

**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”**

**Виконав: Молчембаєв Я. А.  
Студент групи KB-23  
Telegram: @m3r7v4i444c0m  
Github: [kpi\\_databases/lab2 at main · r1pth3sl1t/kpi\\_databases](https://github.com/r1pth3sl1t/kpi_databases)**

**Київ 2024**

### Завдання на лабораторну роботу

1. Перетворити модуль “Модель” з шаблону MVC РГР 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 19

19	<i>BTree, BRIN</i>	<i>before insert, delete</i>
----	--------------------	------------------------------

### Опис структури бази даних

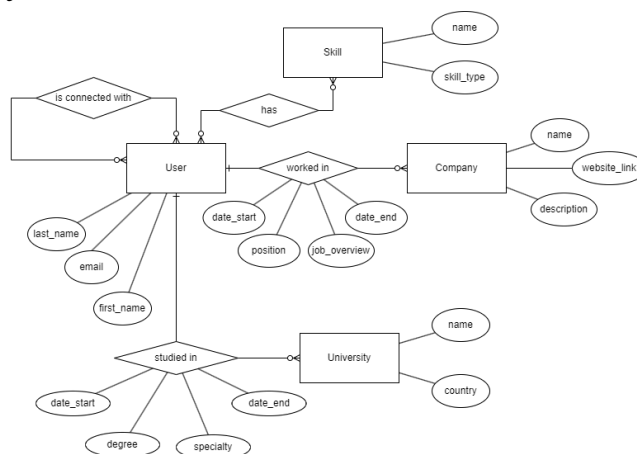
Тема бази даних: Соціальна мережа для професійних зв'язків.

Сутності бази даних:

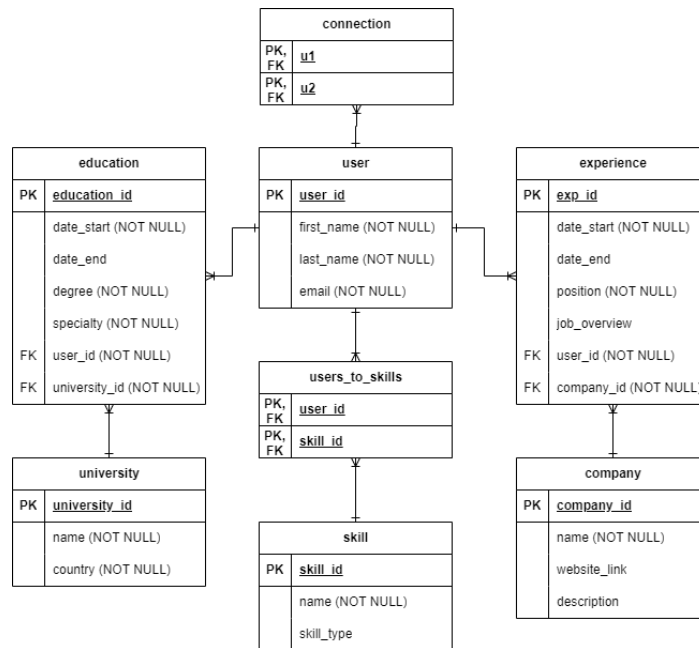
- User - сутність користувача соціальної мережі.
- Skill - сутність навички користувача. Кожен користувач може налаштувати у своєму профілі набір навичок, якими він володіє, для підбору доступних вакансій або для роботодавця.
- Company - сутність компанії. Користувач може мати попередній досвід роботи в певних компаніях і опублікувати інформацію про цей досвід у своєму профілі.
- University - сутність навчального закладу. Користувач може мати освіту і опублікувати інформацію про неї у своєму профілі.

Зв'язки моделі:

- M:N користувач - користувач
- 0:N користувач - навчальний заклад
- 0:N користувач - компанія
- M:N користувач - навичка



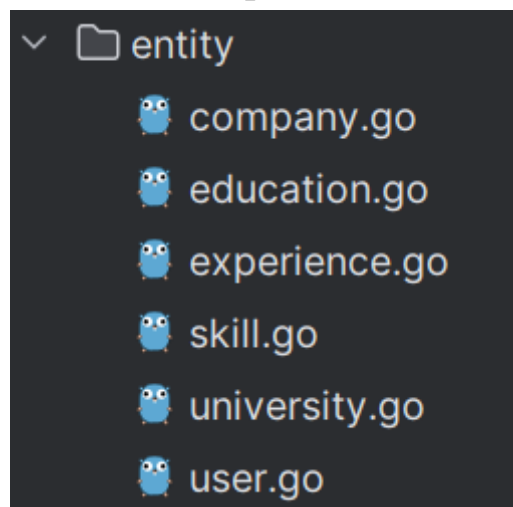
*ER-діаграма бази даних*



*схема бази даних*

## Завдання №1

### Опис об'єктно-реляційної моделі



*файлова структура*

Для створення об'єктно-реляційної моделі було використано бібліотеку GORM. Вона передбачає конфігурацію сутностей бази даних за допомогою структур та тегів їх полів.

### Сутність Company

```

package entity

type Company struct {
    CompanyId int `gorm:"column:company_id; primaryKey"
    Name      string `gorm:"column:name"`
    Website   string `gorm:"column:website_link"`
    Description string `gorm:"column:description"`
}
  
```

```

}

func (Company) TableName() string {
    return "company"
}

func (Company) Create(fields map[string]string) *Company {
    return &Company{
        Name:      fields["Name"],
        Website:   fields["Website"],
        Description: fields["Description"],
    }
}

```

## Сутність User

```

package entity

type User struct {
    UserId      int          `gorm:"column:user_id;primaryKey"
requiredToInput:"false"`
    FirstName   string       `gorm:"column:first_name"`
    LastName    string       `gorm:"column:last_name"`
    Email       string       `gorm:"column:email"`
    Skills      []Skill
    `gorm:"many2many:users_to_skills;joinForeignKey:user_id;joinReferences:skill_
id;onDelete:SET NULL" requiredToInput:"false"`
    Connections []*User
    `gorm:"many2many:connection;joinForeignKey:u1;joinReferences:u2;onDelete:SET
NULL" requiredToInput:"false"`
    Education   []Education `gorm:"foreignKey:UserId;onDelete:CASCADE"
requiredToInput:"false"`
    Experience   []Experience `gorm:"foreignKey:UserId;onDelete:CASCADE"
requiredToInput:"false"`
}

func (User) TableName() string {

    return "user"
}

func (User) Create(fields map[string]string) *User {

    return &User{
        FirstName: fields["FirstName"],
        LastName:  fields["LastName"],
        Email:     fields["Email"],
    }
}

```

## Сутність Skill

```

package entity

```

```

type Skill struct {
    SkillId    int    `gorm:"column:skill_id; primaryKey"
requiredToInput:"false"`
    SkillName  string `gorm:"column:name"`
    SkillType  string `gorm:"column:skill_type"`
}

func (Skill) Create(fields map[string]string) *Skill {
    return &Skill{
        SkillName: fields["SkillName"],
        SkillType: fields["SkillType"],
    }
}

func (Skill) TableName() string {
    return "skill"
}

```

## Сутність University

```

package entity

type University struct {
    UniversityId int    `gorm:"column:university_id;primaryKey"
requiredToInput:"false"`
    Name          string `gorm:"column:name"`
    Country       string `gorm:"column:country"`
}

func (University) TableName() string {
    return "university"
}

func (University) Create(fields map[string]string) *University {
    return &University{
        Name:      fields["Name"],
        Country:   fields["Country"],
    }
}

```

GORM передбачає опис зв'язків з атрибутами як окремі сутності, також у вигляді структур.

## Зв'язок Education

```

package entity

import (
    "strconv"
    "time"
)

type Education struct {

```

```

    EducationId int `gorm:"column:education_id;primaryKey "`
requiredToInput:"false"`
    UniversityId int
    UserId int `gorm:"column:user_id"`
    University *University `gorm:"foreignKey:UniversityId"
requiredToInput:"false"`
    DateStart time.Time `gorm:"column:date_start"`
    DateEnd time.Time `gorm:"column:date_end"`
    Degree string `gorm:"column:degree"`
    Specialty string `gorm:"column:specialty"`
}

func (Education) TableName() string {
    return "education"
}

func (Education) Create(fields map[string]string) (*Education, error) {
    education := Education{}
    universityId, err := strconv.Atoi(fields["UniversityId"])
    education.UniversityId = universityId
    education.DateStart, err = time.Parse("2006-01-02", fields["DateStart"])
    education.DateEnd, err = time.Parse("2006-01-02", fields["DateEnd"])
    if err != nil {
        return nil, err
    }
    education.Degree = fields["Degree"]
    education.Specialty = fields["Specialty"]
    return &education, err
}

```

## Зв'язок Experience

```

package entity

import (
    "strconv"
    "time"
)

type Experience struct {
    ExperienceId int `gorm:"column:exp_id;primaryKey"
requiredToInput:"false"`
    UserId int `gorm:"column:user_id"`
    Company *Company `gorm:"foreignKey:CompanyId"
requiredToInput:"false"`
    CompanyId int `gorm:"column:company_id"`
    DateStart time.Time `gorm:"column:date_start"`
    DateEnd time.Time `gorm:"column:date_end"`
    Position string `gorm:"column:position"`
    JobOverview string `gorm:"column:job_overview"`
}

func (Experience) TableName() string {
    return "experience"
}

```

```

}

func (Experience) Create(fields map[string]string) (*Experience, error) {
    experience := Experience{}
    companyId, err := strconv.Atoi(fields["CompanyId"])

    experience.CompanyId = companyId
    experience.Position = fields["Position"]
    experience.JobOverview = fields["JobOverview"]
    experience.DateStart, err = time.Parse("2006-01-02", fields["DateStart"])
    experience.DateEnd, err = time.Parse("2006-01-02", fields["DateEnd"])
    if err != nil {
        return nil, err
    }
    return &experience, err
}

```

Таблиці `users_to_skills` та `connection` не мають власних атрибутів, а лише зовнішні ключі-посилання на таблиці, тому окремого визначення не потребують.

## Приклади запитів з використанням ORM

Вставка:

```

Select table:
Education
Experience
Connection
UserSkill
User
Skill
University
Company
Table: User
FirstName: ORM
LastName: TEST
Email: ENTITY

2024/12/02 22:48:00 D:/kpi_databases/lab2/model/repository/repository.go:15
[12.861ms] [rows:1] INSERT INTO "user" ("first_name","last_name","email") VALUES ('ORM','TEST','ENTITY') RETURNING "user_id"

```

*вставка запису користувача*

```

2024/12/02 22:53:48 D:/kpi_databases/lab2/model/repository/repository.go:29
[0.710ms] [rows:1] SELECT * FROM "user" WHERE "user"."user_id" = 3

2024/12/02 22:53:48 D:/kpi_databases/lab2/model/repository/repository.go:29
[0.548ms] [rows:1] SELECT * FROM "user" WHERE "user"."user_id" = 8

2024/12/02 22:53:48 D:/kpi_databases/lab2/model/repository/user_repository.go:20
[0.591ms] [rows:0] INSERT INTO "user" ("first_name","last_name","email","user_id") VALUES ('ORM','TEST','ENTITY',8) ON CONFLICT DO NOTHING RETURNING "user_id"

2024/12/02 22:53:48 D:/kpi_databases/lab2/model/repository/user_repository.go:20
[2.333ms] [rows:1] INSERT INTO "connection" ("u1","u2") VALUES (3,8) ON CONFLICT DO NOTHING
Main page

```

*створення зв'язку між користувачами*

## Редагування:

```
Enter primary key:
UserId: 8
Select columns or type '-' to end selecting:
FirstName
LastName
Email
Column:
Email
-
Email: EDITED

2024/12/02 22:54:53 D:/kpi_databases/lab2/model/repository/repository.go:29
[1.402ms] [rows:1] SELECT * FROM "user" WHERE "user"."user_id" = 8

2024/12/02 22:54:53 D:/kpi_databases/lab2/model/repository/repository.go:19
[5.524ms] [rows:1] UPDATE "user" SET "email"='EDITED' WHERE "user_id" = 8
```

## Видалення:

```
2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[0.766ms] [rows:1] SELECT * FROM "connection" WHERE "connection"."u1" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[0.533ms] [rows:1] SELECT * FROM "user" WHERE "user"."user_id" = 3

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[3.643ms] [rows:0] SELECT * FROM "education" WHERE "education"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[3.087ms] [rows:0] SELECT * FROM "experience" WHERE "experience"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[5.100ms] [rows:0] SELECT * FROM "users_to_skills" WHERE "users_to_skills"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/model.go:270
[18.993ms] [rows:1] SELECT * FROM "user" WHERE "user"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/repository/user_repository.go:47
[2.024ms] [rows:0] DELETE FROM "users_to_skills" WHERE "users_to_skills"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/repository/user_repository.go:51
[2.286ms] [rows:1] DELETE FROM "connection" WHERE "connection"."u1" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/repository/user_repository.go:55
[0.600ms] [rows:0] DELETE FROM "experience" WHERE "experience"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/repository/user_repository.go:55
[0.000ms] [rows:0] DELETE FROM "education" WHERE "education"."user_id" = 8

2024/12/02 23:10:53 D:/kpi_databases/lab2/model/repository/user_repository.go:55
[3.139ms] [rows:1] DELETE FROM "user" WHERE "user"."user_id" = 8
```



*видалення відбувається не тільки для запису самого користувача, а і усіх його зв'язків з іншими сутностями*

## Завдання №2

### BTree

Для тестування індексів створимо окрему таблицю:

```
DROP TABLE IF EXISTS btree_idx;  
CREATE TABLE btree_idx (  
    id SERIAL PRIMARY KEY NOT NULL,  
    str varchar(50),  
    attr int  
);
```

```
INSERT INTO btree_idx(str, attr)  
SELECT  
    gen_random_uuid(),  
    (random() * 100000)::int  
from generate_series(1, 100000);
```

Виконаємо тестові запити до бази та заміряємо їх швидкодію:

```
SELECT * FROM btree_idx;
```

✓ Successfully run. Total query runtime: 111 msec. 100000 rows affected. ✕

```
SELECT * FROM btree_idx WHERE attr = 55;
```

✓ Successfully run. Total query runtime: 98 msec. 1 rows affected. ✕

```
SELECT * FROM btree_idx WHERE str ilike '10%1';
```

✓ Successfully run. Total query runtime: 257 msec. 33 rows affected. ✕

```
SELECT * FROM btree_idx WHERE attr BETWEEN 1 AND 10;
```

✓ Successfully run. Total query runtime: 70 msec. 8 rows affected. ✕

Створимо індекс

```
CREATE INDEX btree ON btree_idx USING BTREE(attr);
```

Виконаємо ті ж запити:

```
SELECT * FROM btree_idx;
```

✓ Successfully run. Total query runtime: 94 msec. 100000 rows affected. ✕

```
SELECT * FROM btree_idx WHERE attr = 55;
```

✓ Successfully run. Total query runtime: 69 msec. 1 rows affected. ✕

```
SELECT * FROM btree_idx WHERE str ilike '10%1';
```

✓ Successfully run. Total query runtime: 263 msec. 33 rows affected. ✕

```
SELECT * FROM btree_idx WHERE attr BETWEEN 1 AND 10;
```

✓ Successfully run. Total query runtime: 66 msec. 8 rows affected. ✕

Як видно з результатів, після створення індекса для атрибута attr, швидкість запитів, пов'язана з фільтрацією за цим атрибутом, є вищою. При цьому, швидкість запитів, у яких фільтрація відбувається за іншими стовпцями, є приблизно такою ж, як і без використання індекса. При цьому, швидкість запитів, де обираються усі записи, або значна їх частина(80-90%), індекс не дає помітного ефекту. Хоча BTree індекс є найбільш універсальним, найбільше він пришвидшує виконання запитів з фільтрацією за атрибутами, які можна легко відсортувати, наприклад числа, тому часте виконання пошуку записів за атрибутом у межах діапазону, рівному значенню, мінімальному, максимальному є ґрунтовною причиною створити індекс саме цього типу.

### BRIN

Створимо таблицю для тестування BRIN індексу:

```
DROP TABLE IF EXISTS brin_idx;  
CREATE TABLE brin_idx (  
    id SERIAL PRIMARY KEY NOT NULL,  
    str varchar(50),  
    attr int  
);
```

```
INSERT INTO brin_idx(str, attr)
SELECT
gen_random_uuid(),
(random() * 100000)::int
from generate_series(1, 1000000);
```

Виконаємо різні запити:

```
SELECT * FROM BRIN_IDX;
```

✓ Successfully run. Total query runtime: 643 msec. 1000000 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE STR LIKE 'a*b%';
```

✓ Successfully run. Total query runtime: 314 msec. 3945 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE id < 1000;
```

✓ Successfully run. Total query runtime: 94 msec. 999 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE id BETWEEN 1 AND 10;
```

✓ Successfully run. Total query runtime: 123 msec. 10 rows affected. ✕

Створимо індекс:

```
CREATE INDEX brin ON brin_idx USING BRIN("id");
```

Виконаємо ті ж запити:

```
SELECT * FROM BRIN_IDX;
```

✓ Successfully run. Total query runtime: 587 msec. 1000000 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE STR LIKE 'a*b%';
```

✓ Successfully run. Total query runtime: 224 msec. 3945 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE id < 1000;
```

✓ Successfully run. Total query runtime: 82 msec. 999 rows affected. ✕

```
SELECT * FROM BRIN_IDX WHERE id BETWEEN 1 AND 10;
```

✓ Successfully run. Total query runtime: 74 msec. 10 rows affected. ✕

Як видно з результатів, використання BRIN дає невеликі переваги у швидкодії для запитів, пов'язаних з порівнянням атрибута id. Це відбувається через те, що цей тип індексу є ефективним, коли записи мають деяку кореляцію з їхнім фізичним положенням у таблиці, іншими словами, якщо запит вибірки без явного вказання ORDER BY дає приблизно відсортований у деякому порядку набір рядків. У випадку тестової таблиці, де атрибут id має тип SERIAL, при операції вибірки усі рядки таблиці будуть відсортовані за зростанням. При цьому, цей тип індекса має менші потреби у додатковій пам'яті і мінімальні ресурсні витрати на підтримку.

### Завдання №3

Створимо тригер для таблиці connection:

```
CREATE OR REPLACE FUNCTION BIDIRECTIONAL()
RETURNS TRIGGER AS $$
DECLARE
connections CURSOR(_u1 int) FOR SELECT * FROM "user" WHERE user_id in
(SELECT U2 FROM "connection" WHERE U1 = _u1);
BEGIN
    IF TG_OP = 'INSERT' THEN
        IF NEW.U1 = NEW.U2 THEN
            RAISE EXCEPTION 'Users cannot be connected to themself';
        END IF;
        FOR _user IN connections(_u1 := NEW.U1) LOOP
            RAISE NOTICE 'User %', _user;
        END LOOP;
        IF pg_trigger_depth() = 1 THEN
            INSERT INTO "connection"(U1, U2) VALUES (NEW.U2, NEW.U1);
        END IF;
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        IF pg_trigger_depth() = 1 THEN
            DELETE FROM "connection" WHERE U1 = OLD.U2 AND U2 = OLD.U1;
        END IF;
        RETURN OLD;
    END IF;
END;
```

```
$$ LANGUAGE PLPGSQL;
```

```
CREATE or REPLACE TRIGGER BIDIRECTIONAL_CONNECTION_PROVIDER  
BEFORE INSERT OR DELETE ON "connection"  
FOR EACH ROW  
EXECUTE FUNCTION BIDIRECTIONAL();
```

Функція тригера полягає в забезпеченні “симетричності” відношення `connection`, як цього вимагає предметна область. Без тригера можлива неконсистентність даних, а також ускладнюється логіка роботи з базою даних: перед кожною вставкою/видаленням/пошуком необхідно шукати `user_id` як в стовпчику `u1`, так і `u2`. Предметна область вимагає двосторонності зв’язків між користувачами, тобто щоб  $\{u1, u2\} = \{u2, u1\}$ . Після створення тригера кількість даних про зв’язки збільшилась вдвічі, проте відпала необхідність виконувати операції об’єднання двох запитів пошуку зв’язків зі сторони `u1` та `u2`, контролі вставки/видалення симетричного відповідника заданому запису, що підвищує ефективність роботи з базою даних та спрощує подальшу розробку. Також тригерна функція виконує валідацію на предмет спроби встановити зв’язок користувача самого з собою. Крім того, вона виводить усі контакти користувача за допомогою курсорного циклу, що не є важливим для бізнес-логіки, проте демонструє вміння користуватися ними.

Приклад використання тригера при вставці:

```
1 INSERT INTO "connection"(u1, u2) values (3, 304163);
```

Data Output	Messages	Notifications
-------------	----------	---------------

ЗАМЕЧАНИЕ:	User (2340980,Anastasia,Savchenko,anastasia_savchenko@mail.com)	
ЗАМЕЧАНИЕ:	User (4148679,Viktor,Kryvenko,viktor_kryvenko@mail.com)	
ЗАМЕЧАНИЕ:	User (5495058,Stepan,Petrenko,stepan_petrenko@mail.com)	
ЗАМЕЧАНИЕ:	User (537708,Serhii,Shevchenko,serhii_shevchenko@mail.com)	
ЗАМЕЧАНИЕ:	User (213662,Anna,Tkach,anna_tkach@mail.com)	
ЗАМЕЧАНИЕ:	User (293382,Ivanna,Korol,ivanna_korol@mail.com)	
INSERT 0	1	

38	304163	3
39	3	304163

Приклад використання тригера при видаленні:

1	DELETE FROM "connection" where u1 = 3 and u2 = 304163;	
Data Output	Messages	Notifications
DELETE 1		










1

```
select * from "connection"
```

Data Output

Messages

Notifications



	u1 [PK] bigint	u2 [PK] bigint
25	4437721	5130518
26	5488262	6251786
27	767172	1708851
28	1851842	2415336
29	5183157	6214752
30	1238339	591805
31	4621937	3287650
32	5512661	6406051
33	6959473	1
34	3	537708
35	3	213662
36	293382	3
37	3	293382

#### Завдання №4

#### Теоретичні відомості про рівні ізоляції транзакцій та проблеми при їх паралельному виконанні

Рівень ізолюваності транзакцій — значення, що задає рівень, при якому в транзакції дозволяються неузгоджені дані, тобто ступінь ізолюваності однієї транзакції від іншої. Більш високий рівень ізолюваності підвищує точність даних, але при цьому може знижуватись кількість транзакцій, що виконуються паралельно. З іншого боку, більш низький рівень ізолюваності дозволяє виконувати більше паралельних транзакцій, але знижує точність даних.

Проблеми паралельного виконання транзакцій:

- *Lost update* - при одночасній зміні одного блоку даних різними транзакціями, одна із змін втрачається
- *Dirty read* - читання даних, які додані чи змінені транзакцією, яка потім не підтвердиться
- *Non-repeatable read* - при повторному читанні в рамках однієї транзакції, раніше прочитані дані з'являються зміненими
- *Phantom read* - коли в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків

Для демонстрації кожного з рівнів ізоляції транзакцій та проблем їх паралельного виконання, створимо тестову таблицю:

	id [PK] bigint	product_name character varying	price money
1	1	Book	10,00 ?
2	2	Pen	1,00 ?
3	3	Eraser	3,50 ?

### Read Uncommitted

Найнижчий рівень ізоляції транзакцій, при якому одна транзакція може мати доступ до змін, внесених іншою транзакцією, які ще не були зафіксовані командою COMMIT. Фактично, цей рівень лише забезпечує послідовне виконання паралельних запитів на редагування даних, а отже захищає від втраченого оновлення. Read Uncommitted допускає усі можливі аномалії читання.

### Read Committed

Зазвичай, більшість СУБД використовують цей рівень ізоляції транзакцій за замовчуванням:

```
professional_social_network=# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
-----
read committed
(1 row)
```

Цей рівень ізоляції захищає від проблеми брудного читання - в різних СУБД це досягається різними способами. В PostgreSQL це робиться наступним чином: спроба читання даних в рамках транзакції, що були змінені в ході іншої, але не були зафіксовані командою COMMIT призведе до читання даних версії до початку виконання іншої транзакції, при чому

дані, що були змінені в рамках цієї ж транзакції, але не були зафіксовані, будуть доступні у їх найбільш актуальному вигляді. При цьому він не захищає від фантомного читання та неповторюваного читання.

```
заклики ігноруються
professional_social_network=# ROLLBACK;
ROLLBACK
professional_social_network=# BEGIN;
BEGIN
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,50 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,50 ?
(3 rows)

professional_social_network=#
```

```
professional_social_network=# BEGIN;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# ROLLBACK
professional_social_network=# ;
ROLLBACK
professional_social_network=# BEGIN;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,99 ?
(3 rows)

professional_social_network=#
```

## приклад використання Read Committed: незафіксовані зміни даних у транзакції справа не призводять до брудного читання

```
professional_social_network=# BEGIN;
BEGIN
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,50 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,50 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,99 ?
(3 rows)
```

```
8-bit characters might not work correctly. See psql refer
ence
page "Notes for Windows users" for details.
Type "help" for help.

professional_social_network=# BEGIN;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# ROLLBACK
professional_social_network=# ;
ROLLBACK
professional_social_network=# BEGIN;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          |  1,00 ?
 3 | Eraser       |  3,99 ?
(3 rows)

professional_social_network=# COMMIT;
COMMIT
```

при цьому залишається проблема неповторюваного читання



## Repeatable Read

Рівень ізоляції, при якому при виконанні транзакції дані, які були прочитані в її рамках, блокуються для редагування або видалення

```
-----
repeatable read
(1 row)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,50 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,50 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,50 ?
(3 rows)

professional_social_network=#
```

```
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,99 ?
(3 rows)

professional_social_network=# ROLLBACK;
ROLLBACK
professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL RE
PEATABLE READ;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE=3
.99 WHERE ID=3;
UPDATE 1
professional_social_network=# COMMIT;
COMMIT
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,99 ?
(3 rows)

professional_social_network=#
```

приклад використання Repeatable Read: навіть зафіксовані зміни в транзакції справа не призводять до неповторюваного або брудного читання

```
professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL RE
PEATABLE READ;
BEGIN
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,99 ?
(3 rows)

professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,99 ?
(3 rows)

professional_social_network=#
```

```
UPDATE 1
professional_social_network=# COMMIT;
COMMIT
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
 id | product_name | price 
-----+-----+-----
 1 | Book         | 10,00 ?
 2 | Pen          | 1,00 ?
 3 | Eraser       | 3,99 ?
(3 rows)

professional_social_network=# rollback;
ПРЕДУПРЕЖДЕНИЕ: нет незавершенной транзакции
ROLLBACK
professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL RE
PEATABLE READ;
BEGIN
professional_social_network=# INSERT INTO TRANSACTION_DEMO(ID, PR
ODUCT_NAME, PRICE) VALUES(4,'Ruler', 3.49);
INSERT 0 1
professional_social_network=#
```

при цьому залишається проблема фантомного читання

## Serializable

Найвищий рівень ізоляції транзакцій. Забезпечує повністю послідовний хід виконання транзакцій, блокуючи будь-які паралельні запити. Цей рівень транзакції захищає від усіх аномалій паралельного виконання, але при цьому є найбільш важким для СУБД, та значно сповільнює її роботу.

```

professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
1 | Book          | 10,00 ?
2 | Pen           | 1,00 ?
3 | Eraser        | 3,99 ?
(3 rows)

professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE = 3.20 WHERE ID =3;
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
professional_social_network=# CO

```

```

professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
professional_social_network=# INSERT INTO TRANSACTION_DEMO(ID, PRODUCT_NAME, PRICE) VALUES(4, 'Ruler', 3.49);
INSERT 0 1
professional_social_network=# rollback;
ROLLBACK
professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
professional_social_network=# UPDATE TRANSACTION_DEMO SET PRICE = 3.50 WHERE ID=3;
UPDATE 1
professional_social_network=# COMMIT;
COMMIT
professional_social_network=#

```

## приклад блокування оновлення даних

```

ROLLBACK
professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
professional_social_network=# SELECT * FROM TRANSACTION_DEMO;
id | product_name | price
-----+-----+-----
1 | Book          | 10,00 ?
2 | Pen           | 1,00 ?
3 | Eraser        | 3,50 ?
(3 rows)

professional_social_network=#

```

```

professional_social_network=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
professional_social_network=# INSERT INTO TRANSACTION(ID, PRODUCT_NAME, PRICE) VALUES(4, 'Ruler', 3.25);
ОШИБКА: отношение "transaction" не существует
LINE 1: INSERT INTO TRANSACTION(ID, PRODUCT_NAME, PRICE) VALUES(4, '...
^
professional_social_network=# INSERT INTO TRANSACTION_DEMO(ID, PRODUCT_NAME, PRICE) VALUES(4, 'Ruler', 3.25);
ОШИБКА: текущая транзакция прервана, команды до конца блока транзакции игнорируются
professional_social_network=#

```

## приклад блокування запису нових даних