

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
Микитенко І. П.
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 16

У другому завданні проаналізувати індекси GIN, Hash.

Умова для тригера – after delete, insert

Завдання 1

Інформація про модель та структуру бази даних

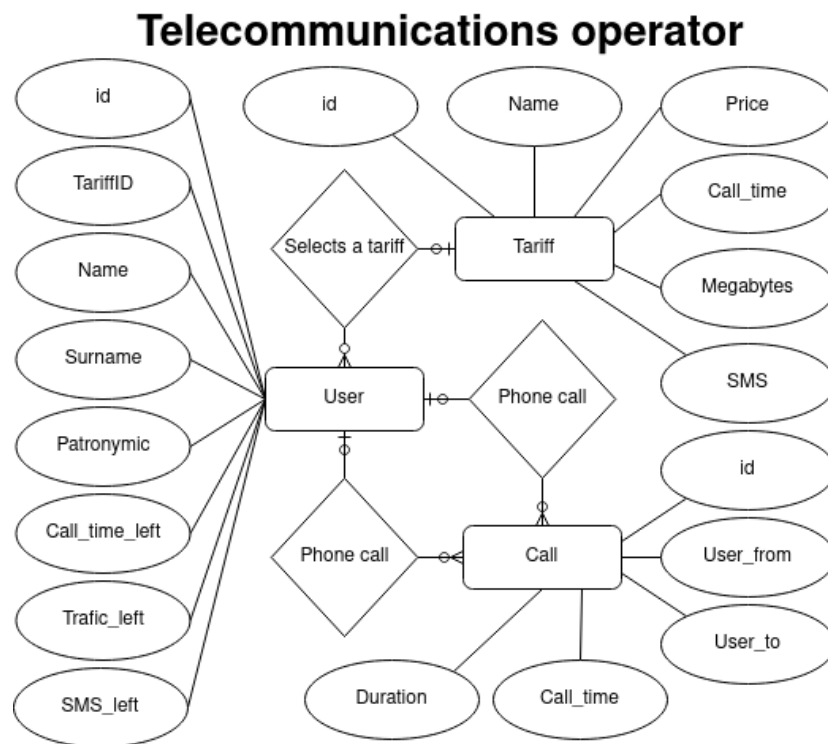


Рис. 1 - Концептуальна модель предметної області “Облік книгозбірні”

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

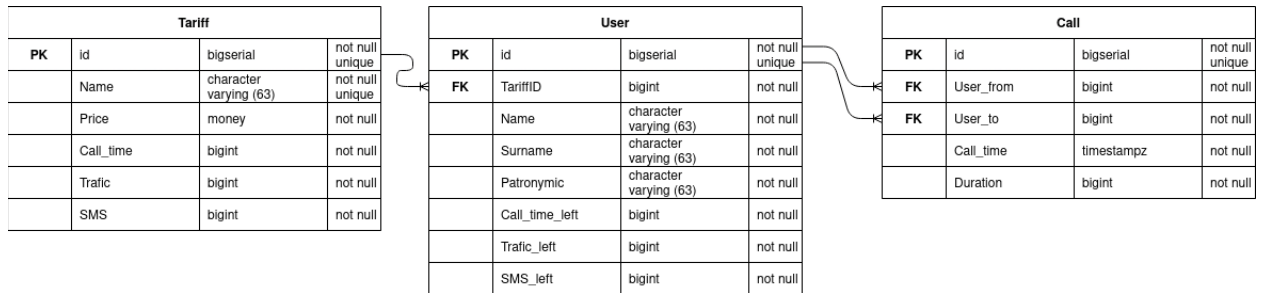


Рис. 2 – Логічна модель бази даних

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

```
class Tariff(Telecommunications_table):
    Name = peewee.CharField(max_length=63, null=False)
    Price = peewee.DecimalField(null=False)
    Call_time = peewee.BigIntegerField(null=False)
    Traffic = peewee.BigIntegerField(null=False)
    SMS = peewee.BigIntegerField(null=False)
```

```
class User(Telecommunications_table):
    TariffID = peewee.ForeignKeyField(Tariff, backref="used_by")
    Name = peewee.CharField(max_length=63, null=False)
    Surname = peewee.CharField(max_length=63, null=False)
    Patronymic = peewee.CharField(max_length=63, null=False)
    Address = peewee.CharField(max_length=255, null=False)
    Call_time_left = peewee.BigIntegerField(null=False)
    Traffic_left = peewee.BigIntegerField(null=False)
    SMS_left = peewee.BigIntegerField(null=False)
```

```
class Call(Telecommunications_table):
    User_from = peewee.ForeignKeyField(User, backref="called_to")
    User_to = peewee.ForeignKeyField(User, backref="called_by")
    Name = peewee.CharField(max_length=127, null=False)
    Call_time = peewee.DateTimeField(null=False)
    Duration = peewee.BigIntegerField(null=False, default=0)
```

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

Приклад отримання усіх даних з таблиці «User».

```
User.select()
```

Завдання 2

GIN

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

Створення таблиці та її заповнення:

```
DROP TABLE IF EXISTS "test_gin";

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

INSERT INTO "test_gin"("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:

```
lab3=# SELECT COUNT(*) FROM "test_gin" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_gin" WHERE "id" % 2 = 0 OR "test_text"::text LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_gin" WHERE "test_text"::text LIKE 'b%' GROUP BY "id" % 2;
count
-----
500000
(1 row)

Time: 79,765 ms
count
-----
500000
(1 row)

Time: 137,058 ms
count | sum
-----+-----
(0 rows)

Time: 150,075 ms
lab3=#
```

Сворюємо індекс:

```
DROP INDEX IF EXISTS "test_gin_test_text_index";

CREATE INDEX "test_gin_test_text_index" ON "test_gin" USING gin ("test_text");
```

Вибір даних з створеним індексом:

```
lab3=# SELECT COUNT(*) FROM "test_gin" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_gin" WHERE "id" % 2 = 0 OR "test_text"::text LIKE 'b%';
SELECT COUNT(*), SUM("id") FROM "test_gin" WHERE "test_text"::text LIKE 'b%' GROUP BY "id" % 2;
count
-----
500000
(1 row)

Time: 79,966 ms
count
-----
500000
(1 row)

Time: 135,493 ms
count | sum
-----+-----
(0 rows)

Time: 150,690 ms
lab3=#
```

Hash

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 Hash. У таблицю занесено 1000000 записів.

Створення таблиці та її заповнення:

```
DROP TABLE IF EXISTS "test_hash";

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

INSERT INTO "test_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;
```

Вибір даних без індексу:

```
lab3=# SELECT COUNT(*) FROM "test_hash" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_hash" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505';
SELECT COUNT(*), SUM("id") FROM "test_hash" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505' GROUP BY "id" % 2;
count
-----
500000
(1 row)

Time: 88,555 ms
count
-----
329826
(1 row)

Time: 80,095 ms
count |      sum
-----+-----
164799 | 82296186977
165027 | 82422677836
(2 rows)

Time: 134,189 ms
lab3=#
```

Сворюємо індекс:

```
DROP INDEX IF EXISTS "test_hash_test_time_index";

CREATE INDEX "test_hash_test_time_index" ON "test_hash" USING hash ("test_time");
```

Вибір даних з створеним індексом:

```
lab3=# SELECT COUNT(*) FROM "test_hash" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "test_hash" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505';
SELECT COUNT(*), SUM("id") FROM "test_hash" WHERE "test_time" >= '20200505' AND "test_time" <= '20210505' GROUP BY "id" % 2;
count
-----
500000
(1 row)

Time: 105,716 ms
count
-----
329826
(1 row)

Time: 69,664 ms
count |      sum
-----+-----
165027 | 82422677836
164799 | 82296186977
(2 rows)

Time: 164,235 ms
lab3=#
```

Завдання 3

Розробити тригер бази даних PostgreSQL.

Умова для тригера – after delete, 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:

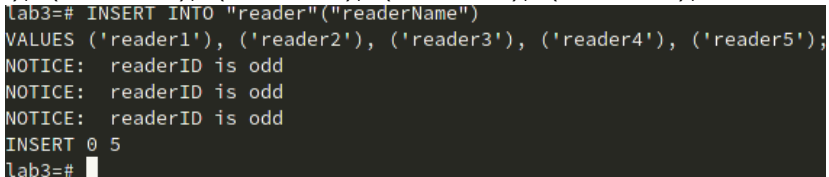
Тригер спрацьовує після оновлення у таблиці чи при додаванні нових рядків у таблицю reader. Якщо значення ідентифікатора запису, який дододється або оновлюється, парне, то цей запис заноситься у додаткову таблицю readerLog. Також, з кожного значення «readerName» видаляються символи «x» на початку і кінці.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

Занесемо тестові дані до таблиці:

```

INSERT INTO "reader"("readerName")
VALUES ('reader1'), ('reader2'), ('reader3'), ('reader4'), ('reader5');

```



```

lab3=# 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
lab3=#

```

```

lab3=# SELECT * FROM "reader";
SELECT * FROM "readerLog";

```

readerID	readerName
1	reader1
2	reader2
3	reader3
4	reader4
5	reader5

(5 rows)

id	readerLogID	readerLogName
1	2	yyreader2yy
2	4	yreader4y

(2 rows)

```

lab3=#

```


Оновимо дані в одному з рядків:

```
lab3=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" = 5;
NOTICE: readerID is odd
UPDATE 1
lab3=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
-----+-----
          1 | reader1
          2 | reader2
          3 | reader3
          4 | reader4
          5 | reader5Lx
(5 rows)

 id | readerLogID | readerLogName
-----+-----
  1 |           2 | yyyreader2yyy
  2 |           4 | yyreader4yy
(2 rows)

lab3=#
```

Оскільки id рядку який було оновлено є непарним числом, то це призвело до додавання до кожного значення «readerLogName» у таблиці readerLog строки “у” на початку і кінці.

Змінемо значення парного рядка:

```
lab3=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" = 4;
UPDATE 1
lab3=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
-----+-----
          1 | reader1
          2 | reader2
          3 | reader3
          5 | reader5Lx
          4 | reader4Lx
(5 rows)

 id | readerLogID | readerLogName
-----+-----
  1 |           2 | yyyreader2yyy
  2 |           4 | yyreader4yy
  3 |           4 | reader4L
(3 rows)

lab3=#
```

Як бачимо, при оновленні парного рядка його значення буде занесено у таблицю "readerLog" з прибраними символами «x» 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.

lab3=#	id	num	char
1	100	ABC	
2	200	BCA	
3	300	CAB	

(3 rows)

lab3=#	id	num	char
1	100	ABC	
2	200	BCA	
3	300	CAB	

(3 rows)

lab3=#

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

lab3=#	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=#

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).

lab3=#	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=#

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

```
lab3=# UPDATE "task4" SET "num" = "num" + 4;
UPDATE 3
lab3=#
lab3=#
lab3=#
lab3=#
lab3=#
lab3=#
lab3=#
lab3=#
lab3=#
```

id	num	char
1	100	ABC
2	200	BCA
3	300	CAB

```
(3 rows)

lab3=# UPDATE "task4" SET "num" = "num" + 4;
```

```
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=#
```

```
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=#
```

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.

```

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

main

1 branch

0 tags

Go to file

Add file

Code

About

No description, website, or topics provided.

Readme

0 stars

1 watching

0 forks

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

mykytenkoi initial release

01ad094 19 seconds ago 1 commit

GIN

initial release

19 seconds ago

Lab

initial release

19 seconds ago

.gitignore

initial release

19 seconds ago

README.md

initial release

19 seconds ago

Schema.png

initial release

19 seconds ago

lab3.py3

initial release

19 seconds ago

README.md

✎

Лабораторна робота №3 з дисципліни «Бази даних і засоби управління» Тема: «Засоби оптимізації роботи СУБД PostgreSQL» Студент групи KB-94 Микитенко Ілля

Логічна модель бази даних

Tariff			
PK	Id	bi	serial
	Name	character	not null
	Price	money	unique
	Call_time	bigint	not null
	Traffic	bigint	not null
	SMS	bigint	not null

User			
PK	Id	bi	serial
FK	TariffID	bigint	not null
	Name	character	not null
	Surname	character	not null
	Patronymic	character	not null
	Call_time_left	bigint	not null
	Traffic_left	bigint	not null
	SMS_left	bigint	not null

Call			
PK	Id	bi	serial
FK	User_from	bigint	not null
FK	User_to	bigint	not null
	Call_time	timestamp	not null
	Duration	bigint	not null

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