

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»
Тема: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав: студент III курсу
ФПМ групи КВ-94
Заварін В. О.
Перевірів: доц. Петрашенко А. В.

Київ – 2021

Мета роботи: здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Загальне 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 6

У другому завданні проаналізувати індекси BTree, BRIN.

Умова для тригера – after update, insert

Завдання 1

Інформація про модель та структуру бази даних

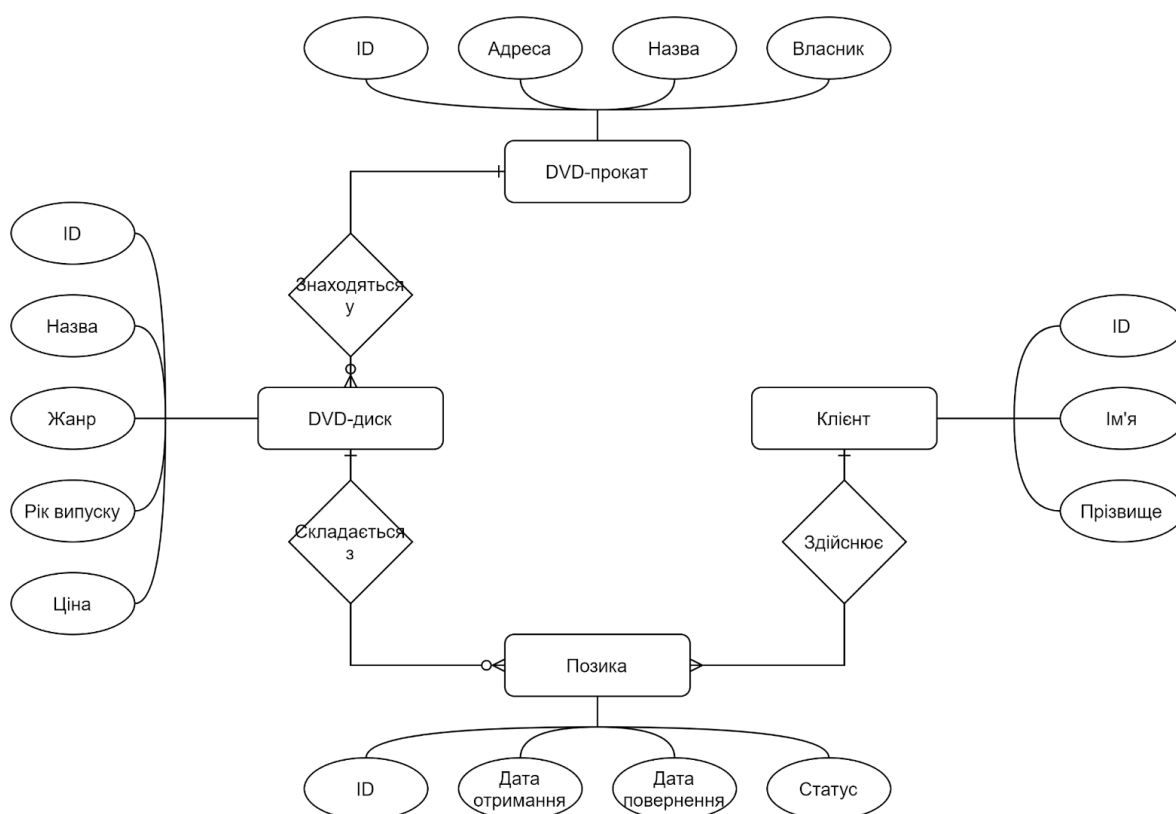


Рис. 1 - Концептуальна модель предметної області “DVD-rental-store”

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

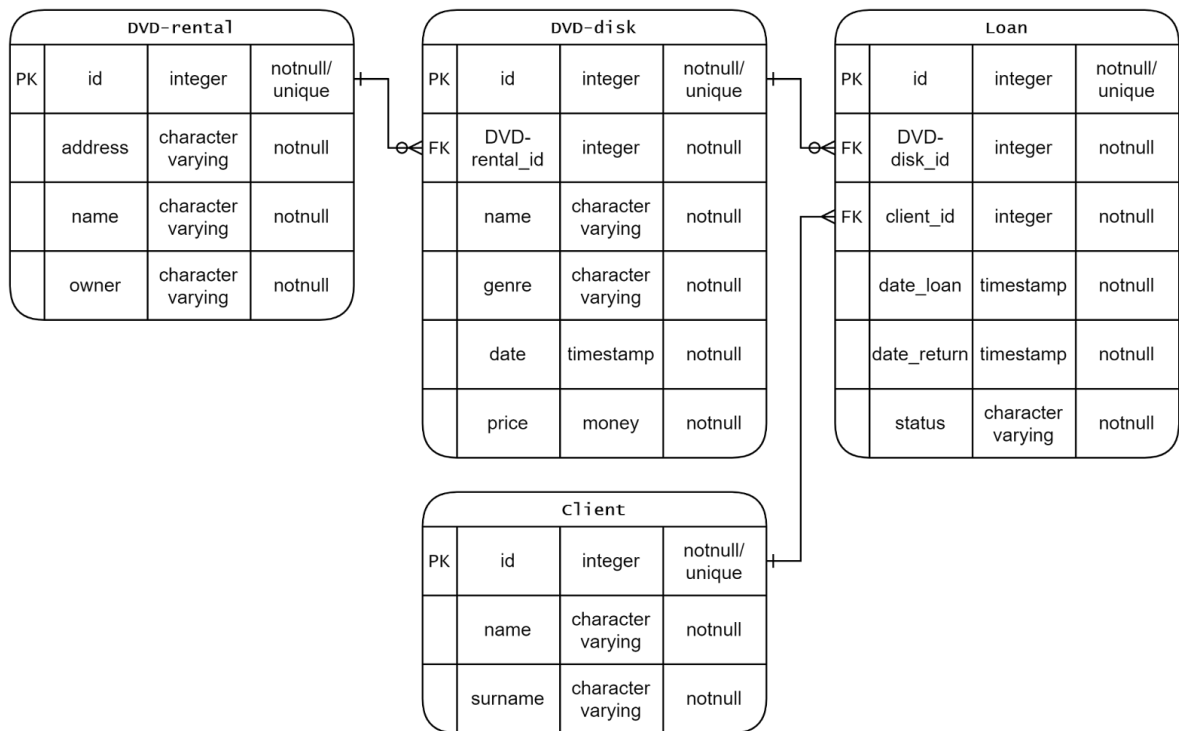


Рис. 2 – Логічна модель бази даних

Для перетворення модуля “Model” програми, створеного в 2 лабораторній роботі, у вигляд об’єктно-реляційної моделі було використано бібліотеку “peewee”

Код сутносних класів програми:

```
database_proxy = peewee.DatabaseProxy()
```

```
class DVD_rental_store_table(peewee.Model):
    class Meta(object):
        database = database_proxy
        schema = f"DVD_rental_store"
```

```
class DVD_rental(DVD_rental_store_table):
    address = peewee.CharField(max_length=255, null=False)
    name = peewee.CharField(max_length=255, null=False)
    owner = peewee.CharField(max_length=255, null=False)
```

```
class DVD_disk(DVD_rental_store_table):
    DVD_rental_id = peewee.ForeignKeyField(DVD_rental, backref="disks")
    name = peewee.CharField(max_length=255, null=False)
    genre = peewee.CharField(max_length=255, null=False)
    date = peewee.DateTimeField(null=False)
    price = peewee.DecimalField(null=False)
```

```
class client(DVD_rental_store_table):
    name = peewee.CharField(max_length=255, null=False)
    surname = peewee.CharField(max_length=255, null=False)
```

```
class loan(DVD_rental_store_table):
    DVD_disk_id = peewee.ForeignKeyField(DVD_disk, backref="loaned")
    client_id = peewee.ForeignKeyField(client, backref="loans")
    date_loan = peewee.DateTimeField(null=False)
    date_return = peewee.DateTimeField(null=False)
    status = peewee.CharField(max_length=255, null=False)
```

Програма працює ідентично програмі з лабораторної роботи 2, за виключенням незначних текстових змін. Інтерфес модуля «model» не було змінено.

Завдання 2

BTree

Для дослідження індексу була створена таблиця, яка має дві колонки: числову і текстову. Вони проіндексовані як BTree. У таблицю було занесено 1000000 записів.

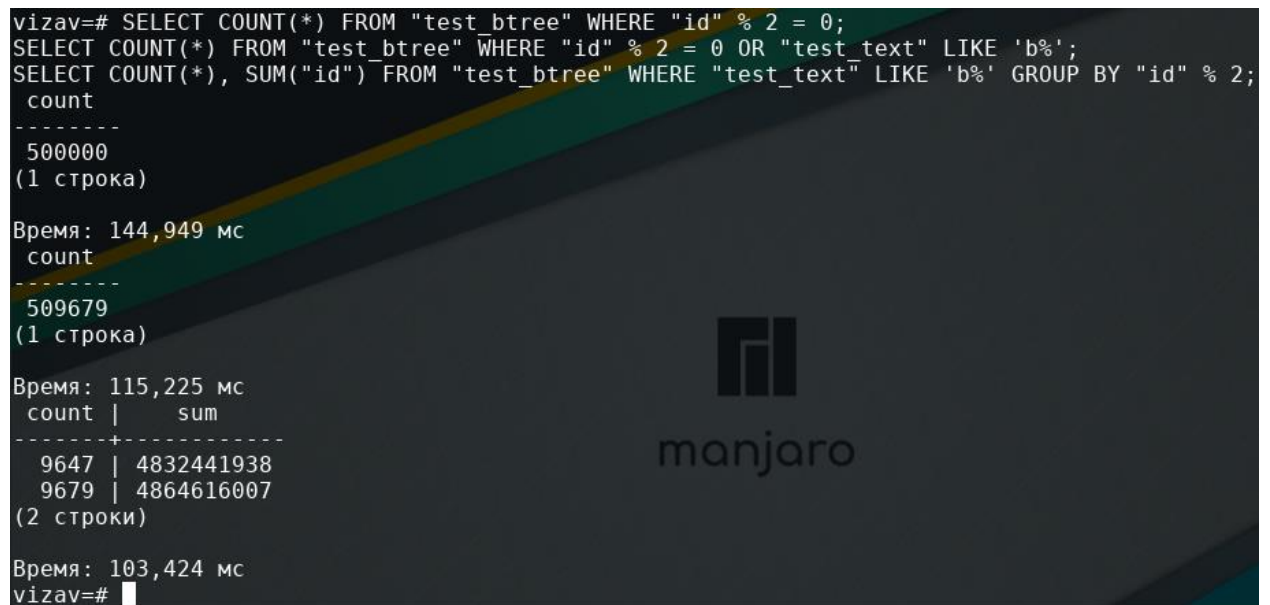
Створення таблиці та її заповнення:

```
DROP TABLE IF EXISTS "test_btree";

CREATE TABLE "test_btree" (
    "id" bigserial PRIMARY KEY,
    "test_text" varchar(255)
);

INSERT INTO "test_btree" ("test_text")
SELECT
    substr(characters, (random() * length(characters) + 1)::integer, 10)
FROM
    (VALUES ('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters),
    generate_series(1, 1000000) as q;
```

Вибір даних без індексу:



```
vizav=# SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
count
-----
500000
(1 строка)

Время: 144,949 мс
count
-----
509679
(1 строка)

Время: 115,225 мс
count | sum
-----+-----
9647 | 4832441938
9679 | 4864616007
(2 строки)

Время: 103,424 мс
vizav=#
```

Сворюємо індекс:

```
DROP INDEX IF EXISTS "test_btree_test_text_index";

CREATE INDEX "test_btree_test_text_index" ON "test_btree" USING gin ("test_text");
```


Вибір даних з створеним індексом:

```
vizav=# SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_btree" WHERE "id" % 2 = 0 OR "test_text" LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_btree" WHERE "test_text" LIKE 'b%' GROUP BY "id" % 2;
count
-----
500000
(1 строка)

Время: 106,441 мс
count
-----
509679
(1 строка)

Время: 123,210 мс
count |      sum
-----+-----
 9647 | 4832441938
 9679 | 4864616007
(2 строки)

Время: 112,568 мс
vizav=#
```



manjaro

BRIN

Для дослідження індексу була створена таблиця, яка дві колонки: test_time типу timestamp without time zone (дата та час (без часового поясу)) і id типу bigserial. Колонка test_time проіндексована як BRIN. У таблицю занесено 1000000 записів.

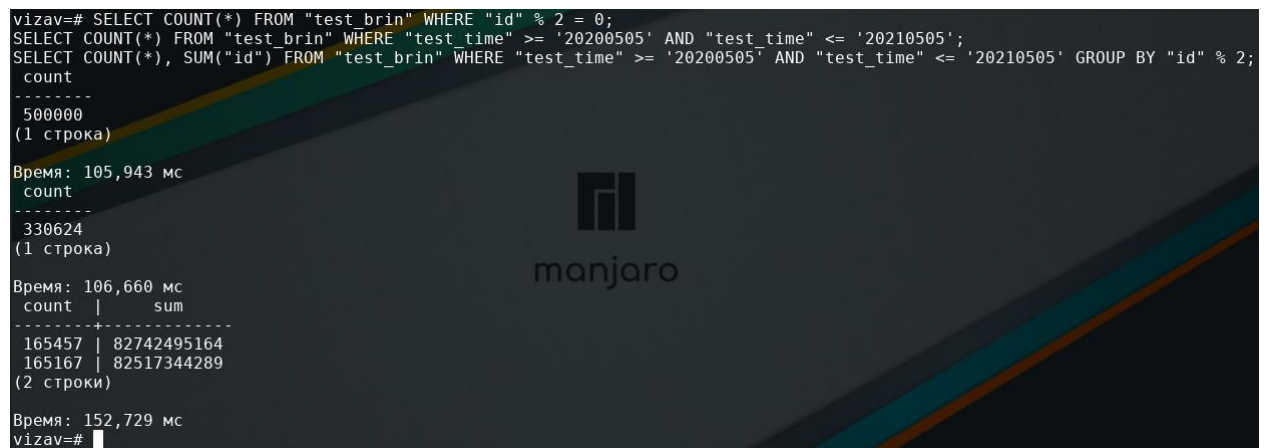
Створення таблиці та її заповнення:

```
DROP TABLE IF EXISTS "test_brin";

CREATE TABLE "test_brin" (
    "id" bigserial PRIMARY KEY,
    "test_time" timestamp
);

INSERT INTO "test_brin"("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;
```

Вибір даних без індексу:



```
vizav=# SELECT COUNT(*) FROM "test_brin" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505';
SELECT COUNT(*), SUM("id") FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505' GROUP BY "id" % 2;
count
-----
500000
(1 строка)

Время: 105,943 мс
count
-----
330624
(1 строка)

Время: 106,660 мс
count |      sum
-----+-----
165457 | 82742495164
165167 | 82517344289
(2 строки)

Время: 152,729 мс
vizav=#
```

Сворюємо індекс:

```
DROP INDEX IF EXISTS "test_brin_test_time_index";

CREATE INDEX "test_brin_test_time_index" ON "test_brin" USING hash ("test_time");
```

Вибір даних з створеним індексом:

```
vizav=# SELECT COUNT(*) FROM "test_brin" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505';
SELECT COUNT(*), SUM("id") FROM "test_brin" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505' GROUP BY "id" % 2;
count
-----
500000
(1 строка)

Время: 102,117 мс
count
-----
330624
(1 строка)

Время: 94,192 мс
count | sum
-----+-----
165457 | 82742495164
165167 | 82517344289
(2 строки)

Время: 126,177 мс
vizav=#
```

Завдання 3

Розробити тригер бази даних PostgreSQL.

Умова для тригера – after update, insert.

Таблиці:

```
DROP TABLE IF EXISTS "reader";
CREATE TABLE "reader" (
    "readerID" bigserial PRIMARY KEY,
    "readerName" varchar(255)
);
```

```
DROP TABLE IF EXISTS "readerLog";
CREATE TABLE "readerLog" (
    "id" bigserial PRIMARY KEY,
    "readerLogID" bigint,
    "readerLogName" varchar(255)
);
```

Тригер:

```
CREATE OR REPLACE FUNCTION update_insert_func() RETURNS TRIGGER as $$

DECLARE
    CURSOR_LOG CURSOR FOR SELECT * FROM "readerLog";
    row_Log "readerLog"%ROWTYPE;

begin
    IF NEW."readerID" % 2 = 0 THEN
        INSERT INTO "readerLog" ("readerLogID", "readerLogName") VALUES (new."readerID",
new."readerName");
        UPDATE "readerLog" SET "readerLogName" = trim(BOTH 'x' FROM "readerLogName");
        RETURN NEW;
    ELSE
        RAISE NOTICE 'readerID is odd';
        FOR row_log IN cursor_log LOOP
            UPDATE "readerLog" SET "readerLogName" = 'y' || row_Log."readerLogName"
|| 'y' WHERE "id" = row_log."id";
        END LOOP;
        RETURN NEW;
    END IF;
END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER "test_trigger"
AFTER UPDATE OR INSERT ON "reader"
FOR EACH ROW
EXECUTE procedure update_insert_func();
```

Принцип роботи:

Тригер спрацьовує після оновлення у таблиці чи при додаванні нових рядків у таблицю reader. Якщо значення ідентифікатора запису, який додається або оновлюється, парне, то цей запис заноситься у додаткову таблицю readerLog. Також, з кожного значення «readerName» видаляються символи «х» на початку і кінці. Якщо значення ідентифікатора непарне, то до кожного значення «readerLogName» у таблиці readerLog додається “у” на початку і кінці.

Занесемо тестові дані до таблиці:

```
INSERT INTO "reader"("readerName")
VALUES ('reader1'), ('reader2'), ('reader3'), ('reader4'), ('reader5');
vizav=# INSERT INTO "reader"("readerName")
VALUES ('reader1'), ('reader2'), ('reader3'), ('reader4'), ('reader5');
NOTICE: readerID is odd
NOTICE: readerID is odd
NOTICE: readerID is odd
INSERT 0 5
Время: 6,436 мс
```

```
vizav=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
-----+-----
      1 | reader1
      2 | reader2
      3 | reader3
      4 | reader4
      5 | reader5
      6 | reader1
      7 | reader2
      8 | reader3
      9 | reader4
     10 | reader5
     11 | reader1
     12 | reader2
     13 | reader3
     14 | reader4
     15 | reader5
     16 | reader1
     17 | reader2
     18 | reader3
     19 | reader4
     20 | reader5
(20 строк)

Время: 0,376 мс
 id | readerLogID | readerLogName
-----+-----
  1 |          2 | ууууууууreader2уууууууу
  2 |          4 | ууууууууreader4уууууууу
  3 |          6 | ууууууууreader1уууууууу
  4 |          8 | ууууууreader3уууууу
  5 |         10 | уууууreader5ууууу
  6 |         12 | ууууreader2уууу
  7 |         14 | уууreader4ууу
  8 |         16 | ууreader1уу
  9 |         18 | уreader3у
 10 |         20 | reader5
(10 строк)

Время: 0,188 мс
vizav=#
```


Оновимо дані в одному з рядків:

```
vizav=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" > 10;
NOTICE: readerID is odd
NOTICE: readerID is odd
NOTICE: readerID is odd
NOTICE: readerID is odd
NOTICE: readerID is odd
UPDATE 10
Время: 5,837 мс
```

```
vizav=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
readerID | readerName
```

readerID	readerName
1	reader1
2	reader2
3	reader3
4	reader4
5	reader5
6	reader1
7	reader2
8	reader3
9	reader4
10	reader5
11	reader1Lx
12	reader2Lx
13	reader3Lx
14	reader4Lx
15	reader5Lx
16	reader1Lx
17	reader2Lx
18	reader3Lx
19	reader4Lx
20	reader5Lx

(20 строк)

Время: 0,275 мс

id	readerLogID	readerLogName
1	2	ууууууууууууreader2уууууууууууу
2	4	ууууууууууууreader4уууууууууууу
3	6	ууууууууууууreader1уууууууууууу
4	8	ууууууууууууreader3уууууууууууу
5	10	ууууууууууууreader5уууууууууууу
6	12	ууууууууууууreader2уууууууууууу
7	14	ууууууууууууreader4уууууууууууу
8	16	ууууууууууууreader1уууууууууууу
9	18	ууууууууууууreader3уууууууууууу
10	20	ууууууууууууreader5уууууууууууу
11	12	ууууууууууууreader2уууууууууууу
12	14	ууууууууууууreader4уууууууууууу
13	16	ууууууууууууreader1уууууууууууу
14	18	ууууууууууууreader3уууууууууууу
15	20	ууууууууууууreader5уууууууууууу

(15 строк)

Время: 0,220 мс

vizav=#

Оскільки серед id рядків, які були оновлено, є непарні числа, то це призвело до додавання до кожного значення «readerLogName» у таблиці readerLog рядка «у» на початку і кінці. Як бачимо, при оновленні парних рядків їх значення буде занесено у таблицю "readerLog" з прибралими символами «х» на початку і кінці.

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 "task4"; id num char -----+----- 1 100 ABC 2 200 BCA 3 300 CAB (3 rows) </pre>	<pre> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE; START TRANSACTION SET lab3=# SELECT * FROM "task4"; id num char -----+----- 1 100 ABC 2 200 BCA 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 оновленні даних в T2(частина фото зправа) бачимо, що T2 блокується поки T1 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 одного 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=# 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 зафіксує зміни або не відмінить їх.


```

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=*# 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)

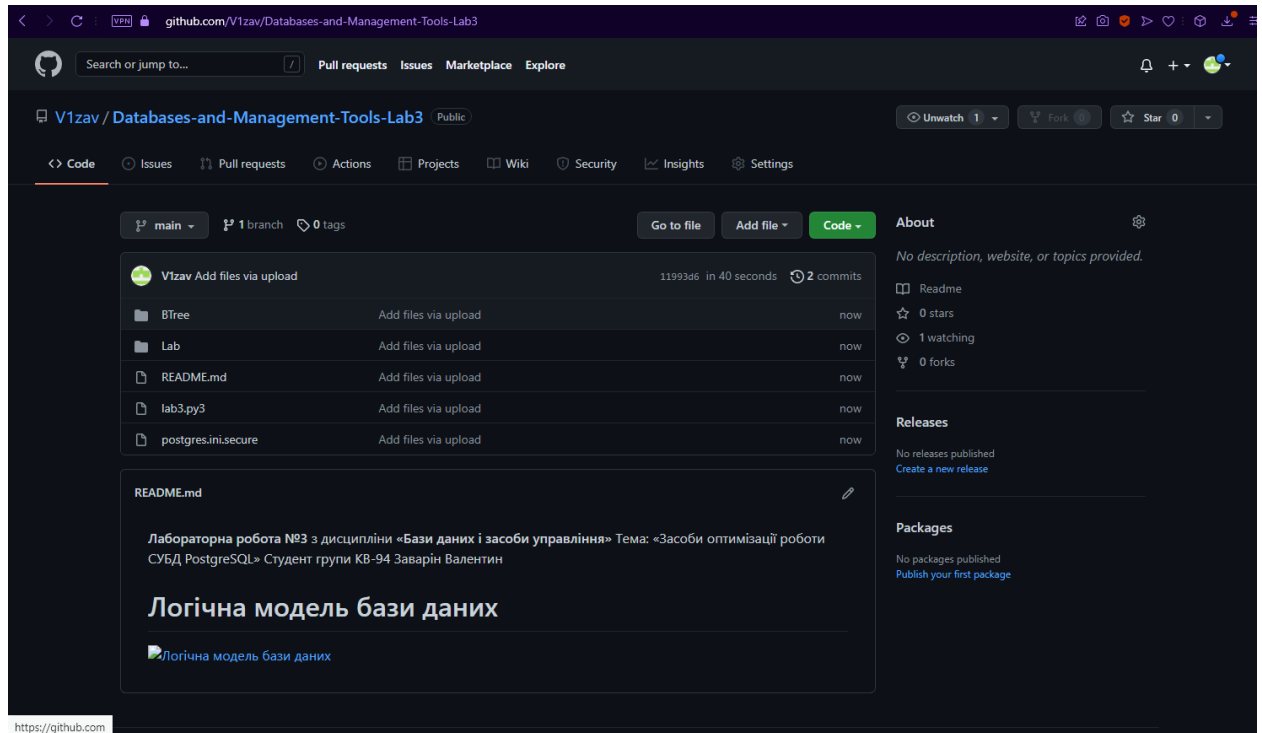
lab3=*#

```

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 же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.

Ілюстрації програмного коду на Github



Посилання на репозиторій: <https://github.com/V1zav/Databases-and-Management-Tools-Lab3>