

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**

*на тему: “* *Засоби оптимізації роботи СУБД PostgreSQL”*

Виконав:

студент ІII курсу

групи КВ-01

Роговий Д.С.

Перевірив: Павловський В. І.

Київ – 2022

**Варіант 11**

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.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

**Логічна модель бази даних**

Нижче (Рисунок 1) наведено логічну модель бази даних:

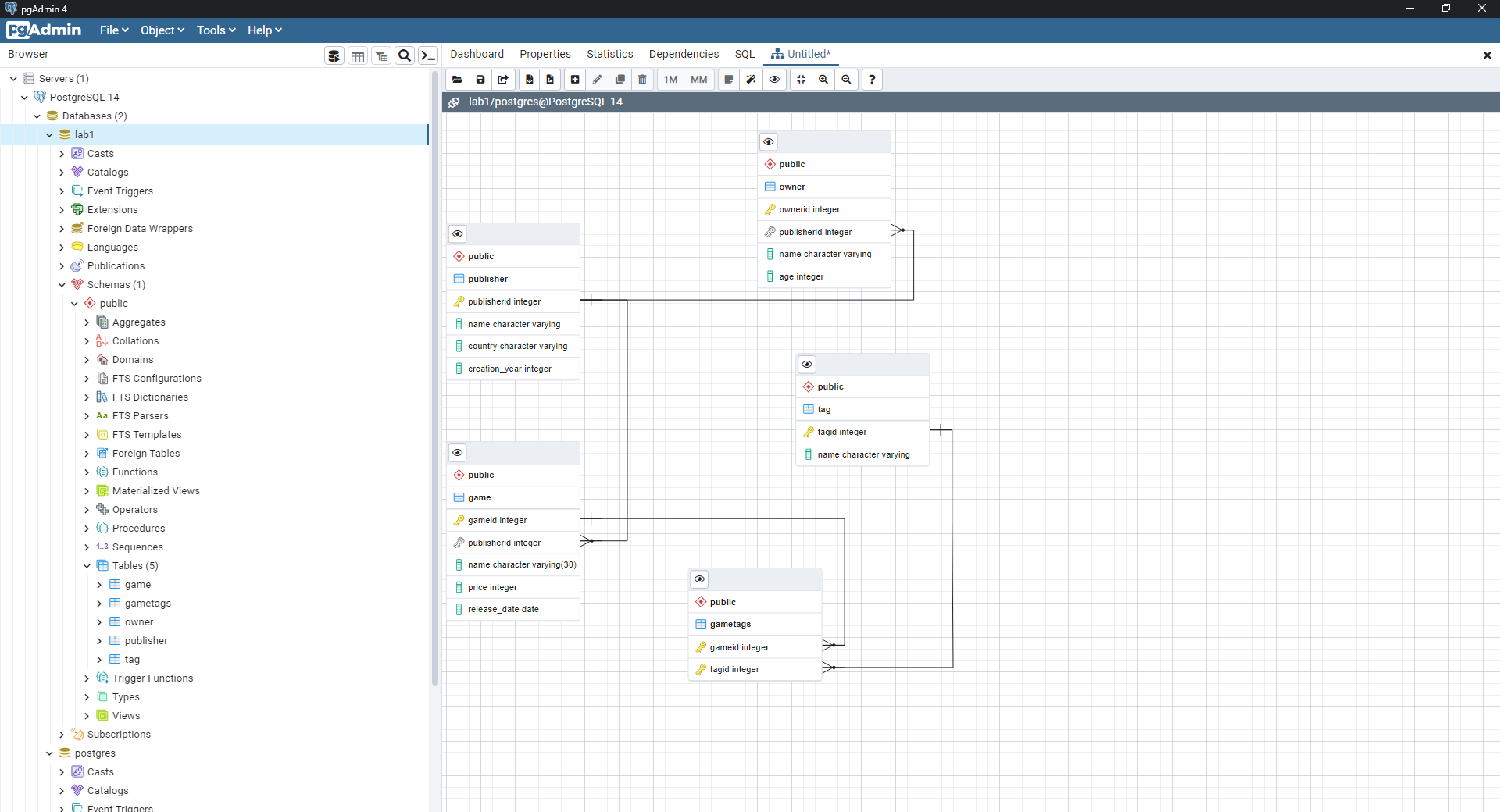


Рисунок 1 – Логічна модель бази даних

**Класи ORM**

Model.py

import psycopg2

from psycopg2 import sql

import sqlalchemy

from sqlalchemy import create\_engine,Column,Integer,String,ForeignKey,Date

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship, backref, sessionmaker

import time

engine = create\_engine('postgresql://postgres:101002@localhost:5432/lab1', echo=True)

Session = sessionmaker(bind = engine)

Shop = declarative\_base()

\_\_context = psycopg2.connect(host="localhost",port="5432",database="lab1",user="postgres",password="101002")

\_\_cursor = \_\_context.cursor()

class Publisher(Shop):

\_\_tablename\_\_='publisher'

publisherid=Column(Integer, primary\_key=True)

name = Column(String)

country = Column(String)

creation\_year = Column(Integer)

game = relationship('Game',order\_by="Game.gameid")

owner = relationship('Owner',uselist=False,back\_populates="publisher")

class Owner(Shop):

\_\_tablename\_\_ = 'owner'

ownerid = Column(Integer, primary\_key=True)

publisherid=Column(Integer,ForeignKey('publisher.publisherid'))

name = Column(String)

age = Column(Integer)

publisher=relationship('Publisher',uselist=False,back\_populates="owner")

class Game(Shop):

\_\_tablename\_\_ = 'game'

gameid = Column(Integer, primary\_key=True)

publisherid = Column(Integer, ForeignKey('publisher.publisherid'))

name = Column(String)

price = Column(Integer)

release\_date = Column(Date)

tags = relationship("Tag", secondary = 'gametags')

publisher = relationship("Publisher",back\_populates="game")

class Tag(Shop):

\_\_tablename\_\_ = 'tag'

tagid = Column(Integer, primary\_key=True)

name = Column(String)

games = relationship("Game", secondary = 'gametags')

class GameTags(Shop):

\_\_tablename\_\_ = 'gametags'

gameid = Column(Integer,ForeignKey('game.gameid'),primary\_key = True)

tagid = Column(Integer,ForeignKey('tag.tagid'),primary\_key = True)

colpub = ["name","country","creation\_year"]

colown = ["publisherid","name","age"]

colgame = ["publisherid","name","price","release\_date"]

coltag = ["name"]

colgt = ["gameid","tagid"]

columns = dict([("publisher",colpub),("owner",colown),("game",colgame),("tag",coltag),("gametags",colgt)])

def insert(tablename,data1):

session = Session()

data = {}

column = columns[tablename]

data.update(zip(column,data1))

try:

if tablename == "publisher":

for i in data1:

data.append("")

new = Publisher(name = data["name"],country=data["country"],creation\_year=data["creation\_year"])

session.add(new)

session.commit()

session.close()

if tablename == "owner":

new = Owner(name = data["name"],publisherid=data["publisherid"],age=data["age"])

session.add(new)

session.commit()

session.close()

if tablename == "game":

new = Game(name = data["name"],publisherid=data["publisherid"],price=data["price"],release\_date=data["release\_date"])

session.add(new)

session.commit()

session.close()

if tablename == "tag":

new = Tag(name = data["name"])

session.add(new)

session.commit()

session.close()

if tablename == "gametags":

new = GameTags(tagid=data["tagid"],gameid=data["gameid"])

session.add(new)

session.commit()

session.close()

except Exception as e:

print(e)

def update(tablename,dataid,data1):

session = Session()

data = {}

column = columns[tablename]

data.update(zip(column,data1))

try:

if tablename == "publisher":

edit = session.query(Publisher).get(dataid)

edit.name = data["name"]

edit.country = data["country"]

edit.creation\_year = data["creation\_year"]

session.commit()

session.close()

if tablename == "owner":

edit = session.query(Owner).get(dataid)

edit.name = data["name"]

edit.publisherid = data["publisherid"]

edit.age = data["age"]

session.commit()

session.close()

if tablename == "game":

edit = session.query(Game).get(dataid)

session.commit()

session.close()

if tablename == "tag":

edit = session.query(Tag).get(dataid)

session.commit()

session.close()

if tablename == "gametags":

edit = session.query(GameTags).filter(GameTags.gameid==dataid).first()

edit.gameid = data["gameid"]

edit.tagid = data["tagid"]

session.commit()

session.close()

except Exception as e:

print(e)

def delete(tablename,dataid):

session = Session()

try:

if tablename == "publisher":

new = session.query(Publisher).get(dataid)

session.delete(new)

session.commit()

session.close()

if tablename == "owner":

new = session.query(Owner).get(dataid)

session.delete(new)

session.commit()

session.close()

if tablename == "game":

new = session.query(Game).get(dataid)

session.delete(new)

session.commit()

session.close()

if tablename == "tag":

new = session.query(Tag).get(dataid)

session.delete(new)

session.commit()

session.close()

if tablename == "gametags":

new = session.query(GameTags).filter(GameTags.gameid==dataid).first()

session.delete(new)

session.commit()

session.close()

except Exception as e:

print(e)

return 0

def search\_data(flag):

result = list(tuple())

if flag == 1:

string = "SELECT \* FROM test WHERE test\_text[1] LIKE '%zx%'"

elif flag == 2:

string = f"SELECT \* FROM tag WHERE name = '{input('Write name: ')}'"

elif flag == 3:

string = f"SELECT \* FROM publisher WHERE name = '{input('Write name: ')}'"

try:

start\_time=time.time()

\_\_cursor.execute(string)

stop\_time = time.time()

result = \_\_cursor.fetchall()

print("Time to search:",stop\_time-start\_time)

#return result

except Exception as e:

print(e)

def get\_column\_types(self, table\_name):

self.\_\_cursor.execute("""SELECT column\_name, data\_type FROM information\_schema.columns WHERE table\_name = %(TableName)s ORDER BY table\_schema, table\_name""", {'TableName':table\_name})

return self.\_\_cursor.fetchall()

def get\_foreign\_key\_info(table\_name):

\_\_cursor.execute("""

SELECT kcu.column\_name, ccu.table\_name AS foreign\_table\_name,ccu.column\_name AS foreign\_column\_name

FROM information\_schema.table\_constraints AS tc

JOIN information\_schema.key\_column\_usage AS kcu

ON tc.constraint\_name = kcu.constraint\_name

AND tc.table\_schema = kcu.table\_schema

JOIN information\_schema.constraint\_column\_usage AS ccu

ON ccu.constraint\_name = tc.constraint\_name

AND ccu.table\_schema = tc.table\_schema

WHERE tc.constraint\_type = 'FOREIGN KEY' AND tc.table\_name=%s;""", (table\_name,))

return \_\_cursor.fetchall()

def generate\_data(self, table\_name, count):

types = get\_column\_types(table\_name)

fk\_array = get\_foreign\_key\_info(table\_name)

select\_subquery = ""

insert\_query = "INSERT INTO " + table\_name + " ("

for i in range(1, len(types)):

t = types[i]

name = t[0]

type = t[1]

fk = [x for x in fk\_array if x[0] == name]

if fk:

select\_subquery += ('(SELECT {} FROM {} ORDER BY RANDOM(), ser LIMIT 1)'.format(fk[0][2], fk[0][1]))

elif type == 'integer':

select\_subquery += 'trunc(random()\*100)::INT'

elif type == 'character varying':

select\_subquery += 'chr(trunc(65 + random()\*25)::INT) || chr(trunc(65 + random()\*25)::INT)'

elif type == 'date':

select\_subquery += """ date(timestamp '2014-01-10' + random() \* (timestamp '2020-01-20' - timestamp '2014-01-10'))"""

elif type == 'time without time zone':

select\_subquery += "time '00:00:00' + DATE\_TRUNC('second',RANDOM() \* time '24:00:00')"

else:

continue

insert\_query += name

if i != len(types) - 1:

select\_subquery += ','

insert\_query += ','

else:

insert\_query += ') '

try:

\_\_cursor.execute(insert\_query + "SELECT " + select\_subquery + "FROM generate\_series(1," + str(count) + ") AS ser")

\_\_context.commit()

except Exception as e:

print(e)

View.py

import Model

table\_names = list(["publisher","owner","game","tag","gametags"])

class View():

def \_\_init\_\_(self):

self.table\_names=''

def main\_menu(self):

print("Choose a task:")

print("1.Insert data")

print("2.Remove data")

print("3.Edit data")

print("4.Generate data")

print("5.Search data")

task\_number=int(input("Enter number of choosen table:"))

if 0 < task\_number < 6:

return task\_number

else:

return -1

def output\_table\_names(self):

print("Choose table to work with:")

i=0

while i<len(table\_names):

print(str(i+1)+'.'+ table\_names[i])

i+=1

return table\_names[int(input("Enter number of choosen table:"))-1]

def output\_table\_data(self, values):

if type(values) is list:

for i in values:

print(i)

else:

print(values[0])

for i in range(len(values[1])):

print(values[1][i])

def search\_menu():

print("Choose data to search:")

print("1. Search test like '%zx%'")

print("2. Search tag")

print("3. Search publisher company")

Controller.py

from Model import \*

from View import View

import time

import os

table\_names = list(["publisher","owner","game","tag","gametags"])

class Controler():

def \_\_init\_\_(self):

self.view=View()

def insert\_menu(self):

table\_name = self.view.output\_table\_names()

data = list()

for i in columns[table\_name]:

data.append(input(f"Enter {i}"))

insert(table\_name,data)

def delete\_menu(self):

table\_name = self.view.output\_table\_names()

dataid = input("Enter ID")

delete(table\_name,dataid)

def change\_menu(self):

table\_name = self.view.output\_table\_names()

data = list()

dataid = input("Enter ID")

for i in columns[table\_name]:

data.append(input(f"Enter {i}"))

update(table\_name,dataid,data)

def search\_menu(self):

View.search\_menu()

n = int(input("Choose what u want to find"))

data = search\_data(n) if 1 <= n <= 3 else print("Wrong parameter")

if len(data) == 0:

print("Nothing found")

else:

self.view.output\_table\_data(data)

if int(input("Type 1 to continue working")) == 1:

print('|\n'\*20)

self.main\_control()

def generate\_menu(self):

table = View.output\_table\_names()

print('='\*70)

generate\_data(table, int(input("Write how much to generate: ")))

if int(input("Type 1 to continue working")) == 1:

print('|\n'\*20)

self.main\_control()

def main\_control(self):

flag = self.view.main\_menu()

if flag == 1:

print('='\*70)

self.insert\_menu()

elif flag == 2:

print('='\*70)

self.delete\_menu()

elif flag == 3:

print('='\*70)

self.change\_menu()

elif flag == 4:

print('='\*70)

self.generate\_menu()

elif flag == 5:

print('='\*70)

self.search\_menu()

elif flag ==-1:

print('\*'\*70)

print("invalid input")

time.sleep(3)

print('\*'\*70)

self.main\_control()

if \_\_name\_\_ == '\_\_main\_\_':

controller = Controler()

controller.main\_control()

**Приклад запитів ORM**

Додавання нового рядка до таблиці

new = Tag(name =’test’)

s.add(new)

Видалення рядка з таблиці

new = s.querry(Tag).filter(GameTags.gameid=5).first()

Редагування рядка

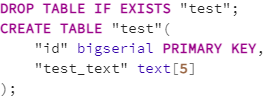
edit = session.query(GameTags).filter(GameTags.gameid==3).first()

edit.gameid = 3

edit.tagid = 1

**GIN**

Для дослідження індексу була створена таблиця, яка має дві колонки: test\_text типу text[] і id типу integer. Колонка test\_text проіндексована як GIN. У таблицю занесено 1000000 записів за допомогою запита на генерацію з минулої лабораторної роботи



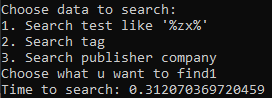


Рисунок 2 – швидкість пошуку без індексації



Рисунок 3 – Створення індекса

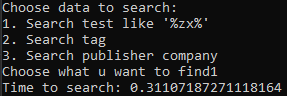


Рисунок 4 – швидкість пошуку з індексацією

GIN – індексація дає незначний приріст в швидкості при пошуку. Це пов’язано з порівняно малою кількістю елементів в таблиці. Також варто додати що індекс доволі довго створювався

**HASH**

Для дослідження індексу скористаємося таблицею tag. Спочатку занесемо у таблицю 1000000 значень за допомогою запита на генерацію з минулої лабораторної роботи.

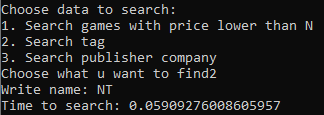


Рисунок 5 – швидкість пошуку без індексації



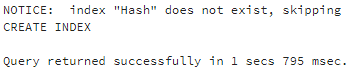


Рисунок 6 – створення індексу

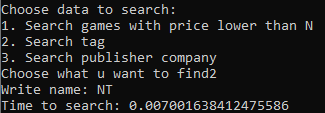


Рисунок 7 – швидкість пошуку з індексацією

Як можна побачити, HASH-індексація дає величезний(майже в десять разів) приріст у швидкості. Пов’язано це з тим, що HASH-індексація прекрасно працює саме з пошуком конкретних даних (оператор =).

**Тригери бази данний PostgreSQL. Умова для тригера – before update, delete**

**Тригер:**

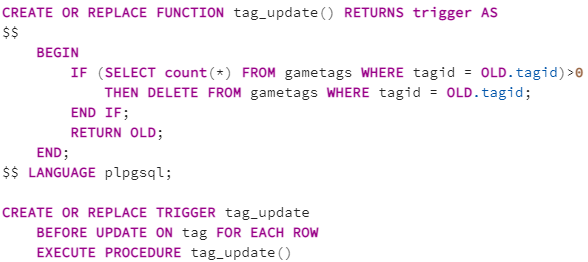
****

Рисунок 8 – створення тригера

**Принцип роботи тригеру**

Тригер спрацьовує перед оновленням таблиці tag. При зміні назви будь-якого тегу, будуть видалені усі рядки з таблиці gametags, де був індекс тегу, що буде оновлюватися.

**Приклад роботи**

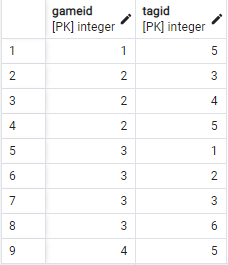
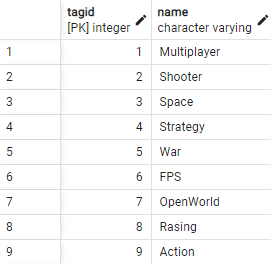


Рисунок 9 – таблиці до редагування



Рисунок 10 – Змінюємо назву тегу

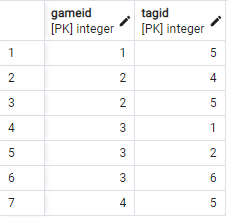
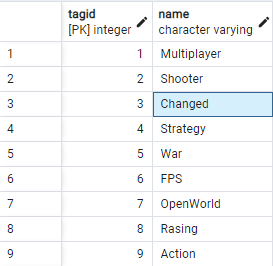


Рисунок 11 – таблиці після редагування

**Рівні ізоляції транзакцій у PostgreSQL**

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім невиконуються. Ізольованість транзакції показує те, наскільки сильно вони впливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у виборі між високою узгодженістю даних між транзакціями та швидкістю виконання цих транзакцій.

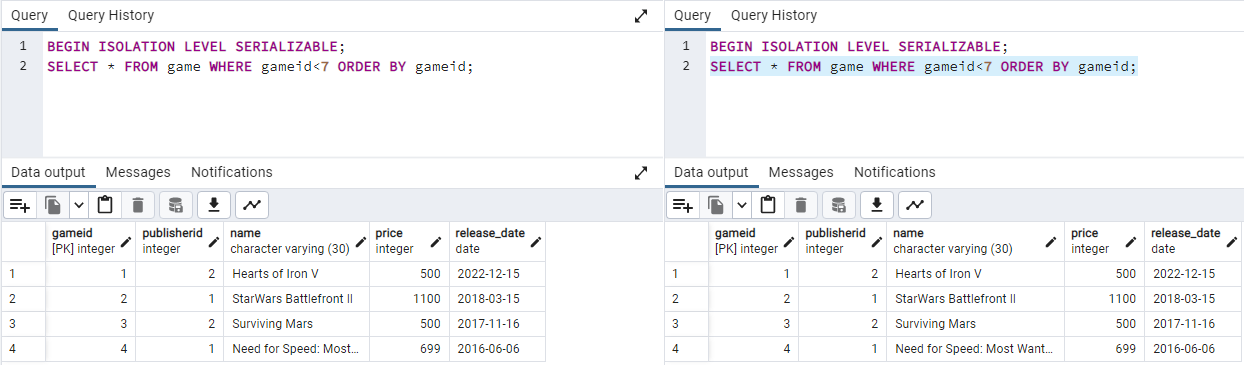
Варто зазначити, що найвищу швидкість виконання та найнижчу узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та найвищу узгодженість — serializable.

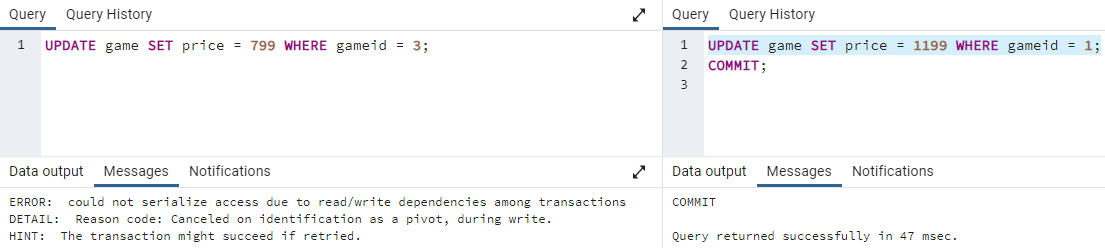
При паралельному виконанні транзакцій можливі виникненя таких проблем:

1. Втрачене оновлення - ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.
2. «Брудне» читання- читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).
3. Неповторюване читання - ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.
4. Фантомне читання - ситуація, коли 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).

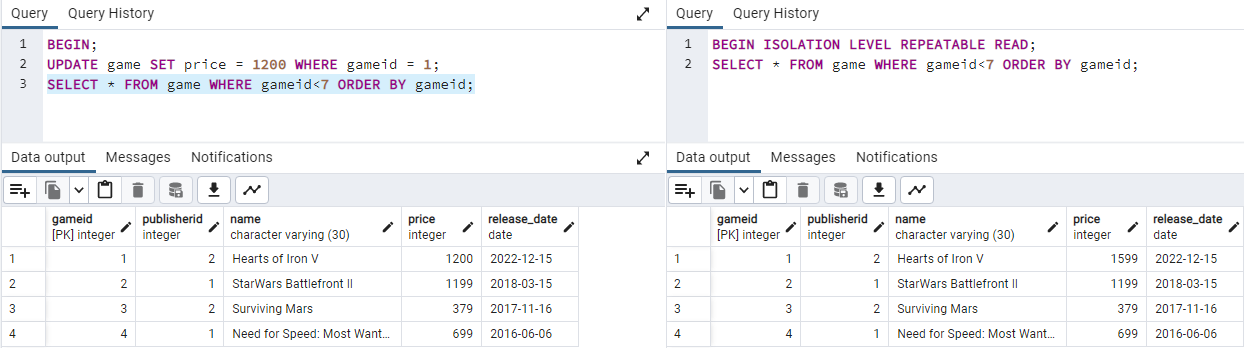


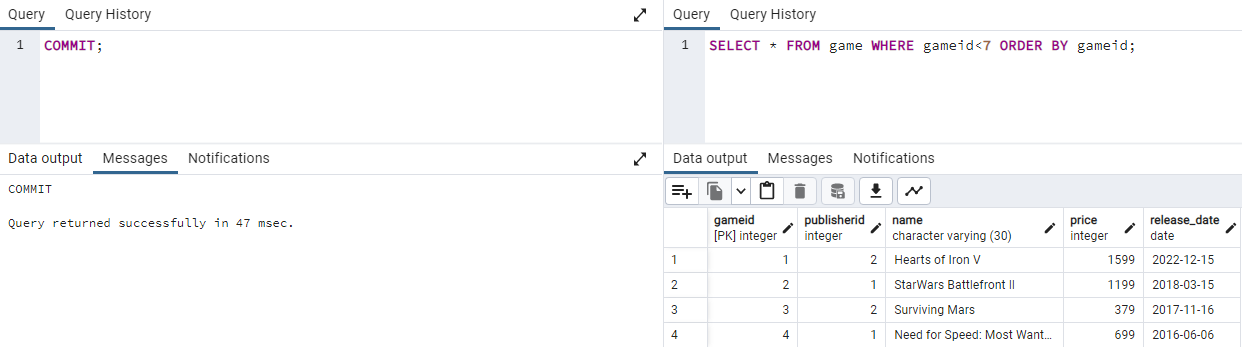


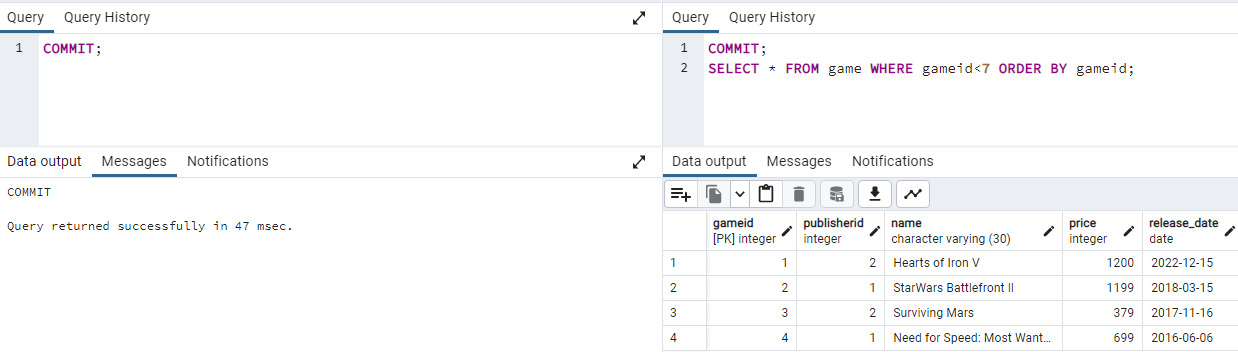
На скріншотах видно, що serializable, запобігає усім аномаліям.

**REPEATABLE READ**

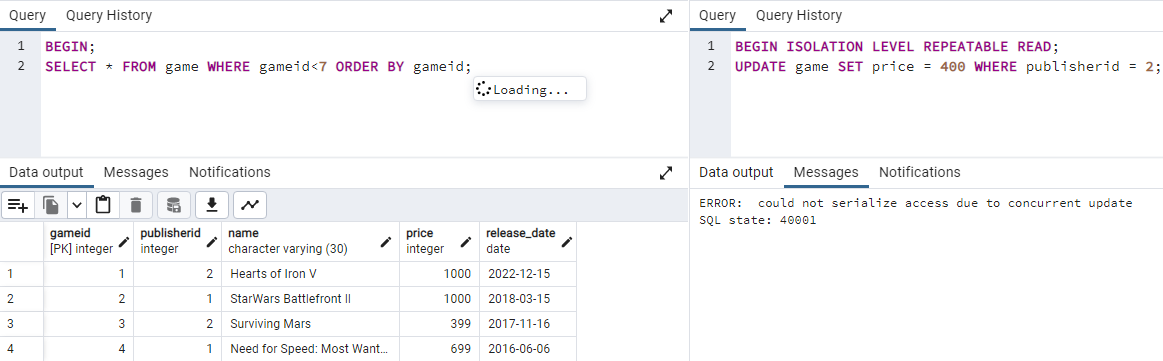
Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані).







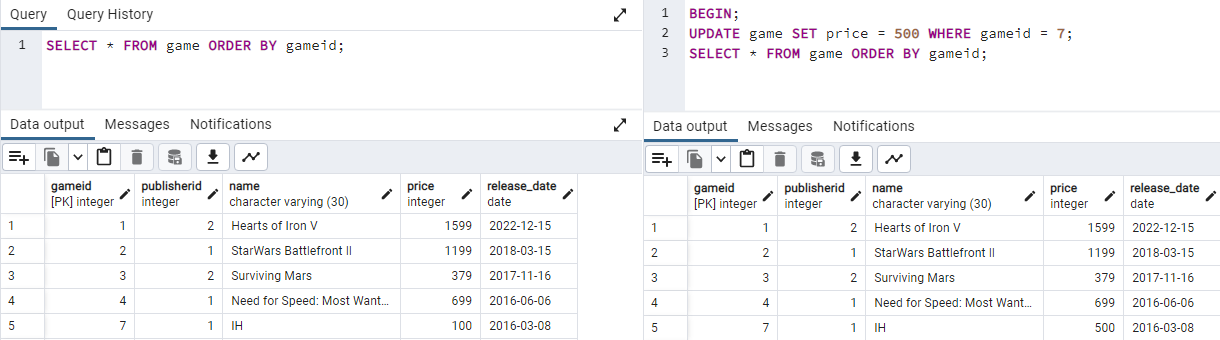
На скріншотах видно, що друге вікно не побачить змін, зроблених і закомічених в першому, поки саме не закінчить транзакцію.

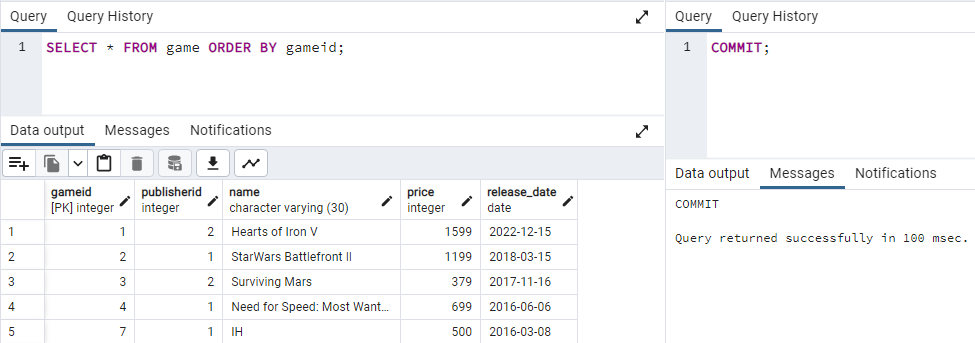


На скріншотах видно, що repeatable read не дасть другому вікну внести будь-які зміни в рядки, що були прочитані(модифіковані) в першому

**READ COMMITED**

Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє брудне читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання.





На скріншотах видно, що в першому вікні можна побачити зміни, зроблені в другому лише після їх коміту. Це означає, що брудне читання відсутнє

**READ UNCOMITTED**

Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.