

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-92
Рукавіцин К. В.
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 проєкції (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 24

У другому завданні проаналізувати індекси BTree, BRIN.

Умова для тригера – before update, delete.

Завдання 1

Інформація про модель та структуру бази даних

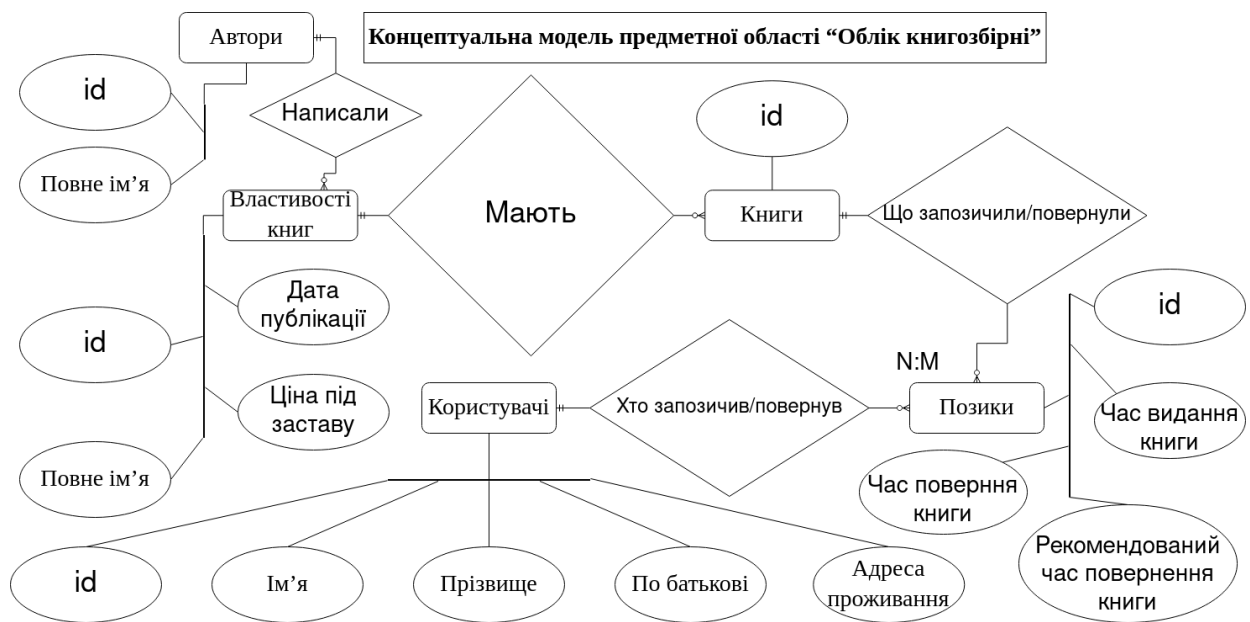


Рис. 1 - Концептуальна модель предметної області “Облік книгозбірні”

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

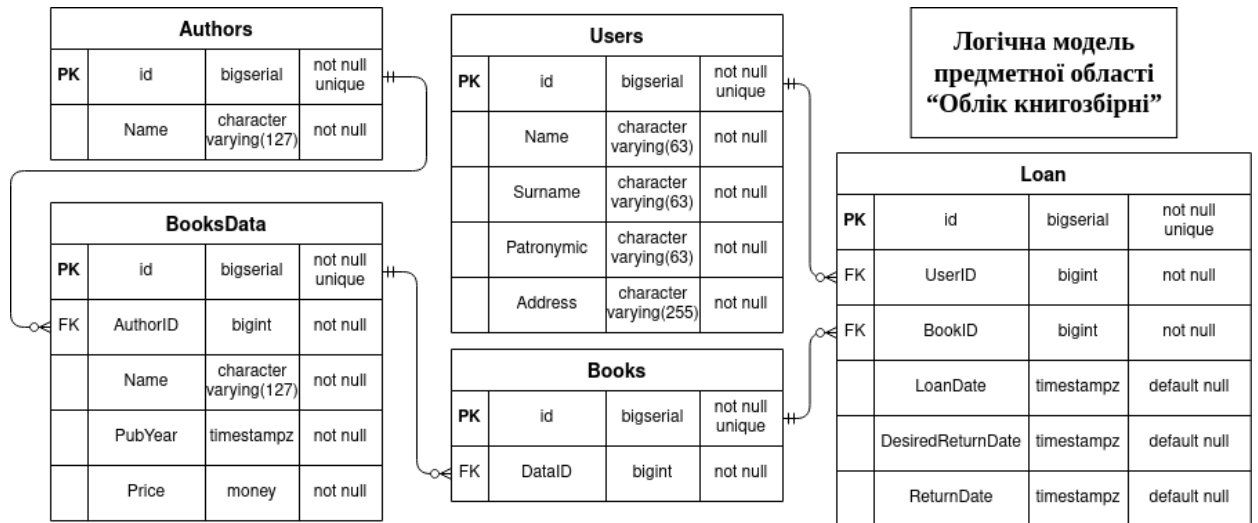


Рис. 2 – Логічна модель бази даних

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 Library_table(peewee.Model): # (SchemaTable):
    class Meta:
        database = database_proxy
        schema = f"Library"

class Authors(Library_table):
    Name = peewee.CharField(max_length=127, null=False)

class BooksData(Library_table):
    AuthorID = peewee.ForeignKeyField(Authors, backref="publications")
    Name = peewee.CharField(max_length=127, null=False)
    PubYear = peewee.DateTimeField(null=False)
    Price = peewee.DecimalField(null=False)

class Books(Library_table):
    DataID = peewee.ForeignKeyField(BooksData, backref="books")

class Users(Library_table):
    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)

class Loan(Library_table):
    UserID = peewee.ForeignKeyField(Users, backref="loans")
    BookID = peewee.ForeignKeyField(Books, backref="loaned")
    LoanDate = peewee.DateTimeField()
    ReturnDate = peewee.DateTimeField()
    DesiredReturnDate = peewee.DateTimeField()
```

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 «Authors».

```
Authors.select()
```

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:

```
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=#
```

Сворюємо 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_data 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 без індексу:

```
caricardo=# 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 row)

Time: 99,141 ms
count
-----
329977
(1 row)

Time: 79,558 ms
count | sum
-----+-----
164936 | 82310274166
165041 | 82460391352
(2 rows)

Time: 105,937 ms
caricardo=#
```

Сворюємо 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_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 row)

Time: 77,745 ms
count
-----
329977
(1 row)

Time: 66,911 ms
count |      sum
-----+-----
165041 | 82460391352
164936 | 82310274166
(2 rows)

Time: 160,372 ms
```

Завдання 3

Розробити тригер бази даних PostgreSQL.

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 оновленні 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.

Змінемо 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 заносяться до таблиці "readerLog", проте, зміна або видалення непарного рядка призводить до того, що до всіх значень "readerLog"."readerLogName" у початок та в кінець додаються символи «x».

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 відмінити 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 ($N \geq 1$) 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 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).

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

caricardo=# START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
START TRANSACTION
SET
caricardo=# UPDATE "task4" SET "num" = "num" + 1;
UPDATE 3
caricardo=# SELECT * FROM "task4";
 id | num | char
-----+-----+-----
  1 | 301 | AAA
  2 | 401 | BBB
  3 | 801 | CCC
(3 rows)

caricardo=#
```

```
caricardo=# START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE 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=#
```

Як бачимо, дані у транзакціях ізольовано.

```
caricardo=# SELECT * FROM "task4";
 id | num | char
-----+-----+-----
  1 | 301 | AAA
  2 | 401 | BBB
  3 | 801 | CCC
(3 rows)

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;

```

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

```
UPDATE 3
caricardo=# SELECT * FROM "task4";
 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).

<pre>[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; █</pre>	<pre>[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=# █</pre>
---	--

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

<pre>caricardo=# UPDATE "task4" SET "num" = "num" + 1; UPDATE 3 caricardo=# █</pre>	<pre>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;</pre>
---	--

<pre>caricardo=# UPDATE "task4" SET "num" = "num" + 1; UPDATE 3 caricardo=# COMMIT; COMMIT caricardo=# SELECT * FROM "task4"; id num char -----+-----+----- 1 301 AAA 2 401 BBB 3 801 CCC (3 rows) caricardo=# █</pre>	<pre>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; ERROR: could not serialize access due to concurrent update caricardo=# UPDATE "task4" SET "num" = "num" + 1; ERROR: current transaction is aborted, commands ignored until end of transaction block caricardo=# ROLLBACK; ROLLBACK caricardo=# SELECT * FROM "task4"; id num char -----+-----+----- 1 301 AAA 2 401 BBB 3 801 CCC (3 rows) caricardo=# █</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.

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

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

caricardo=# START TRANSACTION;
START TRANSACTION
caricardo=# UPDATE "task4" SET "num" = "num" + 1;
UPDATE 3
caricardo=# COMMIT;
COMMIT
caricardo=#
```

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

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=# SELECT * FROM "task4";
 id | num | char
-----+-----+-----
  1 | 301 | AAA
  2 | 401 | BBB
  3 | 801 | CCC
(3 rows)

caricardo=#
```

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