



Міністерство освіти і науки України  
Національний технічний університет України  
«Київський політехнічний інститут»

## **Лабораторна робота № 2**

«Засоби оптимізації роботи СУБД PostgreSQL»

Виконав студент групи: КВ-11  
ПІБ: Горбуль А.О.  
Telegram: [@AOSity](#)  
Перевірив:

## Звіт

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 №4

Види індексів: *GIN, BRIN*

Умови для тригера: *after delete, insert*

### Пункт 1

#### «Система управління резюме та вакансіями»

Перелік сутностей і опис їх призначення:

**Користувач (User)**, сутність призначена для збереження даних про користувача системи – ім’я, вік, пошту

**Резюме (Resume)**, сутність призначена для збереження даних про резюме завантажене користувачем для подальшого відгуку на вакансію

**Вакансія (Vacancy)**, сутність призначена для збереження даних про вакансію – назву позицій, деталі роботи

#### Resume & Vacancy management system

User

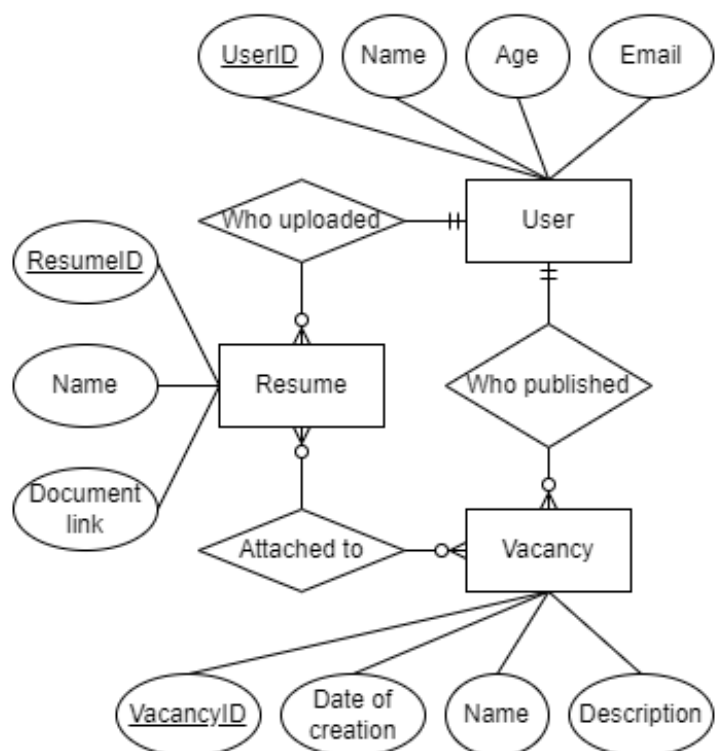
Resume

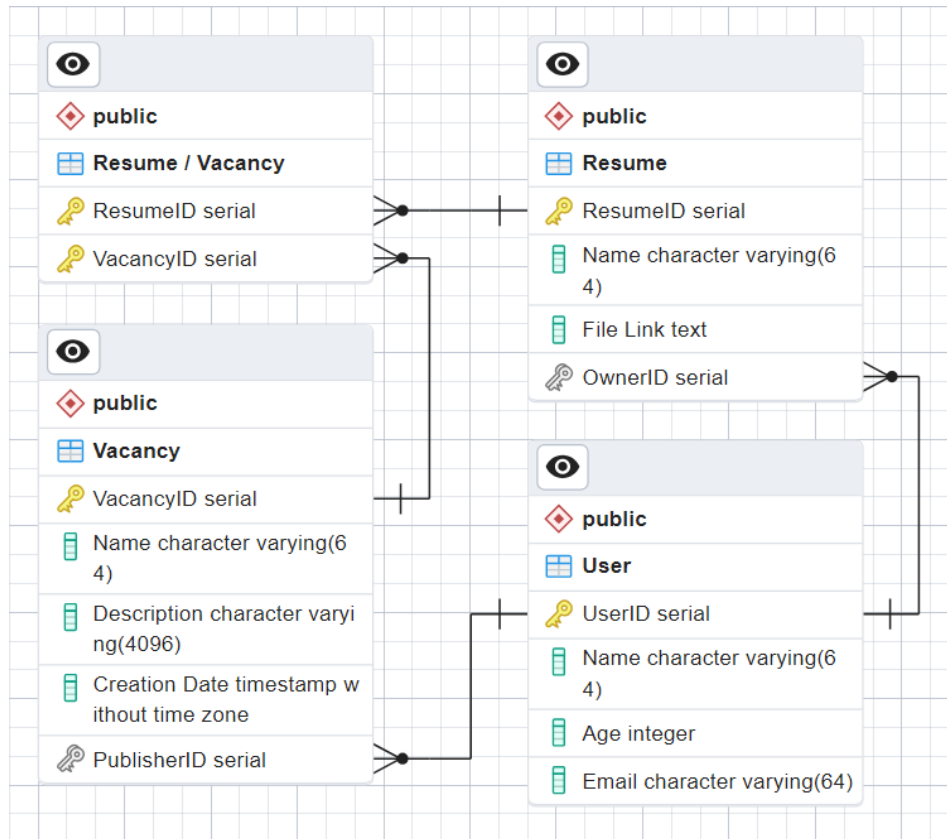
Vacancy

User can upload their resume

User can publish new vacancy

Resume can be attached to vacancy





Розроблені класи-сутності для об'єктів-сутностей:

```

class User(Base):
    __tablename__ = 'User'

    UserID = Column(Integer, primary_key=True, autoincrement=True)
    Name = Column(String(64), nullable=False)
    Age = Column(Integer, nullable=False)
    Email = Column(String(64), nullable=False)
    resumes = relationship('Resume', back_populates='owner')
    vacancies = relationship('Vacancy', back_populates='publisher')

class Resume(Base):
    __tablename__ = 'Resume'

    ResumeID = Column(Integer, primary_key=True, autoincrement=True)
    Name = Column(String(64), nullable=False)
    File_Link = Column(Text, nullable=False)
    OwnerID = Column(Integer, ForeignKey('User.UserID', onupdate='CASCADE',
ondelete='CASCADE'), nullable=False)
    owner = relationship('User', back_populates='resumes')

class Vacancy(Base):
    __tablename__ = 'Vacancy'

    VacancyID = Column(Integer, primary_key=True, autoincrement=True)
    Name = Column(String(64), nullable=False)
    Description = Column(String(4096))
  
```

```

    Creation_Date = Column(DateTime(timezone=False), server_default=func.now())
    PublisherID = Column(Integer, ForeignKey('User.UserID', onupdate='CASCADE',
ondelete='CASCADE'), nullable=False)
    publisher = relationship('User', back_populates='vacancies')

class ResumeVacancy(Base):
    __tablename__ = 'ResumeVacancy'

    ResumeID = Column(Integer, ForeignKey('Resume.ResumeID', onupdate='CASCADE',
ondelete='CASCADE'), primary_key=True)
    VacancyID = Column(Integer, ForeignKey('Vacancy.VacancyID', onupdate='CASCADE',
ondelete='CASCADE'), primary_key=True)
    resume = relationship('Resume')
    vacancy = relationship('Vacancy')

```

Було переписано модуль «Модель» у вигляд об'єктно-реляційної проекції (ORM). Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) залишились без змін. Для користувача все виглядає і працює так само як і у РГР, змінився лише бекенд. Детальніше можна подивитись в репозиторії, файл model.py.

## Пункт 2

Створимо тестову таблицю і заповнимо її 1000000 записами:

Query	Query History
1	<b>CREATE TABLE</b> index_test (
2	<b>id</b> serial <b>PRIMARY KEY</b> ,
3	vec tsvector,
4	txt TEXT
5	);
6	
7	<b>INSERT INTO</b> index_test (vec, txt)
8	<b>SELECT</b>
9	to_tsvector(md5(random()::text)),
10	md5(random()::text)
11	<b>FROM</b> generate_series(1, 1000000) <b>AS</b> id;
Data Output	Messages
INSERT 0 1000000	
Query returned successfully in 17 secs 333 msec.	

## Запит без використання індекса:

```
13 EXPLAIN ANALYZE SELECT * FROM index_test ORDER BY txt
```

Data Output	Messages	Notifications
<b>QUERY PLAN</b> text		
1	Gather Merge (cost=78320.00..175549.09 rows=833334 width=82) (actual time=3288.311..5766.917 rows=1000000 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Sort (cost=77319.98..78361.65 rows=416667 width=82) (actual time=3157.232..4101.779 rows=333333 loops=3)	
5	Sort Key: txt	
6	Sort Method: external merge Disk: 31400kB	
7	Worker 0: Sort Method: external merge Disk: 28992kB	
8	Worker 1: Sort Method: external merge Disk: 30336kB	
9	-> Parallel Seq Scan on index_test (cost=0.00..18487.67 rows=416667 width=82) (actual time=0.482..46.928 rows=333333 loop...)	
10	Planning Time: 1.262 ms	
11	Execution Time: 5817.845 ms	

Сортування без індексу зайняло майже 6 секунд, що дуже повільно

## Запит з використанням GIN індекса:











```
13 CREATE INDEX ON index_test using GIN (vec);  
14 EXPLAIN ANALYZE SELECT * FROM index_test ORDER BY vec
```

Data Output	Messages	Notifications
<b>QUERY PLAN</b> text		
1	Gather Merge (cost=78320.00..175549.09 rows=833334 width=82) (actual time=1140.405..2051.029 rows=1000000 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Sort (cost=77319.98..78361.65 rows=416667 width=82) (actual time=1064.719..1387.932 rows=333333 loops=3)	
5	Sort Key: vec	
6	Sort Method: external merge Disk: 32720kB	
7	Worker 0: Sort Method: external merge Disk: 26584kB	
8	Worker 1: Sort Method: external merge Disk: 31424kB	
9	-> Parallel Seq Scan on index_test (cost=0.00..18487.67 rows=416667 width=82) (actual time=0.472..65.663 rows=333333 loop...)	
10	Planning Time: 1.479 ms	
11	Execution Time: 2096.753 ms	

GIN індекс зменшив час виконання у 3 рази, що дуже хороший результат. Даний індекс створений для повнотекстового пошуку і добре оптимізований для великих рядків. Він працює з складеними типами даних. При цьому індексуються не самі значення, а окремі елементи. GIN індекс є ефективним для роботи з повнотекстовим пошуком, але через високу вартість оновлення індексу при вставці/оновленні даних не є ефективним.

### Запит з використанням BRIN індекса:










```
16 CREATE INDEX ON index_test using BRIN (txt);
17 EXPLAIN ANALYZE SELECT * FROM index_test ORDER BY txt;
```

Data Output		Messages	Notifications
		        	
	<b>QUERY PLAN</b> text <div>  </div>		
1	Gather Merge (cost=78320.00..175549.09 rows=833334 width=82) (actual time=3068.339..5637.045 rows=1000000 loops=1)		
2	Workers Planned: 2		
3	Workers Launched: 2		
4	-> Sort (cost=77319.98..78361.65 rows=416667 width=82) (actual time=2988.773..3955.087 rows=333333 loops=3)		
5	Sort Key: txt		
6	Sort Method: external merge Disk: 31400kB		
7	Worker 0: Sort Method: external merge Disk: 28992kB		
8	Worker 1: Sort Method: external merge Disk: 30336kB		
9	-> Parallel Seq Scan on index_test (cost=0.00..18487.67 rows=416667 width=82) (actual time=0.373..60.082 rows=333333 loop...)		
10	Planning Time: 1.492 ms		
11	Execution Time: 5689.590 ms		

З BRIN індексом час виконання трохи зменшився. Даний індекс поділяє дані на секції для того, щоб при пошуку не проходити по уже пройдених секціях, але в цьому випадку це йому не дуже допомагає.

Спробуємо інший запит. Запит без використання індекса:

```
19 EXPLAIN ANALYZE SELECT * FROM index_test WHERE id > 500000 AND id < 1000000
```

Data Output		Messages	Notifications
<div></div>			
	<b>QUERY PLAN</b> <div>text</div>		
1	Index Scan using index_test_pkey on index_test (cost=0.42..22677.65 rows=500511 width=82) (actual time=0.058..135.961 rows=499999 loop...		
2	Index Cond: ((id > 500000) AND (id < 1000000))		
3	Planning Time: 0.912 ms		
4	Execution Time: 152.626 ms		

### Запит з використанням BRIN індекса:

```
19 CREATE INDEX ON index_test using BRIN (id);
20 EXPLAIN ANALYZE SELECT * FROM index_test WHERE id > 500000 AND id < 1000000
```

Data Output		Messages	Notifications
<div> <div> <div>+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div> </div>			
	<div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div>		
1	Bitmap Heap Scan on index_test (cost=138.95..22093.89 rows=500511 width=82) (actual time=1.185..82.916 rows=499999 loops=...		
2	Recheck Cond: ((id > 500000) AND (id < 1000000))		
3	Rows Removed by Index Recheck: 8394		
4	Heap Blocks: lossy=7281		
5	-> Bitmap Index Scan on index_test_id_idx2 (cost=0.00..13.82 rows=508929 width=0) (actual time=0.180..0.180 rows=72810 loop=...		
6	Index Cond: ((id > 500000) AND (id < 1000000))		
7	Planning Time: 2.537 ms		
8	Execution Time: 98.747 ms		

З BRIN індексом час виконання зменшився майже вдвічі. BRIN-індекси призначені для обробки дуже великих таблиць і спроектовані для роботи з даними, які мають "природну кореляцію" за певним критерієм впорядкування, наприклад, коли значення у стовпці рівномірно зростають або спадають. В таких випадках BRIN може швидко ідентифікувати блоки даних, які містять в собі потрібні значення або діапазони значень, що забезпечує більш ефективний пошук і зменшення кількості даних, які потрібно обробляти.

### Пункт 3

Запит для створення триггеру:

```
CREATE TABLE IF NOT EXISTS public."Mailing"  
(  
    "Name" character varying(64) NOT NULL,  
    "Email" character varying(64) NOT NULL  
);  
  
CREATE OR REPLACE FUNCTION insert_mailing_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
    INSERT INTO public."Mailing" ("Name", "Email")  
    VALUES (NEW."Name", NEW."Email");  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER after_insert_user_trigger  
AFTER INSERT ON public."User"  
FOR EACH ROW  
EXECUTE FUNCTION insert_mailing_trigger();  
  
CREATE OR REPLACE FUNCTION delete_mailing_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
    DELETE FROM public."Mailing"  
    WHERE "Email" = OLD."Email";  
    RETURN OLD;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER after_delete_user_trigger  
AFTER DELETE ON public."User"  
FOR EACH ROW  
EXECUTE FUNCTION delete_mailing_trigger();
```

---

Data Output   **Messages**   Notifications

---

CREATE TRIGGER

Query returned successfully in 112 msec.

Цей тригер записує в нову таблицю "Mailing" ім'я та пошту користувача при його додаванні, та видаляє цей запис при видаленні користувача

Перевіримо

Додамо користувача за допомогою розробленої програми:

Menu:

```
1. Add
2. View
3. Update
4. Delete
5. Quit
Enter your choice: 1
1. User
2. Resume
3. Vacancy
4. Resume / Vacancy
5. Cancel
Enter your choice: 1
Enter user name: Trigger
Enter user age: 123
Enter user email: Test
```

User added successfully!

Перевіримо чи додалися дані в нову таблицю:

Query

Query History

1

2

SELECT \* FROM public."Mailing"

Data Output

Messages

Notifications

Тепер видалимо користувача:

Menu:

```
1. Add
2. View
3. Update
4. Delete
5. Quit
Enter your choice: 4
1. User
2. Resume
3. Vacancy
4. Cancel
```














Enter your choice: 1

Enter ID: 1

User deleted successfully!

Разом з користувачем по тригеру видалився і запис з “Mailing”:

Data Output	Messages	Notifications
        		
Name	Email	
character varying (64) 	character varying (64) 	

## Пункт 4

При паралельному виконанні транзакцій можливі такі феномени:

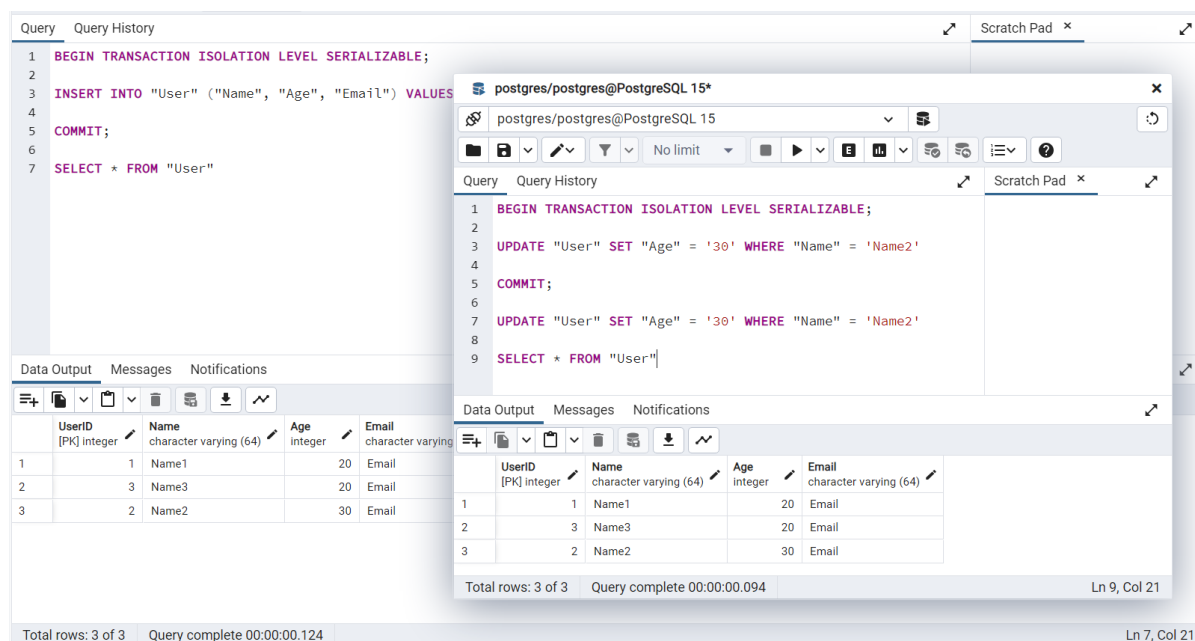
Втрачене оновлення – при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.

“Брудне” читання – читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (ролбекнеться).

Неповторюване читання – при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.

Фантомне читання – при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

Serializable – найбільш високий рівень ізольованості. Транзакції повністю ізолюються одна від одної. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими, що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку).



The screenshot displays a PostgreSQL IDE interface with two query windows and their corresponding data output tables.

**Query Window 1:**

```
1 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2
3 INSERT INTO "User" ("Name", "Age", "Email") VALUES
4
5 COMMIT;
6
7 SELECT * FROM "User"
```

**Data Output 1:**

UserID [PK] integer	Name character varying (64)	Age integer	Email character varying
1	Name1	20	Email
2	Name3	20	Email
3	Name2	30	Email

**Query Window 2:**

```
1 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2
3 UPDATE "User" SET "Age" = '30' WHERE "Name" = 'Name2'
4
5 COMMIT;
6
7 UPDATE "User" SET "Age" = '30' WHERE "Name" = 'Name2'
8
9 SELECT * FROM "User"
```

**Data Output 2:**

UserID [PK] integer	Name character varying (64)	Age integer	Email character varying (64)
1	Name1	20	Email
2	Name3	20	Email
3	Name2	30	Email

Total rows: 3 of 3    Query complete 00:00:00.124    Ln 9, Col 21

Repeatable read – рівень на якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Дані не можна змінити поки транзакція не закінчена).

Query

```
1 BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
2  
3 INSERT INTO "User" ("Name", "Age", "Email") VALUES ('Name4', 40, 'Email');  
4  
5 SELECT * FROM "User"  
6  
7 COMMIT;  
8  
9 SELECT * FROM "User"
```

Data Output

	UserID [PK] integer	Name character varying (64)	Age integer	Email character varying (64)
1	1	Name1	20	Email
2	3	Name3	20	Email
3	2	Name2	30	Email
4	4	Name4	40	Email

Total rows: 4 of 4 Query complete 00:00:00.097

Ln 1, Col 1

Read committed - рівень ізоляції, який гарантує, що будь-які прочитані дані будуть закомічені в момент читання.

Query

```
1 BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
2  
3 SELECT * FROM "User"  
4  
5 INSERT INTO "User" ("Name", "Age", "Email") VALUES ('Name5', 50, 'Email');  
6  
7 SELECT * FROM "User"  
8  
9 COMMIT;  
10  
11 SELECT * FROM "User"
```

Data Output

	UserID [PK] integer	Name character varying (64)	Age integer	Email character varying (64)
1	1	Name1	20	Email
2	3	Name3	20	Email
3	2	Name2	30	Email
4	4	Name4	40	Email
5	5	Name5	50	Email

Total rows: 5 of 5 Query complete 00:00:00.206

Ln 3, Col 1