

**Міністерство освіти і науки України  
Національний технічний університет України  
“Київський політехнічний інститут ім. Ігоря Сікорського”**

**Факультет прикладної математики  
Кафедра спеціалізованих комп’ютерних систем**

**Лабораторна робота № 3**  
з дисципліни «Бази даних і засоби управління»  
«Засоби оптимізації роботи СУБД PostgreSQL»

Виконав:  
студент групи КП-81  
Длубак Родіон

Перевірив:

---

---

---

## Завдання

*Завдання роботи полягає у наступному:*

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проєкції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.

### *Вимоги до пункту 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. Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

### *Вимоги до пункту завдання №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).

### *Вимоги до пункту завдання №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.

## Виконання роботи

### Графічна ER модель

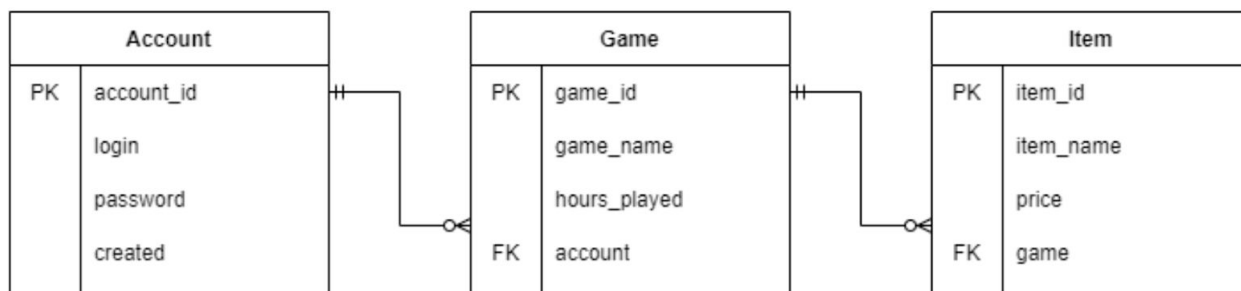


Рис 1. Логічна модель бази даних

### Сутності БД

Relation	Attribute	Data type
account	<code>account_id</code> - unique identifier <code>login</code> - account login <code>password</code> - account password <code>created</code> - the date the account was created	Integer String String Date
game	<code>game_id</code> - unique identifier <code>game_name</code> - name of the game <code>hours_played</code> - time spent in game <code>account</code> - ID of related account.	Integer String Integer Integer
item	<code>item_id</code> - unique identifier <code>item_name</code> - name of the item <code>price</code> - real cost of item (in USD) <code>game</code> - ID of related game	Integer String Numeric Integer

# Класи

```
class Account(Base):
    __tablename__ = 'account'

    account_id = Column(Integer, primary_key=True)
    login = Column(String)
    password = Column(String)
    created = Column(Date)

    games = relationship("Game")

    def __init__(self, login=None, password=None, created=None):
        self.login = login
        self.password = password
        self.created = created
```

```
class Game(Base):
    __tablename__ = 'game'

    game_id = Column(Integer, primary_key=True)
    game_name = Column(String)
    hours_played = Column(Integer)
    account = Column(Integer, ForeignKey('account.account_id'))

    items = relationship("Item")

    def __init__(self, game_name=None, hours_played=None, account=None):
        self.game_name = game_name
        self.hours_played = hours_played
        self.account = account
```

```
class Item(Base):
    __tablename__ = 'item'

    item_id = Column(Integer, primary_key=True)
    item_name = Column(String)
    price = Column(Numeric)
    game = Column(Integer, ForeignKey('game.game_id'))

    def __init__(self, item_name=None, price=None, game=None):
        self.item_name = item_name
        self.price = price
        self.game = game
```

## Запити ORM

```
def insert(self, tname, columns, values):
    columns = [c.strip() for c in columns.split(',')]
    values = [v.strip() for v in values.split(',')]

    pairs = dict(zip(columns, values))
    object_class = TABLES[tname]
    obj = object_class(**pairs)

    session.add(obj)
```

```
def get(self, tname, condition):

    object_class = TABLES[tname]
    objects = session.query(object_class)

    if condition:
        try:
            pairs = self.pairs_from_str(condition)
        except Exception as err:
            raise Exception('Incorrect input')
        objects = self.filter_by_pairs(objects, pairs, object_class)

    return list(objects)
```

```
def delete(self, tname, condition):
    pairs = self.pairs_from_str(condition)
    object_class = TABLES[tname]

    objects = session.query(object_class)
    objects = self.filter_by_pairs(objects, pairs, object_class)

    objects.delete()
```

```
def update(self, tname, condition, statement):
    pairs = self.pairs_from_str(condition)
    new_values = self.pairs_from_str(statement)
    object_class = TABLES[tname]

    objects = session.query(object_class)
    objects = self.filter_by_pairs(objects, pairs, object_class)

    for obj in objects:
        for field_name, value in new_values.items():
            setattr(obj, field_name, value)
```

# Індекси

## В-tree індекс:

```
CREATE INDEX treeIndex ON account using btree (account_id);
```

Запит з фільтрацією без використання індексу:

```
1 explain analyze select * from account where account_id<40000
```

Data Output Notifications Explain Messages

	QUERY PLAN
	text
1	Seq Scan on account (cost=0.00..305.07 rows=10006 width=28) (actual time=0.055..1.179 rows=10006 loops=1)
2	Filter: (account_id < 40000)
3	Planning Time: 0.089 ms
4	Execution Time: 1.376 ms

Пошук з фільтрацією з використанням b-tree:

```
1 explain analyze select * from account where account_id<150
```

Data Output Notifications Explain Messages

	QUERY PLAN
	text
1	Index Scan using treeindex on account (cost=0.29..8.30 rows=1 width=28) (actual time=0.340..0.342 rows=5 loops=1)
2	Index Cond: (account_id < 150)
3	Planning Time: 0.690 ms
4	Execution Time: 0.358 ms

## GIN індекс:

```
ALTER TABLE account  
ADD COLUMN ts_vector tsvector;
```

```
UPDATE account  
SET ts_vector = to_tsvector(login)  
WHERE true;
```

```
CREATE INDEX ginIndex ON account USING gin (ts_vector);
```

Звернення до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук без використання індексу):

```

1 EXPLAIN ANALYZE SELECT * FROM account a
2 WHERE a.password LIKE '%45e98390e5%'

```

Data Output   Notifications   Explain   Messages

	QUERY PLAN
	text
1	Seq Scan on account a (cost=0.00..305.07 rows=1 width=28) (actual time=0.355..1.571 rows=1 loops=1)
2	Filter: ((password)::text ~~ '%45e98390e5%':text)
3	Rows Removed by Filter: 10005
4	Planning Time: 0.759 ms
5	Execution Time: 1.592 ms

Звернення до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук відбувається за допомогою створеного індексу):

```

1 EXPLAIN ANALYZE SELECT * FROM account a
2 WHERE to_tsquery('45e98390e5') @@ ts_vector

```

Data Output   Notifications   Explain   Messages

	QUERY PLAN
	text
1	Bitmap Heap Scan on account a (cost=20.25..24.51 rows=1 width=51) (actual time=0.166..0.167 rows=0 loops=1)
2	Filter: (to_tsquery('45e98390e5':text) @@ ts_vector)
3	-> Bitmap Index Scan on ginindex (cost=0.00..20.25 rows=1 width=0) (actual time=0.165..0.165 rows=0 loops=1)
4	Index Cond: (ts_vector @@ to_tsquery('45e98390e5':text))
5	Planning Time: 1.084 ms
6	Execution Time: 0.297 ms

Висновки: і b-tree і GIN індекси виграють в швидкодії, в порівнянні з запитам без індексів. Індекс b-tree добре підходить для даних, які можна відсортувати. В нашому випадку це account\_id. GIN працює з типами даних, значення яких не є атомарними, а складаються з елементів, що підходить до обраної колонки login.



## Тригери

### BEFORE UPDATE

Якщо новий логін аккаунта збігається з вже існуючим, то виводимо відповідне повідомлення і змінюємо логін на 'duplicate'.






Код:

```
CREATE OR REPLACE FUNCTION before_update()
returns trigger
language plpgsql
AS $$
DECLARE
    logi text;
    log_id int;
BEGIN
    FOR logi, log_id IN
        SELECT login, account_id from account
    LOOP
        IF NEW.login = logi AND NEW.account_id != log_id THEN
            RAISE INFO 'LOGIN ALREADY EXISTS';
            NEW.login = 'duplicate';
            EXIT;
        END IF;
    END LOOP;
    return NEW;
END;
$$;
CREATE TRIGGER change_date_when_account_update
BEFORE UPDATE
ON "account"
FOR EACH ROW
EXECUTE PROCEDURE before_update();
```

Приклади результатів:

Аккаунти до змін:

```
1 select * from account
```

	Data Output	Notifications	Explain	Messages
	 account_id [PK] integer 	login character varying (100) 	password character varying (100) 	created date 
1	30110	sfsfs	dfddsdf	2020-01-01
2	1	login1	Frank123	2019-05-21
3	3	asfasdasd@gmail.com	Bbk2ng1969	2017-06-01
4	2	kekl	dolph1ns!	2018-02-03
5	4	somelogin	gladiator7	2018-11-02
6	5	anotherone	heineken	2020-09-22



Змінимо логін аккаунта з ідентифікатором 30110 на вже існуючий:

Отримуємо відповідне повідомлення:

```
1 UPDATE account
2 SET login = 'login1'
3 WHERE account_id=30110;
```

Data Output   Notifications   Explain   **Messages**

ИНФОРМАЦИЯ: LOGIN ALREADY EXISTS  
UPDATE 1

Query returned successfully in 46 msec.

Логін змінено на 'duplicate':

```
1 select * from account
2
```

Data Output   Notifications   Explain   Messages

	account_id [PK] integer	login character varying (100)	password character varying (100)	created date
1	30110	duplicate	dfddsdfd	2020-01-01
2	1	login1	Frank123	2019-05-21
3	3	asfasdasd@gmail.com	Bbk2ng1969	2017-06-01

При зміні логіну на такий, що не існує в БД:

```
1 UPDATE account
2 SET login = 'newlogin'
3 WHERE account_id=30110;
4
```

Data Output   Notifications   Explain   **Messages**

UPDATE 1

Query returned successfully in 69 msec.

```
1 select * from account
```

Data Output   Notifications   Explain   Messages

	account_id [PK] integer	login character varying (100)	password character varying (100)	created date
1	30110	newlogin	dfddsdf	2020-01-01
2	1	login1	Frank123	2019-05-21
3	3	asfasdasd@gmail.com	Rhk2nn1969	2017-06-01

Тригер не спрацьовує.

## BEFORE DELETE

При видаленні даних з таблиці 'items' вони переносяться в таблицю 'removedItems'.

Код:

```
CREATE OR REPLACE FUNCTION before_update()
returns trigger
language plpgsql
AS $$
BEGIN
    INSERT INTO removedItems
    ( item_id,
      item_name,
      price,
      game)
    VALUES
    ( NEW.item_id,
      NEW.item_name,
      NEW.price,
      NEW.game );
END;
$$;
CREATE TRIGGER move_deleted_items
BEFORE DELETE
ON "item"
FOR EACH ROW
EXECUTE PROCEDURE before_update();
```

Приклади результатів:

```
delete from item where item_id = 15
```

```
1 select * from removedItems
```

Data Output   Notifications   Explain   Messages

	item_id integer	item_name character varying (30)	price numeric (5)	game integer
1	15	some item	3213	2

Видалимо ще один запис в 'item'

```
1 delete from item where item_id =30111
```

Data Output   Notifications   Explain   **Messages**

DELETE 1

Query returned successfully in 45 msec.

```
1 select * from removedItems
```

Data Output   Notifications   Explain   Messages

	<b>item_id</b> integer	<b>item_name</b> character varying (30)	<b>price</b> numeric (5)	<b>game</b> integer
1	15	some item	3213	2
2	30111	somename	22	2