

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 та засоби управління»**

*Тема:* « **Засоби оптимізації роботи СУБД PostgreSQL***»*

Виконав: студент ІІІ курсу

ФПМ групи КВ-93

Зеленяк І.О.

Перевірив:

Київ, 2021

Посилання на репозиторій <https://github.com/AmiriVANYCH/DataBase>

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.

|  |  |
| --- | --- |
| *Види індексів* | *Умови для тригера* |
| *BTree, BRIN* | *after update, insert* |

В даній лабораторній буде розроблятися програма керування парковкою з розділеними персональними місцями для кожної автівки, розроблена в першій лабораторній роботі. З таким приципом роботі:

Для того, щоб особа користувалася парковкою необхідно підписати договір на окреме паркувальне місце.

Вважаємо що в одного користувача може бути доступ до декількох автівок (декілька договорів).

Одному паркомісцю відповідає тільки одна машина яка може там стояти (за договором), якщо воно орендоване, інакше жодна машина не може там стояти.

### Перетворення модуля “Модель” з шаблону MVC ЛР №2 у вигляд об’єктно-реляційної проекції (ORM)

Перетворення модуля “Модель” шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM) будемо виконувати з використанням програмного забезпечення SQLAlchemy. Це програмне забезпечення з відкритим вихідним кодом для роботи з базами даних за допомогою мови SQL. Воно реалізує технологію програмування ORM (Object-Relational Mapping), яка пов'язує бази даних із концепціями об'єктно-орієнтованих мов програмування. SQLAlchemy дозволяє описувати структури баз даних та способи взаємодії з ними прямо на мові Python.

**З'єднання з базою даних**

Перш за все необхідно встановити з'єднання з базою даних та відкриття сесії за потребою:

**def** **get\_connection**(self):

**if** **not** self.\_engine:

self.\_engine =

create\_engine('postgresql://ivan:12345678@192.168.1.224:5432/Parking' )

**return** self.\_engine

**def** **get\_cursor**(self):

**if** self.\_session:

**return** self.\_session

**else**:

engine = self.get\_connection()

Session = sessionmaker(bind=engine)

self.\_session = Session()

**return** self.\_session

За завданням до лабораторної роботи, назви функцій, та їх зміст мають залишатися незмінними, тому використовуємо існуючи.

### Опис бази даних

Концепція SQLAlchemy полягає у тому, що звернення до бази даних полягає у тому, що таблиці бази даних репрезентують об'єкти, із зв'язками між ними. Тому необхідно описати ці об'єкти та зв'язки:

**class** **CarType**(Base):

\_\_tablename\_\_ = 'CarType'

CarTypeID = Column('CarTypeID', SmallInteger, primary\_key=True)

CarTypeName = Column('CarTypeName', String(**255**), nullable=False)

cars = relationship("Cars", back\_populates="cartype")

# Ця частина коду використовується, коли ми повертаємо

# інформацію у контролер. Контролер очірує дані у форматі

# python Dictionary

**def** **to\_dict**(self):

dict = {}

dict["CarTypeID"] = self.CarTypeID

dict["CarTypeName"] = self.CarTypeName

**return** dict

# Клас, що описує таблицю Makes

**class** **Makes**(Base):

\_\_tablename\_\_ = 'Makes'

MakeID = Column('MakeID', SmallInteger, primary\_key=True)

MakeName = Column('MakeName', String(**50**), nullable=False)

MakeDescription = Column('MakeDescription', String)

cars = relationship("Cars", back\_populates="makes")

**def** **to\_dict**(self):

dict = {}

dict["MakeID"] = self.MakeID

dict["MakeName"] = self.MakeName

dict["MakeDescription"] = self.MakeDescription

**return** dict

# Клас, що описує таблицю Cars

**class** **Cars**(Base):

\_\_tablename\_\_ = 'Cars'

CarRegNum = Column('CarRegNum', String(**8**), primary\_key=True)

CarMakeID = Column(ForeignKey('Makes.MakeID'), primary\_key=True)

CarType = Column('CarType', SmallInteger, ForeignKey('CarType.CarTypeID'))

cartype = relationship("CarType", back\_populates="cars")

makes = relationship("Makes", back\_populates="cars")

person = relationship(

"Contract", cascade="all, delete, delete-orphan", back\_populates="cars")

**def** **to\_dict**(self):

dict = {}

dict["CarRegNum"] = self.CarRegNum

dict["CarMakeID"] = self.CarMakeID

dict["CarType"] = self.CarType

**return** dict

# Клас, що описує таблицю ParkingPlace

**class** **ParkingPlace**(Base):

\_\_tablename\_\_ = 'ParkingPlace'

ParkingPlaceID = Column('ParkingPlaceID', SmallInteger, primary\_key=True)

ParkingPlaceDesc = Column('ParkingPlaceDesc', String)

contract = relationship("Contract", back\_populates="parking")

**def** **to\_dict**(self):

dict = {}

dict["ParkingPlaceID"] = self.ParkingPlaceID

dict["ParkingPlaceDesc"] = self.ParkingPlaceDesc

**return** dict

# Клас, що описує таблицю Persons

**class** **Persons**(Base):

\_\_tablename\_\_ = 'Persons'

PersonID = Column('PersonID', BigInteger, primary\_key=True)

PersonLastName = Column('PersonLastName', String(**50**), nullable=False)

PersonName = Column('PersonName', String(**50**), nullable=False)

PersonMidleName = Column('PersonMidleName', String(**50**))

cars = relationship(

"Contract", cascade="all, delete, delete-orphan", back\_populates="person")

phones = relationship(

"Phones", cascade="all, delete, delete-orphan", back\_populates="person")

**def** **to\_dict**(self):

dict = {}

dict["PersonID"] = self.PersonID

dict["PersonLastName"] = self.PersonLastName

dict["PersonName"] = self.PersonName

dict["PersonMidleName"] = self.PersonMidleName

**return** dict

# Клас, що описує таблицю Phones

**class** **Phones**(Base):

\_\_tablename\_\_ = 'Phones'

Phone = Column('Phone', BigInteger, primary\_key=True)

PersonID = Column(ForeignKey('Persons.PersonID'), primary\_key=True)

person = relationship("Persons", back\_populates="phones")

**def** **to\_dict**(self):

dict = {}

dict["Phone"] = self.Phone

dict["PersonID"] = self.PersonID

**return** dict

# Клас, що описує таблицю Contract, у той самий час є асоціативним

# класом, що утворює звязок N:M багато до багатьох між сутностями

# Persons та Cars

**class** **Contract**(Base):

\_\_tablename\_\_ = 'Contract'

ContractID = Column('ContractID', BigInteger, primary\_key=True)

CarNumber = Column(ForeignKey('Cars.CarRegNum'), primary\_key=True)

PersonID = Column(ForeignKey('Persons.PersonID'), primary\_key=True)

ParkingID = Column(ForeignKey(

'ParkingPlace.ParkingPlaceID'), primary\_key=True)

ContractStart = Column('ContractStart', Date, nullable=False)

ContractEnd = Column('ContractEnd', Date, nullable=False)

cars = relationship("Cars", back\_populates="person")

person = relationship("Persons", back\_populates="cars")

parking = relationship("ParkingPlace", back\_populates="contract")

**def** **to\_dict**(self):

dict = {}

dict["ContractID"] = self.ContractID

dict["CarNumber"] = self.CarNumber

dict["PersonID"] = self.PersonID

dict["ParkingID"] = self.ParkingID

dict["ContractStart"] = self.ContractStart

dict["ContractEnd"] = self.ContractEnd

**return** dict

### Зв'язок 1:N

Як видно з наведеної структури бази даних, зв'язок 1:N реалізований наступним чином:

**class** **Makes**(Base):

\_\_tablename\_\_ = 'Makes'

MakeID = Column('MakeID', SmallInteger, primary\_key=True)

MakeName = Column('MakeName', String(**50**), nullable=False)

MakeDescription = Column('MakeDescription', String)

**cars = relationship("Cars", back\_populates="makes")**

**def** **to\_dict**(self):

dict = {}

dict["MakeID"] = self.MakeID

dict["MakeName"] = self.MakeName

dict["MakeDescription"] = self.MakeDescription

**return** dict

# Клас, що описує таблицю Cars

**class** **Cars**(Base):

\_\_tablename\_\_ = 'Cars'

CarRegNum = Column('CarRegNum', String(**8**), primary\_key=True)

CarMakeID = Column(ForeignKey('Makes.MakeID'), primary\_key=True)

CarType = Column('CarType', SmallInteger, ForeignKey('CarType.CarTypeID'))

cartype = relationship("CarType", back\_populates="cars")

**makes = relationship("Makes", back\_populates="cars")**

person = relationship(

"Contract", cascade="all, delete, delete-orphan", back\_populates="cars")

**def** **to\_dict**(self):

dict = {}

dict["CarRegNum"] = self.CarRegNum

dict["CarMakeID"] = self.CarMakeID

dict["CarType"] = self.CarType

**return** dict

На прикладі таблиць  **Makes** та **Cars** реалізований зв'язок 1:N, а саме один виробник **Makes** зав'язаний

**cars = relationship("Cars", back\_populates="makes")**

з об'єктом **Cars** та цей зв'язок двосторонній, адже **Cars** пов'язаний зворотно з об'єктом  **Makes**:

**makes = relationship("Makes", back\_populates="cars")**

Таким чином, один об'єкт **Makes** зв'язаний із багатьма об'єктами **Cars**, та багато об'єктів **Cars**, посилаються на один об'єкт **Makes**.

### Реалізація зв'язку M:N

SQLAlchemy використовує два типи реалізації зв'язків багато до багатьох: з використанням асоціативних таблиць та з використанням асоціативного об'єкта.

У першому випадку зв'язки утворюються між таблицями безпосередньо, та використання додаткових стовпців, що не беруть участь у зв'язках між об'єктами неможливе. Тому я обрав другий метод, метод асоціативних об'єктів, який дозволяє використовувати додаткові стовпці:

**class** **Cars**(Base):

\_\_tablename\_\_ = 'Cars'

CarRegNum = Column('CarRegNum', String(**8**), primary\_key=True)

CarMakeID = Column(ForeignKey('Makes.MakeID'), primary\_key=True)

CarType = Column('CarType', SmallInteger, ForeignKey('CarType.CarTypeID'))

cartype = relationship("CarType", back\_populates="cars")

makes = relationship("Makes", back\_populates="cars")

**person = relationship(**

**"Contract", cascade="all, delete, delete-orphan", back\_populates="cars")**

**def** **to\_dict**(self):

dict = {}

dict["CarRegNum"] = self.CarRegNum

dict["CarMakeID"] = self.CarMakeID

dict["CarType"] = self.CarType

**return** dict

**class** **Persons**(Base):

\_\_tablename\_\_ = 'Persons'

PersonID = Column('PersonID', BigInteger, primary\_key=True)

PersonLastName = Column('PersonLastName', String(**50**), nullable=False)

PersonName = Column('PersonName', String(**50**), nullable=False)

PersonMidleName = Column('PersonMidleName', String(**50**))

**cars = relationship(**

**"Contract", cascade="all, delete, delete-orphan", back\_populates ="person")**

phones = relationship(

"Phones", cascade="all, delete, delete-orphan", back\_populates="person")

**def** **to\_dict**(self):

dict = {}

dict["PersonID"] = self.PersonID

dict["PersonLastName"] = self.PersonLastName

dict["PersonName"] = self.PersonName

dict["PersonMidleName"] = self.PersonMidleName

**return** dict

**class** **Contract**(Base):

\_\_tablename\_\_ = 'Contract'

ContractID = Column('ContractID', BigInteger, primary\_key=True)

CarNumber = Column(ForeignKey('Cars.CarRegNum'), primary\_key=True)

PersonID = Column(ForeignKey('Persons.PersonID'), primary\_key=True)

ParkingID = Column(ForeignKey(

'ParkingPlace.ParkingPlaceID'), primary\_key=True)

ContractStart = Column('ContractStart', Date, nullable=False)

ContractEnd = Column('ContractEnd', Date, nullable=False)

cars = relationship("Cars", back\_populates="person")

**person = relationship("Persons", back\_populates="cars")**

parking = relationship("ParkingPlace", back\_populates="contract")

**def** **to\_dict**(self):

dict = {}

dict["ContractID"] = self.ContractID

dict["CarNumber"] = self.CarNumber

dict["PersonID"] = self.PersonID

dict["ParkingID"] = self.ParkingID

dict["ContractStart"] = self.ContractStart

dict["ContractEnd"] = self.ContractEnd

**return** dict

Строки виділені жирним шрифтом реалізують зв'язок M:N між об'єктами **Persons** та **Cars**

### Функції, що реалізують додавання, зміну, вилучення, та читання

Функції додавання, зміни, вилучення, та читання зводяться до роботи з відповідними об'єктами мови python:

**def** **\_get\_CarType\_by\_name**(self, CarTypeName):

session = self.get\_cursor()

record = session.query(CarType).filter\_by(

CarTypeName=CarTypeName).first()

**return** record

**def** **get\_CarType\_by\_name**(self, CarTypeName):

**return** self.\_get\_CarType\_by\_name(CarTypeName).to\_dict()

**def** **\_get\_CarType\_by\_id**(self, CarTypeID):

session = self.get\_cursor()

record = session.query(CarType).filter\_by(

CarTypeID=CarTypeID).first()

**return** record

**def** **get\_CarType\_by\_id**(self, CarTypeID):

**return** self.\_get\_CarType\_by\_id(CarTypeID).to\_dict()

**def** **create\_CarType**(self, CarTypeName):

session = self.get\_cursor()

record = self.\_get\_CarType\_by\_name(CarTypeName)

**if** **not** hasattr(record, 'CarTypeID'):

cartype = CarType(

CarTypeName=CarTypeName)

session.add(cartype)

session.commit()

**def** **read\_CarType**(self,):

session = self.get\_cursor()

record = session.query(CarType).all()

**return** self.list2dict(record)

**def** **update\_CarType**(self, CarTypeID, CarTypeName):

session = self.get\_cursor()

record = self.\_get\_CarType\_by\_id(CarTypeID)

**if** hasattr(record, 'CarTypeID'):

record.CarTypeName = CarTypeName

session.commit()

**def** **del\_CarType**(self, CarTypeID):

session = self.get\_cursor()

CarType.query.filter(CarType.CarTypeID == CarTypeID).delete()

session.commit()

### Вилучення записів з звязками N:M

Згідно із концепцією SQLAlchemy таблиця, що забезпечує зв'язок багато до багатьох, автоматично підпорядковується операторам INSERT і DELETE, оскільки об’єкти додаються або видаляються з колекції. Немає необхідності видаляти з цієї таблиці вручну. Акт видалення запису з колекції матиме ефект видалення рядка під час очищення:

**def** **del\_contract\_by\_temp\_table\_personID**(self, personID):

session = self.get\_cursor()

person = self.\_get\_person\_by\_id(personID)

contracts = self.\_get\_contract\_by\_PersonID(personID)

**for** c **in** contracts:

**person.cars.remove(c)**

session.commit()

self.del\_person(personID)

Запис **person.cars.remove(c)** забезпечує видалення із таблиці Contract, використовуючи звязок cars = relationship("Contract", cascade="all, delete, delete-orphan", back\_populates="person").

### Пошук інформації

Повернення інформації з бази даних відбувається з використанням методу **query**. Приклад реалізації пошуку інформації у таблицях, об'єднаних з використанням JOIN:

**def** **select**(self, parking\_start, parking\_end, PersonPart, Date):

session = self.get\_cursor()

query = session.query(Contract, Cars, CarType, Makes, Persons)

query = query.join(Cars, Cars.CarRegNum == Contract.CarNumber)

query = query.join(CarType, Cars.CarType == CarType.CarTypeID)

query = query.join(Makes, Cars.CarMakeID == Makes.MakeID)

query = query.join(Persons, Persons.PersonID == Contract.PersonID)

query = query.filter(Contract.ParkingID.between(parking\_start, parking\_end),

Persons.PersonLastName.like(PersonPart),

Contract.ContractEnd > Date)

record = query.all()

#print(type(record))

**print**(record)

**return** self.sqlaListRecordsToDict(record)

Перш за все, необхідно вказати об'єкти, які повертаються запитом:

query = session.query(Contract, Cars, CarType, Makes, Persons)

Далі, вказати умови об'єднання з іншими таблицями:

query = query.join(Cars, Cars.CarRegNum == Contract.CarNumber)

Вказати умови, за якими відбираються обєкти:

query = query.filter(Contract.ParkingID.between(parking\_start, parking\_end),

Persons.PersonLastName.like(PersonPart),

Contract.ContractEnd > Date)

### Генерація записів бази даних

У моєму випадку генерація даних бази даних виконується збереженою процедурою, тому з боку SQLAlchemy необхідно виконати збережено процедуру:

**def** **random\_create**(self, param):

session = self.get\_cursor()

session.execute('public.generate\_data ?', [param])

session.commit()

### Проблеми, що виникли під час перетворення

SQLAlchemy працює з таблицями, як з класами, а із строками як з екземплярами класу, отже усі результати, що повертаються командою **query** містять об'єкти, або список об'єктів. У свою чергу Контролер очікує дані у форматі Dictionary, тому довелося виконати перетворення об'єктів у Словники:

**class** **CarType**(Base):

\_\_tablename\_\_ = 'CarType'

CarTypeID = Column('CarTypeID', SmallInteger, primary\_key=True)

CarTypeName = Column('CarTypeName', String(**255**), nullable=False)

cars = relationship("Cars", back\_populates="cartype")

# Ця частина коду використовується, коли ми повертаємо

# інформацію у контролер. Контролер очірує дані у форматі

# python Dictionary

**def** **to\_dict**(self):

dict = {}

dict["CarTypeID"] = self.CarTypeID

dict["CarTypeName"] = self.CarTypeName

**return** dict

Функції, що використовуються “локально” працюють з об'єктами, ті що повертають інформацію у контролер повертають її у вигляді словників:

**def** **\_get\_CarType\_by\_name**(self, CarTypeName):

session = self.get\_cursor()

record = session.query(CarType).filter\_by(

CarTypeName=CarTypeName).first()

**return** record

**def** **get\_CarType\_by\_name**(self, CarTypeName):

**return** self.\_get\_CarType\_by\_name(CarTypeName).to\_dict()

Окрім того, запити, що повертають результати з декількох таблиць, повертають перелік об'єктів, коли повертається лице одна строка та перелік переліків обєктів, коли багато:

**def** **sqlaRecord2dict**(self, record):

dict = {}

**for** r **in** record:

dict.update(r.to\_dict())

**return** dict

**def** **sqlaListRecordsToDict**(self, record):

dict = {}

**if** record **is** **not** None:

**for** row **in** record:

d = self.sqlaRecord2dict(row)

dict.append(d)

**return** dict

### Аналіз різних типів індексів у PostgreSQL

Погано спроектовані індекси та їх недостатня кількість – основне джерело вузьких місць у додатках баз даних. Проектування ефективних індексів має першорядну важливість для досягнення високої продуктивності баз даних та додатків.

В СУБД PostgreSQL існують наступні типи індексів:

* Btree — індекс бінарного дерева
* BRIN — індекс для впорядкованих даних. Розроблений для надвеликих баз даних.

Вихідні дані

Для аналізу створимо та використаємо таблиці, що містять щонайменше 200000 записів:



Під час виконання лабораторної роботи, я помітив, що хоч база даних і містить досить велику кількість записів, це майже не впливає на час, що витрачається на виконання запитів. Тому аналіз часу виконання кожного SQL запиту не надає інформацію, що можна аналізувати та порівнювати. Тому для аналізу роботи SQL запитів я буду порівнювати результати, що надає команда **EXPLAIN**

SQL запити

Тестування роботи різних типів індексів будемо проводити за наступним SQL запитом:

explain **select** \* **from** **public**."Contract"

**inner** join **public**."Cars" **on** "CarNumber" = "CarRegNum"

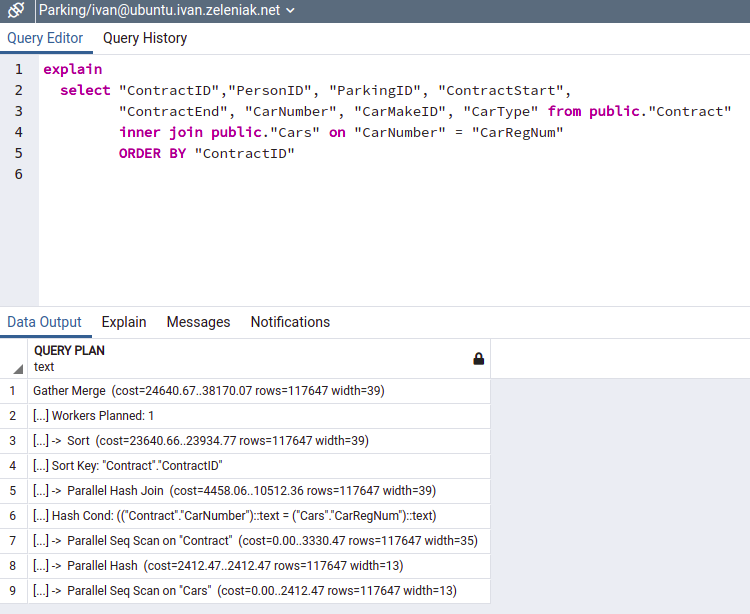
За для створення різноманітності, та перевірки роботи індексів з різними вхідними даними використаємо додаткові умови:

* Умова **дорівнює** для типів даних числові, строкові та дати
* Умова **between** для типів даних числові та дати. Цілком зрозуміло, що за умови працездатності індексів з директивою **between**, додаткова перевірка **більше** або **менше** буде зайвою.
* Перевіримо роботу індексів з агрегатними функціями.

Треба зауважити, що PostgreSQL автоматично створює індекси для первинних ключів тому пошук за первинними ключами буде індексованим.

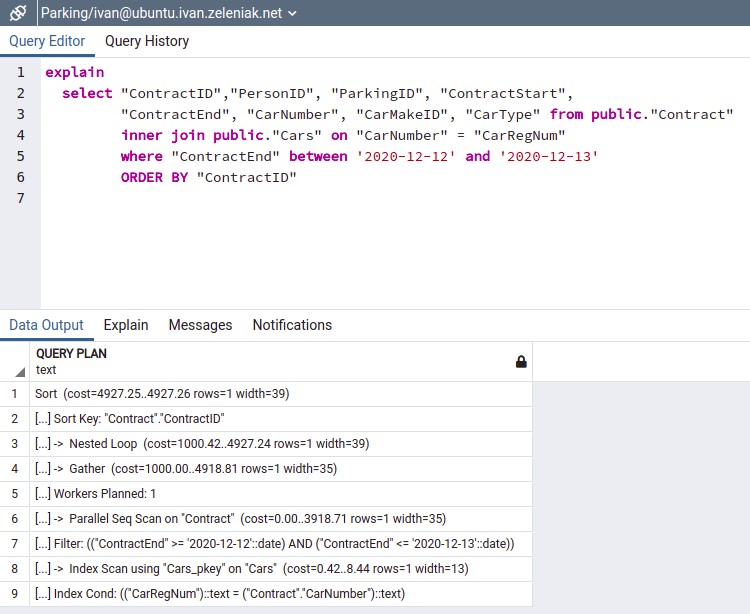
### Анализ SQL запитів

Виконаємо запит без додаткових умов **where**:



Як видно, вартість цього запиту сягає 38170.07 та під час його обробки індекси не використовуються.

Якщо додати умову, то вартість запиту різко знижується, навіть коли ця умова не індексується:

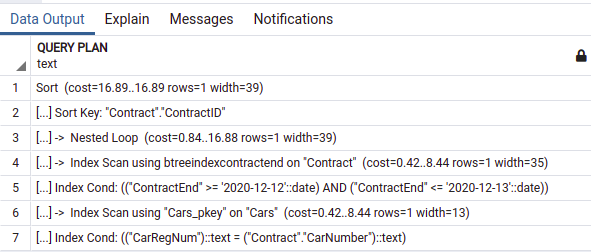


Як видно з результатів запиту, його вартість знизилась у 10 разів, до 4927,26, а також використовуються індекси для об'єднання INNER JOIN

Якщо створити індекс для поля "ContractEnd":

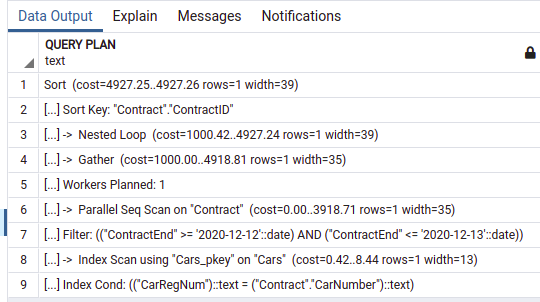
**create** **index** BtreeIndexContractEnd **on** **public**."Contract" using btree("ContractEnd")

Отримаємо зниження вартості запиту до 16.89, тобто майже у 300 разів



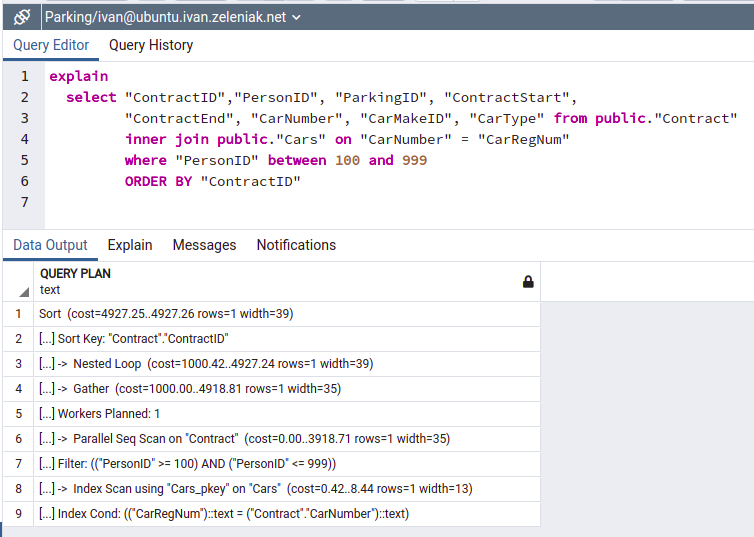
Напроти створення Brin індексу не впливає на вартість запиту, тому що СУБД не використовує його:

**create** **index** BrinIndexContractEnd **on** **public**."Contract" using brin("ContractEnd")



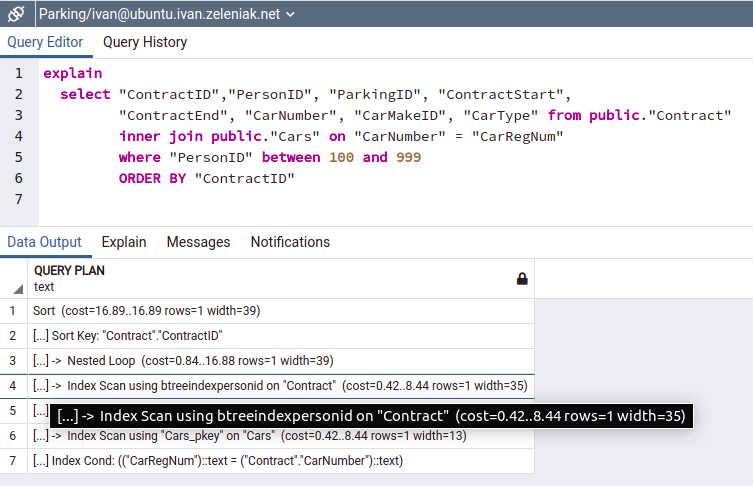
### Перевіримо роботу індексів для цілих чисел.

Вартість запиту для не індексованого цілого PersonID відповідає запиту з датами:

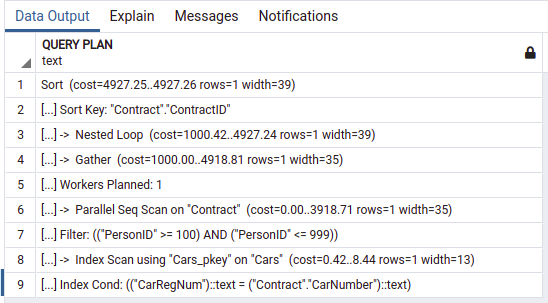


### Після створення індексів btree та Brin

**create** **index** BtreeIndexContractEnd **on** **public**."Contract" using btree("PersonID")



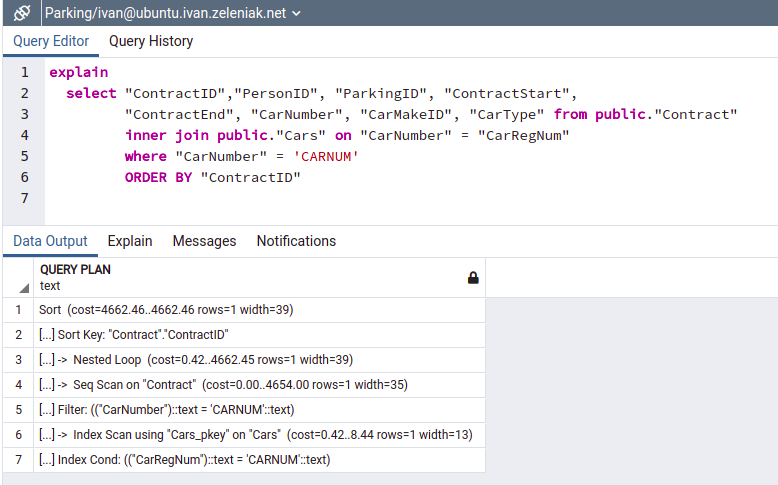
**create** **index** BrinIndexPersonID **on** **public**."Contract" using Brin("PersonID")



Як видно у моєму випадку СУБД не використовує індекси Brin, але досить ефективно використовує індекси Btree.

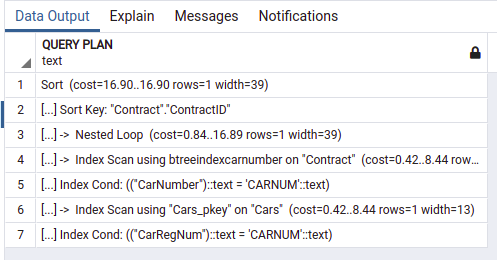
### Перевіримо роботу індексів з строковими типами даних.

За відсутності індексів:

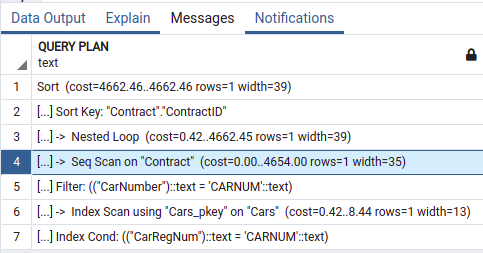


### З індексом Btree:

**create** **index** BtreeIndexCarNumber **on** **public**."Contract" using btree("CarNumber")

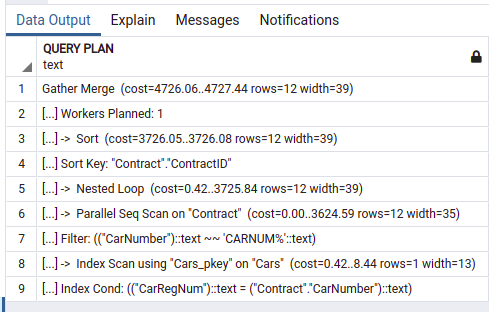


**create** **index** BrinIndexCarNumber **on** **public**."Contract" using Brin("CarNumber")



Для Btree пошук індексований, для випадку Brin індекс не використовується.

Але, якщо умову дорівнює замінити на умову LIKE індекси більше не використовуються:

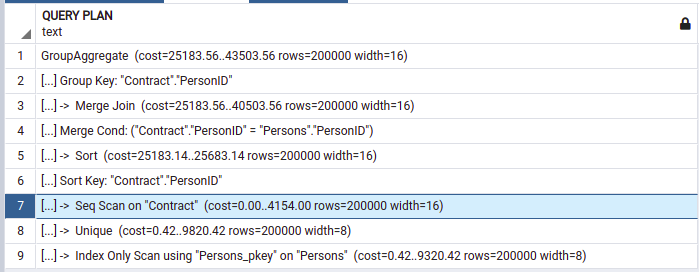


### Індекси та агрегатні функції:

Для Btree індексів використовуються індексований запит:



Для Brin індекси не використовуються



### Висновок

Для моєї бази даних тип індексу Btree є ефективним механізмом пришвидшення роботи бази даних та додатку у цілому. Напроти використання індексів BRIN є недоцільним та СУБД не використовує їх. Можливо це викликано характером даних які зберігаються, а саме випадково згенерованих, а не впорядкованих, для яких і утворювався індекс типу BRIN

### Створити тригер бази даних PostgreSQL за варіантом 6 *after update, insert*

Тригер PostgreSQL – це функція, яка автоматично викликається щоразу, коли відбувається подія, пов’язана з таблицею. Подією може бути будь-яка з таких: INSERT, UPDATE, DELETE або TRUNCATE.

Тригер — це спеціальна визначена користувачем функція, пов’язана з таблицею. Щоб створити новий тригер, спочатку потрібно визначити функцію тригера, а потім прив’язати цю функцію тригера до таблиці. Різниця між тригером і визначеною користувачем функцією полягає в тому, що тригер автоматично викликається, коли відбувається запуск події.

PostgreSQL надає два основних типи тригерів: тригери на рівні рядків і операторів. Різниця між цими двома видами полягає в тому, скільки разів викликається тригер і в який час.

Я розроблю тригер рядків, який буде викликатися для кожного рядка після виконання запитів UPDATE або INSERT.

За звичай тригери типу **after** використовуються після того, як записи були успішно змінені, та фактично використовуються для аудиту тих змін, що відбулися з даними. У межах своєї бази даних я не бачу сенсу у використанні такого типу тригерів, тому розроблений тригер буде виконувати зайві операції задля того, щоб виконати умови до лабораторної роботи.

### Суть тригера after insert

Після того як буде додано новий запис у таблицю **Contract** функція тригера буде знаходити усі активні контракти (з незакінченим терміном дії) для авто та встановлювати завершення дії контракту на дату початку доданого договору. Таким чином будуть автоматично завершуватись договори після створення нового договору.

**CREATE** **TRIGGER** after\_insert\_contract\_trigger

**AFTER** **INSERT**

**ON** **public**."Contract"

**FOR** EACH ROW

**EXECUTE** **FUNCTION** **public**.insert\_contract\_trigger\_function();

### Суть тригера after update

Після того як буде змінено дату початку дії договору та/або дати закінчення для існуючого запису у таблиці **Contract** тригер буде знаходити усі договори, що знаходяться у цьому ж періоді, для цього авто та паркомісця, та видаляти їх. Якщо дата початку контракту менша за дату початку поточного запису, а дата завершення підпадає під поточний, встановити дату завершення на дату поточного початку -1. Якщо існують контракти для цього авто з датою початку, що припадає на період поточного договору дата початку дії такого договору встановлюється, як кінцева дата поточного.

**CREATE** **TRIGGER** after\_update\_contract\_trigger

**AFTER** **UPDATE** **OF** "ContractStart", "ContractEnd"

**ON** **public**."Contract"

**FOR** EACH ROW

**EXECUTE** **FUNCTION** **public**.update\_contract\_trigger\_function();

### Реалізація функції тригера insert\_contract\_trigger\_function

**CREATE** OR REPLACE **FUNCTION** insert\_contract\_trigger\_function()

**RETURNS** **TRIGGER**

LANGUAGE PLPGSQL

**AS** $$

**DECLARE**

Contract RECORD;

**BEGIN**

-- Створимо цикли та перевірки для задоволення умов лабораторної

-- роботи, а саме: умовні оператори, цикли та обробку виключних ситуацій

**FOR** Contract IN

**SELECT** "ContractID", "PersonID", "ParkingID", "CarNumber",

"ContractStart", "ContractEnd"

**FROM** **public**."Contract"

**WHERE** "ContractEnd" > new."ContractStart" and

"CarNumber" = new."CarNumber" and

"ContractID" <> new."ContractID"

LOOP

-- Якщо дата початку договору більша, або дорівнює новому,

-- Запис необхідно видалити бо утвориться "виключна" ситуація -

-- два контракти в одному періоді

**IF** Contract."ContractStart" >= new."ContractStart"

**THEN**

**DELETE** **FROM** **public**."Contract"

**WHERE** "ContractID" = Contract."ContractID";ґ

-- Інакше необхідно змінити дату завершення договору на поточну.

**ELSE**

**UPDATE** **public**."Contract"

**SET** "ContractEnd"= new."ContractStart"

**WHERE** "ContractID"=Contract."ContractID";

**END** IF;

**END** LOOP;

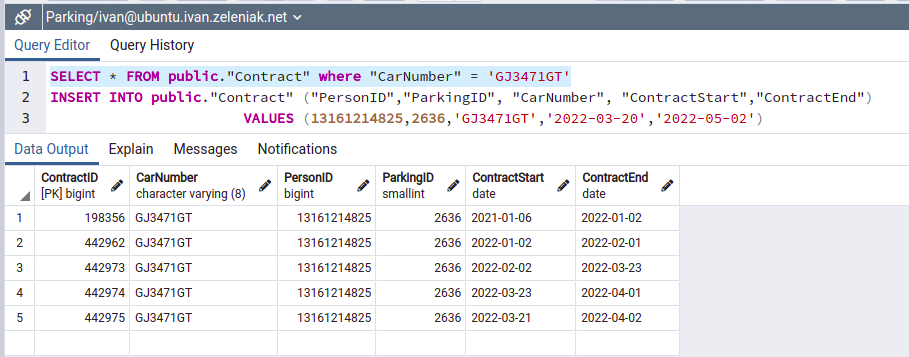
**RETURN** new;

**END**;

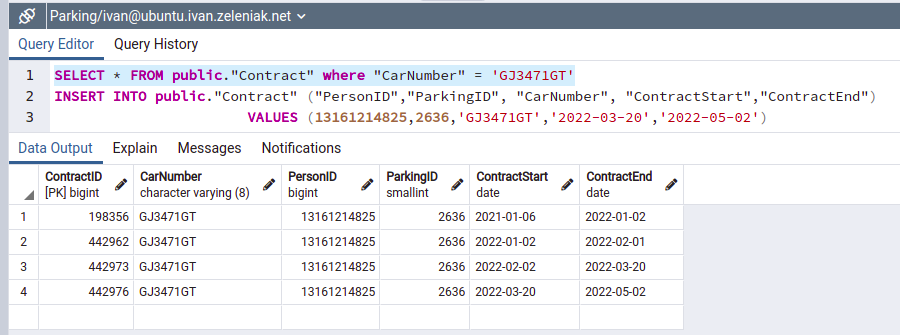
$$

### Результати роботи

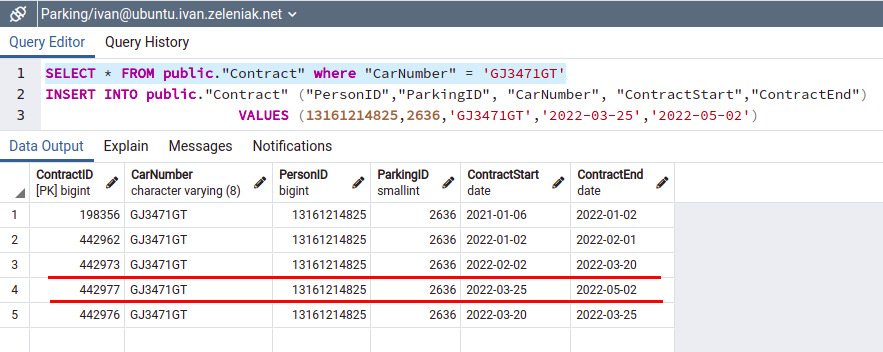
Початковий стан



У двох останніх записах перетинаються дати початку та дати завершення з майбутнім записом. Після виконання додавання, два записи вилучено та створено один запис:



Якщо дати перетинаються, то дата завершення існуючого договору змінюється



### Реалізація функції тригера **update\_contract\_trigger\_function**

**CREATE** OR REPLACE **FUNCTION** update\_contract\_trigger\_function()

**RETURNS** **TRIGGER**

LANGUAGE PLPGSQL

**AS** $$

**DECLARE**

Contract RECORD;

**BEGIN**

-- Створимо цикли та перевірки для задоволення умов лабораторної

-- роботи, а саме: умовні оператори, цикли та обробку виключних ситуацій

-- Видалемо усі записи, що перетинаються повністю з поточним договором

**DELETE** **FROM** **public**."Contract"

**WHERE** "ContractID" <> NEW."ContractID" AND

"CarNumber" = new."CarNumber" AND

"ContractStart" >= NEW."ContractStart" AND

"ContractEnd" <=NEW."ContractEnd";

**FOR** Contract IN

**SELECT** "ContractID", "PersonID", "ParkingID",

"CarNumber", "ContractStart", "ContractEnd"

**FROM** **public**."Contract"

**WHERE** "ContractEnd" > new."ContractEnd" and

"ContractStart" < new."ContractEnd" and

"CarNumber" = new."CarNumber" and

"ContractID" <> new."ContractID"

**LOOP**

-- Якщо дата початку договору менша, нового,

-- Встановимо дату початку на кінець поточного

**IF** Contract."ContractStart" < new."ContractEnd"

**THEN**

**UPDATE** **public**."Contract"

**SET** "ContractStart"= new."ContractEnd"

**WHERE** "ContractID" = Contract."ContractID";

**END IF**;

**END LOOP**;

**FOR** Contract IN

**SELECT** "ContractID", "PersonID", "ParkingID",

"CarNumber", "ContractStart", "ContractEnd"

**FROM** **public**."Contract"

**WHERE** "ContractEnd" > new."ContractStart" and

"ContractStart" < new."ContractStart" and

"CarNumber" = new."CarNumber" and

"ContractID" <> new."ContractID"

LOOP

-- Якщо дата початку договору менша, нового, та дата завершення більша

-- Встановимо дату завершення на початок поточного

IF Contract."ContractStart" < new."ContractEnd"

**THEN**

**UPDATE** **public**."Contract"

**SET** "ContractEnd"= new."ContractStart"

**WHERE** "ContractID" = Contract."ContractID";

**END** IF;

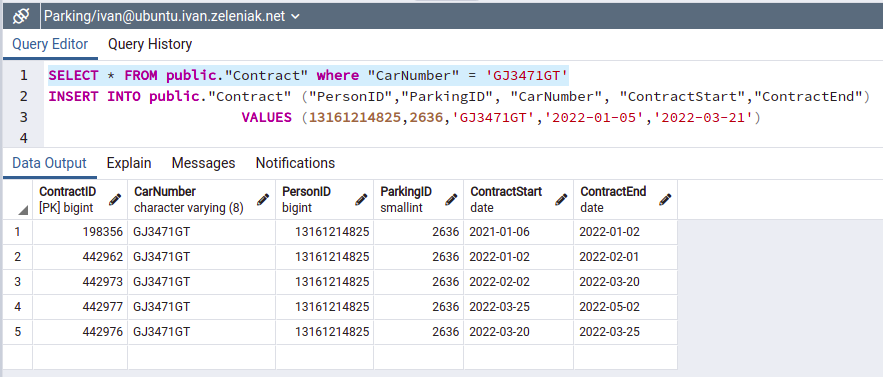
**END** LOOP;

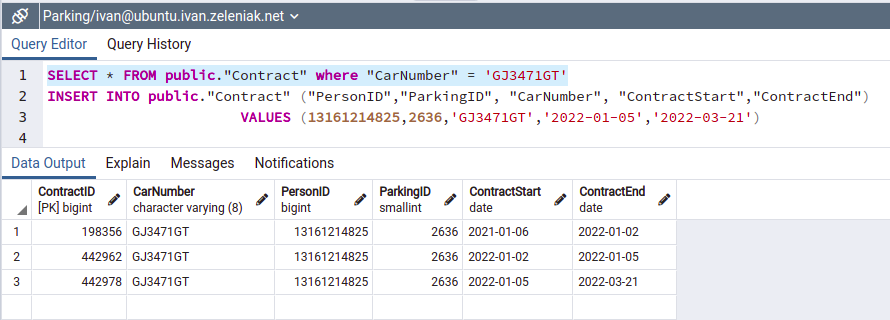
**RETURN** new;

**END**;

### Результат роботи тригера

Початкові дані

Після виконання запиту оновлення:



### Приклади та аналіз рівнів ізоляції транзакцій у PostgreSQL

Відповідно до стандарту SQL-92 існують наступні рівні ізоляції транзакції:

**Serializable (впорядкованість)**

Найбільш високий рівень ізольованості; Усі оператори поточної транзакції бачать лише рядки, які були зафіксовані перед першим запитом на вибірку або зміну даних, виконаним у цій транзакції. Якщо накладання операцій читання та запису паралельних транзакцій, що серіалізуються, може призвести до ситуації, неможливої під час послідовного виконання (коли одна транзакція виконується за іншою), відбудеться відкат однієї з транзакцій з помилкою serialization\_failure (збій серіалізації)..

**Repeatable read (повторюваність читання)**

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).

**Read committed (читання фіксованих даних)**

Цей рівень є рівнем за замовченням для PostreSQL. 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.

### При паралельному виконанні транзакцій можливі такі проблеми:

**Втрачене оновлення** — при одночасній зміні одного блоку даних різними транзакціями, одна із змін втрачається;

**«Брудне» читання** — читання даних, які додані чи змінені транзакцією, яка потім не підтвердиться (відкотиться);

**Неповторюване читання** — при повторному читанні в рамках однієї транзакції, раніше прочитані дані з’являються зміненими;

**Фантомне читання** — одна транзакція в ході свого виконання декілька разів вибирає множину рядків за одними і тими ж критеріями. Інша транзакція в інтервалах між цими вибірками додає чи видаляє рядки чи змінює стовпці деяких рядків, що використовується в критеріях вибірки першої транзакції, і успішно закінчується. В результаті отримаємо, що одні і ті ж вибірки в першій транзакції дають різні множини рядків.

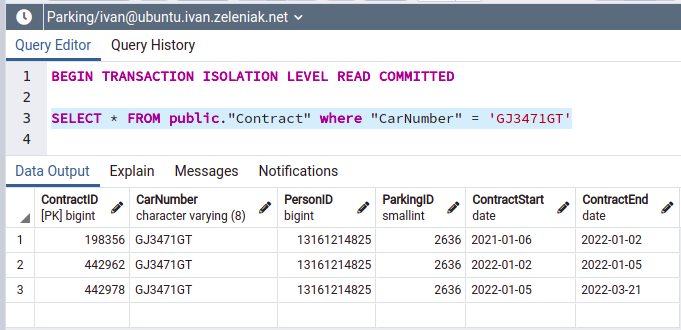
### Розглянемо ситуації, в яких можливо виникнення даних проблем

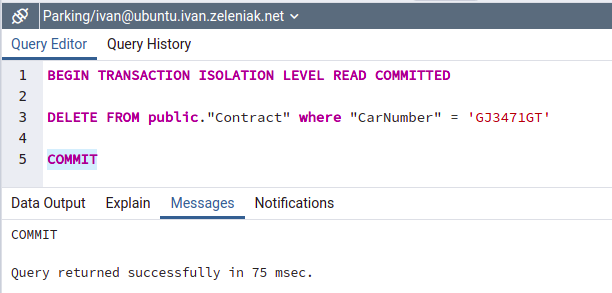
**«Брудне» читання**

Читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).

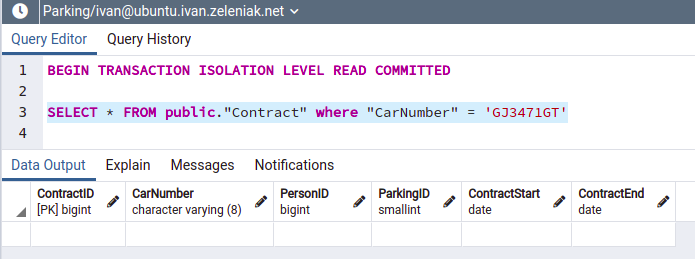
Припустимо, є дві транзакції READ COMMITED, відкриті в різних з’єднаннях з базою, в яких виконані такі SQL-оператори:

Для першої транзакції виконаємо

Для паралельної транзакції виконаємо



У той самий час для першої транзакції отримаємо “брудне” читання



За цим самим принципом можна відтворити усі інші феномени.