

**Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”**

**Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп’ютерних
систем**

ЛАБОРАТОРНА РОБОТА № 3

з дисципліни

“Бази даних і засоби управління”

Група: KB-03

Виконав: Палажченко Максим

Оцінка:

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.

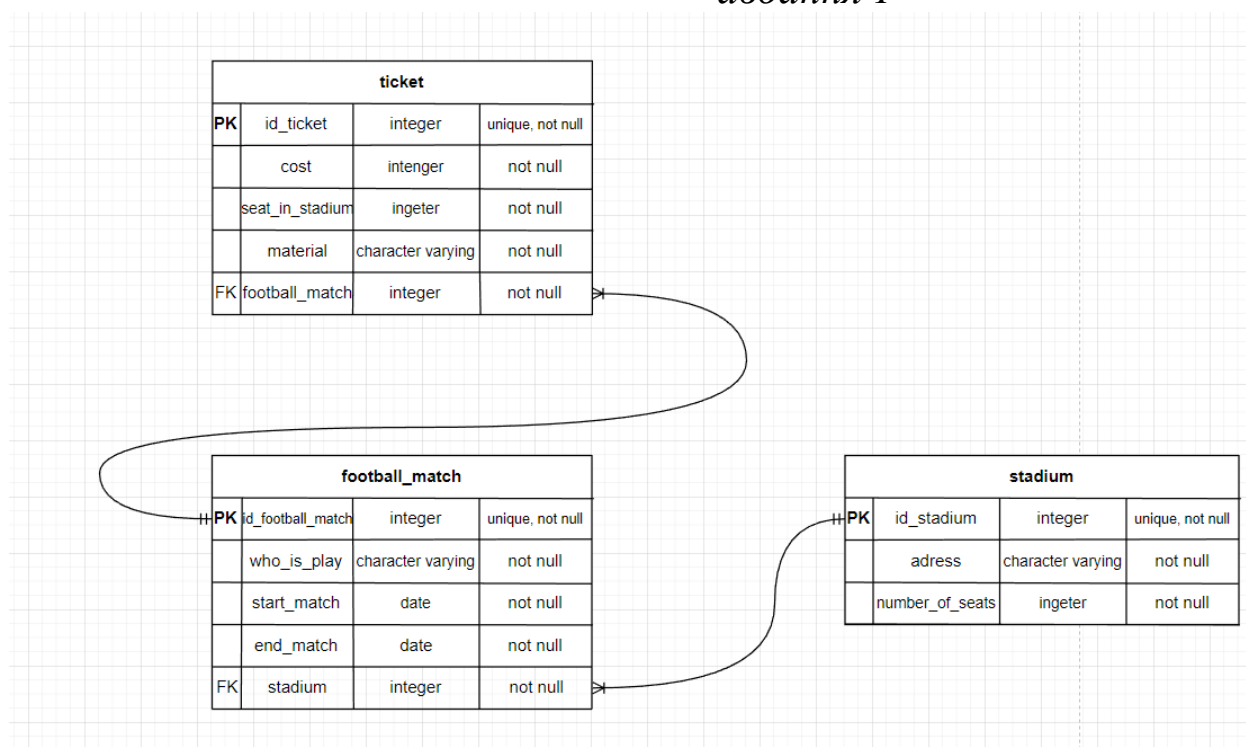
24	GIN, BRIN	after update, insert
----	-----------	----------------------

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

Git

3

авдання 1



Сутність	Атрибут	Опис Атрибути	Тип	Обмеження
ticket	id	unique identifier	integer	not null unique

	cost	ціна в €	integer	not null
	seat_in_the_stadium	Місце на стадіонні для вболівальник	integer	not null
	material	Матеріал з якого виготовлено квиток	character varying	not null
	football_match	посилання на характеристику	integer	not null
football_match	id	unique identifier	integer	not null unique
	who_is_play	які команди грають?	character varying	not null
	start_match	час початку матчу	date	not null
	end_match	час кінця матчу	date	not null
	stadium	посилання на характеристику	integer	not null
stadium	id	unique identifier	integer	not null unique
	adress	адреса стадіону	character varying	not null
	number_of_seats	кількість місць	integer	not null

Опис функціоналу меню:

Update – оновлення даних в певній колонці,

Add – додання нової колонки,

Delete – видалення нової колонки

Random - рандомна генерація нових значень для колонок на n раз

Search – пошук по базовим даним якоїсь колонки

Info about tables – повна інформація по бд

Класи сутностей в ORM

```
class Stadium(Base):
    __tablename__ = 'stadium'
    id_stadium = Column('id_stadium', Integer, primary_key=True, autoincrement=True)
    adress = Column('adress', String(99))
    number_of_seats = Column('number_of_seats', Integer)
    stadium_football_match = relationship('Football_match', cascade='delete')

    def __repr__(self):
        return "<Stadium(id_stadium='{}', adress='{}',
number_of_seats='{}')>".format(self.id_stadium, self.adress,
self.number_of_seats)

class Football_match(Base):
    __tablename__ = 'football_match'
    id_football_match = Column('id_football_match', Integer, primary_key=True,
autoincrement=True)
    who_is_play = Column('who_is_play', String(50))
    start_match = Column('start_match', Date)
    end_match = Column('end_match', Date)
    stadium = Column('stadium', Integer, ForeignKey('Stadium.id_stadium',
onupdate='cascade'), primary_key=True)
    football_match_ticket = relationship('Ticket', cascade='delete')

    def __repr__(self):
        return "<Football_match(id_football_match='{}', who_is_play='{}', start_match='{}',
end_match='{}')>".format(
self.id_football_match,
self.who_is_play,
self.start_match,
self.end_match)

class Ticket(Base):
    __tablename__ = 'ticket'
    id ticket = Column('id ticket', Integer, primary_key=True, autoincrement=True)
    cost = Column('cost', Integer)
    seat_in_the_train = Column('seat_in_the_train', Integer)
    material = Column('material', String(50))
    football_match = Column('football_match', Integer,
ForeignKey('Football_match.id_football_match', onupdate='cascade'), primary_key=True)

    def __repr__(self):
        return "Ticket(id ticket='{}', cost='{}', seat_in_the_train='{}',
material='{}')".format(self.id_ticket,
```

```
self.cost,  
self.seat_in_the_train,  
self.material)
```

авдання 2

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

GIN

GIN – так званий обернений індекс. 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, у яких він зустрічається. Індекс GIN зберігає набір пар виду: ключ, список появи ключа – де список появи — набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з'являється багато разів.

Запити мовою SQL

```
DROP TABLE IF EXISTS "GIN_test";
```

```
CREATE TABLE "GIN_test"("id" bigserial PRIMARY KEY, "doc" text, "doc_tsv"  
tsvector); INSERT INTO "GIN_test"("doc") SELECT chr(trunc(65 +  
random()*25)::int)||chr(trunc(65 +
```

```
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +  
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +  
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +  
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int) FROM  
generate_series(1, 1000000) as q;
```

```
UPDATE "GIN_test" SET "doc_tsv" = to_tsvector("doc");
```

```
SELECT COUNT(*) FROM "GIN_test" where "id" % 11 = 0;
```

```
SELECT COUNT(*) FROM "GIN_test" WHERE ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
```

```
SELECT AVG("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP')) or  
("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
```

```
SELECT MAX("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP'))  
GROUP BY "id" % 11 = 0;
```

```
DROP INDEX IF EXISTS "GIN_time_index";
```

```
CREATE INDEX "GIN_time_index" ON "GIN_test" USING gin("doc_tsv");
```

Перевірка результатів

До індексування:

```
SQL Shell (psql)
```

```
test=# SELECT COUNT(*) FROM "GIN_test" where "id" % 11 = 0;
count
-----
90909
(1 řĖřĕър)

Время: 123,695 мс
test=# SELECT COUNT(*) FROM "GIN_test" WHERE ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
count
-----
1
(1 řĖřĕър)

Время: 1138,001 мс (00:01,138)
test=# SELECT AVG("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYPB')) or ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
avg
-----
826.5000000000000000
(1 řĖřĕър)

Время: 2059,870 мс (00:02,060)
test=# SELECT MAX("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYPB')) GROUP BY "id" % 11 = 0;
max
----
774
(1 řĖřĕър)

Время: 1085,142 мс (00:01,085)
test=#
```

Після індексування:

```
SQL Shell (psql)
test=# CREATE INDEX "GIN_time_index" ON "GIN_test" USING gin("doc_tsv");
CREATE INDEX
Время: 5620,484 мс (00:05,620)
test=# SELECT COUNT(*) FROM "GIN_test" where "id" % 11 = 0;
count
-----
90909
(1 строка)

Время: 134,281 мс
test=# SELECT COUNT(*) FROM "GIN_test" WHERE ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
count
-----
1
(1 строка)

Время: 2,462 мс
test=# SELECT AVG("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP')) or ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
avg
-----
826.500000000000000
(1 строка)

Время: 120,670 мс
test=# SELECT MAX("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP')) GROUP BY "id" % 11 = 0;
max
-----
774
(1 строка)

Время: 0,546 мс
test=#
```

З результатів бачимо, що використання індексування GIN значно підвищило швидкість пошуку даних (окрім першого запиту, тому що на даний виклик індексування не впливає). В цілому, така поведінка є очікуваною, тому що основна ідея індексування GIN це те, що кожне значення шуканого ключа зберігається лише один раз і запит йде лише по тим даним, що містяться у списку появи цього ключа.

BRIN

BRIN – техніка індексації даних, призначена для обробки великих таблиць, в яких значення індексованого стовпця має деяку природну кореляцію з фізичним положенням рядка в таблиці. Вони мають такі якості партиціонованих таблиць, як швидка вставка рядка, швидке створення індексу, без необхідності явного оголошення партицій. Спрощено кажучи, BRIN добре працює для тих стовпців, значення яких корелюють з їх фізичним розташуванням у таблиці.

Працює це наступним чином. Таблиця розбивається на зони (range) розміром кілька сторінок (або блоків, що те саме) — звідси й назва: Block Range Index, BRIN. Для кожної зони в індексі зберігається інформація про дані в цій зоні. Як правило, це мінімальне та максимальне значення, але буває інакше. Якщо при виконанні запиту, що містить умову на стовпець, шукані значення не потрапляють у діапазон, всю зону можна сміливо пропускати; якщо ж потрапляють - усі рядки у всіх блоках зони доведеться переглянути та вибрати серед них підходящі.

Запити мовою SQL

```
DROP TABLE IF EXISTS "BRIN_test";
```

```
CREATE TABLE "BRIN_test"("id" int PRIMARY KEY, "date" timestamp NOT NULL, "level" integer, "msg" text);
```

```
INSERT INTO "BRIN_test"("id", "date", "level", "msg") SELECT q,  
CURRENT_TIMESTAMP + ( q || 'minute' ) :: interval, random() * 6, md5(q::text) FROM  
generate_series(1,1000000) as q;
```

```
SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196'  
and  
'2021-12-27 02:03:10.318196';
```



```
SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27
00:33:10.318196' and '2021-12-27 02:45:10.318196';
```

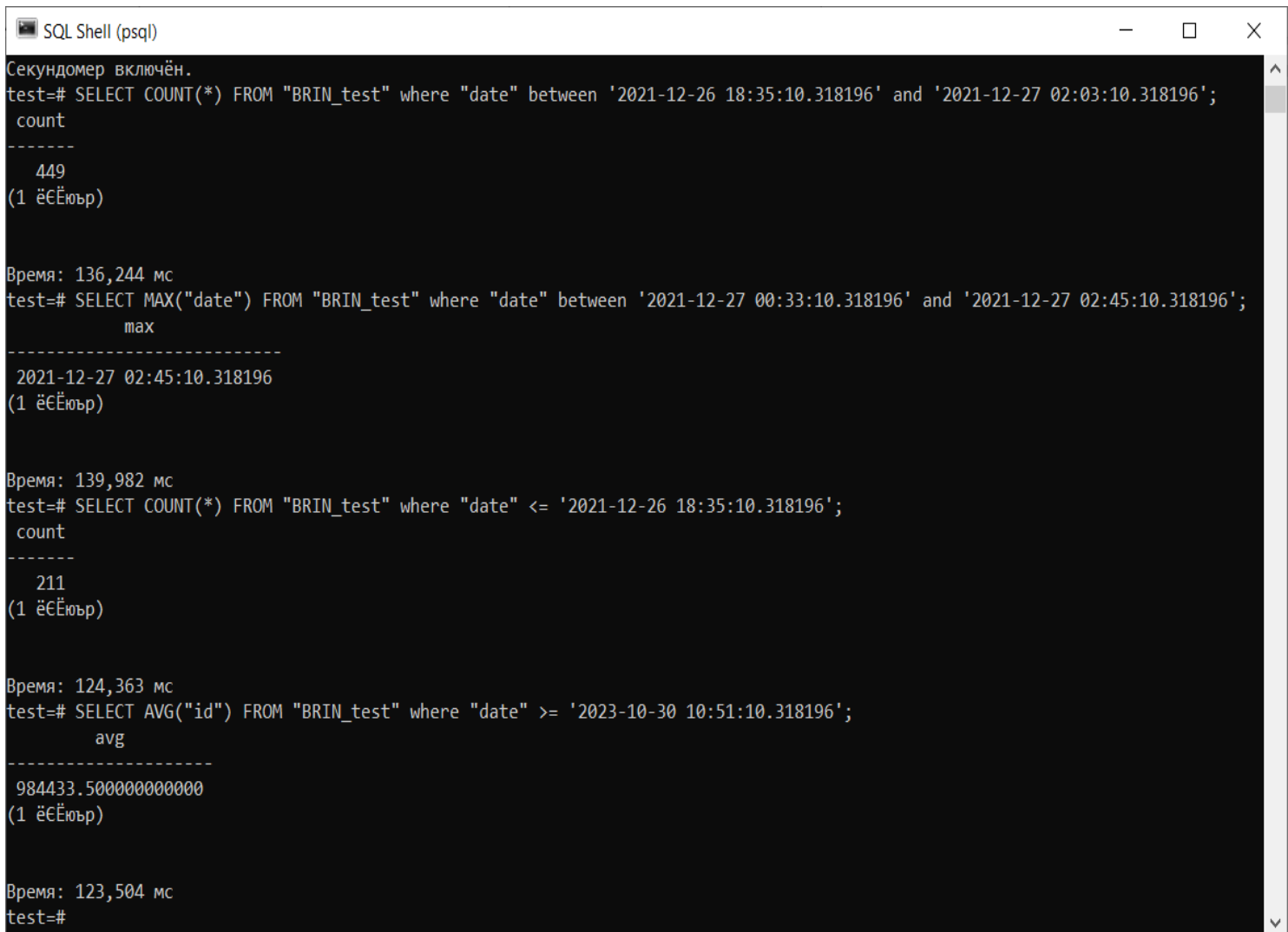
```
SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26
18:35:10.318196'; SELECT AVG("id") FROM "BRIN_test" where "date" >=
'2023-10-30 10:51:10.318196';
```

```
DROP INDEX IF EXISTS "BRIN_test_index";
```

```
CREATE INDEX "BRIN_test_index" ON "BRIN_test" USING brin("date");
```

Перевірка результатів

До індексування:



```
SQL Shell (psql)
Секундомер включён.
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196' and '2021-12-27 02:03:10.318196';
count
-----
    449
(1 ёёёёёё)

Время: 136,244 мс
test=# SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27 00:33:10.318196' and '2021-12-27 02:45:10.318196';
max
-----
2021-12-27 02:45:10.318196
(1 ёёёёёё)

Время: 139,982 мс
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26 18:35:10.318196';
count
-----
    211
(1 ёёёёёё)

Время: 124,363 мс
test=# SELECT AVG("id") FROM "BRIN_test" where "date" >= '2023-10-30 10:51:10.318196';
avg
-----
984433.500000000000
(1 ёёёёёё)

Время: 123,504 мс
test=#
```

Після індексування:

```
SQL Shell (psql)
test=# CREATE INDEX "BRIN_test_index" ON "BRIN_test" USING brin("date");
CREATE INDEX
Время: 177,393 мс
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196' and '2021-12-27 02:03:10.318196';
count
-----
      449
(1 ėĖĖĖĖĖĖ)

Время: 3,100 мс
test=# SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27 00:33:10.318196' and '2021-12-27 02:45:10.318196';
max
-----
2021-12-27 02:45:10.318196
(1 ėĖĖĖĖĖĖ)

Время: 1,965 мс
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26 18:35:10.318196';
count
-----
      211
(1 ėĖĖĖĖĖĖ)

Время: 1,824 мс
test=# SELECT AVG("id") FROM "BRIN_test" where "date" >= '2023-10-30 10:51:10.318196';
avg
-----
984433.500000000000000
(1 ėĖĖĖĖĖĖ)

Время: 4,081 мс
test=#
```

З результатів, отриманих до і після використання індексування BRIN бачимо, що швидкість пошуку необхідних даних значно збільшилася. Знову ж таки, така поведінка є очікуваною, тому що індекси створені для пришвидшення пошуку необхідної інформації. Індекси BRIN ефективні, якщо впорядкування значень ключів відповідає організації блоків на рівні зберігання. У найпростішому випадку це може вимагати фізичного впорядкування таблиці, яке часто є порядком створення рядків у ній, щоб відповідати порядку ключа. Ключі до згенерованих порядкових номерів або створених даних є найкращими кандидатами для індексу BRIN.

Завдання №3

Для тестування тригера було створено дві таблиці:

```
DROP TABLE IF EXISTS
"trigger_test"; CREATE TABLE
"trigger_test"(
    "trigger_testID" bigserial PRIMARY KEY,
    "trigger_testName" text
);
DROP TABLE IF EXISTS
"trigger_test_log"; CREATE TABLE
"trigger_test_log"(
    "id" bigserial PRIMARY KEY,
    "trigger_test_log_ID" bigint,
    "trigger_test_log_name" text
);
```

Початкові дані у таблицях:

```
INSERT INTO "trigger_test"("trigger_testName")
VALUES ('trigger_test1'), ('trigger_test2'), ('trigger_test3'), ('trigger_test4'), ('trigger_test5'),
('trigger_test6'), ('trigger_test7'), ('trigger_test8'), ('trigger_test9'), ('trigger_test10');
```

Команди, що ініціюють виконання тригера:

```
CREATE TRIGGER "after_update_insert_trigger"
BEFORE DELETE OR UPDATE ON "trigger_test"
FOR EACH ROW
EXECUTE procedure after_update_insert_func();
```

Текст тригера:

```
CREATE OR REPLACE FUNCTION after_update_insert_func() RETURNS TRIGGER as $trigger$
DECLARE
    CURSOR_LOG CURSOR FOR SELECT * FROM
    "trigger_test_log"; row_ "trigger_test_log"%ROWTYPE;

BEGIN
    IF old."trigger_testID" % 2 = 0 THEN IF
        old."trigger_testID" % 3 = 0 THEN
            RAISE NOTICE 'trigger_testID is multiple of 2 and 3'; FOR
            row_ IN CURSOR_LOG LOOP
                UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_. "trigger_test_log_name" || '_log' WHERE "id" = row_. "id";
            END
            LOOP;
            RETUR
            N OLD;
        ELSE
            RAISE NOTICE 'trigger_testID is even';
            INSERT INTO "trigger_test_log"("trigger_test_log_ID", "trigger_test_log_name") VALUES
(old."trigger_testID", old."trigger_testName");
            UPDATE "trigger_test_log" SET "trigger_test_log_name" = trim(BOTH '_log' FROM
"trigger_test_log_name");
            RETUR
            N NEW;
        END IF;
    ELSE

```

```

        RAISE NOTICE 'trigger_testID is odd';
    FOR row_ IN CURSOR_LOG LOOP
        UPDATE "trigger_test_log" SET "trigger_test_log_name" = '_' ||
row_."trigger_test_log_name" || '_log' WHERE "id" = row_."id";
    END
    LOOP;
    RETUR
    N OLD;
E
ND
IF;
END
;
$trigger$ LANGUAGE plpgsql;

```

Скріншоти зі змінами у таблицях бази даних

Початковий стан

```
SELECT * FROM "trigger_test";
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	3	trigger_test3
4	4	trigger_test4
5	5	trigger_test5
6	6	trigger_test6
7	7	trigger_test7
8	8	trigger_test8
9	9	trigger_test9
10	10	trigger_test10

```
SELECT * FROM "trigger_test_log";
```

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Після виконання запиту на оновлення

```
UPDATE "trigger_test" SET "trigger_testName" = "trigger_testName" || '_log' WHERE "trigger_testID" % 2 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	3	trigger_test3
3	5	trigger_test5
4	7	trigger_test7
5	9	trigger_test9
6	2	trigger_test2_log
7	4	trigger_test4_log
8	6	trigger_test6
9	8	trigger_test8_log
10	10	trigger_test10_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	trigger_test2
2	2	4	trigger_test4
3	3	8	trigger_test8
4	4	10	trigger_test10

Наочно можемо переконатись, що виконалась та гілка алгоритму тригера, що відповідає за парні рядки (оскільки є умова для парних), а для 6 рядка він також виконав, проте пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на видалення потрібн повертати новий стан, а при запиті на оновлення старий.

Після виконання запиту на видалення

```
DELETE FROM "trigger_test" WHERE "trigger_testID" % 3 = 0;
```

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	2	trigger_test2
3	4	trigger_test4
4	5	trigger_test5
5	7	trigger_test7
6	8	trigger_test8
7	10	trigger_test10

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text

Якщо виконувати ці запити окремо одне від одного, то у таблиці trigger_test видаляються кратні трьом рядки, але таблиця trigger_test_log виявляється пустою. Так відбувається тому, що у гілці алгоритму для чисел кратних трьом у trigger_test_log лише модифуються існуючі записи, але нові не додаються.

Оскільки до цього не було виконан оновлення, ця таблиця пуста і модифікувати нема чого.

Якщо зробити вищезгадані запити підряд побачимо наступне:

	trigger_testID [PK] bigint	trigger_testName text
1	1	trigger_test1
2	5	trigger_test5
3	7	trigger_test7
4	2	trigger_test2_log
5	4	trigger_test4_log
6	8	trigger_test8_log
7	10	trigger_test10_log

	id [PK] bigint	trigger_test_log_ID bigint	trigger_test_log_name text
1	1	2	__trigger_test2_log_log_log
2	2	4	__trigger_test4_log_log_log
3	3	8	__trigger_test8_log_log_log
4	4	10	__trigger_test10_log_log_log

Бачимо, що записи кратні трьом видалились з trigger_test, а до текстових полів цих записів у кінці додалось "_log".

До текстових полів trigger_test_log на початку додалися два вимови "_", а в кінці три "_log". Один "_log" в кінці додався завдяки виконанню запит update для всіх паних рядків. А інші два "_log" та два символи "_" на початку додалися тому, що запит на видалення для записів 3 та 9 виконались за тією самою гілкою алгоритму (кратні трьом), а запит на видалення запису 6 виконався за іншою гілкою (кратність 2 та 3)

Завдання 4

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Самі транзакції особливих пояснень не вимагають, транзакція — це N ($N \geq 1$) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. Ізольованість транзакції показує те, наскільки сильно вони впливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у виборі між високою узгодженістю даних між транзакціями та швидкістю виконання цих транзакцій.

Варто зазначити, що найвищу швидкість виконання та найнижчу узгодженість має рівень `read uncommitted`. Найнижчу швидкість виконання та найвищу узгодженість — `serializable`.

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.

Стандарт SQL-92 визначає наступні рівні ізоляції:

1. *Serializable (впорядкованість)*

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). Як бачимо, дані у транзакціях ізолювано.

<pre>START TRANSACTION lab3=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE; SET lab3=# SELECT * FROM "tab4"; id num char -----+-----+----- 1 100 ABC 2 200 BCL 3 300 CAB (3 rows)</pre>	<pre>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE START TRANSACTION SET lab3=# SELECT * FROM "tab4"; id num char -----+-----+----- 1 100 ABC 2 200 BCL 3 300 CAB (3 rows)</pre>
---	---

<pre> 3 300 CAB (3 rows) lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) lab3=# </pre>	<pre> lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# UPDATE "task4" SET "num" = "num" + 1; UPDATE 3 lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 101 ABC 2 201 BCA 3 301 CAB (3 rows) lab3=# </pre>
--	---

Тепер при оновленіданих в T2(частина фото зправа) бачимо, що T2 блокується поки T1 не не зафіксує зміни або не відмінить їх.

<pre> lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) lab3=# UPDATE "task4" SET "num" = "num" + 1; ERROR: could not serialize access due to concurrent update lab3=# ROLLBACK lab3=# ; ROLLBACK lab3=# </pre>	<pre> lab3=# lab3=# UPDATE "task4" SET "num" = "num" + 1; UPDATE 3 lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 101 ABC 2 201 BCA 3 301 CAB (3 rows) lab3=# COMMIT; COMMIT lab3=# </pre>
--	--

2. Repeatable read (повторюваність читання)

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

<pre> lab3=# START TRANSACTION; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE; START TRANSACTION SET lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 101 ABC 2 201 BCA 3 301 CAB (3 rows) lab3=# UPDATE "task4" SET "num" = "num" + 1; UPDATE 3 lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# </pre>	<pre> lab3=# START TRANSACTION; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE; START TRANSACTION SET lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 101 ABC 2 201 BCA 3 301 CAB (3 rows) lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 101 ABC 2 201 BCA 3 301 CAB (3 rows) lab3=# </pre>
---	---

Тепер транзакція T2(зправа) буде чекати поки T1 не не зафіксує зміни або не відмінить їх.

<pre> lab3=# UPDATE "task4" SET "num" = "num" + 4; UPDATE 3 lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# </pre>	<pre> lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) lab3=# UPDATE "task4" SET "num" = "num" + 4; </pre>
---	---

<pre> lab3=# lab3=# lab3=# lab3=# COMMIT; COMMIT lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 104 ABC 2 204 BCA 3 304 CAB (3 rows) lab3=# </pre>	<pre> lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) lab3=# UPDATE "task4" SET "num" = "num" + 4; ERROR: could not serialize access due to concurrent update lab3=# ROLLBACK; ROLLBACK lab3=# </pre>
---	---

Як бачимо, Repeatable read не дозволяє виконувати операції зміни даних, якщо дані вже було модифіковано у іншій незавершеній транзакції. Тому використання Repeatable read рекомендоване тільки для режиму читання.

3. *Read committed (читання фіксованих даних)* Прийнятий за замовчуванням рівень для PostgreSQL. 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.

<pre> lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 104 ABC 2 204 BCA 3 304 CAB (3 rows) lab3=# START TRANSACTION; START TRANSACTION lab3=# UPDATE "task4" SET "num" = "num" - 4; UPDATE 3 lab3=# COMMIT; COMMIT lab3=# </pre>	<pre> lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 104 ABC 2 204 BCA 3 304 CAB (3 rows) lab3=# START TRANSACTION; START TRANSACTION lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 104 ABC 2 204 BCA 3 304 CAB (3 rows) lab3=# SELECT * FROM "task4"; id num char -----+-----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) lab3=# </pre>
--	--

4. Read uncommitted

(читання незафіксованих даних) Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той ж рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.