coach.coach_id'))
    player_s = relationship("Contract_offer", back_populates="club_r")

    def __init__(self, club_id, club_name, club_league, coach_id):
        self.club_id = club_id
        self.club_name = club_name
        self.club_league = club_league
        self.coach_id = coach_id

```

```

class Head_coach(Model.Base):
    __tablename__ = 'head_coach'
    coach_id = Column(Integer, primary_key=True)
    coach_name = Column(String(32))

    def __init__(self, coach_id, coach_name):
        self.coach_id = coach_id
        self.coach_name = coach_name

```

Вставка в таблицу (Insert)

Вставка ряда у таблицю club :

```

Main menu :
1.INSERT TO TABLE
2.UPDATE DATA
3.DELETE FROM TABLE
4.SPECIFIC SELECT
5.SHOW TABLE
>? 1

Tables :
1.player
2.contract_offer
3.club
4.head_coach
5.agent
>? 3
club_id: >? 50
club_name: >? Veres
club_league: >? UkraineLeague
coach_id: >? 11
Successfully inserted

```

Вигляд таблиці club після вставки :

club_id	club_name	club_league	coach_id
1	ZoryaLuhansk	UkraineLeague	1
2	Liverpool	PremierLeague	4
3	ManCity	PremierLeague	3
4	Genoa	SeriaA	2
5	Roma	SeriaA	5
6	Bayern	BundesLeague	10
7	Shakhtar	UPL	6
8	RealMadrid	LaLiga	9
9	Barcelona	LaLiga	7
10	Napoli	SeriaA	8
11	a308e8846514b2	a308e8846514b2	1
12	fc55292e72009b	fc55292e72009b	9
13	876c0f1767fbec	876c0f1767fbec	8
14	e0da5b50a65f7c	e0da5b50a65f7c	5
15	809c73be521674	809c73be521674	5
50	Veres	UkraineLeague	11

Редагування таблиці (Update)

Вигляд таблиці Player до редагування :

player_id	player_name	player_salary	agent_id
1	Tsygankov	120000	1
2	Mykolenko	110000	4
3	Shaparenko	135000	None
4	Buyalkiy	130000	2
5	Buschan	90000	1
6	Zabarnyi	83000	None
7	Syrota	60000	None
8	Verbic	100000	3
9	Karavaiev	90000	4
10	Sydorchuk	150000	2
11	Besedin	90000	2
12	3d488bab9c96d0	164698	3
13	a2a8c16b974910	122323	2
14	16faf4d83d9687	150930	3
15	23c9c2b0be510b	194928	4
16	02ce0e19a49dd8	117277	2

Редагування рядка таблиці Player :

```

Main menu :
1.INSERT TO TABLE
2.UPDATE DATA
3.DELETE FROM TABLE
4.SPECIFIC SELECT
5.SHOW TABLE
>? 2

Tables :
1.player
2.contract_offer
3.club
4.head_coach
5.agent
>? 1

Edit line where player_id = :>? 11
player_name: >? Yarmolenko
player_salary: >? 500000
agent_id: >? 1
Successfully updated
```

Вигляд таблиці Player після редагування :

player_id	player_name	player_salary	agent_id
1	Tsygankov	120000	1
2	Mykolenko	110000	4
3	Shaparenko	135000	None
4	Buyalkiy	130000	2
5	Buschan	90000	1
6	Zabarnyi	83000	None
7	Syrotka	60000	None
8	Verbic	100000	3
9	Karavaiev	90000	4
10	Sydorchuk	150000	2
11	Yarmolenko	500000	1
12	3d488bab9c96d0	164698	3
13	a2a8c16b974910	122323	2
14	16faf4d83d9687	150930	3
15	23c9c2b0be510b	194928	4
16	02ce0e19a49dd8	117277	2

Видалення даних з таблиці (Delete)

Вигляд таблиці Contract offer перед видаленням даних :

offer_id	player_id	club_id	proposed_salary	proposed_duration
1	1	6	130000	4
2	3	3	104735	2
3	2	2	101178	1
4	5	5	133424	3
6	9	9	177968	5
7	10	9	183963	5
8	6	6	143239	3
9	7	7	152913	4
10	1	1	86335	1
11	11	7	159012	4
12	16	10	199068	5
13	7	4	127132	2
14	13	9	176312	5
15	10	7	152482	4
16	15	9	187202	5
17	2	2	94954	1
18	15	9	187351	5
19	10	7	153095	4
20	11	7	161430	4

Видалення рядка з таблиці Contract offer за значенням поля id :

```
Main menu :
1.INSERT TO TABLE
2.UPDATE DATA
3.DELETE FROM TABLE
4.SPECIFIC SELECT
5.SHOW TABLE
>? 3

Tables :
1.player
2.contract_offer
3.club
4.head_coach
5.agent
>? 2

Delete line for this offer_id: >? 20
Successfully deleted
```

Вигляд таблиці Contract offer після видалення даних :

offer_id	player_id	club_id	proposed_salary	proposed_duration
1	1	6	130000	4
2	3	3	104735	2
3	2	2	101178	1
4	5	5	133424	3
6	9	9	177968	5
7	10	9	183963	5
8	6	6	143239	3
9	7	7	152913	4
10	1	1	86335	1
11	11	7	159012	4
12	16	10	199068	5
13	7	4	127132	2
14	13	9	176312	5
15	10	7	152482	4
16	15	9	187202	5
17	2	2	94954	1
18	15	9	187351	5
19	10	7	153095	4


Завдання №2

Для тестування індексів було створено окремі таблиці у базі даних на 1000000 записів через середовище pgAdmin 4.

Створення таблиці hash_test для тестування індексу hash :

```
1 DROP TABLE IF EXISTS hash_test;
2 CREATE TABLE hash_test(id bigserial PRIMARY KEY, num integer);
3 INSERT INTO hash_test(num) SELECT random()*999999 FROM generate_series(1, 1000000) as q;
```

Data Output

 No data output. Execute a query to get output.

Messages


ПОВІДОМЛЕННЯ: таблиця "hash_test" не існує, пропускається
INSERT 0 1000000

Query returned successfully in 10 secs 926 msec.

Створення таблиці brin_test для тестування індексу BRIN :

```
1 DROP TABLE IF EXISTS brin_test;
2 CREATE TABLE brin_test(id bigserial PRIMARY KEY, num integer);
3 INSERT INTO brin_test(num) SELECT random()*999999 FROM generate_series(1, 1000000) as q;
```

Data Output

 No data output. Execute a query to get output.

Messages

ПОВІДОМЛЕННЯ: таблиця "brin_test" не існує, пропускається
INSERT @ 10000000

Query returned successfully in 8 secs 420 msec.

Hash

Ідея хешування у тому, щоб значенню будь-якого типу даних зіставити деяке невелике число (від 0 до N-1, всього N значень). Таке зіставлення називають хеш-функцією. Отримане число можна використовувати як індекс звичайного масиву, куди і складати посилання на рядки таблиці (TID). Елементи даного масиву називають кошиками хеш-таблиці - в одному кошику можуть лежати кілька TID-ів, якщо одне і те саме проіндексоване значення зустрічається в різних рядках. Коли відбувається запит за допомогою індексу хешування, PostgreSQL бере значення індексу і застосовує хеш-функцію, щоб визначити, яка комірка може містити потрібні дані та швидко знайти відповідний TID.

При пошуку в індексі ми обчислюємо хеш-функцію для ключа та отримуємо номер корзини. Залишається перебрати весь вміст кошика і повернути тільки відповідні TID з потрібними хеш-кодами. Це робиться ефективно, оскільки пари «хеш-код – TID» зберігаються впорядковано.

Запити без індексування :

```
1 SELECT COUNT(*) FROM "hash_test" WHERE "id" % 66 = 0;
```

Data Output

	count bigint
1	15151

Messages

Successfully run. Total query runtime: 520 msec.
1 rows affected.

```
1 SELECT COUNT(*) FROM "hash_test" WHERE "num" >= '77777';
```

Data Output

	count bigint	
1	922378	

Messages

Successfully run. Total query runtime: 500 msec.
1 rows affected.

```
1 SELECT AVG("id") FROM "hash_test" WHERE "num" >= '77777' AND "id" <= '333333';
```

Data Output

	avg numeric	
1	166707.658097218020	

Messages

Successfully run. Total query runtime: 428 msec.
1 rows affected.

```
1 SELECT MAX("id") FROM "hash_test" WHERE "num" >= '22222' AND "num" <= '33333' GROUP BY "id" % 2;  
2
```

Data Output

	max bigint	
1	999928	
2	999985	

Messages

Successfully run. Total query runtime: 426 msec.
2 rows affected.

Створення хеш-індексу :

```
1 DROP INDEX IF EXISTS "hash_idx";
2 CREATE INDEX "hash_idx" ON "hash_test" USING hash("id");
3
```

Data Output

 No data output. Execute a query to get output.

Messages


ПОВІДОМЛЕННЯ: індекс "hash_idx" не існує, пропускається
CREATE INDEX

Query returned successfully in 19 secs 718 msec.

Запити з індексуванням :

```
1 SELECT COUNT(*) FROM "hash_test" WHERE "id" % 66 = 0;
```

Data Output


	count bigint 
1	15151

Messages

Successfully run. Total query runtime: 291 msec.
1 rows affected.

```
1 SELECT COUNT(*) FROM "hash_test" WHERE "num" >= '77777';
```

Data Output

	count bigint 
1	922378

Messages

Successfully run. Total query runtime: 281 msec.
1 rows affected.

```
1 SELECT AVG("id") FROM "hash_test" WHERE "num" >= '77777' AND "id" <= '333333';
```

Data Output

	avg numeric	
1	166707.658097218020	

Messages

Successfully run. Total query runtime: 279 msec.
1 rows affected.

```
1 SELECT MAX("id") FROM "hash_test" WHERE "num" >= '22222' AND "num" <= '33333' GROUP BY "id" % 2;  
2
```

Data Output

	max bigint	
1	999928	
2	999985	

Messages

Successfully run. Total query runtime: 277 msec.
2 rows affected.

Індексування за допомогою hash пришвидшує пошук даних у таблиці, але іноді результати не вражають і та значно гірші від тих, які можуть дати інші методи індексування. Це можна пояснити тим, що це один із найпримітивніших методів індексування і для пошуку потрібних даних алгоритм все одно проходить через усі записи у таблиці. Його використання доцільне до поля числового типу.

BRIN

Ідея BRIN індексації полягає в створенні діапазону блоків або ж групи сторінок, які прилягають одна до одної, а інформація про них збережена в індексі. Тобто, більший пріоритет в тому, щоб уникнути перегляду непотрібних рядків. Цей спосіб працює добре для тих даних, які уже практично посортовані, так як фізичне місцезнаходження буде корелюватися із значенням стовпчиків. В іншому випадку перевага блоків буде знехтуваною. Цей вид індексу доцільно застосовувати до чисельних даних та дат. Перевагою є порівняно невеликий розмір і мінімальні ресурси для підтримки функціонування.

Алгоритм такий: послідовно проглядається карта блоків. За допомогою покажчиків визначаються індексні рядки зі зведеною інформацією по кожній ділянці. Якщо ділянка точно не містить шуканого значення, вона пропускається; якщо може

містити (або якщо зведена інформація відсутня) - всі сторінки ділянок додаються до бітової карти. Ця бітова карта і використовується в подальшому пошуку. І хоч даний алгоритм може жертвувати ефективністю, однак він буде у виграші через можливість ефективно працювати з даними дуже великих розмірів.

Запити без індексування :

```
1 SELECT COUNT(*) FROM "brin_test" WHERE "id" % 66 = 0;
```

Data Output

	count bigint
1	15151

Messages

Successfully run. Total query runtime: 355 msec.
1 rows affected.

```
1 SELECT COUNT(*) FROM "brin_test" WHERE "num" >= '77777';
```

Data Output

	count bigint
1	922247

Messages

Successfully run. Total query runtime: 300 msec.
1 rows affected.

```
1 SELECT AVG("id") FROM "brin_test" WHERE "num" >= '77777' AND "id" <= '333333';
```

Data Output

	avg numeric
1	166655.397282879341

Messages

Successfully run. Total query runtime: 322 msec.
1 rows affected.


```
1 SELECT MAX("id") FROM "brin_test" WHERE "num" >= '22222' AND "num" <= '33333' GROUP BY "id" % 2;  
2  
3
```

Data Output

	max bigint	
1	999896	
2	999885	


Messages

Successfully run. Total query runtime: 300 msec.
2 rows affected.

Створення індексу BRIN :

```
1 DROP INDEX IF EXISTS "brin_idx";  
2 CREATE INDEX "brin_idx" ON "brin_test" USING brin("num");  
3
```

Data Output

 No data output. Execute a query to get output.

Messages

ПОВІДОМЛЕННЯ: індекс "brin_idx" не існує, пропускається
CREATE INDEX

Query returned successfully in 627 msec.

Запити з індексуванням :

```
1 SELECT COUNT(*) FROM "brin_test" WHERE "id" % 66 = 0;  
2  
3  
4
```

Data Output


	count bigint	
1	15151	

Messages

Successfully run. Total query runtime: 373 msec.
1 rows affected.

```
1 SELECT COUNT(*) FROM "brin_test" WHERE "num" >= '77777';
2
3
```

Data Output


	count bigint 
1	922247

Messages

Successfully run. Total query runtime: 298 msec.
1 rows affected.

```
1 SELECT AVG("id") FROM "brin_test" WHERE "num" >= '77777' AND "id" <= '333333';
```

Data Output

	avg numeric 
1	166655.397282879341

Messages

Successfully run. Total query runtime: 265 msec.
1 rows affected.

```
1 SELECT MAX("id") FROM "brin_test" WHERE "num" >= '22222' AND "num" <= '33333' GROUP BY "id" % 2;
2
```

Data Output

	max bigint 
1	999896
2	999885

Messages

Successfully run. Total query runtime: 262 msec.
2 rows affected.

З отриманих результатів бачимо, що в усіх випадках, де застосовуються дані індексованого стовпця, пошук з індексацією відбувається швидше, ніж пошук без індексації. Це відбувається завдяки головній особливості індексування BRIN: блоки, у яких точно немає шуканого значення, не розглядаються.

Завдання №3

Для тестування тригера створимо тестові таблиці test_1 та test_2 :

```
1 DROP TABLE IF EXISTS "test_1";
2 CREATE TABLE "test_1"(
3     "id" bigserial PRIMARY KEY,
4     "text" text);
5
6 DROP TABLE IF EXISTS "test_2";
7 CREATE TABLE "test_2"(
8     "ID" bigserial PRIMARY KEY,
9     "id_from_test_1" bigint,
10    "test_2_text" text);
```

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

ПОВІДОМЛЕННЯ:	таблиця "test_1" не існує, пропускається		
ПОВІДОМЛЕННЯ:	таблиця "test_2" не існує, пропускається		
CREATE TABLE			

Query returned successfully in 587 msec.

Додаємо 10 рядків у таблицю test_1 :

```
1 INSERT INTO "test_1"("text")
2
3 VALUES ('elem1'),('elem2'), ('elem3'),
4         ('elem4'), ('elem5'), ('elem6'),
5         ('elem7'), ('elem8'), ('elem9'), ('elem10');
```

Текст тригера :

```
1 CREATE OR REPLACE FUNCTION before_delete_update_func()
2     RETURNS TRIGGER AS $$
3 DECLARE
4     CURSOR_2 CURSOR FOR SELECT * FROM "test_2";
5     row_ "test_2"%ROWTYPE;
6
7 BEGIN
8     IF old."id" % 2 = 0 THEN
9         RAISE NOTICE 'id(test_1) is multiple of 2';
10        INSERT INTO "test_2"("id_from_test_1", "test_2_text") VALUES (old."id", 'is_multiple_of_2_text');
11        RETURN OLD;
12    ELSE
13        RAISE NOTICE 'id(test_1) is not multiple of 2';
14        INSERT INTO "test_2"("id_from_test_1", "test_2_text") VALUES (old."id", 'is_not_multiple_of_2_text');
15        RETURN NEW;
16    END IF;
17 END;
18 $$ LANGUAGE plpgsql;
```

Команди, що ініціюють виконання тригера:

```
1 CREATE TRIGGER "before_delete_update_trigger"
2 BEFORE DELETE OR UPDATE ON "test_1"
3 FOR EACH ROW EXECUTE procedure before_delete_update_func();
```

Початковий вид тестових таблиць :

	id [PK] bigint	text text
1	1	elem1
2	2	elem2
3	3	elem3
4	4	elem4
5	5	elem5
6	6	elem6
7	7	elem7
8	8	elem8
9	9	elem9
10	10	elem10

ID [PK] bigint	id_from_test_1 bigint	test_2_text text

Запит на видалення :

```
1 DELETE FROM "test_1" WHERE "id" <= 5;
```

Explain Notifications Data Output Messages

ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
DELETE 2

Вигляд таблиць після видалення :

	id [PK] bigint	text text
1	1	elem1
2	3	elem3
3	5	elem5
4	6	elem6
5	7	elem7
6	8	elem8
7	9	elem9
8	10	elem10

	ID [PK] bigint	id_from_test_1 bigint	test_2_text text
1	1	1	is_not_multiple_of_2_text
2	2	2	is_multiple_of_2_text
3	3	3	is_not_multiple_of_2_text
4	4	4	is_multiple_of_2_text
5	5	5	is_not_multiple_of_2_text

Запит на оновлення :

```
1 UPDATE "test_1" SET "text" = "text" || '_update' WHERE "id" >= 5;
2
```

Explain Notifications Data Output Messages

ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is not multiple of 2
ПОВІДОМЛЕННЯ: id(test_1) is multiple of 2
UPDATE 6

Вигляд таблиць після оновлення :

	id [PK] bigint	text text
1	1	elem1
2	2	elem2
3	3	elem3
4	4	elem4
5	5	elem5_update
6	6	elem6
7	7	elem7_update
8	8	elem8
9	9	elem9_update
10	10	elem10

	ID [PK] bigint	id_from_test_1 bigint	test_2_text text
1	1	5	is_not_multiple_of_2_text
2	2	6	is_multiple_of_2_text
3	3	7	is_not_multiple_of_2_text
4	4	8	is_multiple_of_2_text
5	5	9	is_not_multiple_of_2_text
6	6	10	is_multiple_of_2_text

Завдання №4

Створимо тестову таблицю для моделювання транзакцій :

```
1 DROP TABLE IF EXISTS "transaction_test";
2 CREATE TABLE "transaction_test"("id" bigserial PRIMARY KEY,
3                                     "num" bigint,
4                                     "text" text );
5 INSERT INTO "transaction_test"("num", "text")
6 VALUES (111, 'text_1'), (222, 'text_2'), (333, 'text_3');
```

[Explain](#) [Notifications](#) [Data Output](#) [Messages](#)

ПОВІДОМЛЕННЯ: таблиця "transaction_test" не існує, пропускається
INSERT 0 3

READ COMMITTED

На цьому рівні ізоляції певна транзакція не бачить змін у базі даних, викликаних іншою транзакцією до її завершення.

<pre>postgres=# START TRANSACTION; START TRANSACTION postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE; SET postgres=# UPDATE transaction_test SET num=555; UPDATE 3 postgres=#</pre>	<pre>postgres=# SELECT * FROM transaction_test; id num text ----+----+----- 1 111 text_1 2 222 text_2 3 333 text_3 (3 рядки) postgres=#</pre>
--	---

Бачимо, що до моменту, коли ми не завершили виконання транзакції командою commit, зміни у іншій транзакції не відображаються :

<pre>postgres=# START TRANSACTION; START TRANSACTION postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE; SET postgres=# UPDATE transaction_test SET num=555; UPDATE 3 postgres=# commit; COMMIT postgres=# SELECT * FROM transaction_test; id num text ----+----+----- 1 555 text_1 2 555 text_2 3 555 text_3 (3 рядки)</pre>	<pre>postgres=# SELECT * FROM transaction_test; id num text ----+----+----- 1 111 text_1 2 222 text_2 3 333 text_3 (3 рядки) postgres=# SELECT * FROM transaction_test; id num text ----+----+----- 1 555 text_1 2 555 text_2 3 555 text_3 (3 рядки)</pre>
--	---

Також можемо побачити, що виконання наступної транзакції можливе лише за умови закінчення попередньої :

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# UPDATE transaction_test SET num=123;
UPDATE 3
postgres=#
postgres=# UPDATE transaction_test SET num=345;
```

Лише після закінчення лівої транзакції, запит виконала друга, знову змінивши дані. При чому, дані в першій транзакції зміняться і після власного коміта і потім ще після коміта другої транзакції :

```
postgres=# UPDATE transaction_test SET num=111;
UPDATE 3
postgres=# commit;
COMMIT
postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 111 | text_1
  2 | 111 | text_2
  3 | 111 | text_3
(3 рядки)

postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 222 | text_1
  2 | 222 | text_2
  3 | 222 | text_3
(3 рядки)

postgres=# UPDATE transaction_test SET num=222;
UPDATE 3
postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 222 | text_1
  2 | 222 | text_2
  3 | 222 | text_3
(3 рядки)

postgres=# commit;
COMMIT
postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 222 | text_1
  2 | 222 | text_2
  3 | 222 | text_3
(3 рядки)
```

Коли певна транзакція бачить дані запитів UPDATE, DELETE попередньої транзакції, виникає феномен повторного читання, а для запиту INSERT – читання фантомів. Цей рівень ізоляції може забезпечити захист від явища брудного читання.

REPEATABLE READ

На цьому рівні ізоляції транзакція №2 не побачить зміни транзакції №1, але і не зможе отримати доступ до змінених даних

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# DELETE FROM transaction_test where id = 2;
DELETE 1
postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 555 | text_1
  3 | 555 | text_3
(2 рядки)

postgres=# commit;
COMMIT

postgres=# SELECT * FROM transaction_test;
 id | num | text
----+----+-----
  1 | 555 | text_1
  2 | 555 | text_2
  3 | 555 | text_3
(3 рядки)

postgres=# DELETE FROM transaction_test where id = 2;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне видалення
postgres=# SELECT * FROM transaction_test;
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції не будуть виконані
postgres=# SELECT * FROM transaction_test;
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції не будуть виконані
postgres=# commit;
ROLLBACK
```

При спробі доступу до змінених даних отримуємо помилку. При такій ізоляції, не виникає читання фантомів та повторного читання, а також заборонено одночасний доступ до не збережених даних. Проте, зазвичай цей рівень ізоляції призначений для попередження повторного читання.

SERIALIZABLE

На цьому рівні транзакції не можуть вплинути одна на одну і одночасний доступ строго заборонений.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# DELETE FROM transaction_test where id = 1;
DELETE 1
postgres=# SELECT * FROM transaction_test;
 id | num | text
-----+-----
  3 | 555 | text_3
(1 рядок)

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

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# SELECT * FROM transaction_test;
 id | num | text
-----+-----
  1 | 555 | text_1
  3 | 555 | text_3
(2 рядки)

postgres=# DELETE FROM transaction_test where id = 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне видалення
postgres=# commit;
ROLLBACK
postgres=#
```

На даному рівні ізоляції ми можемо отримати максимальну узгодженість даних і можемо бути впевнені, що зайві дані не будуть зафіксовані.