



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”

Виконав:
студент III курсу
групи KB-91
Селетков В. Р.
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 |
|------------|-------------------|-----------------------------|
| 18 | <i>BTree, GIN</i> | <i>after update, insert</i> |

Посилання на репозиторій у GitHub з вихідним кодом програми та прикладеним звітом: <https://github.com/vladsel/database>

Відомості про предметну галузь з лабораторної роботи №1

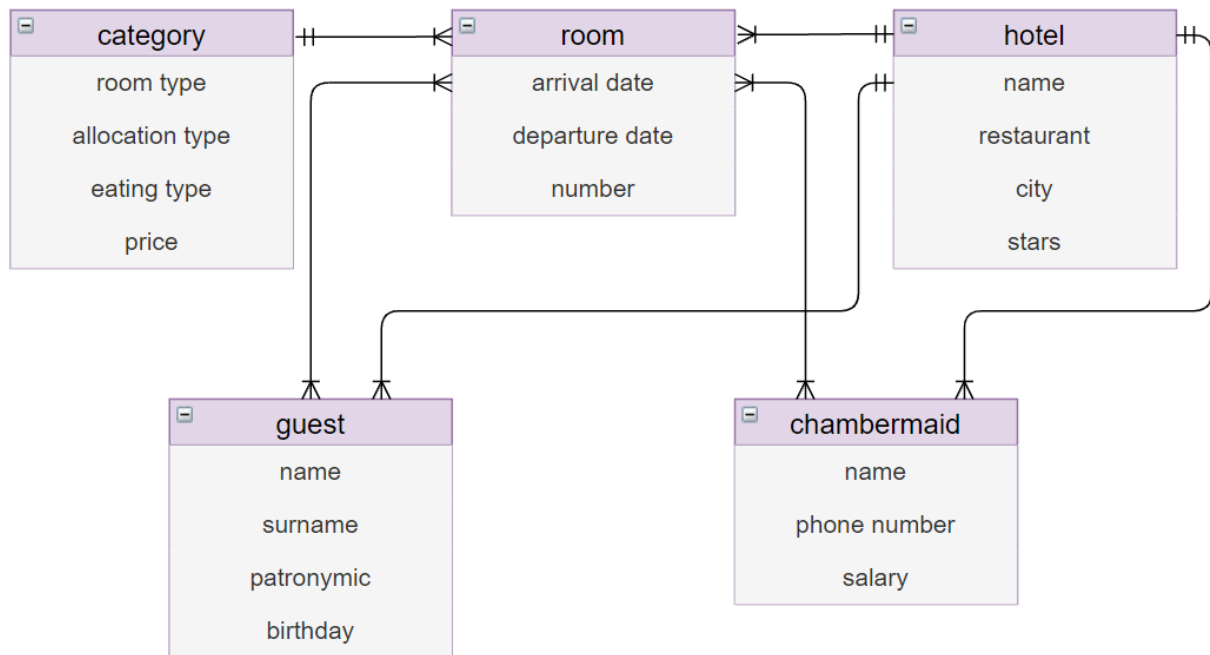


Рисунок 1 - ER-діаграма побудована за нотацією “Пташиної лапки (Crow’s foot)”, задана ER-діаграма була побудована у додатку draw.io

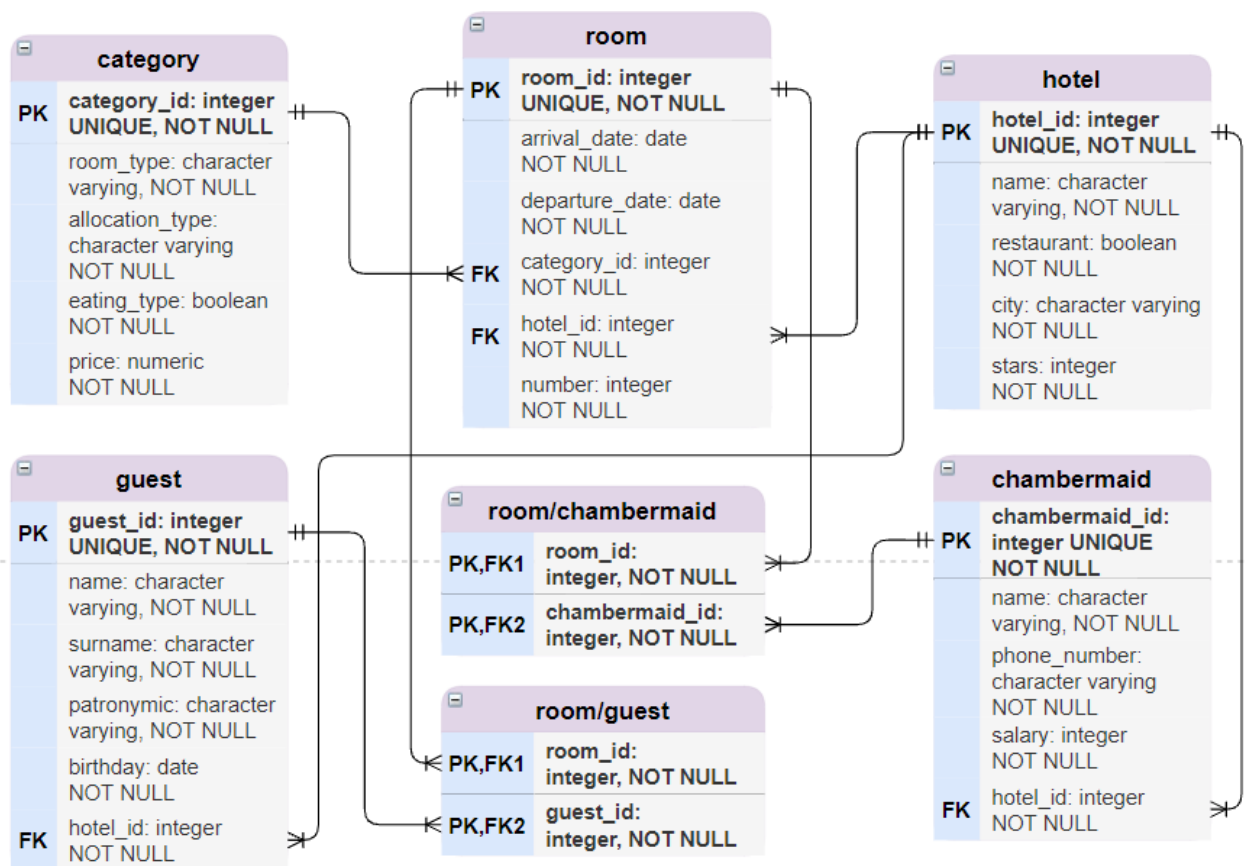


Рисунок 2 - Схема бази даних, побудовано у додатку draw.io

Таблиця 1 - Опис структури БД.

| Відношення | Атрибут | Тип атрибуту |
|--|--|--|
| hotel – містить дані про готель | hotel_id – унікальний ідентифікатор name – назва готелю restaurant – наявність ресторану city – місто stars – кількість зірок | integer (числовий) character varying (рядок) boolean (булевий) character varying (рядок) integer (числовий) |
| category – містить дані про категорію номеру у готелі | category_id – унікальний ідентифікатор room_type – тип номеру allocation_type – тип розселення в номері eating_type – наявність харчування price – ціна | integer (числовий) character varying (рядок) character varying (рядок) boolean (булевий) numeric (фіксований) |
| guest – містить дані про постояльців готелю | guest_id – унікальний ідентифікатор name – ім'я surname – прізвище patronymic – по батькові birthday – день народження hotel_id – ідентифікатор готелю | integer (числовий) character varying (рядок) character varying (рядок) character varying (рядок) date (дата) integer (числовий) |
| room – містить дані щодо номеру | room_id - унікальний ідентифікатор arrival_date – дата заселення departure_date – дата виселення category_id – ідентифікатор категорії hotel_id – ідентифікатор готелю number – номер кімнати | integer (числовий) date (дата) date (дата) integer (числовий) integer (числовий) integer (числовий) |
| chambermaid – містить дані про покоївок готелю | chambermaid_id – унікальний ідентифікатор name – ім'я phone – номер телефону salary – заробітня плата hotel_id – ідентифікатор готелю | integer (числовий) character varying (рядок) character varying (рядок) integer (числовий) integer (числовий) |
| room/ chambermaid - відношення покоївок до кімнат | room_id – ідентифікатор номера chambermaid_id – ідентифікатор покоївки | integer (числовий) integer (числовий) |

| | | |
|---|--|--|
| room/guest - відношення постояльців до кімнат | room_id – ідентифікатор номера chambermaid_id – ідентифікатор постояльця | integer (числовий) integer (числовий) |
|---|--|--|

У Обраній базі даних «Готель» можна виділити наступні таблиці: загальні відомості про готель (hotel), тип заданого номера (room), категорія номера (category), загальні відомості про постояльця (guest), інформація про покоївку (chambermaid), відношення покоївок до кімнат (room/chambermaid), відношення постояльців до кімнат (room/guest).

Стовпці заданих таблиць:

1. hotel: hotel_id, name, restaurant, city, stars.
2. room: room_id, arrival date, departure date, category_id, hotel_id, number.
3. category: category_id, room type, allocation type, eating type, price.
4. guest: guest_id, name, surname, patronymic, birthday, hotel_id.
5. chambermaid: chambermaid_id, name, phone number, salary, hotel_id.

Завдання №1

Класи ORM у реалізованому модулі Model

```
class Category(base):
    __tablename__ = 'category'
    category_id = Column(Integer, primary_key=True, nullable=False)
    room_type = Column(String(30), nullable=False)
    allocation_type = Column(String(30), nullable=False)
    eating_type = Column(Boolean, nullable=False)

    def __init__(self, room_type, allocation_type, eating_type,
category_id=-1):
        self.room_type = room_type
        self.allocation_type = allocation_type
        self.eating_type = eating_type
        if category_id != -1:
            self.category_id = category_id

    def __repr__(self):
        return "{:^12}{:^15}{:^20}{:^15}".format(self.category_id,
self.room_type, self.allocation_type, self.eating_type)

    def __str__(self):
        return
f"{'category_id':^12}{ 'room_type':^15}{ 'allocation_type':^20}{ 'eating_
type':^15}"
        # return f"""category_id = {self.category_id}, room_type =
{self.room_type}, """ \
        # f"""allocation_type = {self.allocation_type},
eating_type = {self.eating_type}"""

class Hotel(base):
    __tablename__ = 'hotel'
    hotel_id = Column(Integer, primary_key=True, nullable=False)
    name = Column(String(20), nullable=False)
    restaurant = Column(Boolean, nullable=False)
    city = Column(String(25), nullable=False)
    star = Column(Integer, nullable=False)

    def __init__(self, name, restaurant, city, star, hotel_id=-1):
        self.name = name
        self.restaurant = restaurant
        self.city = city
        self.star = star
        if hotel_id != -1:
            self.hotel_id = hotel_id

    def __repr__(self):
```

```

        return "{:^10}{:^15}{:^10}{:^15}{:^5}".format(self.hotel_id,
self.name, self.restaurant, self.city, self.star)

    def __str__(self):
        return
f"{'hotel_id':^10}{ 'name':^15}{ 'restaurant':^10}{ 'city':^15}{ 'star':^5}
}"
        # return f""""hotel_id = {self.hotel_id}, name = {self.name},
"""" \
        #         f""""restaurant = {self.restaurant}, city =
{self.city}, star = {self.star}""""

class Chambermaid(base):
    __tablename__ = 'chambermaid'
    chambermaid_id = Column(Integer, primary_key=True, nullable=False)
    name = Column(String(100), nullable=False)
    phone_number = Column(String(20), nullable=False)
    salary = Column(Integer, nullable=False)
    hotel_id = Column(Integer, ForeignKey('hotel.hotel_id'),
nullable=False)

    def __init__(self, name, phone_number, salary, hotel_id,
chambermaid_id=-1):
        self.name = name
        self.phone_number = phone_number
        self.salary = salary
        self.hotel_id = hotel_id
        if chambermaid_id != -1:
            self.chambermaid_id = chambermaid_id

    def __repr__(self):
        return
"{:^15}{:^15}{:^15}{:^8}{:^10}".format(self.chambermaid_id, self.name,
self.phone_number, self.salary, self.hotel_id)

    def __str__(self):
        return
f"{'chambermaid_id':^15}{ 'name':^15}{ 'phone_number':^15}{ 'salary':^8}{
'hotel_id':^10}"
        # return f""""chambermaid_id = {self.chambermaid_id}, name =
{self.name}, """" \
        #         f""""phone_number = {self.phone_number}, salary =
{self.salary}, hotel_id = {self.hotel_id}""""

class Guest(base):
    __tablename__ = 'guest'
    guest_id = Column(Integer, primary_key=True, nullable=False)
    name = Column(String(100), nullable=False)
    surname = Column(String(100), nullable=False)

```

```

    patronymic = Column(String(100), nullable=False)
    birthday = Column(DATETIME, nullable=False)
    hotel_id = Column(Integer, ForeignKey('hotel.hotel_id'),
nullable=False)

    def __init__(self, name, surname, patronymic, birthday, hotel_id,
guest_id=-1):
        self.name = name
        self.surname = surname
        self.patronymic = patronymic
        self.birthday = birthday
        self.hotel_id = hotel_id
        if guest_id != -1:
            self.guest_id = guest_id

    def __repr__(self):
        return
"{:^10}{:^15}{:^15}{:^15}\t{ }{:^12}".format(self.guest_id, self.name,
self.surname, self.patronymic, self.birthday, self.hotel_id)

    def __str__(self):
        return
f"{'guest_id':^10}{ 'name':^15}{ 'surname':^15}{ 'patronymic':^15}{ 'birth
day':^12}{ 'hotel_id':^10}"
        # return f"" "guest_id = {self.guest_id}, name = {self.name},
surname {self.surname}, "" " \
        #         f"" "patronymic = {self.patronymic}, birthday =
{self.birthday}, hotel_id = {self.hotel_id}"" "

class Room(base):
    __tablename__ = 'room'
    room_id = Column(Integer, primary_key=True, nullable=False)
    arrival_date = Column(DATETIME, nullable=False)
    departure_date = Column(DATETIME, nullable=False)
    category_id = Column(Integer, ForeignKey('category.category_id'),
nullable=False)
    hotel_id = Column(Integer, ForeignKey('hotel.hotel_id'),
nullable=False)
    number = Column(Integer, nullable=False)
    price = Column(Float, nullable=False)

    def __init__(self, arrival_date, departure_date, category_id,
hotel_id, number, price, room_id=-1):
        self.arrival_date = arrival_date
        self.departure_date = departure_date
        self.category_id = category_id
        self.hotel_id = hotel_id
        self.number = number
        self.price = price
        if room_id != -1:

```



```

        self.room_id = room_id

    def __repr__(self):
        return "{:^10}\t
{} \t\t{}{:^22}{:^6}{:^8}{:^10}".format(self.room_id,
self.arrival_date, self.departure_date,

self.category_id, self.hotel_id, self.number, self.price)

    def __str__(self):
        return
f"{'room_id':^10}{ 'arrival_date':^18}{ 'departure_date':^18}{ 'category_
id':^15}{ 'hotel_id':^10}{ 'number':^8}{ 'price':^8}"
        # return f""""room_id = {self.room_id}, arrival_date =
{self.arrival_date}, departure_date {self.departure_date}, """" \
        #         f""""category_id = {self.category_id}, hotel_id =
{self.hotel_id}, hotel_id = {self.number}, price = {self.price}""""

class RoomChambermaid(base):
    __tablename__ = 'room/chambermaid'
    room_id = Column(Integer, ForeignKey('room.room_id'),
primary_key=True, nullable=False)
    chambermaid_id = Column(Integer,
ForeignKey('chambermaid.chambermaid_id'), primary_key=True,
nullable=False)

    def __init__(self, room_id, chambermaid_id):
        self.room_id = room_id
        self.chambermaid_id = chambermaid_id

    def __repr__(self):
        return "{:^10}{:^18}".format(self.room_id,
self.chambermaid_id)

    def __str__(self):
        return f"{'room_id':^10}{ 'chambermaid_id':^18}"
        # return f""""room_id = {self.room_id}, chambermaid_id =
{self.chambermaid_id}""""

class RoomGuest(base):
    __tablename__ = 'room/guest'
    room_id = Column(Integer, ForeignKey('room.room_id'),
primary_key=True, nullable=False)
    guest_id = Column(Integer, ForeignKey('guest.guest_id'),
primary_key=True, nullable=False)

    def __init__(self, room_id, guest_id):
        self.room_id = room_id
        self.guest_id = guest_id

```

```
def __repr__(self):
    return "{:^10}{:^10}".format(self.room_id, self.guest_id)

def __str__(self):
    return f"{'room_id':^10}{'guest_id':^10}"
    # return f""""room_id = {self.room_id}, guest_id =
    {self.guest_id}"""
```

Запити у вигляді ORM

Продемонструємо вставку, вилучення, редагування даних на прикладі таблиці **guest**.

Початкові вхідні дані:

```
1. INSERT data in table
2. EDIT data in table
3. DELETE data from table
4. PRINT rows
5. GENERATE random data
6. SEARCH data from tables
0. Exit

        Choose an option 1-6 or 0: 4

1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3
Input quantity of rows to print: 10
```

| guest_id | name | surname | patronymic | birthday | hotel_id |
|----------|-----------|-------------|---------------|------------|----------|
| 8 | e5ac7a31d | d936fae2244 | 974c7a82b1bef | 1980-09-04 | 1 |
| 9 | 93948a951 | ee83a73cc03 | 5b37bf6873545 | 1961-03-15 | 2 |
| 10 | 38748db2f | eb6eb753f59 | b89354ce2ef1c | 1965-04-05 | 2 |
| 11 | 808f75fc0 | bbee2914935 | dbed5ce97b720 | 1972-10-21 | 3 |
| 12 | d3fcc917e | 70f2d89f245 | c4ad55fb240fa | 1975-12-19 | 3 |
| 13 | f4a51766b | b102b9fe3ee | cb0ec1b405b2e | 1970-08-09 | 2 |
| 14 | 683a84c41 | 2a742052d7b | 669914424bd1f | 1969-08-02 | 3 |

Для перевірки роботи розглянемо запити на видалення даних з даної таблиці **guest**. Спробуємо видалити рядок з **guest_id = 11**. Нижче наведене виконання заданих запитів.

1. INSERT data in table
2. EDIT data in table
3. DELETE data from table
4. PRINT rows
5. GENERATE random data
6. SEARCH data from tables
0. Exit

Choose an option 1-6 or 0: 3

1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3

Enter id of row that you want to DELETE

'p' => print rows

'r' => return to menu

11

The row DELETED successfully

Choose an option 1-6 or 0: 4

1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3

Input quantity of rows to print: 10

| guest_id | name | surname | patronymic | birthday | hotel_id |
|----------|-----------|-------------|---------------|------------|----------|
| 8 | e5ac7a31d | d936fae2244 | 974c7a82b1bef | 1980-09-04 | 1 |
| 9 | 93948a951 | ee83a73cc03 | 5b37bf6873545 | 1961-03-15 | 2 |
| 10 | 38748db2f | eb6eb753f59 | b89354ce2ef1c | 1965-04-05 | 2 |
| 12 | d3fcc917e | 70f2d89f245 | c4ad55fb240fa | 1975-12-19 | 3 |
| 13 | f4a51766b | b102b9fe3ee | cb0ec1b405b2e | 1970-08-09 | 2 |
| 14 | 683a84c41 | 2a742052d7b | 669914424bd1f | 1969-08-02 | 3 |

Для перевірки роботи розглянемо запити на вставку даних до даної таблиці **guest**. Спробуємо додати рядок з деякими даними, даний рядок повинен мати **guest_id = 15**. Нижче наведено виконання заданих запитів.

```
1. INSERT data in table
2. EDIT data in table
3. DELETE data from table
4. PRINT rows
5. GENERATE random data
6. SEARCH data from tables
0. Exit

    Choose an option 1-6 or 0: 1

1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3
Input data separated by comma
Table: guest. Input: name->text, surname->text, patronymic->text, birthday->date, hotel_id->int
Alex,Patron,Orange,1980-09-18,5
Successfully INSERTED data into table:
    guest_id      name      surname      patronymic      birthday      hotel_id
Data INSERTED successfully
```

```
1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3
Input quantity of rows to print: 10

    guest_id      name      surname      patronymic      birthday      hotel_id
      8      e5ac7a31d      d936fae2244      974c7a82b1bef      1980-09-04      1
      9      93948a951      ee83a73cc03      5b37bf6873545      1961-03-15      2
     10      38748db2f      eb6eb753f59      b89354ce2ef1c      1965-04-05      2
     11      808f75fc0      bbee2914935      dbed5ce97b720      1972-10-21      3
     12      d3fcc917e      70f2d89f245      c4ad55fb240fa      1975-12-19      3
     13      f4a51766b      b102b9fe3ee      cb0ec1b405b2e      1970-08-09      2
     14      683a84c41      2a742052d7b      669914424bd1f      1969-08-02      3
     15      Alex      Patron      Orange      1980-09-18      5
```

Для перевірки роботи розглянемо запити на редагування даних до даної таблиці **guest**. Спробуємо відредагувати рядок з **guest_id = 9**. Нижче наведене виконання заданих запитів.

```
Choose the table: 3
Enter id of row that you want to UPDATE
'p' => print rows
'r' => return to menu
9
  guest_id      name      surname      patronymic      birthday      hotel_id
    9         93948a951    ee83a73cc03    5b37bf6873545    1961-03-15        2
If you don't want to UPDATE column -> write as it was
Input data separated by comma
Table: guest. Input: name->text, surname->text, patronymic->text, birthday->date, hotel_id->int
Raw,was,changeed,2021-12-27,8
UPDATED successfully
```

1. category
2. chambermaid
3. guest
4. hotel
5. room
6. room/chambermaid
7. room/guest
0. menu

Choose the table: 3

Input quantity of rows to print: 10

| guest_id | name | surname | patronymic | birthday | hotel_id |
|----------|-----------|-------------|---------------|------------|----------|
| 8 | e5ac7a31d | d936fae2244 | 974c7a82b1bef | 1980-09-04 | 1 |
| 9 | Raw | was | changeed | 2021-12-27 | 8 |
| 10 | 38748db2f | eb6eb753f59 | b89354ce2ef1c | 1965-04-05 | 2 |
| 11 | 808f75fc0 | bbee2914935 | dbed5ce97b720 | 1972-10-21 | 3 |
| 12 | d3fcc917e | 70f2d89f245 | c4ad55fb240fa | 1975-12-19 | 3 |
| 13 | f4a51766b | b102b9fe3ee | cb0ec1b405b2e | 1970-08-09 | 2 |
| 14 | 683a84c41 | 2a742052d7b | 669914424bd1f | 1969-08-02 | 3 |
| 15 | Alex | Patron | Orange | 1980-09-18 | 5 |

Завдання №2

Для тестування індексів було створено окремі таблиці у базі даних з 100000-200000 записів.

BTree

1. Для початку створимо пусту таблицю з двома текстовими полями, одну з використанням індексів, одну без.

```
create table btree_test(  
    elem varchar,  
    elem_indexed varchar  
);
```

2. Заповнимо обидва стовбця 100000 випадковими даними.

```
INSERT INTO "btree_test" SELECT  
md5(random()::text),  
md5(random()::text) from (  
SELECT * FROM generate_series(1,100000) AS id) AS ser;
```

3. Тепер за допомогою SELECT виберемо і відсортуємо всі значення за спаданням і зрівняємо час виконання кожної з команд.

```
CREATE INDEX btree_index ON  
btree_test using btree (elem_indexed);  
SELECT * FROM btree_test ORDER BY elem desc;  
SELECT * FROM btree_test ORDER BY elem_indexed desc;
```

Без використання індексів:

btree_gin/postgres@PostgreSQL 13

Query Editor

История запросов

2

3

4

select * from btree_test order by elem desc;

select * from btree_test order by elem_indexed desc;

Результат

План выполнения

Сообщения

Notifications

| | elem character varying | elem_indexed character varying |
|----|----------------------------------|-----------------------------------|
| 1 | ffff665d00aa572ae72ebd888b0dd2e9 | 3e16cbb135976eca82b16d77a86ff915 |
| 2 | ffff48d2c571a2bc03fbf82be536cf2d | d5168d25324e877c749cda36d4e0e2bf |
| 3 | ffff41fea3af592c06973f7a44b8ad69 | f7cfcfac7d6f697eab2e80241a0d0c |
| 4 | ffff0ffe79a2291bbfe138023b586484 | f57a82130cbcd5a2c7a3e674e18a5d95 |
| 5 | ffde7d734b20b17b593d27ebd9f9183 | c855df9318363e96f752c74ad406e5ae |
| 6 | fffdb9acfec0cbd7b6e4914e030997ca | c45368c90e9ea68ea2ee125373b88e0c |
| 7 | fffd96dd59bb07d3c68e6b4387076276 | 61c4f9223c2213673451d615369ca095 |
| 8 | fffd4d88aeb72e347aa5c946ca68f3f0 | f17c7b4b3202a628ef033b89baa9ceff |
| 9 | fffc4218c813769179cf5cac9efb5f2d | b7791b3f37fd17ef4b2317d39623b823 |
| 10 | fffc15c16e792e0dc5e2dfab771ed80d | b0654c4ceb86d5039fa204c468cec69e |
| 11 | fffb360d06c767b28331979a5b4ad41 | c3fb34a07b96c73638dd8a95552e1cfa |
| 12 | fffb1fc34720a9f3a020de74f4facd78 | 954024370c1691c93841b37638914bc1 |
| 13 | fffaac754df7ec0c1c25042c9ceef897 | 3f928517ee4fe135d58385238c069e6c |
| 14 | fff982088f284d1a05921e97b958f060 | ab4fc325bbaa5f3a25e144dcaab5394e |
| 15 | fff946a0913be29fc1e8dedeb8a57a43 | 16f0e608cf72fa33e9f52dbc12e9a0f9 |
| 16 | fff8b1585db08ad590f7a79234afac57 | 521dbaa4e39ab18d28ef77fb0f98125c |
| 17 | fff797874723ca37cabe5b0b4cb7a6a9 | 48d6d38c478495666ab35dbca12dbec9 |
| 18 | fff772ea180744ede7464ae2f9fbc2f | 2d7177918717324f0de3c5f0a6c7fa9a |
| 19 | fff7580adb98e2b3475b9d3991d151e1 | 699ac648029926d55488ffb6ff3e09166 |
| 20 | fff737bc9adba008bfb6cb630e5ef9b | ffb3e1479803cc0c7e17cd0051090f6 |

✓ Запрос выполнен успешно. Общее время выполнения: 933 msec. обработано строк: 100000.

З використанням індексів:

btree_gin/postgres@PostgreSQL 13 ▾

Query Editor

История запросов

2

3

4

select * from btree_test order by elem desc;

select * from btree_test order by elem_indexed desc;

Результат

План выполнения

Сообщения

Notifications

| | elem character varying | elem_indexed character varying |
|----|-----------------------------------|-----------------------------------|
| 1 | 8fc46c6a8886609904f242552ed966d0 | ffff9145b7db7bd39082c31db4a0db8 |
| 2 | e303745e39d6c455afe37a1a082cc452 | fffed5b4ede4150effdef8f49c0e128d |
| 3 | 7ee3c92806e8d38839ad2d89e32d15c5 | fffe46c5ff73a676de79830181d5de90 |
| 4 | f4b670e9ccdef590daa9e67d8690f433 | fffd959678069ad40678e8652874ecd |
| 5 | 1e9fd2ae56a2755dbe02174643c4ca51 | fffd18f16f4b011fb11386ee99ab2997 |
| 6 | aba7ee509d7bd64466b6bc161fe99fe7 | fffcfe183365c061633d14b26db7e101 |
| 7 | 1bfa713948c72696d668cff2aa9d3fda | fffc92dd00d6a8fa64c76c906fe320d4 |
| 8 | dea381f7d9f8108705036fac1b43cf1c | fffb83d41df56885b1babf0c1c575822 |
| 9 | c43c86c8187031fd9b4ddf38cc4014d3 | fffb3cead6c2f48c32677b71a5fc0682 |
| 10 | 6ed4a37be46b0ac98e493e986689bbe7 | fffa8feabfec76b9e5061c7c48663a35 |
| 11 | ca9e4ad9da546a1efb1159a43d3811b0 | fff702abfd66a0520a54b3f456057125 |
| 12 | 07e86ce915b9af324fee1e4eecf63c15 | fff6a052b3ad1fdd012594b0a7f94773 |
| 13 | 364c3869f20e669921aеac589a8a60bc | fff67000bf0ecb4c1963af85edfce64c |
| 14 | 5637e7fcd558eade30a78f50d6ef89b6 | fff654590bae63213023e9bfcfa4a4a7 |
| 15 | 6d7e99c4d853da714e8f21e3aaca7fb1 | fff63da2a576341d97c359f0a4a10fab |
| 16 | 18931f66e43090bcf41e088e3ca95b27 | fff4f0adb88a161fbe7092837b1b93d5 |
| 17 | f84351ad461ea943d7349848a307647f | fff418abc1161d15e4d84a4ec7d32ccd |
| 18 | 8a6957d22fecae9d9ede29aee2d8b3b10 | fff395a72b01191637a4863408050de0 |
| 19 | dac9c9bd3a6790b3e5bce07abccеc1518 | fff33757b19c5e4f78c36f4d76d893cf |
| 20 | 9dbcd82008b4b5c0fc8819f8c4e4c10a | fff306e0a6f4778e394152d2f1785946 |

✓

Запрос выполнен успешно. Общее время выполнения: 290 msec. обработано строк: 100000.

Можна помітити, що навіть при такій кількості даних, різниця в виконанні досить значна.

4. Збільшимо кількість даних на 200000 і зрівняємо знову швидкість виконання сортування.

Без використання індексів:

8

select * from btree_test order by elem desc;

9

select * from btree_test order by elem_indexed desc;

Результат

План выполнения

Сообщения

Notifications

| | elem character varying | elem_indexed character varying | |
|----|-----------------------------------|-----------------------------------|---|
| 1 | ffff90d1a021ad4d4ba1ff455d11cd25 | 228a7cf9ef13babce9bbfc7eea447bde | |
| 2 | ffff7f7bad7f5a44b8b07489c6cd2f80 | 4c9370d506fb586608da94e4fe35a990 | |
| 3 | ffff665d00aa572ae72ebd888b0dd2e9 | 3e16cbb135976eca82b16d77a86ff915 | |
| 4 | ffff48d2c571a2bc03fbf82be536cf2d | d5168d25324e877c749cda36d4e0e2bf | |
| 5 | ffff41fea3af592c06973f7a44b8ad69 | f7cfffacfac7d6f697eab2e80241a0d0c | |
| 6 | ffff40c525b619262a9a2744f56bdfc3 | 9840a8d89b0d79f3095a8961b0815822 | |
| 7 | ffff0ffe79a2291bbfe138023b586484 | f57a82130cbcd5a2c7a3e674e18a5d95 | |
| 8 | fffeb57eb1793dc33846fea2119fa28a | 4ec9e790045282585a9bd6b72f321062 | |
| 9 | fffe9d9bc53d73f760d223e86bd08666 | 722fdd139b16870d48d5e5bfe60b28d2 | |
| 10 | fffe5ed405abe8b1eda5febbe02dd552 | c32fab78ac3e9fe9755d393d10811ee | |
| 11 | fffded7d734b20b17b593d27ebd9f9183 | c855df9318363e96f752c74ad406e5ae | |
| 12 | fffdb9acfec0cbd7b6e4914e030997ca | c45368c90e9ea68ea2ee125373b88e0c | |
| 13 | fffd55e4271b5ec191a71b03071fdb1 | c703f427fb540aa14702a2feb8846c61 | |
| 14 | fffd96dd59bb07d3c68e6b4387076276 | 61c4f9223c2213673451d615369 | |
| 15 | fffd87eb019cf949322e7c8396f24f1a | ed8aef4669f7b7ebb83cda530b3 | ✓ Запрос выполнен успешно. Общее время выполнения: 2 secs 979 msec. обработано строк: 300000. |

З використанням індексів:

8

select * from btree_test order by elem desc;

9

select * from btree_test order by elem_indexed desc;

Результат

План выполнения

Сообщения

Notifications

| | elem character varying | elem_indexed character varying | |
|----|----------------------------------|-----------------------------------|--|
| 1 | 6b9f4611c059ee703eb269a806bcf10b | ffffaa43465fb17a82dce2bbf2c864aa | |
| 2 | 8fc46c6a8886609904f242552ed966d0 | ffff9145bf7db7bd39082c31db4a0db8 | |
| 3 | 8b7ca576c4202f2ecd4f348c9397e71a | ffff86286e85c659272b1fe9d54f7d46 | |
| 4 | e1c9925e140701c883de386a18b9b844 | ffff59ba80b8d3f9f3f152c785234917 | |
| 5 | 2914b545678f4927befbe5a35ac02cb5 | ffff1d68694a052ba7c4a11106f5572f5 | |
| 6 | 2d6c9fd9c90035b878c1c751e9ba35d2 | ffff0dd4f190688083e82d07c1bf5f61 | |
| 7 | e303745e39d6c455afe37a1a082cc452 | fffed5b4ede4150effdef8f49c0e128d | |
| 8 | c58fc730d352963595a887220a109242 | fffec23b44445ce4edede0dae6d27f22 | |
| 9 | 7ee3c92806e8d38839ad2d89e32d15c5 | fffe46c5ff73a676de79830181d5de90 | |
| 10 | d5128f1e62afdb3860eaa5099d725c2b | fffe0ee4ebe2ee1f4817e29508025a5d | |
| 11 | f4b670e9ccdef590daa9e67d8690f433 | fffd959678069ad40678e8652874ecd | |
| 12 | da2fb21a83c7aa2eac793f0aa59cf188 | fffd8b18fa8e3abedea7ff52d802571 | |
| 13 | 4f7344bb9876577202769432eb3c4396 | fffdbe6ed8f41b5cd7365a245d1ca94c6 | |
| 14 | 63923208ca69f0d80a349f77a6ab8c5d | fffd89f6e57d831f09facff21c5ad6f3 | |
| 15 | 1e9fd2ae56a2765dbe02174643c4ca51 | fffd18f16f4b011fb11386ee99ab2997 | ✓ Запрос выполнен успешно. Общее время выполнения: 683 msec. обработано строк: 300000. |

Результат показує, що використання індексів значно прискорює виконання запиту, а якщо даних дуже багато, то й швидкість більша в значну кількість разів.

GIN

GIN призначений для обробки випадків, коли елементи, що підлягають індексації, є складеними значеннями (наприклад - реченнями), а запити, які обробляються індексом, мають шукати значення елементів, які з'являються в складених елементах (повторювані частини слів або речень). Індекс GIN зберігає набір пар (ключ, список появи ключа), де список появи – це набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з'являється багато разів. Цей індекс може взаємодіяти тільки з полем типу **tsvector**.

Створення таблиці БД:

```
DROP TABLE IF EXISTS "gin_test";
CREATE TABLE "gin_test"("id" bigserial PRIMARY KEY,
"string" text, "gin_vector" tsvector);
INSERT INTO "gin_test"("string") SELECT substr(characters,
(random() * length(characters) + 1)::integer, 10) FROM
(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))
as symbols(characters), generate_series(1, 1000000) as q;
UPDATE "gin_test" set "gin_vector" = to_tsvector("string");
```

Запити для тестування:

```
SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector"
@@ to_tsquery('bnm'));
SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector"
@@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('bnm'));
SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE
("gin_vector" @@ to_tsquery('bnm')) GROUP BY "id" % 2;
```

Створення індексу:

```
DROP INDEX IF EXISTS "gin_index";
CREATE INDEX "gin_index" ON "gin_test" USING gin("gin_vector");
```

Результати і час виконання на скріншотах:

Без використання індексів:

```
Секундомер включён.
postgres=# DROP INDEX IF EXISTS "gin_index";
ПОВІДОМЛЕННЯ:  індекс "gin_index" не існує, пропускається
DROP INDEX
Время: 1,634 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 203,518 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
count
-----
19142
(1 строка)

Время: 474,229 мс
postgres=# SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('bnm'));
sum
-----
23943769938
(1 строка)

Время: 1188,034 мс (00:01,188)
postgres=# SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm')) GROUP BY "id" % 2;
min | max
-----+-----
100 | 999994
45  | 999937
(2 строки)

Время: 1120,586 мс (00:01,121)
```

З використанням індексів:

```
postgres=# CREATE INDEX "gin_index" ON "gin_test" USING gin("gin_vector");
CREATE INDEX
Время: 355,983 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE "id" % 2 = 0;
count
-----
500000
(1 строка)

Время: 156,321 мс
postgres=# SELECT COUNT(*) FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm'));
count
-----
19142
(1 строка)

Время: 25,425 мс
postgres=# SELECT SUM("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('QWERTYUIOP')) OR ("gin_vector" @@ to_tsquery('bnm'));
sum
-----
23943769938
(1 строка)

Время: 243,217 мс
postgres=# SELECT MIN("id"), MAX("id") FROM "gin_test" WHERE ("gin_vector" @@ to_tsquery('bnm')) GROUP BY "id" % 2;
min | max
-----+-----
45  | 999937
100 | 999994
(2 строки)

Время: 13,533 мс
```

З отриманих результатів бачимо, що в усіх заданих випадках пошук з індексацією відбувається значно швидше, ніж пошук без індексації (окрім першого, оскільки на перший запит дана індексація не впливає). Це відбувається завдяки головній особливості індексування GIN: кожне значення шуканого ключа зберігається один раз і запит іде не по всій таблиці, а лише по тим даним, що містяться у списку появи цього ключа. Для даних типу numeric даний тип індексування використовувати недоцільно і неможливо.

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 "before_delete_update_trigger"
BEFORE DELETE OR UPDATE ON "trigger_test"
FOR EACH ROW EXECUTE procedure before_delete_update_func();
```

Текст тригера:

```
CREATE OR REPLACE FUNCTION before_delete_update_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;
            RETURN 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");
            RETURN 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;
        RETURN OLD;
    END 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 |

```
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 |

| | 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 |

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 виконався, але пішов іншою (вкладеною) гілкою алгоритму та повернув старий стан (OLD). При запиті на оновлення потрібно повертати новий стан, а при запиті на видалення 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 |

| | 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 модифікуються існуючі записи, але нові не додаються. Оскільки до цього не було виконано 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 |

| | 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 |

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 вимволи "_", а в кінці три "_log". Один "_log" в кінці додався завдяки виконанню запиту update для всіх парних 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

Для цього завдання також створювалась окрема таблиця з деякими початковими даними:

```
DROP TABLE IF EXISTS "transactions";
CREATE TABLE "transactions"(
    "id" bigserial PRIMARY KEY,
    "numeric" bigint,
    "text" text
);

INSERT INTO "transactions"("numeric", "text")
VALUES (111, 'string1'), (222, 'string2'), (333, 'string3');
```

READ COMMITTED

На цьому рівні ізоляції одна транзакція не бачить змін у базі даних, викликаних іншою доки та не завершить своє виконання (командою COMMIT або ROLLBACK).

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=#
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |     111 | string1
  2 |     222 | string2
  3 |     333 | string3
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |     112 | string1
  2 |     223 | string2
  3 |     334 | string3
(3 строки)
```

Дані після вставки та видалення так само будуть видні другій тільки після завершення першої.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=#

postgres=#
postgres=#
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=#

postgres=#
postgres=# COMMIT;
COMMIT
postgres=#
```

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

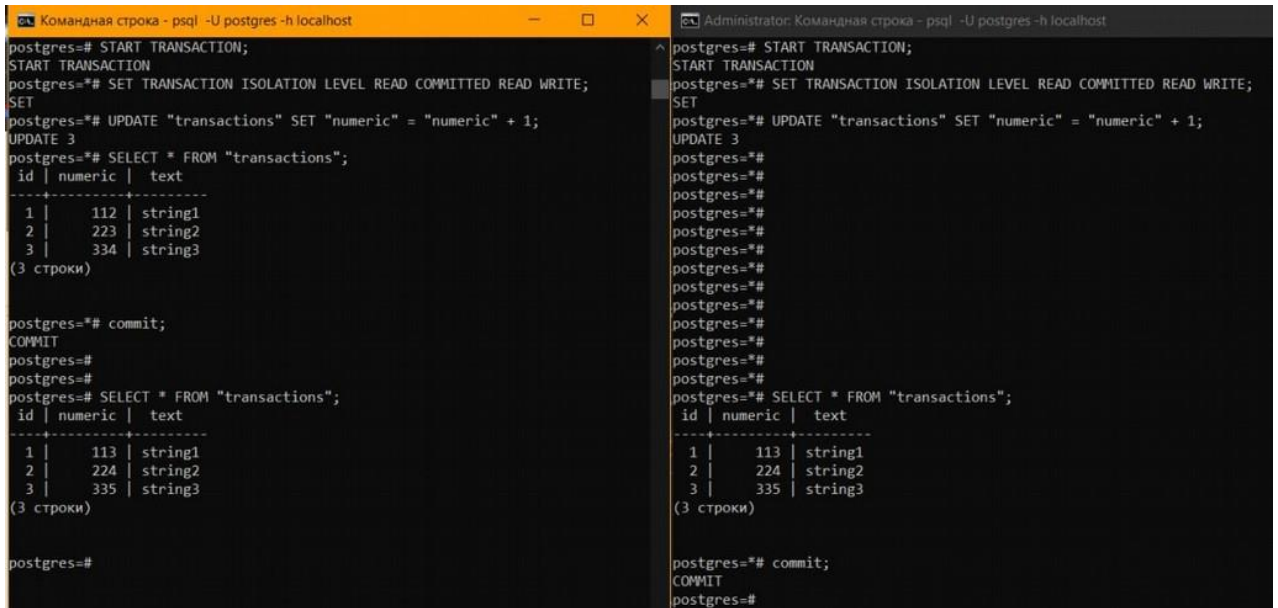
postgres=#
```

На цьому знімку також бачимо, що друга транзакція (справа) не може внести дані у базу, доки не завершилась попередня.

```
Командная строка - psql -U postgres -h localhost
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
```

```
Administrator: Командная строка - psql -U postgres -h localhost
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
```

А тут бачимо, що після завершення першої, друга транзакція виконала запит, змінивши вже ті дані, що були закомічені першою транзакцією



```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=#
```

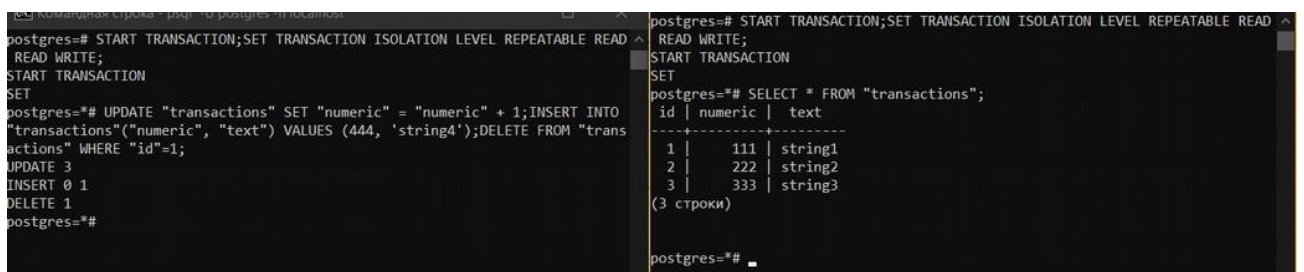
```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=#
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    113 | string1
  2 |    224 | string2
  3 |    335 | string3
(3 строки)

postgres=# commit;
COMMIT
postgres=#
```

Коли T2 бачить дані T1 запитів UPDATE, DELETE виникає феномен повторного читання, а коли бачить дані запиту INSERT – читання фантомів. Цей рівень ізоляції забезпечує захист від явища брудного читання.

REPEATABLE READ

На цьому рівні ізоляції T2 не бачитиме змінені дані транзакцією T1, але також не зможе отримати доступ до тих самих даних. Тут видно, що друга не бачить змін з першої:



```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
READ WRITE;
START TRANSACTION
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;INSERT INTO
"transactions"("numeric", "text") VALUES (444, 'string4');DELETE FROM "trans
actions" WHERE "id"=1;
UPDATE 3
INSERT 0 1
DELETE 1
postgres=#
```

```
postgres=# START TRANSACTION;SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
READ WRITE;
START TRANSACTION
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=#
```

А тут, що отримуємо помилку при спробі доступу до тих самих даних:

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# COMMIT;
COMMIT
postgres=#

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# SELECT * FROM "transactions";
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT;
ROLLBACK
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    112 | string1
  2 |    223 | string2
  3 |    334 | string3
(3 строки)
```

Бачимо, що не виникає читання фантомів та повторного читання, а також заборонено одночасний доступ до незбережених даних. Хоча класично цей рівень ізоляції призначений для попередження повторного читання.

SERIALIZABLE

На цьому рівні транзакції поведуть себе так, ніби вони не знають одна про одну. Вони не можуть вплинути одна на одну і одночасний доступ строго заборонений.

```
postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
UPDATE 3
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
INSERT 0 1
postgres=# DELETE FROM "transactions" WHERE "id"=1;
DELETE 1
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# COMMIT;
COMMIT
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  2 |    223 | string2
  3 |    334 | string3
  4 |    444 | string4
(3 строки)

postgres=# START TRANSACTION;
START TRANSACTION
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;
SET
postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=# SELECT * FROM "transactions";
 id | numeric | text
-----+-----+-----
  1 |    111 | string1
  2 |    222 | string2
  3 |    333 | string3
(3 строки)

postgres=# UPDATE "transactions" SET "numeric" = "numeric" + 1;
ПОМИЛКА: не вдалося серіалізувати доступ через паралельне оновлення
postgres=# INSERT INTO "transactions"("numeric", "text") VALUES (444, 'string4');
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# DELETE FROM "transactions" WHERE "id"=1;
ПОМИЛКА: поточна транзакція перервана, команди до кінця блока транзакції пропускаються
postgres=# COMMIT
postgres=# ROLLBACK
postgres=# COMMIT
postgres=#
```

У попередньому випадку вдалось “відкатити” другу транзакцію і це не вплинуло на подальшу можливість роботи в терміналі. На цьому ж рівні навіть після завершення першої не вдалося зробити ні COMMIT ні ROLLBACK для другої транзакції. Взагалі, в класичному представленні цей рівень призначений для недопущення явища читання фантомів. На цьому рівні ізоляції ми отримуємо максимальну узгодженість даних і можемо бути впевнені, що зайві дані не будуть зафіксовані.