



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря
Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих
комп'ютерних систем

Лабораторна робота №3
з дисципліни
«Бази даних і засоби управління»

Тема: «Засоби оптимізації роботи СУБД
PostgreSQL»

Виконав: студент III курсу

ФПМ групи КВ-81

Прокопчук М.О.

Перевірів:

Київ – 2020

Завдання

Загальне 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.

Деталізоване завдання:

1. Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlalchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:M, M:M та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

2. Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5 прикладів запитів SELECT (із виведенням результуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).
3. Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

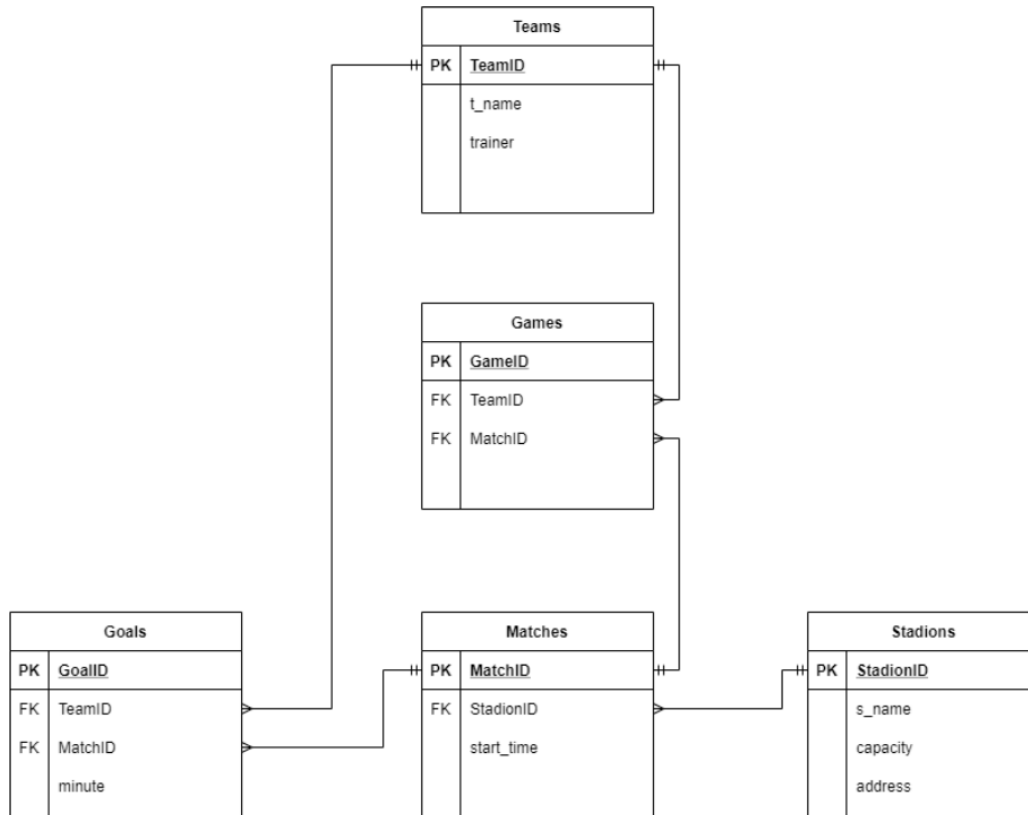
Варіант №11

11	GIN, Hash	before update, delete
----	-----------	-----------------------

Пункт №1

Для реалізації першого завдання було використано entity framework (database first approach).

Модель бази даних має такий вигляд:



Було створено такі класи:

Games:

```
15 references
public partial class Games : IModel
{
    7 references
    public override string ToString()
    {
        return DecorationLine() + $"|{GameId,-15}|{MatchId,-15}|{TeamId,-15}|";
    }

    7 references
    public string Properties()
    {
        return DecorationLine() + $"|{"GameId",-15}|{"MatchId",-15}|{"TeamId",-15}|";
    }

    17 references
    public string DecorationLine()
    {
        return "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+\n";
    }

    8 references
    public int GameId { get; set; }
    6 references
    public int MatchId { get; set; }
    6 references
    public int TeamId { get; set; }

    1 reference
    public virtual Matches Match { get; set; }
    1 reference
    public virtual Teams Team { get; set; }
}
```

Goals:

```
public partial class Goals : IModel
{
    public override string ToString()
    {
        return DecorationLine() + $"|{GoalId,-15}|{Minute,-15}|{TeamId,-15}|{MatchId,-15}|";
    }

    public string Properties()
    {
        return DecorationLine() + $"|{"GoalId",-15}|{"Minute",-15}|{"TeamId",-15}|{"MatchId",-15}|";
    }

    public string DecorationLine()
    {
        return "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+\n";
    }

    public int GoalId { get; set; }
    public int Minute { get; set; }
    6 references
    public int TeamId { get; set; }
    6 references
    public int MatchId { get; set; }

    1 reference
    public virtual Matches Match { get; set; }
    1 reference
    public virtual Teams Team { get; set; }
}
```

Matches:

```
16 references
public partial class Matches : IModel
{
    2 references
    public Matches()
    {
        Games = new HashSet<Games>();
        Goals = new HashSet<Goals>();
    }

    7 references
    public override string ToString()
    {
        return DecorationLine() + $"{MatchId,-15}|{StartTime,-15}|{StadiumId,-15}|";
    }

    7 references
    public string Properties()
    {
        return DecorationLine() + $"{MatchId",-15}|{StartTime,-15}|{StadiumId,-15}|";
    }

    17 references
    public string DecorationLine()
    {
        return "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+\n";
    }

    10 references
    public int MatchId { get; set; }
    7 references
    public DateTime StartTime { get; set; }
    6 references
    public int StadiumId { get; set; }

    1 reference
    public virtual Stadiums Stadium { get; set; }
    2 references
    public virtual ICollection<Games> Games { get; set; }
    2 references
    public virtual ICollection<Goals> Goals { get; set; }
}
```

Stadiums:

```
11 references
public partial class Stadiums : IModel
{
    1 reference
    public Stadiums()
    {
        Matches = new HashSet<Matches>();
    }

    7 references
    public override string ToString()
    {
        return DecorationLine() + $"{StadiumId,-15}|{SName,-15}|{Capacity,-15}|{Address,-15}|";
    }

    7 references
    public string Properties()
    {
        return DecorationLine() + $"{StadiumId",-15}|{SName,-15}|{Capacity,-15}|{Address,-15}|";
    }

    17 references
    public string DecorationLine()
    {
        return "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+\n";
    }

    8 references
    public int StadiumId { get; set; }
    5 references
    public string SName { get; set; }
    4 references
    public int Capacity { get; set; }
    4 references
    public string Address { get; set; }

    2 references
    public virtual ICollection<Matches> Matches { get; set; }
}
```

Teams:

```
public partial class Teams : IModel
{
    public Teams()
    {
        Games = new HashSet<Games>();
        Goals = new HashSet<Goals>();
    }

    public override string ToString()
    {
        return DecorationLine() + $"{TeamId,-15}|{TName,-15}|{Trainer,-15}|";
    }

    public string Properties()
    {
        return DecorationLine() + $"{TeamId",-15}|{TName,-15}|{Trainer,-15}|";
    }

    public string DecorationLine()
    {
        return "+" + new string('-', 15) + "+" + new string('-', 15) + "+" + new string('-', 15) + "+\n";
    }

    public int TeamId { get; set; }
    public string TName { get; set; }
    public string Trainer { get; set; }

    public virtual ICollection<Games> Games { get; set; }
    public virtual ICollection<Goals> Goals { get; set; }
}
```

Тепер запити CRUD виглядають таким чином (прикладі будуть надані для таблиці Stadiums):

Insert:

```
public Stadiums InsertStadium(string s_name, int capacity, string address)
{
    ctx.Stadiums.Add(new Stadiums
    {
        SName = s_name,
        Capacity = capacity,
        Address = address
    });
    ctx.SaveChanges();
    return ctx.Stadiums.OrderByDescending(u => u.StadiumId).FirstOrDefault();
}
```

Виклик Insert:

```
View.ShowRead(m.InsertStadium("MyStadium", 30000, "Myadrs"));
```

Результат Insert:

StadiumId	SName	Capacity	Address
100179	MyStadium	30000	Myadrs

Update:

```
public Stadiums UpdateStadium(string s_name, int capacity, string address, int StadiumID)
{
    Stadiums stadium = ctx.Stadiums.Single(u => u.StadiumId == StadiumID);
    stadium.SName = s_name;
    stadium.Capacity = capacity;
    stadium.Address = address;
    ctx.SaveChanges();
    return ctx.Stadiums.Single(u => u.StadiumId == StadiumID);
}
```

Виклик Update:

```
View.ShowRead(m.UpdateStadium("MyNewStadium", 40000, "MyNewadrs", 100179));
```

Результат Update:

StadiumId	SName	Capacity	Address
100179	MyNewStadium	40000	MyNewadrs

Delete:

```
public int DeleteStadium(int StadiumID)
{
    Stadiums stadium = ctx.Stadiums.Where(u => u.StadiumId == StadiumID).FirstOrDefault();
    ctx.Stadiums.Remove(stadium);
    return ctx.SaveChanges();
}
```

Або

```
public int DeleteStadiums()
{
    var stadiums = ctx.Set<Stadiums>();
    ctx.Stadiums.RemoveRange(stadiums);
    return ctx.SaveChanges();
}
```

Виклик Delete:

```
View.Report(m.DeleteStadium(100179).ToString());
```

Або

```
View.Report(m.DeleteStadiums().ToString());
```

У результаті повертає число видалених рядків.

Read:

```
public List<T> ReadData<T>() where T:class
{
    return ctx.Set<T>().ToList();
}
```

Виклик Read:

```
View.ShowRead(m.ReadData<Stadiums>().ToList<IModel>());
```






Результат Read:

StadiumId	SName	Capacity	Address	
100159	E0	39724	JFW	
100160	new st	421313	bonjornoi	
100161	new st	421313	bonjornoi	
100162	new st	421313	bonjornoi	
100163	new st	421313	bonjornoi	
100164	new st	421313	bonjornoi	
100165	new st	421313	bonjornoi	
100166	new st	421313	bonjornoi	
100167	new st	421313	bonjornoi	
100168	new st	421313	bonjornoi	
100169	new st	421313	bonjornoi	
100157	name	100000	new dads	
100170	dfad	12321	dwada	
100171	dfad	12321	dwada	
100172	dfad	12321	dwada	
100173	dfad	12321	dwada	
100174	dfad	12321	dwada	
100175	dfad	12321	dwada	
100176	dfad	12321	dwada	
100177	dfad	12321	dwada	
100178	dfad	12321	dwada	

Пункт №2

Згенеруємо 100 000 рядків за допомогою SQL запиту:

```
INSERT INTO public."Stadiums" ("s_name", "capacity", "address")
SELECT chr(trunc(65 + random() * 25)::int) || chr(trunc(65 + random() * 25)::int) || chr(trunc(65 + random() * 25)::int),
round(random() * 10000),
chr(trunc(65 + random() * 25)::int) || chr(trunc(65 + random() * 25)::int) || chr(trunc(65 + random() * 25)::int)
FROM generate_series(1, 100000)
```

	 StadiumID [PK] integer 	s_name character varying (30) 	capacity integer 	address character varying (60) 
1	200180	APE	1013	HSU
2	200181	ABL	6758	QXI
3	200182	OVT	5463	ROB
4	200183	TNT	465	MJJ
5	200184	OYA	1431	BIF
6	200185	NVL	9093	CQW
7	200186	KNN	6956	LLF
8	200187	BXH	4168	OQA
9	200188	QNF	3890	EEI
10	200189	GCF	3322	LEQ
11	200190	GKA	7013	QKN
12	200191	VVA	6690	COF
13	200192	RNM	9927	VRO
14	200193	IJY	1150	MVK
15	200194	RKN	5091	HAK
16	200195	FME	208	RRL
17	200196	JDP	1760	CCE
18	200197	DXV	5351	ENS
19	200198	DWQ	2815	CEH
20	200199	NHV	2469	WXT

Створимо GIN індекс:

```
1 CREATE INDEX GINIndex ON public."Stadiums" USING GIN (s_name gin_trgm_ops, address gin_trgm_ops)
```

Messages Explain Notifications Data Output

CREATE INDEX

Query returned successfully in 399 msec.

Створимо HASH індекс:

```
1 CREATE INDEX HASHIndex ON public."Stadiums" USING HASH(capacity)
```

Messages Explain Notifications Data Output

CREATE INDEX

Query returned successfully in 253 msec.

Проаналізуємо запити:

Вибірка всієї таблиці:

```
1 explain analyze SELECT * FROM public."Stadiums"
```

Messages Explain Notifications Data Output

QUERY PLAN		text	
1	Seq Scan on "Stadiums" (cost=0.00..2082.00 rows=100000 width=16) (actual time=0.387..12.188 rows=100000 loops=1)		
2	Planning Time: 0.077 ms		
3	Execution Time: 14.888 ms		

Для вибірки всієї таблиці індексування не використовується, так як вся інформація знаходиться послідовно в пам'яті.

Фільтрація:

```
1 explain analyze SELECT * FROM public."Stadiums" where s_name ilike '%aaa%';
```

Messages Explain Notifications Data Output

	QUERY PLAN	
	text	🔒
1	Bitmap Heap Scan on "Stadiums" (cost=16.08..53.32 rows=10 width=16) (actual time=0.020..0.030 rows=6 loops=1)	
2	Recheck Cond: ((s_name)::text ~* '%aaa%':text)	
3	Heap Blocks: exact=6	
4	-> Bitmap Index Scan on ginindex (cost=0.00..16.07 rows=10 width=0) (actual time=0.012..0.012 rows=6 loops=1)	
5	Index Cond: ((s_name)::text ~* '%aaa%':text)	
6	Planning Time: 0.206 ms	
7	Execution Time: 0.055 ms	

```
1 explain analyze SELECT * FROM public."Stadiums" where s_name ilike '%aaa%' or address ilike '%aaa%';
```

Messages Explain Notifications Data Output

	QUERY PLAN	
	text	🔒
1	Bitmap Heap Scan on "Stadiums" (cost=32.16..104.30 rows=20 width=16) (actual time=0.033..0.057 rows=11 loops=1)	
2	Recheck Cond: (((s_name)::text ~* '%aaa%':text) OR ((address)::text ~* '%aaa%':text))	
3	Heap Blocks: exact=11	
4	-> BitmapOr (cost=32.16..32.16 rows=20 width=0) (actual time=0.025..0.025 rows=0 loops=1)	
5	-> Bitmap Index Scan on ginindex (cost=0.00..16.07 rows=10 width=0) (actual time=0.011..0.011 rows=6 loops=1)	
6	Index Cond: ((s_name)::text ~* '%aaa%':text)	
7	-> Bitmap Index Scan on ginindex (cost=0.00..16.07 rows=10 width=0) (actual time=0.013..0.013 rows=5 loops=1)	
8	Index Cond: ((address)::text ~* '%aaa%':text)	
9	Planning Time: 0.324 ms	
10	Execution Time: 0.087 ms	

Видно, що у цих запитах було використано GIN. Це пов'язано з тим що знайти (вибрати частину таблиці) якійсь конкретні значення набагато простіше за допомогою індексування, ніж проходити всю таблицю порядково.

```
1 explain analyze SELECT * FROM public."Stadiums" where capacity between 5000 and 6000
```

Messages Explain Notifications Data Output

	QUERY PLAN	
	text	🔒
1	Seq Scan on "Stadiums" (cost=0.00..2582.00 rows=10032 width=16) (actual time=0.619..11.657 rows=9957 loops=1)	
2	Filter: ((capacity >= 5000) AND (capacity <= 6000))	
3	Rows Removed by Filter: 90043	
4	Planning Time: 0.087 ms	
5	Execution Time: 12.041 ms	

HASH індекс не може використовуватися для порівняння більше/менше, тому у цьому запиті використовуватися не може (більш детально у висновках).

```
1 explain analyze SELECT * FROM public."Stadiums" where capacity = 5000
```

Messages Explain Notifications Data Output

QUERY PLAN		text	
1	Bitmap Heap Scan on "Stadiums"	(cost=4.08..41.32 rows=10 width=16) (actual time=0.424..0.437 rows=10 loops=1)	
2	Recheck Cond: (capacity = 5000)		
3	Heap Blocks: exact=10		
4	-> Bitmap Index Scan on hashindex	(cost=0.00..4.08 rows=10 width=0) (actual time=0.385..0.385 rows=10 loops=1)	
5	Index Cond: (capacity = 5000)		
6	Planning Time: 0.074 ms		
7	Execution Time: 0.463 ms		

При знаходженні конкретного значення планувальник використовує HASH індексацію. HASH індекс дозволяє швидко виконати запит на рівність.

Агрегатні функції, групування:

```
2 explain analyze SELECT s_name, count(s_name) FROM public."Stadiums" group by s_name
```

Messages Explain Notifications Data Output

QUERY PLAN		text	
1	HashAggregate	(cost=2332.00..2480.59 rows=14859 width=4) (actual time=37.320..39.129 rows=15594 loops=1)	
2	Group Key: s_name		
3	-> Seq Scan on "Stadiums"	(cost=0.00..2082.00 rows=100000 width=4) (actual time=0.420..10.115 rows=100000 loops=1)	
4	Planning Time: 0.115 ms		
5	Execution Time: 40.308 ms		

У цьому випадку планувальник використовує тимчасову хеш-таблицю для групування записів. Операція HashAggregate не вимагає попередньо упорядкованого набору даних, натомість вона використовує великий обсяг пам'яті для матеріалізації проміжного результату.

Сортування:

```
2 explain analyze SELECT * FROM public."Stadiums" order by s_name
```

Messages Explain Notifications Data Output

	QUERY PLAN text	
1	Sort (cost=10386.82..10636.82 rows=100000 width=16) (actual time=424.243..451.413 rows=100000 loops=1)	
2	Sort Key: s_name	
3	Sort Method: external merge Disk: 2552kB	
4	-> Seq Scan on "Stadiums" (cost=0.00..2082.00 rows=100000 width=16) (actual time=0.540..13.245 rows=100000 loops=1)	
5	Planning Time: 0.098 ms	
6	Execution Time: 455.177 ms	

Postgres не використовує індексацію для сортування, так як для проведення сортування всієї таблиці потрібно повністю її просканувати. Послідовне сканування у цьому випадку набагато швидше, ніж індексне сканування.

Отже, HASH індекс зберігає не значення, а їхні хеші. Такий спосіб індексування зменшує розмір і збільшує швидкість на обробку полів. Запит з використанням індексів хешу буде порівнюватися не зі значенням поля, а із хеш-значенням потрібних хеш-полів.

Оскільки хеш-функції нелінійні, такий індекс неможливо сортувати. Це спричиняє неможливість використовувати порівняння більше/менше та "IS NULL" з цим індексом.

Великою перевагою є швидкість роботи ($O(1)$), а також те що, при додаванні нових значень в таблицю індекс не треба перебудовувати. Недоліком є чутливість до колізій і їх використовують тільки для порівняння.

GIN корисні, коли індекс повинен відображати багато значень в один рядок. GIN добре підходять для індексації значень масивів, а також для здійснення повнотекстового пошуку.

Перевагою є те, що:

- добре підходить для повнотекстового пошуку;
- добре підходить для пошуку напівструктурованих даних;
- добре працює для частого повторення елементів (і тому ідеально підходить для повнотекстового пошуку).

Пункт №3

Створимо тригер:

```
1 DECLARE
2   i public."Stadiums"%ROWTYPE;
3 BEGIN
4   if(TG_OP = 'UPDATE') THEN
5     IF new.capacity <= 0 THEN RAISE EXCEPTION 'Capacity is negative or zero';
6     END IF;
7     RETURN new;
8   ELSEIF (TG_OP = 'DELETE') THEN
9     FOR i IN (SELECT * FROM public."Stadiums" WHERE address = old.address) LOOP
10      UPDATE public."Stadiums" SET address = NULL WHERE "StadiumID" = i."StadiumID" AND "StadiumID" <> old."StadiumID";
11    END LOOP;
12    RETURN old;
13  END IF;
14  return NULL;
15 END;
```

```
CREATE TRIGGER tr BEFORE UPDATE OR DELETE ON public."Stadiums"
FOR EACH ROW EXECUTE PROCEDURE tr_fn_Stadiums()
```

Перевірка роботи тригера на Update:

```
1 UPDATE public."Stadiums"
2 SET "s_name"= 'Stadium' , "capacity" = 0, "address" = 'adrs' WHERE "StadiumID" = 200180
```

Messages Explain Notifications Data Output

ERROR: Capacity is negative or zero
CONTEXT: PL/pgSQL function tr_fn_stadiums() line 6 at RAISE
SQL state: P0001

При становленні значення кількості місць нуль або менше нуля видаємо помилку.

Перевірка роботи тригера на DELETE:

Початкова таблиця має такий вигляд:

```
1 INSERT INTO public."Stadiums"("s_name", "capacity", "address")
2 VALUES('st1', 3000, 'adr1');
3 INSERT INTO public."Stadiums"("s_name", "capacity", "address")
4 VALUES('st2', 3000, 'adr1');
5 INSERT INTO public."Stadiums"("s_name", "capacity", "address")
6 VALUES('st3', 4000, 'adr3');
```

	StadiumID [PK] integer	s_name character varying (30)	capacity integer	address character varying (60)
1	300184	st1	3000	adr1
2	300185	st2	3000	adr1
3	300186	st3	4000	adr3

Після видалення рядку отримуємо:

```
1 DELETE FROM public."Stadiums" where "StadiumID" = 300184
```

Messages Explain Notifications Data Output

DELETE 1

	StadiumID [PK] integer	s_name character varying (30)	capacity integer	address character varying (60)
1	300185	st2	3000	[null]
2	300186	st3	4000	adr3

Тригер знаходить в таблиці стадіони які мають такий самий адрес і замінює ці адреси на NULL.

Пункт №4

Код програми можна знайти за посиланням [тут](#) (репозиторій github).

master ▾

DataBaseLabs / Lab3 /

Go to file

Add file ▾

FaceandControl Delete ~ \$b3_ProkopchukMaksym.docx

f8db011 now History

..

Models

Add Lab3

1 minute ago

Controller.cs

Add Lab3

1 minute ago

DataBaseModel.png

Add Lab3

1 minute ago

Lab3_ProkopchukMaksym.docx

Add Lab3

1 minute ago

Program.cs

Add Lab3

1 minute ago

README.md

Add Lab3

1 minute ago

View.cs

Add Lab3

1 minute ago

README.md

DataBaseLabs

Лабораторна робота № 3. Завдання роботи полягає у наступному: Перетворити модуль "Модель" з шаблону MVC лабораторної роботи №2 у вигляд об'єктно-реляційної проєкції (ORM). Створити та проаналізувати різні типи індексів у PostgreSQL. Розробити тригер бази даних PostgreSQL.