

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

Рябенко Б. Ю.

Перевірив:

Павловський В. І.

Київ – 2022

*Метою роботи* є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

*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:

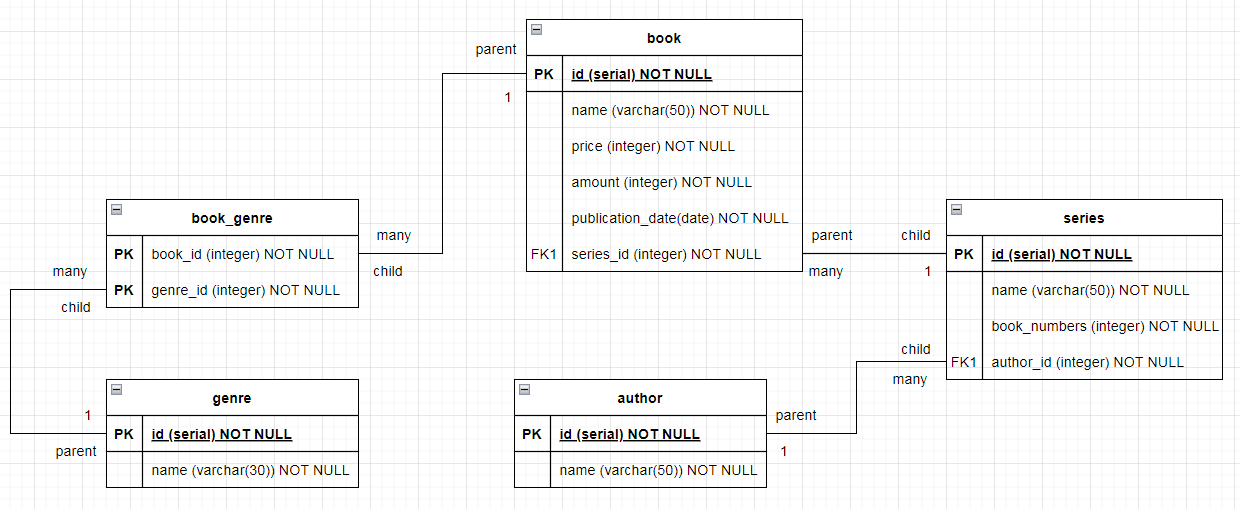


Рисунок 1 – Логічна модель бази даних

**Середовище розробки та налаштування підключення до бази даних**

Для розробки використовувалась мова програмування Python, середовище розробки PyCharm Community Edition, стороння бібліотека, що надає API для доступу до PostgreSQL – psycopg2, а також бібліотека для реалізації ORM – SQLAlchemy.

**Посилання на гіт із текстом програми:**

<https://github.com/RyabenkoBoris/DataBases/tree/lab3>

**Структура програми та її опис**

На рисунку 2 відображено деревовидну структуру програми:

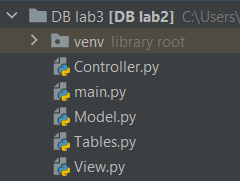


Рисунок 2 – структура програми

Програма поділена на 5 файлів: main.py, Controller.py, Model.py, View.py, Tables.py.

У файлі View.py описані функції, що виводять дані у консоль.

У файлі Model.py описаний клас, який реалізує методи управління базою даних

У файлі Controller.py описаний клас, який використовує функції та методи файлів View.py та Model.py.

У файлі Tables.py описані класи усіх таблиць із відповідними зв’язками.

**Лістинги програм перетворених для ORM**

Лістинг фрагменту програми для внесення даних

**def** **insert\_data**(self, table\_name, values):

**if** table\_name == 'author':

data = Author(name=values[**0**])

**elif** table\_name == 'series':

data = Series(name=values[**0**], book\_numbers=values[**1**], author\_id=values[**2**])

**elif** table\_name == 'book':

data = Book(name=values[**0**], price=values[**1**], amount=values[**2**],

publication\_date=values[**3**], series\_id=values[**4**])

**elif** table\_name == 'genre':

data = Genre(name=values[**0**])

**elif** table\_name == 'book\_genre':

data = BookGenre(book\_id=values[**0**], genre\_id=values[**1**])

self.s.add(data)

self.s.commit()

self.s.close()

Лістинг фрагменту програми для видалення даних

**def** **delete\_data**(self, table\_name, cond):

**if** table\_name == 'author':

data = self.s.query(Author).get(cond)

**elif** table\_name == 'series':

data = self.s.query(Series).get(cond)

**elif** table\_name == 'book':

data = self.s.query(Book).get(cond)

**elif** table\_name == 'genre':

data = self.s.query(Genre).get(cond)

**elif** table\_name == 'book\_genre':

data = self.s.query(BookGenre).get(cond)

self.s.delete(data)

self.s.commit()

self.s.close()

Лістинг фрагменту програми для зміни даних

**def** **change\_data**(self, table\_name, values, cond):

**if** table\_name == 'author':

data = self.s.query(Author).get(cond)

data.name = values[**0**]

**elif** table\_name == 'series':

data = self.s.query(Series).get(cond)

data.name = values[**0**]

**elif** table\_name == 'book':

data = self.s.query(Book).get(cond)

data.name = values[**0**]

data.price = values[**1**]

data.amount = values[**2**]

data.publication\_date = values[**3**]

data.series\_id = values[**4**]

**elif** table\_name == 'genre':

data = self.s.query(Genre).get(cond)

data.name = values[**0**]

self.s.commit()

self.s.close()

**Текст файлу Table.py (класу ORM)**

**from** **sqlalchemy** **import** Column, Integer, String, Date, ForeignKey

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **sqlalchemy.orm** **import** relationship, backref

Base = declarative\_base()

**class** **Author**(Base):

\_\_tablename\_\_ = 'author'

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

name = Column(String)

**class** **Series**(Base):

\_\_tablename\_\_ = 'series'

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

name = Column(String)

book\_numbers = Column(Integer)

author\_id = Column(Integer, ForeignKey('author.id'))

author = relationship("Author", backref=backref('author'))

**class** **Book**(Base):

\_\_tablename\_\_ = 'book'

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

name = Column(String)

price = Column(Integer)

amount = Column(Integer)

publication\_date = Column(Date)

series\_id = Column(Integer, ForeignKey('series.id'))

series = relationship("Series", backref=backref('series'))

**class** **Genre**(Base):

\_\_tablename\_\_ = 'genre'

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

name = Column(String)

book = relationship("Book", secondary='book\_genre')

**class** **BookGenre**(Base):

\_\_tablename\_\_ = 'book\_genre'

book\_id = Column(Integer, ForeignKey('book.id'), primary\_key=True)

genre\_id = Column(Integer, ForeignKey('genre.id'), primary\_key=True)

**Приклад запитів ORM:**

s = Session()

#Вставка елементу

author = Author(name='author1')

s.add(author)

#Видалення знайденого елементу

genre = s.query(Genre).get(**10**)#пошук по ip

s.delete(genre)

#Зміна елементу

author = s.query(Author).get(**2**)

author.name = 'author2'

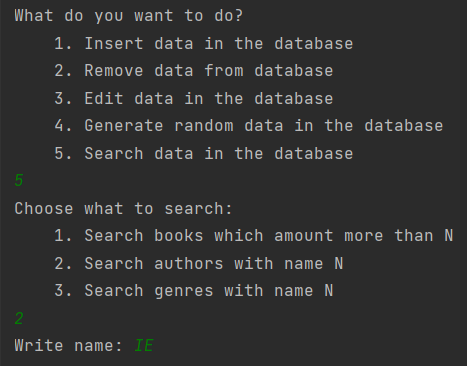
s.commit()

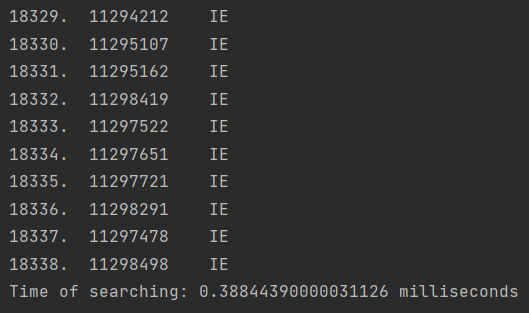
s.close()

**Hash**

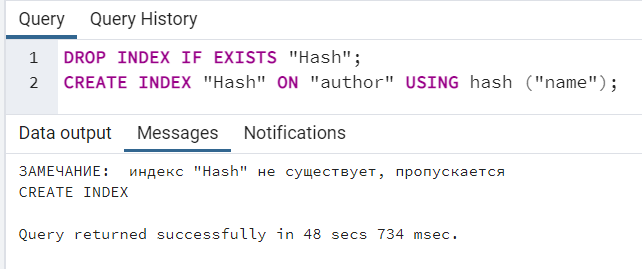
Для дослідження індексу була використана таблиця authors яка має більше ніж 10000000 даних.

Пошук без індексації

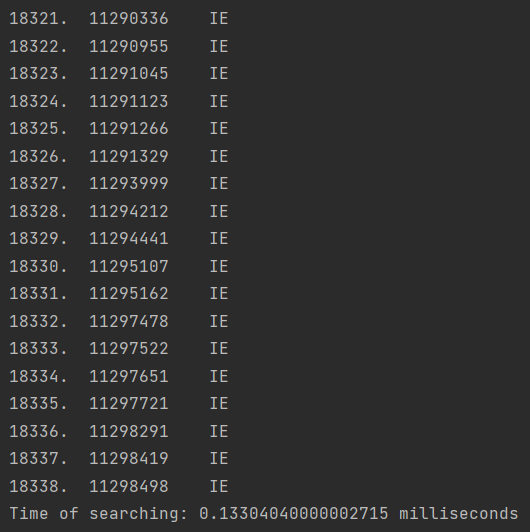




Проіндексуємо таблицю



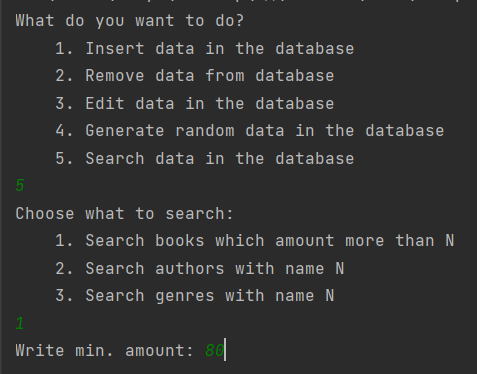
Пошук із індексацією

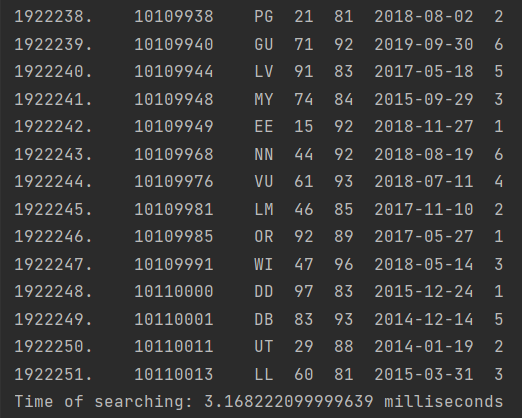


**BRIN**

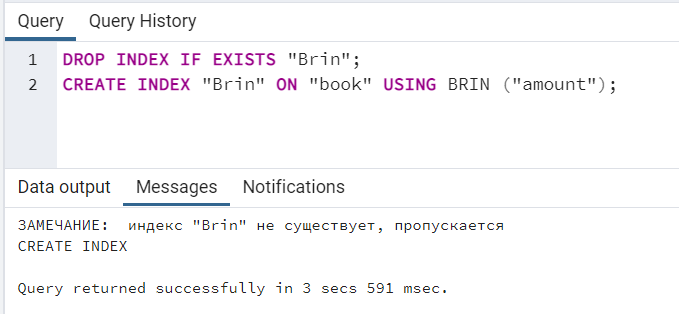
Для дослідження індексу була використана таблиця book яка має більше ніж 10000000 даних.

Пошук без індексації

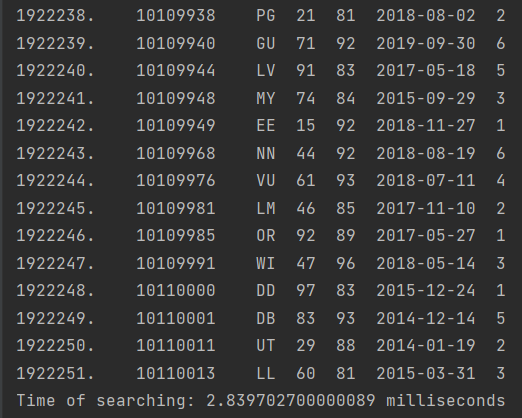




Проіндексуємо таблицю



Пошук із індексацією

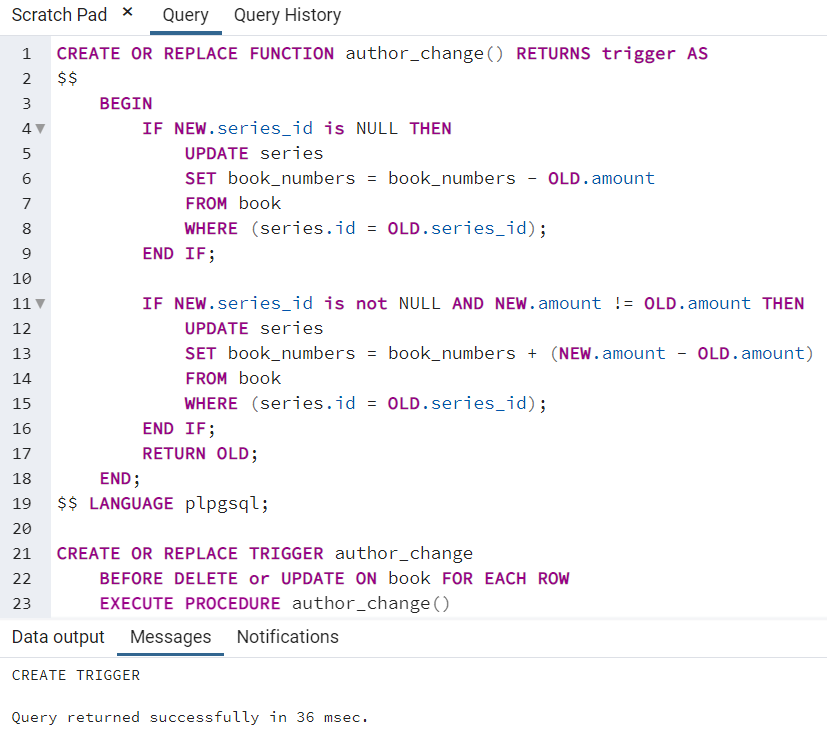


Як можна побачити, в першому випадку індекс пришвидшив роботу аж у три рази, це пов’язано із тим, що Hash індексація дуже добре працює із порівнянням даних. У другому випадку індекс пришвидшив роботу алгоритму, але не набагато, це пов’язано із тим, що ідея самої індексації полягаю не у тому, щоб знайти потрібні строки, але для того, щоб уникнути пошук свідомо непотрібних.

Індекси відмінно себе показують, доки ми виконуєте запити на вибірку даних оператором SELECT, але, як тільки у базі даних починається частий виклик операторів INSERT, UPDATE та DELETE, тоді індекси можуть зробити навіть гірше. Якщо ці оператори призводять до перестановки або розділення сторінки, тоді індексу може знадобитися реорганізація, що стосується всіх пов'язаних індексів та операцій, в результаті буде повсюдне падіння продуктивності.

**Тригер бази даних PostgreSQL. Умова для тригера – before delete, update**

Тригер



Принцип роботи тригеру

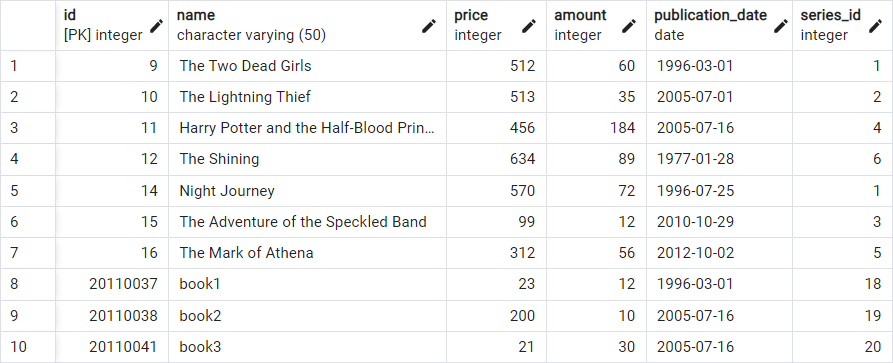
Тригер спрацьовує до видалення або зміни в таблиці *“book”*.

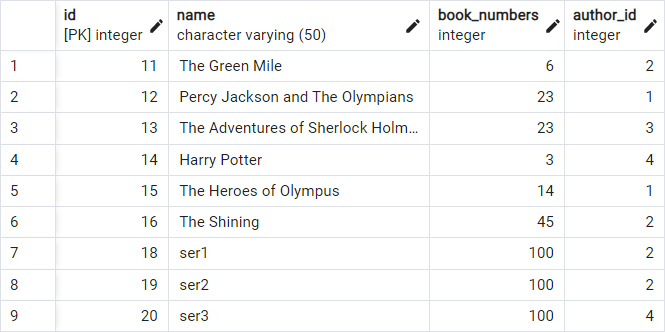
При видалені рядку у якого значення *amount = 30*, а *series\_id = 20*, у таблиці *“series”* рядок із *id = 20* зменшить значення *book\_numbers* на *30*, тобто на те значення яке було у amount.

При зміні рядка у таблиці *“book”*, тригер перевіряє, чи було змінено значення amount, якщо так, тоді рядок *book\_numbers* у таблиці *“series”* змінюється відповідно (якщо *amount* було зменшено на *20*, тоді і *book\_numbers* зменшиться на *20*, якщо *amount* було збільшено на *10*, тоді і *book\_numbers* збільшиться на *10*)

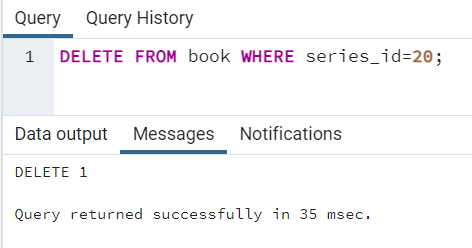
**Приклади**

Таблиці до змін

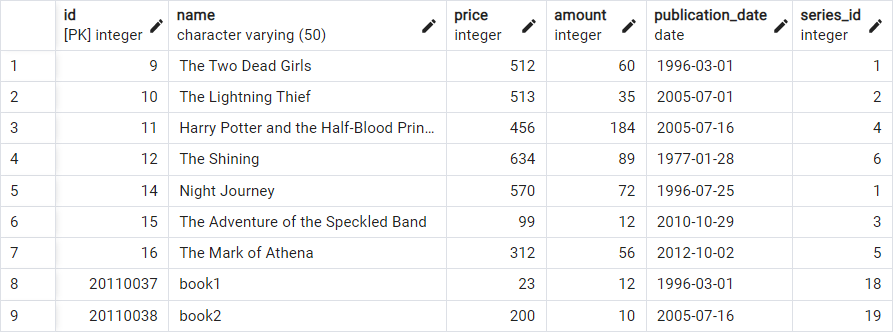




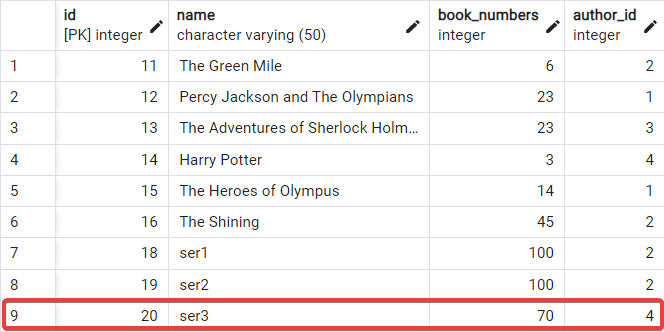
Видалення у таблиці “book”



Таблиця “book” після видалення

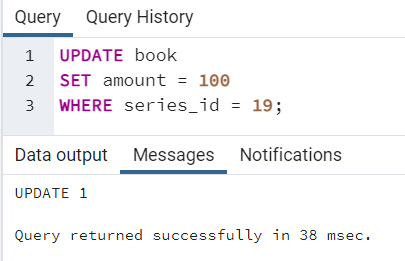


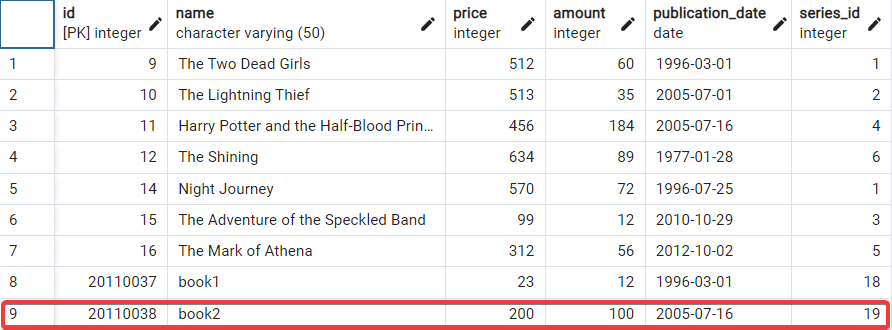
Таблиця “series” після зміни таблиці “book”

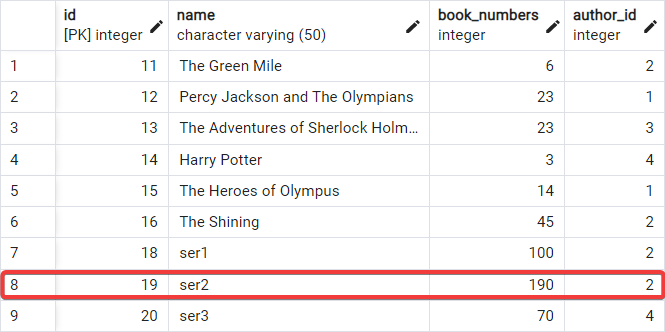


Значення *amount* у видаленого рядка було 30, тому і значення *book\_number­* зменшилось на 30.

Зміна таблиці “book” (збільшення значення amount)

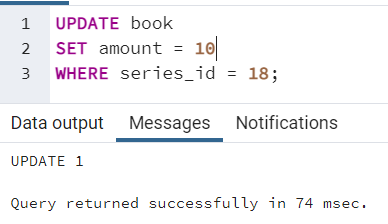


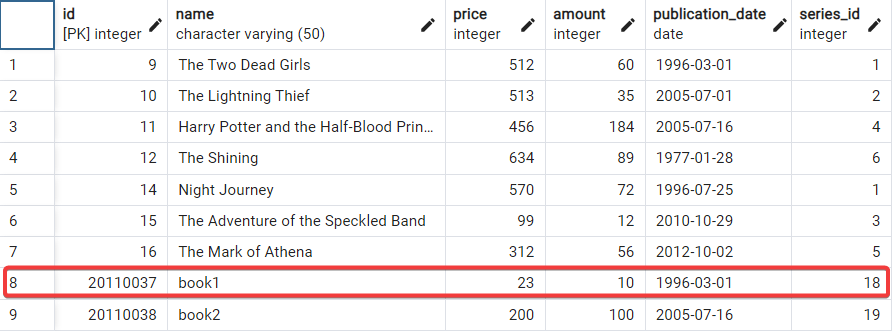


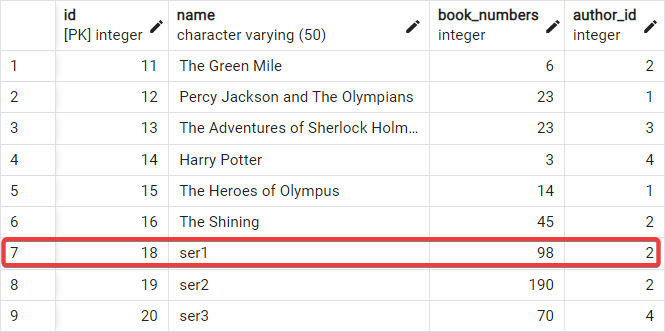


Значення *amount* збільшилося на 90, тому і значення *book\_numbers* збільшилося на 90.

Зміна таблиці “book” (зменшення значення amount)







Значення *amount* зменшилось на 2, тому і значення *book\_numbers* зменшилось на 2.

**Рівні ізоляції транзакцій у PostgreSQL**

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім не виконуються. 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 виникнення 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.

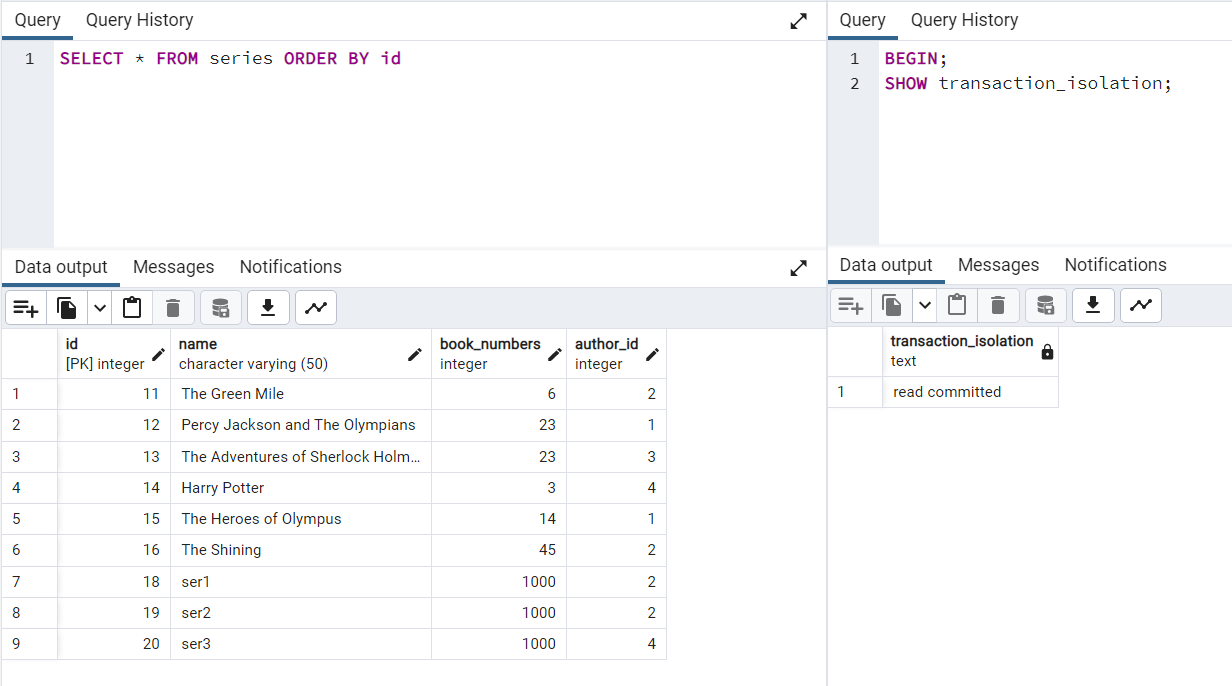
4. Фантомне читання - ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

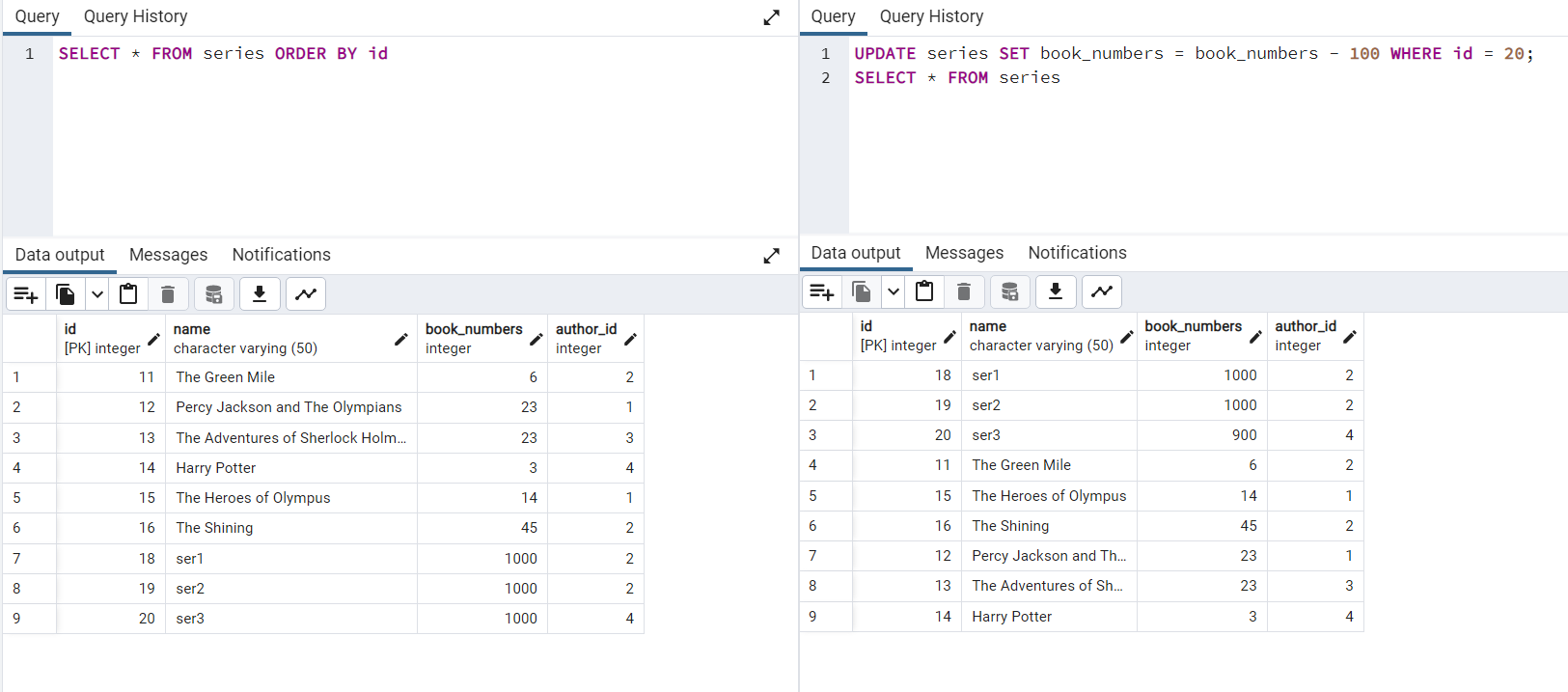
**READ COMMITTED**

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.

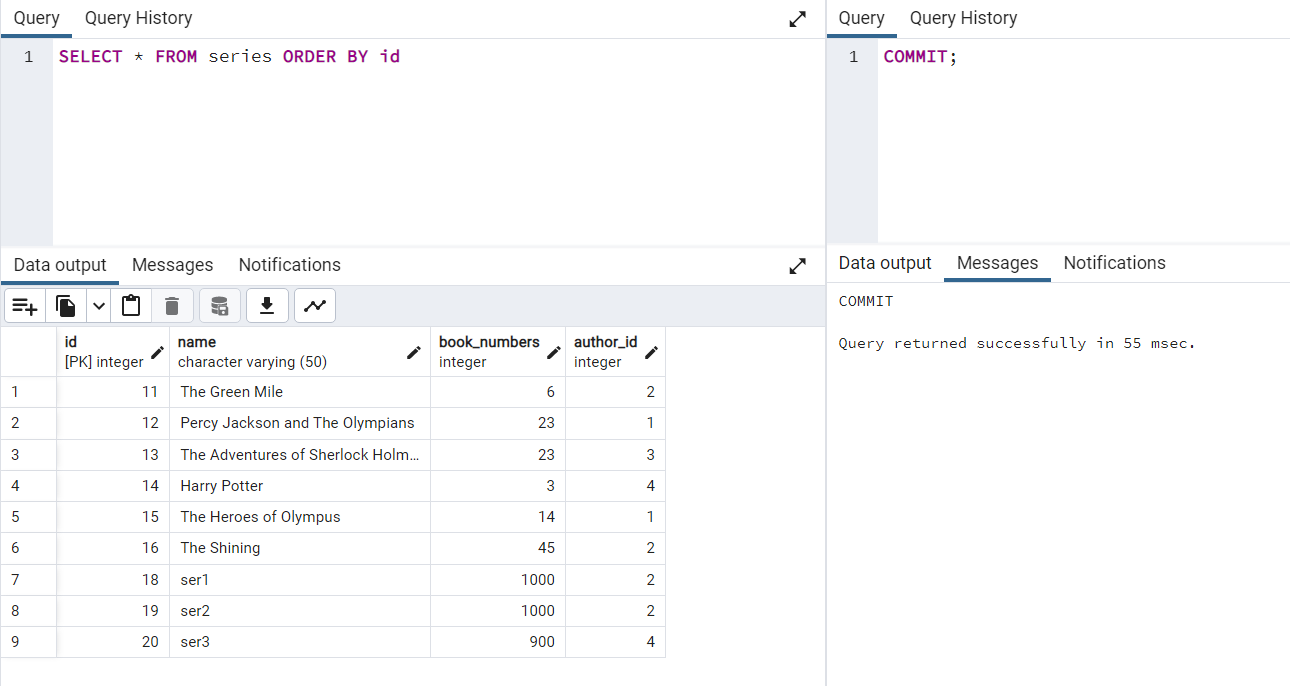
**Приклад відсутності брудного читання**

Відкриємо два вікна, перше вікно буде переглядати дані, а друге їх змінювати.

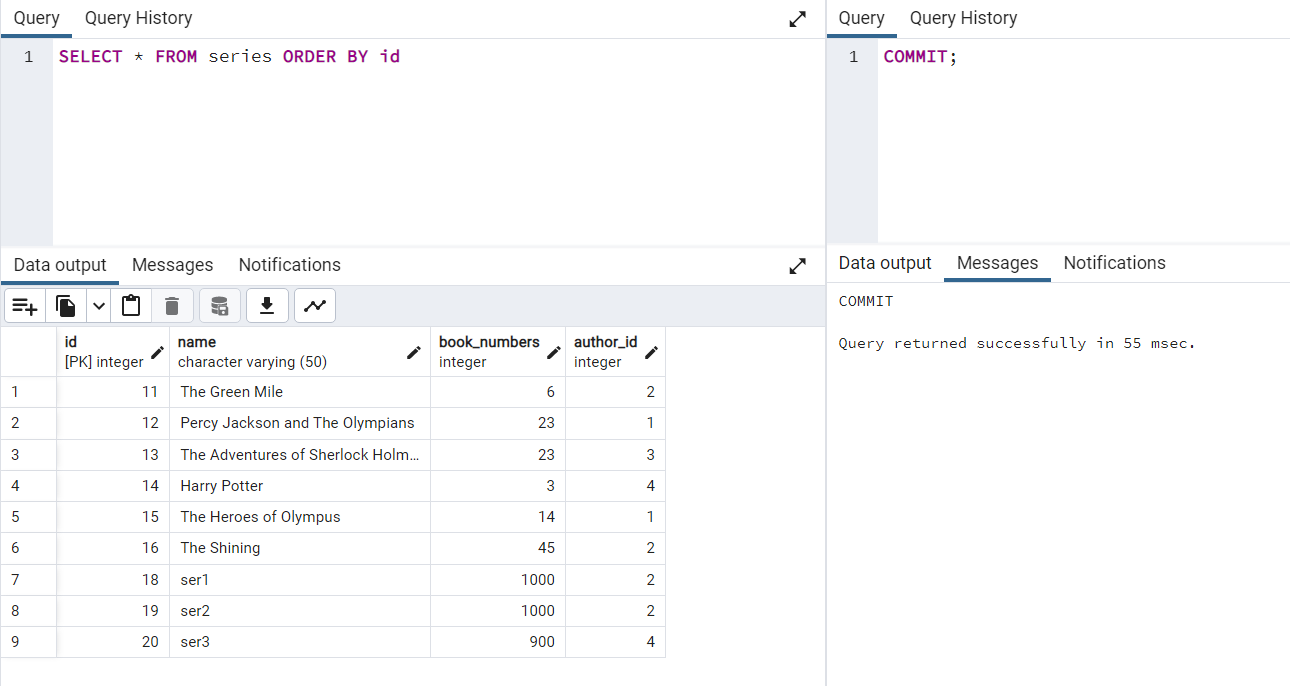




Як можна побачити, перше вікно не бачить зміни які виконані у другому, поки ми не виконаємо commit. Тобто перше вікно не бачить незафіксовані дані у другому вікні - брудне читання не допускається.



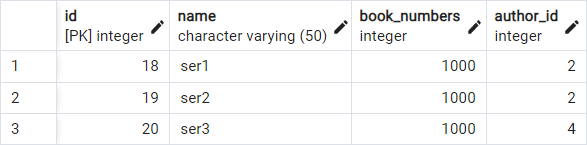
Після commit, перше вікно починає бачити зміни, які були виконанні у другому.

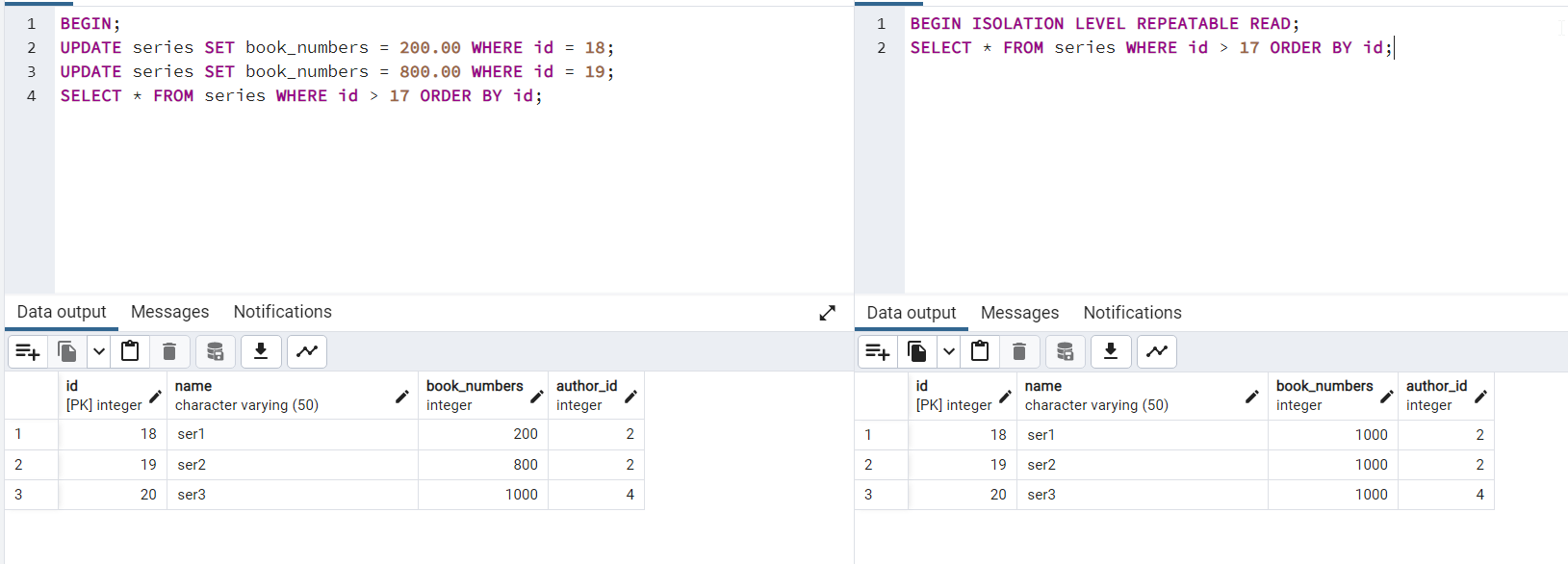


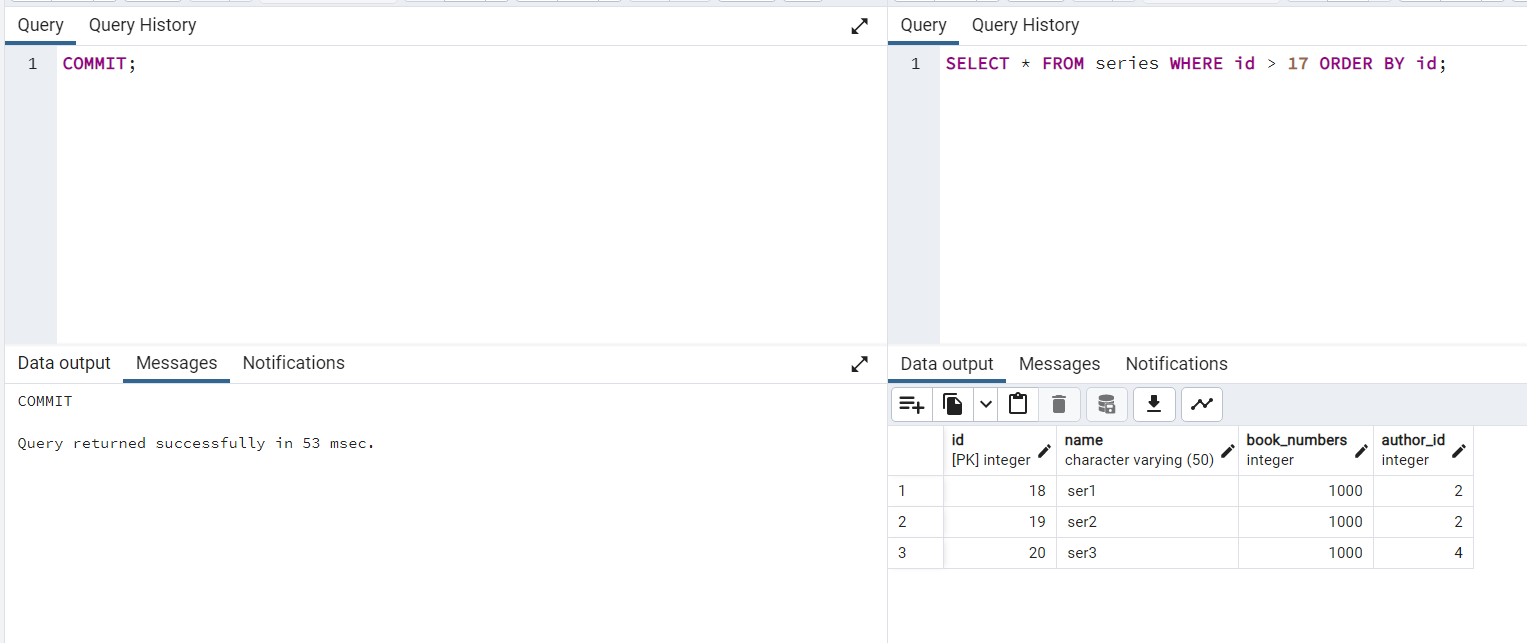
**REPEATABLE READ**

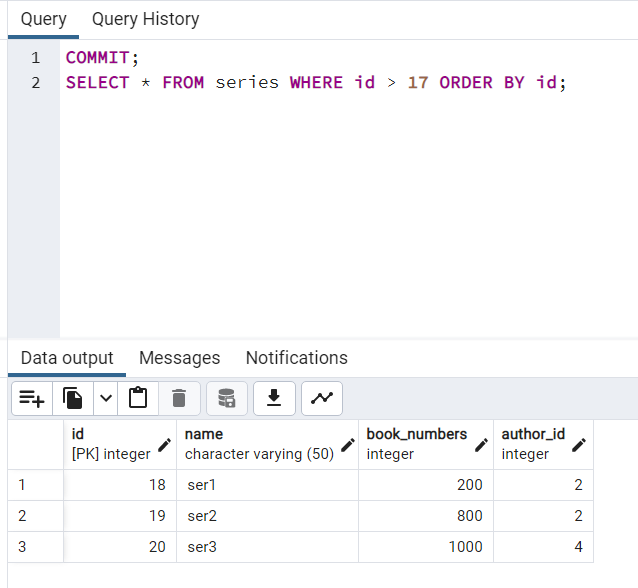
Рівень, при якому читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. (Поки транзакція не закінчена, ніякі інші транзакції не можуть змінити ці дані).

**Приклад відсутності неповторного та фантомного читань**





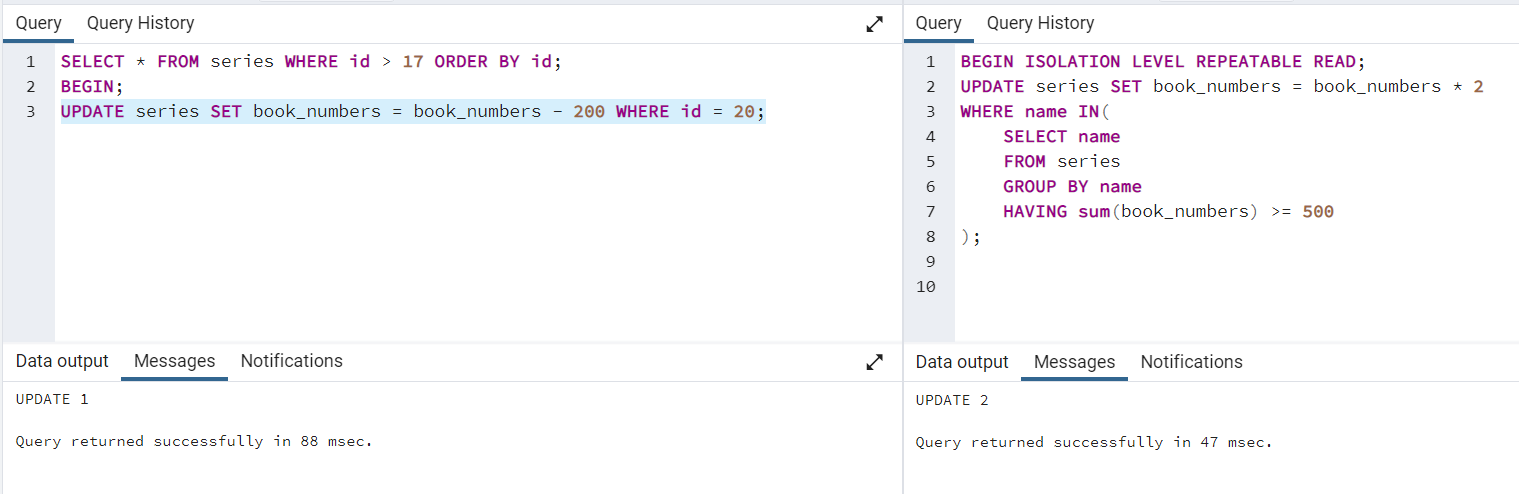




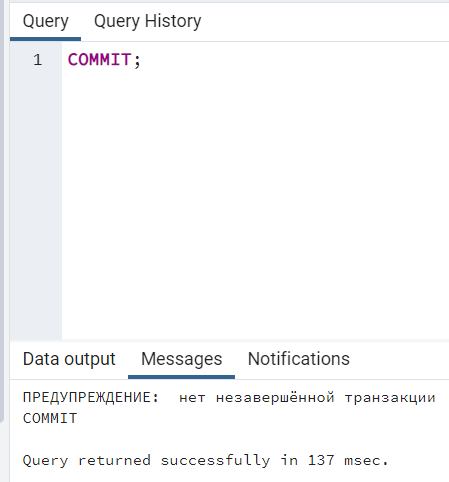
Як можна побачити із другого вікна, рівень repeatable read не бачить зміни інших вікон, поки не буде виконана команда commit.

**Приклад відсутності неповторюваного читання**

Нехай перше та друге вікно зроблять зміни у однієї таблиці

****

Зробимо commit у першому вікні.

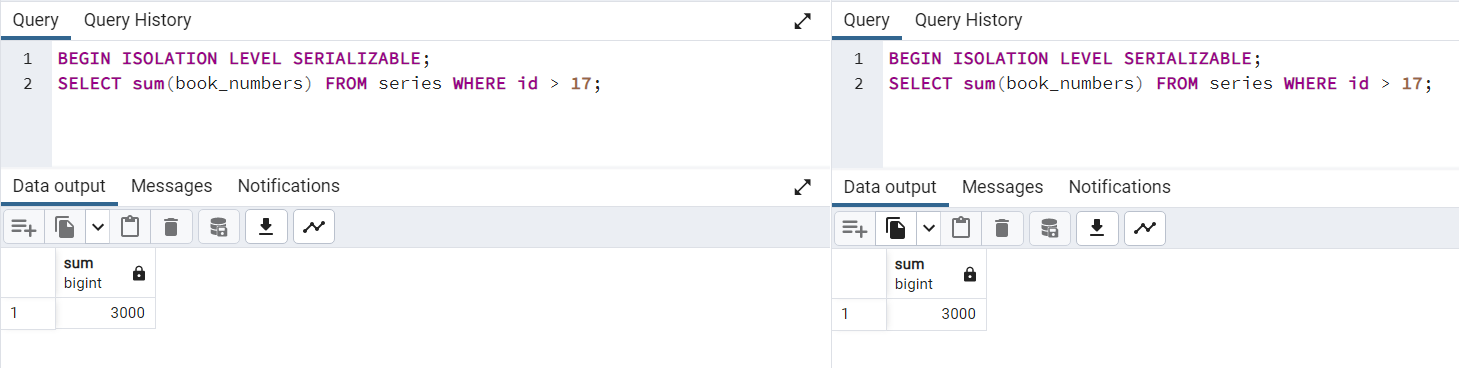
****

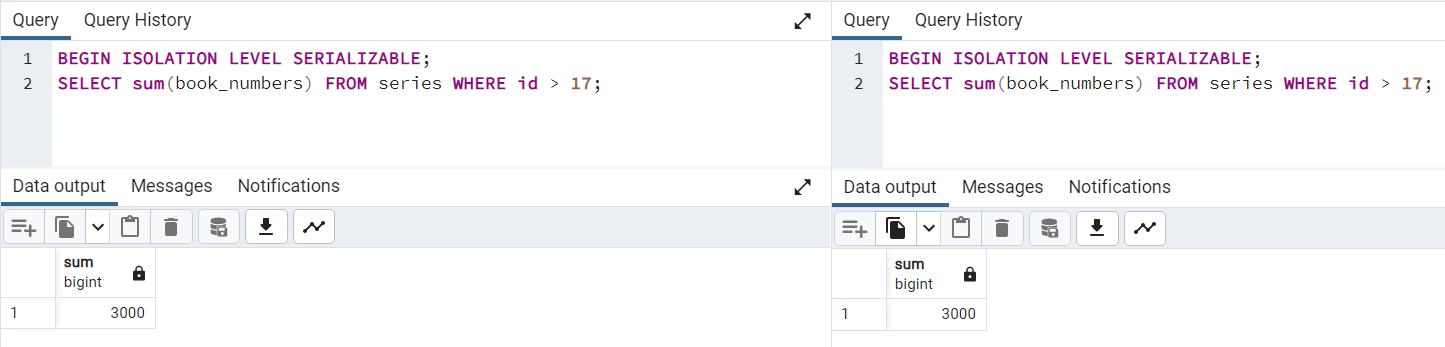
Як можна побачити, repeatable read не дозволяє виконувати операції зміни даних, якщо дані вже було модифіковано у іншій незавершеній транзакції.

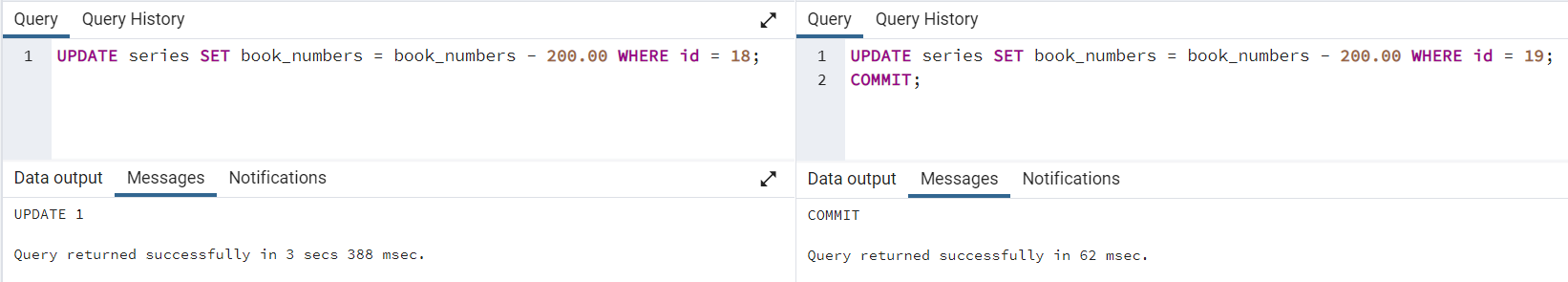
**SERIALIZABLE**

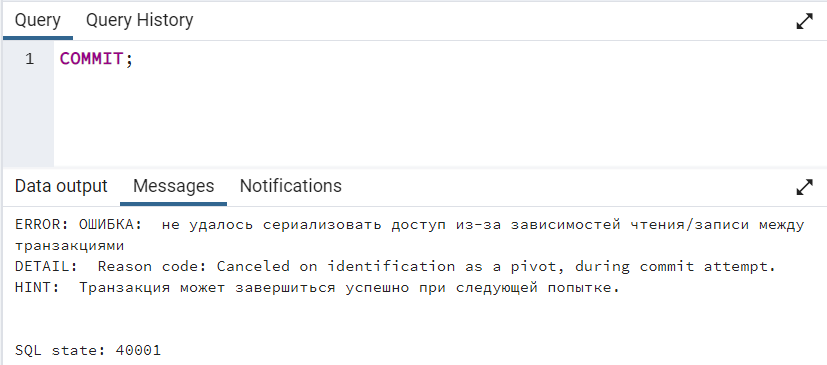
Найбільш високий рівень ізольованості. На цьому рівні результати паралельного виконання транзакцій для бази даних у більшості випадків можна вважати такими, що збігаються з послідовним виконанням тих же транзакцій (по черзі в будь-якому порядку).

Приклад

****

****

****

****

Як бачимо serializable повністю запобігає усім можливим аномаліям.