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 3 курсу

групи КВ-93

Федорова А.Ю.

Перевірив:

Павловський В.І.

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
26	*BTree, BRIN*	*before update, delete*

Посилання на репозиторій у GitHub з вихідним кодом програми та звітом:
https://github.com/nutasanchik/DB_Lab3

Завдання №1

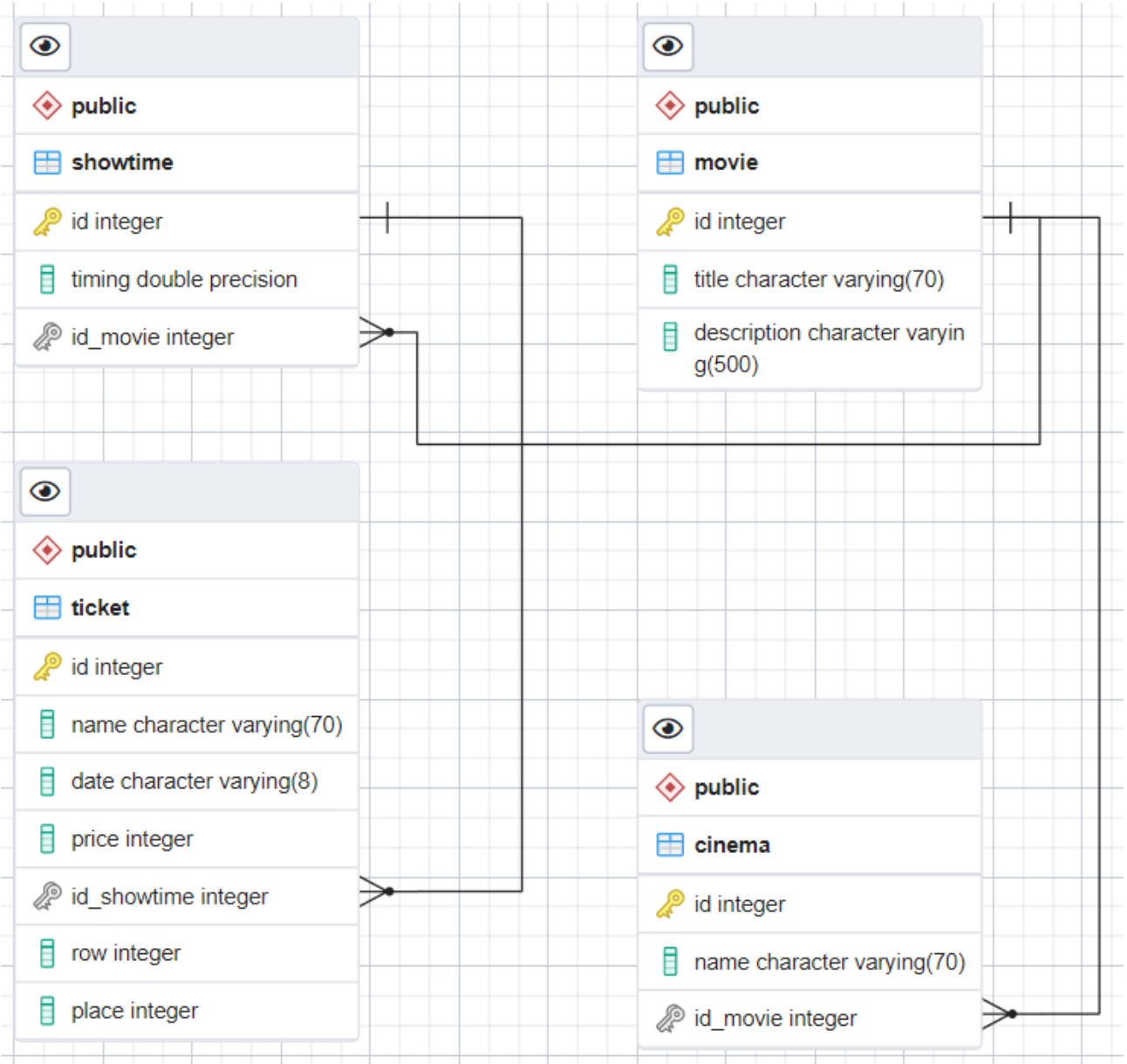


Рисунок 1 - Схема бази даних у pgAdmin 4

Таблиці бази даних у середовищі PgAdmin4

```
BEGIN;

CREATE TABLE IF NOT EXISTS public."Cinema"
(
    id integer NOT NULL,
    name character varying(70) NOT NULL,
    id_movie integer NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public."Movie"
(
    id integer NOT NULL,
```

```
    title character varying(70) NOT NULL,
    description character varying(500) NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public."Showtime"
(
    id integer NOT NULL,
    timing integer NOT NULL,
    id_movie integer NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS public."Ticket"
(
    id integer NOT NULL,
    name character varying(70) NOT NULL,
    date character varying(8) NOT NULL,
    price integer NOT NULL,
    id_showtime integer NOT NULL,
    "row" integer NOT NULL,
    place integer NOT NULL,
    PRIMARY KEY (id)
);

ALTER TABLE public."Cinema"
    ADD FOREIGN KEY (id_movie)
    REFERENCES public."Movie" (id)
    NOT VALID;


ALTER TABLE public."Showtime"
    ADD FOREIGN KEY (id_movie)
    REFERENCES public."Movie" (id)
    NOT VALID;


ALTER TABLE public."Ticket"
    ADD FOREIGN KEY (id_showtime)
    REFERENCES public."Showtime" (id)
    NOT VALID;

END;
```

Класи ORM у реалізованому модулі Model

```
class Cinema(Orders):
    __tablename__ = 'Cinema'
    id = Column(Integer, primary_key=True)
```

```
    name = Column(String)
    id_movie = Column(Integer, ForeignKey('Movie.id'))
    movies = relationship("Movie")

    def __init__(self, key, name, id_movie):
        self.id = key
        self.name = name
        self.id_movie = id_movie

    def __repr__(self):
        return "{:>10}{:>15}{:>10}" \
            .format(self.id, self.name, self.id_movie)


class Movie(Orders):
    __tablename__ = 'Movie'
    id = Column(Integer, primary_key=True)
    title = Column(String)
    description = Column(String)

    def __init__(self, key, title, description):
        self.id = key
        self.title = title
        self.description = description

    def __repr__(self):
        return "{:>10}{:>15}{:>50}" \
            .format(self.id, self.title, self.description)


class Showtime(Orders):
    __tablename__ = 'Showtime'
    id = Column(Integer, primary_key=True)
```

```
    timing = Column(Float)
    id_movie = Column(Integer, ForeignKey('Movie.id'))
    movies = relationship("Movie")

    def __init__(self, key, timing, id_movie):
        self.id = key
        self.timing = timing
        self.id_movie = id_movie

    def __repr__(self):
        return "{:>10}{:>15}{:>10}" \
            .format(self.id, self.timing, self.id_movie)


class Ticket(Orders):
    __tablename__ = 'Ticket'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    date = Column(String)
    price = Column(Integer)
    id_showtime = Column(Integer, ForeignKey('Showtime.id'))
    showtimes = relationship("Showtime")
    row = Column(Integer)
    place = Column(Integer)

    def __init__(self, key, name, date, price, id_showtime, row,
place):
        self.id = key
        self.name = name
        self.date = date
        self.price = price
        self.id_showtime = id_showtime
        self.row = row
```

```
        self.place = place


    def __repr__(self):
        return "{:>10}{:>15}{:>15}{:>10}{:>10}{:>10}{:>10}" \
            .format(self.id, self.name, self.date, self.price,
self.id_showtime, self.row, self.place)
```

Запити у вигляді ORM

Продемонструємо вставку, виучення, редагування даних на прикладі таблиці Cinema.

Початковий стан:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
```

Вставка запису:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py insert_record Cinema 6 Multiplex 2
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
Cinema table:
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
         6      Multiplex         2
```

Редагування запису:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py update_record Cinema 6 Multiplex 4
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
Cinema table:
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
         6      Multiplex         4
```

Видалення запису:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py delete_record Cinema 6
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
Cinema table:
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
```

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.

Запит на генерацію даних продемонструємо на прикладі таблиці Cinema.

Початковий стан:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
Cinema table:
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
```

Вставка 2-х випадково згенерованих записів:

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py generate_randomly Cinema 2
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py print_table Cinema
Cinema table:
        id           name  id_movie
         1      Multiplex         1
         2      Multiplex         3
         3      Multiplex         2
         4      Multiplex         4
         5      Multiplex         1
         6       gmnkvqkx         4
         7       akgscsel         2
```

Пошук за трьома атрибутами у двох таблицях, за трьома атрибутами у трьох таблицях, за чотирма атрибутами у чотирьох таблицях (виводяться відповідні записи з таблиці Showtime, Movie, Cinema віповідно):

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py search_records
specify the number of tables you`d like to search in: 2
search result:
        2           12.0        1
        5           18.0        3
```

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py search_records
specify the number of tables you`d like to search in: 3
search result:
        3          DunePaul Atreides, a brilliant and gifted young man born into a great destiny beyond his un
derstanding, must travel to the most dangerous planet in the universe to ensure the future of his family and his
 people. As malevolent forces explode into conflict over the planet's exclusive supply of the most precious reso
urce in existence, only those who can conquer their own fear will survive.
```

```
PS F:\Ann\Uni\Third_year\BD\Lab3> python main.py search_records
specify the number of tables you`d like to search in: 4
search result:
        1       Multiplex       1
        5       Multiplex       1
```

Завдання 2

BTree

Для дослідження індексу була створена таблиця, яка має дві колонки: числову та 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:

```
caricardo=# 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 row)

Time: 138,022 ms
 count
--------
 509681
(1 row)

Time: 164,118 ms
 count |    sum
-------+------------
  9583 | 4757592584
  9681 | 4794577027
(2 rows)

Time: 135,871 ms
caricardo=#
```

Створюємо індекс:

```
DROP INDEX IF EXISTS "test_btree_test_text_index";
```

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

Вибір даних з створеним індексом:

```
caricardo=# 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 row)

Time: 84,094 ms
 count
--------
 509681
(1 row)

Time: 124,855 ms
 count |    sum
-------+------------
  9583 | 4757592584
  9681 | 4794577027
(2 rows)

Time: 103,060 ms
```

BRIN

Для дослідження індексу була створена таблиця, яка має дві колонки: t_date типу timestamp without time zone (дата та час (без часового поясу)) і t_number типу integer. Колонка t_data 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 з створеним індексом:

Створюємо індекс:

```
DROP INDEX IF EXISTS "test_brin_test_time_index";
CREATE INDEX "test_brin_test_time_index" ON "test_brin" USING brin
("test_time");
```

Вибір даних з створеним індексом:

Завдання 3

Розробити тригер бази даних PostgreSQ

Умова для тригера – before update, delete.

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_delete_func() RETURNS TRIGGER as $$

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

begin
	IF old."readerID" % 2 = 0 THEN
		INSERT INTO "readerLog"("readerLogID", "readerLogName") VALUES
(old."readerID", old."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" = 'x' ||
row_Log."readerLogName" || 'x' WHERE "id" = row_log."id";
		END LOOP;
		RETURN NEW;
	END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER "test_trigger"
BEFORE UPDATE OR DELETE ON "reader"
FOR EACH ROW
EXECUTE procedure update_delete_func();
```

Принцип роботи:

Тригер спрацьовує після видалення з таблиці чи при оновлені у таблиці reader. Якщо значення ідентифікатора запису, який видаляється 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 “x” на початку і кінці.

Занесемо тестові дані до таблиці:

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

```
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+------------
        1 | reader1
        2 | reader2
        3 | reader3
        4 | reader4
        5 | reader5
(5 rows)

Time: 0,331 ms
 id | readerLogID | readerLogName
----+-------------+---------------
(0 rows)

Time: 0,229 ms
```

Оновимо дані в одному з рядків:

```
caricardo=# UPDATE "reader" SET "readerName" = "readerName" || 'LEL' WHERE "readerID" = 5;
NOTICE:  readerID is odd
UPDATE 1
Time: 151,930 ms
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+------------
        1 | reader1
        2 | reader2
        3 | reader3
        4 | reader4
        5 | reader5LEL
(5 rows)

Time: 0,286 ms
 id | readerLogID | readerLogName
----+-------------+---------------
(0 rows)

Time: 0,074 ms
```

Оскільки id рядку який було оновлено є непарним числом, та оскільки таблиця "readerLog" є пустою, то отримано просте оновлення запису.

Змінимо значення парного рядка:

```
caricardo=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" = 4;
UPDATE 1
Time: 77,198 ms
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+------------
        1 | reader1
        2 | reader2
        3 | reader3
        5 | reader5LEL
        4 | reader4Lx
(5 rows)

Time: 0,301 ms
 id | readerLogID | readerLogName
----+-------------+---------------
  1 |           4 | reader4
(1 row)

Time: 0,084 ms
```

Як бачимо, перед оновленням рядка його значення буде занесено у таблицю "readerLog".

Виконаємо такий самий запит ще раз:

```
caricardo=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" = 4;
UPDATE 1
Time: 14,053 ms
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+-------------
        1 | reader1
        2 | reader2
        3 | reader3
        5 | reader5LEL
        4 | reader4LxLx
(5 rows)

Time: 0,289 ms
 id | readerLogID | readerLogName
----+-------------+---------------
  1 |           4 | reader4
  2 |           4 | reader4L
(2 rows)

Time: 0,086 ms
```

Як бачимо, перед оновленням рядка його значення з видаленими з початку та кінця символами «x» буде занесено у таблицю "readerLog".

Оновимо дані в одному з непарних рядків:

```
caricardo=# UPDATE "reader" SET "readerName" = "readerName" || 'Lx' WHERE "readerID" = 1;
NOTICE:  readerID is odd
UPDATE 1
Time: 82,207 ms
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+------------
        2 | reader2
        3 | reader3
        5 | reader5LEL
        4 | reader4LxLx
        1 | reader1Lx
(5 rows)

Time: 0,297 ms
 id | readerLogID | readerLogName
----+-------------+---------------
  1 |           4 | xreader4x
  2 |           4 | xreader4Lx
(2 rows)

Time: 0,097 ms
```

Як бачимо, значення непарних рядків не заносятся до таблиці "readerLog", проте, зміна або видалення непарного рядка призводить до того, що до всіх значень ’’readerLog"."readcrLogNamc" у початок та в кінець додаються символи «х».

Видалення рядку:

```
caricardo=# DELETE FROM "reader" WHERE "readerID" = 3;
NOTICE:  readerID is odd
DELETE 0
Time: 15,102 ms
caricardo=# SELECT * FROM "reader";
SELECT * FROM "readerLog";
 readerID | readerName
----------+------------
        2 | reader2
        3 | reader3
        5 | reader5LEL
        4 | reader4LxLx
        1 | reader1Lx
(5 rows)

Time: 0,343 ms
 id | readerLogID | readerLogName
----+-------------+---------------
  1 |           4 | xxreader4xx
  2 |           4 | xxreader4Lxx
(2 rows)

Time: 0,091 ms
```

Як бачимо, операція видалення також призводить до спрацьовування тригера. Також можна зазначити, що тригер може відміняти операцію видалення для рядка, в залежності від того яке значення повертається: «old» для видалення, «new» для відміни видалення.

Завдання 4

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

Самі транзакції особливих пояснень не вимагають, транзакція — це N (N>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.

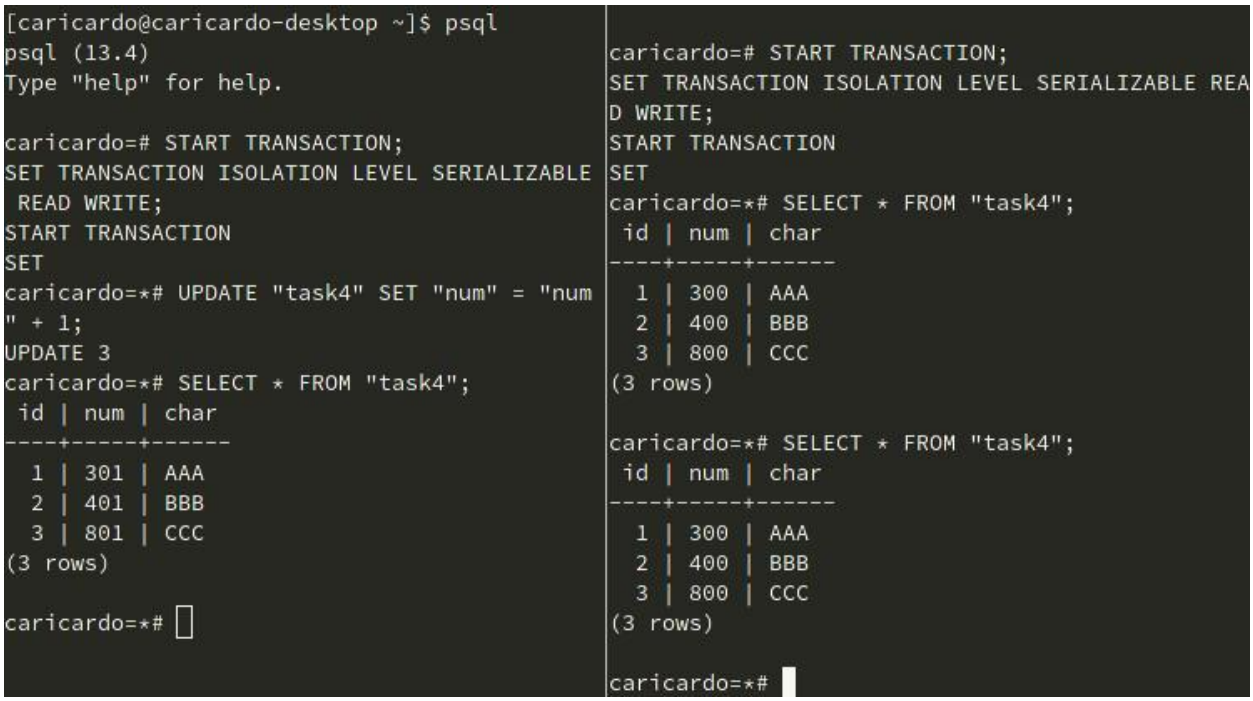
При паралельному виконанні транзакцій можливі виникнення таких проблем:

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.

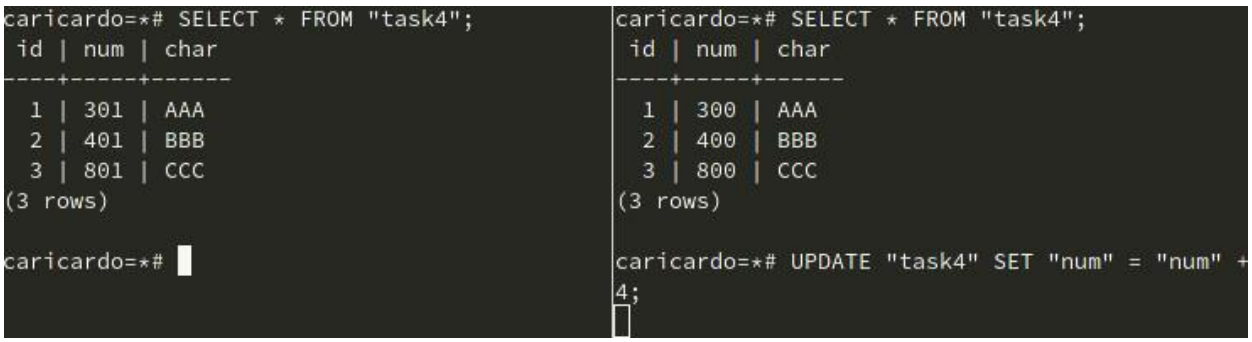
Стандарт SQL-92 визначає наступні рівні ізоляції:

1. Serializable (впорядкованість)

   Найбільш високий рівень ізольованості; 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 ізольовано.



Тепер при оновлені даних в T2 бачимо, що T2 блокується поки T1 не зафіксує зміни або не відмінить їх.

```
UPDATE 3
caricardo=*# SELECT * FROM "task
4";
 id | num | char
----+-----+------
  1 | 301 | AAA
  2 | 401 | BBB
  3 | 801 | CCC
(3 rows)

caricardo=*# COMMIT;
COMMIT
caricardo=#
```

```
caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*# UPDATE "task4" SET "num" = "num" + 4;
ERROR:  could not serialize access due to concurrent update
caricardo=!# ROLLBACK;
ROLLBACK
caricardo=#
```

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).

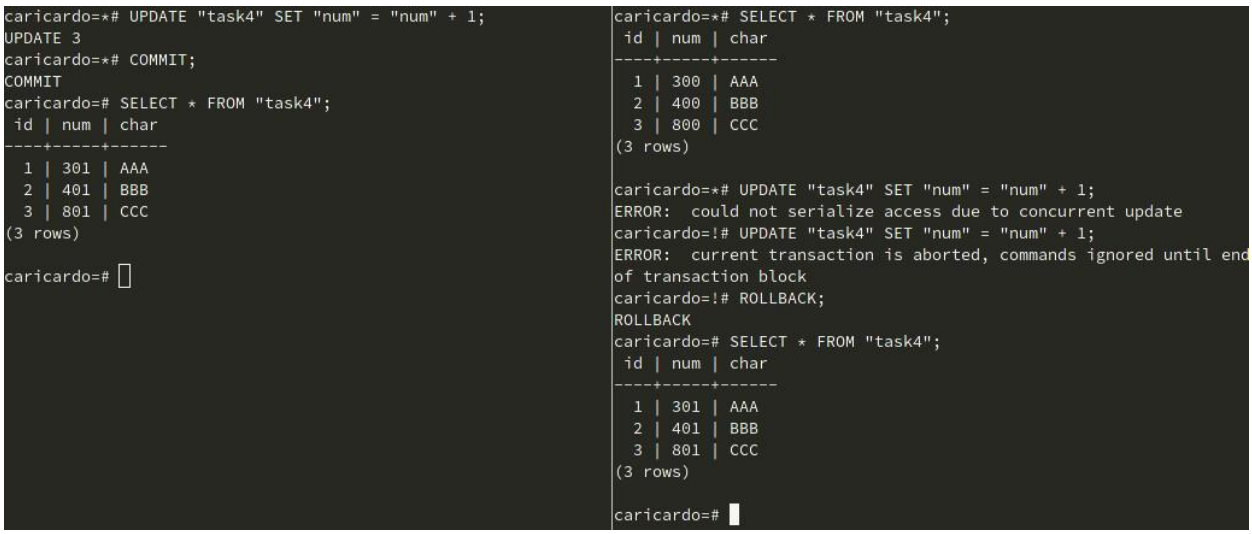
```
[caricardo@caricardo-desktop ~]$ psql
psql (13.4)
Type "help" for help.

caricardo=# START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
START TRANSACTION
SET
caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*# UPDATE "task4" SET "num" = "num" + 1;
```

```
[caricardo@caricardo-desktop lab3]$ psql
psql (13.4)
Type "help" for help.

caricardo=# START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
START TRANSACTION
SET
caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*#
```

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

```
caricardo=*# UPDATE "task4" SET "num" = "num" + 1;
UPDATE 3
caricardo=*#
```

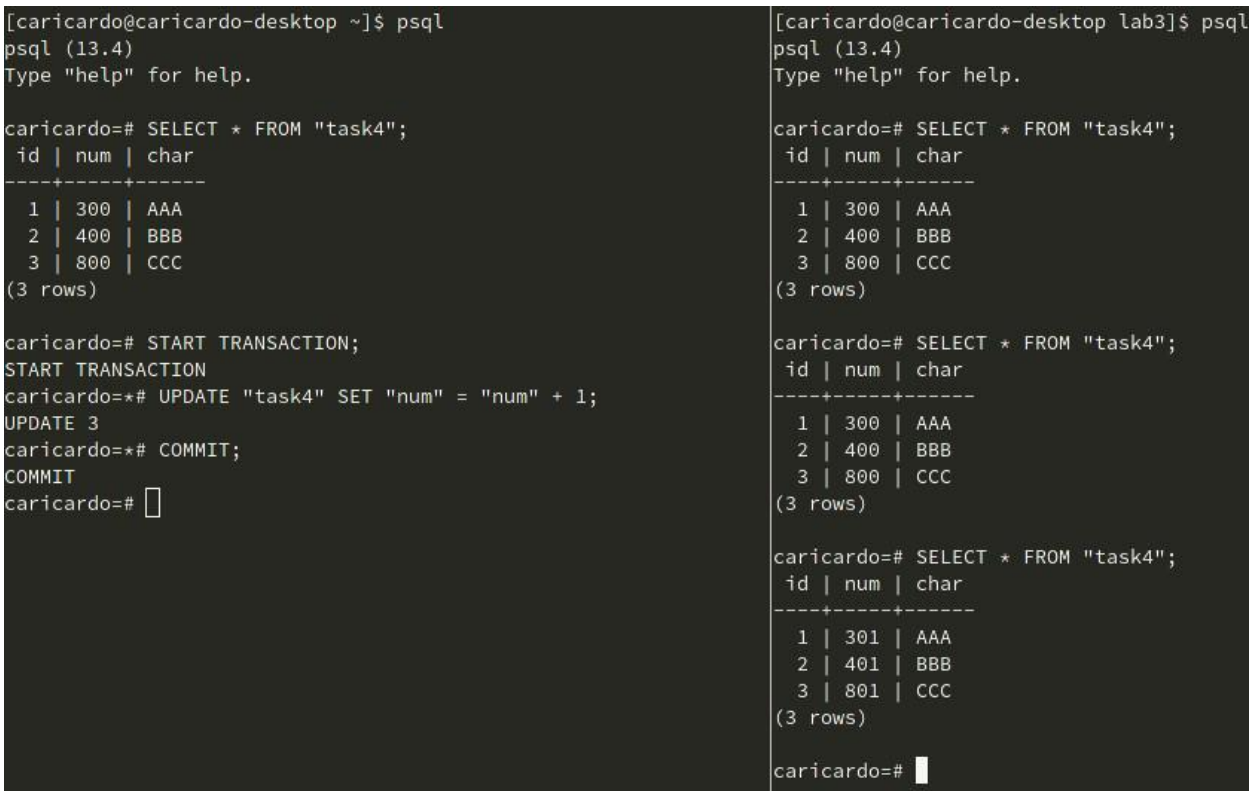
```
caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*# SELECT * FROM "task4";
 id | num | char
----+-----+------
  1 | 300 | AAA
  2 | 400 | BBB
  3 | 800 | CCC
(3 rows)

caricardo=*# UPDATE "task4" SET "num" = "num" + 1;
```



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.



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 транзакцию. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.