



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 **Бази даних і засоби управління**
на тему: “Засоби оптимізації роботи СУБД PostgreSQL”

Виконала:
студентка III курсу
групи КВ-94
Іус І. О.
Перевірів:
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.

№ варіанта	Види індексів	Умови для тригера
8	<i>BTree, GIN</i>	<i>After, insert, update</i>

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

<https://github.com/van2ivan/KV-94-Ivan-Ius>

Завдання №1

Обрана предметна галузь передбачає ведення обліку оцінок у навчальному закладі

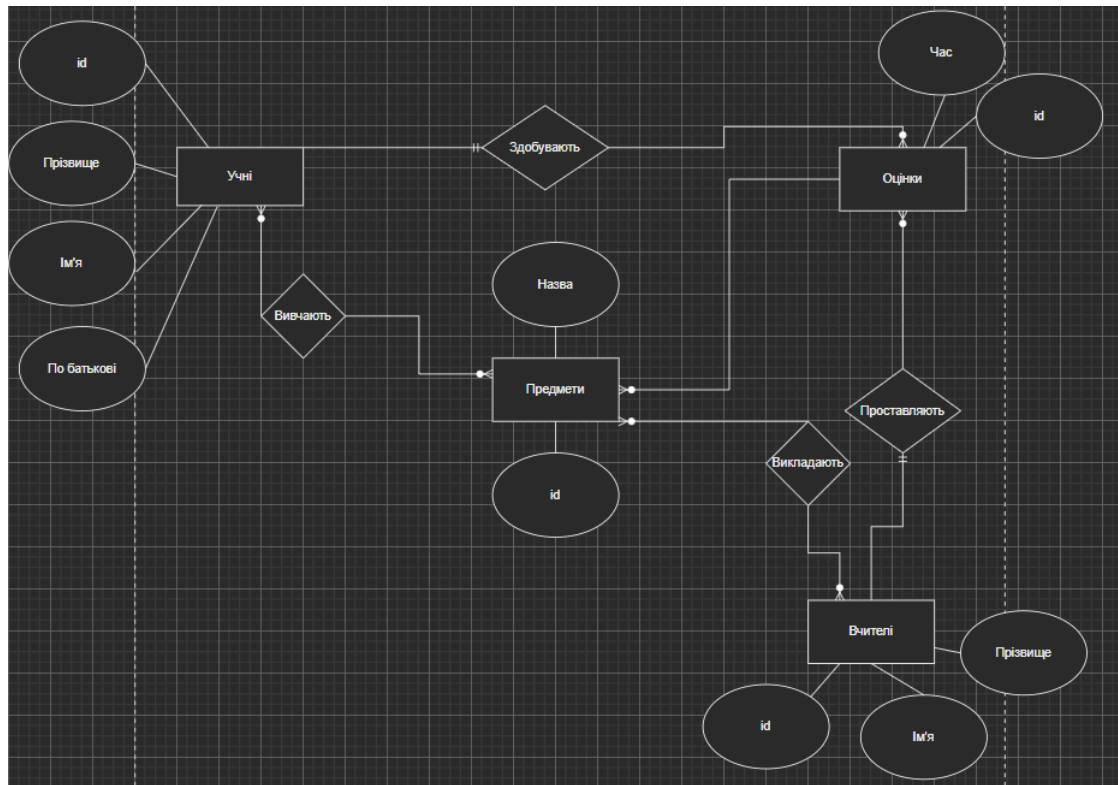


Рисунок 1. ER-діаграма

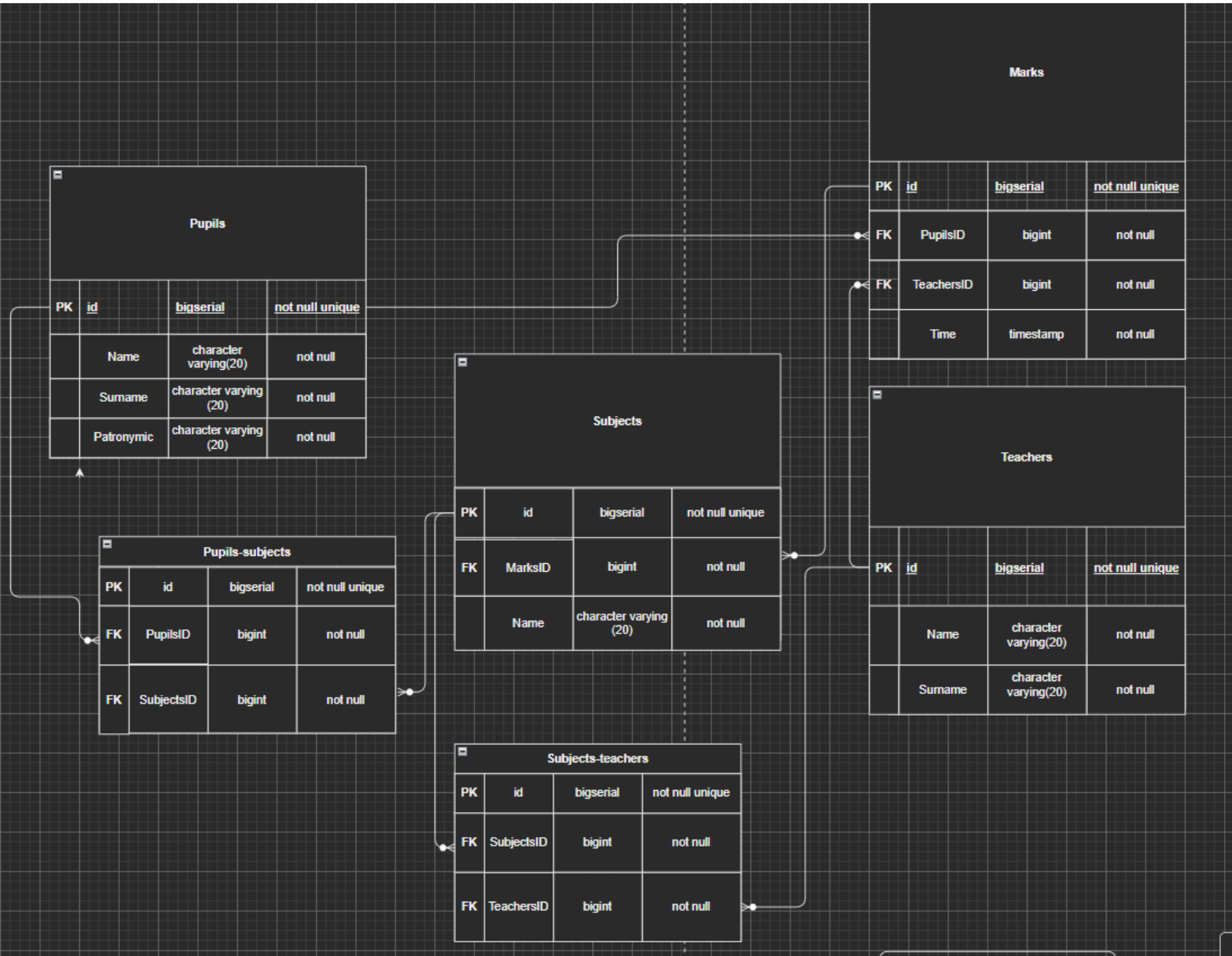


Рисунок 2. Схема бази даних

Класи ORM у реалізованому модулі Model

```
from sqlalchemy import BigInteger, Column, DateTime, Float, ForeignKey, Numeric, String, Table, Text, text
from sqlalchemy.dialects.postgresql import OID
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()
metadata = Base.metadata
```

```
class Pupil(Base):
    __tablename__ = 'Pupils'

    Id = Column(BigInteger, primary_key=True,
server_default=text("nextval('\\"Pupils_Id_seq\\"':regclass)))
    Name = Column(String(20), nullable=False)
    Patronymic = Column(String(20), nullable=False)
    Surname = Column(String(20), nullable=False)
```

```
class Teacher(Base):
    __tablename__ = 'Teachers'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"Teachers_id_seq\\"':regclass)))
    name = Column(String(20), nullable=False)
    surname = Column(String(20), nullable=False)
```

```
class Mark(Base):
    __tablename__ = 'Marks'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"Marks_id_seq\\"':regclass)))
    time = Column(DateTime, nullable=False)
    pupilsid = Column(ForeignKey('Pupils.Id'), nullable=False)
    teachersid = Column(ForeignKey('Teachers.id'), nullable=False)

    Pupil = relationship('Pupil')
    Teacher = relationship('Teacher')
```

```
class Subject(Base):
    __tablename__ = 'Subjects'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"Subjects_id_seq\\"':regclass)))
    name = Column(String(20), nullable=False)
    marksid = Column(ForeignKey('Marks.id'), nullable=False)

    Mark = relationship('Mark')
```

```
class PupilsSubject(Base):
    __tablename__ = 'PupilsSubjects'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"PupilsSubjects_id_seq\\"':regclass)))
    pupilsid = Column(ForeignKey('Pupils.Id'), nullable=False)
    subjectsid = Column(ForeignKey('Subjects.id'), nullable=False)
```

```

Pupil = relationship('Pupil')
Subject = relationship('Subject')

class TeachersSubject(Base):
    __tablename__ = 'TeachersSubjects'

    id = Column(BigInteger, primary_key=True, unique=True,
server_default=text("nextval('\\"TeachersSubjects_id_seq\\"'::regclass)"))
    teachersid = Column(ForeignKey('Teachers.id'), nullable=False)
    subjectsid = Column(ForeignKey('Subjects.id'), nullable=False)

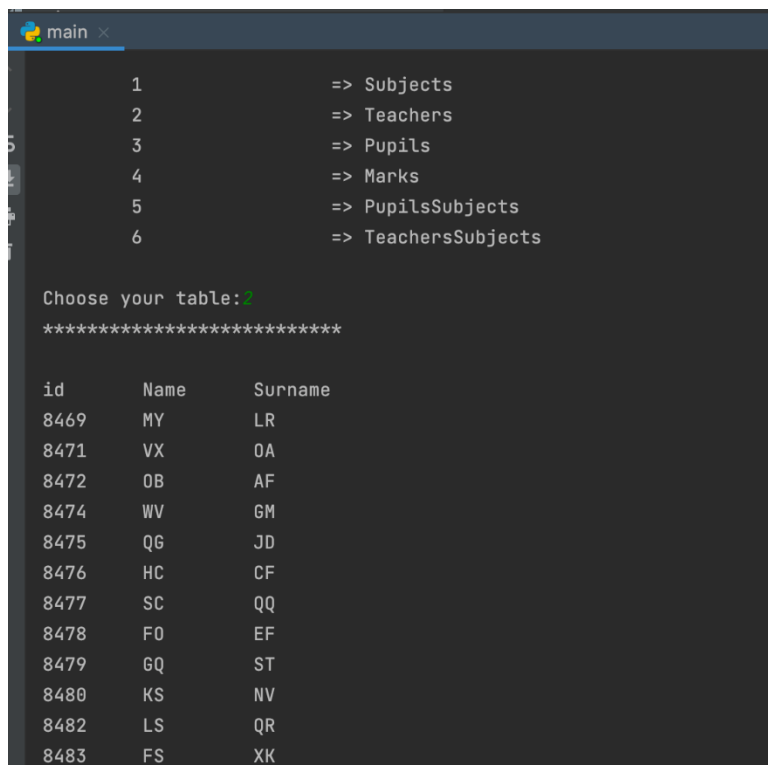
    Subject = relationship('Subject')
    Teacher = relationship('Teacher')

```

Запити у вигляді ORM

Продемонструємо вставку, виучення, редагування даних.

Початковий стан:



```

main x
1      => Subjects
2      => Teachers
3      => Pupils
4      => Marks
5      => PupilsSubjects
6      => TeachersSubjects

Choose your table: 3
*****

id      Name  Surname
8469    MY    LR
8471    VX    OA
8472    OB    AF
8474    WV    GM
8475    QG    JD
8476    HC    CF
8477    SC    QQ
8478    FO    EF
8479    GQ    ST
8480    KS    NV
8482    LS    QR
8483    FS    XK

```

Видалення запису:

```
Choose your table:2
Attribute to delete ID = 8471
'deleted'
1 => Continue delete, 2 => Stop delete => 2
Continue to work with db => 1, stop => 2. Your choice =>1

    1 => One table
    2 => All tables
    3 => Insertion
    4 => Delete some inf
    5 => Updating
    6 => Selection
    7 => Random inf
    ...
    0 => Exit

Your choice is: 1

    1          => Subjects
    2          => Teachers
    3          => Pupils
    4          => Marks
    5          => PupilsSubjects
    6          => TeachersSubjects

Choose your table:2
*****

id      Name      Surname
8469    MY        LR
8472    OB        AF
8474    WV        GM
8475    QG        JD
8476    HC        CF
...
```

Вставка запису:

```
Choose your table: 2
Name = Ins
Surname = Ert
'added'
```

Choose your table:

id	Name	Surname
8469	MY	LR
8472	OB	AF
8474	WV	GM
8475	QG	JD
8476	HC	CF
8477	SC	QQ
8478	FO	EF
8479	GQ	ST
8480	KS	NV
8482	LS	QR
8483	FS	XK
8484	LZ	UR
8485	XA	MO
8486	OO	XG
8487	OX	PM
8489	KP	ZF
8490	QB	MD
8491	YY	JY
8492	RL	WN
8493	Teacher	Teacherovich
8488	Petro	Poroshenko
8494	sql	alchemy
8495	Ins	Ert

Редагування запису:

Запити пошуку та генерації рандомізованих даних також було реалізовано, логіку пошуку було змінено у порівнянні з лабораторною роботою №2 (усі дані для пошуку передвизначено, тепер вони не вводяться з клавіатури). Запити на пошук ті самі, що і л.р. №2.

```
Choose your table:2
Row to update where id = 8469
Name = Ann
Surname = Us
'updated'
1 => Continue update, 2 => Stop update => 2
*****
```

8488	Petro	Poroshenko
8494	sql	alchemy
8495	Ins	Ert
8469	'Ann'	'Us'

Завдання №2

Для тестування індексів було створено окремі таблиці у базі даних з 1000000 записів.

Hash

Хеш-індекси в PostgreSQL використовують форму структури даних хеш-таблиці (використовують хеш-функцію). Хеш-коди поділені на обмежену кількість комірок. Коли до індексу додається нове значення, PostgreSQL застосовує хеш-функцію до значення і поміщає хеш-код і вказівник на кортеж у відповідну комірку. Коли відбувається запит за допомогою індексу хешування, PostgreSQL бере значення індексу і застосовує хеш-функцію, щоб визначити, яка комірка може містити потрібні дані.

Стверення таблиці БД:

```
DROP TABLE IF EXISTS "hash_test";
CREATE TABLE "hash_test"("id" bigserial PRIMARY KEY, "time" timestamp);
INSERT INTO "hash_test"("time") SELECT (timestamp '2021-01-01' + random()) * (timestamp
```

```
'2020-01-01' - timestamp '2022-01-01')) FROM  
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as  
symbols(characters), generate_series(1, 1000000) as q;
```

Запити для тестування:

```
SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;  
SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';  
SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';  
SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;
```

Створення індексу:

```
DROP TABLE IF EXISTS "hash_test";  
CREATE INDEX "time_hash_index" ON "hash_test" USING hash("id");
```

Результати і час виконання на скріншотах з psql

Запити без індексування:

```
Секундомер включён.
postgres=# DROP INDEX IF EXISTS "time_hash_index";
ПОВІДОМЛЕННЯ:  індекс "time_hash_index" не існує, пропускається
DROP INDEX
Время: 2,492 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 111,332 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';
count
-----
626919
(1 строка)

Время: 163,147 мс
postgres=# SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';
avg
-----
500254.786621557171
(1 строка)

Время: 176,904 мс
postgres=# SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;
sum      | max
-----+-----
82418773060 | 999999
82457311990 | 999994
(2 строки)

Время: 120,725 мс
```

Запити з індексуванням:

```
postgres=# CREATE INDEX "time_hash_index" ON "hash_test" USING hash("id");
CREATE INDEX
Время: 3745,561 мс (00:03,746)
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 95,619 мс
postgres=# SELECT COUNT(*) FROM "hash_test" WHERE "time" >= '20191001';
count
-----
626336
(1 строка)

Время: 100,444 мс
postgres=# SELECT AVG("id") FROM "hash_test" WHERE "time" >= '20191001' AND "time" <= '20211207';
avg
-----
500044.170913056251
(1 строка)

Время: 155,144 мс
postgres=# SELECT SUM("id"), MAX("id") FROM "hash_test" WHERE "time" >= '20200505' AND "time" <= '20210505' GROUP BY "id" % 2;
sum      | max
-----+-----
82587011394 | 999994
82348391481 | 999989
(2 строки)

Время: 155,314 мс
```

Очевидно, що індексування за допомогою hash не значно пришвидшує пошук даних у таблиці, а іноді навіть показує гірші результати, ніж запити без ідексування. Це впливає з того, що це один із найпримітивніших методів індексування і для пошуку потрібних даних алгоритм все одно проходить через усі записи у таблиці (на відміну від GIN). Він ефективний при застосуванні до поля числового типу.

GIN

GIN призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями (наприклад - реченнями), а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах (повторювані частини слів або речень). Індекс GIN зберігає набір пар (ключ, список появи ключа), де список появи — це набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з'являється багато разів. Цей індекс може взаємодіяти тільки з полем типу tsvector.

Стверення таблиці БД:

```
DROP TABLE IF EXISTS "gin_test";
CREATE TABLE "gin_test"("id" bigserial PRIMARY KEY, "string" text, "gin_vector"
tsvector);
INSERT INTO "gin_test"("string") SELECT substr(characters, (random() *
length(characters) + 1)::integer, 10) FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as
symbols(characters), generate_series(1, 1000000) as q;
UPDATE "gin_test" set "gin_vector" = to_tsvector("string");
```

Запити для тестування:

```
SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR
("gin_vector" @@ to_tsquery('bnm'));
SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'))
GROUP BY "id" % 2;
```

Створення індексу:

```
DROP INDEX IF EXISTS "gin_index";
CREATE INDEX "gin_index" ON "gin_test" USING gin("gin_vector");
```

Результати і час виконання на скріншотах з psql

Запити без індексування:

```
Секундомер включён.
postgres=# DROP INDEX IF EXISTS "gin_index";
ПОВІДОМЛЕННЯ:  індекс "gin_index" не існує, пропускається
DROP INDEX
Время: 1,634 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 203,518 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
count
-----
19142
(1 строка)

Время: 474,229 мс
postgres=# SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('bnm'));
sum
-----
23943769938
(1 строка)

Время: 1188,034 мс (00:01,188)
postgres=# SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm')) GROUP BY "id" % 2;
min | max
-----+-----
100 | 999994
 45 | 999937
(2 строки)

Время: 1120,586 мс (00:01,121)
```

Запити з індексуванням:

```
postgres=# CREATE INDEX "gin_index" ON "gin_test" USING gin("gin_vector");
CREATE INDEX
Время: 355,983 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 156,321 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
count
-----
19142
(1 строка)

Время: 25,425 мс
postgres=# SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('bnm'));
sum
-----
23943769938
(1 строка)

Время: 243,217 мс
postgres=# SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm')) GROUP BY "id" % 2;
min | max
-----+-----
 45 | 999937
100 | 999994
(2 строки)

Время: 13,533 мс
```

З отриманих результатів бачимо, що в усіх заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). Це відбувається завдяки головній особливості індексування GIN: кожне значення шуканого ключа зберігається один раз і запит іде не по всій таблиці, а лише по тим даним, що містяться у списку появи цього ключа. Для даних типу `numeric` даний тип індексування використовувати недоцільно і неможливо.

Завдання №3

Тригер створений для таблиці Pupils. Під час додання нового учня він прив'язується до викладача та предмета.

Команди, що ініціюють виконання тригера:

```
CREATE TRIGGER new_pupil_arrived  
AFTER INSERT ON public."Pupils"  
FOR EACH ROW EXECUTE PROCEDURE assignPupilToSubject();
```

Текст тригера:

```
CREATE OR REPLACE FUNCTION assignPupilToSubject() RETURNS TRIGGER AS $$  
declare  
    teacherID bigint;  
    subjectID bigint;  
BEGIN  
    select id into teacherID from public."Teachers" order by random() limit 1;  
    select id into subjectID from public."Subjects" order by random() limit 1;  
    insert into public."PupilsSubjects"(pupilsid, subjectsid) values (NEW."Id",  
subjectID);  
    insert into public."TeachersSubjects"(teachersid, subjectsid) values (teacherID,  
subjectID);  
    RETURN NULL;  
END  
$$ LANGUAGE 'plpgsql'
```

Результат роботи тригера

itqsairb/itqsairb@ElephantDB

Query Editor

Query History

Scratch Pad

1

2

3

4

5

6

INSERT INTO public."Pupils"("Name", "Patronymic", "Surname") values ('Testing', 'New', 'User');

select * from public."Pupils"

join public."PupilsSubjects" on public."PupilsSubjects".pupilsid = public."Pupils".Id

join public."Subjects" on public."PupilsSubjects".subjectsid = public."Subjects".id;

Data Output

Explain

Messages

Notifications

	Id bigint	Name character varying (20)	Patronymic character varying (20)	Surname character varying (20)	id bigint	pupilsid bigint	subjectsid bigint	id bigint	name character varying (20)	marksid bigint
1	24	FN	LH	GC	9	24	42	42	VS	8
2	25	IY	UM	IN	8	25	41	41	ZX	8
3	26	ZB	LA	VW	7	26	35	35	LC	6
4	36	Testing	New	User	10	36	34	34	LK	6

За результатами видно що після додання нового студента з’явилися відповідні записи у Subjects та PupilsSubjects

Завдання №4

Для цього завдання також створювалась окрема таблиця з деякими початковими даними:

```
DROP TABLE IF EXISTS "transactions";  
CREATE TABLE "transactions"(  
    "id" bigserial PRIMARY KEY,  
    "numeric" bigint,  
    "text" text  
);
```

```
INSERT INTO "transactions"("numeric", "text") VALUES (111, 'string1'), (222,  
'string2'), (333, 'string3');
```

ДНОГО ЧИТАННЯ.

REPEATABLE READ

На цьому рівні ізоляції T2 не бачитиме змінені дані транзакцією T1, але також не зможе отримати доступ до тих самих даних.

Тут видно, що друга не бачить змін з першої:

```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ^
READ WRITE;
START TRANSACTION
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;INSERT INTO
"transactions"("numeric", "text") VALUES (444, 'string4');DELETE FROM "trans
actions" WHERE "id"=1;
UPDATE 3
INSERT 0 1
DELETE 1
postgres=#
```

```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ^
READ WRITE;
START TRANSACTION
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)
postgres=#
```

А тут, що отримуємо помилку при спробі доступу до тих самих даних:

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# SELECT * FROM "transactions";
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT;
ROLLBACK
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)
```

Бачимо, що не виникає читання фантомів та повторного читання, а також заборонено одночасний доступ до незбережених даних. Хоча класично цей рівень ізоляції призначений для попередження повторного читання.

SERIALIZABLE

На цьому рівні транзакції поведуть себе так, ніби вони не знають одна про одну.

Вони 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=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
 2  |    223  | string2
 3  |    334  | string3
 4  |    444  | string4
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
 2  |    223  | string2
 3  |    334  | string3
 4  |    444  | string4
(3 строки)

postgres=# COMMIT;
COMMIT
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
 2  |    223  | string2
 3  |    334  | string3
 4  |    444  | string4
(3 строки)

postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# DELETE FROM "transactions" WHERE "id"=1;
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT
postgres=# ROLLBACK
postgres=# COMMIT
postgres=#
```

У попередньому випадку вдалось “відкатити” другу транзакцію і це не вплинуло на подальшу можливість роботи в терміналі. На цьому ж рівні навіть після завершення першої не вдалося зробити ні COMMIT ні ROLLBACK для другої транзакції. Взагалі, в класичному представленні цей рівень призначений для недопущення явища читання фантомів. На цьому рівні ізоляції ми отримуємо максимальну узгодженість даних і можемо бути впевнені, що зайві дані не будуть зафіксовані.

READ COMMITTED

На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=#
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)
```

```
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)
```

```
postgres=#
```

Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=#

postgres=#
postgres=#
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=#

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

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=#
```

На цьому знімку також бачимо, що друга транзакція (справа) не може внести дані у базу, доки не завершилась попередня.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
-
```

А тут бачимо, що після завершення першої, друга транзакція виконала запит, змінивши вже ті дані, що були закомічені першою транзакцією

```

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=#

```

```

^ postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#

```

Коли T2 бачить дані T1 запитів UPDATE, DELETE виникає феномен повторного читання, а коли бачить дані запиту INSERT – читання фантомів.