Міністерство освіти та науки України
Національний технічний університет України “Київський політехнічний інститут імені Ігоря Сікорського”
Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп’ютерних систем

ЛАБОРАТОРНА РОБОТА №2

з дисципліни

“Бази даних”

“Засоби оптимізації роботи СУБД PostgreSQL”

Виконала Лукащук Ю.
Студентка групи КВ-22
Telegram: @sunshine_o9
Github: yuliya2409/db-kpi (github.com)

Київ 2024

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC РГР 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 14

14	*Btree, Hash*	*after insert, update*

Схема бази даних

Тема: “Платформа для бронювання та управління майданчиками для подій”

Перелік сутностей:

Організатор (Manager) - сутність організатора події.

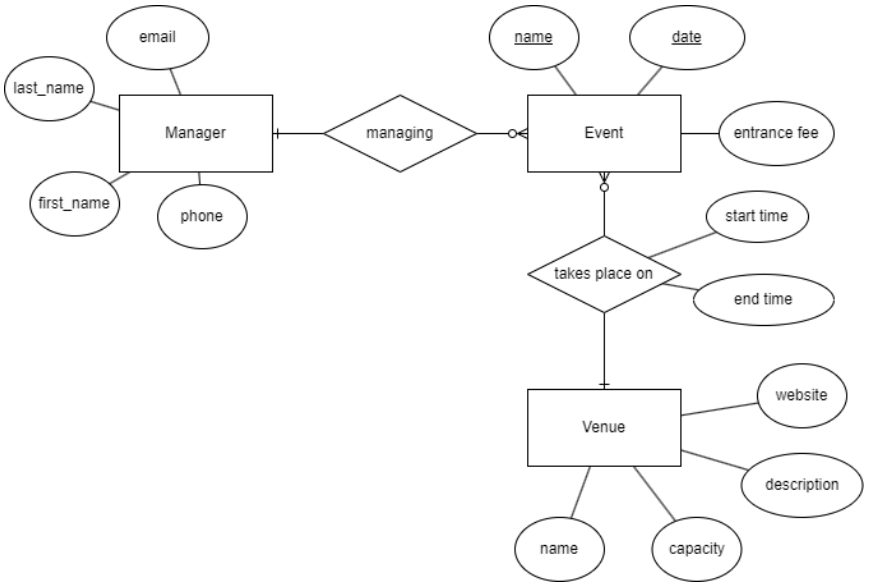
Подія (Event) - сутність події, якою керує організатор.

Майданчик (Venue) - сутність майданчика, на якому може проходити подія.

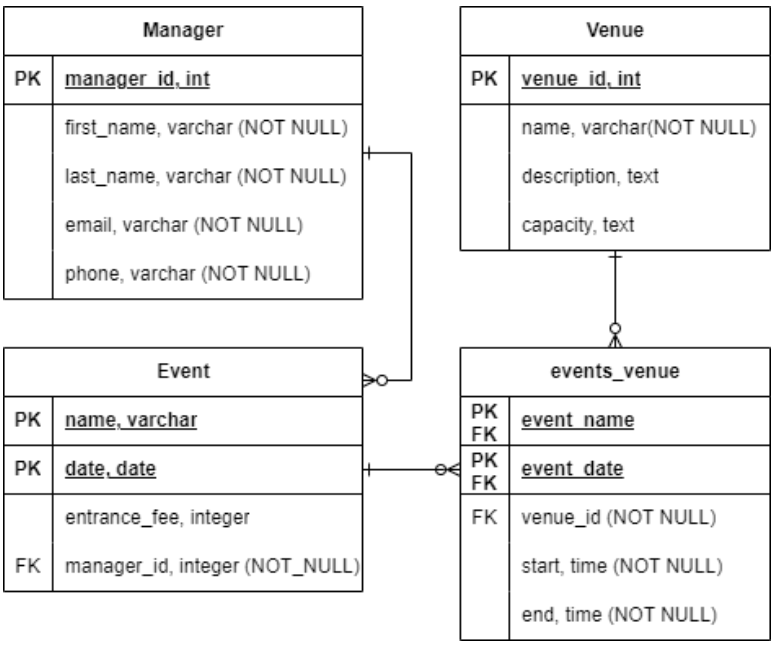
Між сутностями проходять наступні зв’язки:

Організатор(1) -> влаштовує -> подію(0...N)

Подія(0…N) -> відбувається на -> майданчику(1)



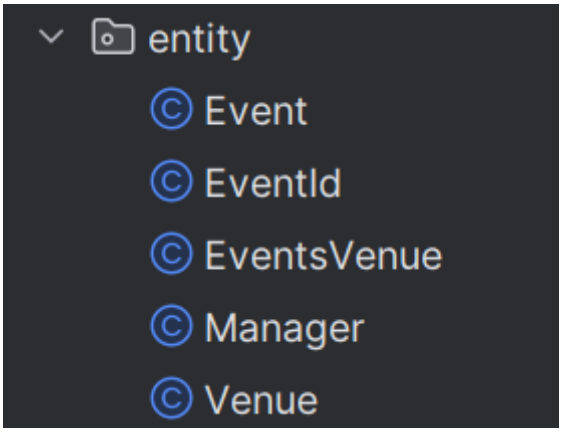
ER-діаграма



представлення бази даних у вигляді таблиць

Завдання №1

Опис класів ORM-моделі



Для створення сутностей об’єктно-реляційної проєкції було використано фреймворк Hibernate для Java.

Event

```
@Entity
@Table(name = "event")
@IdClass(EventId.class)
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Setter
@Getter
public class Event {

    @Id
    private String name;
```

```
    @Id
    @Temporal(TemporalType.DATE)
    private Date date;

    @Column(name = "entrance_fee")
    private int entranceFee;

    @ManyToOne
    @JoinColumn(name = "manager_id")
    private Manager manager;

}

@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class EventId implements Serializable {
    private String name;
    private Date date;
}
```

Venue

```
@Entity
@Table(name = "venue")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class Venue {

    @Id
    @Column(name = "venue_id")
    private int venueId;

    private String name;

    private String description;

    private String website;

    private int capacity;
}
```

Manager

```
@Entity
@Table(name = "manager")
```

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Manager {

    @Id
    @Column(name = "manager_id")
    private int managerId;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    private String email;


    private String phone;

    @OneToMany(mappedBy = "manager", cascade = CascadeType.ALL)
    private List<Event> events = null;

    public Manager(int managerId, String firstName, String lastName, String
email, String phone) {
        this(managerId, firstName, lastName, email, phone, null);
    }
}
```

EventsVenue

```
@Entity
@Table(name = "events_venue")
@IdClass(EventId.class)
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class EventsVenue {

    @Id
    @Column(name = "event_name")
    private String name;

    @Id
    @Column(name = "event_date")
    private Date date;
```

```
    @OneToOne
    private Event event;

    @OneToOne
    @JoinColumn(name = "venue_id")
    private Venue venue;

    @Temporal(TemporalType.TIME)
    private Time start;

    @Temporal(TemporalType.TIME)
    @Column(name = "\"end\"")
    private Time end;
}
```

Приклади запитів за допомогою ORM

```
Columns:
manager_id:
145
first_name:
Manager
last_name:
Managerov
email:
m@mail.com
phone:
380-0505-1212
Hibernate: insert into manager (email,first_name,last_name,phone,manager_id) values (?,?,?,?,?)
```

```
Columns:
name:
Event
date:
2021-01-10
entrance_fee:
1000
manager_id:
145
Hibernate: insert into event (entrance_fee,manager_id,date,name) values (?,?,?,?)
```

вставка

```
Primary key:
manager_id:
145
Columns:
manager_id:
145
first_name:
Manager_
last_name:
Managerov
email:
m_@mail.ua
phone:
01101010
Hibernate: select m1_0.manager_id,m1_0.email,m1_0.first_name,m1_0.last_name,m1_0.phone from manager m1_0 where m1_0.manager_id=?
Hibernate: update manager set email=?,first_name=?,last_name=?,phone=? where manager_id=?
```

редагування

```
Table:
1. event
2. manager
3. venue
4. events_venue
2
Primary key:
manager_id:
145
Hibernate: select m1_0.manager_id,m1_0.email,m1_0.first_name,m1_0.last_name,m1_0.phone from manager m1_0 where m1_0.manager_id=?
Hibernate: select e1_0.manager_id,e1_0.date,e1_0.name,e1_0.entrance_fee from event e1_0 where e1_0.manager_id=?
Hibernate: delete from event where date=? and name=?
Hibernate: delete from manager where manager_id=?
```

видалення

Як видно, видалення відбувається не тільки самого менеджера, а і для пов’язаних з ним подій.

Завдання №2

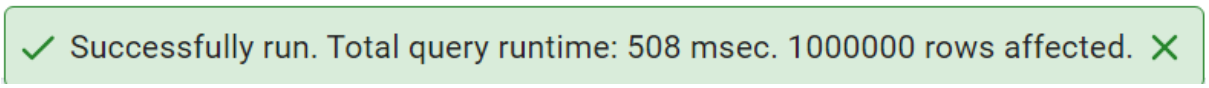
BTREE індекс

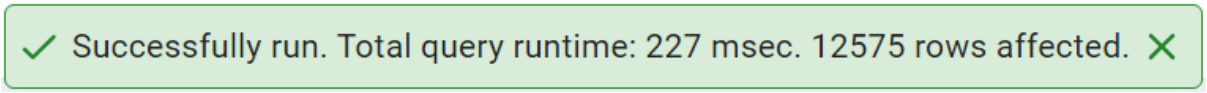
Для роботи з індексами створимо таблицю:

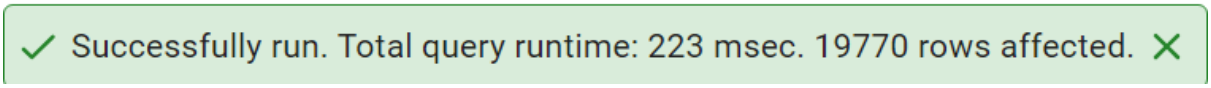
```
DROP TABLE IF EXISTS BTREE_TABLE;

CREATE TABLE BTREE_TABLE(
     ID SERIAL PRIMARY KEY,
     NAME VARCHAR(16),
     AGE INT
);

INSERT INTO BTREE_TABLE(NAME, AGE)
SELECT
CHR((65 + random() * 25)::INT) || CHR((65 + random() *
25)::INT),
(RANDOM() * 80)::INT
FROM GENERATE_SERIES(1, 1000000);
```

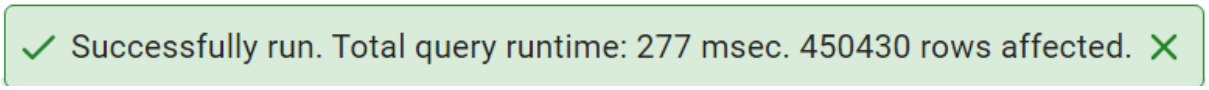
Підготуємо запити для тестування швидкодії:

```
SELECT * FROM BTREE_TABLE;
SELECT * FROM BTREE_TABLE WHERE AGE = 72;
SELECT * FROM BTREE_TABLE WHERE NAME LIKE '%A';
SELECT * FROM BTREE_TABLE WHERE AGE BETWEEN 25 AND 60;
```



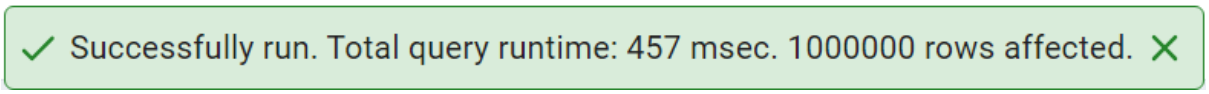


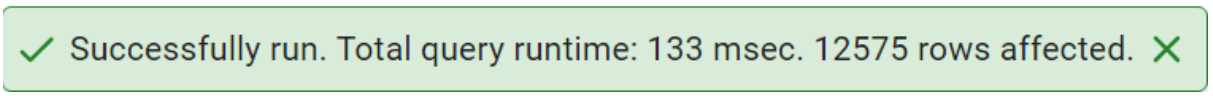


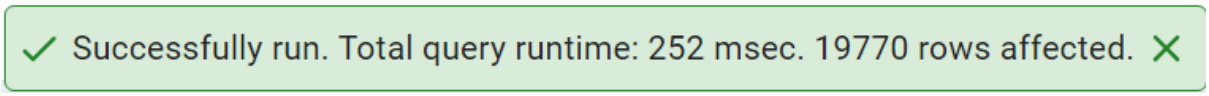


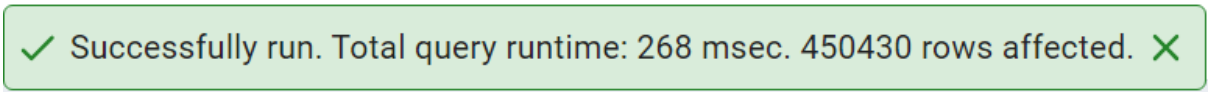
Створимо BTREE індекс для атрибуту age:

```
CREATE INDEX BTREE_TEST ON BTREE_TABLE USING BTREE(AGE);
```









Як бачимо, створення індексу прискорило виконання запитів, пов’язаних з атрибутом age. Для атрибута name швидкість виконання запиту приблизно така сама. Помітно, що для вибірки кортежів з певним значенням age виконується набагато швидше, а для вибірки з певного інтервалу швидкість не сильно зросла. Справа у тому, що індекс BTREE показує кращі результати, якщо у запитах в блоці фільтрації даних використовуються дані, що можна порівняти(>,<,=,!=, BETWEEN), а також рядки та шаблони рядків, що не починаються з маски. При цьому, швидкість виконання збільшується особливо коли вибірка є невеликою в порівнянні з повним об’ємом даних(до 5-10%).

HASH індекс

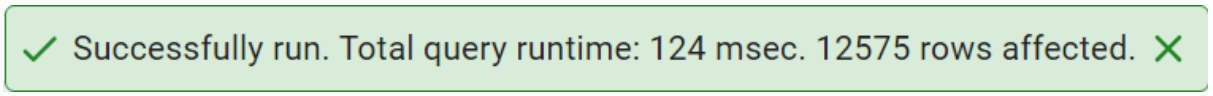
Створимо таку саму таблицю для хеш-індексу.

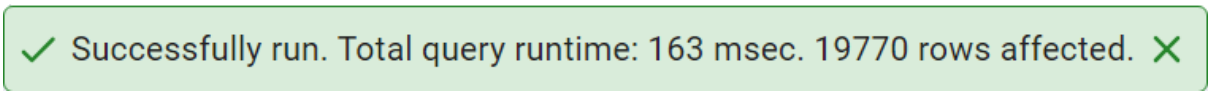
```
DROP TABLE IF EXISTS HASH_TABLE;

CREATE TABLE HASH_TABLE(
     ID SERIAL PRIMARY KEY,
     NAME VARCHAR(16),
     AGE INT
);

INSERT INTO HASH_TABLE(NAME, AGE)
SELECT
CHR((65 + random() * 25)::INT) || CHR((65 + random() *
25)::INT),
(RANDOM() * 80)::INT
FROM GENERATE_SERIES(1, 1000000);
```

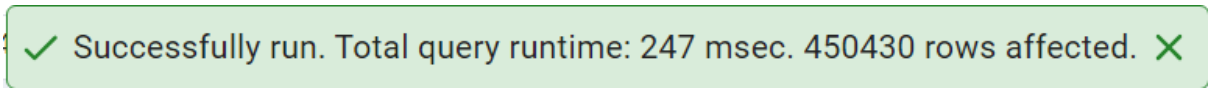
Оскільки таблиці однакові як за розміром, так і за наповненням, не будемо дублювати запити без використання індексів. Одразу створимо хеш-індекс

```
CREATE INDEX HASH_TEST ON HASH_TABLE USING HASH(AGE)
```

Тепер виконаємо запити в тій же послідовності:







У цьому випадку спостерігається здебільшого пришвидшення виконання другого запиту(WHERE AGE = 72), оскільки хеш-індекс є ефективним лише коли відбувається пошук для певного значення. При його використанні такі запити виконуються значно швидше, хоча він і не дає такої гнучкості, як індекси, що базуються на деревах.

Завдання №3

Текст коду тригерної функції:

```
CREATE OR REPLACE FUNCTION TRIGGER_CHECK()
RETURNS TRIGGER AS $$
DECLARE
EVENTS CURSOR(VENUE__ID INT) FOR
            SELECT EVENT_NAME FROM EVENTS_VENUE
            WHERE VENUE_ID = VENUE__ID;
BEGIN
      IF TG_OP = 'INSERT' THEN
            IF NEW.CAPACITY < 0 THEN
                  RAISE EXCEPTION 'Capacity can''t be negative';
            END IF;
            INSERT INTO VENUE_TIMESTAMPS(VENUE_ID, CREATED_AT)
VALUES(NEW.VENUE_ID, NOW()::TIMESTAMP);
            RETURN NEW;
      ELSIF TG_OP = 'UPDATE' THEN
            IF NEW.CAPACITY < 0 THEN
                  RAISE EXCEPTION 'Invalid time in updated record';
            END IF;
            FOR "EVENT" IN EVENTS(NEW.VENUE_ID) LOOP
                  RAISE NOTICE '%', "EVENT".NAME;
            END LOOP;
            UPDATE VENUE_TIMESTAMPS SET LAST_UPDATED = NOW()::TIMESTAMP
WHERE VENUE_ID = NEW.VENUE_ID;
            RETURN NEW;
      END IF;
END;
$$ LANGUAGE PLPGSQL;

CREATE OR REPLACE TRIGGER LAB_TRIGGER
AFTER INSERT OR UPDATE ON VENUE
FOR EACH ROW EXECUTE FUNCTION TRIGGER_CHECK();
```

Робота тригера полягає у перевірці коректності введення кількості відвідувачів майданчика, оскільки вона не може бути від’ємною. У випадку помилки створюється відповідний виняток. Курсорним циклом виводяться усі події, що відбуваються на цьому майданчику, а також в окрему таблицю записуються часові мітки створення та останнього оновлення запису.

Переконаємося у правильності роботи тригера:

```
INSERT INTO VENUE(VENUE_ID, NAME, DESCRIPTION, CAPACITY)
VALUES (555, 'Trigger', 'Trigger', 100);
```

created_at timestamp without time zone	last_updated timestamp without time zone	venue_id [PK] bigint
2024-12-24 22:09:58.010211	[null]	555

```
UPDATE VENUE
SET NAME = 'Trr' WHERE VENUE_ID = 555
```

created_at timestamp without time zone	last_updated timestamp without time zone	venue_id [PK] bigint
2024-12-24 22:09:58.010211	2024-12-24 22:11:21.314049	555

```
UPDATE VENUE
SET CAPACITY = -3 WHERE VENUE_ID = 555
```

```
ERROR:  Capacity can't be negative
CONTEXT:  функция PL/pgSQL trigger_check(), строка 15, оператор RAISE

ОШИБКА:  Capacity can't be negative
SQL state: P0001
```

Завдання №4

Перелік аномалій доступу до даних при паралельному виконанні транзакцій:

- **Втрачене оновлення -** коли у результаті виконання кількох транзакцій виконується оновлення одного і того ж запису, усі оновлення крім одного втрачаються
- **Брудне читання -** читання однією транзакцією даних, що були змінені, проте не зафіксовані іншою транзакцією
- **Неповторюване читання -** коли у межах транзакції повторне читання дає різні результати
- **Фантомне читання -** коли у межах транзакції повторна вибірка даних дає різні множини рядків

Переглянемо кожен з рівнів ізоляції транзакцій на прикладі таблиці Event.

READ UNCOMMITTED

Найнижчий рівень ізоляції транзакцій, забезпечує лише послідовність виконання паралельних запитів оновлення за допомогою блокувань. Не

захищає від жодних аномалій читання, оскільки не блокує таблицю під час запитів на читання.

READ COMMITTED

Наступний рівень ізоляції транзакцій, забезпечує захист від брудного читання шляхом подання транзакції “версії” таблиці, що була зафіксована останньою. При цьому залишаються проблеми фантомного читання та неповторюваного читання.

Приклад захисту від брудного читання:

```
events_managing_system=# begin;
BEGIN
events_managing_system=*# update event set entrance_fee =
121 where name = 'asd';
UPDATE 1
```

транзакція 1

```
events_managing_system=*# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
(10 rows)


events_managing_system=*# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
(10 rows)
```

транзакція 2

Помітно, що зміни у першій транзакції, що не були зафіксовані, не відображаються у рамках другої.

Приклад неповторюваного читання:

```
events_managing_system=# begin;
BEGIN
events_managing_system=*# update event set entrance_fee =
 75 where name = 'Olya Polyakova';
UPDATE 1
events_managing_system=*# Commit;
COMMIT
events_managing_system=#
```

транзакція 1

```
events_managing_system=# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           50 |          3
(10 rows)


events_managing_system=# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
```

транзакція 2

Приклад фантомного читання:

```
events_managing_system=# begin
events_managing_system-# inser into event(name, date, entr
ance_fee, manager_id) values('a', '2012-01-01',100,3);
ОШИБКА:  ошибка синтаксиса (примерное положение: "inser")
LINE 2: inser into event(name, date, entrance_fee, manager
_id) value...
        ^
events_managing_system=# insert into event(name, date, ent
rance_fee, manager_id) values('a', '2012-01-01',100,3);
INSERT 0 1
events_managing_system=#
```

транзакція 1

```
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
(10 rows)


events_managing_system=# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 FRW             | 2025-09-16 |           41 |          2
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
 a               | 2012-01-01 |          100 |          3
```

транзакція 2

REPEATABLE READ

Наступний рівень ізоляції транзакцій, при якому дані, які були вже прочитані в межах транзакції, зберігають той же стан, не отримуючи змін з будь-яких інших транзакцій, при цьому зміни в цих даних у межах цієї ж транзакції, очевидно будуть видимими для неї. При цьому ці дані блокуються для редагування іншими транзакціями. Отже, цей рівень захищає від брудного та неповторюваного читання, але не захищає від фантомного читання.

Приклад захисту від повторюваного читання:

```
events_managing_system=# BEGIN TRANSACTION ISOLATION LEVEL
 REPEATABLE READ;
BEGIN
events_managing_system=*# update event set entrance_fee =
15 where name = 'FRW';
UPDATE 1
events_managing_system=*# COMMIT;
COMMIT
```

транзакція 1

```
events_managing_system=*# SELECT * FROM EVENT;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
 FRW             | 2025-09-16 |           12 |          2
(10 rows)


events_managing_system=*# SELECT * FROM EVENT;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
 FRW             | 2025-09-16 |           12 |          2
(10 rows)
```

транзакція 2

Помітно, що навіть після фіксування змін, у межах другої транзакції відображаються ті дані, що були отримані під час першого читання.

SERIALIZABLE

Найвищий (найбільш строгий) рівень ізоляції транзакцій. Забезпечує повністю послідовний і атомарний хід виконання транзакцій, блокуючи будь-які спроби паралельного доступу до даних. Є при цьому найменш ефективним з точки зору швидкодії, проте захищає від будь-яких аномалій доступу до даних.

Приклад захисту від паралельного виконання запитів запису нових даних:

```
events_managing_system=# BEGIN TRANSACTION ISOLATION LEVEL
 SERIALIZABLE;
BEGIN
events_managing_system=*# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
 FRW             | 2025-09-16 |           15 |          2
 a               | 2010-01-05 |           16 |          2
 AAA             | 2013-01-05 |           16 |          2
(12 rows)


events_managing_system=*# insert into event(name, date, en
trance_fee, manager_id) values ('AAAb', '2013-01-05', 16,
2);
INSERT 0 1
events_managing_system=*# commit;
COMMIT
```

транзакція 1

```
events_managing_system=# BEGIN TRANSACTION ISOLATION LEVE
L SERIALIZABLE;
BEGIN
events_managing_system=*# select * from event;
      name       |    date    | entrance_fee | manager_id
-----------------+------------+--------------+------------
 DREVO           | 2024-10-23 |          450 |          3
 Anna Trincher   | 2024-11-01 |          700 |          2
 KOLA            | 2024-10-19 |          700 |          1
 UET             | 2025-07-30 |           27 |          1
 REK             | 2025-05-10 |           61 |          3
 OSB             | 2025-07-16 |           21 |          1
 YYM             | 2025-09-09 |           77 |          2
 asd             | 1970-01-01 |          123 |          2
 Olya Polyakova  | 2024-10-26 |           75 |          3
 FRW             | 2025-09-16 |           15 |          2
 a               | 2010-01-05 |           16 |          2
 AAA             | 2013-01-05 |           16 |          2
(12 rows)


events_managing_system=*# insert into event(name, date, e
ntrance_fee, manager_id) values('ABC', '2012-01-03', 155,
 1);
INSERT 0 1
events_managing_system=*# commit;
ОШИБКА:  не удалось сериализовать доступ из-за зависимост
ей чтения/записи между транзакциями
DETAIL:  Reason code: Canceled on identification as a piv
ot, during commit attempt.
HINT:  Транзакция может завершиться успешно при следующей
 попытке.
```

транзакція 2

Помітно, що друга транзакція блокується СУБД, отже рівень Serializable захищає від паралельного редагування/вставки даних у транзакціях.